FH·W-S

Hochschule
für angewandte Wissenschaften
Würzburg-Schweinfurt

Fakultät Elektrotechnik

Bachelorarbeit bei ZF

Entwicklung und Optimierung einer Software-in-the-Loop Teststrategie für elektrische Antriebsfunktionen

15. Januar 2022 bis 30. April 2022

Verfasser:	Xaver Gruber
Matrikelnummer:	3618080
ZF-Betreuer:	Michael Hantschel
	Christian Kuklinski
FHWS-Betreuer:	Prof. Dr. Markus Mathes
	Prof. Dr. Jan Hansmann

Inhaltsverzeichnis

Sperrvermerk

Die vorliegende Bachelorarbeit mit dem Titel „Entwicklung und Optimierung einer Software-in-the-Loop-Teststrategie für elektrische Antriebsfunktionen“ enthält schützenswerte Informationen der ZF Friedrichshafen AG. Die Veröffentlichung, Vervielfältigung -auch in schriftlicher Form-, Weitergabe -auch auszugsweise- sowie das Einstellen dieser Abschlussarbeit in eine Hochschulbibliothek oder öffentlich zugängliche Bibliothek sowie jede weitere öffentliche Zugänglichmachung sind vor Ablauf der angegebenen Sperrfrist ohne vorherige schriftliche Zustimmung der ZF Friedrichshafen AG nicht gestattet. Ausgenommen von diesem Zustimmungserfordernis ist die Verwendung, die aufgrund der jeweils anwendbaren Prüfungs- oder Studienordnung zwingend erforderlich ist.
Die Sperrfrist gilt mit Abgabe der Abschlussarbeit an der Hochschule 3 Jahre.

Schweinfurt, 30.04.22

Abkürzungsverzeichnis

TPT Time Partition Testing

SiL Software-in-the-Loop

MiL Model-in-the-Loop

PiL Processor-in-the-Loop

HiL Hardware-in-the-Loop

ZF ZF Friedrichshafen AG

SUT System under Test

API Application Program Interface

XML Extensible Markup Language

exe executable

API Application Programming Interface

Abstract

This bachelor thesis is about two topics as Software-in-the-Loop test strategy. One topic is Interface Testing; the other one is Equivalence Classes. Both topics are worked out within TPT, which is a software for testing embedded systems by the company PikeTec. Steps for Interface Testing include analyzing the existing Interface Test, modelling a C source and automatically generating test cases using TPT. A concept for Equivalence Classes is pointed out, which is about validating existing Software-in-the-Loop tests and self-tests checking if the defined Equivalence Classes are reached. It is explained that defining the latter is a way of creating a database that can be used for further tests.

Kurzfassung

Es werden zwei Themen als Software-in-the-Loop Teststrategie bearbeitet, nämlich ein Schnittstellentest und ein Äquivalenzklassentest. Beide Themen werden mit TPT erarbeitet. TPT ist ein Programm der Firma PikeTec zum Testen von eingebetteten Systemen. Die Arbeitspakete für den Schnittstellentest sind einen bestehenden Schnittstellentest zu analysieren, eine C Datei nachzustellen und Testfälle automatisiert in TPT generieren lassen. Für den Äquivalenzklassentest wird ein Konzept erarbeitet, wie diese eingesetzt werden können. Es geht zum Einen darum, Äquivalenzklassen zu nutzen, um bestehende Software-in-the-Loop Tests zu überprüfen. Zum Anderen geht es um eine Selbstüberprüfung der Äquivalenzklassen, ob die definierten Äquivalenzklassen alle erreichbar sind. Es wird auch aufgezeigt, dass durch das Definieren von Äquivalenzklassen eine gewisse Datenbank für Tests entsteht und diese auch für weitere Tests eingesetzt werden kann.

1. Einleitung

1.1. Vorstellung der ZF

ZF Friedrichshafen AG (ZF) ist ein Technologiekonzern mit rund 153.500 Mitarbeitern (Stand 2020) weltweit in den Bereichen Mobilität und Industrietechnik. ZF übernahm zuletzt WABCO und konnte dadurch ihre Kompetenz im Bereich von Technologien für schwere Nutzfahrzeuge, Busse und Trailer steigern. ZF gibt jährlich sieben Prozent des Umsatzes (Stand 2020) in Forschung und Entwicklung aus. ZF will die Veränderungen in der Mobilitätsbranche mit der Strategie „Next Generation Mobility“ vorantreiben. Das Ziel der Strategie ist eine einfache, saubere Mobilität, die automatisiert, komfortabel und bezahlbar ist. Sie soll für jedermann und überall erreichbar sein. Für die Bachelorarbeit bin ich in der Abteilung Softwareentwicklung elektrische Antriebsfunktionen tätig [vgl. 1].

1.2. Beschreibung des Themas

Antriebsfunktionen elektrischer Maschinen sind als Software geschrieben, laufen auf einer Hardware und übernehmen die Steuerung der Maschine. Das macht ein eingebettetes System aus. Ein eingebettetes System ist in der Regel ein in-the-Loop System. In einem in-the-Loop System bilden Ausgänge des Systems eine Rückkopplung zu den Eingängen. Dies muss auch beim Testen berücksichtigt werden. Software-in-the-Loop Test bedeutet, dass die Software als in-the-Loop System getestet wird, wobei die Hardware simuliert wird. Bei Hardware-in-the-Loop hingegen ist die Hardware vorhanden und die Software wird, während es auf der Hardware ausgeführt wird, getestet. Mit SIL wird eine Möglichkeiten geschaffen Software ohne Hardware zu testen. Damit können Kosten von Hardware eingespart werden. Des Weiteren kann dadurch auch in den frühen Phasen der Entwicklung getestet werden [vgl. 2, S. 1 f.][vgl. 3]. Es ist wichtig Fehler schnell zu beheben, denn je länger ein Fehler unentdeckt bleibt, umso teurer wird dieser Fehler [vgl. 4].

Diese Arbeit handelt davon TPT als Programm (siehe Abschnitt 1.4) einzusetzen, um einen Schnittstellentest zu überführen (siehe Kapitel 5) und ein neues Testkonzept mit Äquivalenzklassen (siehe Kapitel 6) zu erarbeiten.

1.3. Einordnung der Arbeit

In der Abteilung Softwareentwicklung elektrische Antriebsfunktion bei ZF, in der ich für die Bachelorarbeit tätig bin, wird nach dem V-Modell entwickelt (siehe Abbildung 1.1).

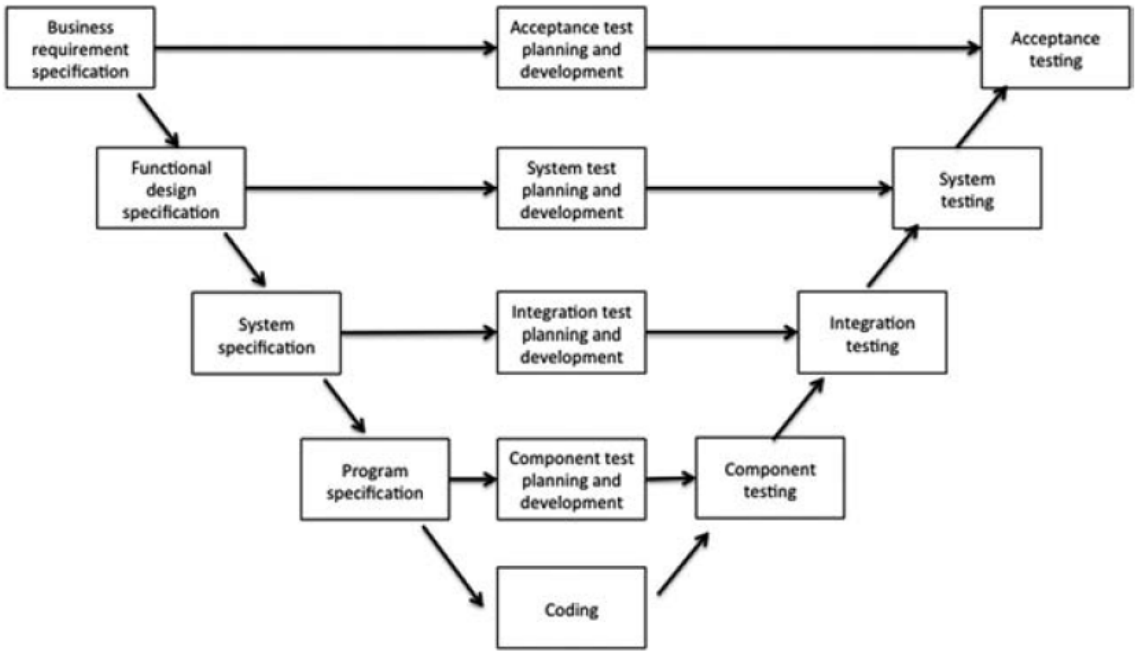

Abbildung 1.1.: V-Modell [5, S. 68]

Ein Merkmal des V-Modells ist, dass jede Stufe der Entwicklung einen dazugehörigen Test besitzt. Die folgenden Testebenen beschreiben die einzelnen Teststufen [vgl. 6, S. 41 f., S. 51]:

Komponententest Es wird eine Komponente isoliert von der Software - funktional und strukturell - getestet.

Integrationstest Es ist ein Test, der das Zusammenspiel bereits getesteter Komponenten überprüft.

Systemtest Es wird die Funktionsfähigkeit des ganzen Systems getestet. Dazu zählen auch Performanz-, Stress und Robustheitstests.

Abnahmetest Mit diesem Test wird überprüft, ob die Erwartungen der Anwender erfüllt werden. Der Test soll unter realen Einsatzbedingungen stattfinden.

Der Schnittstellentest, in dem die Signalverbindungen zwischen Komponenten überprüft werden, ist ein Teil des Integrationstests [vgl. 6, S. 51, S. 217]. Der Äquivalenzklassentest

ist als funktionaler Test im Komponententest wiederzufinden [vgl. 6, S. 41 f.][vgl. 7, S. 114 ff.].

Wie der Äquivalenzklassentest einzuordnen ist, wird in Abbildung 1.2 deutlich. Dieses Diagramm ist nicht vollständig, sondern soll nur als Einordnung dienen. Eine Art der Analyse in der Software ist eine dynamische Analyse, eine weitere ist die statische Analyse. In der dynamischen Analyse wird der Code ausgeführt, in der statischen hingegen nicht. Eine Art der dynamischen Analyse ist funktional, eine weitere Art ist strukturell. In der funktionalen Analyse wird das System als Kasten betrachtet, sprich Informationen über den Programmcode sowie der inneren Struktur sind nicht von Bedeutung. Deshalb wird es auch als Blackbox-Testverfahren bezeichnet. In der strukturellen Analyse hingegen wird der innere Ablauf im Testobjekt analysiert. Der Blickwinkel wandert in das System. Es wird auch Whitebox-Testverfahren genannt. Ein funktionaler Test ist der Äquivalenzklassentest. Ein weiterer ist die Grenzwertanalyse [vgl. 7, S. 114 ff.][vgl. 8][vgl. 9]. Die Grenzwertanalyse eignet sich hervorragend, um sie mit anderen Tests wie dem Äquivalenzklassentest zu kombinieren [vgl. 6, S. 36].

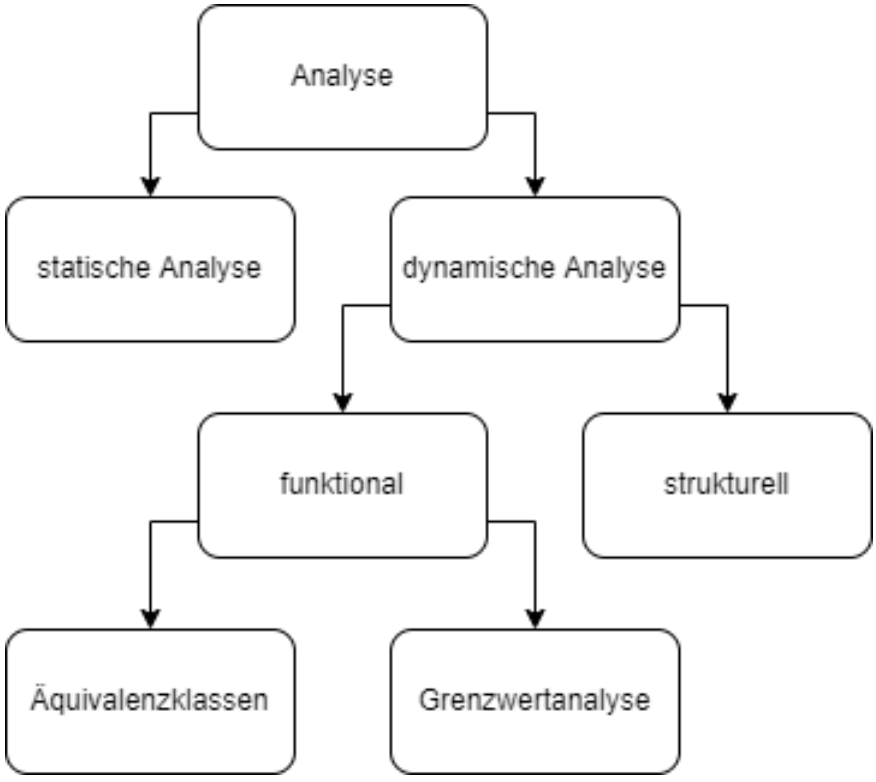

Abbildung 1.2.: Einordnung der Äquivalenzklassen [vgl. 8][vgl. 9][vgl. 7, S. 114 ff.]

1.4. TPT

Eingebettete Systeme zeichnen sich aus, dass sie über kontinuierliche Größen, etwa Sensoren, mit der Umgebung verbunden sind. Time Partition Testing (TPT) wurde im Rahmen einer Dissertation von Eckard Lehmann bei der Daimler AG entwickelt, um das kontinuierliche Verhalten eingebetteter Systeme testen zu können. Die ersten Versionen gibt es schon im Jahre 2000. Es wird jahrelang bei der Daimler AG in der Fahrzeugentwicklung benutzt und weiterentwickelt. Heutzutage besitzt TPT die Firma PikeTec. Es werden eine Anbindung zu verschiedenen Programmen wie Autosar, Matlab, Labcar und Targetlink und verschiedene Testarten wie Software-in-the-Loop (SiL), Hardware-in-the-Loop (HiL) Model-in-the-Loop (MiL), Processor-in-the-Loop (PiL) unterstützt. Das Programm ist von der Anforderungsanalyse über die Testfallerstellung bis zur Auswertung der Ergebnisse einsetzbar [vgl. 10][vgl. 11]. TPT ist [vgl. 12, S. 120]:

- plattformunabhängig: Plattformen können ausgetauscht werden, ohne dass Testfälle neu geschrieben werden müssen
- echtzeitfähig: Echtzeitanforderungen eines eingebetteten Systems werden erfüllt
- reaktiv: beim Eintritt eines Systemzustands kann der Testfall darauf reagieren

Nun soll TPT bei ZF in der Abteilung Softwareentwicklung elektrische Antriebsfunktionen eingeführt werden. Es soll als einheitliche Testumgebung für SiL dienen und andere Tools wie Softcar, das ein ZF internes Tool ist, ablösen.

1.5. Ziele

Ein Ziel der Arbeit ist einen bestehenden Schnittstellentest in TPT zu überführen und zu optimieren. TPT soll als einheitliche Testumgebung eingeführt werden. Dabei werden bestehende Tests, so auch der Schnittstellentest, in TPT überführt. Der Schnittstellentest hat Potenziale in der Laufzeit (siehe Kapitel 3), was opimiert werden soll. Ein zweites Ziel ist ein Konzept für den Äquivalenzklassentest in TPT zu erarbeiten (siehe Kapitel 6).

2. Grundlagen

In diesem Kapitel werden die Grundlagen erarbeitet, worauf die Umsetzung basieren wird. Es wird auf Schnittstellen eingegangen und wie diese gestestet werden können. Ein weiterer Punkt sind die Äquivalenzklassen. Dabei werden die Äquivalenzklassen mathematisch beschrieben und erklärt, wie sie in der Informatik angewandt werden. Die Umsetzung erfolgt in TPT, weshalb im Abschnitt 2.3 erklärt wird, welche Funktionalitäten TPT hat und wie es konfiguriert wird.

2.1. Schnittstellen

Ein Teil des Integrationstests ist der Schnittstellentest. Schnittstellen sorgen dafür, dass Softwareteile, insbesondere Softwarekomponenten miteinander verbunden sind und Daten austauschen können. Es werden verschiedene Arten von Schnittstellendefinitionen vorgestellt und es wird gezeigt, wie eine Schnittstelle getestet werden kann. In Abbildung 2.1 ist zu sehen, wie zwei Komponenten durch eine Schnittstellendefinition verbunden sind.

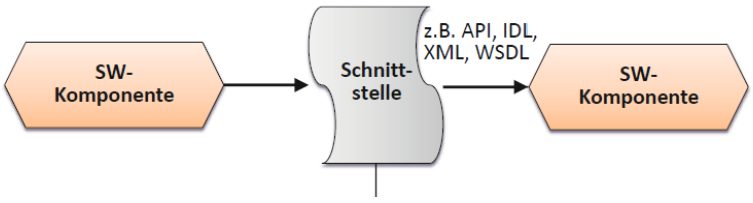

Abbildung 2.1.: Schnittstellen [6, S.218]

Es gibt verschiedene Arten, wie man eine Schnittstelle definieren kann [vgl. 6, S. 218 ff.]:

unstrukturierte Datenübergabe Es gibt keine feste Schnittstellendefinition. Der Empfänger muss die Daten selbst interpretieren.

Gemeinsame globale Datenbereiche Komponenten können darauf zugreifen, Werte ablegen und entnehmen. Wenn beide Komponenten jedoch eine eigene Definition des Datentyps haben, wird die Wartbarkeit bei Änderungen schwierig. Abhilfe kann hier eine Header

Datei sein, in der die Datenstruktur definiert ist. Eine Header Datei dient als Schnittstelle zwischen Dateien. Damit können Daten für mehrere Dateien sichtbar gemacht werden.

Application Programming Interface (API) Daten werden über eine API übertragen. Dabei werden die Daten in einen Stapel gespeichert und an die Zielkomponente übergeben, welche die Daten vom Stapel entpackt. Es kann ein Rückgabewert (Return) an die Senderkomponente zurückgesendet werden.

Datenbanken Der Datenaustausch ist über eine gemeinsame Datenbank definiert. Die erste Komponente legt Daten ab, die zweite Komponente holt die Daten. Datenbanken werden oft in IT-Systemen eingesetzt. Der Test einer Datenbank kann erfolgen, indem Dateninhalte überprüft werden. Es bietet sich auch an, die Datenbank zu manipulieren, indem einzelne Zeilen oder Spalten verändert werden, um die Auswirkungen zu überprüfen.

Schnittstellendefinitionssprachen Sie haben die Aufgabe eine sichere Datenübertragung zu ermöglichen, indem die Struktur der zu sendenden Daten klar definiert wird. Eine Schnittstellendefinitionssprache ist Extensible Markup Language (XML). In einem Schema wird die Struktur und der Inhalt der Schnittstelle festgelegt. Anhand dieses Schemas wird überprüft, ob die Daten richtig sind. Ein Vorteil von XML ist, dass es einfach erweiterbar ist. Es ist auch sehr flexibel, sodass eine Schema nach den eigenen Bedingungen erstellt werden kann. Es ist sinnvoll, strenge Regeln aufzustellen, damit eine Überprüfung besser möglich ist.

Schnittstellentest

Für den Schnittstellentest benötigt es zwei Schritte, wie in Abbildung 2.2 zu sehen ist. Ein Schritt ist, dass getestet wird, ob das Format der Schnittstellendefinition eingehalten wird. Ein Entwickler schreibt die Schnittstelle und die Überprüfung erfolgt von einer zweiten Person. Der Überprüfer hat einen anderen Blickwinkel und kann Denkfehler besser erkennen und sofort lösen [vgl. 6, S. 226]. Als Beispiel einer Schnittstelle wird im Folgenden XML genommen. Es gibt automatisierte Tests wie einen XML-Analysator. Dieser überprüft XML-Daten gegen die Struktur im vorgegebenen Schema [vgl. 6, S. 237].

Damit werden nur die XML-Daten an sich überprüft. Ein zweiter wichtiger Punkt ist die Überprüfung des Zugriffs auf die Daten, die in einer XML hinterlegt sind. Dafür muss der Code geprüft werden, in der die Schnittstelle verwendet wird. Die Schnittstelle wird auf Empfänger- und Senderseite überprüft. Dabei können Parameter wie der Typ, die Reihenfolge oder die Nutzungsart eine Rolle spielen [vgl. 6, S. 228].

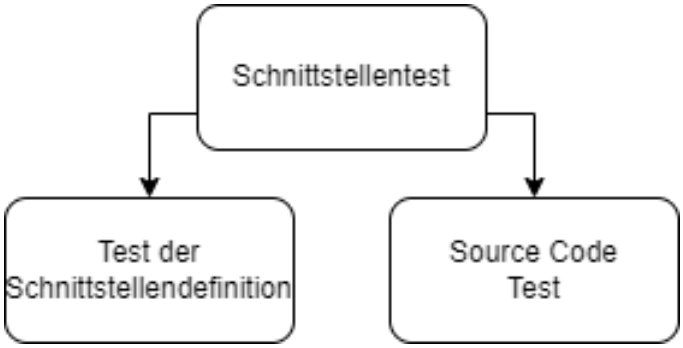

Abbildung 2.2.: Schnittstellentest [vgl. 6, S. 226]

2.2. Äquivalenzklassentest

In diesem Punkt geht es um die Äquivalenzklassen. Sie werden mathematisch beschrieben und danach wird gezeigt, wie sie in der Informatik Anwendung finden.

Mathematische Beschreibung der Äquivalenzklassen

Seien zwei Elemente zu einer Teilmenge R zugeordnet, so spricht man von einer Relation der beiden Elemente und schreibt $\sim$. Eine Äquivalenzrelation liegt vor, wenn eine Relation reflexiv, symmetrisch und transitiv ist, wie in Abbildung 2.3 dargestellt. Eine reflexive Relation ist gegeben, wenn jedes Element einer Menge in Relation zu sich selbst steht. Eine symmetrische Relation ist gegeben, wenn zwei Elemente einer Menge gegeneinander austauschbar sind, wobei die Relation zueinander gegeben bleibt. Eine transitive Relation ist gegeben, wenn ein Element in Relation zu zwei anderen Elementen steht, so stehen jene beide Elemente auch in Relation zueinander [vgl. 13, S. 66].

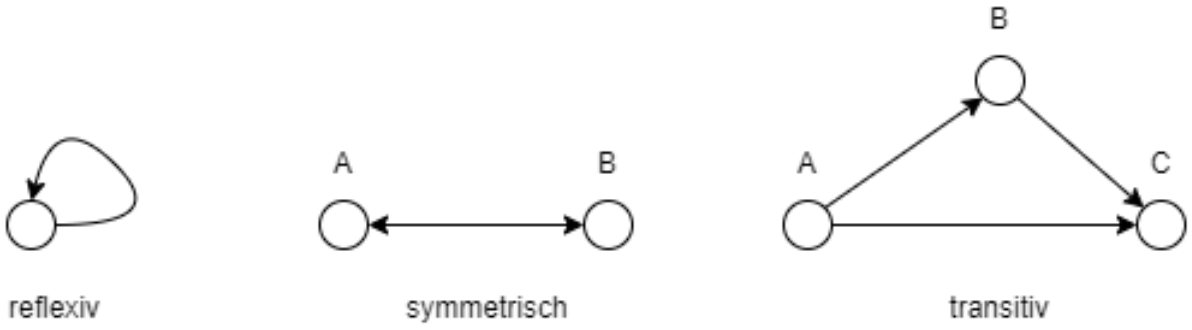

Abbildung 2.3.: Eigenschaften einer Äquivalenzklasse [14][15]

Definition der Äquivalenzklasse

$$[a] = \{b \in M; b \sim a\}$$

bezeichnet man als Äquivalenzklasse des Elements a in M,
wobei $\sim$ eine Äquivalenzrelation auf einer Menge M ist und

$$a, b \in M.$$

Alle Elemente einer Äquivalenzrelation in der Gesamtheit bezeichnet man als Äquivalenzklasse. Jedes einzelne Element davon bezeichnet man als Repräsentanten der Äquivalenzklasse [vgl. 13, S. 66].

Äquivalenzklassen in der Informatik

Um den Bezug zur Informatik herzustellen, wird als Menge der Wertebereich eines Parameters bezeichnet. Das Prinzip der Äquivalenzklassen kann so auf Parameter angewendet werden, indem ein Parameter in Wertebereiche eingeteilt wird, wie es in Abbildung 2.4 zu sehen ist. Ein Repräsentant einer Äquivalenzklasse ist ein beliebiger Wert innerhalb des Wertebereichs. Es ist gewollt, dass auch Äquivalenzklassen definiert werden, die ein kritisches oder fehlerhaftes Verhalten des Systems erzeugen können. Wie sie im Detail definiert werden, soll aus dem Requirement hervorgehen. Sie werden so gewählt, dass sich das System für alle Werte innerhalb der Äquivalenzklasse gleich/ähnlich verhält. Somit reicht es aus, nur einen Repräsentanten zu testen. Der Sinn der Äquivalenzklassen ist es, dass redundante Testfälle vermieden werden, indem jeweils nur Repräsentanten getestet werden. Die Anzahl der Testfälle entsteht durch das Produkt aller Äquivalenzklassen pro Parameter. Bei mehreren Parametern sowie der dazugehörigen definierten Wertebereiche können schnell viele Testfälle entstehen [vgl. 7, S. 114 ff.].

Möglichkeiten der Testfalleinschränkung [vgl. 7, S. 120]:

- Testfälle filtern, dass nur oft vorkommende Kombinationen getestet werden
- Testfälle mit Grenzwerten bevorzugen
- paarweise Kombination: jeweils zwei Repräsentanten unterschiedler Äquivalenzklassen werden kombiniert
- Minimalkriterium: jeder Repräsentant ist in mindestens einem Testfall vorhanden

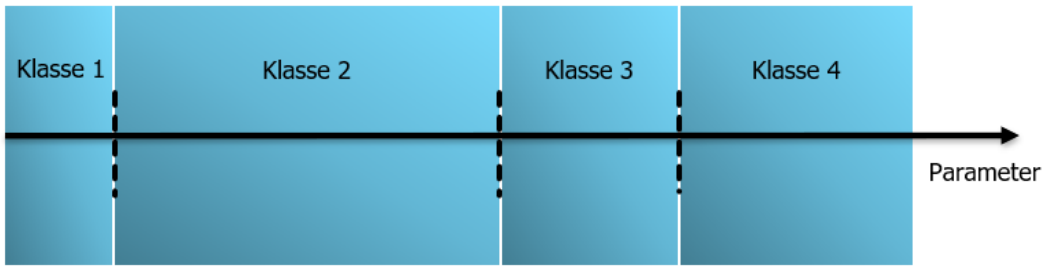

Abbildung 2.4.: Einteilung eines Parameters in Äquivalenzklassen [vgl. 7, S. 114 ff.]

- Repräsentanten, die ein kritisches/fehlerhaftes Verhalten hervorrufen, werden nicht miteinander kombiniert

2.3. Konfiguration von TPT

In Abbildung 2.5 ist ein Überblick zu sehen, welche Schritte nötig sind, um TPT für einen Test einzurichten [vgl. 16, S. 39].

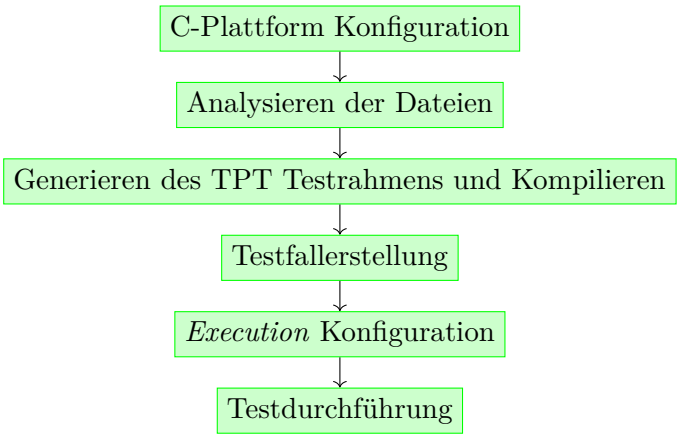

Abbildung 2.5.: Flussdiagramm TPT Einrichtung [vgl. 16, S. 39]

Platform Configuration und *Platform Execution*

Die Plattformkonfigurationen in TPT dienen dazu, das zu testende System in TPT für die Analyse zu konfigurieren. Eine Plattform ist die C-Platfform, die in dieser Arbeit ausschließlich behandelt wird. In der Plattform *Execution* wird festgelegt, welche Tests durchgeführt werden sollen. Die Plattform Konfiguration und die Plattform *Execution* sind streng getrennt und unabhängig voneinander. Der große Vorteil dieser Trennung ist, dass die Plattformkonfiguration ausgetauscht werden kann und die Testfälle nicht neu geschrieben werden müssen. Alle Einstellungen der Plattformkonfiguration sind unabhängig von der Testfallerstellung[vgl. 12, S. 120].

C-Plattform Konfiguration

Es gibt folgende Konfigurationsmöglichkeiten [vgl. 16, S. 862 ff.]:

Compiler auswählen Pfad zu einem installierten Compiler

Sourcen auswählen Diese werden später gelinkt und kompiliert, zu analysierende Sourcen müssen auf analysieren gestellt werden

Scheduler Ein (Prozess-)Scheduler berechnet und entscheidet, wann und wie lange ein Prozess die CPU Zeit bekommt [vgl. 17, S. 44]. Hier ist ein Prozess eine Funktion, die ausgeführt wird, sobald sie die CPU Zeit bekommt. Im TPT Scheduler werden vom Benutzer Funktionen ausgewählt, die während der Testdurchführung ausgeführt werden sollen. Es können die ausgewählten Funktionen auf *Startup, Initial* und *Periodic* gestellt werden. Mit *Startup* wird die Funktion nur einmal am Anfang der Testdurchführung ausgeführt. Mit *Initial* wird die Funktion einmal am Anfang eines Zyklus ausgeführt und mit *Periodic* wird die Funktion einmal pro definierter Abtastrate ausgeführt.

Step size die Abtastrate. Gibt beispielsweise die Zeiteinheit der periodischen Scheduler Funktionen an.

Test Execution und *Test Assessment*

Die *Test Execution* und das *Test Assessment* sind zwei getrennte Prozesse. Die *Test Execution* ist dafür zuständig die Testfälle auszuführen und dabei Daten zu sammeln. Diese gesammelten Daten werden beim *Test Assessment* ausgewertet. Daten sind beispielsweise die Werte der Signale, wobei wichtig sein kann, wann und wie lange ein bestimmter Wert des Signals

überschritten wurde [vgl. 16, S. 1212 ff.]. Ein Assessment ist beispielsweise die Auswertung eines Compare Steps. Es gibt auch ein Assessment für Äquivalenzklassen. Dabei kann festgelegt werden, welche Äquivalenzklassen erreicht oder auch nicht erreicht werden sollen [vgl. 18, S. 41 ff.].

Analysieren der Dateien

Durch das Analysieren werden Signale sowie Funktionen der Dateien TPT bekannt gegeben. Es wird auch TPT Binding genannt [vgl. 16, S. 870 ff.]. Das Analysieren erfolgt mit einem Parser. Die Aufgabe eines Parsers ist es, Daten auf Korrektheit in Bezug auf einer „Grammatik“ zu untersuchen und in einer definierten Struktur intern weiterzuverarbeiten [vgl. 19, S. 1 ff.].

Generieren und Kompilieren

Ein Testrahmen wird generiert und inklusive der analysierten Dateien kompiliert. Neben dem TPT Binding sind auch Scheduling Informationen im Testrahmen zu finden [vgl. 16, S. 868 ff.].

Testfallerstellung

es gibt folgende Steps in der Step Liste [vgl. 16, S. 437 ff.]:

Channel Step Wertzuweisung auf ein Signal

Compare Step Ein Signal wird durch das Assessment überprüft, ob es eine definierte Bedingung erfüllt

if, while Step wie in Programmiersprachen (Java, C++)

wait Damit läuft die Zeit weiter und Signale können überschrieben werden. Mit @ wird die Zeiteinheit auf die Abtastrate gestellt (siehe C-Plattform Konfiguration Step size).

Neben der Step Liste gibt es auch noch eine Partition Liste. Diese ermöglicht graphisches Erstellen von Testfallen mithilfe von sogenannten Automatons [vgl. 16, S. 437 ff.].

System under Test (SUT)

In Abbildung 2.6 ist zu sehen, wie das zu testende System mit TPT interagiert. TPT erzeugt im Sinne des Testfalls ein Signal, sendet dieses an das SUT und es wird eine Antwort zurück

an TPT geschickt. Die Antwort des SUT wird in TPT verarbeitet und gegebenenfalls ein neues Signal an das SUT geschickt, wodurch der Zyklus von Neuem beginnt. Es entsteht ein geschlossener Kreis zwischen SUT und TPT. Dieser Vorgang wird so oft wiederholt, wie es Testfälle gibt [vgl. 20].

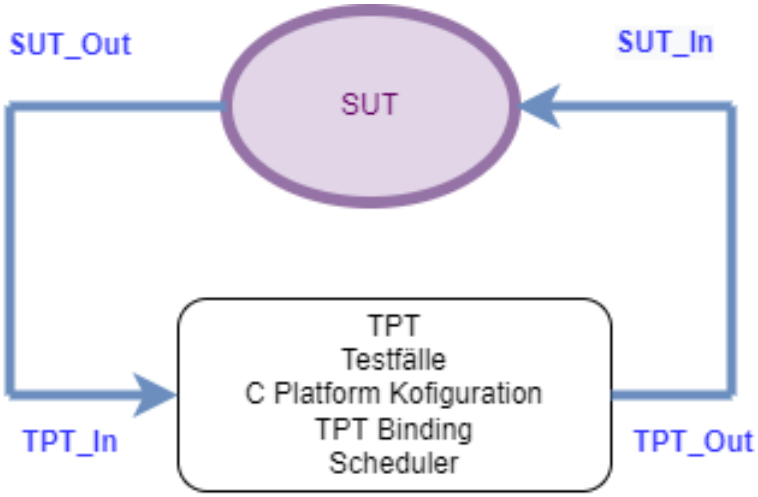

Abbildung 2.6.: Zusammenhang zwischen SUT und TPT [20]

TPTAPI

Mithilfe der TPTAPI kann man das komplette Programm steuern. Die API ist in Java geschrieben. In TPT ist Jython installiert, womit man die Java Module in Python verwenden und TPT steuern kann. Im Listing C.1 im Anhang sind nützliche Funktionen der TPTAPI aufgezählt [vgl. 16, S. 64 f.].

Äquivalenzklassen

Im *Equivalence Class Set Editor* unter *View* können Äquivalenzklassen definiert werden, um sie mehreren Signalen beziehungsweise einem einzigen Signal im Declaration Editor zuzuordnen. Sie können als Assesslet *Equivalence Classes* in der Auswertung der Testfälle eingesetzt werden. Sie können gleichermaßen in der Step Liste eingesetzt werden, um Signale auf bestimmte Äquivalenzklassen zu setzen [vgl. 16, S. 370 ff.].

Assessment Mit dem *Equivalence Class Coverage table* können ausgewählte Testfälle überprüft werden, ob Channels Werte ihrer Äquivalenzklassen annehmen. In der Report Übersicht

wird für jede Äquivalenzklasse der gewählten Channels angezeigt, ob und wie oft sie in den gewählten Testfällen vorkommen. Eine weitere Möglichkeit im Assessment ist, dass man forbidden, also verbotene sowie mandatory, verpflichtende Äquivalenzklassen festlegen kann. Eine Verbotene besteht den Test, wenn sie nicht eintritt. Eine Verpflichtende besteht den Test, wenn sie ein oder mehrere Male eintritt [vgl. 16, S. 1282 ff.].

Step Liste Es gibt vier Schlüsselwörter, um eine Äquivalenzklasse eines Channels in der Step Liste zu benutzen: *Random, representative, min* und *max*. Mit *min* und *max* sind jeweils die Grenzen des definierten Wertebereichs gemeint. Mit *random* wird ein zufälliger Wert gewählt, mit *representative* ein Repräsentant. Wenn kein Schlüsselwort angegeben wird, so wird der Repräsentant gewählt. Falls kein Repräsentant definiert ist, so wird ein zufälliger Wert innerhalb des Wertebereichs gewählt [vgl. 16, S. 379 f.].

Äquivalenzklassen von TPT generieren lassen Dies kann man mit *Generate Test Cases -> from Equivalence Classes* erreichen. Man kann auswählen ob sie in einer Step Liste nacheinander geschrieben werden oder jede in einer eigenen Step Liste. Weitere Einstellungsmöglichkeite sind eine paarweise Kombination und Grenzwerte (siehe Abschnitt 2.2) [vgl. 16, S. 668 ff.].

3. Analyse des Softcar-Quicktests

In diesem Kapitel wird ein bestehender Schnittstellentest analysiert. Der Schnittstellentest heißt bei ZF Quicktest. Der Quicktest wird in Softcar ausgeführt, was ein Programm für SiL Tests ist, das von ZF entwickelt wurde.

Schnittstellen der Projekte in meiner Abteilung werden in einer XML definiert, die Datafield heißt. Wie in Abschnitt 2.1 beschrieben, werden XML Daten an sich sowie der Zugriff auf jene im Sinne eines Schnittstellentests überprüft. Der Quicktest bei ZF begrenzt sich auf den Test des Zugriffs. Um die Schnittstellen im C Code zu verwenden, werden Makros aus der Datafield generiert. Mit den Makros kann auf die Signale zugegriffen werden (siehe Kapitel 4). Die Verbindung zweier Signale, eine Schnittstelle, erfolgt durch den Einsatz der Makros. Der Quicktest überprüft, ob diese Verbindung korrekt ist.

Das Flussdiagramm in Abbildung 3.1 gibt einen Überblick, was beim Aufruf des Quicktests ausgeführt wird. Im Grunde genommen wird eine Quicktest.c, die die Schnittstellentests beinhaltet, generiert und mit dem Gesamtprojekt kompiliert. Die kompilierte executable (exe) Datei wird in Softcar ausgeführt und fehlgeschlagene Tests werden in eine Text-Datei als error log geschrieben [vgl. 21][vgl. 22][vgl. 23].

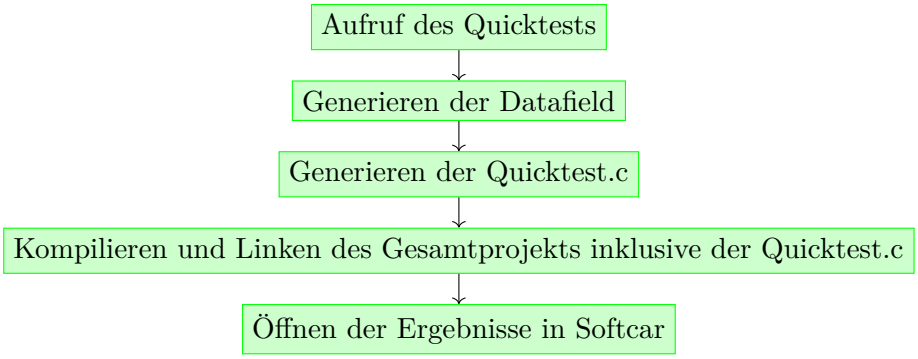

Abbildung 3.1.: Quicktestaufruf [vgl. 22]

Generieren der Datafield

Datafield ist ein ZF interner Begriff. Jede Komponente besitzt eine Datafield, in der im Wesentlichen im XML Format die Signale definiert werden. Aus den Datafields der Komponente wird eine gesamte Datafield generiert, die die Signale aller Komponente beinhaltet [vgl. 23].

Generieren der Quicktest.c

Schnittstellentests werden in diese Datei automatisiert geschrieben. Folgende Daten werden dafür benötigt:

- Signale und deren Schnittstelleninformationen aus der Datafield [vgl. 23]
- Funktionsaufrufe, in denen Schnittstellen definiert sind

Jede Schnittstelle wird mit Werten getestet, die in Abbildung 3.2 zu sehen sind.

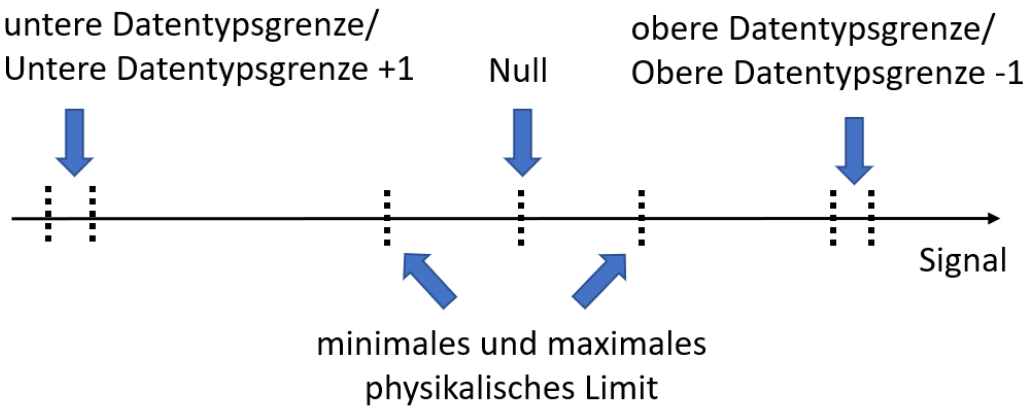

Abbildung 3.2.: Quicktest Werte auf einem Signalstrahl [vgl. 21]

Es werden jeweils die untere und obere Datentypsgrenze als Werte genommen. Das ist im Sinne einer Grenzwertanalyse, denn Grenzwerte verursachen oft ein fehlerhaftes Verhalten [vgl. 6, S. 36]. Es wird jeweils eine Ganzzahl unter der oberen und eine Ganzzahl über der unteren Grenze getestet. Die Null wird auch getestet. Ein physikalisches Limit kommt hier auch vor. Das ist ein Relikt, in dem eine Konvertierung und Skalierung zwischen zeitdiskreten Werten auf den vollen Datentypsbereich stattfand. Es hat jetzt keine Bedeutung mehr.

In Abbildung 3.3 ist ein Flussdiagramm zu sehen. Es zeigt die Logik eines Schnittstellentests, wie sie in der Quicktest.c geschrieben ist. Im späteren Verlauf wird der Einfachheit halber ein Ein- und Ausgangssignal gesprochen, die im Folgenden kurz erklärt werden.

Ausgangssignal Ein Signal, das aus der ersten Komponente austritt.

Eingangssignal Ein Signal, das in die zweite Komponente eintritt.

Eine Schnittstelle verbindet das Ausgangssignal mit dem Eingangssignal.

Zuerst wird das Ausgangssignal auf einen der Werte gesetzt. Es wird die Funktion aufgerufen, in der die Schnittstelle im Code steht. Daraufhin wird das Eingangssignal überprüft. Der Test ist erfolgreich, wenn das Eingangssignal denselben Wert hat wie der Wert, auf die das Ausgangssignal gesetzt wurde. Hat das Eingangssignal einen anderen Wert, so ist der Test fehlgeschlagen [vgl. 21][vgl. 22].

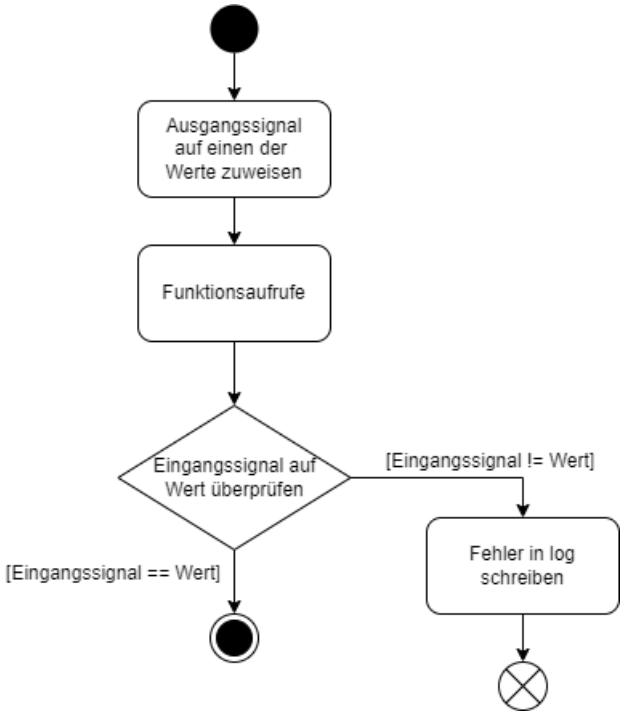

Abbildung 3.3.: Quicktest Logik [vgl. 21]

Potenziale

Ein Potenzial des Quicktests ist, dass die Laufzeit verbessert werden kann. Um die Schnittstellen zu überprüfen, ist es nicht nötig das Gesamtprojekt zu kompilieren, da jeweils nur eine C Datei einer Komponente benötigt wird. Um es möglich zu machen, dass jeweils nur eine C Datei pro Komponente analysiert wird, muss die Verbindung zu anderen Dateien getrennt und mit einem Platzhalter ersetzt werden. Dieses Abtrennen wird auch Stubben genannt [vgl. 6, S. 337]. Nach dem Status Quo ist das Stubben nicht möglich, da das Gesamtprojekt zu einer exe kompiliert wird. Beim Kompilieren zu einer exe werden alle Abhängigkeiten benötigt und es kann nicht gestubbt werden. Mit TPT können Dateien gestubbt werden.

Ein weiteres Potenzial ist, dass ein Ordner mit Artefakten im Projekt hinfällig gemacht werden kann und somit auch nicht gepflegt werden muss. Es gibt nämlich einen Ordner, in denen Funktionsaufrufe des Quicktests je Komponente in Text-Dateien stehen. Indem ein Parser sich für den Quicktest automatisiert die Funktionsaufrufe zusammensucht, müssen die Dateien nicht mehr abgelegt werden. Dadurch erreicht man, dass die Funktionsaufrufe immer up to date sind und nicht mehr händisch gepflegt werden müssen.

4. Sourcenmodell für den Quicktest

In diesem Kapitel wird eine C Datei für den Quicktest nachgestellt. Diese wird benutzt, um einen einfachen Schnittstellentest in TPT durchführen zu können. Weiterer Code aus der Datei, der nicht für den Quicktest benötigt wird, wird weggelassen.

In der Datafield stecken Informationen über die Signale und deren Schnittstellen. Um diese Schnittstelleninformationen im Code zugreifbar zu machen, werden Makros in Header Dateien generiert. Diese Header Dateien werden durch einen include-Guard der C Datei bekannt gegeben (siehe Abbildung 4.1). Damit hat die C Datei Zugriff auf die Schnittstellen. Die Makros werden im Code verwendet, um die Signale zu verbinden, sodass eine Schnittstelle entsteht.

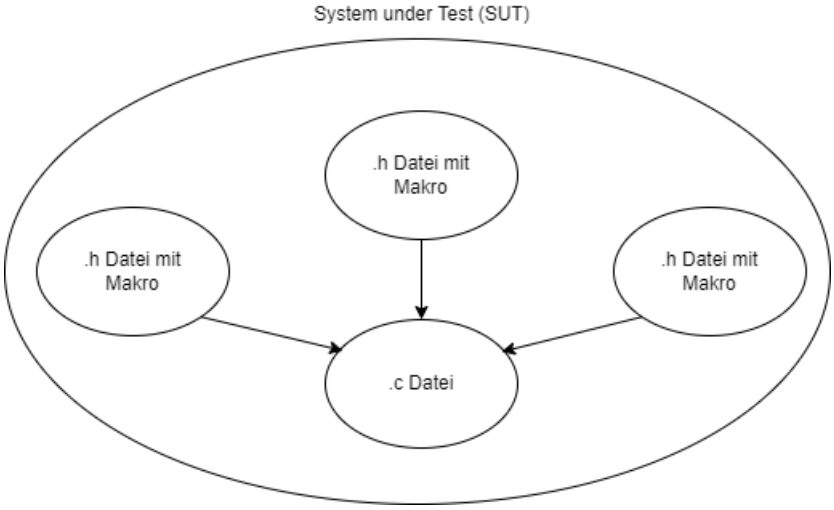

Abbildung 4.1.: Sourcenmodell für den Quicktest

In Listing 4.1 ist die C Datei zu sehen. Im Kommentar (Zeilen beginnend mit //) ist zu sehen, dass es eine einfache Wertzuweisung ist, wenn die Makros aufgelöst sind.

Listing 4.1: C Datei Quicktest

```
#include "C:\TPTlocal\TestfallmitDatafield\SDC_Public_dfi.h"
```

```
#include "C:\TPTlocal\TestfallmitDatafield\STB_Private_dfi.h"
#include "C:\TPTlocal\TestfallmitDatafield\ZIL_STB_dfi.h"

static void STB_InputDataTcl20(void)
{
  DF_STB_In_Set_IDcFild          ( DF_ZIL_STB_In_Get_IDcFild() );
  //m_STB_Private_STB_TASKCLASS20.STB_In.IDcFild=
  //m_SDC_Public_SDC_TASKCLASS20.SDC_Out.IDcFildForTcl20;
}
```

In Listing 4.2 ist eine Header Datei zu sehen. Dieses Makro hat das Prinzip eines Setters. Ein Setter wird in der objektorientierten Programmierung eingesetzt. Es ist eine Methode, die ein Attribut auf einen Wert setzt. Dieser Wert wird der Methode als Parameter übergeben. Hier ist es ein Makro und es wird anstatt eines Attributs ein Signal auf einen Wert gesetzt [vgl. 24, S. 414 f.].

Listing 4.2: Setter Makro

```
#define DF_STB_In_Set_IDcFild( arg_Value ) \
        (m_STB_Private_STB_TASKCLASS20.STB_In.IDcFild = (arg_Value))
```

In Listing 4.3 ist die nächste Header Datei zu sehen. Dieses Makro setzt ein weiteres Makro ein.

Listing 4.3: Makro verweist auf ein weiteres Makro

```
#define DF_ZIL_STB_In_Get_IDcFild() \
        (DF_SDC_Out_Get_IDcFildForTcl20())
```

In Listing 4.4 ist die dritte Header Datei zu sehen. Dieses Makro hat das Prinzip eines Getters und kommt ähnlich wie der vorher beschriebene Setter aus der objektorientierten Programmierung. Das Getter Makro gibt in diesem Fall ein Signal wieder.

Listing 4.4: Getter Makro

```
#define DF_SDC_Out_Get_IDcFildForTcl20() \
        (m_SDC_Public_SDC_TASKCLASS20.SDC_Out.IDcFildForTcl20)
```

Löst man nun alle drei Makros auf, wird wie zuvor schon erwähnt, erkenntlich, dass es eine Wertzuweisung ist. Das Setter Makro bekommt als Parameter das Getter Makro. Der Wert des Signals einer anderen Komponente wird in ein Signal dieser Komponente eingesetzt. Dies entspricht einer Schnittstelle.

5. Implementierung des Quicktests in TPT

In diesem Kapitel wird der Quicktest in TPT überführt. Es werden die Einstellungen in TPT getroffen. Es werden die Testfälle erstellt und schließlich wird ein Skript geschrieben, das die Testfälle automatisiert generiert.

C Platform Konfiguration

Compiler auswählen Pfad zu mingw Installation (C Compiler)

Sourcen auswählen nach design pattern alle <Komponente>_Task.c

Scheduler Funktionsaufrufe in Task Dateien

Step size 125µs

Vom bestehenden Quicktest gibt es einen Ordner, wo pro Komponente die Funktionen, in denen die Schnittstellen definiert sind, aufgeführt sind (siehe Kapitel 3).

Testfallerstellung

Die notwendigen Schnittstelleninformation für die Testfallerstellung können aus der Datafield entnommen werden (siehe Kapitel 3). Das physikalische Limit wird nicht vom bestehenden Softcar-Quicktest aufgenommen, da es wie vorher beschrieben ein Relikt ist und nicht mehr benötigt wird. In TPT werden die Testfälle als Step Liste mit folgendem Inhalt erstellt:

Channel Step Ausgangssignal auf einen der Werte zuweisen

Compare Step Eingangssignal mit Wert vergleichen, falls float: mit 0,00001 Fehlertoleranz

wait @

Hier ist ein Beispiel eines Testfalls in der Step Liste:

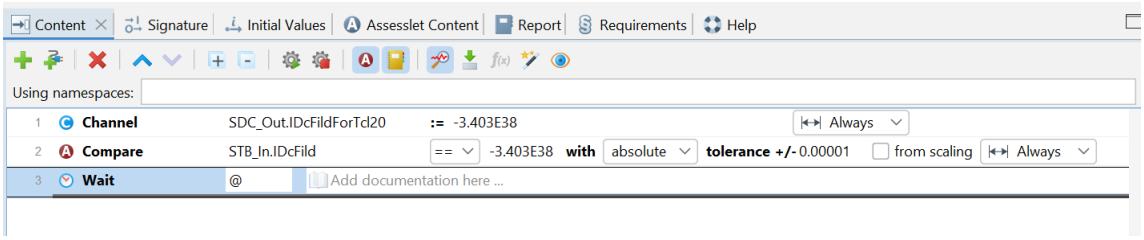

Abbildung 5.1.: Schnittstellen Testdesign

Automatisierung der Testfallerstellung

Mithilfe der TPTAPI werden die Testfälle automatisch erstellt. Die Funktionalität wird anhand von Klassendiagrammen, die sich im Anhang befinden, erklärt. Es gibt drei wesentliche Klassen im Jython Skript, nämlich Types, Interface und InterfacesByUnit, wobei diese durch eine Aggregation verbunden sind (siehe Abbildung A.1). Eine Aggregation ist eine Ganzes-Teile-Beziehung [vgl. 6, S.61 f.]. In allen Klassen kommen Getter und Setter vor, die in Kapitel 4 erklärt werden.

Die Objekte der Klasse Types (siehe Abbildung A.3) enthalten Informationen des Datentyps für die Signale. Sie besitzt die Attribute für den Namen des Datentyps, untere und obere Wertegrenze und ob es eine Fließkommazahl ist sowie dazugehörige Getter und Setter Methoden.

Die Objekte der Klasse Interface (siehe Abbildung A.4) enthalten Informationen über eine Schnittstelle, sprich das Ausgangssignal sowie das Eingangssignal (siehe Kapitel 3) sowie der Datentyp, der durch ein Objekt der Klasse Types eingebunden wird. Sie besitzt die Attribute für den Namen des Testfalls und der beiden Signale sowie dazugehörige Getter und Setter. Des Weiteren besitzt sie Methoden zur Erstellen der Step Liste.

Die Objekte der Klasse InterfacesByUnit (siehe Abbildung A.2) enthalten jeweils alle Interface Objekte einer Unit/Komponente. Sie besitzt die Attribute für den Namen, den Pfad zur C Datei, in der die Schnittstellen definiert sind und eine Liste, die sich mit Objekten der Klasse Interface füllt, sowie dazugehörige Getter und Setter.

In Abbildung B.1 ist in einem Flussdiagramm zu sehen, wie die Testfälle generiert werden. Es besteht im Wesentlichen aus zwei Schleifen. Die große Schleife ist dafür zuständig Testfallordner in TPT zu erstellen. Jeder Ordner ist einer Komponente zugeordnet, wobei die Ordnerstruktur die gleiche ist wie im Softwareprojekt. Die zweite Schleife ist innerhalb der ersten Schleife. Darin wird für jede Schnittstelle ein Testfall erstellt und mit Steps gefüllt.

6. Konzept für einen Äquivalenzklassentest in TPT

In diesem Kapitel soll ein Konzept erarbeitet werden, wie der Äquivalenzklassentest sinnvoll eingesetzt werden kann. Es wird gezeigt, wie sie definiert werden können. Es wird eine Selbstüberprüfung gezeigt. Es werden bestehende SIL Tests überprüft, ob alle Wertebereiche abgedeckt werden.

Abbildung 6.1 soll als Beispiel einer Berechnung gelten. Es beinhaltet eine Switch, die

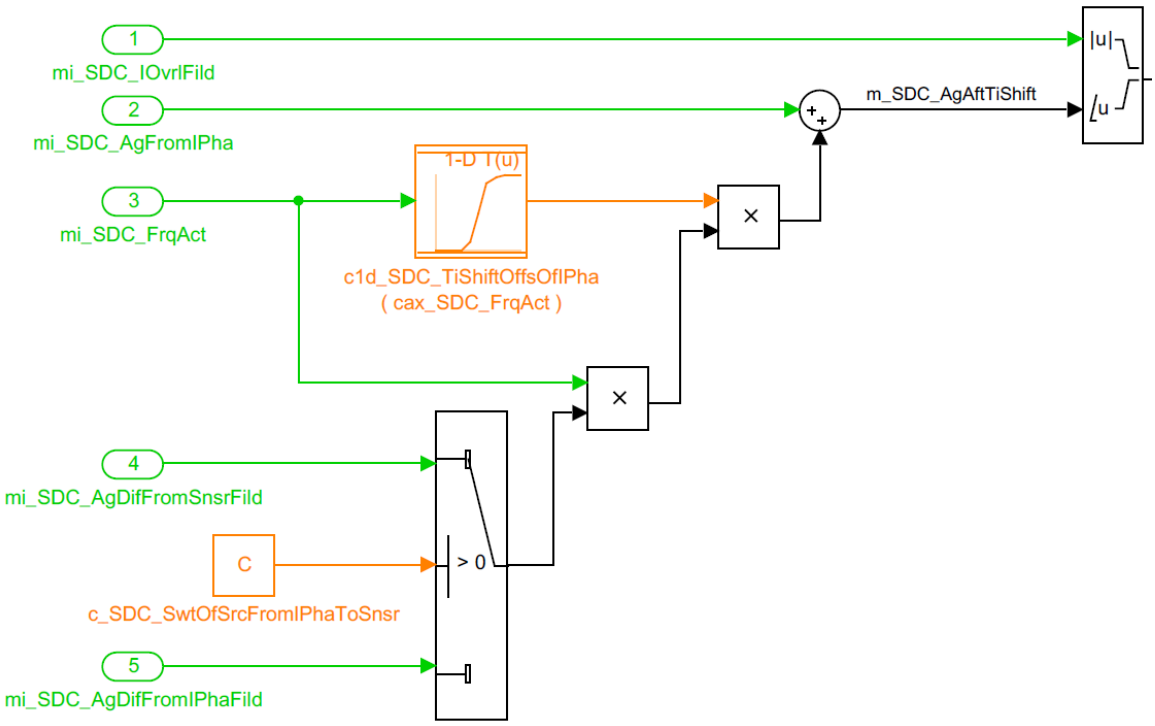

Abbildung 6.1.: Beispiel einer Berechnung

zwischen zwei Eingangsgrößen umschaltet. Es sind insgesamt vier mögliche Eingangsgrößen, die durch Addition und Multiplikation sowie einer Funktion ein Ergebnis berechnen.

Definieren

Als Grundlage dafür wird ein bestehender SIL Test genommen. Wenn der Test nach dem Requirement vollständig und richtig ist, so sind darin alle Werte der Signale enthalten, die für das Requirement notwendig sind. Diese Werte können als Repräsentanten verwendet werden und dafür ein Wertebereich um den Repräsentanten festgelegt werden. Eine Äquivalenzklasse soll ein einziges bestimmtes Verhalten abbilden. Dabei sollen alle Werte sich gleich verhalten. Wenn es hier Unterschiede gibt, ist es sinnvoll, sie in kleinere Wertebereiche zu zerlegen. Beispielsweise sollen nach dem Requirement verschiedene Frequenzen getestet werden. So werden Äquivalenzklassen für niedrige, für mittlere, eine für hohe und eine für gefährlich hohe Werte erstellt. Es soll zweitrangig sein, welcher Wert in der jeweiligen Klasse genommen wird, da alle Werte nach dem Requirement gleich anzusehen sind.

Selbstüberprüfung

Mit diesem Test soll überprüft werden, ob sie in sich Sinn machen. Es wird getestet, ob alle Äquivalenzklassen erreichbar sind. In dem Fall des Beispiels werden alle Wertebereiche der Eingangsgrößen der Reihe nach gesetzt und es wird überprüft, ob alle Wertebereiche des Ergebnissignals erreicht werden. Zuerst werden alle Eingangsgrößen auf einen Repräsentanten zur Initialisierung gesetzt. Eine Step Liste wird nach dem Minimalkriterium mit den gesetzten Äquivalenzklassen durch TPT generiert. Bei einer korrekten Definition sollten alle erreicht werden und die Selbstüberprüfung erfolgreich sein.

Überprüfen bestehender SiL Tests

Es werden alle Parameter eines SiL Tests überprüft. Dies wird mit dem *Equivalence Class Coverage table* durchgeführt. Der Sinn dieses Tests ist, den SiL Test zu validieren. Durch diesen Test wird ersichtlich, ob der SiL Test bezüglich der Wertebereiche vollständig ist.

weitere Einsatzmöglichkeiten

Um den Äquivalenzklassentest nutzen zu können, müssen zunächst Äquivalenzklassen definiert werden. Das Definieren ist umfangreich und zeitaufwendig. Es bietet jedoch Vorteile, denn dadurch schafft man eine Datenbasis an Werten als Repräsentanten beziehungsweise Wertebereichen. Nach dem Status Quo steht der Testfall im Vordergrund und die Auswahl der Werte für den Test ist nur ein Mittel zum Zweck. Durch die neue Testart wird der Blickwinkel auch auf die

Testwerte gelegt. Die Auswahl der Werte ist genauso wichtig wie der Testfall selbst, denn ein Test mit den falsch gewählten Werten ist genauso hinderlich wie ein schlechtes Test Design. Ein weiterer wichtiger Punkt ist, dass, sobald die Äquivalenzklassen definiert sind, man sie vielfältig in allen möglichen Testfällen wiederverwenden kann. In TPT ist der Zugriff darauf beispielsweise in der Step Liste sehr einfach möglich durch *<Signalname>->ec.random* sowie anderen Schlüsselwörtern (siehe Abschnitt 2.3).

7. Diskussion der Ergebnisse

Schnittstellentest

Es ist nicht gelungen die Laufzeit des Quicktests zu verbessern. Es ist noch ein großes Potenzial vorhanden, denn TPT muss noch besser für das Softwareprojekt konfiguriert werden. Derzeit ist es nicht möglich einzelne Dateien von TPT analysieren und kompilieren zu lassen. Da die Schnittstellen jeweils nur in einer Datei einer Komponente definiert sind, könnte sehr viel Laufzeit eingespart werden. Das Jython Skript, das die Testfälle für den Quicktest automatisiert generiert, funktioniert soweit. Es legt Ordner an, so wie es im Softwareprojekt auch der Fall ist, sodass sich Entwickler schnell zurecht finden können. Es ist zwar nicht die Gesamtlaufzeit verbessert worden, aber es ist jetzt möglich, dass einzelne Komponenten oder gar Schnittstellen alleine getestet werden.

Äquivalenzklassentest

Es wurde eine gute Grundlage geschaffen, sodass ein Äquivalenzklassentest in naher Zukunft eingeführt werden kann. Es wurde ein Konzept erarbeitet, das man schnell einführen kann. Es wurde gezeigt, wie SiL Tests überprüft werden kann. Es wurde gezeigt wie Testfälle mit Äquivalenzklassen von TPT generiert werden können. Die große Arbeit ist alle Äquivalenzklassen für jedes Signal zu definieren. Durch das Definieren wird eine Datenbasis geschaffen, die auch für andere Tests eingesetzt werden kann.

8. Zusammenfassung und Ausblick

Zusammenfassung

Der bestehende Quicktest von ZF wird analysiert und in TPT übertragen. Es wird ein Code nachgestellt, in der die Schnittstelle definiert ist. Es wird ein Skript in Jython geschrieben, das automatisiert die Testfälle in TPT generiert.

Es wird ein Konzept für den Äquivalenzklassentest vorgestellt. Der Äquivalenzklassentest ist eine gute Erweiterung, insbesondere für bestehende SiL Tests. Es wird auch gezeigt, wie Äquivalenzklassen selbst überprüft werden können. Es wird erklärt, wie Äquivalenzklassen gebildet werden können.

Ausblick

Als Potenzial für den Quicktest wird in Kapitel 3 genannt, dass ein Parser Funktionsaufrufe, in denen die Schnittstellen definiert sind, finden könnte. Dadurch müssen die Funtionsaufrufe nicht mehr in einem Ordner abgespeichert werden. Dieser Parser wurde noch nicht umgesetzt. TPT muss weiter für die bestehende Software konfiguriert werden. Wenn es durch die Konfiguration möglich ist, dass nur die Dateien, die für den Quicktest nötig sind, kompiliert werden, so ergibt sich auch eine schnellere Laufzeit des Quicktests in TPT. Die Äquivalenzklassen müssen für jedes Signal erstellt werden, um Äquivalenzklassentests einführen zu können.

Eidesstattliche Erklärung

Die vorliegende Abschlussarbeit wurde von mir selbstständig verfasst und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt. Wörtliche und sinngemäße Zitate im Text sind als solche gekennzeichnet.

Schweinfurt, 30. April 2022

Xaver Gruber

Abbildungsverzeichnis

Quellcodeverzeichnis

A. Klassendiagramme der Quicktestimplementierung

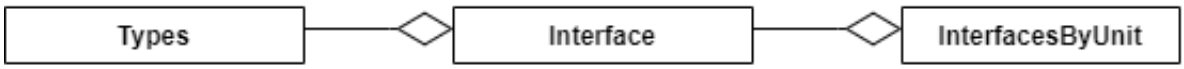

Abbildung A.1.: Aggregation der Klassen

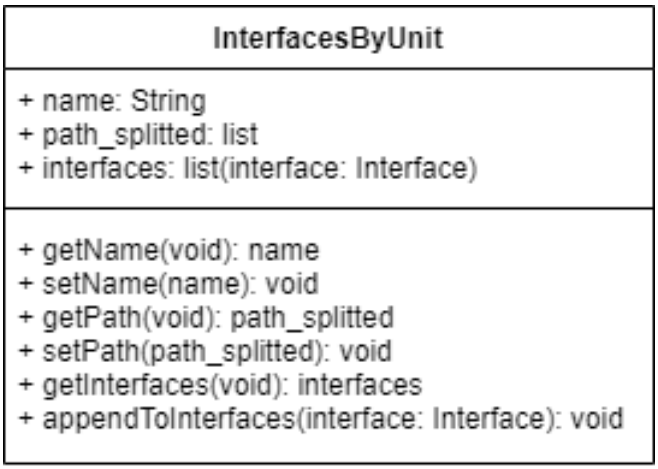

Abbildung A.2.: Klasse InterfacesByUnit

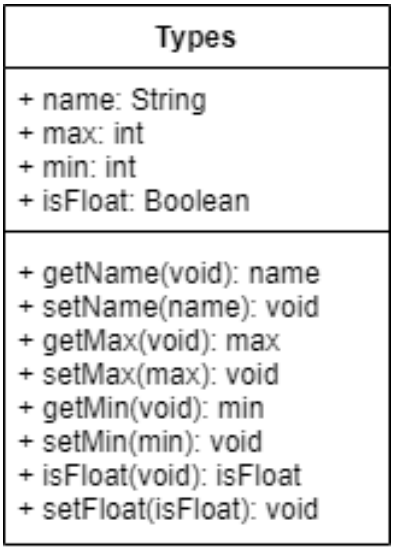

Abbildung A.3.: Klasse Type

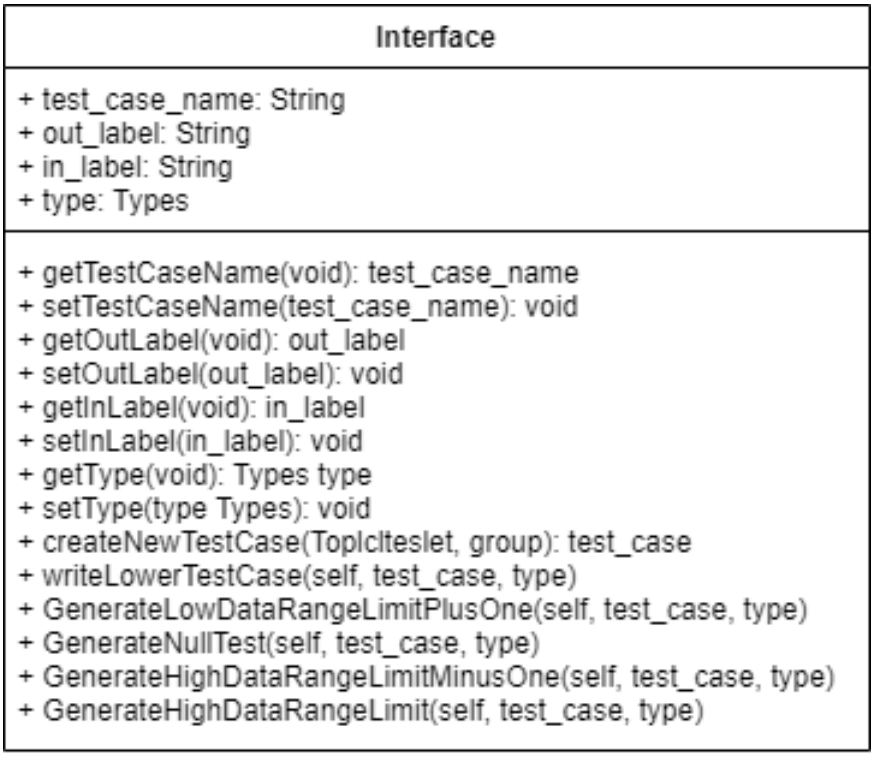

Abbildung A.4.: Klasse Interface

B. Flussdiagramm der Quicktestimplementierung

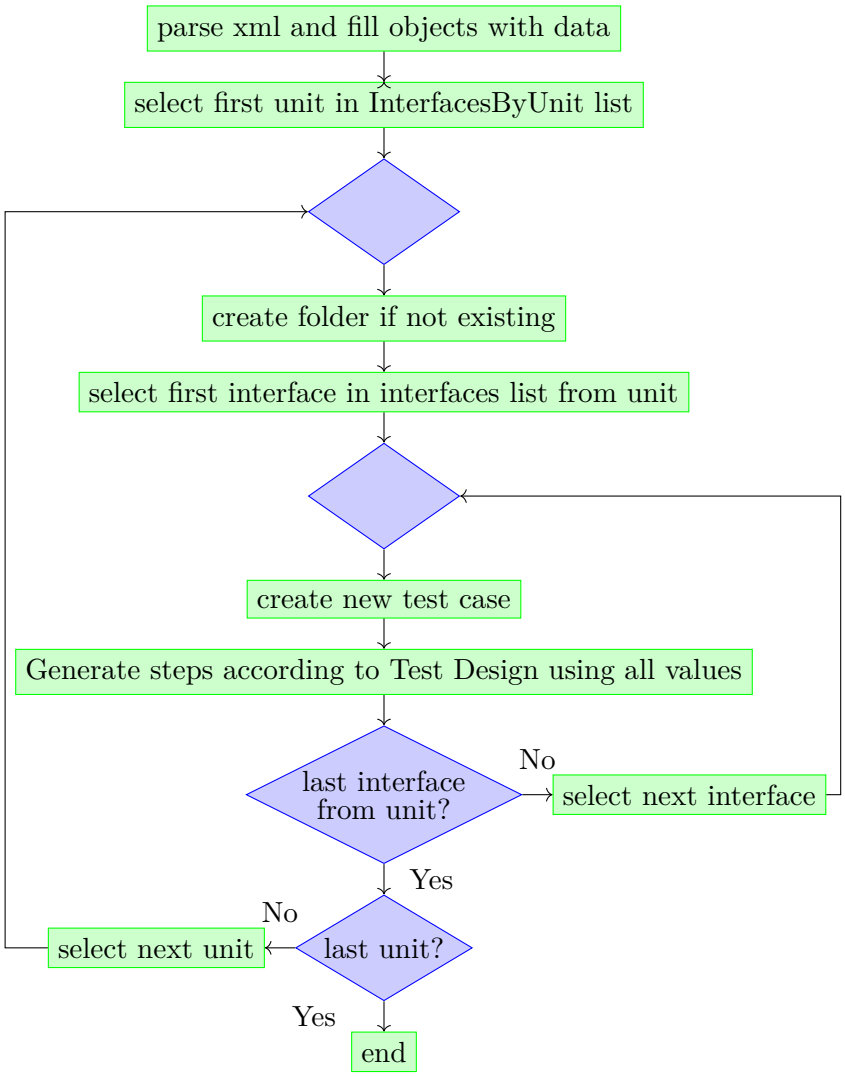

Abbildung B.1.: Automatisierung der Testfallerstellung

C. TPTAPI Funktionen

Listing C.1: TPTAPI Funktionen

```
#1: ein Projekt oeffnen
openResult =
    TPTAPI.openProject(File("C:\TPTlocal\ProgrammTestfaelle\GenerateTestCases.tptz"))
    # open tpt-File
project = openResult.getProject()


#2: eine Step Liste erstellen und mit Steps fuellen
#topLevelTslt = project.getTopLevelTestlet()
#listemitsteps = topLevelTslt.createSLVariant("Test Stepliste", None)
#dererstestep=listemitsteps.createStep(0, "channel" )
#listemitsteps.importTestSpecification("Set
    m_SDC_Public_SDC_TASKCLASS20.SDC_Out.IDcFildForTcl20 to -2147483648
    (once)")
#listemitsteps.importTestSpecification("Compare
    m_STB_Private_STB_TASKCLASS20.STB_In.IDcFild == -2147483648 (First)")
#listemitsteps.importTestSpecification("Wait 1")


#3: einen Testfall exportieren
testfallfuerexport=
    project.getScenarioOrGroupByName("zuExportierenderTest")
exportedscenario= testfallfuerexport.exportTestSpecification()


#4: eine TestSet erstellen
gruppe=project.createTestSetGroup("EinNeuesTestSet", None)
print(gruppe.getName())


#5: ein Assessment erstellen
project.createAssessmentGroup("einNeuesAssessment", None)
```

```
#6: eine neue Gruppe/Ordner fuer Testfaelle erstellen
topLevelTslt = project.getTopLevelTestlet()
neuegruppe=topLevelTslt.createVariantGroup("NeueGruppe", None) #auf
    hoechster Ebene
topLevelTslt.createVariantGroup("NeueUnterGruppe", Ordner)  #innerhalb
    eines Ordners

#7: eine/n Gruppe/Ordner bzw Testfall(Scenario) finden
gruppe=project.getScenarioOrGroupByName("STB")
gefundenegruppe=project.getScenarioOrGroupByName("PTS_In_ACS_Out_UmodPhy")


#8: Steps innerhalb einer Step Liste loeschen
gefundenegruppe.getSteps().clear()


#9: Testfaelle innerhalb einer Gruppe loeschen
gruppe.clear()


#10: Testfall benennen
ScenarioGroup.setName("EinNeuerName")


#11: Testfall(Scenario) durch Namen finden
gefunden=ScenarioGroup.getName("zuFindenderTestfall")
```

Literatur

[1] *ZF Webseite*, 2. Feb. 2022. Adresse: https://www.zf.com/mobile/de/company/company.html.

[2] O. Maibaum, A. Herrmann, H. Schumann u. a., "Abschlussbericht des Projekts SiLEST (Software in the Loop for Embedded Software Test) in der BMBF-Forschungsoffensive Software Engineering 2006," 2004.

[3] *Hardware in the Loop - Wikipedia Eintrag*, 25. Apr. 2022. Adresse: https://de.wikipedia.org/wiki/Hardware_in_the_Loop.

[4] *Die größten Kostenfallen - AP Verlag*, 25. Apr. 2022. Adresse: https://ap-verlag.de/die-groessten-kostenfallen-in-der-softwareentwicklung-und-was-man-dagegen-tun-kann/58993/.

[5] R. Black und G. Coleman, *Agile Testing Foundations*. BCS Learning und Development, 2017, ISBN: 1780173369.

[6] M. Winter, M. Ekssir-Monfared, H. M. Sneed, R. Seidl und L. Borner, *Der Integrationstest*. Hanser Fachbuchverlag, 2012, ISBN: 9783446429512.

[7] A. Spillner und T. Linz, *Basiswissen Softwaretest*. dpunkt.verlag, 2012, ISBN: 9783864900242.

[8] *Gegenüberstellung und Bewertung verschiedener Testentwurfsverfahren*, 25. Apr. 2022. Adresse: https://heicon-ulm.de/gegenueberstellung-und-bewertung-verschiedener-testentwurfsverfahren/.

[9] *Statische Analyse und Dynamischer Test: Wo liegen die Stärken und Schwächen?* 25. Apr. 2022. Adresse: https://heicon-ulm.de/statische-analyse-und-dynamischer-test-wo-liegen-die-staerken-und-schwaechen/.

[10] *TPT - PikeTec Webseite*, 25. Apr. 2022. Adresse: https://piketec.com/de/tpt/.

[11] *TPT - Wikipedia Eintrag*, 25. Apr. 2022. Adresse: https://de.wikipedia.org/wiki/TPT_(Software).

[12] E. Lehmann, "Time Partition Testing-Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen - Dissertation," 2003.

[13] S. Bosch, *Lineare Algebra*. Springer Berlin Heidelberg, 2014, ISBN: 978-3-642-55259-5.

[14] *Relationen - Theoretische Informatik*, 23. Apr. 2022. Adresse: `www.theoretische-informatik.com/relationen`.

[15] *Transitive Relation - Wikipedia Eintrag*, 23. Apr. 2022. Adresse: `https://de.wikipedia.org/wiki/Transitive_Relation`.

[16] TPT, "User Guide," 4. Apr. 2022.

[17] R. Bharadwaj, *Mastering Linux Kernel Development : A Kernel Developer's Reference Manual.* Packt Publishing, 2017, ISBN: 9781785883057.

[18] TPT, "TPT Tutorial," 15. Jan. 2022.

[19] S. Kübler, *Memory-based Parsing.* (Natural Language Processing v. 7). John Benjamins Publishing Company, 2004, ISBN: 9789027249913.

[20] TPT, *TPT or SUT synchronous Test Execution*, 7. Apr. 2022. Adresse: `https://files.piketec.com/downloads/releases/TPT17/manuals/Content/Test%20Execution/TPT%20or%20SUT%20synchronous%20Test%20Execution.htm`.

[21] ZF, "Quicktest.c," 7. März 2022.

[22] ZF, "Quicktest.log," 20. Jan. 2022.

[23] ZF, "Datafield.xml," 7. März 2022.

[24] C. Ullenboom, *Java ist auch eine Insel. Einführung, Ausbildung, Praxis.* Rheinwerk Computing, 2016, ISBN: 978-3-8362-4119-9.