Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 249

Analyse und Erweiterung des ibb TestDesigners

David Michel

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer/in:	M. Sc. Asim Abdulkhaleq Dipl.-Ing. (FH) Peter Heidenwag
Beginn am:	20. Juli 2015
Beendet am:	3. Dezember 2015
CR-Nummer:	D.2.2, J.2

Kurzfassung

Das Testen von Steuergeräten in Automobilen gewinnt zunehmend an Bedeutung. Das Ingenieurbüro Brinkmeyer & Partner hat sich auf den Aufbau von individuellen Testsystemen und die Erstellung von Tests für Steuergeräte spezialisiert. Zur Generierung von Programm Code für Tests wird der vom Unternehmen entwickelte ibb TestDesigner genutzt. Diese Software erzeugt XML-Code, der auf der Testumgebung ausgeführt werden kann.

Im Rahmen dieser Arbeit soll das bestehende Konzept beschrieben und die Erzeugung eines weiteren Ausgabeformats ermöglicht werden. Dieses weitere Format für die Testbeschreibung ist eine Programmiersprache namens CAPL. Mithilfe von CAPL sollen sich mehr Möglichkeiten für eine Testdefinition ergeben und Funktionalitäten erreichbar sein, die allein mit XML nicht möglich waren.

Abstract

The importance of testing electronic control units is significantly increasing. The company Ingenieurbüro Brinkmeyer & Partner has specialized in building individual test systems and creating tests for electronic control units. In order to generate program code for tests, the company developed the ibb TestDesigner. This software creates XML-code which can be run on the test environment.

Within the scope of this thesis, the existing concept is to be described and the generation of an additional output format shall be realized. This additional format for test description is a programming language named CAPL. Via CAPL, more possibilities for test definition shall result, along with new functionalities which were not possible by solely using XML.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation

Der *ibb TestDesigner* findet sowohl beim Entwickler, dem *Ingenieurbüro Brinkmeyer & Partner (ibb)*, als auch bei dessen Kunden seit längerem Anwendung bei der Definition von automatischen Elektroniktests. Das hierbei verwendete Ausgabeformat *Extensible Markup Language (XML)* bietet ausreichend Möglichkeiten zur Beschreibung der einzelnen Tests. Trotzdem können weitere Ausgabeformate wie z.B. *CAN Access Programming Language (CAPL)* die Testdefinition verbessern, da die Sprache speziell für *CANoe* entwickelt wurde. Es ist daher ein Anliegen von *ibb*, dass mit dem bestehenden Konzept ein weiteres Ausgabeformat realisiert wird. Dies soll im Rahmen einer Bachelorarbeit im Hause *ibb* geschehen. Dipl.-Ing. (FH) Peter Heidenwag, einer der Geschäftsführer von *ibb*, entwickelte die Konzepte hinter der Software und betreut diese Bachelorarbeit innerhalb der Firma.

1.2 Problemstellung

Jahr für Jahr erhöht sich die Komplexität elektronischer Verschaltung in Automobilen. Bereits 2011 waren in einem Oberklasse Wagen ungefähr 80 verschiedene Steuergeräte verbaut [Rei11]. Je mehr Aufgaben durch Steuergeräte übernommen werden, desto wichtiger ist deren einwandfreie Funktion. Um diese während der Entwicklung zu erreichen und am fertigen Produkt zu überprüfen, werden automatisierte Tests durchgeführt. Eine Testumgebung für solche automatischen Tests stellt die Software *CANoe* dar. Lange Zeit jedoch war die einzige Möglichkeit Tests für *CANoe* zu definieren, sie in der Sprache *CAPL* manuell zu verfassen. Um diesen zeitaufwändigen Prozess zu optimieren, entwickelte *ibb* den *ibb TestDesigner*, bei dem Tests über eine graphische Oberfläche modular aufgebaut werden können. Anschließend können die Tests im Dateiformat *XML* exportiert und in *CANoe* eingelesen werden. Da ein *CAPL* Testmodul jedoch eine Reihe von Funktionen bietet, die mit einer *XML*-Datei nicht oder nur schwer realisiert werden können, wäre es als weiteres Ausgabeformate für den *ibb TestDesigner* wünschenswert. Dadurch wird eine noch schnellere und mächtigere Testerstellung ermöglicht.

1.3 Ziele

Das primäre Ziel dieser Arbeit ist die Erweiterung der Ausgabeformate des *ibb TestDesigners* um die Sprache *CAPL*. Damit sollen zukünftig Tests erstellt und mit *CANoe* ausgeführt werden können. Anhand eines konkreten Beispiels wird demonstriert, ob und in wie weit dies gelungen ist.

Weiterhin soll eine detaillierte Beschreibung der Software selbst und des Konzepts, das der Software zugrunde liegt, erfolgen. Da es sich um ein umfangreiches und vielseitiges Tool handelt, wird allerdings nur auf die wesentlichen Funktionalitäten näher eingegangen.
Außerdem soll geprüft und aufgezeigt werden, für welche weiteren Anwendungsbereiche sich das allgemein gehaltene Konzept des *ibb TestDesigners* eignet. Anstelle von Programm Code könnte nämlich auch Text jeglicher Form für unterschiedliche Zwecke generiert werden.

1.4 Aufbau der Arbeit

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: In diesem Kapitel werden die Grundlagen der Arbeit beschrieben. Dazu gehören sowohl Software als auch Hardware Komponenten und deren Zusammenhang.

Kapitel 3 – Analyse des ibb TestDesigners: Dieses Kapitel befasst sich mit dem *ibb TestDesigner*, erklärt dessen Funktionsweise und Konzepte und grenzt das Programm gegenüber vergleichbarer Software ab.

Kapitel 4 – Entwurf & Implementierung: Hier wird zuerst näher auf die Voraussetzungen und Anforderungen an eine Implementierung eingegangen. Anschließend wird dargelegt, wie die Umsetzung erfolgt ist.

Kapitel 5 – Evaluation: In diesem Kapitel wird anhand eines konkreten Beispiels aufgezeigt, wie die neu implementierte Funktionalität in der Praxis angewandt werden kann. Hierzu wird ein Test für ein Steuergerät erzeugt, ausgeführt und analysiert.

Kapitel 6 – Weitere Anwendungsgebiete: Dieses Kapitel zeigt, wie das Konzept des *ibb TestDesigners* auf weitere Anwendungsgebiete ausgeweitet und angepasst werden kann.

Kapitel 7 – Zusammenfassung & Ausblick: Das letzte Kapitel fasst die Arbeit zusammen und liefert einen Ausblick in die Zukunft.

2 Grundlagen

Im Folgenden soll auf die Grundlagen eingegangen werden, die notwendig sind um diese Arbeit zu verstehen. Diese umfassen einen Überblick über *CANoe*, das *VT System*, die Programmiersprache *CAPL*, den *ibb TestDesigner* und das Zusammenwirken dieser Komponenten.

2.1 CANoe

Die Software *CANoe* wurde erstmals 1996 von der *Vector Informatik GmbH* veröffentlicht und seitdem stetig weiterentwickelt. Die aktuelle Version ist 8.5 [Vec15a]. Das Hauptanwendungsgebiet von *CANoe* ist die Entwicklung und das Testen von Steuergeräten im Automobilbereich. Solche Steuergeräte sind eingebettete Systeme mit Sensoren, die Messwerte sammeln. Weichen diese Messwerte vom Sollwert ab, so wird mittels Aktoren versucht, den Sollwert einzustellen. Die Basis für die Kommunikation der Steuergeräte sind Bussysteme wie *Controller Area Network (CAN)* oder *Local Interconnect Network (LIN)*. Derartige Steuergeräte sind aus modernen Autos nicht mehr wegzudenken. Sie sind z.B. zuständig für die Funktionsweise folgender Elemente:

- Motor
- Bremsen
- Fensterheber
- Anhängerkupplung

Um Steuergeräte zu simulieren und zu testen, kann eine Software wie *CANoe* eingesetzt werden. Darin können Tests importiert oder auch erstellt werden, die dann vollautomatisch durchgeführt und anschließend ausgewertet werden können. Dies kann jedoch nicht nur zum Testen von virtuell erzeugten Modellen von Steuergeräten eingesetzt werden, sondern auch um tatsächliche Hardware Komponenten zu testen. Zu diesem Zweck hat *Vector Informatik* das *VT System* entwickelt.

2.2 VT System

Das *VT System* besteht aus modularen Hardware Komponenten, mithilfe derer die Eingabe- und Ausgabeschnittstellen der Steuergeräte mit einem Computer verbunden werden können, auf dem Tests ausgeführt werden [Vec15d]. Diese Module sind z.B. Netzteile, elektronische Widerstandslasten oder allgemein Platinen, die analoge und digitale Signale verarbeiten. Durch das Zusammenfügen der einzelnen Module kann eine Hardware Testumgebung geschaffen werden, die speziell auf einen

bestimmten Prüfling zugeschnitten ist und dessen korrekte Funktionsweise validiert. Mittels *CANoe* können dann z.B. einzelne Relais geschalten oder Eingänge angesteuert werden.
Zum Erstellen von Tests liefert *CANoe* vorgefertigte Pakete und einen Editor. Die Programmiersprache, die den Tests zugrunde liegt, heißt *CAN Access Programming Language (CAPL)*.

2.3 CAN Access Programming Language

Die *CAN Access Programming Language (CAPL)* wurde von *Vector Informatik* entwickelt und wird als native Sprache für Tests und Knoten in *CANoe* genutzt [Vec09]. *CAPL* basiert auf *C* und optimiert dieses dahingehend, dass u.a. Messungen und Stimulationen mehrerer verschiedener Kanäle ermöglicht werden. Ein grundlegender Unterschied zu *C* ist der ausschließlich ereignisbasierte Ablauf des Programm Codes. Zu solchen Ereignissen zählen

- Ablauf eines Timers
- Änderung einer Variable
- Erhalten einer Nachricht

Einige Konzepte von *C* werden in *CAPL* nicht unterstützt, da sie keine Relevanz für das beabsichtigte Anwendungsgebiet haben. Diese weggelassenen Elemente umfassen z.B.

- Pointer
- Header Files
- Structures
- Definieren von Makros

Die Programmierung von *CAPL* erfolgt idealerweise im *CANoe*-eigenen *CAPL Editor*, es kann allerdings auch in einem beliebigen Text Editor programmiert werden [Vec04].
Um den Prozess der Testerstellung zu optimieren, sodass nicht jeder Test von Hand in *CAPL* geschrieben werden muss, hat das *Ingenieurbüro Brinkmeyer & Partner (ibb)* eine eigene Software entwickelt, den *ibb TestDesigner*.

2.4 ibb TestDesigner

Der *ibb TestDesigner* ist ein vielseitiges Tool, welches zum Ziel hat, die Arbeit mit *CANoe* maßgeblich zu erleichtern. Das Hauptaugenmerk liegt hierbei auf der einfachen Erstellung von Tests, doch auch weitere Funktionalitäten werden im späteren Verlauf dieser Arbeit dargestellt. Über die Jahre ist das Programm stetig weiterentwickelt worden, um sich neu ergebenden Anforderungen anzupassen.
Die Erstellung von Tests erfolgt in der Regel Requirement-basiert, was durch den Import von *Microsoft Excel*-Dateien unterstützt wird. Durch das Zurückgreifen auf eine von *ibb* entwickelte Methodenbibliothek können dann Schritt für Schritt Tests erstellt werden, die die Requirements überprüfen. Der Aufbau eines Tests erfolgt mittels einer zweidimensionalen Tabelle, was die Übersicht gegenüber dem

manuellen Programmieren in *CAPL* oder Schreiben einer *XML*-Datei wesentlich erhöht. Am Ende des Testerstellungsprozesses steht ein *XML* Dokument, das einen genau spezifizierten Testablauf enthält. Diese Datei kann in *CANoe* importiert werden, um dann die Funktionsweise des Prüflings zu testen.

2.5 Prozesskette

Eine grobe Prozesskette für das Testen von Steuergeräten unter Einsatz von *CANoe* als Testumgebung und dem *ibb TestDesigner* zur Definition von Tests ist in Abb. 2.1 skizziert.

Der *ibb TestDesigner* wird genutzt, um Schritt für Schritt Tests zu definieren. Hierbei orientiert er sich an den Requirements, die für den Test vorgegeben werden. Das Resultat der Testerstellung ist eine Datei, die von *CANoe* eingelesen werden kann. Über das *VT System* steuert *CANoe* den Prüfling an und führt die Tests durch. Das *VT System* befindet sich in ständiger Kommunikation mit *CANoe* und liefert Daten. Wenn der Test beendet ist, generiert *CANoe* eine detaillierte Auswertung des Testablaufs in Form eines Reports. Diese Report Datei kann dann sowohl in *CANoe* angesehen werden, als auch in den *ibb TestDesigner* geladen werden, um wiederum systematische Analysen zu ermöglichen.

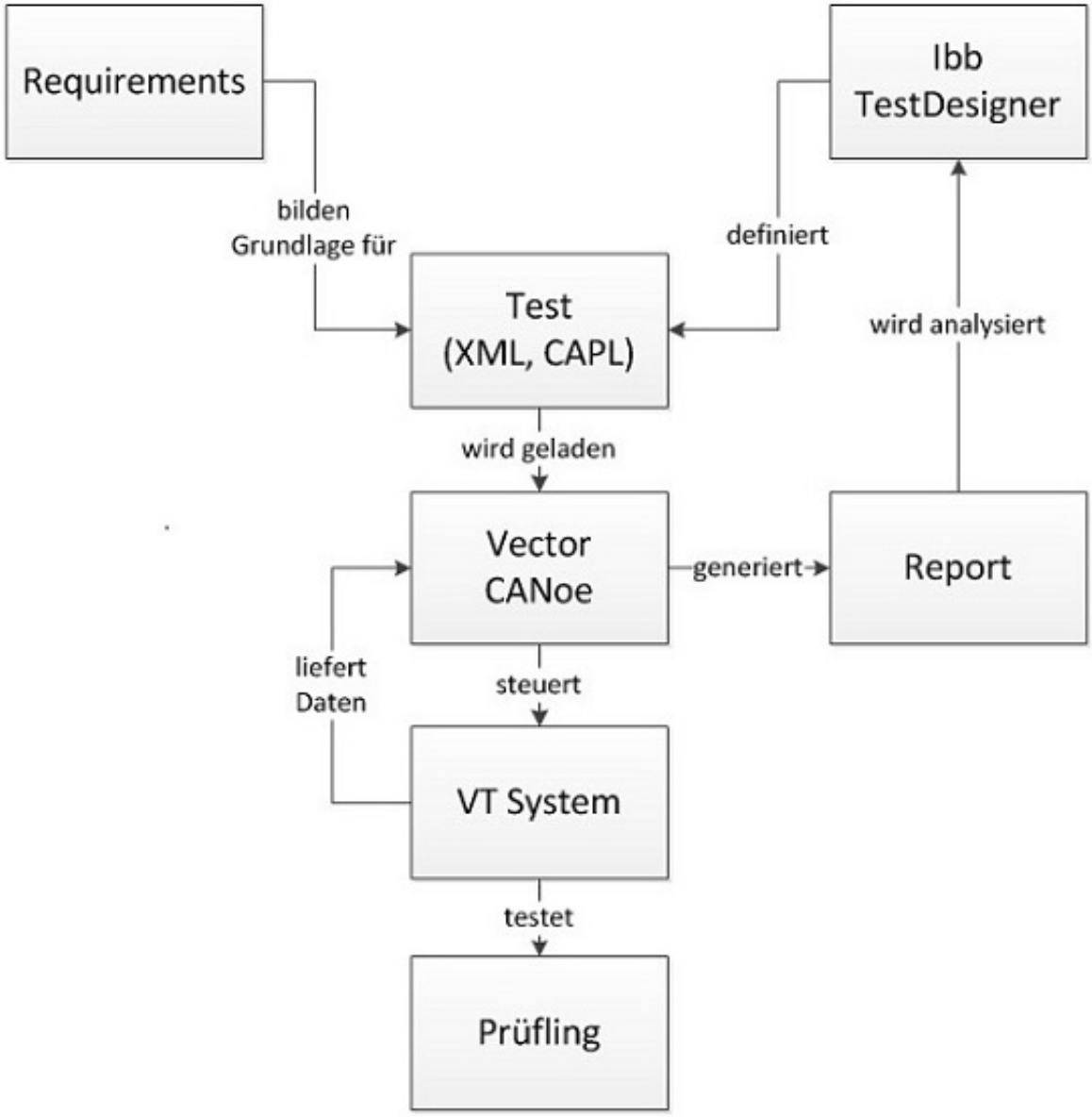

Abbildung 2.1: Prozesskette von Hardware Tests mittels *ibb TestDesigner* und *CANoe*

3 Analyse des ibb TestDesigners

3.1 Allgemeines

Der *ibb TestDesigner* ist in der Programmiersprache *Microsoft Visual Basic 8.0* geschrieben und die Weiterentwicklung im Rahmen dieser Arbeit erfolgt mit der Entwicklungsumgebung *Microsoft Visual Studio 2010*. Das programmeigene Arbeitsformat ist *Test Feature Editor (.tfe)*. Jedes Test-Projekt wird in einer .tfe-Datei gespeichert.
Der *ibb TestDesigner* ist ein Programm, das in erster Linie Dateien mit Testabläufen für *CANoe* generiert. Die von *CANoe* akzeptierten Formate sind hierbei *XML*-Dateien (.xml) und *CAPL*-Dateien (.can). Aufgrund der Vielseitigkeit und Strukturiertheit von *XML* spielt dieses Format auch eine zentrale Rolle im *ibb TestDesigner*. Das Test Setup selbst, die Methodenbibliotheken und Reports basieren auf *XML*.
Ein weiterer Dateityp, der im *ibb TestDesigner* zum Einsatz kommt, sind *Microsoft Excel*-Dateien (.xlsx). Der Grund dafür liegt in der Requirement-basierten Arbeitsweise mit dem *ibb TestDesigner*. Wenn ein System getestet werden soll, werden gewisse Anforderungen formuliert, die durch einen Test überprüft werden sollen. Diese Requirements können sowohl in Textform als auch in einer Tabelle formuliert sein. Wenn die Beschreibung der Requirements in einer *Excel*-Datei vorliegt, so kann diese in den *ibb TestDesigner* importiert werden. Jedes Requirement verfügt in der Regel über eine kurze, aussagekräftige Beschreibung sowie eine einmalige ID. Der *ibb TestDesigner* unterstützt den Testersteller bei der Arbeit, indem er anzeigt, welche Requirements durch den Test bereits abgedeckt sind und welche noch fehlen. Ein Beispiel für die Darstellung von Requirements kann wie in Tabelle 3.1 dargestellt, aussehen.

Tabelle 3.1: Beispielhafte Darstellung von tabellarisch verfassten Requirements

	Requirement ID	**Anforderung**
1	sup	Spannungsversorgung
2	sup_01	Nach Einschalten soll das System funktionsfähig sein
3	sup_02	Bei Spannungsänderungen um 2V soll das System stabil laufen
4	sup_03	Das System soll nach Spannungseinbruch wieder normal laufen
5	led	Funktion der LED
6	led_01	Der Takt der LED soll konstant sein
7	led_02	Bei Spannung unter 5V soll die LED ausgehen

Die Gliederung eines Tests erfolgt im *ibb TestDesigner* in Testgroups und Testcases, zusätzlich noch optional in Varianten. Eine Testgroup besteht üblicherweise aus mehreren Testcases, wobei diese nicht weiter aufgeteilt werden können. Zum Überprüfen eines Requirements wird eine Testgroup

mit mehreren einzelnen Testcases erzeugt. Außerdem können verschiedene Varianten in mehreren Durchgängen ausgeführt werden, in denen Parameter variieren.

Der *ibb TestDesigner* wurde im Laufe der Jahre ständig weiterentwickelt, um sich neuen Aufgabenbereichen anzupassen. Den Kern der Anwendung bilden *Ports*, *Devices* und *Methoden*.

3.2 Ports

Die Ports im *ibb TestDesigner* sind Kanäle, die gemessen oder manipuliert werden können. Bei einem Testablauf in *CANoe* sind das:

- Signale (CAN, LIN,...)
- Umgebungsvariablen
- Systemvariablen
- Nachrichten
- Kanäle von Hardware Modulen wie *VT System*

Mehrere gleichartige Ports werden in einem Device zusammengefasst. Eine Methode, die z.B. einen Wert setzt oder ausliest, ist immer einem bestimmten Port zugeordnet. Für jeden Port gibt es in der Tabelle mit dem Testablauf eine eigene Spalte. Die Definition der einzelnen Ports erfolgt entweder manuell im *ibb TestDesigner*, oder automatisch in *CANoe*. Oftmals genügt es jedoch, die von *CANoe* erzeugte Beschreibung zu importieren. Es sollten allerdings keine Ports erzeugt werden, die in *CANoe* nicht existieren, da *CANoe* die Hardware testet und ihr angepasst werden muss. Wenn *CANoe* also z.B. ein Signal auswerten soll, das es gar nicht gibt, so kann dies zu Fehlern im Test führen.
Jedes Steuergerät hat individuelle Schnittstellen und kein Testaufbau gleicht dem anderen. Das Dateiformat, in dem der *CANoe* zugehörige Datenbankeditor *CANdb++* die Ports beschreibt, ist .dbc. Solche Dateien können vom *ibb TestDesigner* eingelesen und die Ports dementsprechend erzeugt werden. Einige Ports für VT Module und eigens von *ibb* erzeugte Variablen sind bereits in den Template Files für den *ibb TestDesigner* enthalten. Somit können in nur wenigen Schritten sämtliche für einen Test benötigten Ports definiert werden.

3.3 Devices

Ein Device besteht aus mehreren Ports, die strukturell gleichartig sind. Jedem Device werden meist eine, manchmal auch mehrere Klassen von Methoden zugewiesen. Auf dem Port eines Device sollten nur Methoden ausgeführt werden, deren Klasse dem Device zugehörig ist. Oft ist dies eine eins-zu-eins Zuweisung, es gibt z.B. ein Device „Systemvariablen“ und die zugehörige Klasse der ausführbaren Methoden ist ebenfalls „Systemvariablen“. Die Gliederung der Devices in die oben genannten Kategorien ist deshalb sinnvoll, weil eine Methode auf verschiedenen Ports eines Device gleich ausgeführt werden kann. Für die Methode ist es irrelevant, ob sie auf dem Port Systemvariable A oder Systemvariable B

angewendet wird. Wichtig ist nur, dass eine Methode für das Device der Systemvariablen nicht ohne weiteres z.B. auf einem Device für *CAN* Signale ausgeführt werden kann.

3.4 Methoden

Im Mittelpunkt der Anwendung stehen die Methoden, aus denen ein Test modular aufgebaut wird.

Abb. 3.1 zeigt den Methoden Editor, mit dem die Methoden des *ibb TestDesigners* verwaltet werden.

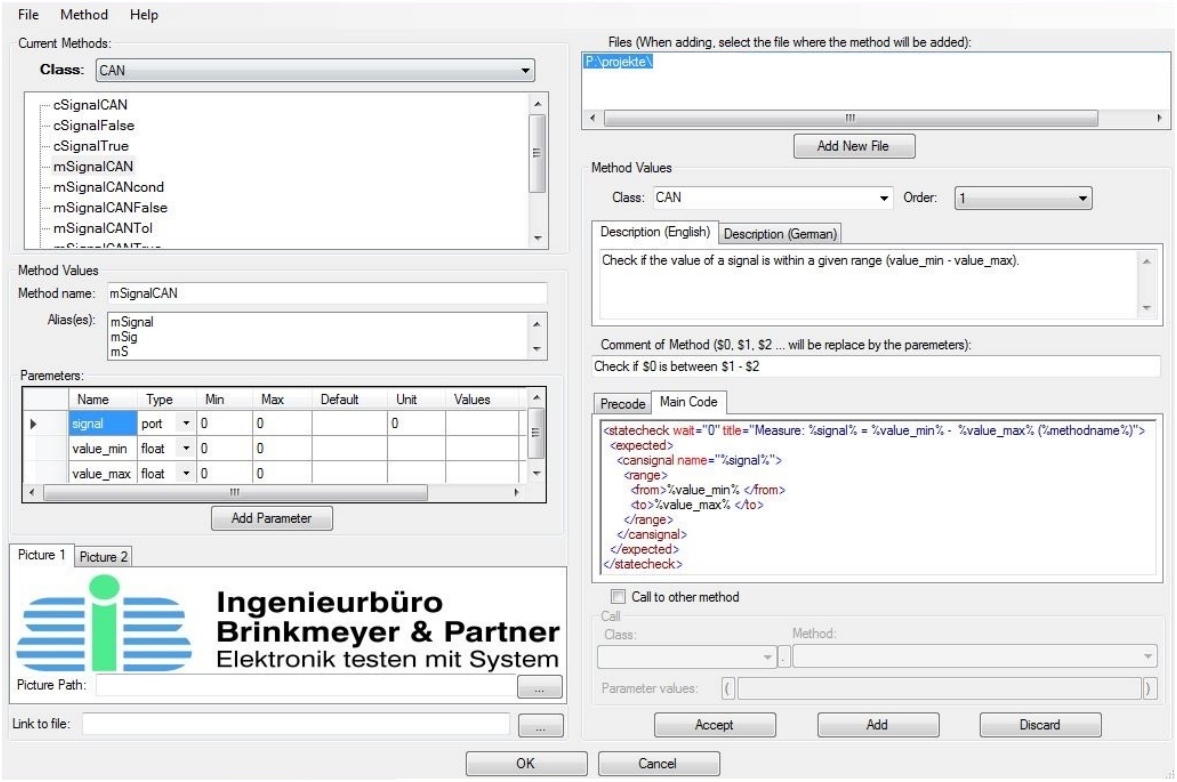

Abbildung 3.1: Graphische Oberfläche des Methoden Editors im *ibb TestDesigner*

Die Methoden werden in einer Bibliothek gespeichert und verwaltet, die auf *XML* basiert. Sämtliche Methoden können im *ibb TestDesigner* angesehen und modifiziert werden. Zusätzlich können auch neue Methoden erstellt werden. Sie bestehen aus:

- **Name und Alias**
  Eine Methode muss einen Namen haben, der sie klar identifiziert. Idealerweise sollte ein aussagekräftiger Name gewählt werden. Optional können auch Aliasse vergeben werden, die die Lesbarkeit bei großen Tests erleichtern.
- **Parameter**
  Für jede Methode können beliebig viele Parameter bestimmt werden. Als mögliche Typen stehen

Port, *Boolean*, *Integer*, *Float* und *String* zur Verfügung. Zusätzlich können auch Beschränkungen, Default-Werte und Einheiten festgelegt werden.

- **Main Code**
  Der Main Code ist das, was beim Generieren eines Tests in die Ausgabedatei geschrieben wird. Da das aktuelle Ausgabeformat *XML* ist, ist der Main Code ebenfalls in *XML* verfasst. Beim Erstellen der Ausgabedatei werden die Methoden und ihre Parameter durch den Main Code ersetzt, der für die jeweilige Methode hinterlegt ist.
- **Klasse**
  Jede Methode benötigt eine Klasse, der sie zugehörig ist. Somit kann die Methode von allen Ports eines Device aufgerufen werden, wenn dem entsprechenden Device diese Klasse zugewiesen wurde.
- **Bilder**
  Um das Verständnis des Anwenders für die Methoden zu verbessern, können einer Methode Bilder hinzugefügt werden.
- **Bibliothek**
  Jede Methode ist in einer Bibliothek gespeichert. Die von *ibb* angelegte *IbbMethodLib* beinhaltet alle wichtigen *XML* Methoden, die zum Testen von Hardware mittels *CANoe* notwendig sind. Bei Bedarf können diese Methoden jedoch auch modifiziert werden, oder es kann eine neue Bibliothek mit eigenen Methoden erstellt und eingebunden werden. In jedes Test Setup können beliebig viele Bibliotheken geladen und deren Methoden benutzt werden.
- **Beschreibung**
  Obwohl die Namen der Methoden möglichst vielsagend gewählt werden sollen, ist es manchmal unvermeidbar, die Funktionsweise einer Methode mittels einer Beschreibung näher zu erläutern. Auch die einzelnen Parameter, die beim Methodenaufruf eingesetzt werden müssen, können näher beschrieben werden.
- **Kommentar**
  Eine Methode hat außerdem einen Kommentar, der standardmäßig als Titel für den Testcase gesetzt wird, um beim schrittweisen Erstellen eines Tests den Überblick zu behalten. Dieser Kommentar kann auch Platzhalter beinhalten, die beim Festlegen der Parameter durch ihren tatsächlichen Wert ersetzt werden.

3.5 Testablauf

In Abbildung 3.2 ist der Hauptbildschirm der Anwendung zu sehen.

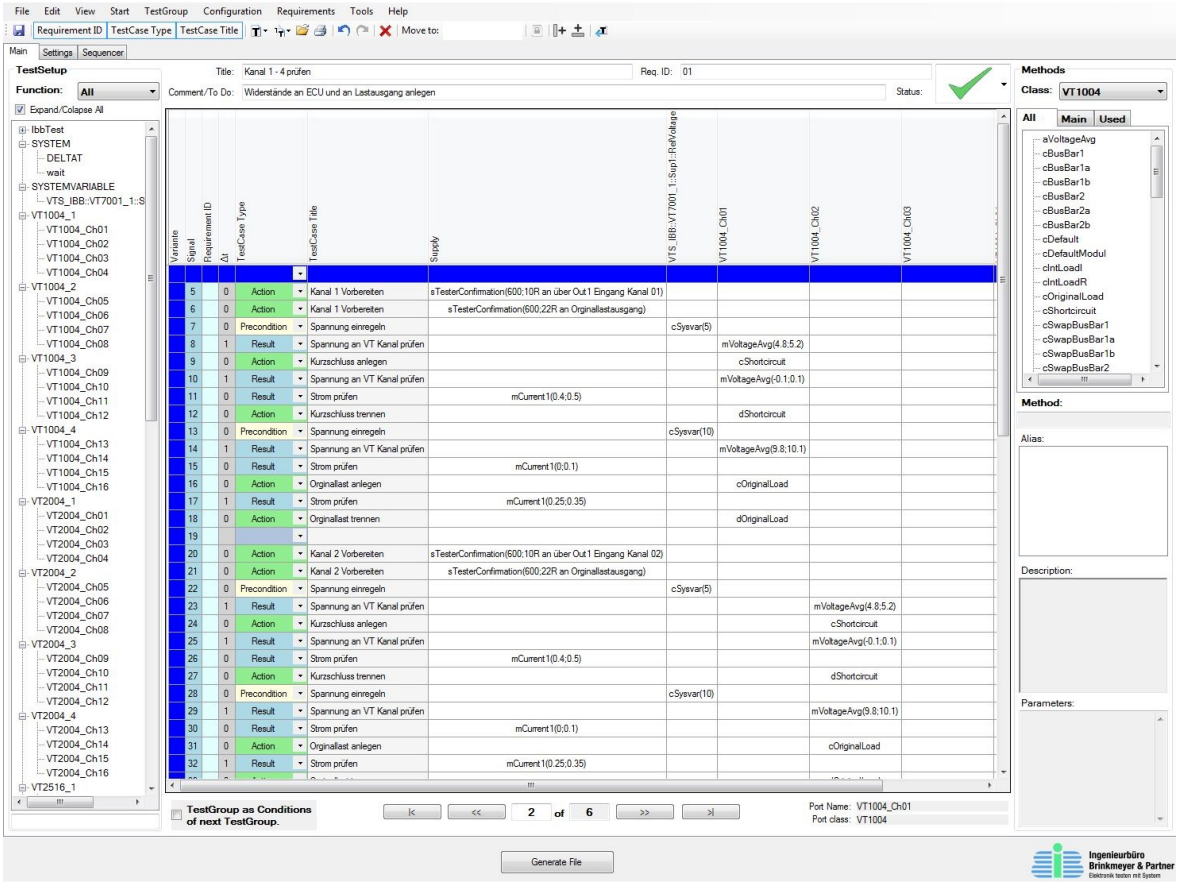

Abbildung 3.2: Hauptbildschirm des *ibb TestDesigners*

Links im Bild ist die Liste mit den verschiedenen Devices des jeweiligen Projekts zu sehen. Zusätzlich enthält diese Liste die einzelnen Ports der Devices. Auf der rechten Seite stehen die Methoden einer jeweiligen Klasse. In der Mitte ist die Tabelle zu sehen, in der ein Anwender des *ibb TestDesigners* einen Test aufbaut. Jede Zeile entspricht einem Testcase und jede Tabelle einer Testgroup. Die Spalten der Tabelle setzen sich von links nach rechts wie folgt zusammen:

- **Variante**
  In dieser Spalte kann eine Zugehörigkeit zu einer oder mehrerer Varianten angegeben werden. Je nach dem wird diese Zeile dann nur in bestimmten Varianten berücksichtigt, ansonsten wird sie ignoriert. Durch eintragen eines Rautensymbols (#) können einzelne Testcases oder ganze Testgroups auskommentiert werden.

- **Signal**
  Jeder Testcase hat eine Nummer, die ihn eindeutig identifiziert. Diese Nummern werden aufsteigend vom ersten Testcase der ersten Testgroup bis zum letzten Testcase der letzten Testgroup vergeben. Wenn der erzeugte Test in *CANoe* ausgeführt wird, ermöglichen die Nummern eine fehlerfreie Zuordnung.

- **Requirement ID**
  Wenn eine *Excel*-Datei mit Requirements in den *ibb TestDesigner* geladen wurde, wird automatisch für jedes Requirement eine Testgroup erstellt und mit der entsprechenden Requirement ID versehen. Ansonsten kann die Requirement ID manuell eingetragen werden.
- **Δt**
  In dieser Spalte wird angegeben, wie lange gewartet werden soll, bevor ein Testcase ausgeführt werden soll. Wird die Spalte leer gelassen, fährt *CANoe* nach Beendigung eines Testcase umgehend mit dem nächsten fort. Um in diesen Vorgang einzugreifen, kann eine Wartezeit eingetragen werden. Wenn z.B. geprüft werden soll, ob nach spätestens 5 Sekunden ein Relais geschaltet hat, muss ein Mechanismus zum expliziten Warten vorhanden sein. Dieser ist mit Δt gegeben.
- **TestCase Type**
  Jedem Testcase kann ein Typ zugewiesen werden. Mögliche Typen sind: *Precondition*, *Action*, *Result*, *Condition*, *Postcondition* und *Remark*. Durch diese optionale Festlegung kann die Struktur eines Tests feiner beschrieben und die Übersichtlichkeit gesteigert werden.
- **TestCase Title**
  Diese Spalte weist einem Testcase einen Titel zu, entweder automatisch über die Beschreibung der Methode oder durch manuelle Eingabe eines Titels.
- **Ports**
  Für jeden Port, der in einer Testgroup benötigt wird, wird der Tabelle eine Spalte hinzugefügt. Dies erfolgt per Drag & Drop aus der linken Liste der Ports, die nach ihren jeweiligen Devices sortiert sind. In jedem Testcase kann ein Methodenaufruf pro Port getätigt werden. Wenn eine Zelle eines Ports angewählt wird, stehen in der rechten Liste die geeigneten Methoden, also die Methoden aller Klassen die dem Device zugeordnet wurden. Durch Doppelklicken auf eine Methode erscheint ein Dialogfeld, in dem die nötigen Parameter eingetragen werden müssen. Anschließend befindet sich die Methode in der ausgewählten Zelle.
- **Bemerkungen**
  Ganz rechts in der Tabelle befindet sich eine letzte Spalte, in die Bemerkungen zu einem Testcase geschrieben werden können.

Durch die einfache und übersichtliche Bedienung kann nach und nach ein beliebig komplexer Test mit einer Vielzahl von Testgroups erstellt werden. Der Anwender muss keine Kenntnis über den tatsächlichen Code haben, da er auf der Abstraktionsebene der graphischen Oberfläche arbeitet.

3.6 Test Generierung

Wenn der Test vollständig aufgebaut wurde, kann mit einem Klick auf den „Generate File“ Knopf eine Ausgabedatei erzeugt werden. Dies ist die *XML*-Datei, die von *CANoe* eingelesen und ausgeführt werden kann. Sie beinhaltet einen vom *ibb TestDesigner* automatisch generierten Rahmen mit notwendigen Voreinstellungen und der Strukturierung in die einzelnen Testgroups. Diese Testgroups werden dann mittels Iteration durch die einzelnen Zellen der tabellarischen Testcases mit den nötigen

Informationen und dem Main Code befüllt. Alle im Main Code auftretenden formalen Parameter werden an dieser Stelle durch den angegeben Wert der tatsächlichen Parameter ersetzt. Am Ende entsteht eine Datei, die alle von *CANoe* gestellten Anforderungen an einen in *XML* verfassten Test erfüllt. Der Test kann ohne weitere Modifizierung auf einer simulierten Umgebung oder einem tatsächlichen Hardware Aufbau durchgeführt und ausgewertet werden.

3.7 Weitere Funktionen des ibb TestDesigners

Der *ibb TestDesigner* verfügt über eine Vielzahl weiterer Funktionen, die für diese Arbeit jedoch nebensächlich sind und nur kurz erläutert werden. Die in der Praxis wichtigsten dieser Funktionalitäten sind:

- **Sequencer**
  Diese Funktion erlaubt eine Generierung von mehreren Testdateien, die im .tfe Format spezifiziert sind, gleichzeitig. Dazu muss ein Ordner angegeben werden, in dem sich die Dateien befinden. Durch das Setzen von Haken können mehrere ausgewählte Tests aus verschiedenen Dateien generiert werden.
- **Report Analyzer**
  CANoe erzeugt nach dem Durchlauf eines jeden Tests einen Report. Darin enthalten ist eine Aufschlüsselung sämtlicher Testcases mit Zeitstempel und Ergebnis. Der *Report Analyzer* ermöglicht eine Überprüfung der Ergebnisse in Bezug auf die Requirements, die zu Beginn einer Testerstellung aus einer *Excel*-Datei eingelesen werden können.
- **Report Comparator**
  Nach der Analyse durch den *Report Analyzer* kann ein detaillierter Vergleich der Reports vorgenommen werden. Sowohl ein Vergleich der Reports selbst, als auch ein Vergleich einzelner Ports zu bestimmten Zeitpunkten ist möglich. Unterstützt wird dies durch eine dynamische Darstellung der ausgewählten Ports in einem Graphen.

3.8 Vergleich mit ähnlichen Produkten

Ein Vergleich zu anderen Produkten, die eine ähnliche Aufgabe erfüllen, gestaltet sich als schwierig. Das liegt daran, dass *CANoe* als Testumgebung eine zentrale Rolle bei *ibb* selbst aber auch bei dessen Kunden spielt. Es gibt zwar andere Software für Testsysteme, aber durch die Nutzung von *VT Modulen*, die wie *CANoe* selbst von *Vector Informatik* stammen, bietet sich *CANoe* als Testumgebung an. Außerdem entwickelt *ibb* selbst ebenfalls Module, die mit dem *VT System* kompatibel sind. Aus diesen Gründen ist jede andere Testumgebung hinfällig.
Ein vergleichbares Produkt müsste also für *CANoe* ausführbaren Code erzeugen können. Als *ibb* mit der Entwicklung des *ibb TestDesigners* begann, war lange Zeit kein anderes Programm verfügbar, das diese Aufgabe erfüllte. Mittlerweile hat auch *Vector Informatik* die Notwendigkeit einer komfortablen Testerstellung erkannt und eine eigene Software entwickelt, das *vTESTstudio*.

vTESTstudio

Die Software *vTESTstudio* wurde von *Vector Informatik* im Jahre 2013 veröffentlicht [Vec15c]. Da es vom selben Unternehmen entwickelt wurde wie *CANoe*, ist die Verknüpfung dieser beiden Programme miteinander sehr eng. Mit *vTESTstudio* können Tests in den Sprachen *CAPL* und *C#* erstellt werden. Außerdem gibt es einen „Test Table Editor", in dem in einer Baumstruktur die Testcases auf der Abstraktionsebene einer graphischen Oberfläche erstellt werden können. Zusätzlich kann der Testablauf als Flussdiagramm angezeigt oder selbst definiert werden [Vec15b].
Auf den ersten Blick bietet *vTESTstudio* eine Reihe von Vorzügen gegenüber dem *ibb TestDesigner*. In einigen wesentlichen Punkten zeigt sich jedoch, warum *ibb* am *ibb TestDesigner* festhält:

- **Übersichtlichkeit**
  Obwohl es sich dem Namen nach beim „Test Table Editor" im *vTESTstudio* um eine Tabelle handelt, ist es tatsächlich eine als Liste dargestellte Baumstruktur. Einzelne Testgroups sowie Testcases sind Knoten in einer Liste, die auf- und zugeklappt werden können. Die Blätter der Knoten sind die einzelnen Teststeps. Bei größeren Tests wächst diese Liste sehr weit nach unten und wird dadurch schnell unübersichtlich.
  Beim *ibb TestDesigner* erfolgt die Darstellung in einer zweidimensionalen Tabelle mit einer Seite pro Testgroup. Bei zunehmender Anzahl an Testcases wächst die Tabelle vertikal, bei einer Zunahme von Teststeps, also Ports, horizontal. Durch die einfach verständliche Ansicht sind nicht nur mehr Testcases auf einmal zu sehen, sondern für alle Testcases sämtliche Einzeloperationen, die durchgeführt werden.

- **Einfachheit**
  Innerhalb weniger Minuten kann ein Anwender im *ibb TestDesigner* für ein neues Projekt mit der Testerstellung beginnen. Hierzu müssen nur bereits fertige Dateien importiert werden. Die Namen der zu benutzenden Methoden sind aussagekräftig und die Ausführung auf einem Port ermöglicht ein leichtes Verständnis darüber, was genau in einem Testschritt gemacht wird.
  vTESTstudio ist komplexer und ermöglicht durch seine Nähe zu *CANoe* viele verschiedene Funktionen, was aber zu Lasten von Übersichtlichkeit und Verständlichkeit geht. Viele der Funktionalitäten werden für normale Tests gar nicht gebraucht. Der Funktionsumfang ist größer, aber es ist daher auch komplizierter für den Anwender einen Test zu erstellen.

- **Modifizierbarkeit**
  Bei *ibb* wird viel Wert auf Flexibilität und Anpassbarkeit an unvorhergesehene Szenarien gelegt. Deshalb ist ein eigenes Tool, das bei Bedarf abgeändert werden kann, sehr wichtig für das Unternehmen. Falls nötig, können dem *ibb TestDesigner* einfach neue Methoden hinzugefügt werden.
  Wenn ein externes Programm wie *vTESTstudio* genutzt wird, so geht dies auf Kosten der Flexibilität. Das Unternehmen muss sich an die vorgegebenen Muster halten und kann das Programm selbst nicht modifizieren. Da der *ibb TestDesigner* über viele Jahre hinweg zielführend weiterentwickelt wurde und sich bei *ibb* und dessen Kunden bewährt hat, wird weiterhin an diesem Konzept und der klaren Struktur festgehalten werden.

Alles in allem ist *vTESTstudio* ein mächtiges und vielseitiges Programm, doch aufgrund der oben genannten Gründe bringt der *ibb TestDesigner* trotzdem einen Mehrwert für *ibb*.

4 Entwurf & Implementierung

4.1 Vergleich von Extensible Markup Language und CAN Access Programming Language

4.1.1 Gegenüberstellung der Funktionsweise

Die zwei Formate für vordefinierte Tests, die von *CANoe* akzeptiert werden, sind *XML*-Dateien und *CAPL* Code. Diese beiden Formate unterscheiden sich jedoch in einigen wesentlichen Punkten voneinander. Tabelle 4.1 stellt die wichtigsten Unterschiede gegenüber [Kra09].

Tabelle 4.1: Gegenüberstellung von Testcases in *XML* und *CAPL* nach [Kra09]

	Test in *XML*	**Test in *CAPL***
Ausführung eines Testcase	Jeder Testcase kann maximal einmal ausgeführt werden	Jeder Testcase kann beliebig oft ausgeführt werden
Reihenfolge der Ausführung	Statisch durch die Struktur der *XML*-Datei vorgegeben	In der MainTest Funktion dynamisch festlegbar
Kontrolle über die Ausführung	In der graphischen Oberfläche von *CANoe* kann für jeden Testcase ein Haken gesetzt werden, ob er ausgeführt werden soll, oder nicht	In der MainTest Funktion programmiert
Testgroups	Statisch in der *XML*-Datei festgelegt, eins-zu-eins Zuweisung von Testcases zu Testgroups	Dynamisch in der MainTest Funktion definiert, eine Zuweisung eines Testcase zu einer oder mehrerer Testgroups erfolgt während der Laufzeit
Testcase Definition	Durch *XML* Struktur Muster vorgegeben	Frei programmiert in *CAPL*
Test Report	Enthält Informationen über alle Testcases, die in der *XML*-Datei definiert wurden, aber nicht zwingend ausgeführt wurden	Enthält nur Informationen über tatsächlich ausgeführte Testcases

Man sieht anhand von Tabelle 4.1, dass *CAPL* eine flexiblere Art der Testdefinition ermöglicht. *XML* hat eine feste Struktur und beschreibt einen statischen Ablauf. Erweiterungen an einem Test in *XML* sind nachträglich nur schwer durchzuführen. Im Gegenzug dazu ist es einfacher zu verstehen und es sind keine Kenntnisse über Programmiersprachen nötig.

4.1.2 Vergleich am Code

Um den Unterschied zwischen den beiden Arten der Testdefinition besser zu verstehen, hilft die Gegenüberstellung eines einfachen Testcase, der einmal in *XML* und einmal in *CAPL* verfasst ist. Listing 4.1 zeigt einen Testcase in der Sprache *CAPL*.

Listing 4.1: Beispiel eines Testcase in *CAPL*

```
testcase testcase_1()
{
      testCaseTitle("1", "Measure ibb::Voltage = 12 - 13");
      testresult = checkSignalInRange(sysvar::ibb::Voltage, 12, 13);
      if (testresult == 1) TestStepPass("System Variable ibb::Voltage is between 12 and 13");
      else TestStepFail("Value of System Variable ibb::Voltage is not in the allowed range");
}
```

Listing 4.2 zeigt einen funktional identischen Testcase in *XML*.

Listing 4.2: Beispiel eines Testcase in *XML*

```
<testcase title="testcase_1">
      <statecheck wait="0" title="Measure: ibb::Voltage = 12 - 13 (mSysvar)">
            <expected>
                  <sysvar name="Voltage" namespace="ibb">
                        <range>
                              <from>12</from>
                              <to>13</to>
                        </range>
                  </sysvar>
            </expected>
      </statecheck>
</testcase>
```

Der *XML* Code wird genau einmal so ausgeführt, wie er definiert ist. Der *CAPL* Code kann in der MainTest Funktion beliebig oft aufgerufen werden.

4.2 Gliederung eines CAN Access Programming Language Testmoduls

Abb. 4.1 zeigt die Gliederung eines Tests, der in *CAPL* verfasst ist.

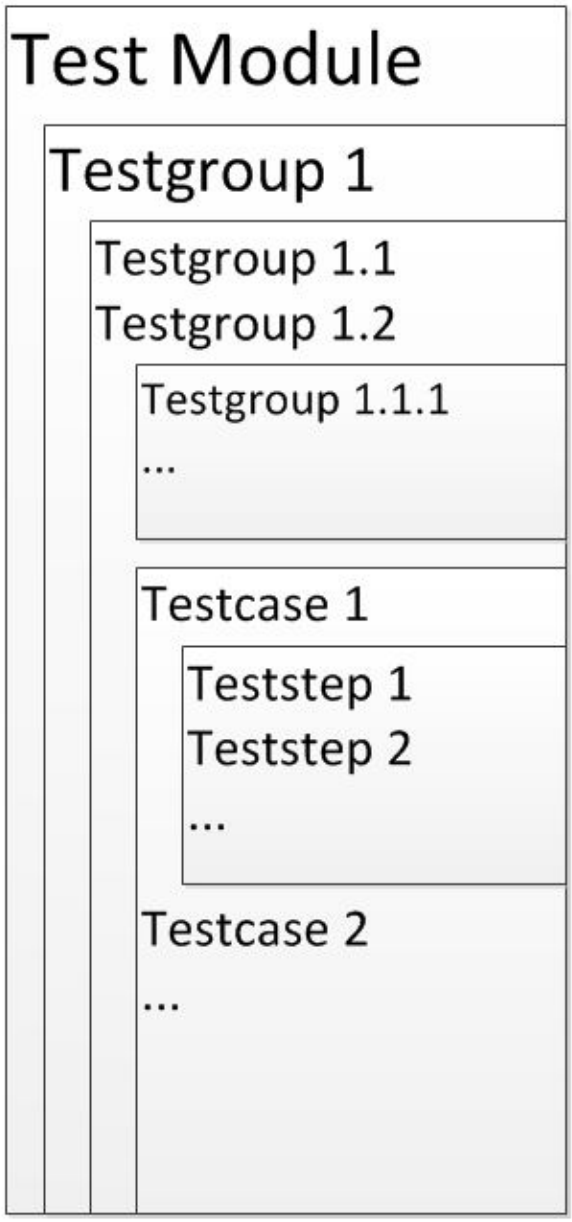

Abbildung 4.1: Gliederung eines Testmoduls in *CAPL*

Ein Test bzw. Test Module besteht aus einer oder mehreren Testgroups. Diese können auch ineinander verschachtelt sein, so dass eine Testgroup selbst aus mehreren Testgroups besteht. Innerhalb einer Testgroup gibt es einen oder mehrere Testcases. In diesen Testcases finden die eigentlichen Tests statt. Ein Testcase kann wiederum aus mehreren Teststeps bestehen, die das kleinste Element innerhalb eines Tests darstellen. Ein Teststep kann nicht noch feingranularer aufgeteilt werden.
Ein Test gilt nur als bestanden, wenn alle seine Untereinheiten, also seine Testgroups, als bestanden gelten. Für diese gilt dasselbe: eine Testgroup ist fehlerfrei durchlaufen, wenn alle ihr zugeordneten Testcases korrekt waren. Ebenso verhält es sich für die Teststeps.

4.3 Anforderungen an die Implementierung

An die Erweiterung des *ibb TestDesigners* werden gewisse Anforderungen gestellt, die einzuhalten sind. Diese lassen sich aufteilen in funktionale und nichtfunktionale Anforderungen.
Funktionale Anforderungen beschreiben Eigenschaften bzw. ein Verhalten, das ein Produkt unbedingt zu erfüllen hat. Dies beinhaltet die fehlerfreie Funktion des erwarteten und erwünschten Verhaltens [RR06].
Nichtfunktionale Anforderungen werden formuliert, um ein Produkt möglichst befriedigend zu gestalten. Sie sind nicht zwingend für die korrekte Funktion eines Produkts nötig, doch je mehr nichtfunktionale Anforderungen erfüllt werden, desto besser ist es. Unter nichtfunktionale Anforderungen an eine Software fallen Aspekte wie Bedienbarkeit, Geschwindigkeit oder Skalierbarkeit [RR06].
In Rücksprache mit dem Betreuer in der Firma, Herrn Dipl.-Ing. (FH) Peter Heidenwag, wurden einige Anforderungen an das Ergebnis dieser Arbeit formuliert.

4.3.1 Funktionale Anforderungen

Die Anforderungen, die unbedingt von einer Erweiterung des *ibb TestDesigners* zur Generierung von *CAPL* Code erfüllt werden müssen, lauten wie folgt:

- Das bisherige Programm muss weiterhin wie bereits zuvor funktionstüchtig sein. Eine Einschränkung, Behinderung oder Beschädigung der Generierung von Tests in *XML* darf nicht auftreten.
- Das schon bestehende Konzept mit Devices, Ports und Methoden soll genutzt werden.
- Eine Erweiterung des grundlegenden Konzepts darf nicht vorgenommen werden.
- Die Ausgabedatei muss im Format *.can* vorliegen und darf ausschließlich Funktionen beinhalten, die von *CAPL* unterstützt werden.
- Nach der Generierung muss eine Datei vorliegen, die ohne weitere Modifikation als Testmodul in *CANoe* ausgeführt werden kann.
- Es muss eine Methodenbibliothek erzeugt werden, die losgelöst von der schon existierenden Bibliothek für *XML* Methoden ist.
- Es müssen mehrere Durchläufe und mehrere Varianten ein und des selben Tests automatisch erzeugt werden können.
- Es sollen Funktionalitäten im Testablauf möglich sein, die zuvor mit *XML* allein nicht zu erreichen waren.

4.3.2 Nichtfunktionale Anforderungen

Anforderungen, die nicht zwingend erfüllt werden müssen, deren Einhaltung aber wünschenswert ist, sind:

- Änderbarkeit
  Der Code soll möglichst allgemein gehalten sein, um etwaige Änderungen vornehmen zu können.
- Performanz
  Die Performanz darf nicht wesentlich schlechter sein als bei der bisherigen Generierung von *XML*.
- Handhabung
  Die Erstellung eines Tests soll weiterhin genau so einfach sein wie bisher. Ein Anstieg der Komplexität für den Anwender muss vermieden werden.
- Verständlichkeit
  Einem Anwender, der die bisherige Testerstellung mit *ibb TestDesigner* beherrscht, soll auch die *CAPL* Test Generierung ohne Weiteres verständlich sein.
- Logfile
  Eine bereits existierende Logfile, die wichtige Aktivitäten des Programms protokolliert, soll weiterhin gepflegt werden.

4.3.3 Weitere Anforderungen

Als Anforderungen, die sich weniger dem Produkt selbst sondern vielmehr dem Arbeitsprozess zuordnen lassen, wurden die folgenden Aspekte identifiziert:

- Während der Entwicklung der Erweiterung soll regelmäßig Rücksprache mit dem Betreuer in der Firma gehalten werden, um die angestrebten Ziele nicht aus dem Blick zu verlieren.
- Es soll so früh wie möglich eine ausführbare Erweiterung erstellt werden, die dann Schritt für Schritt mit mehr Funktionalität ausgestattet werden kann.
- Es muss eine Evaluation der fertiggestellten Erweiterung an einem Testsystem durchgeführt werden, um die fehlerfreie Funktionsweise zu demonstrieren. Dazu soll ein umfangreicher Test in *CAPL* erzeugt und ausgeführt werden.

4.4 Agile Softwareentwicklung

Die im letzten Abschnitt formulierten Anforderungen decken sich stark mit dem Vorgehen der Agilen Softwareentwicklung, genauer gesagt dem Agilen Manifest [BBB+01]. Das Agile Manifest wurde im Jahre 2001 von führenden Personen im Bereich der Softwareentwicklung formuliert. Die zentralen Punkte lauten im Original wie folgt:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Die Entwicklung der Software Erweiterung im Rahmen dieser Arbeit orientiert sich sehr stark an diesen vier Prinzipien. Da die Softwareentwicklung von einer Einzelperson vorgenommen wird, ist es wichtig, regelmäßig Rücksprache mit dem Auftraggeber zu halten, was in diesem Fall das Unternehmen ist. Auf einzelne Menschen und Interaktionen in Form von immer wieder stattfindenden Treffen und Rücksprachen wird in dieser Arbeit viel Wert gelegt. Dadurch entstehen auch öfter neue Sichtweisen und neue Ideen bzw. Modifikationen, die umgesetzt werden können. Natürlich gibt es zu Beginn einen groben Plan, wie vorgegangen werden soll, doch durch die Flexibilität bei der Entwicklung wurden immer wieder Veränderungen und Ergänzungen vorgenommen.
Das Resultat der Arbeit soll in erste Linie eine funktionierende Software sein, die Dokumentation der selbigen ist dabei zunächst einmal eher nachrangig.

4.5 Konzept

Das Ziel der Testerstellung mittels des *ibb TestDesigners* ist, dass ein Anwender ohne detaillierte Kenntnis der nötigen Sprache einen Test erstellen kann. Dies wird dadurch ermöglicht, dass der Nutzer den Test auf einer graphischen Oberfläche zusammenbaut. Diese graphische Oberfläche muss sich nicht unterscheiden, wenn verschiedene Ausgabeformate generiert werden sollen. Auch die Datenhaltung innerhalb der Anwendung muss nicht für die Erweiterung um das *CAPL* Format angepasst werden.
Die Generierung kann gedanklich in zwei Bereiche unterteilt werden, einen Rahmen und einen dynamischen Teil. Der Rahmen enthält den Testablauf und die Gliederung, er bildet somit die statische Grundstruktur. Der dynamische Teil besteht aus den einzelnen Testcases bzw. Methoden, die bei der Generierung durch den eigentlichen Code ersetzt werden. Dieser Code ist in der Methodenbibliothek als Main Code abgelegt.
Die zentralen Änderungen für die *CAPL* Generierung müssen also im Rahmen der erzeugten Datei und in der Methodenbibliothek liegen, da eine *XML*-Datei eine grundsätzlich andere Struktur als eine *CAPL*-Datei im .can Format hat.

4.5.1 Rahmen

Der Aufbau eines *CAPL* Testmoduls folgt einem gewissen Muster. Jeder funktionierende Testablauf lässt sich gliedern in:

- **Globale Variablen**
  Der erste Abschnitt besteht aus Definitionen von Variablen, Timern, Nachrichten und ähnlichem. Für einen normalen Testablauf sind diese aber nicht zwingend notwendig. Die einzigen Variablen, die global definiert werden sollten, sind ein *Long* mit dem Namen *testresult* und ein

Float namens *delay*. In der Variable *testresult* kann immer das aktuelle Ergebnis eines Vergleichs oder einer Zuweisung gespeichert werden. In *delay* können die Ergebnisse von Zeitmessungen zwischengespeichert werden.
Die globalen Variablen müssen einmal vom *ibb TestDesigner* generiert werden, weshalb dieser Teil dem Rahmen zugeordnet werden kann.

- **MainTest**
  Die Funktion *void MainTest()* entspricht der *main()* Funktion eines *C* Programms. Sie wird beim Start des Tests automatisch aufgerufen. Der gesamte Ablauf des Tests sowie die Gliederung in diverse Testgroups und der Aufruf der einzelnen Testcases erfolgen hier. Jeder Testcase, der später im Code definiert wird, muss hier aufgerufen werden, insofern er Teil des Testablaufs sein soll. Wenn die *MainTest* Funktion ihr Ende erreicht hat, endet auch die Ausführung des Tests.
  Die *MainTest* Funktion an sich muss vom *ibb TestDesigner* einmal erzeugt werden. Sie gehört also ebenfalls zum Rahmen. Die Befüllung der *MainTest* Funktion, also die Gliederung in verschiedene Testgroups und der Aufruf der Testcases, ist allerdings etwas dynamisches, das von der Testdefinition auf der graphischen Oberfläche abhängt. Jeder Testcase im *ibb TestDesigner*, also jede Zeile im Hauptfenster mit dem Testablauf, muss in der *CAPL*-Datei aufgerufen werden.

- **Testcases**
  Jeder Testcase wird wie eine Funktion in *C* definiert. Der Identifier hierfür ist *testcase*. Innerhalb eines Testcase sollte ein Titel vergeben werden und der eigentliche Ablauf des jeweiligen Testcase stattfinden. Am Ende muss das Resultat ein *„pass“* oder ein *„fail“* sein. Ein Testcase kann beliebig komplex sein, es bietet sich jedoch an so wenig Schritte auf einmal wie möglich zu machen.
  Die Testcases können nicht statisch und jedes Mal gleich erzeugt werden. Sie bilden den Kern des Tests und sind im *ibb TestDesigner* in der Methodenbibliothek gespeichert. Jeder Testcase, der dann auf der graphischen Oberfläche des *ibb TestDesigners* aufgerufen wird, muss in der *CAPL* Testdatei definiert sein. Hierbei müssen für die Parameter und Platzhalter die tatsächlichen Werte eingesetzt werden.

4.5.2 Methodenbibliothek

Jede Methode im *ibb TestDesigner* wird in einer Methodenbibliothek gespeichert bzw. aus dieser geladen. Da in der bisherigen Version der Software jedoch die Erzeugung von Tests in *XML* im Mittelpunkt stand, können die existierenden Methoden nicht für die Erzeugung von *CAPL* Code benutzt werden. Jede Methode, die in einem *CAPL* Testmodul verwendet werden soll, muss also geschrieben und in einer entsprechenden Bibliothek gespeichert werden. Hierzu wird eine neue Bibliothek angelegt, die *„ibbMethodenLibCapl.xml“*. Dafür kann der bereits existierende Mechanismus zur dauerhaften Speicherung von Methoden genutzt werden. Alle Informationen über eine Methode werden im *XML*-Format gespeichert und können vom *ibb TestDesigner* eingelesen werden. Zu diesen Informationen gehören Name, Klasse, zweisprachige Beschreibung, Kommentar, Parameter und Main Code. Nach dem Einlesen können die Methoden in einem Testprojekt verwendet oder modifiziert werden.

Das Anlegen der Methodenbibliothek ist ein aufwändiger Vorgang, aber eine einmal angelegte Methode kann anschließend beliebig oft in einer Vielzahl von Tests angewandt werden.

4.5.3 Beispiel eines vollständigen Testmoduls in der CAN Access Programming Language

Zum besseren Verständnis vom Aufbau eines *CAPL* Testmoduls zeigt Listing 4.3 einen kurzen aber vollständigen Test. Weitere Testcases müssen in der *MainTest* Funktion aufgerufen und innerhalb eines Blocks weiter unten in gleicher Weise definiert werden.

Listing 4.3: Funktionsfähiges Beispiel eines Testmoduls in *CAPL*

```
/*@!Encoding:1252*/
Variables
{
      long testresult = -1;
}

void MainTest()
{
      TestModuleTitle("Einfaches Testmodul");

      TestGroupBegin("Einfache Testgroup", "Dies ist eine einfache Testgroup");

            testcase_1();

      TestGroupEnd();
}

testcase testcase_1()
{
      testCaseTitle("1", "Dies ist ein einfacher Testcase");
      TestWaitForTimeout(1000);
      TestStepPass("Erfolgreich 1s gewartet!")
}
```

4.6 Implementierung

4.6.1 Entwicklungsumgebung

Die Entwicklungsumgebung mit der der *ibb TestDesigner* im Rahmen dieser Arbeit erweitert wurde, ist *Microsoft Visual Studio 2010*.
Visual Studio ist eine integrierte Entwicklungsumgebung (IDE), die von der Firma *Microsoft* erstmals im Jahre 1997 veröffentlicht wurde [Kir97]. Es werden eine Vielzahl an Sprachen unterstützt, u.a. *C*, *HTML* oder auch *Visual Basic*, die Sprache die dem *ibb TestDesigner* zugrunde liegt. Seit der Erstveröffentlichung erschienen regelmäßig neue Versionen, die den Funktionsumfang stetig erweiterten.
Visual Studio 2010 wurde am 12. April 2010 veröffentlicht [MSD15].

Es wurde diese Version gewählt, da sie stabil und zuverlässig läuft und alle Bedürfnisse abdeckt, die beim Entwickeln mit der Sprache *Visual Basic 8.0* auftreten.

4.6.2 Durchläufe

Beim Generieren eines *CAPL* Tests ist es möglich, mehrere Durchläufe auszuführen. Dafür gibt es eine eigene Tabelle in den Testeinstellungen, die in Abb. 4.2 dargestellt ist.

Abbildung 4.2: Definition verschiedener Durchläufe

Der Gedanke dahinter ist, dass verschiedene Werte für Parameter eingesetzt werden sollen. Wenn z.B. geprüft werden soll, wie sich ein Steuergerät bei verschiedenen Spannungen verhält, definiert man verschiedene Durchläufe. Vorstellbar wären drei Durchläufe: einer mit normaler Spannung, einer mit reduzierter Spannung und einer mit erhöhter Spannung. An der Stelle im Test, wo die Spannung gesetzt wird, trägt der Nutzer statt einem konkreten Wert eine Variable ein, die er durch Umschließen mit %-Symbolen kennzeichnet. In diesem Fall wäre diese Variable %Usup%. Im Drop-Down-Menu beim Einstellen eines Parameters für eine Methode werden alle Variablen vorgeschlagen.
Der *ibb TestDesigner* erzeugt nun mehrere Durchläufe des selben Tests und in jedem Durchlauf werden ein oder mehrere Variablen durch zuvor festgelegte Werte ersetzt. Es können beliebig viele Durchläufe definiert werden, in denen wiederum mehrere Variablen durch ihren jeweils festgeschriebenen Wert ersetzt werden.

4.6.3 Varianten

Zusätzlich zu mehreren Durchläufen können auch noch verschiedene Varianten definiert werden. Die dafür vorgesehene Tabelle in den Einstellungen ist in Abb. 4.3 zu sehen.

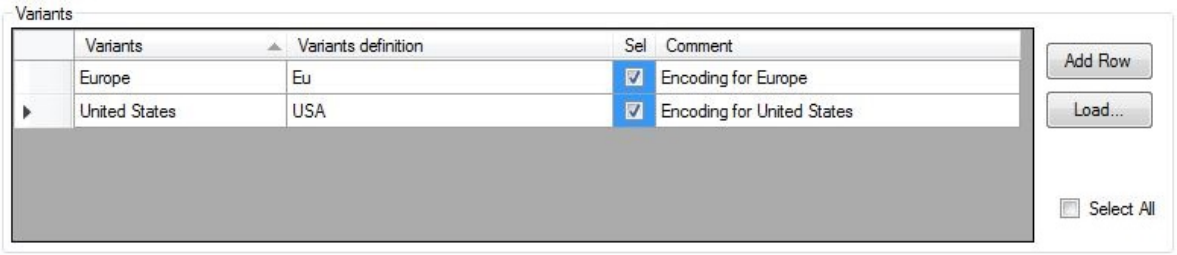

Abbildung 4.3: Definition unterschiedlicher Varianten

Der Sinn für verschiedene Varianten liegt darin, dass manche Testcases vielleicht nur in bestimmten Varianten ausgeführt werden sollen. Nimmt man das Beispiel mit verschiedenen Gebietscodierungen wie in Abb. 4.3, so kann es vorkommen, dass sich ein und das selbe Steuergerät in verschiedenen Regionen leicht unterschiedlich verhält. Um dem Rechnung zu tragen, können in die erste Spalte eines jeden Testcase in der Tabelle mit dem Testablauf Varianten eingetragen werden. Diese Testcases werden dann nur ausgeführt, wenn sie der aktuell durchlaufenden Variante zugeschrieben wurden. Eine Variante in der obersten Zeile zählt für die gesamte Testgroup. Wenn die Varianten Spalte eines Testcase leer gelassen wird, wird er bei jeder Variante ausgeführt. Ein Ausrufezeichen (!) wird als Negation betrachtet, d.h. wenn in der Varianten Spalte z.B. „!USA“ steht, dann wird der Testcase in allen Varianten außer der USA Variante ausgeführt. Durch Kommata getrennt können auch mehrere Varianten in eine Zelle eingetragen werden.
Weil auch die unterschiedlichen Durchläufe in jeder Variante stattfinden sollten, sind die Varianten den Durchläufen übergeordnet. Wenn zwei Varianten und zwei Durchläufe definiert wurden, wird der eigentliche Test also vier mal ausgeführt. Zuerst die beiden Durchläufe mit Variante 1, dann die selben zwei Durchläufe, allerdings mit Variante 2.

4.6.4 Generieren eines Testmoduls

Das Erzeugen der Datei im .can Format beginnt, wenn der Anwender auf den *„Generate File“* Knopf im Hauptfenster der Anwendung drückt. Zuvor muss in den Optionen das *CAPL*-Format für die Ausgabedatei angewählt werden. Zu diesem Zeitpunkt hat der Nutzer den gewünschten Test modular in der Tabelle des *ibb TestDesigners* aufgebaut. Weiterhin hat er eventuell Durchläufe und Varianten definiert. Jede Zeile in der Tabelle entspricht später einem Testcase und jede Seite einer Testgroup. Sowohl einzelne Testcases wie auch ganze Testgroups können auskommentiert werden, indem man in die erste Spalte ein Rauten-Symbol (#) einfügt.
Zum Schreiben der Ausgabedatei wird ein *StreamWriter* Objekt genutzt. Dieser *StreamWriter* schreibt zunächst den statischen Teil des Rahmens, also den Block mit der Variablendeklaration und den Beginn der *MainTest* Funktion. Die *MainTest* Funktion besteht aus drei Ebenen von Testgroups. Diese Schachtelung von Testgroups ist notwendig, um Durchläufe und Varianten sinnvoll realisieren zu können.

- **Unterste Ebene**
  Auf der untersten Ebene stehen die jeweiligen Testgroups aus der Tabelle. In der Struktur des Testmoduls in *CAPL* stehen alle Testgroups auf der selben Ebene.

- **Mittlere Ebene**
  Die mittlere Ebene bilden die Durchläufe. Für jeden Durchlauf müssen alle Testgroups aufgerufen werden und die Variablen entsprechend ersetzt werden.
- **Oberste Ebene**
  Auf der obersten Ebene stehen die Varianten. Für jede Variante wird eine eigene übergeordnete Testgroup erstellt, die wiederum jeweils alle Durchläufe enthält.

Zur Veranschaulichung dieser Verschachtelung in der *MainTest* Funktion ist in Listing 4.4 ein Beispiel mit zwei Varianten und zwei Durchläufen gegeben.

Listing 4.4: Verschachtelung von Varianten, Durchläufen und Testgroups in der MainTest Funktion

```
1  void MainTest()
2  {
3        TestModuleTitle("Verschateltes Testmodul");
4
5        TestGroupBegin("Variante 1", "Region USA");
6              TestGroupBegin("Durchlauf Normalspannung", "12V");
7                    TestGroupBegin("Eigentlicher Test", "");
8                          testcase_1();
9                    TestGroupEnd();
10             TestGroupEnd();
11             TestGroupBegin("Durchlauf Ueberspannung", "20V");
12                   TestGroupBegin("Eigentlicher Test", "");
13                         testcase_2();
14                   TestGroupEnd();
15             TestGroupEnd();
16       TestGroupEnd();
17
18       TestGroupBegin("Variante 2", "Region Europa");
19             TestGroupBegin("Durchlauf Normalspannung", "12V");
20                   TestGroupBegin("Eigentlicher Test", "");
21                         testcase_1();
22                   TestGroupEnd();
23             TestGroupEnd();
24             TestGroupBegin("Durchlauf Ueberspannung", "20V");
25                   TestGroupBegin("Eigentlicher Test", "");
26                         testcase_2();
27                   TestGroupEnd();
28             TestGroupEnd();
29       TestGroupEnd();
30 }
```

Die ineinander verschachtelten Testgroups werden in die Ausgabedatei geschrieben. In der innersten Testgroup wir dann durch die Zeilen in der Tabelle mit dem Test iteriert. Jeder nicht auskommentierte Testcase wird als *CAPL* Funktionsaufruf in die Ausgabedatei geschrieben. An dieser Stelle ist es noch völlig egal, ob ein Testcase mehrere oder gar keine Methoden beinhaltet.
Durch mehrere Schleifen wird so die gesamte *MainTest* Funktion in die Ausgabedatei geschrieben. Für ein vollständiges Testmodul fehlt jetzt nur noch die Definition der einzelnen Funktionen bzw. Testcases, die aufgerufen werden.
Hierfür wird ein zweites mal über die Spalten der einzelnen Zeilen in der Tabelle iteriert, um für jede

Zeile einen eigenen Testcase in *CAPL* zu formulieren. Zu jeder Methode in einer Tabellenzeile wird der Main Code aus der Methodenbibliothek genommen, die Parameter eingesetzt, gegebenenfalls für den jeweiligen Durchlauf Variablenwerte gesetzt und das Ergebnis in die Ausgabedatei geschrieben. Wenn mehr als eine Methode in einer Zeile der Tabelle steht, dann werden die Main Codes der Methoden untereinander in den selben *CAPL* Testcase geschrieben.
Für die jeweiligen Durchläufe ist es am sinnvollsten, unterschiedliche Testcases in *CAPL* zu formulieren. Zum einen erfolgt die Ersetzung der Variablen für die jeweiligen Durchläufe bereits im *ibb TestDesigner* und nicht erst zur Laufzeit des Test Codes. Zum anderen ist nur so eine aufsteigende Nummerierung der Testcases möglich. Da in den verschiedenen Durchläufen auch tatsächlich verschiedene Methoden aufgerufen werden, soll z.B. eine Methode „Setze Spannung auf 5V“ nicht die selbe Nummer haben wie „Setze Spannung auf 10V“. Für verschiedene Durchläufe durch ein und den selben Test sollen die Testcases aufsteigend nummeriert werden.
Für die Varianten ist jedoch keine fortlaufende Nummerierung nötig. Die Testcases in den einzelnen Varianten sind genau gleich, sie unterscheiden sich nur darin, ob sie ausgeführt werden oder nicht. Jede Variante hat also eine Nummerierung der Testcases von 1 bis zum letzten Testcase.
Die Methoden des *ibb TestDesigners*, mit denen der Main Code generiert wird, mussten für die Erweiterung um *CAPL* als Ausgabeformat neu erstellt werden. Insgesamt wurden im Verlauf der Arbeit über 70 unterschiedliche Methoden erzeugt, deren Komplexität stark variiert. So gibt es vergleichsweise einfache Methoden, wie z.B. „setC(value)“. Diese Methode setzt ein CAN-Signal auf einen bestimmten Wert, der als Parameter übergeben wird. Der Main Code dieser Methode ist in Listing 4.5 zu sehen.

Listing 4.5: Main Code einer Methode zum Setzen eines Signals

```
1 setSignal(%signal%, %value%);
```

Die Variablen sind im Main Code durch umschließende Prozent-Zeichen (%) gekennzeichnet. Hierbei bezeichnet %signal% den Port, auf dem die Methode aufgerufen wird. Bei diesem Beispiel wird ein CAN-Signal auf einen bestimmten Wert gesetzt. Der Port ist also das Signal, das gesetzt werden soll. Die zweite auftretende Variable, hier %value%, wird durch den übergebenen Parameterwert beim Methodenaufruf ersetzt. Wird bei der Testerstellung in einem Testcase die Methode *„setC(1)“* mit einer Wartezeit von einer Sekunde auf dem Port „Umdrehungen“ aufgerufen, so sieht die Funktion in der Ausgabedatei wie in Listing 4.6 dargestellt aus.

Listing 4.6: Funktion zum Setzen eines Signals in der Ausgabedatei

```
1 testcase testcase_1()
2 {
3       testCaseTitle("1", "1 'Action' Set CAN Signal Umdrehungen to 1");
4       TestWaitForTimeout(1000);
5       setSignal(Umdrehungen, 1);
6 }
```

Komplexere Methoden wie das Einstellen eines VT Moduls zur Spannungsversorgung umfassen leicht fünfzehn Zeilen Code und mehr.
Der Vorteil der Methodenbibliothek ist, dass nach und nach die benötigten Methoden angelegt und bei Bedarf abgeändert werden können. Durch das Nutzen der Variablen in den %-Zeichen und das anschließende Suchen und Ersetzen dieser Variablen werden die Methoden allgemein gehalten. Die

Methoden müssen einmal richtig angelegt werden und können dann beliebig oft in verschiedensten Tests genutzt werden.
Natürlich sind auch Tests denkbar, die tausende verschiedene Testcases beinhalten, welche wiederum alle mehrfach in unterschiedlichen Varianten und Durchläufen aufgerufen werden. In einem solchen Fall zeigt sich deutlich der Vorteil eines Generator-Konzepts, wie es der *ibb TestDesigner* anstrebt. Der Aufwand wird durch den tabellarischen Testaufbau so gering wie möglich gehalten, gleichzeitig wird die Übersichtlichkeit maximiert.

4.6.5 Neue Funktionalitäten

Eines der Hauptprobleme bei einem in *XML* verfassten Testmodul ist das Fehlen von Variablen, z.B. zum Zwischenspeichern von Ergebnissen bei Berechnungen. Um dies trotzdem zu ermöglichen, musste bisher eine Methodenbibliothek in *CANoe* importiert werden. Diese Datei im .can-Format beinhaltet *CAPL* Funktionen, die mit *XML* nicht realisierbar sind. Die Funktionen werden dann im *XML* Test mit den entsprechenden Werten aufgerufen. Diese zusätzliche Methodenbibliothek muss extra in jede *CANoe* Testumgebung importiert werden, da sonst der *XML*-Code zur Laufzeit auf nicht definierte Funktionen zugreifen würde. Dieser Weg widerspricht dem Generator-Konzept und wird durch das direkte Erzeugen von *CAPL* hinfällig.
Einige der Funktionalitäten, die bisher nicht oder nur durch Umwege erreichbar waren, werden im Folgenden erläutert:

- **Syscall**
  CAPL bietet die Möglichkeit von Systemaufrufen. Mit diesen Systemaufrufen können über die Kommandozeile des Betriebssystems Dateien ausgeführt werden. Hierfür werden der Pfad einer Datei und eine Wartedauer als Parameter beim Methodenaufruf übergeben. Wenn der Systemaufruf innerhalb dieser Zeit erfolgreich durchgeführt wurde, so gilt der Testcase als bestanden.
  Eine Mögliche Anwendung für eine solche Methode wäre das Ausführen von Batch-Dateien, die zur Laufzeit eines Tests den Prüfling beeinflussen müssen. In so einem Fall kann der *ibb TestDesigner* selbst die Datei aufrufen, auf deren Ausführung warten und anschließend mit dem veränderten Prüfling fortfahren.

- **Time to Signal/Sysvar/Envvar**
  Eine sehr wichtige Funktionalität, die bisher immer nur über den Umweg der zusätzlichen *CAPL* Bibliothek möglich war, ist die Zeitmessung. Oft ist es in einem Test von Relevanz, wie lange die Reaktion auf eine bestimmte Aktion gedauert hat. Es wurden *CAPL* Methoden entwickelt, die für Signale, Systemvariablen oder auch Umgebungsvariablen messen, ob sie in einem bestimmten Zeitrahmen einen gewissen Wertebereich betreten oder verlassen. Wird der Wertebereich während der Wartezeit betreten oder verlassen, wird mit dem nächsten Testcase fortgefahren und im Report protokolliert, wie lange das Warten auf das Signal oder die Variable gedauert hat. Falls das gewünschte Ergebnis der Messung im gegebenen Zeitrahmen nicht erreicht wurde, so wird der Testcase als nicht bestanden vermerkt und der Wert am Ende der Wartezeit protokolliert.
  Da *CAPL* nun direkt erzeugt werden kann, können Testcases zur Zeitmessung in wenigen Zeilen direkt im Testmodul realisiert werden.

- **Wait Random**
  Um den Testablauf zumindest teilweise zu randomisieren, wurde eine Methode zum zufällig lange Warten entwickelt. Als Parameter werden minimale und maximale Wartezeit angegeben und der Test wartet diese Zeitdauer in Sekunden. Auch dieser Fall war in *XML* nicht möglich, da man damit keine Zufallszahlen generieren oder zwischenspeichern kann.

- **Rechnungen**
  Eine weitere neu hinzugekommene Funktionalität ist das Ausführen von einfachen Rechnungen, z.B. der Form y = ax + b. Auf dem Port einer beliebigen Variablen kann eine Rechnung ausgeführt werden, wobei a, x und b als Parameter angegeben werden. Sie können entweder konkrete Werte sein, oder wiederum selbst aus Variablen bestehen. Der Variablen des aufrufenden Ports wird dann das Ergebnis der Berechnung zugewiesen.
  Auch diese Funktion lässt sich aufgrund des Fehlens der Variablendeklaration in *XML* nicht realisieren.

- **Verzweigungen**
  Der Kontrollfluss bei einem *XML* Test ist absolut statisch und nicht veränderbar. *CAPL* jedoch verfügt als Programmiersprache über gängige Kontrollstrukturen wie if/else oder while.
  Um die Nutzung dieser Kontrollstrukturen in einem *CAPL* Testmodul aufzuzeigen, wurden mehrere Methoden implementiert. Bei diesen Methoden wurde die Wertzuweisung an eine Bedingung geknüpft, ein Signal soll also z.B. nur auf einen Wert gesetzt werden, wenn es zuvor einen bestimmten anderen Wert hatte. Falls der erwartete Wert bei der Überprüfung nicht vorliegt, so soll die Wertzuweisung nicht stattfinden oder ein anderer Wert zugewiesen werden.

- **Globale Variablen**
  Falls für einen Test globale Variablen notwendig sind, können diese entweder wie der restliche Code erzeugt werden oder nachträglich einfach in die Ausgabedatei hinzugefügt werden. Ist es etwa innerhalb eines Tests nötig, mehrere Zeitpunkte zu speichern und am Schluss auszugeben oder Berechnungen über die Dauer eines einzelnen Testcases hinaus zu speichern, so kann dies durch globale Variablen getan werden. Die entsprechenden Variablen können im *ibb TestDesigner* selbst als Ports angelegt werden oder als Parameter bei Methodenaufrufen eingesetzt werden. Da *XML* keine Erzeugung von Variablen ermöglicht, mussten diese bisher in der für jeden Test mitgeführten *CAPL* Bibliothek definiert und verwaltet werden.

5 Evaluation

Die Evaluation der Erweiterung des *ibb TestDesigners* erfolgt an einem voll funktionsfähigen Testsystem für ein Steuergerät. Dieses Steuergerät ist für die Ansteuerung der verschiedenen Lichter eines Autoanhängers zuständig.

5.1 Beschreibung des Testaufbaus

Der Testaufbau kann in die drei wesentlichen Bestandteile *Steuergerät*, *Originallast* und *VT System* gegliedert werden. Diese werden im Folgenden näher beschrieben.

5.1.1 Steuergerät

Das hier verwendete Steuergerät hat mehrere Ein- und Ausgänge. An den Ausgängen sind die Aktoren, in diesem Fall also Lampen, angeschlossen. Die Eingänge sind die Spannungsversorgung und diverse Sensoren, die z.B. angeben, ob ein Anhänger mit dem Auto verbunden ist oder nicht. Die Verarbeitung der Signale erfolgt durch einen oder mehrere Prozessoren und Speichereinheiten. Zusätzlich ist das Steuergerät an einen CAN Bus angeschlossen, der verschiedene elektronische Komponenten im Bordnetz eines Autos verbindet. Über diesen CAN Bus können CAN Signale vom Steuergerät empfangen und versendet werden.

5.1.2 Originallast

Die Lasten, die an den Ausgängen des Steuergeräts hängen, sind Lampen. Diese Lasten könnten mithilfe des *VT Systems* simuliert werden. Im vorliegenden Testsystem sind diese Lampen jedoch alle tatsächlich innerhalb einer Box vorhanden. Diese Box wird an das *VT System* angeschlossen, das alle wichtigen Schnittstellen des Testaufbaus überwacht. Dadurch, dass die Lampen angeschlossen sind, kann deren Ansteuerung durch das Steuergerät auch sichtbar gemacht werden. Die einzelnen Lampen des Anhängers sind:

- Bremslicht
- Rückfahrlicht
- Nebelschlussleuchte
- Schlusslicht links

- Schlusslicht rechts
- Blinker links
- Blinker rechts

Jedes dieser Lichter kann vom Steuergerät einzeln angesteuert werden.

5.1.3 VT System

Das *VT System* bildet die Schnittstelle zwischen der Software *CANoe* und dem Prüfling, der getestet werden soll. Für jedes Testsystem müssen die richtigen Module ausgewählt und entsprechend verkabelt werden [Vec15e]. In dem zur Evaluation genutzten Testsystem sind folgende Module enthalten:

- 2x **VT1004**
  Das Modul VT1004 kann an maximal vier Ausgänge eines Steuergeräts angeschlossen werden. Entweder simuliert es durch interne Widerstände selbst die Last, oder es wird zwischen ein Steuergerät und dessen Aktoren angeschlossen. An dem Modul können dann u.a. Spannungen und Ströme gemessen oder verschiedene Relais geschalten werden.
  In diesem Aufbau werden zwei VT1004 zwischen das Steuergerät und die insgesamt sieben verschiedenen Lampen angeschlossen. Durch das Schalten unterschiedlicher Relais können Szenarien wie Kurzschlüsse oder Kabelbrüche am Ausgang des Steuergeräts simuliert werden.
- 1x **VT2004**
  Das Modul VT2004 kann an maximal vier Eingänge eines Steuergeräts angeschlossen werden. Das Modul VT2004 kann entweder selbst Sensoren simulieren, oder wird zwischen ein Steuergerät und dessen tatsächliche Sensoren angeschlossen. Auch dieses Modul kann durch Schalten verschiedener Relais Fehlerfälle simulieren. Außerdem kann es verschiedene Spannungen und Impulse erzeugen.
  Bei diesem Testaufbau wird eine VT2004 zwischen die Spannungsversorgung und das Steuergerät angeschlossen. Durch Ansteuern dieses Moduls kann das Steuergerät mit verschiedenen Spannungen versorgt werden.
- 1x **VT7001**
  Das Modul VT7001 hat zwei separate Ausgänge für die Stromversorgung von Steuergeräten. Bei diesem Testsystem wird die VT7001 zur Messung und Regelung von Spannungen und Strömen genutzt.
- 1x **VT6050**
  Das Modul VT6050 ist ein Realtime-Modul, das über Ethernet an den Rechner angeschlossen werden kann, auf dem *CANoe* ausgeführt wird. Echtzeitkritische Teile des Tests werden von dem Prozessor dieses Moduls ausgeführt.
- 1x **VT8012**
  Das Modul VT8012, auch als Backplane bezeichnet, versorgt alle anderen Module mit der Betriebsspannung von 12 Volt. Außerdem ermöglicht es eine Kommunikation der einzelnen Module mit *CANoe*.

5.1.4 Gesamtaufbau

In Abb. 5.1 ist ein Foto des beschriebenen Testsystems zu sehen. Die *VT Module* werden vorne in den Schaltschrank eingeschoben und im Inneren verkabelt. Auf dem Schrank befindet sich die Box mit der Originallast, also den verschiedenen Lichtern. Das Steuergerät ist ebenfalls innerhalb des Schranks verkabelt.
Einige der im Bild sichtbaren *VT Module* sind nicht für diesen konkreten Steuergerätetest notwendig.

Abbildung 5.1: Testsystem von *ibb* für einen Hardware Test

Dieser Aufbau erlaubt das Ansteuern aller Ein- und Ausgänge des Steuergeräts, sowie eine Spannungs- und Strommessung an allen notwendigen Stellen. Somit ermöglicht das vorliegende Testsystem eine umfangreiche Prüfung des angeschlossenen Steuergeräts gemäß einer Liste von Requirements.

5.2 Requirements

Um zu zeigen, dass mit der entstandenen Erweiterung ein *CAPL* Testmodul generiert werden kann, werden eine Reihe von Requirements für einen Test benötigt. Die in Tabelle 5.1 aufgeführten Requirements sollen mithilfe eines so generierten Testmoduls überprüft werden.

Tabelle 5.1: Liste von Requirements für einen Steuergerätetest

	Requirement ID	**Anforderung**
0		Sämtliche Testcases sollen bei 12, 14 und 16 Volt erfüllt werden
1	BrL	Bremslicht
2	BrL_01	Nach Setzen von Signal BrkLgt_On_Rq soll das Bremslicht leuchten
3	BrL_02	Bei Kabelbruch soll Signal TrlrBrkLmp_Flt nach 40 Sekunden gesetzt sein
4	SlR	Schlusslicht Rechts
5	SlR_01	Nach Setzen von Signal PkLmp_Rt_On_Rq soll das rechte Schlusslicht leuchten
6	SlR_02	Bei Kabelbruch soll Signal TrlrTlLmp_Rt_Flt nach 40 Sekunden gesetzt sein
7	SlR_03	Bei Kurzschluss nach Plus soll Signal TrlrTlLmp_Rt_Flt nach 40 Sekunden nicht gesetzt sein
8	BlL	Blinker Links
9	BlL_01	Nach Setzen der Signale TurnInd_Lt_On = 1 und TurnLmpOnDur = Blinkdauer in ms soll der Blinker aufleuchten
10	BlL_02	Bei Kabelbruch soll Signal TrlrTurnLmp_Lt_Flt nach 2 Sekunden gesetzt sein
11	BlL_03	Bei Kurzschluss nach Plus soll Signal TrlrTurnLmp_Lt_Flt nach 2 Sekunden nicht gesetzt sein

Für jedes einzelne Licht gibt es ein „On Request (On_Rq)“ Signal, das dem Steuergerät geschickt wird. Im Auto würde ein solches Signal z.B. nach dem Drücken eines Knopfes gesendet werden. Wenn das Steuergerät ein solches Signal empfängt, wird es verarbeitet und die notwendigen Schritte werden ausgeführt, das entsprechende Licht wird also mit Spannung versorgt und leuchtet. Falls ein Fehler auftritt, sendet das Steuergerät ein „Fault (Flt)“ Signal, damit an der entsprechenden Anzeigeeinheit ein Fehler gemeldet werden kann.
Das hier getestete Steuergerät ist so konzipiert, dass bei einem Kabelbruch, das heißt einer Unterbrechung am Ausgang des Steuergeräts hin zum entsprechenden Licht, nach ca. 30 Sekunden ein Fehlersignal gesendet wird. Für die Simulation der Unterbrechung wird ein Relais des VT1004 Moduls

geöffnet, sodass kein Strom mehr durch die Lampe fließen kann.
Wenn ein Kurzschluss nach Plus an einer Lampe auftritt, d.h. eine Lampe wird mit Spannung versorgt, obwohl kein „On Request“ Signal vorliegt, dann soll kein Fehler Signal gesendet, sondern anders verfahren werden.
Diese Liste von Requirements stellt aus Platz- und Zeitgründen nur einen Teil der notwendigen Testcases dar, die zur vollständigen Validierung eines Steuergeräts notwendig wären. Weitere Testcases können jedoch ebenfalls mit der *CAPL* Erweiterung erstellt werden.

5.3 Test

Die zuvor beschriebene Verhaltensweise soll mittels eines Tests an einer Auswahl von Lichtern validiert werden. Hierzu muss zunächst der Test mit dem *ibb TestDesigner* erstellt werden.

5.3.1 Testerstellung

Abb. 5.2 zeigt das schrittweise Vorgehen, dem ein Anwender bei jeder Testerstellung mit dem *ibb TestDeisgner* folgt.

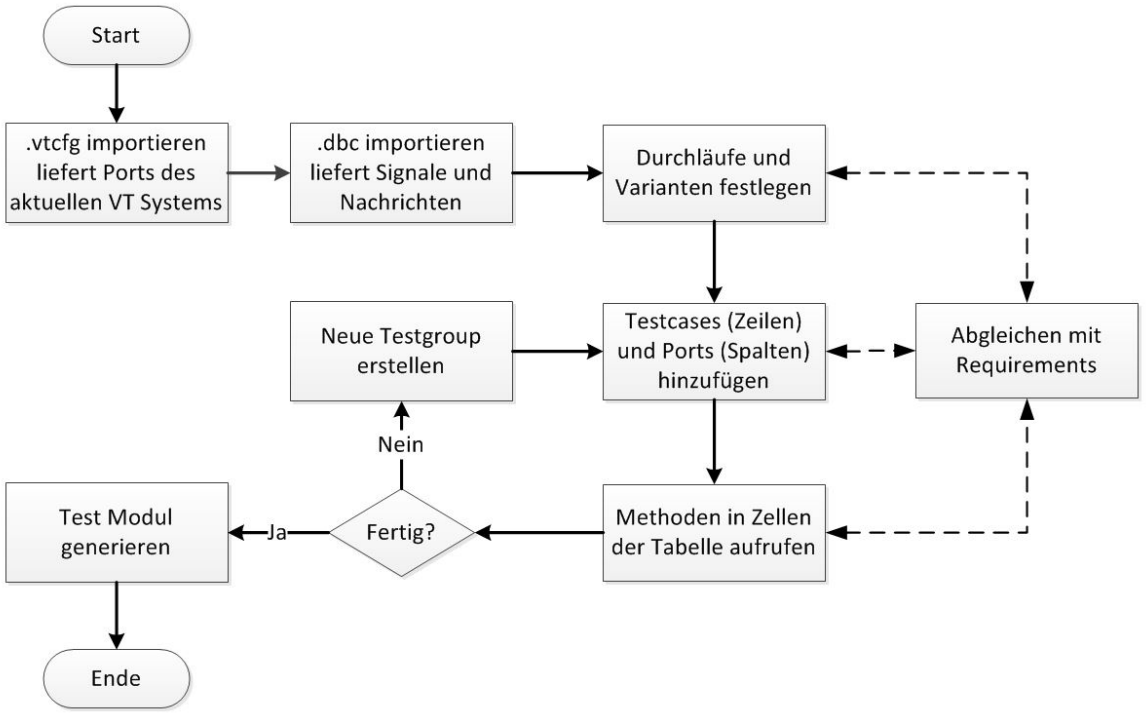

Abbildung 5.2: Ablauf der Testerstellung im *ibb TestDesigner*

Die Beschreibung des Testsystems in *CANoe* wird erstellt und abgespeichert. Diese Konfigurationsdatei (.vtcfg) wird vom *ibb TestDesigner* eingelesen und die Ports und Devices automatisch definiert.

Weiterhin werden die Signale und Nachrichten des Steuergeräts benötigt, diese speichert *CANoe* in einer .dbc Datei. Wurden die notwendigen Dateien importiert, so kann mit der Erstellung des Tests begonnen werden.
Zu Beginn empfiehlt es sich, einmal die nötigen Varianten und Durchläufe festzulegen. Dies kann auch später geschehen, aber dann müssen unter Umständen die Testcases angepasst werden, um statt konkreten Parameterwerten Variablen zu benutzen.
Da beim Programmstart bereits eine leere Testgroup erzeugt wird, kann mit dem Hinzufügen von Testcases und den dafür nötigen Ports begonnen werden. Dabei muss ein ständiger Abgleich mit dem Requirements erfolgen, um das Ziel nicht aus den Augen zu verlieren.
Nachdem Schritt für Schritt eine Testgroup vervollständigt wurde, kann entweder eine weitere Testgroup erstellt werden und analog fortgefahren werden, oder der Nutzer entscheidet sich, dass sein Test vorerst fertig ist. Wenn dies der Fall ist, so kann per Knopfdruck das Testmodul für *CANoe* generiert werden. Die damit erzeugte .can Datei enthält einen syntaktisch korrekten Testablauf in *CAPL*, der ohne weitere Modifikation ausgeführt werden kann.

Bei dem Beispiel des Steuergerätetests bietet es sich anfänglich an, Durchläufe und Varianten zu definieren. In den Durchläufen kann die Spannung verändert werden, also werden drei Durchläufe mit je 12, 14 und 16 Volt Versorgungsspannung festgelegt. Als Varianten können in diesem Fall einmal Kurzschlüsse und einmal Kabelbrüche definiert werden. Die entsprechenden Tabellen sehen dafür dann wie in Abb. 5.3 gezeigt aus.

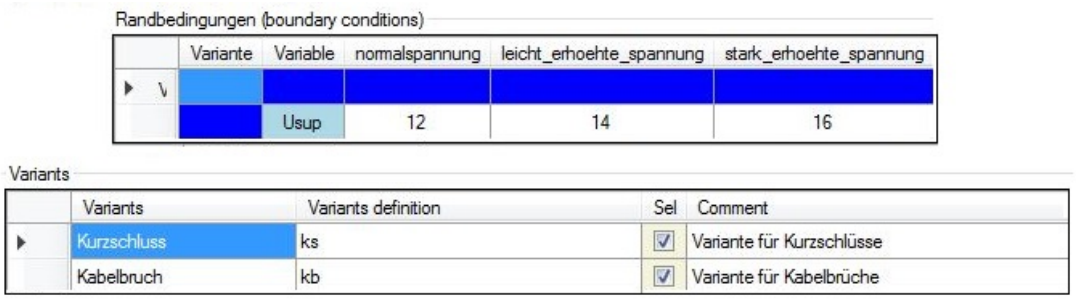

Abbildung 5.3: Durchgänge und Varianten eines Beispieltests

Im Hauptfenster des *ibb TestDesigners* kann anschließend mit den einzelnen Testgroups begonnen werden. In so einem Fall ist es sinnvoll, eine Testgroup für Preconditions (Relais für Originallasten verbinden, Versorgungsspannung Usup einstellen, prüfen ob das Steuergerät betriebsbereit ist) und eine für Postconditions (Herunterfahren der Versorgungsspannung, Originallasten abtrennen) zu erstellen.
Die mittleren drei Testgroups bilden Bremslicht, Schlusslicht rechts und Blinker links. Eine dieser Testgroups ist in Abb. 5.4 dargestellt.

	Variante	Signal	Requirement ID	Δt	TestCase Type	TestCase Title	KL15	B_EIS_ISw_Stat	PkLmp_Rt_On_Rq	TlLmp_Rt_On_Rq	TrlrTlLmp_Rt_Flt	Schlusslicht_rechts	Bemerkungen
		14	SIR		Precondition	Set Voltage KL15 to 12,14,16	cVoltageCAPL(%Usup%)	kl15CAPL					
	kb	15	SIR_01	1	Action	Set CAN Signal PkLmp_Rt_On_Rq to 1			cSignalCANCAPL(1)				
	kb	16	SIR_01	1	Result	Check if value Schlusslicht_rechts is 12,14,16 +- 0.5;						mVoltageTolCAPL(%Usup%;0.5)	
	kb	17	SIR_02	1	Action	disconnect OriginalLoad Schlusslicht_rechts						dOriginalLoadCAPL	
	kb	18	SIR_02		Result	Check if desired Value TrlrTlLmp_Rt_Flt is reached in 0 to 40 s;					mTimeToSignalCANCAPL(1;1;0;40;1)		
	kb	19	SIR_02		Action	Set CAN Signal PkLmp_Rt_On_Rq to 0, connect original load			cSignalCANCAPL(0)				
	kb	20	SIR_02	1	Action	Switch Original Load to Schlusslicht_rechts						cOriginalLoadCAPL	
	ks	21	SIR_03		Action	Set CAN Signal PkLmp_Rt_On_Rq to 0			cSignalCANCAPL(0)				
	ks	22	SIR_03	1	Action	Connect Schlusslicht_rechts to Busbar1						cBusBar1CAPL	
	ks	23	SIR_03	1	Result	Check if value Schlusslicht_rechts is 12,14,16 +- 0.5;						mVoltageTolCAPL(%Usup%;0.5)	
	ks	24	SIR_03		Action	Wait for 40s;					waitCAPL(40)		
	ks	25	SIR_03		Result	Check if a TrlrTlLmp_Rt_Flt value is 0;					mSignalCANFalseCAPL		
	ks	26	SIR_03		Action	Disconnect Schlusslicht_rechts from Busbar1						dBusBar1CAPL	

Abbildung 5.4: Testgroup eines Beispieltests im *ibb TestDesigner*

Im *ibb TestDesigner* hat sich die Konvention durchgesetzt, dass Methoden die etwas verbinden oder setzen mit „c“ (für „connect“) beginnen, Methoden die etwas messen mit „m“ (für „measure“) und solche, die etwas trennen, mit „d“ (für disconnect).
Zu Beginn wird die vom jeweiligen Durchgang abhängige Versorgungsspannung an den Eingang des Steuergeräts gelegt, was das Modul VT2004 übernimmt. Dafür wird beim Methodenaufruf kein fester Wert als Parameter übergeben, sondern die Variable aus der oben gezeigten Tabelle für Durchgänge, %Usup%. Zusätzlich wird eine Systemvariable gesetzt, die dem Steuergerät den Status eines laufenden Motors mitteilt. Da diese Aktionen sowohl beim Kurzschluss als auch beim Kabelbruch ausgeführt werden sollen, wird die Varianten Spalte leer gelassen. Die folgenden Schritte simulieren einen Kabelbruch, weshalb sie nur in der Variante „kb“ berücksichtigt werden.
Das CAN Signal „On Request“ wird gesendet und anschließend überprüft, ob die Spannung am Ausgang des Steuergeräts Richtung Schlusslicht anliegt. Danach wird die Verbindung zur Originallast getrennt, der Kurzschluss wird simuliert. Als nächstes wird 40 Sekunden lang gewartet, ob das Fehlersignal gesendet wird. Zum Schluss der Kabelbruch Variante wird die Originallast wieder verbunden und das „On Request“ Signal auf null gesetzt.
Der hier beschriebene Test deckt die Requirements SlR_01 und SlR_02 ab. Für das Requirement SlR_03 wird ähnlich vorgegangen: statt die Originallast abzukoppeln wird allerdings ein Relais geschlossen, das die Lampe mit Spannung versorgt, ohne dass ein entsprechendes „On Request“ Signal gesendet wurde. Das Licht leuchtet also dauerhaft, obwohl es gar nicht leuchten sollte. Dann wird nach 40 Sekunden überprüft, ob das Fehler Signal auf null steht, also nicht gesendet wurde. Am Ende wird der Kurzschluss wieder entfernt.

Der gesamte Test besteht aus fünf Testgroups und zweimal drei Durchläufen, wobei je nach Variante nur ein Teil der Testcases ausgeführt wird. Der generierte *CAPL* Code für dieses Beispiel umfasst über 1600 Zeilen Code.

5.3.2 Ausführung

Wenn der Test im *ibb TestDesigner* fertig definiert wurde, kann er mit einem Klick auf den „Generate File“ Knopf erstellt werden. Anschließend wird der Test in *CANoe* geladen und ohne weitere Ände-

rungen am Code des Tests ausgeführt. *CANoe* kommuniziert mit dem *VT System* und führt den Test durch. Während der Durchführung zeigt *CANoe* den Fortschritt wie in Abb. 5.5 dargestellt an.

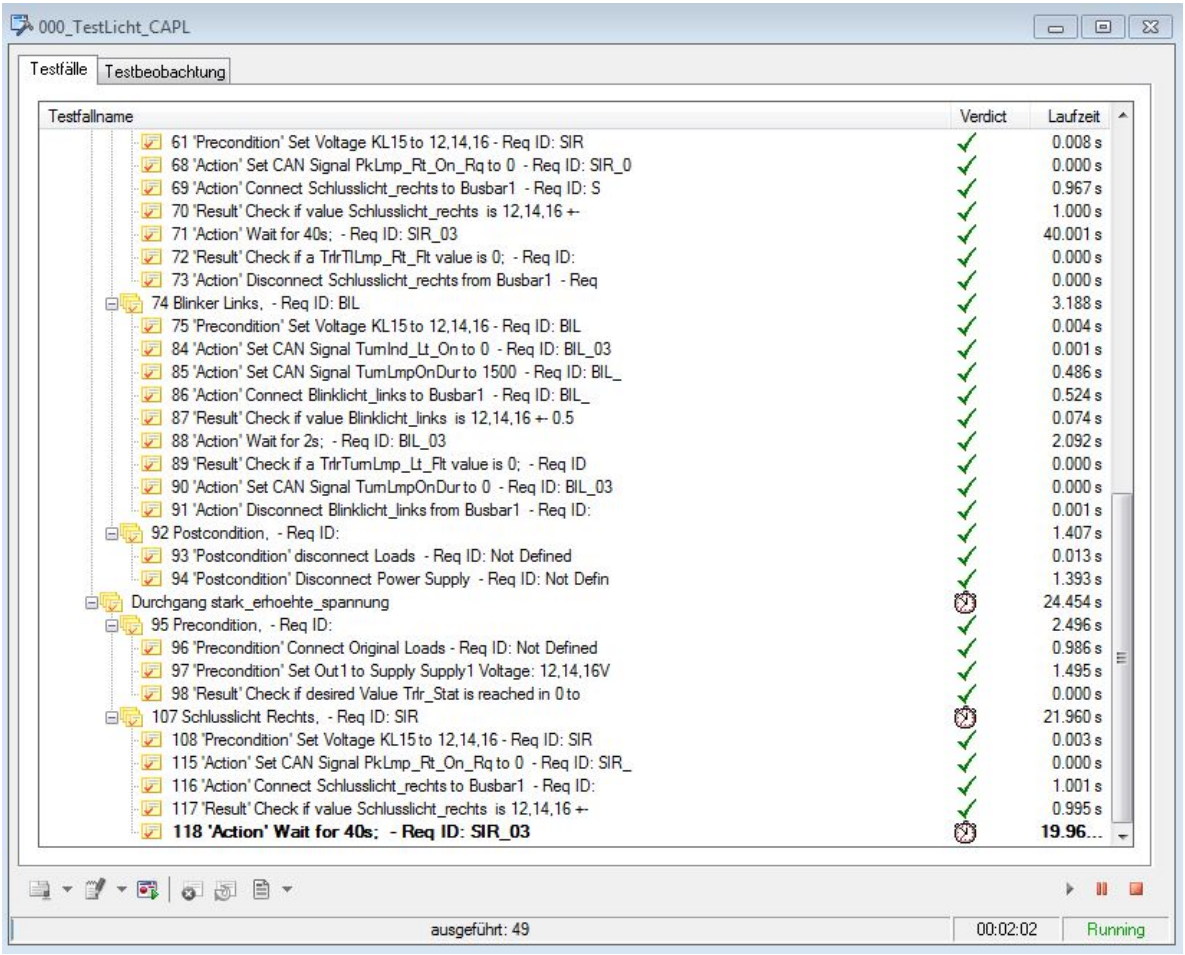

Abbildung 5.5: Ausführung eines Tests in *CANoe*

Nach Beendigung der Ausführung liegt ein umfassender Report vor. Die Ausführungszeit des hier erstellten Beispieltest beträgt 06:34 Minuten bei 147 ausgeführten Testcases.

5.3.3 Report

Der Report über den Ablauf eines Tests wird von *CANoe* wahlweise in *XML* oder *HTML* erstellt. Darin enthalten ist eine Übersicht über die einzelnen Testcases und deren Ergebnis wie es in Abb. 5.6 zu sehen ist.

Statistics

Overall number of test cases	147	
Executed test cases	147	100% of all test cases
Not executed test cases	0	0% of all test cases
Test cases passed	147	100% of executed test cases
Test cases failed	0	0% of executed test cases

Test Case Results

1		**Variante Kurzschluss**	
1.1		**Durchgang normalspannung**	
1.1.1		**1 Precondition, - Req ID:**	
1.1.1.1	2	2 'Precondition' Connect Original Loads - Req ID: Not Defined	pass
1.1.1.2	3	3 'Precondition' Set Out1 to Supply Supply1 Voltage: 12,14,16V, MaxCurrent: 5A - Req ID: Not Defined	pass
1.1.1.3	4	4 'Result' Check if desired Value Trlr_Stat is reached in 0 to 10 s; - Req ID: Not Defined	pass
1.1.2		**13 Schlusslicht Rechts, - Req ID: SIR**	
1.1.2.1	14	14 'Precondition' Set Voltage KL15 to 12,14,16 - Req ID: SIR	pass
1.1.2.2	21	21 'Action' Set CAN Signal PkLmp_Rt_On_Rq to 0 - Req ID: SIR_03	pass
1.1.2.3	22	22 'Action' Connect Schlusslicht_rechts to Busbar1 - Req ID: SIR_03	pass
1.1.2.4	23	23 'Result' Check if value Schlusslicht_rechts is 12,14,16 +- 0.5; - Req ID: SIR_03	pass
1.1.2.5	24	24 'Action' Wait for 40s; - Req ID: SIR_03	pass
1.1.2.6	25	25 'Result' Check if a TrlrTlLmp_Rt_Flt value is 0; - Req ID: SIR_03	pass
1.1.2.7	26	26 'Action' Disconnect Schlusslicht_rechts from Busbar1 - Req ID: SIR_03	pass

Abbildung 5.6: Statistiken und Ergebnisse eines Testablaufs in *CANoe*

Jeder einzelne Testcase kann noch näher betrachtet werden, wie in Abb. 5.7 dargestellt. In dieser detaillierten Aufschlüsselung enthalten sind genaue Zeitstempel, Beschreibungen der einzelnen Teststeps und Zwischenresultate. Falls ein Testcase nicht das gewünschte Ergebnis hat, kann hier genau nachgeschaut werden, was zum Scheitern des Testcase geführt hat.

[−] **2.2.3.4 Test Case 64: 64 'Action' disconnect OriginalLoad Schlusslicht_rechts - Req ID: SIR_02: Passed**

Test case begin: 2015-11-06 10:56:55 (logging timestamp 271.606598)
Test case end: 2015-11-06 10:56:56 (logging timestamp 272.606598)

Main Part of Test Case

Timestamp	Test Step	Description	Result
272.606598	Resume reason	Elapsed time=1000ms (max=1000ms)	-
272.606598		Original Load Schlusslicht_rechts sucessfully disconnected.	pass

[−] **2.2.3.5 Test Case 65: 65 'Result' Check if desired Value TrlrTlLmp_Rt_Flt is reached in 0 to 40 s; - Req ID: SIR_02: Passed**

Test case begin: 2015-11-06 10:56:56 (logging timestamp 272.606598)
Test case end: 2015-11-06 10:57:27 (logging timestamp 303.596866)

Main Part of Test Case

Timestamp	Test Step	Description	Result
303.596866	Resume reason	Resumed on signal 'Signal:CAN1/BODY/Trailer_Stat_AR/TrlrTlLmp_Rt_Flt' Elapsed time=30990.3ms (max=40000ms)	-
303.596866		Time to signal: 30.990268, condition: 0 < delay time < 40, dataMin = 1, dataMax = 1, Signal = TrlrTlLmp_Rt_Flt, Value = 1.000000	pass

Abbildung 5.7: Detaillierter Ablauf von Testcases in *CANoe*

5.4 Rückblick auf die gestellten Anforderungen

Zum Schluss der Evaluation erfolgt eine Überprüfung der in Abschnitt 4.3 formulierten Anforderungen, die im Laufe dieser Arbeit erfüllt werden sollten.

- Die Generierung von *XML*-Code ist weiterhin möglich, da keine Veränderung an den Stellen des Programms vorgenommen wurde, die eine *XML* Ausgabedatei erzeugen
- Es wurde ein Weg gefunden, das bestehende Konzept mit Devices, Ports und Methoden zu nutzen. Dieses Konzept wurde nicht maßgeblich verändert
- Die Ausgabedatei liegt im .can Format vor und beinhaltet nur Funktionen, die von *CAPL* unterstützt werden. Diese Datei kann ohne Abänderung in *CANoe* geladen und erfolgreich ausgeführt werden
- Es wurde eine eigene Methodenbibliothek für die *CAPL* Methoden des *ibb TestDesigners* angelegt. Diese Methodenbibliothek ist die *ibbMethodenLibCapl.xml*
- Es können mehrere Varianten und Durchläufe automatisch erzeugt werden
- Es wurden eine Reihe von Funktionalitäten hinzugefügt, die mit einem *XML* Test nicht oder nur über Umwege erreichbar waren. Bei einem Test in *CAPL* muss keine weitere Bibliothek in *CANoe* importiert werden, da *CAPL* bei der Testdefinition mächtiger ist als *XML*
- Der Code ist so allgemein gehalten wie möglich, Kommentare an vielen Stellen steigern die Lesbarkeit
- Die Performanz ist gleich gut wie bei der Erstellung von *XML*-Code. Selbst bei großen Tests ist während der Generierung keine merkbare Wartezeit festzustellen.
- Dadurch, dass der Anwender wie zuvor bei der *XML* Generierung auf der graphischen Oberfläche einen Test mit den Methoden des *ibb TestDesigners* erstellt, ist es genauso gut verständlich und handhabbar wie bisher
- Die bestehende Logfile über die einzelnen Schritte während der Generierung der Ausgabedatei wird aktualisiert
- Es erfolgte eine regelmäßige Rücksprache mit Herrn Dipl.-Ing. (FH) Peter Heidenwag über den aktuellen Stand und das weitere Vorgehen. Dies machte den Entwicklungsprozess für den Kunden *ibb* transparent und für den Entwickler zielführend
- Bereits früh gelang die Generierung einer einzelnen Testgroup mit wenigen Testcases. Schritt für Schritt folgte eine Erweiterung um eine beliebige Anzahl von Testgroups, später dann auch Durchläufen und Varianten. Parallel dazu wurde die Methodenbibliothek laufend erweitert
- In der Evaluation wurde gezeigt, dass durch die Erweiterung ein umfangreiches *CAPL* Testmodul generiert und auch fehlerfrei ausgeführt werden kann

6 Weitere Anwendungsgebiete

Die Ausgabe des *ibb TestDesigners* hängt maßgeblich vom Main Code ab, der für die einzelnen Methoden hinterlegt ist. Wenn dieser entsprechend formuliert ist und statt einem *XML*- oder *CAPL*-Gerüst ein anderer Rahmen in die Ausgabedatei geschrieben wird, dann können beliebige Ausgaben erzeugt werden, die auf Text basieren und in irgendeiner Weise strukturiert sind. Maßgeblich dafür verantwortlich ist das allgemein gehaltene Konzept von Devices, deren einzelnen Ports und Methoden, die auf den Ports eines Device aufgerufen werden können. Im Folgenden soll die Möglichkeit für drei weitere Anwendungsszenarien aufgezeigt und näher erläutert werden.

6.1 Planungssoftware für Bauingenieure

Ein denkbares Gebiet für das eine Anpassung des *ibb TestDesigners* möglich wäre, ist eine Software für Bauingenieure zur Planung von Gebäuden. Mit einer solchen Software könnte eine Kalkulation für die Kosten eines Hausbaus oder einer Sanierung erzeugt werden. Die generierte Ausgabe könnte als Eingabe in eine *Microsoft Excel*-Datei dienen. Mit den bereits vorhandenen Mechanismen zum Import und Export von *Excel*-Dateien könnte statt einer Textdatei auch gleich die gewünschte *Excel*-Datei erzeugt werden.
Jede Testgroup würde einem Raum des Gebäudes entsprechen und in jedem Raum müssen mehrere Arbeitsschritte durchgeführt werden, was den einzelnen Testcases entspricht.
Unterschiedliche Devices könnten z.B. sein:

- **Bauen**
  In einem Device würden sich alle Ports befinden, die etwas mit dem Rohbau von Gebäuden zu tun haben. Die Methoden dieses Device könnten dann u.a. Wände einziehen, Böden legen oder auch Fenster einsetzen sein. Als Parameter kann der Baustoff und ein Preis pro Quadrat- oder Kubikmeter übergeben werden.
- **Oberfläche bearbeiten**
  Dieses Device würde die Bearbeitung unterschiedlicher Oberflächen beinhalten. Es wären Ports denkbar für das Verlegen von Bodenbelägen, Tapezieren von Wänden oder Streichen von Decken. Als Parameter könnten auch hier die Baustoffe und die Größe der zu bearbeitenden Fläche übergeben werden.
- **Elektrik**
  Eine weiteres Device für die Elektrik eines jeden Raumes wäre möglich. In diesem Device könnte es Ports für das Verlegen von Leitungen oder Anbringen von Lampen geben. Parameter hierfür wären Länge und Art der Leitung oder Anzahl und Kategorie der Lampen.

Für jeden Methodenaufruf können die verschiedenen Durchläufe und Varianten genutzt werden. In den Durchläufen können die Baustoffe oder die Preise variieren, um minimale und maximale Versionen eines Gebäudes durchzurechnen. Bei verschiedenen Varianten könnte ein Haus einmal mit und einmal ohne Keller entworfen werden.
Es zeigt sich, dass das Konzept, das vom *ibb TestDesigner* verfolgt wird, nicht alleine an das Anwendungsgebiet der Testerstellung gebunden ist. Je nach Interpretation der Tabelle und der Methoden können völlig andere Aufgaben erfüllt werden.

6.2 Abrechnung von Dienstleistungen

Ein weiteres Aufgabenfeld, für das der *ibb TestDesigner* abgewandelt werden kann, wäre das Abrechnen von Dienstleistungen einer Firma. Es kann geplant werden, welche Mitarbeiter innerhalb eines Projekts wie viel Zeit in bestimmte Aufgaben investieren müssen und wie dies abzurechnen ist. Auch hier könnte die Datei für die Ausgabe wieder eine *Microsoft Excel*-Tabelle sein. Dieses Format bietet sich besonders an, da die Ausgabe des *ibb TestDesigners* immer eine strukturierte Textdatei ist.
Als Devices könnten in diesem Fall verschiedene Kategorien von Mitarbeitern identifiziert werden, die bestimmte Aufgabenbereiche in der Firma übernehmen. Die Ports könnten in einem kleineren Unternehmen die Mitarbeiter selbst beschreiben. In einer größeren Firma würden die Ports eine feinere Untergliederung der Devices darstellen.
Die einzelnen Methoden wären bestimmte Dienstleistungen, die ein Mitarbeiter bzw. eine Gruppe von Mitarbeitern erbringt. Parameter hierfür könnten Dauer der Arbeit und Stundensatz für die jeweilige Tätigkeit sein.
Ein Projektplaner kann sich dann überlegen, welche Tätigkeiten im Rahmen eines Projekts nötig sind und welche Mitarbeiter dafür wie viel Zeit zu investieren haben. Auch die Durchläufe und Varianten können hier genutzt werden, um unterschiedliche Planungen ein und des selben Projekts anzustellen. Das Ergebnis ist wieder ein klar strukturierter Plan, der auf einer graphischen Oberfläche erstellt wurde.

6.3 Erzeugen von Texten

Ein letztes mögliches Anwendungsgebiet, das kurz beleuchtet werden soll, ist die Erzeugung von strukturiertem Fließtext.
Im Rahmen der Testerstellung wäre es z.B. möglich, zusätzlich zu der Ausgabedatei mit der Testdefinition eine weitere Datei zu erzeugen, die den Test und dessen einzelne Schritten textuell beschreibt. So wie die Requirements als Tabelle oder Text eine Eingabe für den Prozess der Testerstellung darstellen, könnte eine genaue Beschreibung der Abläufe und einzelnen Schritte der Durchführung als Ausgabe generiert werden. Zum Schluss kann ein Vergleich der vorab formulierten Schritte mit den tatsächlich durchgeführten gezogen werden.

Strukturierter Text findet sich auch bei der Erstellung verschiedener Zeugnisse, seien es Schulzeugnisse oder Arbeitszeugnisse. Mit einer Abwandlung des *ibb TestDesigners* könnte auch hier die Arbeit erleichtert werden, indem Zeugnistexte nicht jedes Mal neu geschrieben werden müssen,

sondern auf einen Textbaukasten zurückgegriffen werden kann. Im Main Code könnte statt Programmcode einfach Text hinterlegt sein. Beim Aufruf einer Methode könnte ein Parameter übergeben werden, der aussagt, wie gut die zu beurteilende Person in dem bewerteten Teilgebiet ist. Dementsprechend werden dann die notwendigen Parameter der Schlüsselwörter im Text ersetzt.
So könnte ein Lehrer mit einer einmal angelegten Methodenbibliothek schnell und übersichtlich Zeugnisse für eine ganze Schulklasse generieren, in denen die Schüler in Bezug auf verschiedene Aspekte bewertet werden.

Fazit

Die Möglichkeiten für eine Anwendung des *ibb TestDesigner*-Konzepts sind vielseitig. Viele strukturierte Prozesse lassen sich so beschreiben, dass sie in den Aufbau des Tools überführt werden können. Hierfür müssen lediglich Devices, Ports und Methoden identifiziert werden, sowie eine Unterteilung der Aufgabe in Teilaufgaben (Testgroups, Testcases) möglich sein. Dann muss das Grundgerüst im Programm Code selbst angepasst und notwendige Methoden erstellt werden. Bei der Erstellung der Methoden hilfreich ist dabei der Methoden Editor.
Die graphische Oberfläche des *ibb TestDesigners* muss überhaupt nicht angepasst werden, da sich deren Interpretation immer aus den Ports und Methoden ergibt. Das Ziel einer jeden Anwendung ist hierbei das einfache Erzeugen einer strukturierten Ausgabe mittels einer übersichtlichen zweidimensionalen Tabelle.

7 Zusammenfassung & Ausblick

Diese Bachelorarbeit entstand in enger Zusammenarbeit mit dem *Ingenieurbüro Brinkmeyer & Partner*. Das Ziel war eine Beschreibung und darauf aufbauend die Erweiterung der firmeneigenen Software *ibb TestDesigner*.
Zunächst wurden die Grundlagen des Prozesses erläutert, in den der *ibb TestDesigner* eingebettet ist. Dazu gehört die Testumgebungssoftware *CANoe*, die Hardware Komponenten des *VT Systems* und die Programmiersprache *CAPL*.
Anschließend wurde beschrieben, was der *ibb TestDesigner* genau ist, wie das Konzept hinter dem Programm aussieht und wie es eingesetzt wird. Dazu wurden Stück für Stück Teile des Programms benannt und näher erläutert. Anschließend wurde erklärt, wie die Testerstellung mit diesem Tool funktioniert und welche weiteren Funktionen es erfüllt. Außerdem wurde ähnliche Software beschrieben und verglichen.
Im Entwurf der Erweiterung wurde das bereits existierende Ausgabeformat mit dem zu erzeugenden verglichen. Weiterhin wurden Aufbau und Besonderheiten eines *CAPL* Testmoduls aufgezeigt. Es folgte eine Beschreibung der Anforderungen an die Softwareerweiterung selbst und den Arbeitsprozess. Anschließend wurde das Konzept für die Erweiterung vorgestellt und diese auch umgesetzt. Der Softwareentwicklungsprozess orientierte sich hierbei stark an den Prinzipien der Agilen Softwareentwicklung. In regelmäßigen Treffen mit dem Betreuer der Arbeit im Unternehmen, Herrn Dipl.-Ing. (FH) Peter Heidenwag, wurden bisheriger Stand und weiteres Vorgehen abgestimmt. Dadurch konnten unvorhergesehene Schwierigkeiten gelöst werden und es wurde sichergestellt, dass die Software den Ansprüchen von *ibb* gerecht wird.
Die beabsichtigte und korrekte Funktionsweise der Erweiterung wurde anhand eines realistischen Szenarios demonstriert. Dazu wurde eine Hardware- und Software-Testumgebung beschrieben, Requirements für einen Test formuliert und ein Test erstellt, ausgeführt und ausgewertet, der diese Requirements überprüft. Anschließend wurden die anfänglichen Anforderungen an die Erweiterung mit dem Ergebnis der Arbeit abgeglichen.
Zuletzt erfolgte eine Untersuchung und Erklärung, welche weiteren Aufgabengebiete mit dem Konzept des *ibb TestDesigners* erfüllt werden könnten. Hierbei zeigte sich die Vielseitigkeit des allgemein gehaltenen Konzepts mit Devices, einzelnen Ports und deren Methoden.

Ausblick

Zukünftig wird die neue Version des *ibb TestDesigners* auch in der Praxis eingesetzt werden. Neben dem *XML*-Format ist nun ebenfalls die Generierung von *CAPL* Testmodulen möglich, die bei Bedarf mehr Flexibilität und Komplexität für die Testdefinition bieten. Die im Laufe der Arbeit entstandene Methodenbibliothek kann über den Methoden Editor falls nötig einfach um zusätzliche Methoden erweitert werden. Für den Anwender selbst ergeben sich keine nennenswerten Änderungen in Bezug auf den Anspruch bei der Testerstellung. Deshalb wird die Erweiterung des *ibb TestDesigners* in die tägliche Arbeit der Firma Einzug finden und auch den Kunden näher gebracht werden.

Literaturverzeichnis

[BBB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas. Manifesto for Agile Software Development, 2001. URL http://www.agilemanifesto.org/. (Zitiert auf Seite 27)

[Kir97] M. Kirtland. Introducing Visual Studio 97: A Well-stocked Toolbox for Building Distributed Apps, 1997. URL https://www.microsoft.com/msj/0597/visualstudio97.aspx. (Zitiert auf Seite 30)

[Kra09] S. Krauß. Testing with CANoe, 2009. URL http://vector.com/portal/medien/cmc/application_notes/AN-IND-1-002_Testing_with_CANoe.pdf. Seite 19. (Zitiert auf den Seiten 7 und 23)

[MSD15] MSDN. Downloads Visual Studio 2010, 2015. URL https://msdn.microsoft.com/en-us/subscriptions/downloads/default.aspx?pv=18:370#searchTerm=&ProductFamilyId=370&Languages=en&PageSize=10&PageIndex=4&FileId=0. (Zitiert auf Seite 30)

[Rei11] K. Reif. *Bosch Autoelektrik und Autoelektronik*. Vieweg + Teubner Verlag, 6. Auflage, 2011. ISBN: 978-3-8348-1274-2. Seite 12. (Zitiert auf Seite 9)

[RR06] S. Robertson, J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, 2. Auflage, 2006. ISBN: 978-0321419491, Seiten 9-10. (Zitiert auf Seite 26)

[Vec04] Vector CANtech Inc. Programming With CAPL, 2004. URL http://vector.com/portal/medien/vector_cantech/faq/ProgrammingWithCAPL.pdf. (Zitiert auf Seite 12)

[Vec09] Vector CANtech Inc. CANoe Test Feature Set Tutorial, 2009. URL http://vector.com/portal/medien/cmc/application_notes/AN-AND-1-118_CANoe_TFS_Tutorial.pdf. (Zitiert auf Seite 12)

[Vec15a] Vector Informatik GmbH. Company History, 2015. URL http://vector.com/vi_company_history_en.html. (Zitiert auf Seite 11)

[Vec15b] Vector Informatik GmbH. Concept Manual vTESTstudio, 2015. URL http://vector.com/portal/medien/cmc/info/vTESTstudio_ConceptManual_EN.pdf. (Zitiert auf Seite 22)

[Vec15c] Vector Informatik GmbH. News & Events for ECU Testing, 2015. URL http://vector.com/vi_news_ecutest_en.html. (Zitiert auf Seite 22)

[Vec15d] Vector Informatik GmbH. Produktinformation VT System, 2015. URL https://vector.com/portal/medien/cmc/info/VTSystem_ProductInformation_DE.pdf. (Zitiert auf Seite 11)

[Vec15e] Vector Informatik GmbH. VT System - I/O Interface Modules for the ECU Test, 2015. URL http://vector.com/portal/medien/cmc/datasheets/VT_System_DataSheet_DE.pdf. (Zitiert auf Seite 38)

Alle URLs wurden zuletzt am 2. 12. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift