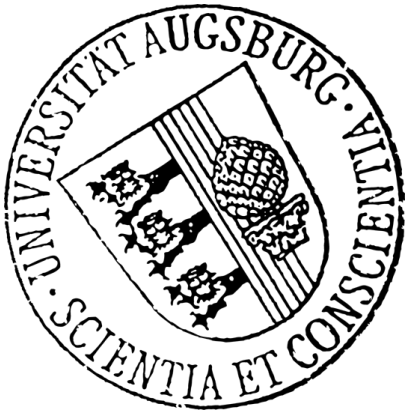
Automatisierung integrierter software-basierter Simulationen von Systemeingaben in verteilten Systemen

Dissertation

zur Erlangung des Doktorgrades
(Dr. rer. nat.)



Andreas Kurtz

Universität Augsburg

Fakultät für Angewandte Informatik
Softwaremethodik für verteilte Systeme

November 2018

Automatisierung integrierter software-basierter Simulationen von Systemeingaben in verteilten Systemen

Erstgutachter:	**Prof. Dr. Bernhard Bauer**, Fakultät für Angewandte Informatik, Universität Augsburg, Deutschland
Zweitgutachter:	**Prof. Dr. Theo Ungerer**, Fakultät für Angewandte Informatik, Universität Augsburg, Deutschland
Tag der Verteidigung:	15 November 2018
Copyright:	

Zusammenfassung - Die stetig steigende Komplexität von Kundenfunktionen aufgrund der wachsenden Herausforderung der vollkommenen Vernetzung von Funktionen erfordert neue software-basierte Methoden um die Qualität der Funktionen sicherzustellen. Die Standardisierung der Software im Automotive Umfeld dient hier als Ankerpunkt und erlaubt eine generische Integration einer software-basierten Methodik in Form eines Standard-Softwaremoduls.

Ziel der Arbeit ist die Entwicklung einer eben solchen software-basierte Methodik, integriert im embedded System, um das Systemmodell bzw. die verteilte Funktion mittels dieser zu testen. Dies wird durch Zuhilfenahme eines modellbasierten Testansatzes von Schieferdecker in den Testing Kontext gebracht. Der verwendete modellbasierte Ansatz ermöglicht die automatisierte Verwendung von Informationen aus dem Deployment und dem Umweltmodell für die Testgenerierung. Dieser zeichnet sich im speziellen durch seinen Angriffspunkt am Fuße der Steuergeräte-Basissoftware bzw. direkt im Treiber der Hardwareschnittstellen aus. Vorteil hierbei ist, dass sich die Anzahl an Signaltypen reduziert aber auch Wirkketten, sich im Vergleich zu bestehenden Ansätzen, verlängern. Durch die Informationen aus dem Deployment, den längeren Wirkketten und der reduzierten Anzahl an Signaltypen, können die erzeugten Testfälle einfach für konventionelle oder auch die in dieser Arbeit entwickelten Methodik, verwendet werden. Durch ein Scheduling innerhalb des eingebetteten Systems, mit dem die Tasks, der Simulation und der Applications synchronisiert werden, kann hierdurch die Reproduzierbarkeit einer Funktionsausführung maßgeblich verbessert werden. Mit Hilfe A-priori und A-posteriori-Analyse wird ermittelt, dass die Auswirkungen auf das embedded System gering sind und keinen wesentlichen Einfluss auf die Funktionsausführung hat. Durch dem Einsatz etablierter Methoden für die Systemreaktionsbewertung zeigt sich anhand von Fallstudien der enorme Vorteil durch die eingeführte Synchronisation der Simulations- und Application-Tasks. In Kombination mit Methoden des modellbasierten Testens schafft der dargestellte Ansatz daher einen enorm hohen Automatisierungsgrad im Bezug auf das Testing bei gleichzeitig minimalem Eingriff in das System under Test. Die Methodik zeichnet sich insbesondere dadurch aus, dass eine exakt vergleichbare Funktionsausführung zum normalen Kundennutzen, sowie eine exakte Reproduzierbarkeit innerhalb des embedded Systems, erreicht werden kann.

Im Rahmen der Validierung bestätigt sich das erwartete Verhalten der entwickelten Methodik bezüglich ihres Zeitverhaltens. Es zeigt sich auch, dass der Einfluss auf die Ressourcen des embedded Systems je nach Anwendungs- bzw. Testfall ausfällt und Grenzen bei der Anwendung einer Simulation im verteilten System, bei nicht zeit-synchronisierter Kommunikation, auftreten. Letztendlich überwiegen die Vorteile durch die Anwendung des modellbasierten Ansatzes mit der Einschränkung, dass manche Funktionsreaktionen im verteilten System nicht mehr deterministisch sind, aber dem Kundennutzen sehr nahe kommen. Für den Einsatz der Methode spricht, dass viele Szenarien einfach, mit wiederverwendbaren Testmodellen reproduziert werden können, die mit heutigen Methoden nur mit hohem Aufwand darstellbar sind.

Abstract - Increasing compexity in customer functions due to the increasing challenge of networking all functions in the automotive domain demands new software-based methods to ensure the quality of customer functions. Standardisation of software in the automotive Domain is an enabler for an generic integration of a methodology in shape of a standard software module.

The objective of the work is the development of a software-based methodology, integrated in the embedded system to test the system model or the distributed function by means of this. This is brought into the testing context by using a model-based test approach of Schieferdecker. The model-based approach that is used enables the automated use of information from deployment and the environmental model for test generation for the developed approach. This is characterised in particular by its point of attack at the bottom of the basic software respectively directly in the driver of the hardware interfaces. The advantage here is that the number of signal types is reduced, but also functional chains are extended compared to existing approaches. Due to the information from the deployment, the longer functional chains and the reduced number of signal types, the generated test cases can easily be used for conventional or the methods developed in this work. Scheduling within the embedded system, with which the tasks of application and simulation are synchronized, significantly improves the reproducibility of function execution. With the help of A-priori and A-posteriori analysis, it is determined that the effects on the embedded system are small and have no significant influence on the function execution. The use of established methods for system reaction evaluation based on case studies shows the enormous advantage of the introduced synchronization of simulation and application tasks. In combination with methods of model-based testing, the approach illustrated here creates an extremely high degree of automation at testing with minimal intervention in the system under test. The methodology characterizes itself in such a way that an exactly comparable function execution for the normal customer benefit as well as an exact reproducibility within the embedded system can be achieved.

The validation confirms the expected behaviour of the developed methodology with regard to its time behavior. It also shows that the influence on the resources of the embedded system is dependent on the application or test case and limits occur when using a simulation in a distributed system, with non-time-synchronized communication. In the end, the advantages outweigh the advantages of using the model-based approach with the restriction that some function reactions in the distributed system are no longer deterministic, but come very close to the customer's benefit. The fact that many scenarios can be easily reproduced with reusable test models, which can only be reproduced with today's methods with great effort, speaks for the use of this method.

„I am not young enough to know everything.“

— Oscar Wilde

An dieser Stelle möchte ich meinen besonderen Dank nachstehenden Personen entgegen bringen, ohne deren Mithilfe die Anfertigung dieser Promotionsschrift so nicht zustande gekommen wäre.

Mein Dank gilt zuallererst Herrn Prof. Dr. Bernhard Bauer, meinem Doktorvater, für die Chance, die er mir gegeben hat, und die Betreuung während der letzten Jahre. Die zahlreichen fachlichen Gespräche, der rege Ideenaustausch, sowie Gespräche auf persönlicher Ebene haben mir immer wieder Ansporn gegeben und die Freude an dieser Arbeit erhalten.

Ich danke Herrn Prof. Dr. Ungerer für die hilfsbereite und wissenschaftliche Betreuung als Zweitgutachter.

Danken möchte ich auch meinen Mitdoktoranden am Lehrstuhl, die mich und meine Forschung während meiner Zeit an der Universität Augsburg in den fachlichen Diskussionen auf den zahlreichen Workshops kritisch hinterfragt und somit maßgeblich zu dem Ergebnis beigetragen haben. Zudem danke ich allen Studenten, die in der Zeit ihren Beitrag geleistet haben, um diese Arbeit zu diesem Ergebnis zu bringen.

Besonderer Dank geht auch an meinen Bruder Stefan Kurtz, der in den Jahren immer wieder ein offenes Ohr hatte und mit mir über die Themen meiner Arbeit diskutiert hat. Danke für die persönliche Unterstützung, die Motivation und den Zuspruch über die ganzen Jahre hinweg.

Außerordentlichen Dank verdient Doris Weiß, die ihre kostbare Zeit geopfert hat, meinen - wie sie sagt - „Agenten-Roman“ zu lesen. Sie unterstützte mich bei der sprachlichen Korrektur der Arbeit dafür möchte ich ihr besonders danken.

Inhaltsverzeichnis

1. Einleitung

„Vernetzung prägt unsere Zeit und dominiert die Zukunft.“
— Norbert Reithofer, BMW Geschäftsbericht 2014 [BMWb]

Die Vernetzung des Automobils mit der Umwelt beeinflusst die Entwicklung aktuell stärker als die Entwicklung alternativer Antriebe [BMWb]. Immer mehr Wettbewerber und neue Zulieferer suchen den Zugang zum Automobil z.B. über die Schnittstelle Internet. Die Eigenschaften von Automobilen werden heutzutage in großen Teilen von Software bestimmt. 100 Millionen Lines of Code (LOC), tausende Funktionen machen Software zum Innovationstreiber mit einem Anteil von ca. 90 % der Innovationen im Bereich Elektrik Elektronik (E/E). Softwareentwicklung als Ganzes entwickelt sich immer mehr zu einem industriellen Teilgebiet. Die zunehmende Standardisierung von Entwicklungsmethoden und Betriebsprozessen sowie die Verbreitung von Standardprodukten (Baukasten) sind ein wesentlicher Treiber für die Entwicklung effizienter Methoden. Dies betrifft im Besonderen den Bereich des Softwaretesting, als auch die Testautomatisierungsmethoden. Diese Automatisierung beruht aufgrund der Auswirkungen bei Ausfall eines Teilsystems auf dem deutlich höheren Stellenwert der Softwarequalität im Automobil im Gegensatz zur Consumer Elektronik Industrie.

Im Rahmen der fortschreitenden Industrialisierung wurden zunehmends zeitraubende manuelle Tätigkeiten durch Standardisierung soweit angeglichen, dass Automatisierungsmethoden entwickelt werden können. Der Grundgedanke erhebt hier den Anspruch, der Steigerung der Produktivität und Qualität bei Reduktion von Arbeitseinsatz und Zeitaufwand. Im Bestreben dem globalen Konkurrenzdruck bestehen zu können, werden zunehmend Ansätze entwickelt, um die Produktionsmenge, Qualität und Zeit mit dem geringst möglichen Einsatz zu erreichen. Vorbild ist oft die Automobilindustrie, die immer wieder neue Ansätze zur Prozesssteuerung und Produktionsgestaltung oder im Qualitätsmanagement hervorbringt. Höchste Präzision zwischen Mensch und Maschine im Zusammenspiel bei der Produktion und dem Betrieb sind erforderlich, um einen reibungslosen Ablauf zu gewährleisten.

Was Produktionsabläufe angelangt, ist die Automobilindustrie bereits heute auf einem Stand der primär nur noch Potential im Refinement übrig lässt. Betrachtet man im Gegensatz zu den Produktionsabläufen die Softwareentwicklung im Automotive-Bereich, sind enorme Potentiale aufzudecken. Softwareentwicklung im Automotive-Umfeld ist aktuell eine junge Disziplin und wird eher als Einzelfertigung [SBB11] angesehen. Neue Wettbewerber im Automotive-Sektor können im Software-Entwicklungsumfeld deutlich bessere Erfahrungen aufzeigen und liegen dadurch im Vorteil in einer von Software getriebenen Welt.

Besondere Herausforderungen bei der Fahrzeugentwicklung entstehen in der Integration von immer neueren Funktionen unter dem Anspruch und Wunsch nach mehr Sicherheit, Komfort und Vernetzung von bestehenden oder neu entwickelten Informationstechnik-(IT)-Systemen. Besonders Letztere stellen eine Kernherausforderung dar, da die IT-Entwicklung einen deutlich schnelleren Entwicklungs- und Aktualisierungszyklus (Time to Market) hat als eine Fahrzeugarchitektur.

> „Wir aktualisieren das Fahrzeug im Rhythmus der Consumer Electronics“
>
> — Elmar Frickenstein, [ATZ13]

Dieses Zitat lässt auf den Aufwand schließen der sich in der Automotive Branche einstellt. Um dieser Masse an neuen Funktionen und der Komplexität Herr zu werden sind zunehmend neue Methoden erforderlich, um hier eine Basis für intelligentes automatisiertes Testen zu schaffen.

Grundlegend darf hier nicht vergessen werden, dass sich die Fahrzeugentwicklung nicht mehr nur auf den Bereich – es wird ein neues Fahrzeug entwickelt –, sondern auf die Entwicklung von End-to-End Mobilitätsdienstleistungen und Angeboten mit der Vernetzung in bestehende Systeme konzentriert. Somit sind in Zukunft immer mehr Partnerschaften mit z.B. Cloud-Plattformanbietern oder Providern erforderlich, da ein eigenständiges Aufbauen solcher Systeme nicht profitabel ist.

1.1. Problemstellung und Herausforderungen

— Prozess — In der Fahrzeugentwicklung wird streng nach Prozessvorgaben vorgegangen. Grundlegend steht, im Fall der BMW Group, neben den produktionsspezifischen Prozessen wie z.B. „Offer to Order“ oder „Order to Delivery“ der Produktentstehungsprozess „Idea to Offer“ (ItO), aus dem Englischen übersetzt „Von der Idee zum Angebot“, im Vordergrund und definiert alle Phasen der Entwicklung. Dieser Prozess kann auf alle notwendigen Abläufe in der Entwicklung, wie z.B. Sofwaretesting, fahrdynamische Untersuchungen und Weiteres, aufgegliedert werden. Die Anforderungen an das Angebot entstehen unter anderem aus gesetzlichen Vorgaben, Kunden- und Umgebungsanforderungen und werden letztendlich in Lastenheften zusammengefasst und dokumentiert. Diese Lastenhefte dienen in Folge als Zielvorgabe für die Entwicklung und sollen lösungsneutrale Beschreibungen bzw. Spezifikationen enthalten.

Das weitere Vorgehen in der Fahrzeugentwicklung ist teilweise sehr unterschiedlich, angestrebt wird jedoch ein modellbasierter Entwicklungsansatz. Der modellbasierte Ansatz sieht vor, dass bei der Erstellung des Systemmodells parallel ein Testmodell aufgebaut wird, welches

zur Absicherung oder besser gesagt zum Testen des Systemmodells dienen wird. Aufgrund der hohen Varianz von genutzten Softwareentwicklungsplattformen fehlen Standards in der Entwicklung. Die entwickelnden Fachbereiche leben entwicklungsspezifische Prozesse, um die Anforderungen aus den Lastenheften umzusetzen.

Bei genauerer Betrachtung der Entwicklung fällt auf, dass System- und Testmodell in der Regel nur als mentales Modell verfügbar sind. Das ist meist dem enorm hohen Zeitdruck in der Fahrzeugentwicklung geschuldet, der dazu führt, dass ein Mangel an Zeit für schematische Konzepterstellung und Dokumentation vorherrscht. Hinzu kommen, was letztendlich in Zeitdruck endet, späte Vergaben an Entwicklungspartner, spät entschiedene Reaktionen auf Marktänderungen die folglich in Anforderungs- und Funktionsanpassungen enden. Neben dem Zeitdruck übersteigt auch die Komplexität aktueller und zukünftiger Systeme oft die menschliche Auffassungsgabe. Diese Komplexität führt wiederum in verschiedene Arten bzw. Methoden der Erstellung von Test- oder Systemmodellen. Auch sind aufgrund des fehlenden Einsatzes keine Standards entstanden mit denen Modellelemente z.B. wiederverwendbar wären. Eine durchgängige modellbasierte Entwicklung soll die Komplexität auflösen, bzw. mit der Hilfe von IT-Systemen verständlich machen.

— Methodisch — Agile Methoden halten Schritt für Schritt Einzug in die Fahrzeugentwicklung [Ger14] und sind dem starken Einfluss von Software unter immer kürzeren Entwicklungszeiten zu verdanken. Agile Methoden sind im Bereich der IT-Entwicklung bereits lange Standard. Ein wesentlicher Vorteil ist die schnellere Reaktionsfähigkeit auf Änderungen im Umfeld. Unternehmen mit agilen Methoden können die Chancen besser nutzen, die sich im Umfeld ergeben. Die Automobilindustrie beugt sich hier dem enormen Einfluss der Consumer Elektronik Branche, um Schritt halten zu können und um dem Kunden immer ein aktuelles Produkt anbieten zu können.

Modellbasiertes Entwickeln ist im Automotive Bereich ein Enabler für Agilität. Nur durch Modelle und den Einsatz von IT-Systemen kann die Komplexität aufgelöst werden. Durch den richtigen Einsatz von Modellen können agile Methoden angewendet werden und somit in kurzen Zyklen immer ein lauffähiges Produkt für das Testing bereit stehen. Die Automobilbranche hat den Wandel zum aktuellen Zeitpunkt vor sich. Themen und Begriffe wie agile Software-Entwicklung, Scrum, Continious Integration beherrschen die Prozess-Fachstellen. [Web15] Im Rahmen dieser Umstellung entstehen enorme Herausforderungen und viele, schon seit Jahren bestehende Methoden müssen komplett neu durchdacht werden.

— Technisch — Eine Vielzahl an Fahrzeugderivaten und Varianten beherrscht aktuell die Entwicklung. Individualisierung ist das Stichwort, welches die Kunden einfordern. Individualisierung bedeutet einerseits, dass der Kunde ein Fahrzeug mit genau der Funktionskombination haben will, die er fordert, andererseits soll das Produkt eine Vernetzung mit all den im Markt erhältlichen IT-Systemen ermöglichen. Hierbei entwickelt sich aufgrund der Individualisierungsmöglichkeit des Produkts eine enorme Vielfalt und Komplexität von Systemen. Die Kernherausforderung ist die Rückwirkungsfreiheit von Funktionen in jeglicher Kombination, d.h. jede (Kunden-) Funktion darf keine negativen Auswirkungen auf die Robustheit, die Stabilität und die Sicherheit des gesamten Systems haben.

Vernetzung von Funktionen führt zu vielen Vorteilen bezüglich erweiterten Funktionalitäten, hat aber auch den Nachteil zu nicht-linearen Auswirkungen. Einerseits können kleine Fehlfunktionen enorme Auswirkungen an ganz anderen Stellen im Netzwerk haben und andererseits können Reaktionen möglicherweise zeitverzögert auftreten. Es können sich vollkommen neue Fehlerbilder ergeben, die unvorhergesehen auftreten und auf lange Ursache-Wirkungsketten zurückzuführen sind. Besonders zeitverzögerte Wirkungen können oft nicht mit ihrer Ursache in Verbindung gebracht werden. Zusammengefasst stellt das die Absicherung vor ein enormes Komplexitätsproblem und erfordert neue innovative und software-basierte Methoden.

Technisch darf das System nicht mehr nur als das Fahrzeug betrachtet werden, sondern muss End-to-End aus Sicht des Kunden betrachtet werden: Vom Fahrzeug bis zum mobilen Device oder der Smart Home Integration, bei dem Infrastruktur-, Backend- oder auch andere Fahrzeug-Systeme ihre Berücksichtigung finden müssen. Die Verantwortung der fehlerfreien und rückwirkungsfreien Funktionalität liegt aus Sicht des Kunden in der Hand des Herstellers des primär genutzten Systems. Fehler werden in erster Linie dem offensichtlich genutzten Produkt zugeschrieben und nicht der tatsächlichen Fehlerursache. Dies bringt die Automobilindustrie in die Situation, dass auch Fremdsysteme Teil der Betrachtung werden und das hierdurch technisch eine enorme Herausforderung entsteht, dem Kunden ein sicheres und robustes Produkt bzw. eine gute Dienstleistung zu liefern.

1.2. Motivation

„Suche nicht nach Fehlern, suche nach Lösungen.“

— Henry Ford

Aus den in Kapitel 1.1 genannten Herausforderungen in den Bereichen Prozess, Methodik und Technik können diverse Handlungsfelder abgeleitet werden. Im Rahmen dieser Arbeit wird auf die folgenden Aspekte im Bezug auf das Testing von Software in verteilten Systemen genauer eingegangen. Die folgende Einordnung erfolgt ohne Prioritäten.

Komplexität
: von *verteilten Funktionen*. Diese entsteht unter anderem aus der Herangehensweise bei der Systementwicklung durch die verwendete Systemarchitektur und stellt damit einen Bezug zu einem methodischen Problem dar. Die Komplexität spiegelt sich dadurch auch in den Tests für die komplexen und verteilten Funktionen wider.

Automatisierung
: von *manuellen wiederkehrenden Tests*. Automatisierung ist ein Ansatz für die Lösung von Zeitproblemen und bezieht sich hier auf die Herausforderungen im Prozess. Sowohl auf der Entwicklungsseite als auch beim Testing können Prozesse oder Prozessschritte automatisiert werde.

Qualität — der *manuell durchgeführten Tests*. Die Qualität von Tests bezüglich ihrer Durchführung und der Ergebnisse hängt von mehreren Faktoren ab und kann in diesem Punkt auf technische und methodische Herausforderungen für den Ansatz verknüpft werden.

Das Ziel der vorliegenden Arbeit ist die Entwicklung einer Methode für die „Automatisierung von Tests in verteilten Systemen“ mit dem Fokus auf die automobile Funktions- und Softwareentwicklung bzw. den Test der entwickelten Funktionen, die großteils durch Software realisiert sind. Hierbei werden die oben genannten Probleme aufgegriffen und über einen einheitlichen innovativen software-basierten Ansatz angegangen. Eine neu entwickelte Methode soll hier Potentiale in Bezug auf die genannten Herausforderungen aufzeigen und eine zudem kostengünstige, da software-basierte Lösung darstellen und sich möglichst nahtlos in die Prozesse der Produktentwicklung einfügen.

Die aufgeführten Begriffe *Komplexität*, *Automatisierung* und *Qualität* werden im Nachfolgenden bezüglich ihrer Definitionen beleuchtet und in Bezug zu der in der Automotive Softwareentwicklung und dem modellbasierten Testen (MBT) gesetzt.

1.2.1. Komplexität - Verteilte Funktionen -

Komplexität (lat. complexum) bezeichnet das Verhalten eines Systems und Wechselwirkungen dessen Komponenten untereinander. [Joh13] Der Begriff der Komplexität wird je nach Wissenschaftsgebiet unterschiedlich definiert. Im Fachgebiet der Informatik wird der Begriff grundlegend in zwei Bereiche unterschieden, die

Komplexität von Algorithmen — Unter Komplexität von Algorithmen versteht man den maximalen Ressourcenbedarf dieser. Der Ressourcenbedarf gliedert sich in die drei Elemente *Rechenzeit*, *Speicherplatz* und *Datentransfer*. Maßgebendes Kriterium im Bereich der Automotive Softwareentwicklung ist die *Rechen-* bzw. *Laufzeit* der Algorithmen bzw. Funktionen.

Komplexität von Daten — Unter Komplexität von Daten oder Nachrichten versteht man ihren Informationsgehalt. Kurz gesagt, wie viel Information steckt in den Daten.

Die Beherrschung von Komplexität fokussiert auf die Algorithmen, bzw. die Funktionen. Komplexität beschreibt die Menge an Sub-Funktionen zur Darstellung einer einfachen Kundenfunktion. Ein Beispiel verdeutlicht den Begriff Komplexität - Die Kundenfunktion Scheibenwischer konventionell gelöst mittels einer Mechanik und innovativ gelöst durch den Entfall der Mechanik mit Hilfe verteilter Software-Funktionalität.

Konventionell Das Mensch Maschine Interface (MMI) bzw. Human Machine Interface (HMI) zur Aktivierung des Scheibenwischers ist der s.g. Lenkstockschalter. Dieser Schalter ist mit einem Steuergerät verbunden, welches den Scheibenwischermotor aktiviert und hierdurch über einen Exzenter eine Mechanik den Scheibenwischer bewegt.

Innovativ Durch den Entfall der Mechanik wird jeder Scheibenwischerarm durch einen einzelnen Motor (inkl. Lagesensor) angetrieben. Jedes der Motormodule beinhaltet einen Elektronikanteil mit Software zur Ansteuerung des Motors. Diese Softwarekomponenten stellen durch ihre Softwarefunktion bzw. Regelung, ein Wischverhalten analog zu einem mechanischen Wischer dar. Dabei muss die Software z.B. die Wischgrenzen kennen, da hier keine Mechanik die Wischbewegung definiert bzw. begrenzt. Hinzu kommt, dass der Lenkstockschalter am Schaltzentrum Lenksäule (SZL) über ein weiteres Steuergerät ausgewertet wird und mit den einzelnen Motormodulen z.B. über einen BUS kommuniziert, um letztendlich die Funktion Scheibenwischer darzustellen. Für den Kunden stellt sich ein gewohntes Wischverhalten dar, welches in erster Linie nicht von der konventionellen Lösung zu unterscheiden ist.

In der Entwicklung wird bewusst eine Erhöhung der Komplexität von Funktionen und den zugehörigen Tests akzeptiert um innovative Kundenfunktionen realisieren zu können. Vorteile, die sich durch den Einsatz von verteilten Softwarefunktionen ergeben, sind unter anderem in den folgenden Bereichen:

- Energie – Reduzierter Energieverbrauch der Funktionen durch den Einsatz neuer Technologien und optimierter Ansteuerung sowie durch den verringerten Einsatz von Mechanik.
- Gewicht – Verringerung des Gewichts durch den Entfall von Mechanik und Optimierung von elektrischen Komponenten.
- Funktion – Erweiterung der Funktionen durch die Anbindung an Bussysteme, dadurch können Funktionen untereinander verknüpft werden.
- Kosten – Reduzierung von Kosten durch die Vereinfachung von Funktionen und die Möglichkeit der Wiederverwendung von Funktionsteilen und Komponenten.
- Robustheit – Erhöhung der Funktionsrobustheit durch den Entfall von Mechanik sowie durch den Einsatz von Software Fallback-Lösungen.

Nachteile durch den Einsatz von verteilten Funktionen sind unter anderem:

- Komplexität – Erhöhte Komplexität des gesamten Systems aufgrund der verteilten Architektur der Funktionen.

- ▷ Fehler – Erhöhte Fehlerquellen in den Software-Funktionen aufgrund des erhöhten Softwareeinsatzes und der bereits aufgeführten Komplexität.

- ▷ Kosten – Erhöhung der Kosten durch erhöhte Aufwände im Bereich Testing der komplexen Funktionen sowie der Variantenvielfalt bei Wiederverwendung.

Unter Betrachtung aller genannter Aspekte überwiegen die Vorteile, besonders in puncto Kosten durch die Wiederverwendung der Software Komponenten der verteilten Funktionen. Aus diesem Grund wird bereitwillig die Komplexität durch die Verteilung von Funktionen erhöht. Folgen dieses Komplexitätsanstiegs sind enorme Mehraufwände im Bereich des Testing der Gesamtfunktion, durch mehr Schnittstellen innerhalb der Gesamtfunktion und mehr Einflussfaktoren auf die Funktionsbestandteile. Das angeführte Beispiel soll hier zeigen, was sich durch den Einsatz von verteilten Funktionen erreichen lässt, sowie auf die Vor- und Nachteile, aber auch den damit verbundenen Mehraufwand verweisen. Die Vor- und Nachteile aus diesem anschaulichen Beispiel lassen sich auf nahezu alle Funktionen im Fahrzeug übertragen.

In der Regel sind der Kostenfaktor, also Kostenersparnis durch Wiederverwendung, und die funktionellen Möglichkeiten die treibende Kraft für den Einsatz von verteilten Funktionen und Software. Unvermeidbar ist hierbei der Anstieg der Komplexität der Systeme, da diese modulare Funktionsarchitektur dazu führt, dass zunehmend mit dem Ziel der Funktionsmaximierung vernetzt wird. Modulare vernetzte Software führt im Anschluss zu mehr Schnittstellen und dadurch erhöhten Testaufwänden, damit wieder zu einem Kostenanstieg, welchem man mit ebenso modularen bzw. modellbasierten Tests entgegenwirkt.

Ziel ist die Entwicklung einer Methode zur Beherrschung der Komplexität der Tests von verteilten Funktionen in der Funktions-Absicherung für die Anwendung in der Automotive Industrie. Gesucht wird ein standardisierbarer einheitlicher Ansatz der an der richtigen Testebene interagiert. Fokus liegt hierbei die Software bzw. verteilte Funktion auf der Zielhardware zu testen, um die bestmögliche Aussage über die Softwarequalität machen zu können.

1.2.2. Automatisierung - wiederkehrende Tests -

Automatisierung aus dem griechischen *automatia - die von selbst kommende* übersetzt, steht für die Übertragung von manuellen Tätigkeiten vom Menschen auf Maschinen. Die Deutsche Industrie Norm (DIN) V 19233 definiert Automatisierung als „Das Ausrüsten einer Einrichtung, so dass sie ganz oder teilweise ohne Mitwirkung des Menschen bestimmungsgemäß arbeitet“ [Deu07]. Ein zentrales Charakteristikum ist auch, komplexe Aufgaben wiederkehrend mit gleichbleibender Qualität ausführen zu können. Das Kriterium gleichbleibender Qualität ist mit dem Einsatz technischer Systeme im Gegensatz zum Einsatz von Menschen in der Regel leichter zu realisieren.

Gründe, die für Automatisierung sprechen sind [SBB11]:

- Qualität – Verbesserung der Produktqualität durch intensivere, tiefgehendere Testmöglichkeiten sowie kontinuierliche exaktere Arbeitsabläufe. Ein erhöhtes Gleichmaß der Produktqualität durch gleichmäßige wiederkehrende Abläufe.

- Quantität – Erhöhung der Produktionsmenge durch z.B. Reduktion von Unterbrechungen.

- Kosten – Einsparung von Kosten durch die Reduktion von Personal sowie Wiederverwendung von Prozessschritten.

- Zeit – Reduktion von Produktionszyklen durch die Verringerung von Fehlproduktionen sowie einer skalierbaren Produktion.

In der Automobilbranche, im Speziellen im Bereich der Software-/ Funktionsentwicklung, ergeben sich in erster Linie Automatisierungspotentiale in der Codegenerierung und im Testing. Im Testing können die o.g. Gründe für Automatisierung weitestgehend adaptiert werden – Automatisierung ermöglicht ein gleichmäßig wiederkehrendes Testen von Funktionen und ohne großen Personalaufwand, 24 Stunden am Tag sieben Tage die Woche.

Ziel ist es, das Testen von Funktionen, in der Regel Softwarefunktionen, möglichst vollständig zu automatisieren. Herausforderungen sind hierbei die Funktionsstimulation und die Bewertung der Funktions-/ Systemreaktion.

1.2.3. Qualität - der manuellen Tests -

Im Allgemeinen ist die Produktqualität definiert als das Maß für die Übereinstimmung von Leistung mit den gesetzten Ansprüchen. Qualität lässt sich auch aufteilen in spezifischere Subqualitäten wie die funktionale Qualität, technische Qualität oder auch ökologische Qualität. Hierbei beziehen sich die Qualitäten auf spezifischere Ansprüche, dies hilft bei der Differenzierung von Eigenschaften.

Im Automotive Umfeld spielt die funktionale Qualität eine entscheidende Rolle und ist für die meisten Kunden diejenige, auf die sie am meisten Wert legen, insbesondere auch vor Aspekten wie z.B. ökologische Qualität.

Die Qualität eines Produkts, in diesem Fall auch von Softwarefunktionen, kann durch den Einsatz der richtigen Testmethoden gesteigert werden. Der Einsatz von technischen Systemen bringt entscheidende Vorteile, um Funktionen besser testen zu können. Durch den Einsatz der richtigen Methoden können die Testergebnisse eine höhere Aussagequalität erreichen und somit bessere Aussagen über die Funktionsqualität erzeugen. Die Aussagequalität bezieht

sich hierbei auf den Vergleich zweier oder mehrerer Tests, entweder bei Weiterentwicklung oder Bugfixing der Funktion.

Ziel ist daher die Erhöhung der Aussagequalität von Tests zur Verbesserung der Funktionsqualität durch die Reduktion von manuellen System-/ Funktionseingaben, um die System-/ Funktionsreaktion besser bewerten zu können.

1.3. Ziele, Ansätze und Beiträge

1.3.1. Qualität der durchgeführten Tests

Eine grundlegende Herausforderung im Bereich Testen von software-basierten Funktionen besteht darin, dass die Reproduzierbarkeit von Soll- oder Fehlverhalten durch Tests nahezu nicht darstellbar ist. Ein Grund hierfür ist, dass sich ein Systemzustand besonders durch die große Anzahl externer Einflüsse oder konzeptioneller Umsetzungen in der Software nur schwer wieder in den exakt identischen Zustand bringen lässt, der beim ursprünglichen Testablauf vorherrschte.

Externe Einflüsse sind z.B. Umgebungs- bzw. Umwelteinflüsse, inkl. dem Kunden, in denen sich das System aktuell befindet. Erkannt werden diese Umgebungseinflüsse in erster Linie durch die Sensoren des System under Test (SUT). Eine reproduzierbare Systemeingabe lässt sich hierbei nahezu nicht zuverlässig wiederherstellen. Folgendes Beispiel verdeutlicht die Schwierigkeiten der reproduzierten Systemeingabe: Drückt der Tester einen Taster im Fahrzeug, um ein Funktion zu aktivieren, kann dieser Tastendruck einerseits kaum durch den Tester in exakt gleichem Verhalten, speziell dem Zeitverhalten, reproduziert werden. Andererseits kann der Tester nicht erkenne,n ob zum Zeitpunkt der Funktionsaktivierung, also des Tastendrucks, das SUT im ursprünglichen Zustand vorliegt. Dieser Zustand meint im speziellen den Taskzyklus des Steuergeräts, der durch den Tester normalerweise nicht einsehbar ist. Hierdurch erzeugt der Tester bzw. Nutzer des Systems immer wieder einen neuen Gesamtzustand, respektive ein neues Verhalten des SUT. Zwar werden die Eingaben des Systems diskretisiert, aber Eingaben könnten in unterschiedlichen Taskzyklen einer Applikation erfolgen oder das Steuergerät hat bei einer Wiederholung aktuell einen anderen Task mit höherer Priorisierung abzuarbeiten.

Unter diesen Aspekten ergibt sich die Fragestellung, wie das System reproduzierbar stimuliert werden kann. Das Ziel ist hierbei die Verbesserung der Testqualität respektive der Reproduzierbarkeit von Tests durch einen software-basierten Ansatz im SUT mit Hilfe von SUT internen Informationen wie z.B. dem aktuellen Taskzyklus.

Die angestrebte Lösung ist somit ein software-basierter Ansatz integriert im SUT selbst. Dieser eröffnet das Potential, ein Verhalten eines Signals exakt reproduzieren zu können und reduziert damit den Einfluss externer Einflüsse auf das SUT. Dies heißt einerseits, durch die

software-basierte Lösung kann das Zeitverhalten z.B. eines Tastendrucks wiederholt werden. Andererseits kann durch eine Integration im SUT eine Synchronisation auf Tasks z.B. von Applikationen erfolgen.

Ein besonderer Vorteil ergibt sich besonders bei der Virtualisierung von Analogsensorwerten und Funktionen mit vielen Eingangs- bzw. Sensorwerten, da eine software-basierte Lösung eine schnelle und unkomplizierte Wiederholung der Sensorwerte ermöglicht, welche sonst stark von Umweltbedingungen abhängig sind.

1.3.2. SW-Basierter Ansatz versus realistische Wirkketten

Der Lösungsansatz der software-basierten Virtualisierung von Systemeingaben im SUT führt zu der Frage, wie die Wirkketten der Funktionen möglichst realistisch getestet werden können. Eine Wirkkette beschreibt hierbei immer die gesamte Softwarefunktion von der Sensorik bis zur Aktorik der genutzten Funktion. Die Betrachtung im Rahmen der vorliegenden Arbeit fokussiert auf software-basierte Funktionen sowie verteilte Funktionen. Folgend gilt umso länger die getestete Wirkkette der Funktion der Wirkkette der gesamten Funktion entspricht, also von Sensor bis zum Aktor, umso realistischer ist die Aussage des Tests bezüglich des Funktionsverhaltens im Kunden Use Case. Daraus ergibt sich in erster Linie das Ziel der möglichst hardwarenahen Simulation von Sensorwerten zur Maximierung der Wirkkette im Steuergerät bzw. verteilten System.

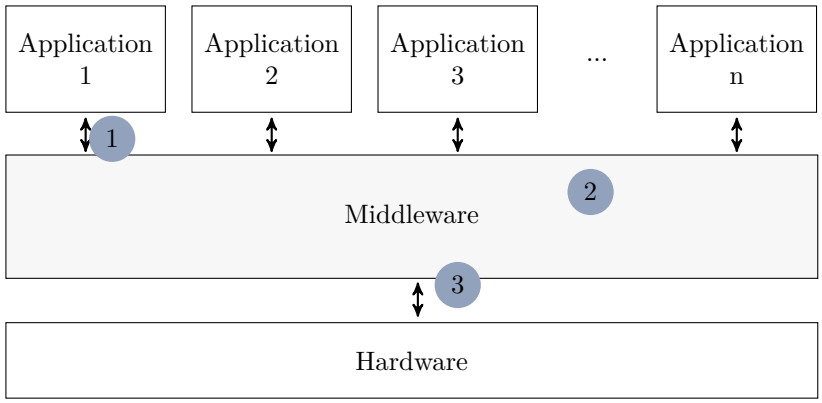


Abb. 1.1.: Interaktionsmöglichkeiten Softwareebenen

Die Abbildung 1.1 zeigt am Beispiel einer einfachen Architektur, an welchen Punkten grundlegend eine Virtualisierung von Signalen in der Software erfolgen könnte ohne die zu testende Funktion an sich zu verändern.

(1) ist die Interaktion an der Schnittstelle zwischen Middleware und der Application, in der die betroffene Funktion agiert bzw. ihre Eingangssignalinformationen aus den unteren Schichten bezieht.

(2) ist die Interaktion innerhalb der Middleware, mit dem Ergebnis, dass hierdurch alle Applikationen bzw. Funktionen über der Middleware angesprochen werden. Dies verringert den Unterschied zu dem normalen Kundenverhalten.

(3) ist die Interaktion unterhalb der Middleware, bzw. so nah wie möglich an der Hardware-Schicht also in den Hardwaretreibern.

Untersucht werden die genannten Interaktionspunkte im Kontext Automotive Open System Architecture (AUTOSAR) 4. Die Abbildung 1.2 zeigt die Interaktionsmöglichkeiten aus Abbildung 1.1 im AUTOSAR-Schichtenmodell (AUTOSAR siehe Kapitel 2.4.3).

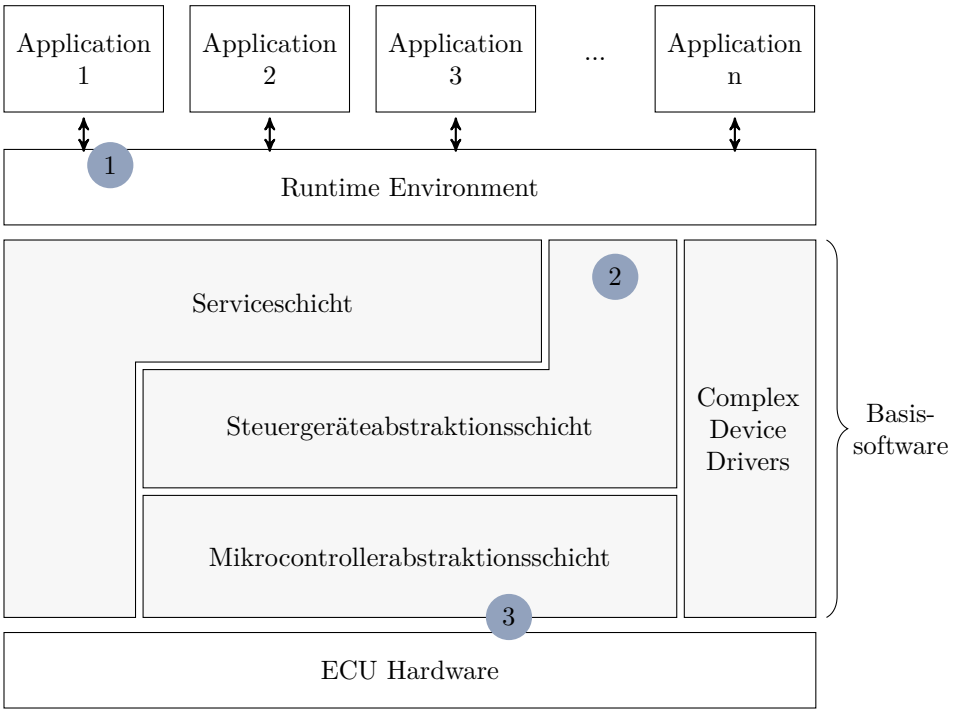


Abb. 1.2.: Interaktionsmöglichkeiten Softwareebenen - AUTOSAR

Untersuchungen, die Schicht für Schicht Wechselwirkungen, Vor- und Nachteile aufzeigen, die bezogen auf den Interaktionspunkt (1) entstehen können, sind im Rahmen der Arbeit „SW-Basierte Kundenfunktionseingabe für die Automatisierung“ [Prö14] untersucht worden. Fokus lag hierbei auf den Wechselwirkungen in der Application- und SUT-Komponente, wenn Signalwerte der Hardwareschicht in einer applikationsnahen Schicht stimuliert wurden.

Durch weitere Untersuchungen der Interaktionspunkte werden Vor- und Nachteile des einzelnen identifiziert, und bei der angestrebten Lösung mit dem Ziel Interaktionspunkt (3) berücksichtigt. Durch die Verschiebung des Interaktionspunkts in Richtung Hardware entsteht eine Generalisierung der Signaltypen, da Ebenen wie eine Steuergeräte-Abstraktion oder Signal-Abstraktion entfallen und man sich bei den Treibermodulen auf wenige standardisierte Typen beschränkt.

1.3.3. Realistische Wirkketten versus software-basierte Simulation

Realistische Wirkketten, wie in Kapitel 1.3.2 beschrieben, erfordern die Stimulation des Systems mit Kontextinformationen und Verhalten aus der Umgebung, inkl. der auch bereits in Kapitel 1.3.1 erwähnten externen Einflüsse bzw. Kundeneinfluss. Die Abbildung 1.3 zeigt das SUT im Kontext sowie den neutralen Beobachter des hier genannten Observer, der das Systemverhalten im Kontext bewerten soll.

Durch eine neu geschaffene Schnittstelle im SUT kann genau diese Kontextinformation sowie das Kontextverhalten in das SUT eingespeist werden. Dies ermöglicht den gezielten Ausschluss von unerwünschten Einflüssen sowie den Gewinn von zusätzlichen internen Informationen des SUT.

Die Lösung ist ein modellbasierter Ansatz, bei dem aus einem Testmodell, welches auch ein Umgebungsmodell beinhaltet, Kontextverhalten extrahiert und im SUT ausgeführt wird. Das Gesamtsystemverhalten wird dem Observer aus Kontext- und SUT-Verhalten angelernt, so dass dieser eine Systemreaktion qualitativ bewerten kann.

Zusammengefasst erfolgt eine Use Case spezifische Virtualisierung des Kontexts und dessen Verhaltensinformationen bzw. des Kundenverhaltens mit anschließender Ausführung im SUT. Abbildung 1.3 zeigt auf abstrakter Ebene das prinzipielle Vorgehen, welches durch die beiden vorher genannten Bausteine, software-basierte Lösung und Interaktion auf möglichst hardware-naher Softwareebene, erst möglich wird.

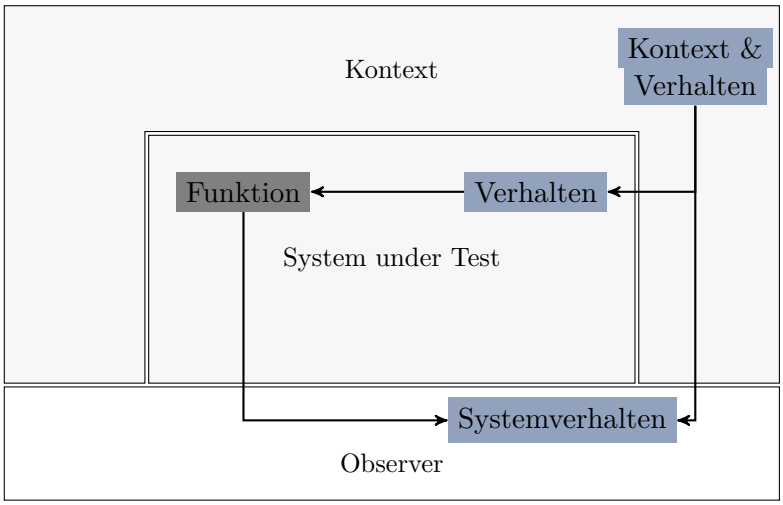


Abb. 1.3.: Kontext SUT

1.3.4. Beiträge durch den Lösungsansatz

Durch die in Kapitel 1.3.1, 1.3.2 und 1.3.3 dargestellten aufeinander aufbauenden spezifischen Problemstellungen und die damit verbundenen Lösungsansätze ergeben sich unter Verwendung eines modellbasierten Testansatzes folgende Beiträge:

- Eine einfache Automatisierung durch die software-basierte, standardisierte und hardwarenahe Schnittstelle in allen Steuergeräten des verteilten Systems.
- Die Verbesserung der Qualität des Testings durch die Schaffung von exakter Reproduzierbarkeit von Systemeingaben innerhalb einer Komponente durch die Interaktion auf Hardware-Treiber-Ebene in Verbindung mit einem Scheduling der Testausführung innerhalb der Software.
- Die Reduktion der Komplexität beim Testing durch die Verwendung eines modellbasierten Ansatzes mit der Möglichkeit der automatisierten Extraktion von relevantem Verhalten aus dem Kontext.

Der im folgenden Kapitel 1.4 formulierte Methodenvorschlag soll in genau diesen genannten Aspekten eine Verbesserung herbeiführen. In der Zusammenfassung, Kapitel 7.1 wird nochmals Bezug auf die Beiträge, durch die in Kapitel 3 entwickelte Methode genommen. Nicht betrachtet werden Aspekte der Systemverhaltensbewertung bzw. des Observers (siehe Abbildung 1.3).

1.4. Methodenvorschlag

Methodik (griech. méthodos) bezeichnet die Kunst des planmäßigen oder systematischen Vorgehens. In der Wissenschaft bezeichnet eine Methodik einfach gesagt den Weg zu einem Ziel. Der Begriff Methode beschreibt, welche Werkzeuge eingesetzt werden, um ein Ziel zu erreichen. Der Begriff Prozess, „ein sich über die Zeit erstreckender Vorgang bei dem etwas (allmählich) entsteht, sich herausbildet“ [[Dud18]] grenzt sich primär durch den Bezug auf Zeit ab. Der Begriff „Prozess“ (lat. procedere) bezeichnet die Gesamtheit aller zusammenhängenden Aktionen mit dem Ziel ein Produkt oder eine Leistung hervorzubringen. Zusammengefasst beschreibt die „Methode“ nur das „Wie“ und der „Prozess“ beschreibt das „Was“. Schlussendlich werden Prozesse benötigt, um Methoden zu erfüllen.

In der folgenden Arbeit soll eine Methode erarbeitet werden, die ein mögliches Vorgehen zeigt, wie die aus der in Kapitel 1.1 aufgeführten Problemstellung die drei Punkte verbessert werden können. Hierbei grenzt sich die Methode stark auf die Stimulation von eingebetteten Systemen bzw. verteilten Systemen ab und versucht zu den jeweiligen Aspekt der Problemstellung Potentiale zu heben, die sich auf Softwareentwicklung und das Testing projizieren lassen.

- ▷ Beherrschung der Komplexität von verteilten Funktionen - *Architektur- und Plattformunabhängigkeit erreichen.*

- ▷ Ablösen von wiederkehrenden manuellen Tätigkeiten bei der Funktionsstimulation - *Integration einer Schnittstelle für die Funktionsstimulation.*

- ▷ Verbesserung der Produktqualität durch den Einsatz von software-technischen Systemen - *Software-basierte Ausführung von Tests.*

Der nachfolgend formulierte Vorschlag soll alle aufgeführten Aspekte in einem zusammenfassen:

„Automatisierung integrierter software-basierter Simulationen von Systemeingaben in verteilten Systemen“

Der Begriff *integriert* steht hierbei für die in das verteilte System eingebettete Software-Lösung. *Software-basiert* soll hier den Aspekten Wiederverwendbarkeit und Kosten gerecht werden. *Simulationen von Systemeingaben* steht hier für die Abgrenzung der Methode. *verteilten Systemen* zeigt auf das Anwendungsszenario der Methode.

Anmerkung: Die Methode wurde im Rahmen der Arbeit von einer Grundidee ausgehend immer weiter verfeinert und tiefer in der Software-Architektur verankert. Die folgenden Kapitel zeigen nur die letztendlich entwickelte finale Methode sowie deren Potentiale. Auf die Entwicklungsschritte hin zu dieser Methode wird nur teilweise eingegangen, falls diese entscheidend für die dargestellte Methode sind.

1.5. Veröffentlichungen

Teilaspekte der Dissertation wurden in Rahmen der nachfolgend aufgelisteten Konferenzen veröffentlicht. Hierbei wurden vorerst grundlegende Aspekte des dargestellten Ansatzes in Bezug auf das modellbasierte Testen dargestellt. Folgend wurde der Ansatz im Bereich Software Trends diskutiert und letztendlich in einer Industriekonferenz eine Implementierung der Methodik dargestellt. Im Rahmen letzterer wurden auch bereits erste Ergebnisse bezüglich Qualität und Effizienz der Methodik veröffentlicht.

1 ▷ Comparing System- and Test Model with Integrated Software-Based Signal Simulation
Andreas Kurtz, Bernhard Bauer and Marcel Köberl
Modelsward 2016 - 4th International Conference on Model-Driven Engineering and Software Development, February 19 - 21, 2016 - Rome, Italy

2 ▷ Software Based Test Automation Approach Using Integrated Signal Simulation
Andreas Kurtz, Bernhard Bauer and Marcel Köberl
Softeng 2016 - The Second International Conference on Advances and Trends in Software Engineering, February 21 - 25, 2016 - Lisbon, Portugal

3 ▷ Developing Software for Mobile Devices: How to Do That Best
Hermann Kaindl, Roberto Meli, Andreas Kurtz, Bernhard Bauer and Petre Dini
Softeng 2016 - The Second International Conference on Advances and Trends in Software Engineering, February 21 - 25, 2016 - Lisbon, Portugal

4 ▷ A Method for an integrated software-based Virtualisation of System Inputs
Andreas Kurtz, Bernhard Bauer and Marcel Köberl
at 7th Conference Simulation and Testing for Vehicle Technology, May 12 - 13, 2016 - Berlin, Germany

Die Publikationen entstanden aus Forschungsarbeiten, die vom Autor dieser Arbeit durchgeführt wurden. Die Publikationen 1,2 und 4 enthalten Beiträge, die auf den Analysen und Ergebnissen des Co-Autors beruhen.

1.6. Überblick

Die allgemeine Struktur dieser Arbeit ist auf Abbildung 1.4 dargestellt. Für das Verständnis der im Folgenden beschriebenen Methodik wurde die Dissertation in vier große Abschnitte unterteilt.

Der erste Teil motiviert die erarbeitete Methode, präsentiert theoretische Grundlagen und gibt einen Überblick über die Beiträge, die im Rahmen der Arbeit entstanden sind. Leserinnen und Leser, die mit den Themenbereichen MBT, Systemarchitekturen im Automotive Bereich, AUTOSAR oder verteilten Systemen bereits vertraut sind, können den Abschnitt Grundlagen überspringen.

In Teil II wird der neue methodische Ansatz ausführlich beschrieben und Bezug auf verwandte Arbeiten und Methodiken, wie z.B. Can Calibration Protocol (CCP), Universal Measurement and Calibration Protocol (XCP) oder Diagnose genommen. Teil III beschreibt die Integration der Methodik in die gegebene Systemarchitektur und spezifische Umsetzung auf der ausgewählten Hardwarekomponente und ist sehr technisch ausgelegt. Abschließend werden in Teil IV die Anwendung, anhand von mehreren Fallstudien gezeigte Validierung der Methode sowie die Beiträge bzw. die Ergebnisse diskutiert.

Der Teil II und Teil III sind auf Abbildung 1.4 parallel Teil IV dargestellt, da oft Bezug zwischen dem methodischen Ansatz bzw. der Integration und der Anwendung und Validierung hergestellt wird.

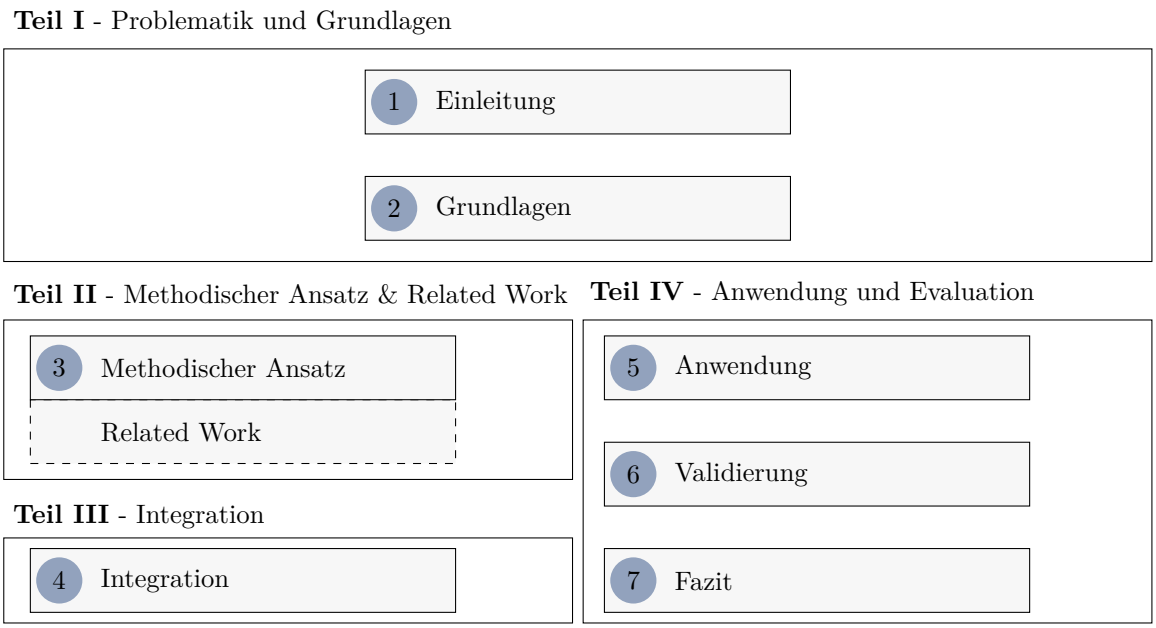


Abb. 1.4.: Struktur der Arbeit

Die folgende Zusammenfassung gibt einen kurzen Überblick über den Inhalt der jeweiligen Teile und der einzelnen Kapitel:

Teil I Problematik und Grundlagen

Kapitel 1 - Einleitung

In diesem Kapitel wird der kontextuelle Rahmen für die Thematik dieses Themas vorgestellt. Die Arbeit wird durch die Angabe der Problembeschreibung bzw. den Herausforderungen eingeleitet. Die Motivation, die ausgewählt wurde, führt zu drei allgemeinen Herausforderungen: Komplexität, Automatisierung und Qualität im Bereich Testing von Software. Aus diesen Herausforderungen ergeben sich spezifische Handlungsfelder, die an drei Ziele sowie einen Methodenvorschlag adressiert werden. Abschließend werden der Methodenvorschlag eingeführt und die wissenschaftlichen Beiträge und Publikationen des Autors aufgelistet.

Kapitel 2 - Grundlagen

Die Arbeit baut stark auf Techniken des modellbasierten Testens auf. Dies umfasst verschiedene Modelle bzw. Varianten sowie die Ebenen des Testens. Dabei spielen auch die in der Automotive Branche etablierten Systemarchitekturen eine wichtige Rolle und werden daher in diesem Kapitel ausführlich erläutert. Abschließend werden verteilte Systeme detailliert beschrieben, da sie im Automotive Bereich unumgänglich sind.

Teil II Methodischer Ansatz und Related Work

Kapitel 3 - Methodischer Ansatz

Zu Beginn wird der Betrachtungsraum des ganzheitlichen methodischen Ansatzes auf das Systemmodell des modellbasierten Testens eingegrenzt. Es folgt die schrittweise Erläuterung des methodischen Ansatzes. Es wird von der Systemarchitektur über die Architektur der Simulationskomponente bis zum Zustandsmanagement und dem Kommunikationsprotokoll alles aufeinanderfolgend erläutert. Anschließend werden die bestehenden Methoden diskutiert und die Gemeinsamkeiten und Unterschiede herausgestellt.

Teil III - Integration

Kapitel 4 - Integration

Basierend auf den Ansatz aus Kapitel 3 wird in diesem Abschnitt beschrieben, wie verschiedene Aspekte des Ansatzes in der Zielarchitektur umgesetzt werden. Dies umfasst die Auswahl der Entwicklungsplattform, sowie das Deployment, damit die Methode auch in der Anwendung richtig arbeitet. Entscheidend sind das im letzten Abschnitt des Kapitels eingeführte Scheduling der Tasks zum simAgent und die Einbindung der Simulationsmethode in genau dieses.

Teil VI - Anwendung und Evaluation

Kapitel 5 - Anwendung

Dieses Kapitel beschreibt in erster Linie das Vorgehen anhand eines eingeführten Prozesses. Dieser deckt der Vollständigkeit halber das Gesamtmodell des modellbasierten Testens (MBT) ab, um die Anwendung anschaulich darzustellen. Dabei werden die einzelnen Stationen des Prozesses sequenziell erläutert und anhand von Beispielen in der Anwendung demonstriert.

Kapitel 6 - Validierung der Methode

Die Anwendbarkeit der vorgeschlagenen Methode wird in Bezug auf die drei Herausforderungen mit Hilfe von Fallstudien, die verschiedene Anwendungsfälle darstellen, evaluiert. Einleitend werden die verwendeten Validierungsmethoden aufgeführt. Jede Fallstudie beschreibt unterschiedliche Use Cases bzw. Nutzungszenarien mit dem Ziel im verteilten System verschiedene Verteilungen der Funktionen zu erreichen. Dabei wird die Anwendung der Methode Bottom-up bis zu Auswirkungen im verteilten System analysiert. Nachfolgend werden die Ergebnisse in Bezug auf den gewählten Ansatz diskutiert. Der letzte Abschnitt des Kapitels fasst die Ergebnisse zusammen und diskutiert die Aussagefähigkeit der Evaluation.

Kapitel 7 - Fazit

Das letzte Kapitel fasst die Beiträge dieser Arbeit zusammen und diskutiert die Implikationen für den Bereich des modellbasierten Testens. Das Kapitel schließt mit einem Ausblick auf die künftige Arbeit, die durch die Bereitstellung der Beiträge aus dieser Arbeit entstanden sind, ab: Erweiterung, Verbesserung und der Etablierung der Simulationsmethodik.

2. Grundlagen

Dieses Kapitel erläutert Grundlagen, auf denen die in Kapitel 3 dargestellte Methode basiert, sowie Themen, die im Rahmen der Integration in Kapitel 4, der Anwendung aus Kapitel 5 oder der Validierung in Kapitel 6 erforderlich sind. Hierbei wird nur für das Verständnis des in dieser Arbeit gezeigten Ansatzes, relevante Begriffe und Methoden eingegangen.

Die in der vorliegenden Arbeit gezeigte Methode basiert auf grundlegenden Methoden des modellbasierten Testens. Hierfür werden in Abschnitt 2.2 Definitionen, Ansätze und Begriffe des modellbasierten Testens erläutert. Aus diesen etablierten Definitionen wird eine Definition für die vorliegende Arbeit erarbeitet.

Automatisierung von Prozessabläufen in der Softwareentwicklung wird erst durch modellbasierte Ansätze effektiv möglich und bedarf aber auch im Zusammenhang mit der hier vorgestellten Methode einer Einführung (Kapitel 2.3) und Klarstellung, welche Ziele verfolgt werden. Zudem wird kurz auf Automatisierung beim Testen (Kapitel 2.3.2) eingegangen, da im Verlauf der Arbeit ein starker Fokus bezüglich dem Ablösen manueller Tätigkeiten liegt.

Systemarchitekturen (Kapitel 2.4) sind das Grundgerüst der Systemgestaltung und stellen Richtlinien und Grenzen der Systementwicklung dar. Aufgeteilt in eine Hard- und Softwarearchitektur ist im speziellen die Softwarearchitektur AUTOSAR die dominierende Architektur für die Softwareentwicklung in der Automotive Industrie. Dieser umfangreiche Standard bietet entscheidende Vorteile, hat aber aufgrund seines Ziels, eine einheitliche Architektur für alle Systeme zu sein, einen enormen Verwaltungsaufwand.

Verteilte Systeme sind in der Produktentwicklung aktuell quasi Standard. Kostenreduktion durch Wiederverwendung von Funktionsbestandteilen macht selbst scheinbar kleinste Systeme zu verteilten Systemen und Teil einer Funktionswirkkette. Die Definition eines verteilten Systems und dessen Bestandteile wird in Kapitel 2.5 detailliert erläutert und definiert.

Neben konstruktiven Maßnahmen, technischen und organisatorischen Maßnahmen, werden grundlegend analytische Maßnahmen eingesetzt, um die Softwarequalität zu sichern bzw. Softwarefehler zu finden und das Systemverhalten bewerten zu können. Unter technische Maßnahmen fallen unter anderem Methoden und Werkzeuge. Organisatorische Maßnahmen sind z.B. Richtlinien, Standards und Checklisten. Analytische Verfahren sind unter anderem Reviews oder Testing. [Bei90]

2.1. Testen

In der heutigen automobilen Produktentwicklung findet ein kontinuierliches Testen statt, bei dem wiederkehrend die Softwarequalität gemessen wird. Der Testbegriff wurde 1979 von Myers als

„Testing is the process of executing a program with intenion of finding errors.“

definiert. Bei dieser Definition liegt der Fokus auf dem Finden von Fehlern in der Software. Es fehlt hierbei der Aspekt der Bestätigung des spezifizierten Systemverhaltens. Aus diesem Grund haben sich später andere Definitionen entwickelt. Aus dem Institute of Electrical and Electronics Engineers (IEEE) Standard [IEE90] lassen sich folgende Definitionen entnehmen.

„The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.“

und

„The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.“

Diese Definitionen beschreiben deutlich spezifischer, dass es um das Ausführen und Beobachten des Systemverhaltens geht und dabei spezifiziertes und nicht bekanntes Systemverhalten zu dokumentieren. Ziele des Testens sind, das Finden von Fehlverhalten inkl. Lokalisation, bzw. des Fehlerauslösers und die Bestätigung des spezifizierten Systemverhaltens. Angewendete Verfahren [Bei90] sind *exploratives*, *systematisches* und *modellbasiertes Testen*.

Exploratives Testen, also das manuelle Ausprobieren der Funktionen. Dies ist ein sehr ungenaues Verfahren, spiegelt aber die Nutzer- bzw. Kundenperspektive wieder. Dieses Verfahren wird in frühen Entwicklungsphasen eingesetzt und bedarf viel Wissen über die Funktionen und deren Verhalten.

Systematisches Testen, hierunter fällt auch das Zufallstesten. Hierbei werden systematisch Eingaben auf das System eingesteuert und das Verhalten bewertet.

Funktionales Testen ist das Ableiten von Testfällen aus der funktionalen Spezifikation. Die funktionale Spezifikation ist eine formale oder informelle Beschreibung des intendierten Verhaltens.

Modellbasiertes Testen, bei dem die Spezifikation in Form von formalen Modellen vorliegt.

Mit steigender Komplexität und Vernetzung von Funktionen ist das präferierte Testverfahren das *modellbasiertes Testen*, siehe Kapitel 2.2.

2.2. Modellbasiertes Testen

MBT zielt in erster Linie darauf ab, Komplexität aufzulösen und die Automatisierung von Abläufen auszudehnen. Durch die Benutzung methodischer Verfahren sollen manuelle Tätigkeiten reduziert werden können. MBT ist somit ein Enabler für die Automatisierung durch ein besseres Verständnis von Systemen aufgrund von Modellierung.

2.2.1. Definition MBT

Für den Begriff des MBT existieren viele Definitionen unter anderem gab Boris Beizer folgende Definition bereits 1990:

„Testing is a process in which we create mental models of the environment, the program, human nature, and the tests themselves. [...] The art of testing consists, selecting, exploring, and revising models.“ [Bei90]

Abgeleitet aus dieser Definition wäre jegliches Testen modellbasiert. Ein Tester nutzt immer vereinfachte Modelle, Abbilder oder vereinfachte Beschreibungen, um Tests für gewisse Szenarien zu beschreiben. Daher stößt diese Definition auch in der Anwendung modellbasierter Methoden auf hohe Akzeptanz.

Eine Definition von Dias definiert MBT sehr allgemein und fokussiert bei der Definition auf den Automatisierungsfaktor:

Model based testing approaches help automatically generate test cases using models extracted from software artifacts. [DNT09]

Gemeinsame Merkmale aus den Definitionen (hier nicht alle aufgeführt) sind zu testende Software, ihre Umgebung, Modelle und automatisierte Erstellung von Tests. Hieraus folgernd ergeben sich zwei einfache Aussagen, welche das modellbasierte Testen beschreiben, Modelle erstellen und Tests aus den Modellen generieren.

Abgeleitet aus den Gemeinsamkeiten aller Definitionen können grundlegend zwei Modelle definiert werden, das *System-* und das *Umgebungsmodell*. Die zwei Modelle sind entscheidend um denen funktional klar Grenzen zu ziehen und die Komplexität zu begrenzen.

Das *Systemmodell* beschreibt das System selbst, was es umfasst und welche Funktionen es erfüllt. Es stellt die Komponenten der statischen Struktur des Systems dar und wie sie miteinander interagieren.

Das *Umgebungsmodell* beschreibt den Ausschnitt der Welt bzw. Umgebung, in der das System agiert und mit dieser es an den Systemschnittstellen interagiert.

Die Zusammenführung dieser beiden Modelle, das s.g. Testmodell, beinhaltet genau die Informationen, um die Funktionen in ihrer Umgebung zu beschreiben. Dieses ist somit auch die Basis für die automatisierte Erstellung von Tests. Einerseits erweitert das Umgebungsmodell die Komplexität des Testmodells, andererseits schränkt es den Funktionsraum, z.B. durch die Limitierung von Temperaturbereichen, in denen das System agiert, ein. Die Abbildung 2.1 zeigt sehr anschaulich, dass das Systemmodell eine Verbindung der beiden Modelle darstellt, sowie den Raum begrenzt.

Das *Testmodell* beschreibt die Funktionen in genau dem Ausschnitt der Welt bzw. Umgebung, in der sie agieren und mit der sie interagieren.

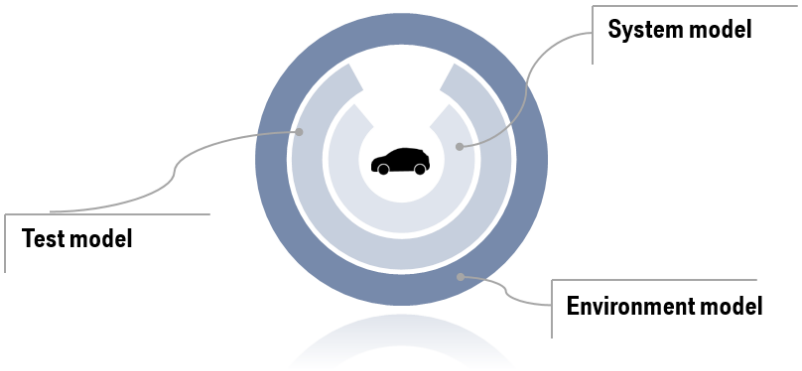


Abb. 2.1.: System-, Umgebungs- und Testmodell [RBGW10]

Dieses formale Testmodell ist dann, wie bereits angesprochen, die Basis für die automatisierte Testfallgenerierung. Ein modellbasierter Testansatz schafft es von schwer verständlichen Testfällen in tabellarischer Form oder Dokumentenform hin zu leicht verständlichen Modellen. Diese Modelle ermöglichen mittels verschiedener Methodiken, Testfälle zu generieren oder auch die Wechselwirkungen zwischen den Funktionen besser erkennen bzw. verstehen zu können.

Die Abbildung 2.2 zeigt das bereits in Kapitel 1.1 angesprochene Problem der mentalen Modelle (*implizites Modell*), aus denen Testfälle erstellt werden. Werden hingegen gleich Testfallmodelle erzeugt, können diese einerseits automatisiert überprüft werden und andererseits gleich als Datenbasis für den Tester dienen. Dieses Vorgehen bringt zwar Verbesserungen in der Qualität des Testens, durch die Möglichkeit der Modellprüfung, erzeugt aber einen nicht unerheblichen Aufwand in der Modellierung. Erst durch den Schritt System-, Umgebungs- und Testmodell zu modellieren, entsteht mit dem Einsatz eines Testfallgenerator ein Einsparpotential durch die Möglichkei der Wiederverwendung der Modelle und eine Verbesserung der Testabdeckung. [RBGW10]

Einordnung des modellbasierten Testens in den Softwaretest – Modellbasiertes Testen finden sowohl als statisches als auch als dynamisches Testverfahren statt. [RBGW10] Mehr Anwendungsfelder ergeben sich jedoch auf der dynamischen Seite im Bereich der systematischen

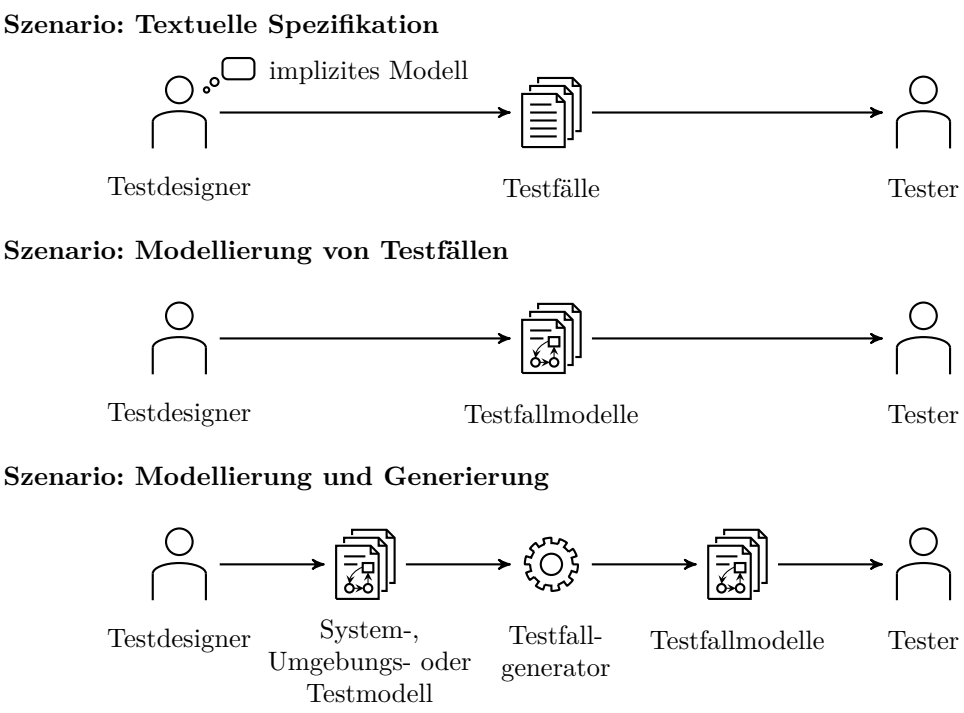


Abb. 2.2.: manuell vs. modellbasiert [RBGW10]

Tests. Diese systematischen Tests lassen sich in aktive und passive unterteilen. Aktive Tests sind definiert durch einen Stimuli, der an das System angelegt wird, gefolgt von einer Analyse und Bewertung der Systemreaktion. Bei passiven Tests wird das Systemverhalten aufgezeichnet und mit dem geforderten Sollverhalten verifiziert. Modellbasiertes Testen ordnet sich als Methodik in die vorhandenen Testmethodiken ein.

„Ein Bild sagt mehr als tausend Worte“ – Visuelle Informationen können vom Menschen leichter verarbeitet werden als textuelle. Aus diesem Grund werden Modelle mit einer graphischen Notation versehen bzw. visualisiert. Neben der leichteren Verständlichkeit und der Visualisierung bieten Modelle eine bessere Prüfbarkeit. Formale Modelle können nicht nur auf den Inhalt, sondern auch formal auf Korrektheit geprüft werden. [Kle09]

2.2.2. Ziele des modellbasierten Testens

Ziel des MBT ist es, informelle Modelle, formal und somit maschinenlesbar zu machen. Dies soll eine höhere Transparenz und Verständlichkeit schaffen und folglich in den Aspekten Kosten und Qualität, Verbesserungen bewirken. Diese formalen Modelle können dann mit Hilfe von Algorithmen automatisiert analysiert und in die jeweils für den Anwendungsfall nützliche Form überführt werden.

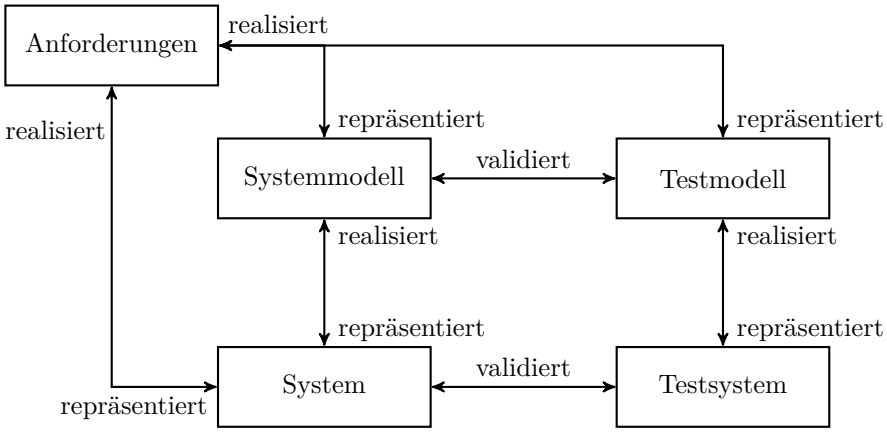


Abb. 2.3.: Ausschnitt - Relationen zwischen Test und System [Sch]

Im speziellen ergeben sich aus Unternehmenssicht folgende Vorteile für den Einsatz von MBT-Methoden.

- Kosten – Kostensenkung durch die Generierung von Testfällen und der Erhöhung der Wartbarkeit von Testfällen.

- Qualität – Verbesserung der Testabdeckung durch die automatische Generierung von Tests aus dem Modell.

Nachteil von MBT-Methoden ist die mangelnde Verbreitung in der Automobilindustrie im Gegensatz zu konventionellen systematischen oder funktionalen Testmethoden.

2.2.3. Varianten des modellbasierten Testens

In Anlehnung an die von Schieferdecker [Sch] dargestellten Varianten des modellbasierten Testens unterschieden sich diese in Systemmodell-, Testmodell- oder Systemmodell- und Testmodellgetrieben.

- Systemmodellgetrieben – ist eine sehr weit verbreitete Variante und nutzt bereits vorhandene Modelle aus der Systementwicklung. Hierbei können einige Verfahren Testfälle aus den Systemmodellen erzeugen, wie z.B. Integrationstests aus Interaktionsmodellen oder Komponenten- bzw. Systemtests aus Zustandsmodellen. Der reduzierte Modellierungsaufwand dieser Methode, da nur ein Modell verwendet wird, spricht für diese. Auch die geringe Anzahl an Inkonsistenzen zwischen System und Test machen diese Methode relativ effizient. Da aber die Unabhängigkeit zwischen System und Test fehlt, können Fehler aus dem Systemmodell in den Test übertragen werden, da System und Test aus der gleichen Quelle entstehen. [RBGW10]

- Testmodellgetrieben – setzt auf manuell oder automatisiert erzeugte Testfälle. Hierbei können auch Teile des Systemmodells aus dem Testmodell erzeugt werden. Grundlegend entsteht in dieser Variante eine erhöhte Inkonsistenz zwischen System- und Testmodell. [RBGW10] Dieser Ansatz erreicht aber durch genau seine Inkonsistenzen die höchste Fehlerentdeckungsquote. Fehler im Systemmodell oder den Anforderungen aus denen das Testmodell erstellt wurde. Spielraum in der Interpretation von Anforderungen führt meist zu Fehlern beim Testing, dabei kann jeder Fehler auf eine fehlerhafte Spezifikation oder einen wirklichen Fehler im Systemmodel zurückverfolgt werden und führt zur Anpassung des jeweiligen.

- System- und Testmodellgetrieben – setzt den ursprünglichen Ansatz des MBT am besten um. Für die Erstellung des Systems und die Testfälle werden eigenständige Modelle verwendet. Diese Variante wird auch von Pretschner [PP02, Pu] und Busch [Han06] als optimale Variante favorisiert. Vorteile dieser Varianten sind unter anderem, dass die Modelle gegenseitig auf Konsistenz oder Überdeckung geprüft werden können. Klarer Nachteil ist der vermehrte Modellierungsaufwand, jedoch kompensiert durch verbesserte Codegenerierungsmöglichkeiten. [RBGW10]

In der Praxis mit dem Fokus auf die Automotive Industrie und dieser Arbeit wird der Ansatz system- und testmodellgetrieben favorisiert. Seine Umsetzung ist aber aufgrund fehlender Systemmodelle nicht realisierbar. Für die in Kapitel 6 dargestellte Validierung der neuen Methode bestand nur die Möglichkeit der Verwendung eines testmodellgetriebenen Ansatzes.

2.2.4. Ebenen des Testens

Für die Bestätigung der Funktionsqualität wird im Testing Bottom-up vorgegangen. Hierfür wird gestuft von der Software Komponente bis zum hoch vernetzten Gesamtsystem getestet. In der Automobilindustrie wird das s.g. V-Modell herangezogen. Abbildung 2.4 zeigt hier ein V-Modell mit seinem rechten Ast, der das Testing repräsentiert. [ERZ14] Die einzelnen Stufen sind *Komponenten-*, *Integrations-*, *System-* und *Akzeptanztest*.

- *Komponententest* – testet die kleinsten teilbaren Einheiten einer Software. Das können Klassen, Methoden oder Funktionen sein. Dieser Test beschreibt den Software in the Loop (SIL)-Test. Hier wird die entwickelte Software-Komponente ohne Integration in Zielumgebung getestet.

- *Integrationstest* – beschreibt die Integration der Komponenten in Kombination. Hier werden die voneinander abhängigen Komponenten auf ihre Wechselwirkungen getestet.

- *Systemtest* – testet den Verbund aus Steuergeräten aller Domänen. Dieser kann auch vorgelagert in Form eines Teilsystemtests erfolgen. Hierbei wird das Gesamtsystem

auf Komponenten einer Domäne reduziert und mit einer statischen bzw. dynamischen Restbussimulation der anderen Domänen getestet.

- ▷ *Abnahmetest* bzw. *Akzeptanztest* – beschreibt den Test auf Fahrzeugebene ohne jegliche Simulationen aus Kundensicht.

Besonders bei System- und Abnahmetest werden Blackbox Verfahren angewendet, die eine abstrakte Sicht auf die Funktionen ermöglichen, d.h die Tests orientieren sich an dem Verhalten des Systems nicht an der Art der Umsetzung im Code. Die Grenzen der Testebenen sind nicht hart gesetzt und können je nach Projekt ineinander übergehen.

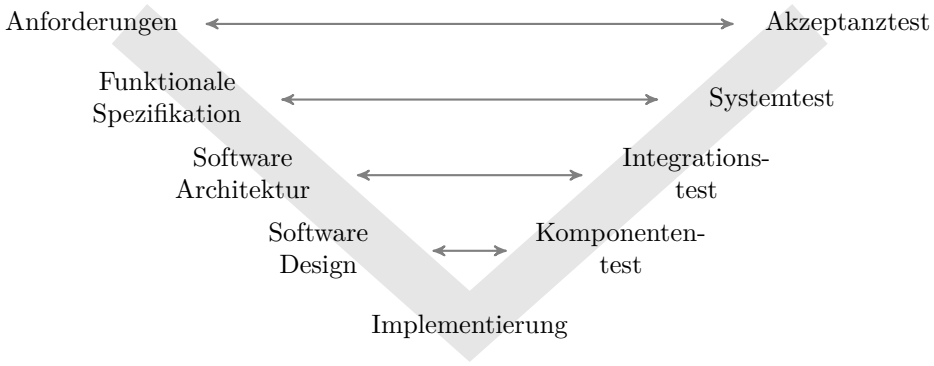


Abb. 2.4.: Ebenen des Testens - V-Modell [SBB11]

2.2.5. Methoden zum Testen eingebetteter Systeme

Methoden zum Testen eingebetteter Systeme im Bereich modellbasierten Testens sind Model in the Loop (MIL), SIL und Hardware in the Loop (HIL), die aufeinander aufbauen. Diese Methoden orientieren sich an den auf Abbildung 2.4 dargestellten Testebenen. Dabei findet MIL auf Implementierungs-, SIL auf Komponententest- und HIL auf Integrationstest-Ebene seinen Einsatz. [SVEH12, Bel07]

Beginnend wird mit MIL das entwickelte Modell in eine Simulation eingebunden. Im Rahmen der Simulation wird das Modell mit einem Umgebungsmodell verbunden und kann hierdurch auf Funktion überprüft werden.

Die nachfolgende Entwicklungsstufe ist SIL bei dem die Software auf einer generischen Hardware als ganzheitliches Tests unterzogen werden kann. Diese Methode kann bereits Anwendung finden, wenn die Zielhardware noch nicht verfügbar ist.

Die letzte Instanz vor dem Akzeptanztest, bei dem der Gesamtverbund unter realen Bedingungen getestet wird, ist der Test der Software auf der Zielhardware, genannt HIL. Hierbei wird die Hardware an Systeme gekoppelt, die die Umgebung simulieren. Beim HIL Testing kann auch eine Bewertung der Hardwarefunktionalität erfolgen. In der Automobilbranche

wird diese Art der Tests in verschiedenen Ausprägungen durchgeführt. Einerseits werden an HIL Systemen virtuelle Umgebungen geschaffen, andererseits werden reale Situationen auf die Systeme in HIL Systemen eingespielt. Diese Art der Verlagerung der Tests ins Labor schafft enorme Potentiale der Vergleichbarkeit und Reproduzierbarkeit von Tests. Limitierung hierbei ist in erster Linie, dass bisher nur effizient Kommunikationssignale abgespielt werden können. Eine weitere Einschränkung tritt dann auf, wenn Sensorwerte simuliert werden müssen. In diesem Fall steigt der technische Aufwand enorm.

2.3. Automatisierung

Der Begriff Automatisierung stammt ursprünglich aus dem Griechischen automatika - die von selbst kommend. Heutzutage steht der Begriff für das - Ablösen manueller Tätigkeiten durch technische Systeme. Dabei enthält der Begriff Automatisierung die Übernahme von Prozesssteuerungs- und Prozessregelungsaufgaben durch technische Systeme.

2.3.1. Ziele der Automatisierung

Grundlegende Ziele von Automatisierung sind die Verbesserung der Qualität und der Einsparung von Kosten und Zeit. In dem Fall des Testens steht die Verbesserung der Reproduzierbarkeit sowie die Genauigkeit von Tests im Fokus. Diese beiden Aspekte verbessern enorm die Qualität der Testaussagen. Einsparung von Kosten entsteht durch das Ablösen manueller Tätigkeiten im Testing. Die Einsparung von Zeit wird durch z.B. erhöhte Geschwindigkeit, Verkürzung von Ausfallzeiten oder durch die Erweiterung des Durchführungszeitraums von Tests erreicht. Ermüdungsresistente Testsysteme haben den Vorteil, Tag und Nacht mit gleichbleibender Qualität zu arbeiten, und ermöglichen somit das Erschließen neuer Zeiträume für das Testen. Des Weiteren kommt hinzu dass durch den nur noch geringen notwendigen Betreuungsaufwand viele Tätigkeiten parallelisiert werden können, was wiederum enorme Potentiale für eine effizientere Nutzung der Zeitrahmens schafft.

2.3.2. Testautomatisierung

Grundlegende manuelle Aufwände im Testing sind die Erstellung von Tests und die Durchführung inkl. der anschließenden Analyse und Bewertung. Diese manuellen Tätigkeiten können durch modellbasierte Testansätze teilweise, bzw. vollständig abgelöst werden. Bei der Erstellung von Tests wird es aber ohne eine manuelle Tätigkeit nicht gehen. Grundlegende Informationen müssen initial manuell bereitgestellt werden, so dass darauf aufbauend Algorithmen Modelle oder Testfälle generieren können (siehe auch Abbildung 2.2). Sinnvoll ist die Aufteilung in das Modellieren von Test- und Umweltmodellen und der anschließenden

Generierung von Tests. D.h. die Tätigkeit des mentalen Modellierens bleibt bestehen, wird nur auf die Modellierung eines realen Modells bzw. mehreren realen Modellen übertragen.

Anmerkung: Denkbar wäre das Überspringen der manuellen bzw. händischen Auflistung von Anforderungen vor der Modellierung und diese gleich zu modellieren. Nachfolgend könnten die textuellen Anforderungen z.B. aus den Modellen abgeleitet und dokumentiert werden.

Die Durchführung kann durch ein entsprechendes Framework und die erforderlichen Schnittstellen vollkommen automatisiert werden. Hierfür gibt es bereits viele Tools und Möglichkeiten, die nicht weiter aufgeführt werden.

2.4. Systemarchitekturen

Der Begriff der Systemarchitektur bezeichnet die Struktur von Systemen und ihren Komponenten und umfasst die Dekomposition des Systems. Architektur steht für *Baukunst*, ist in Verbindung mit dem Systembegriff beschränkt auf die Systemgestaltung und wird auch bei dieser vorwiegend genutzt. Unterschieden wird der Systemarchitekturbegriff je nach Branche und Ziel. Dabei können bzw. sollen z.B. Hardware- und Softwarearchitektur voneinander unabhängig sein, was mit dem in Kapitel 2.4.3 dargestellten AUTOSAR Standard erreicht werden soll.

2.4.1. Definitionen

Im Automotive Umfeld bezeichnet die Systemarchitektur die logische und physische Verknüpfung aller Elemente des Systems und wird auf eine Hardware- sowie eine Software-Architektur aufgeteilt. Diese beiden Architekturen sind in der Regel voneinander weitgehend unabhängig, welches dem AUTOSAR Ansatz (Kapitel 2.4.3) bzw. Gedanken nachkommt. Im Rahmen der modellbasierten Entwicklung werden Modelle für die Hard- und Softwarearchitektur erstellt.

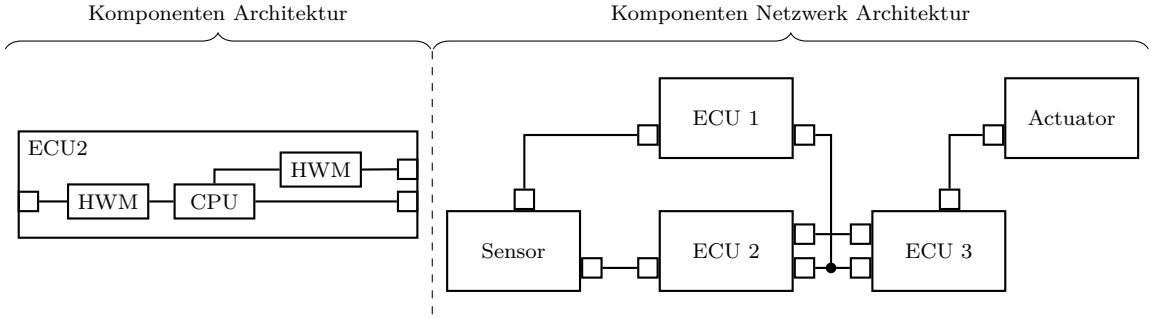


Abb. 2.5.: Beispiel - HW-Architektur

Die beispielhafte Hardware-Architektur auf Abbildung 2.5 zeigt auf der linken Seite die Komponenten Architektur, also den inneren Aufbau einer ECU, bestehend aus den einzelnen Hardware Modul (HWM)en, der Central Processing Unit (CPU) und den Ports. Auf der rechten Seite auf Abbildung 2.5 ist die Komponenten Netzwerk Architektur dargestellt, sie zeigt, in vereinfachter Darstellung, die Verknüpfung der Komponenten untereinander sowie mit den Sensoren und Aktoren.

Ein Beispiel für eine Software-Architektur auf Abbildung 2.6 zeigt, wie die Softwarekomponenten logisch verbunden sind. Die Signalflussrichtungen (dargestellt durch die Pfeile) zwischen den einzelnen Elementen werden hierbei berücksichtigt. Diese werden in der Regel von der logischen Funktionsarchitektur abgeleitet, die aus den ursprünglichen Anforderungen erstellt wurde.

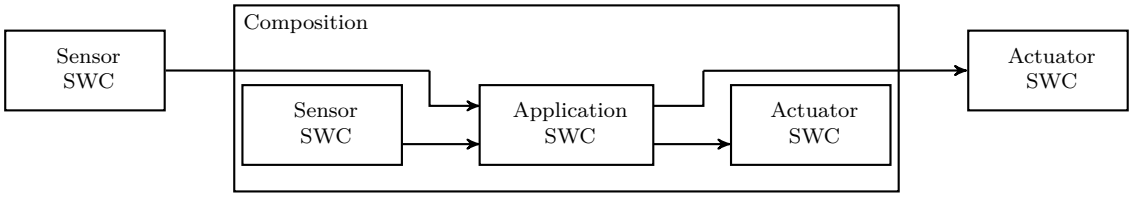


Abb. 2.6.: Beispiel - SW-Architektur

Die Zusammenführung beider Architekturen (Hard- und Software) wird entsprechend dem Anspruch von AUTOSAR bei dem Deployment durchgeführt, siehe Kapitel 4.4.1.

2.4.2. Architekturbeschreibungssprachen

Architekturbeschreibungssprachen (Architecture Description Language (ADL)) haben das Ziel einen Ausschnitt oder auch die Gesamtheit eines Softwareprojekts in einer einheitlichen Sprache zu beschreiben bzw. zu dokumentieren. ADL schaffen eine Basis, die ein einheitliches Verständnis des beschriebenen Softwareprojekts ermöglicht. ADL sind in erster Linie in zwei Gruppen unterteilbar, grafisch orientierte und formale Sprachen. Grafisch orientierte ADL werden z.B. durch Unified Modeling Language (UML)-Diagramme realisiert, sie sind weit verbreitet, da sie auch für Laien leicht verständlich sind, lassen aber oft Fragen bezüglich der Semantik offen. [ZG10]

Eine Modellierung der Abstraktionsebenen ermöglicht die Electronics Architecture and Software Technology (EAST)-ADL, welches sich in die Gruppe der grafisch orientierten ADL einreiht. EAST-ADL ist eine Architekturbeschreibungssprache, die durch das ITEA-EAST-EEA-Projekt definiert wurde und in weiteren Forschungsprojekten weiterentwickelt wurde. EAST-ADL ist ein Ansatz zur Beschreibung von Automotive Elektrik Elektronik (E/E) Systemen durch ein Informationsmodell, welches Entwicklungsinformationen in standardisierter Form beinhaltet.

Das EAST-ADL-Modell ist in folgende Abstraktionsschichten strukturiert: [EAS]

VehicleLevel – diese Schicht beschreibt die technischen Features des Fahrzeugs auf Gesamtsystem-Ebene durch das Technical Feature Model (TFM) im Systemmodell.

AnalysisLevel – beschreibt die abstrakten funktionalen Features in der Functional Analysis Architecture (FAA) des Systemmodells

DesignLevel – beschreibt die Hardware Topologie im Hardware Design Architecture (HDA) und konkrete funktionale Designs im Functional Design Architecture (FDA) des Systemmodells.

ImplementationLevel – beschreibt die Implementierung der übergeordneten Levels und wird repräsentiert durch AUTOSAR (Kapitel 2.4.3).

EAST-ADL berücksichtigt auch das Umweltmodell, welches als orthogonale Schicht auf alle Ebenen des Schichtenmodells einwirkt. Funktionale Wechselwirkungen sind hierbei in erster Linie mit den Schichten FAA, HDA und FDA spezifiziert. Auf der anderen Seite des Schichtenmodells wirken sich Schichten wie Variabilität, Zuverlässigkeit und Anforderungen auf die Gestaltung der einzelnen Abstraktionsebenen aus. Hier definiert EAST-ADL die Spezifikation und Auswirkungen Bottom-up bis zum VehicleLevel. [Schichtenmodellgrafik in [EAS] Seite 7]

Eine einheitliche Architekturbeschreibungssprache hat sich bislang nicht durchgesetzt, da die Unterstützung der Toolumgebungen bisher nicht ausreichend ist und folglich diese Sprache in der Anwendung nicht akzeptiert wird. Aufgrund dessen werden im Rahmen der Arbeit keine weiteren Sprachen genannt, da sie auch nicht verwendet werden. EAST-ADL ist hier genannt und insofern von Bedeutung, da es auf dem letzten Level *Implementation Level* auf AUTOSAR referenziert.

2.4.3. AUTOSAR

AUTOSAR ist ein offener Software Architektur Standard für Automotive Steuergerät (Electronic Control Unit) (ECU)s. Dieser Standard wurde im Jahr 2003 durch die AUTOSAR Partner ins Leben gerufen und bis heute kontinuierlich weiterentwickelt. Mitglieder in der AUTOSAR Gemeinschaft sind unter anderen die Bayerische Motoren Werke (BMW) AG, Volkswagen AG oder Zulieferer wie die BOSCH GmbH und die Continental AG.

Die Ziele von AUTOSAR sind unter anderem Standardisierung von Basic Software (BSW)-Funktionen, Skalierbarkeit auf verschiedene Plattformen, Variabilität zu verschiedenen funktionalen Domänen. Als offener Standard deklariert kann jeder der Partner Einfluss auf die Gestaltung des Standards nehmen. [AUT16, KF09, AUT14]

AUTOSAR definiert ein Metamodell zur Beschreibung der Systemarchitekturen sowie ein Architektur-Schichtenmodell mit standardisierten Schnittstellen. Das Verhalten von Softwarekomponenten wird von den darunterliegenden Schichten bezüglich der Kommunikation durch den Virtual Functional BUS (VFB) abstrahiert. Der VFB ermöglicht die Integration aller Softwarekomponenten (SWC)en, inkl. dem Zugriff auf Funktionen und der BSW, in ein System. Dadurch ergibt sich die Umsetzung eines systemweiten transparenten Kommunikationsprotokolls. [MLD+09, AUTj, AUTk]

Alle Beschreibungen zur Methodik und Spezifikationen der Schichten und Schnittstellen, die nun folgen, orientieren sich am AUTOSAR Standard 4.x.x. Eine detailliertere Eingrenzung ist hier nicht erforderlich, da die Abweichungen in den 4.x.x Standards nur marginal sind und keine Auswirkungen auf die Methodik und Implementierung haben. In erster Linie wurden in den einzelnen AUTOSAR 4.x.x Standards Spezifikationen detailliert und weitere Module hinzugenommen.

Übersicht der AUTOSAR Schichtenarchitektur

AUTOSAR will eine klare Trennung von Applikations-Softwarekomponenten und der Steuergerätehardware erreichen. Hierzu eignet sich grundlegend eine Schichtenarchitektur, die auf Abbildung 2.7 dargestellt ist. AUTOSAR bietet auf dem höchsten Abstraktions-Level drei Software-Schichten. Die Applikation bzw. Anwendungsschicht, die Runtime Environment (RTE) und die Basissoftware BSW. Diese Schichten laufen auf einer Hardware-Schicht (Abbildung 2.7 unten: ECU Hardware).

Die AUTOSAR-BSW selbst wiederum ist unterteilt in die Schichten bzw. funktionalen Cluster: Services, ECU-Abstraktion bzw. Steuergeräteabstraktion, Mikrocontroller-Abstraktion sowie der orthogonal angeordneten Schicht des Complex-Drivers (Abbildung 2.7 rechts). Durch die auf Abbildung 2.7 dargestellte Schichtenarchitektur erreicht die BSW das Ziel, die Abstraktion zur Hardware (HW) herzustellen. Des Weiteren stellt die BSW grundlegende Services für den Betrieb der Steuergeräte-HW bereit.

Auf Abbildung 2.7 ist die Anordnung der einzelnen Schichten, sowie durch ihre Anordnung eine grundlegende Schnittstellenansicht dargestellt. Nachfolgend werden die einzelnen Schichten kurz bezüglich ihrer Aufgaben und Eigenschaften erläutert.

Die Mikrocontroller-Abstraktion ist die unterste Schicht der BSW und beinhaltet interne Treiber für die darunterliegende ECU-Hardwareschicht. Diese sind Softwaremodule mit direktem Zugriff auf den ECU-Hardwareschicht und interne Peripherie.

Aufgabe	Herstellen einer Hardware-Unabhängigkeit zu den höheren Schichten der AUTOSAR-BSW.
Eigenschaften	Implementierung abhängig vom Mikrocontroller.

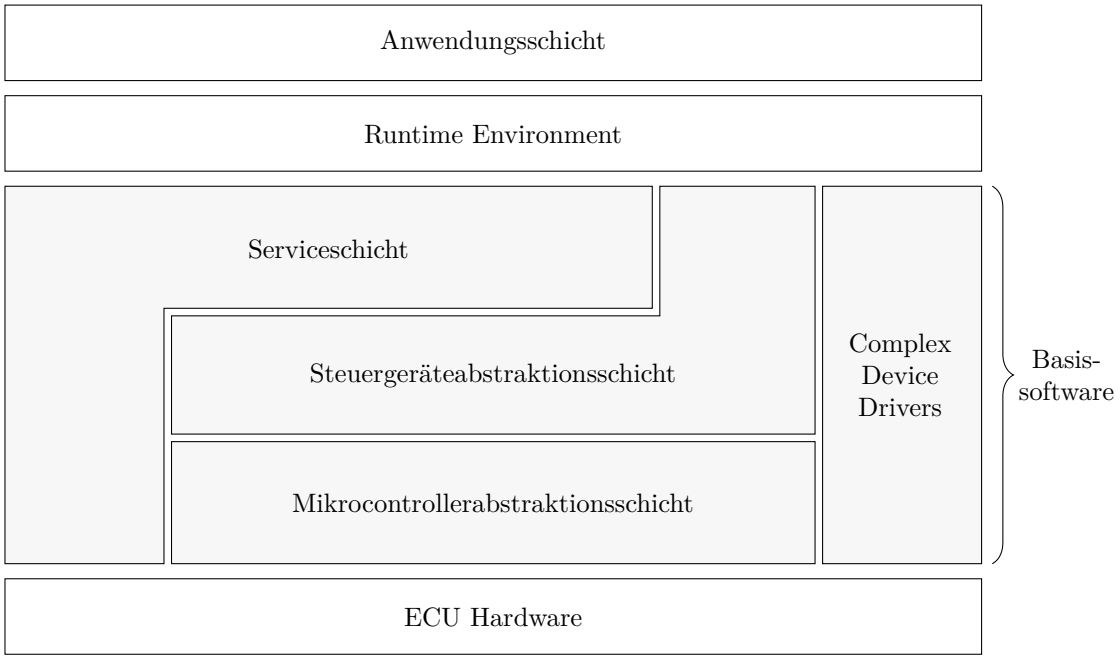


Abb. 2.7.: Übersicht AUTOSAR Layer - Funktionale Cluster

Schnittstelle zu oberer Schicht standardisiert.

Die Steuergeräteabstraktionsschicht verbindet die Treiber der Mikrocontrollerabstraktionsschicht mit der Serviceschicht und der RTE. Die Steuergeräteabstraktionsschicht bietet eine Application Programming Interface (API) für den Zugriff auf die Peripherie unabhängig ihrer Verortung.

Aufgabe — Herstellen der Unabhängigkeit höherer Softwareschichten vom ECU-Hardware-Layout

Eigenschaften — Implementierung abhängig vom Mikrocontroller und ECU-Hardware. Schnittstelle zu oberer Schicht standardisiert.

Die Complex Device Drivers bzw. Complex Drivers „Schicht“ verbindet die Mikrocontrollerschicht direkt mit der RTE und bietet die Möglichkeit nicht standardisierter Umsetzungen. Die Regeln und Zugriffsrechte hierfür sind in den AUTOSAR-Standard Dokumenten [AUTa] und [AUTb] definiert.

Aufgabe — Bietet die Möglichkeit der Integration von speziellen funktionalen Anforderungen, z.B. Treiber, die nicht im AUTOSAR spezifiziert sind oder sehr hohe Timing-Anforderungen erfordern. Des Weiteren kann z.B. direkt auf die Mikrocontroller Hardware zugegriffen werden, solange die Schnittstellen AUTOSAR-konform umgesetzt sind.

Eigenschaften Implementierung abhängig vom Mikrocontroller und ECU-Hardware. Schnittstelle zu oberer Schicht hardwareabhängig.

Die Services-Schicht ist die oberste Schicht der AUTOSAR-BSW und beinhaltet die Betriebssystemfunktionalität mit Zugriff auf I/O-Signale über die Steuergeräteabstraktionsschicht. Weiterhin beinhaltet die Services-Schicht Netzwerkmanagement-Funktionalitäten, Speichermanagement, Diagnose-Funktionen, Debugging-Funktionen, Zustandsmanagement und Watchdog-Funktionalitäten.

Aufgabe Stellt Basisfunktionalitäten für die Anwendungen (oberhalb der RTE) und Basismodule bereit.

Eigenschaften Implementierung unabhängig vom Mikrocontroller und ECU-Hardware. Schnittstelle zu oberer Schicht hardwareunabhängig.

Die RTE ist eine Schicht, die einen Kommunikations-Service bereitstellt. Die RTE repräsentiert den VFB für die jeweiligen SWCs. Alle SWCs müssen auch untereinander über die RTE kommunizieren. Aus Effizienzgründen wird die RTE für jedes Steuergerät aus den Informationen der SWC und der unter der RTE liegenden Schichten generiert.

Aufgabe Herstellen der Unabhängigkeit von Softwarekomponenten und ihrer Partitionierung auf eine ECU.

Eigenschaften Implementierung ECU-spezifisch. Schnittstelle zu oberer Schicht ECU unabhängig.

Die Abbildung 2.9 beinhaltet detailliertere Informationen über die Schnittstellen einzelner Schichten, welche in Abbildung 2.7 nur grundlegend dargestellt sind.

Übersicht der AUTOSAR Schnittstellen

Die Kommunikation zwischen den einzelnen Schichten und den funktionalen Modulen wird durch die Layer-Interaction-Matrix beschrieben. Diese ist im AUTOSAR-Standard [AUTa] definiert. Abbildung 2.8 zeigt in einer Übersicht die orthogonal zu den Schichten angeordneten Modulgruppen in der BSW. Die BSW wird in vier funktionale Cluster, den Geräte-Stack, Speicher-Stack, Kommunikations-Stack und den I/O-Stack unterteilt. Da alle Schnittstellen und Kommunikationspfade im AUTOSAR-Standard explizit spezifiziert sind, kann AUTOSAR sein Potential bezüglich Variabilität aufzeigen.

Die Layer-Interaction-Matrix auf Abbildung 2.9 zeigt die Interaktionsmöglichkeiten zwischen den Modulen, die im AUTOSAR-Standard [AUTa] definiert sind, inkl. der RTE und den Applikations-SWC der Anwendungsschicht (Abbildung 2.7). Die Layer-Interaction-Matrix

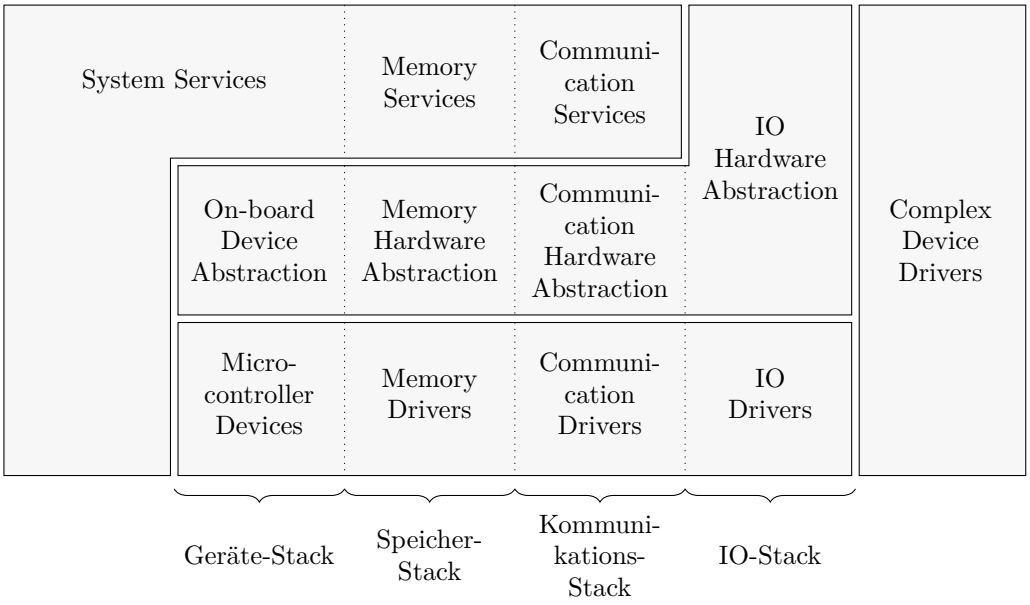


Abb. 2.8.: Übersicht AUTOSAR Schnittstellen der Schichten und Module

ist zeilenweise zu lesen, d.h. z.B. I/O-Drivers können System-Services nutzen. [AUTa] Ein Häkchen (✓) in einem Feld symbolisiert hier einen erlaubten Zugriff des Moduls auf ein anderes. Leere Felder ( ) bedeuten, dass hier kein Zugriff erlaubt ist. Ein Dreieck (△) steht für den eingeschränkten Zugriff (Callback only) auf das jeweilige Modul.

Anmerkung: Diese Verbindungen bzw. Zugriffsregeln aus der Layer-Interaction-Matrix auf Abbildung 2.9 zwischen den Modulen sind unidirektional. Ein spaltenweises Lesen der Matrix ist nicht zulässig.

Complex Drivers besitzen, wie bereits in Kapitel 2.4.3 beschrieben, eine Sonderrolle, da sie entgegen dem generellen AUTOSAR-Ansatz der Schichtenarchitektur über alle drei BSW-Schichten greifen. Der Complex Driver verbindet die Hardwareschicht direkt mit der RTE und hat unter anderem beschränkten Zugriff auf die I/O Treiber, den Kommunikationstreiber und die Speicher Treiber. [AUTb]

Die Zugriffsmöglichkeiten für die Complex Drivers sind auf der Abbildung 2.9 ausgenommen. Die detaillierten Informationen können der AUTOSAR Spezifikation AUTOSAR-Layered Software Architecture [AUTa] [S. 77-78] entnommen werden. Weitere Informationen über die Module und Zugriffe sind im aktuellen Standard AUTOSAR-Layered Software Architecture zu finden [AUTa]. Grundlegende Informationen zum AUTOSAR-Entwick- lungsprozess und der Methodologie von AUTOSAR sind sehr anschaulich in [MLD+09] dargestellt.

Weitere Informationen zum AUTOSAR-Prozess werden hier nicht näher erläutert. Für die vorliegende Arbeit wurde zwar eine AUTOSAR Architektur verwendet, ist aber für den methodischen Ansatz nicht zwingend erforderlich.

	System Services	Memory Services	Communication Services	Complex Drivers	IO HW Abstraction	On-board Device Abstraction	Memory Hw Abstraction	Comm. Hw Abstraction	Microcontroller Drivers	Memory Drivers	Communication Drivers	IO Drivers
AUTOSAR SWCs / RTE	✓	✓	✓	✓	✓							
System Services	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
Memory Services	✓	✓					✓					
Communication Services	✓	✓	✓					✓				
Complex Drivers	komplexere Regeln, hier nicht behandelt											
IO HW Abstraction	✓				✓	✓		✓	✓		✓	✓
On-board Device Abstraction	✓					✓		✓	✓		✓	✓
Memory Hw Abstraction	✓	✓				✓	✓	✓		✓	✓	
Comm. Hw Abstraction	✓		✓			✓		✓			✓	✓
Microcontroller Drivers	✓				✓	✓			△			△
Memory Drivers	✓						✓					
Communication Drivers	✓					✓		✓				✓
IO Drivers	✓				✓	✓			△			△

Abb. 2.9.: AUTOSAR-Interaktionsmatrix [AUTa]

2.5. Verteilte Systeme

„Ein verteiltes System, ist ein System in dem sich Hardware- und Softwarekomponenten auf vernetzen Computern befinden und miteinander über den Austausch von Nachrichten kommunizieren“ (George Coulouris [CDK05]). Eine ähnliche Definition gibt Henri E. Bal in [Bal92]: „Ein verteiltes Rechensystem besteht aus mehreren eigenständigen Prozessoren, die keinen Arbeitsspeicher miteinander teilen, aber über das Senden von Nachrichten in einem Kommunikationsnetz zusammenarbeiten.“ In seiner Definition fügt Tanenbaum der inneren Betrachtungsebene noch den Benutzer hinzu: „Ein verteiltes System ist eine Sammlung unabhängiger Computer, das seinen Benutzern als ein einheitliches, kohärentes System erscheint.“ [ATv07] Diese Definitionen sind durchaus treffend für verteilte Systeme im Automotive Umfeld.

Unterschiede finden sich in der Bezeichnung der Bausteine eines verteilten Systems. Im Automobil spricht man von eingebetteten Systemen (engl. embedded Systems), die über BUS-Systeme kommunizieren. Diese Einschränkung auf BUS-Systeme ist aber zur heutigen Zeit nicht mehr ganz korrekt und bedarf der Erweiterung auf Kommunikationstechnologien zu Backendsystemen. Viele Funktionen beziehen heutzutage Daten bzw. Parameter laufend aus der Infrastruktur. Ein Beispiel ist hierfür die Funktion Real Time Traffic Information (RTTI), welche ihre spezifischen ortsbezogenen Daten kontinuierlich aus dem Backend bezieht. Somit muss die Definition bezüglich ihrer Kommunikation auf nahezu alle gängigen Kommunikationstechnologien erweitert werden. Daher sind die o.g. eher allgemein formulierten Definitionen von Coulouris oder Tanenbaum doch durchaus treffend.

Der Begriff BUS-Systeme beinhaltet unter anderem Technologien von Local Interconnection Network (LIN) über Controller Area Network (CAN) bis hin zu Ethernet (Eth). Die Bezeichnung BUS-System steht für alle Systeme zur Datenübertragung zwischen den Teilnehmern.

Eingebettete Systeme sind nach der Definition von Peter Scholz [Sch05]: „Eingebettete Systeme [...] sind Computersysteme, die aus Hardware und Software bestehen, und die in komplexe technische Umgebungen eingebettet sind. Solche Umgebungen können maschinelle Systeme, wie etwa Kraftfahrzeuge [...] sein, die der Interaktion eines menschlichen Benutzers bedürfen oder vollautomatisch agieren."

Eingebettete Systeme sind somit einzelne Einheiten bestehend aus Hardware und Software die folgenden Aufgaben erfüllen:

- Sensieren (engl. Sense)
- Analysieren (engl. Analyse)
- Entscheiden (engl. Decide)
- Reagieren (engl. Act)

In der Automotive Branche spricht man bei eingebetteten Systemen auch von elektronischen Steuergeräten (engl. Electronic control unit ECU). Der Begriff „Elektronik" setzt sich zusammen aus der Überkreuzung von Elektron und Technik. Nach Definition steht Elektronik für die *Lehre der Steuerung von Elektronen*. Im Folgenden ist bei dem Begriff „Elektronik" inkludiert, dass das Steuergerät aus Hard- und Software besteht. [BBT10]

2.5.1. Definition

In der vorliegenden Arbeit wird eine leicht abgewandelte Definition von George Coulouris, die bereits angesprochen wurde, verwendet: „Ein verteiltes System ist ein System, in dem sich Hardware- und Software-Komponenten auf vernetzten eingebetteten Systemen befinden miteinander über den Austausch von Nachrichten kommunizieren."

Explizit wurde der Begriff *Computer* durch *eingebettete Systeme* substituiert. Hintergrund für die Spezifizierung ist der bereits angesprochene gängige Sprachgebrauch im Automotive Umfeld. Die im Automotive Bereich verwendeten Kleinstcomputer zählen zu den eingebetteten Systemen [Bru15] und werden als ECU bezeichnet. Berücksichtigt wird aber auch eine Vernetzung mit Backendsystemen, doch diese werden im Rahmen der Arbeit nicht explizit untersucht. Grund ist hierfür, dass zum aktuellen Zeitpunkt nur Funktionen wie z.B. RTTI zur Verfügung standen und keine Einflussnahme auf die vom Backend bereitgestellten Daten möglich war.

2.5.2. Verteiltes System im Automotive Umfeld

Der Begriff *Verteiltes System* beschreibt den Zusammenschluss aus mehreren einzelnen eingebetteten Systemen bzw. ECUs. Im Automotive Umfeld gilt es diese Zusammenschlüsse in zwei Klassen zu unterteilen. Diese Unterteilung hat entwicklungstechnische Gründe und das Ziel, die angestrebte Systemqualität des verteilten Systems schneller zu erreichen. Der Produktentwicklungsprozess (PEP) sieht vor, die Komponenten und Systeme gestuft bis zur Serienreife zu entwickeln. Komponenten werden domänenspezifisch geclustert und bilden sogenannte Teilsysteme. Hier können die entwickelnden Fachbereiche in kleinen Stufen die Teilsystemreife auf Serienreife heben und somit Funktionsumfang und Qualität gezielt auf den Sollwert heben. Diese Teilsysteme werden dann gemäß dem PEP in größeren Stufen zu einem Gesamtsystem verbunden, um das Zusammenspiel und die Wechselwirkungen der Teilsysteme untereinander zu untersuchen. Somit verringert eine Aufteilung des Gesamtverbunds in Domänen-/ Funktionscluster die Wechselwirkungen der in der Entwicklung der noch sehr unreifen Funktionen und erlaubt den entwickelnden Fachbereichen eine effizientere Entwicklung.

Teilsysteme sind meist nach Domäne freigeschnittene Systemverbunde, die sich in den Grenzen klar von den anderen Domänen trennen lassen. Domänen sind in der Automobilindustrie Cluster wie z.B. Antrieb, Fahrwerk, Karosserie und Infotainment. Eine andere Art des Freischneidens eines Teilsystems ist die Auftrennung nach Funktionsclustern. Dies bedeutet eine Reduktion des Systemverbunds auf die nur zwingend notwendigen Bestandteile. Ein Beispiel für ein domänenspezifisches Cluster ist das Teilsystem der Domäne *Infotainment und Karosserie*. Dieses beinhaltet z.B. Komponenten wie Fensterheberschalter, Scheibenwischer, Navigationssystem. In diesem Teilsystem sind aber keinerlei elektronische Antriebs- oder Fahrwerkskomponenten zu finden.

Gesamtsysteme, auch Fahrzeug genannt, beschreiben den Verbund aller ECUs. Ein Gesamtsystem geht heutzutage aber auch über die Systemgrenze Fahrzeug hinaus. Längst werden Funktionen in Backendserver (Cloud Web Services) ausgelagert oder das Fahrzeug ist stark mit anderen Systemen vernetzt. Somit ist der Begriff Gesamtsystem in der gegenwärtigen Fahrzeugentwicklung ein Kollektivum von System Fahrzeug, Backendsystemen und weiteren Computern bzw. IT-Systemen.

Gesamtsysteme können daher stark variieren aufgrund der unterschiedlichen Konfigurationsmöglichkeiten eines Fahrzeugs und den damit verbundenen Systemen. Aktuelle Fahrzeuge können in über 20.000 Varianten konfiguriert werden. Ein Beispiel zur Veranschaulichung hierfür sind z.B. drei Varianten eines Kombiinstruments, welches je nach Fahrzeugkonfiguration zwischen *Low* (wenig Funktionalität, analoge Anzeigen), *Mid* (analoge Anzeigen, teilweise digitale Anzeigen) und *High* (voll-digitales Anzeigeinstrument) unterschieden wird. Zu der Hardwarevarianz kommt eine hohe Software-Diversität und lässt die Fülle an Varianten exponentiell ansteigen.

3. Methodischer Ansatz

> „Probleme kann man niemals mit derselben Denkweise lösen, durch die sie entstanden sind.“
>
> — Albert Einstein

Der folgende methodische Ansatz versucht über die „Virtualisierung“ von Systemeingaben die Funktions-Wirkketten auf ein Maximum auszudehnen, um entsprechend Wechselwirkungen im Gesamtsystem aufzudecken. Entscheidend ist hierbei, dass die Lösung software-basiert ist und selbst ein verteiltes System darstellt.

Ziel des Ansatzes ist einerseits eine möglichst hardwarenahe Simulation von Systemeingaben zur Maximierung der Funktions-Wirkketten, andererseits sollen durch eine möglichst hardwarenahe Simulation die Varianten an Signaltypen reduziert werden. Der entwickelte Ansatz ist eine abgewandelte Form der in Kapitel 2.2.5 dargestellten Methode des HIL Testings mit der Veränderung, dass die Umgebung inkl. des Kunden auf der Hardware über eine Softwarelösung simuliert bzw. virtualisiert wird. Somit sollen HW-Bausteine, die der Signalsimulation dienen, von HIL Systemen substituiert werden.

Der Ansatz fokussiert auf das Testing auf System Level Ebene, d.h. auf Systemeingaben, die direkt oder indirekt durch den Kunden an der HW-Schnittstelle eingespielt werden. Grundlegend wird sich in der Produktentwicklung an dem in Kap. 2.2.4 beschriebenen V-Modell orientiert, in dem die Ebenen auch deklariert sind. Der nachfolgend gezeigte Ansatz soll, wie bereits gesagt, in erster Linie auf System Level Ebene greifen, kann jedoch aufgrund seiner Gestaltung in den untergeordneten Ebenen Vorteile bringen und Anwendung finden.

Über eine in das SUT integrierte SWC können in der AUTOSAR-BSW Signalsequenzen in das SUT eingespeist werden. Diese Signalsequenzen werden gemäß dem MBT-Ansatz aus dem Testmodell errechnet und in Signalsequenzen aufgeschlüsselt. Diese Signale werden separiert in System-Input-Signale für die Simulation und System-Reaktions-Signale für die Bewertung des Systemverhaltens.

Herausforderungen in der Methodik entstehen bei der Modellierung des Testmodells, der Testmodellberechnung und der verteilten Simulation in einem verteilten System mittels einer in das verteilte System integrierten Softwarelösung. Der Ansatz fokussiert auf die Stimulation des Gesamtsystems und nutzt bereits etablierte Methoden der Systemreaktionsanalyse, Kapitel 3.1 geht darauf nochmals genauer ein.

3.1. Eingrenzung des Methodischen Ansatzes

In Bezug auf den modellbasierten Testansatz von Schieferdecker aus Kapitel 2.2 beschränkt sich der folgende methodische Ansatz auf die Integration des Softwaremoduls in eine bestehende Architektur des Systemmodells. Hinzu kommt auf der Seite des Systemmodells die Einschränkung auf den Prozess für die Stimulation des verteilten Systems.

Seitens des Testmodells beschränkt sich der methodische Ansatz hier rein informell darauf, das methodische Vorgehen zur Ermittlung der Signalsequenzen zu zeigen. Dies wird beispielhaft, inklusive einer prototypischen Umsetzung dargestellt. (weitere Details sind hier ausgenommen)

Ganzheitlich ausgenommen ist das Thema der Systemreaktionsbewertung, da hier bereits sehr effektive Ansätze in der Praxis im Einsatz sind. Verwendet werden z.B. zustandsbasierte Automaten, die aus dem Testmodell generiert werden können. Eine auch hier verwendete Methode ist mittels Fallunterscheidung die Systemreaktionen anhand von Signalen zu bewerten.

Die vorgestellte Methode soll in erster Linie das methodische Vorgehen zeigen, wie mittels der Integration eines Softwaremoduls die Automatisierung und die Qualität des Testens und der Ergebnisse verbessert werden. Hinzu zeigt diese, wie durch das methodische Vorgehen manuelle Tätigkeiten durch eine solche Software-Lösung reduziert werden können und dadurch der entscheidende Vorteil der Reproduzierbarkeit entsteht.

Die vorgestellte Methode hat Ähnlichkeiten mit den Protokollen CCP und XCP sowie Ähnlichkeiten in der Methode mit Diagnose Funktionen. Unterschiede und Ähnlichkeiten werden in Kapitel 3.4 aufgezeigt und analysiert.

3.2. Herausforderung „SW-basierte Automatisierung“

Wie bereits einleitend in Kapitel 3 angesprochen, soll die zu entwickelnde Lösung softwarebasiert sein, ohne den Einsatz zusätzlicher Hardware. Demnach sollen Softwaremodule integriert im embedded System ein automatisiertes Testen des verteilten Systems ermöglichen. Die Herausforderung hierbei ist eine Art selbstständiges verteiltes System in eine bestehende standardisierte Softwarearchitektur (im Fall hier AUTOSAR-basiert) zu integrieren, aber die bestehenden Kommunikationspfade und Ressourcen zu nutzen.

Die Schwerpunkte Komplexität, Qualität und Automatisierung aus Kapitel 1.4 sollen im Rahmen des *modellbasierten Testens* mittels einer integrierten Softwarelösung angegangen werden.

Durch den Einsatz von Software können die Themen Komplexität und Varianten beherrscht werden, ohne den Business Case (BC) bzw. die Aufwand-Kostenrechnung aus dem Zielwert zu bringen. Software kann hier durch seine positiven Eigenschaften der Wiederverwendbarkeit, Modularität und Skalierbarkeit ein möglicher Lösungsansatz sein, der im Folgenden näher ausgeführt wird. Die Herausforderungen entstehen hier bei der nahtlosen Integration in das zu testende System (SUT).

3.3. Integrierte verteilte Simulation von Systemeingaben

Der folgende Abschnitt erklärt die ausgearbeitete Methode der „Software-basierten Simulation von Systemeingaben in verteilten Systemen“. Beginnend von der in das zu integrierende System wird Top-down die Architektur des Gesamtsystems Fahrzeug, die der verteilten Simulation und der Zusammenschluss der beiden Architekturen, bis hin zu den einzelnen Software Komponenten und deren Funktionslogik hergeleitet.

3.3.1. Die Fahrzeug Architektur als „verteiltes System“

Die Architektur beschreibt alle Aspekte des Systems der zusammenwirkenden Komponenten. Unterteilt wird die Fahrzeug-Architektur in eine HW- und Software (SW)-Architektur. Die HW-Architektur beschreibt alle Komponenten und deren physische Vernetzung, die SW-Architektur beschreibt alle SW-Komponenten und deren logische Verknüpfung.

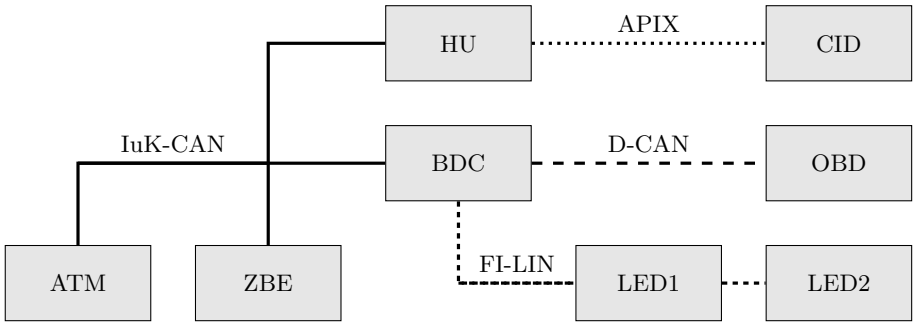


Abb. 3.1.: Hardwarearchitektur - Beispielausschnitt

- Hardwarearchitektur - Die Hardwarearchitektur, auch E/E-Architektur genannt, eines Fahrzeugs ist eine Verknüpfung verschiedener Topologien und kann daher als *vermaschte* Topologie bezeichnet werden. Grundlegende Elemente der ganzheitlichen Topologie sind die Standard-BUS-Systeme, wie z.B. CAN, die eine BUS-Topologie aufzeigen. In Kombination mit Media Oriented System Transport (MOST)-Systemen für Entertainmentfunktionen, LIN-Bussen sowie diversen sogenannten „privaten“ CAN-Bussen entsteht eine vermaschte

Topologie. „Private“ CAN-Busse unterscheiden sich dadurch, dass diese nicht über das zentrale Gateway vernetzt sind.

Die Hardwarearchitektur eines Fahrzeugs ist dargestellt durch die einzelnen ECU, deren Kommunikationsverbindungen und Sensoren bzw. Aktoren. Abbildung 3.1 zeigt einen Ausschnitt aus einer Architektur eines Fahrzeugs in vereinfachter Form. Die Darstellung zeigt Komponenten wie z.B. Headunit (HU), Zentrale Bedieneinheit (ZBE) und ihre möglichen Kommunikationspfade sowie das zentrale Gateway Body Domain Controller (BDC).

- Softwarearchitektur - Von einer generischen Perspektive aus kann die Softwarearchitektur in der VFB-Sicht (Abbildung 3.2) als eine Form von BUS-Topologie bezeichnet werden. Der grundlegende Gedanke hinter dieser Darstellung ist die Entkopplung von HW bei der Softwareentwicklung. Alle SWCs werden über ein AUTOSAR-Interface an den s.g. VFB angeknüpft. Der VFB bündelt alle Kommunikation auf einem virtuellen BUS. Erst in Kombination mit der Hardwarearchitektur werden beim Deployment (siehe Kapitel 4.4.1) die Partitionierung und die physischen Kommunikationspfade definiert.

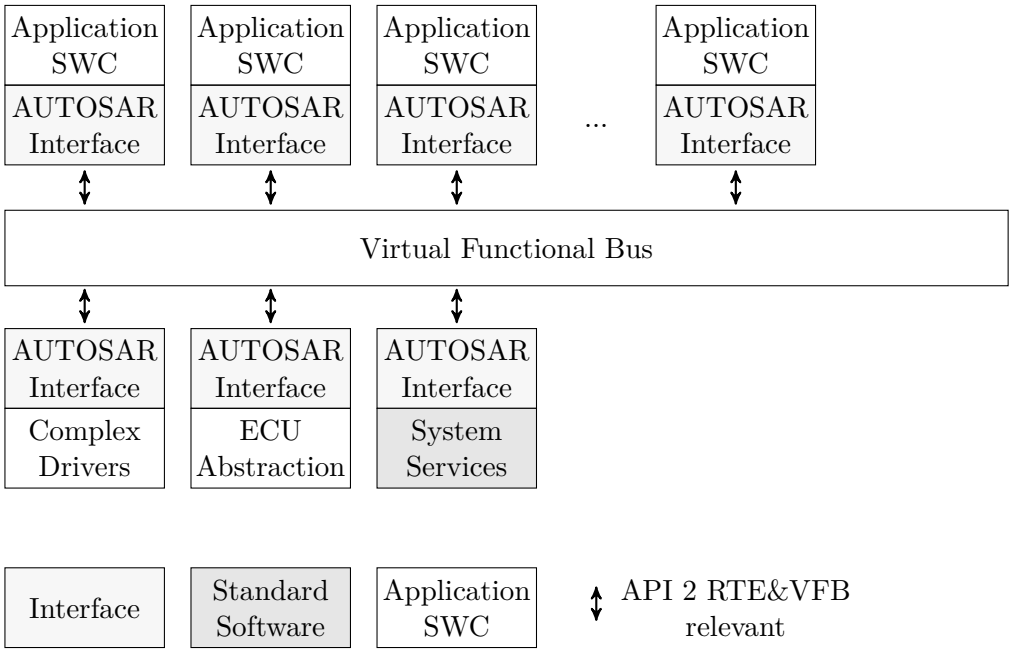


Abb. 3.2.: Softwarearchitektur - VFB

Die VFB-Darstellung zeigt alle Module, und Bausteine die für die Realisierung benötigt werden. Standard-Software Komponenten kommen in der VFB Darstellung nur einfach vor und werden beim Deployment auf jede ECU partitioniert. Dabei werden steuergerät-spezifische Konfigurationen entsprechend Beschreibungsdateien berücksichtigt. Ähnliches gilt für Treiber, diese werden entsprechend der Steuergeräte-Beschreibungsdateien in Zusammenhang mit den partitionierten SWCs konfiguriert, Details siehe Kapitel 4.4.1.

3.3.2. Architektur der verteilten Simulation

Die Architektur der verteilten Simulation ist aus einer generischen Perspektive eine BUS-Topologie. Die Darstellung der Topologie folgt der VFB Darstellung auf Abbildung 3.3. In dieser Ansicht kommt nur ein Modul (simModul) hinzu, alle anderen Anpassungen im Gesamtsystem sind innerhalb bestehender Module und sind in dieser Ansicht nicht zu erkennen. Des Weiteren ist der Off-Board Anteil simMaster dargestellt. Freigeschnitten zeigt sich die Softwarearchitektur der simModule wie auf Abbildung 3.5. Diese Architektur ist unabhängig von der Partitionierung und wird im Folgenden im Detail beschrieben.

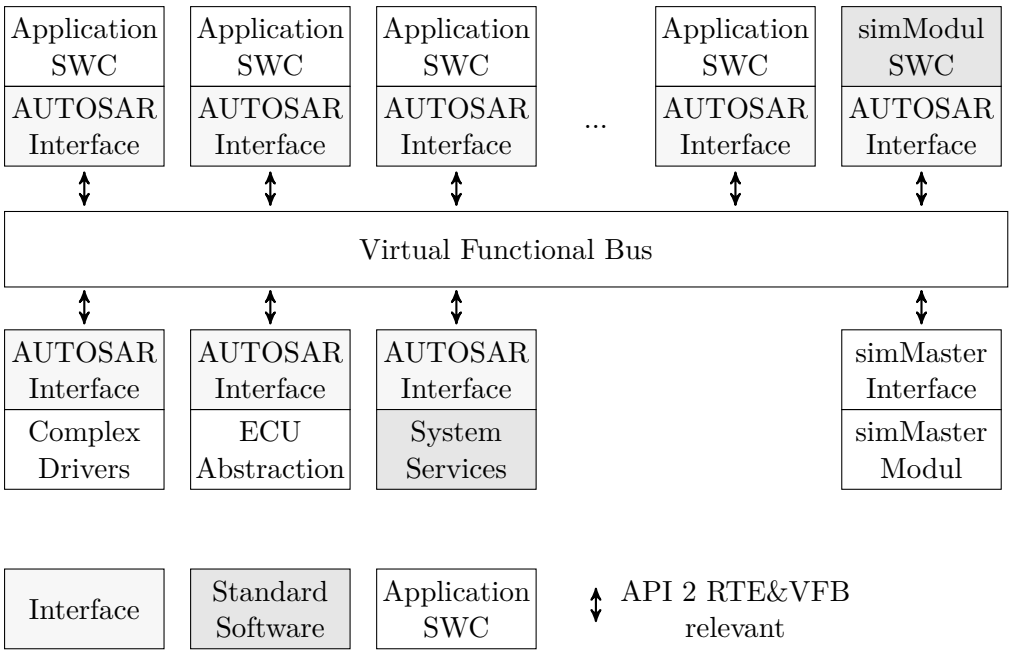


Abb. 3.3.: Softwarearchitektur mit simModul - VFB

Die einzelnen Elemente der verteilten Simulation sind:

simMaster — außerhalb der Systemgrenze des verteilten Systems, Testcase Repository, Koordination, Ausführung, Bewertung und Transaktion von Testcases.

simModule — Ausführendes Element mit Kommunikation über eine BUS- Schnittstelle, Interaktion mit dem übergeordneten verteilten System über die modifizierte Treiber-Schnittstelle.

Kommunikationspfad — simMaster nutzt die Standard Diagnose Kommunikation (im aktuellen Beispiel Abbildung 3.1 D-CAN), um die Testdaten in das Gesamtsystem über das Gateway (BDC) zu verteilen.

Durch das Deployment wird das simModul aus Abbildung 3.3 auf die einzelnen ECUs auf die auf Abbildung 3.1 dargestellte Hardware-Architektur verteilt. D.h. die simModule werden wie Standard Software Module des AUTOSAR-Standards behandelt. Hierdurch entsteht die finale Gesamtsystemarchitektur, die maßgebend für die Schnittstellengestaltung und Kommunikation ist. Die Abbildung 3.4 zeigt einen Ausschnitt der Gesamtsystemarchitektur basierend auf den bereits dargestellten Architekturen.

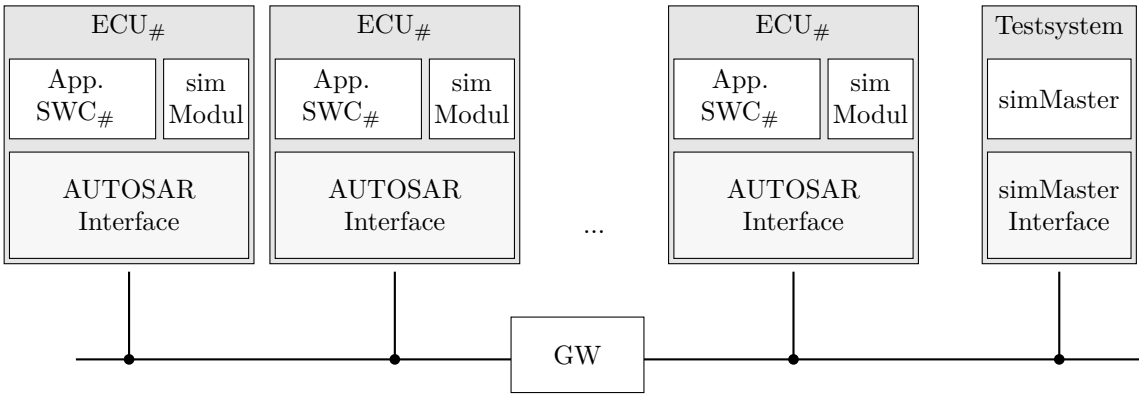


Abb. 3.4.: Ausschnitt der Gesamtsystemarchitektur mit simModul

Für alle weiteren Betrachtungen bezüglich der Funktionsweise macht die finale Topologie keinen Unterschied aus und kann vernachlässigt werden. Grund hierfür ist, dass die Kommunikation zu den einzelnen Teilnehmern (simAgents) über einen Broadcast abläuft. Die Verwendung eines Broadcasts stellt sicher, dass alle Komponenten im verteilten System erreicht werden. Auch die Antwort der simAgents zum simMaster erfolgt über einen Broadcast. D.h. theoretisch können alle im simAgents im SUT die Botschaften empfangen. Durch die Möglichkeit der verschiedenen Arten von Topologien muss eine funktionale, also direkte über die ECU-Identifikator (ID) Adressierung der simAgents erfolgen.

Anmerkung: Interessante Auswirkungen haben die Topologie und die verwendete Technologie der Kommunikation auf die Gesamtsystemreaktion sowie einen Zielparameter. Diese Auswirkungen werden im Kapitel 6 dargestellt.

3.3.3. Architektur Simulationsmodule

Die Abbildung 3.5 zeigt die Architektur der integrierten Simulationsmodule (simModul) bestehend aus den folgenden Komponenten:

simAgent ist die ausführende Softwarekomponente mit Kommunikation zum Off-Board Testsystem über den BUS durch die entsprechenden AUTOSAR-COM-Layer. Die Interaktion mit dem SUT bzw. übergeordneten verteilten System erfolgt über die AUTOSAR-Treiber Schnittstelle.

simGW das Gateway für den Zugriff auf die AUTOSAR-Treiber Ebene. Hierbei ist der Weg über den Complex Device Driver (CDD) erforderlich, da in der [AUTa]-Spezifikation direkte Zugriffsarten auf die Treiber Ebene untersagt sind.

DioSim Die Erweiterung des AUTOSAR-Treibers um eine Schnittstelle zum CDD für die Einspeisung der zu simulierenden Kundeneingaben.

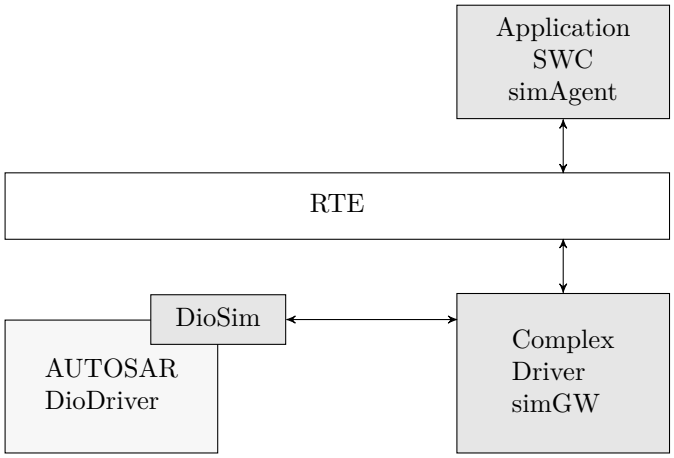


Abb. 3.5.: Architektur - simModule

3.3.4. Zustandsmanagement

Die beiden zentralen Elemente simMaster und simAgent der verteilten Simulation erfordern jeweils ein Zustandsmanagement, um die Abläufe untereinander koordinieren zu können. Des Weiteren sind Fehlerreaktionen, Fehlerzustände und Abbruchbedingungen aus Robustheits- und Sicherheitsgründen unabdingbar.

Abbildung 3.6 zeigt das Zustandsmanagement des simMaster mit folgenden Zuständen:

Init dient der Identifizierung aller benötigten Teilnehmer für die Durchführung eines Testfalls. Der Identifizierungsaufruf wird eine definierte Anzahl von Versuchen wiederholt (hier willkürlich gewählt: 10 Wiederholungen) und auf Antwort der sich im System befindenden simAgents gewartet.

Charging dient dem Upload der Testfall-Daten in das SUT. Es werden alle Testschritte einzeln per Broadcast in das SUT gesendet.

CheckUpload wird benötigt, um den erfolgreichen Upload der Testfall-Daten zu bestätigen. Hierzu wird die Anzahl der hochgeladenen und von den simAgents gespeicherten Testfallschritte abgeglichen.

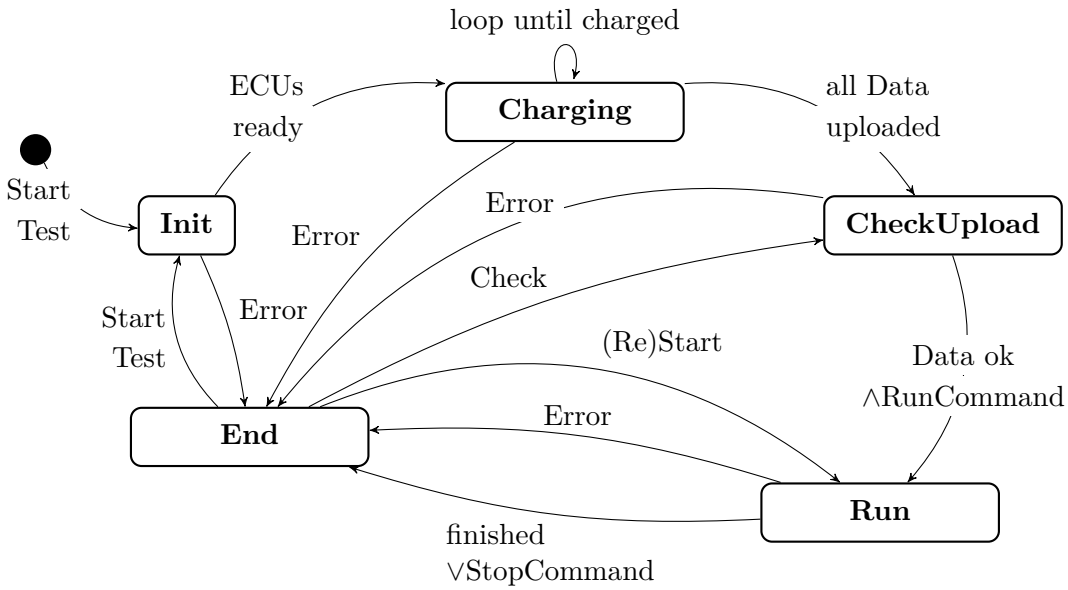


Abb. 3.6.: Zustandsmanagement - simMaster

Run — nach dem Versenden des RunCommands wartet der simMaster eine vorher berechnete Testfalldurchführungsdauer ab und beendet zum Schluss den Testablauf. Alternativ kann wie auf Abbildung 3.6 gezeigt auch gleich eine Funktionsreaktion bewertet werden.

End — ist der letzte Zustand des Ablaufs und kann für Postconditions verwendet werden, z.B. gespeicherte Daten löschen oder die simAgents zurückzusetzen. Des Weiteren entsteht hierbei der finale Report (Umsetzungsabhängig).

Aus allen Zuständen kann im Falle eines Fehlers über einen „Notstopp“ der Testablauf unterbrochen werden. Zu diesem Zweck wird eine BUS-Nachricht an alle simAgents verschickt, welche mit höchster Priorität alle Vorgänge stoppt.

Die in die embedded Systems im SUT implementierten simAgents haben ein zum simMaster komplementäres Zustandsmanagement. Auf Abbildung 3.7 sind die folgend beschriebenen Zustände und Transitionen dargestellt.

Init — dient dem Löschen aller alten Testfallschritte, welche sich noch eventuell im Arbeitsspeicher der ECU befinden können. Gefolgt von dem Löschen der Daten wird die eigene ECU-ID versendet.

Downloading — ist der Zustand des simAgents, indem er die auf dem BUS auf die für ihn gekennzeichneten Daten wartet (WaitForData) und Testfallschritte speichert, sofern diese mit der eigenen ECU-ID gelabelt sind.

CheckUpload ist eine Funktion zur Sicherstellung der Vollständigkeit der Daten im simAgent. Hierzu wird nach dem Download der Daten die Anzahl der gespeicherten Testschritte an den simMaster versendet.

Running beschreibt den Zustand in dem die gespeicherten Testsequenzen ausgeführt werden und über die Treiberschnittstelle in das embedded System zur Stimulation des Gesamtsystems eingespielt werden.

Stopped dieser Zustand wird in der Software bzw. der ECU initial eingenommen sowie nach dem Ausführen der Sequenzen oder durch Abbruchbedingungen. Stopped ist der sicherste Zustand des simAgent und wird daher auch bei Fehlern eingenommen.

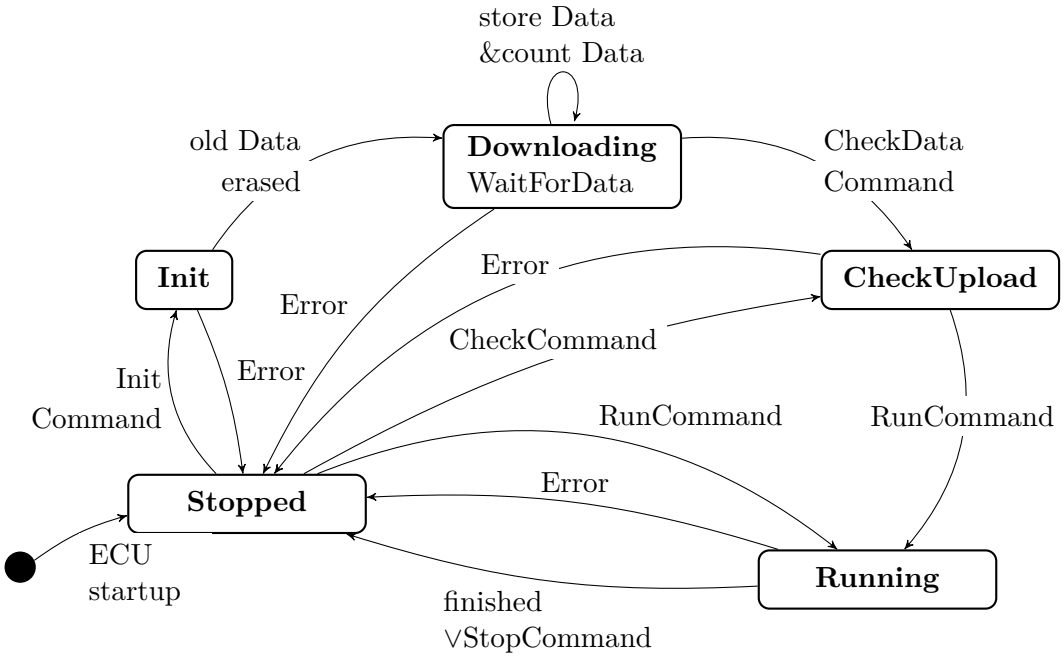


Abb. 3.7.: Zustandsmanagement - simAgent

3.3.5. Protokolle

Protokolle beschreiben das Format, den Inhalt, die Bedeutung und die Reihenfolge der gesendeten Nachrichten zwischen mehreren Teilnehmern. Die jeweiligen Umsetzungen sind letztendlich abhängig von der gewählten Kommunikationstechnologie und werden am Beispiel in Kapitel 4.4.11 erläutert.

Generell wird die Kommunikation zwischen simMaster und simAgent immer durch den simMaster initiiert. Der simAgent ist somit im Rahmen der Single-MasterMulti-Slave Ver-

bindung eine rein antwortende Komponente. Das generische Kommunikationsprotokoll in chronologischer Reihenfolge zwischen simMaster und simAgent wird im folgenden Abschnitt anhand dem abstrakten Sequenzdiagramm 3.8 verdeutlicht und der Inhalt der Kommunikation präzisiert.

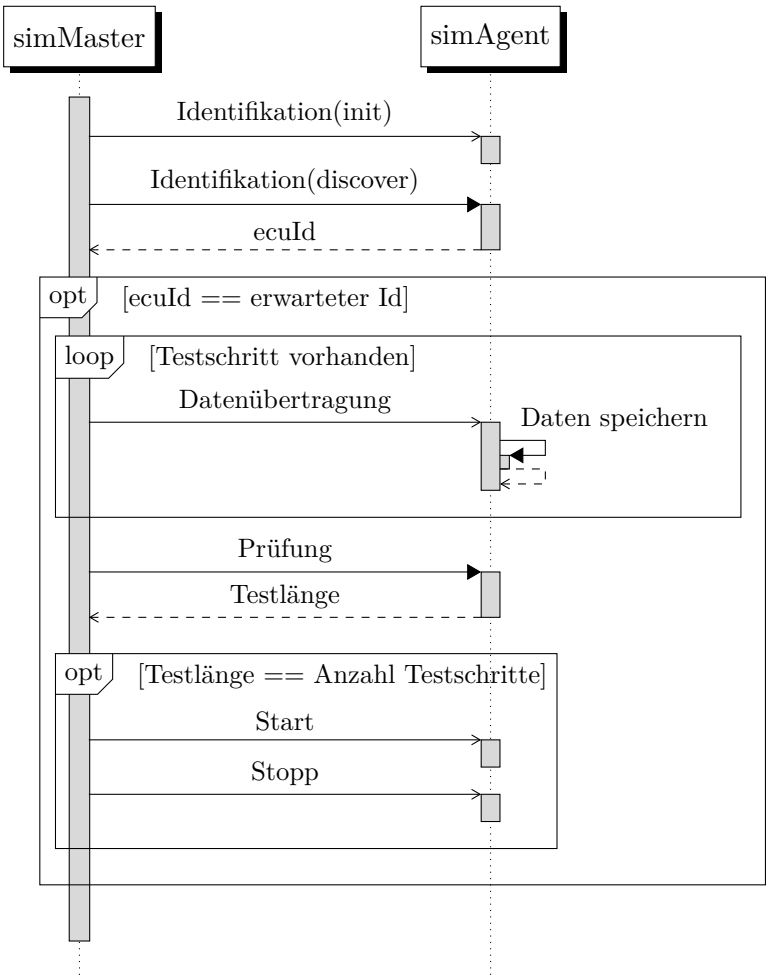


Abb. 3.8.: Sequenzdiagramm - Protokoll simMaster simAgent

Identifikation

dient der Identifikation und Initialisierung der im verteilten System befindlichen simAgents. Die simAgents sind analog der ECU-ID gelabelt und antworten auf die Identifikationsanfrage mit ihrer ID.

ID	0x255
Typ	Rx
Länge	2 Byte Nachrichten-Feld
Inhalt	Empfang des Initialisierungsbefehls des Masters

Frame-Aufbau `SIM_INIT_MASTER`:

Position	Typ	Beschreibung
0	Byte	ECUID
1	Byte	Kommando Init/Discover CMD

In Byte 1 werden die beiden Flags, Bit 8 für Initialisierung und Bit 9 für Discover, transportiert. Discover (`TRUE`) bewirkt eine Antwort des simAgent mit der Botschaft `SIM_INIT_AGENT`. Init bereitet die SWC auf den Empfang von Daten vor.

`SIM_INIT_AGENT` ist die Antwort des simAgents und somit die Bestätigung der Betriebsbereitschaft.

ID	0x258
Typ	Tx
Länge	1 Byte Nachrichten-Feld
Inhalt	Bestätigung der Initialisierungsanfrage des Masters

Frame-Aufbau `SIM_INIT_AGENT`:

Position	Typ	Beschreibung
0	Byte	ECUID

Datenübertragung dient der Verteilung der Daten in das verteilte System. Die Nachrichten müssen mit der Empfänger-ID = ECU_ID gekennzeichnet sein, so dass der entsprechende Empfänger die Daten speichert.

`SIM_DATA` dient der Übertragung der Testschritte vom simMaster zu dem über die in der Nachricht getaggten ECU-ID simAgent.

ID	0x256
Typ	Rx
Länge	8 Byte Nachrichten-Feld
Inhalt	ECU-ID und Testdaten

Frame-Aufbau `SIM_DATA`:

Position	Typ	Beschreibung
0	Byte	ECU-ID
1...7	Byte	Daten

Prüfung dient der Verifikation der übertragenen Simulationsdaten und wird vom simMaster angefragt. Zusätzlich besteht die Möglichkeit über den Value `LastChanID` und `LastSimVal` eine Prüfung durchzuführen, ob die Simulation korrekt abgeschlossen wurde.

`SIM_VERBOSE` ist eine detailreiche Antwortnachricht des simAgents, in der er z.B. die Anzahl der Testschritte zur Verifikation der Übertragung an den simMaster schicken kann.

ID	0x259
Typ	Tx
Länge	6 Byte Nachrichten-Feld
Inhalt	ECU-ID, Anzahl Testschritte und Verbose Flag (VER)

Frame-Aufbau `SIM_VERBOSE`:

Position	Typ	Beschreibung
0	Byte	ECU-ID
1...5	Byte	Daten

Start Stopp

dient der Übertragung des Testausführungszeitpunkts oder dem Abbruch eines laufenden Tests. Aufgrund seiner hohen Priorität für den Stopp müssen die Nachrichten die niedrigste ID haben.

`SIM_RUN` beinhaltet die Start bzw. Stopp Kommandos. Des Weiteren wird die Nachricht für eine Prüfung der Datenübertragung verwendet, indem das Kommando Verbose an den entsprechenden simAgent mit der ECU-ID, auch Verbose Identifier (VID) genannt, versendet wird.

ID	0x254
Typ	Rx
Länge	6 Byte Nachrichten-Feld
Inhalt	Start/Stopp/Verbose Kommando, Start Offset und VID

Frame-Aufbau `SIM_RUN`:

Position	Typ	Beschreibung
0	Byte	Start/Stopp/Verbose
1...4	Byte	Offset
5	Byte	Verbose ECU-ID (VID)

Die Kommunikation innerhalb des embedded Systems ist dem AUTOSAR Standard entsprechend eine Abfrage durch die in Kapitel 2.4.3 genannten Layer. Letztendlich greift der bisherige AUTOSAR-Treiber nicht auf die reale HW-Schicht unterhalb (Abbildung 2.7) zu, sondern auf die im AUTOSAR-Treiber hinzugefügte Schnittstelle (siehe Architektur Kapitel 3.3.3).

Das Sequenzdiagramm 3.9 zeigt die unidirektionale Kommunikation zwischen simAgent und dem SUT im embedded System.

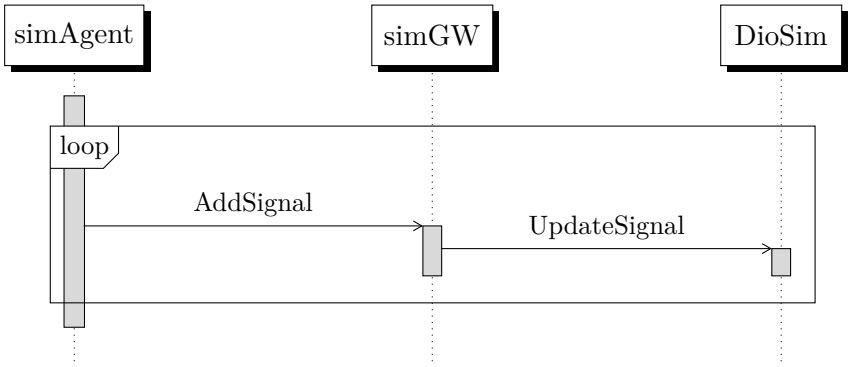


Abb. 3.9.: Sequenzdiagramm - Protokoll simAgent embedded System

AddSignal ist die Übergabe des aktuellen Signals aus dem Testablauf an das simGW im Complex Driver. Über den Funktionsaufruf `AddSignal(Channel, Value)` werden nur die KanalID und der zu simulierende Wert übergeben. Dieser Funktionsaufruf wird solange durchgeführt, solange im simAgent Testschritte zur Abarbeitung ausstehen.

UpdateSignal übergibt das Signal an den Treiber bzw. die hinzugefügte Treiberschnittstelle DioSim. Der Funktionsaufruf `UpdateSignal(Channel, Value)` übergibt analog dem Funktionsaufruf `AddSignal(Channel, Value)` nur die KanalID und den zu simulierenden Wert.

Anmerkung: Die beiden vorgestellten Kommunikationsprotokolle sind voneinander unabhängig. Es bedarf aber dem richtigen Scheduling, so dass die Kommunikation aus Abbildung 3.9 zu dem erwarteten Effekt im SUT führt. Welche Auswirkungen das Scheduling hat, wird in Kapitel 5.6 bzw. im Abschnitt 6.2.3 der Validierung näher ausgeführt.

3.4. Vergleich zu bestehenden Ansätzen

Ursprünglich war CCP, wie es der Name auch beschreibt, für die Kalibrierung von Steuergeräten während Laufzeit über den Transport Layer CAN gedacht, nun wird dieser auch für Automatisierungs- bzw. Testzwecke eingesetzt. Dieses Protokoll bietet den Entwicklern den lesenden und schreibenden Zugriff auf interne Steuergerätegrößen. Aufgrund seiner Verbreitung wurde dann im Rahmen der Weiterentwicklung von BUS-Technologien der nachfolgende Standard XCP erarbeitet, der den gleichen methodischen Ansatz verfolgt, sich aber von dem Transport Layer CAN löst.

Eine weitere Alternative für Testautomatisierung ist der Einsatz von Diagnosefunktionen nach Unified Diagnostic Services (UDS)-Standard. Diagnosefunktionen werden zu den bereits genannten Methoden auch im Rahmen software-basierter Automatisierungsmethoden für

Testzwecke genutzt. Durch die Ressourcenknappheit und Sicherheitsaspekte ist diese Methode der Nutzung von Diagnosefunktionen nicht sinnvoll. Außerdem müssen zusätzliche Funktionen in das Steuergerät implementiert und getestet werden.

Detaillierte Informationen zu UDS sind in Kapitel 3.4.3 aufgeführt.

3.4.1. CAN Calibration Protokoll

Das CCP wird nicht nur für Kalibrierung von Steuergerätefunktionen genutzt, sondern erfüllt auch viele entwicklungstechnische Funktionen wie z.B. die Flash-Programmierung. [CI] Die Nutzung von CCP ermöglicht den online Zugang zum Steuergerät und die Kalibration von Software-Modulen. Primären Einsatz findet das CCP im Bereich der Motorsteuerungsentwicklung von konventionellen Verbrennungsmotoren.

Das CCP ist eine Software-Schnittstelle zur Verbindung von Entwicklungswerkzeugen und dem ECU. Das CCP unterstützt eine Punkt-zu-Punkt-Verbindung sowie Verbindungen zu verteilten Komponenten. Das CCP ermöglicht das Schreiben in/auf RAM, Ports, ROM und Flash. Hinzu kommen Funktionen wie das Lesen von RAM, Ports und dem Flash von Software. [Kle]

Kommunikation

Die Basiskommunikation des CCP nutzt die Form einer Master-Slave-Architektur. Das Entwicklungstool, immer der Master, initiiert eine Kommunikation mittels einer CAN-Botschaft, der Slave, eine ECU antwortet mit einer CAN-Botschaft.

Die CCP-Slave Architektur ist in zwei Module aufgeteilt, den Command Processor und den Data Acquisition (DAQ) Processor. Der Kommando Prozessor erlaubt der ECU Kommandos (Command Receive Object (CRO)s) vom Master zu empfangen und darauf mittels Data Transmission Object (DTO) zu antworten.

Das zweite Modul des CCP-Slave, der bereits genannte DAQ-Prozesor, ist dafür zuständig, die angeforderte DAQ-Liste in angemessener Zeit in Form eines DTO zu schicken.

CRO wird vom Master an einen Slave versendet und erfordert eine Antwort in Form eines DTO, welches eine Command Return Message (CRM) beinhaltet.

Typ	Rx
Länge	8 Byte Nachrichten-Feld
Inhalt	Empfang von Kommandos im Slave

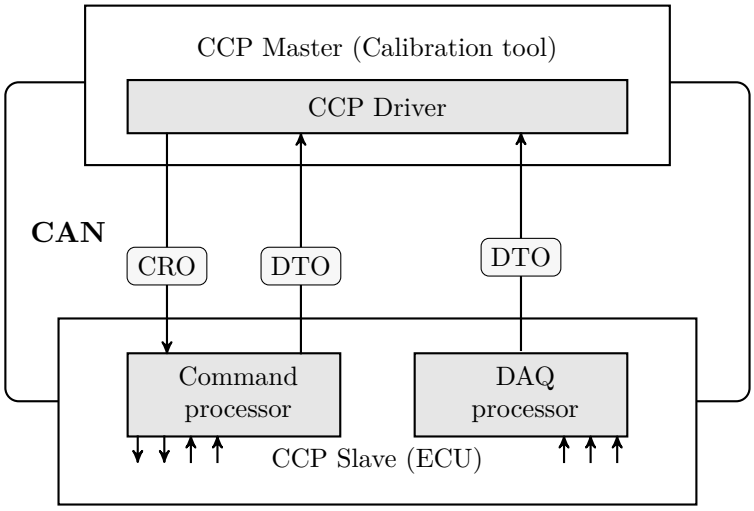


Abb. 3.10.: CCP-Kommunikation - Komponenten [Kle]

Frame-Aufbau CRO:

Position	Typ	Beschreibung
0	Byte	Command Code (CMD)
1	Byte	Command Counter (CTR)
2...7	Bytes	Kommando-Parameter und Daten

DTO beinhaltet alle vom Slave ausgehenden Daten als Datenpakete. Das erste Byte eines Datenpakets wird für die Paket-ID genutzt.

Typ	Tx
Länge	8 Byte Nachrichten-Feld
Inhalt	CRM $\vee$ Ereignis Nachricht $\vee$ DAQ

Frame-Aufbau CRM:

Position	Typ	Beschreibung
0	Byte	Datenpaket-ID (PID)
1	Byte	Kommando Fehler Antwort Code
2	Byte	CTR
3...7	Byte	Parameter und Daten

Frame-Aufbau DAQ-Nachricht:

Position	Typ	Beschreibung
0	Byte	Datenpaket-ID (PID)
1...7	Byte	Daten

Implementierung

Der CCP-Treiber muss auf jeder ECU implementiert sein, mit der interagiert werden soll. Ohne diesen ist eine Interaktion nicht möglich. Die Off-Board Tools sind in der Verantwortung das Kommunikationsprotokoll einzuhalten. In den ECUs wird nur das Antwortverhalten sowie eine Speicherung zusätzlicher Verbindungsdaten implementiert. [CI]

Initialisierung

Die Initialisierung der Kommunikation lässt sich in drei Elemente aufteilen: Das Einleiten der Verbindung mittels `Connect`, Versionsinformationen austauschen über `GET_CCP_Version` und schließlich der Austausch von Identifikationsinformationen durch `Exchange_ID`. [CI]

Connect

der CCP-Treiber beginnt den initialen Verbindungsaufbau mit einem *Connect*-Kommando des Masters, das eine Adresse des Slaves enthalten muss. Ist diese Adresse korrekt bestätigt der Slave die Verbindung.

Beispiel: Der Master versendet einen Connect-Befehl (Codes entnommen aus dem CCP-Standard [Kle])

Byte0	1	2	3	4	5	6	7
0x01	0x45	0x00	0x02				

Der Slave antwortet mit einem DTO dieser beinhaltet Acknowledge (0x00) und dem CTR des CRO

Byte0	1	2	3	4	5	6	7
0xFF	0x00	0x45					

Get_CCP_Version

es folgt eine Versionsprüfung mittels *GET_CCP_VERSION*, um die Kompatibilität der Implementierung sicher zu stellen.

Beispiel: Der Master versendet das *GET_CCP_VERSION*-Kommando zum Slave (CTR des CRO ist aktuell bei 0x27)

Byte0	1	2	3	4	5	6	7
0x1B	0x27	0x02	0x01				

Der Slave antwortet mit einem DTO, dieser beinhaltet Acknowledge (0x00) und die implementierte Version 2.1 (0x02) (0x01).

Byte0	1	2	3	4	5	6	7
0xFF	0x00	0x27	0x02	0x01			

Exchange_ID

Das Kommando kann für eine automatische Session-Konfiguration verwendet werden.

Beispiel: Der Master versendet die Echange_ID zum Slave (CTR des CRO ist aktuell bei 0x23)

Byte0	1	2	3	4	5	6	7
0x17	0x23						

Der Slave antwortet mit einem DTO, dieser beinhaltet Acknowledge (0x00), die Länge (0x04) der Slave-ID und dessen Datentyp (0x02). Weitere Daten können per Upload ermittelt werden, die Ressourcenverfügbarkeit (0x03) und der Status des Ressourcenschutzes (0x03).

Byte0	1	2	3	4	5	6	7
0xFF	0x00	0x23	0x04	0x02	0x03	0x03	

Datenaustausch

Der Master kann mittels Start/Stopp Kommando einen Datenaustausch von den Slaves veranlassen, welche auf diese Anfrage hin die definierten Daten in DAQ-DTOs versenden. Datenelemente in den ECUs sind einer s.g. Object Descriptor Table (ODT) zugewiesen. Diese Tabelle hält alle Adressen, Adresserweiterungen und die Länge der jeweiligen Datenelemente bereit. Ein ODT kann bis zu sieben Elemente referenzieren. All diese referenzierten Elemente können zum Master übertragen werden. Die Packageidentifier (PID) ist referenziert auf die ODT ($0 \leq n \leq$ 0xFD). [Kle]

START_STOP

Um eine Datenübertragung einzuleiten wird das START_STOP Kommando durch den Master verwendet.

Beispiel: Der Master versendet das START_STOP CRO zum Slave (CTR ist aktuell bei 0x23). Das Start/Stopp Byte ist (0x01), die Nummer der DAQ-Liste ist die (0x03) und das zu übertragende Paket ist die (0x07). Die ECU soll den Kanal (0x02) mit einem Format [engl. Prescaler] (0x01) *Motorola-Format* verwenden.

Byte0	1	2	3	4	5	6	7
0x06	0x23	0x01	0x03	0x07	0x02	0x00	0x01

Der Slave antwortet mit einem DTO, dieser beinhaltet Acknowledge (0x00) und den CTR.

Byte0	1	2	3	4	5	6	7
0xFF	0x00	0x23					

WRITE_DAQ

Beispiel: Der Master versendet einen WRITE_DAQ an den Slave. Die Länge des Datenelements ist (0x02), die Adresserweiterung (0x01) und die 32-Bit Adresse ist (0x02004200)

Byte0	1	2	3	4	5	6	7
0x16	0x23	0x02	0x01	0x02	0x00	0x42	0x00

Der Slave antwortet mit einem DTO, dieser beinhaltet Acknowledge (0x00) und den CTR.

Byte0	1	2	3	4	5	6	7
0xFF	0x00	0x23					

Datenverbindung beenden

Disconnect

Trennt die Verbindung zum Slave. Trennung der Verbindung ist unabhängig von der Session, diese kann weiterhin bestehen.

Beispiel: Der Master versendet einen Disconnect an den Slave. Der CTR ist aktuell bei (0x23), das Parameter Byte (0x00) *temporär* und die Slave Adresse (0x0208)

Byte0	1	2	3	4	5	6	7
0x07	0x23	0x00		0x08	0x02		

Der Slave antwortet mit einem DTO, dieser beinhaltet Acknowledge (0x00) und den CTR.

Byte0	1	2	3	4	5	6	7
0xFF	0x00	0x23					

Detaillierte Informationen und Beispiele über das CCP können dem CCP-Standard [Kle] entnommen werden.

Unterschied zu sim-Ansatz

Vergleicht man CCP mit dem sim-Ansatz fallen bezüglich des Kommunikationsprotokolls und der Architektur Ähnlichkeiten auf. Ähnlich ist der generelle Aufbau aus Off-Board Master und den in die ECU implementierten Slaves. Daher ist auch die Kommunikationsart in der Form einer Single-MasterMulti-Slave Kommunikation ähnlich zum sim-Ansatz, bei der die Slaves rein antwortende Komponenten sind.

Die Implementierung in Form eines Treibers ähnelt dem sim-Ansatz, unterscheidet sich aber dadurch, dass die Ebene der Interaktion bei CCP die Steuergeräteabstraktionsschicht ist bzw. eine höhere ist. Der sim-Ansatz mit der Integration im AUTOSAR-Treiber bezieht sich hingegen auf die Mikrokontrollerabstraktionsschicht.

Der größere Unterschied der beiden Ansätze ist bei CCP die offene Kommunikationsverbindung beim Verstellen von internen Steuergerätewerten. Der sim-Ansatz unterscheidet sich hierbei dadurch, dass die Daten im Speicher der ECU abgelegt werden und nach einem *Start*-Kommando in das SUT über die Treiberschnittstelle eingespielt werden. Die Variante der offenen Verbindung des CCP-Ansatzes hat dagegen einen fortlaufenden Datenaustausch über CAN.

Dieser elementare Unterschied ist Ziel des Ansatzes und soll Vorteile beim Testen durch eine bessere Nähe zum Kundennutzen bringen, bei dem auch keine zusätzliche Kommunikation über den BUS stattfindet. Die Auswirkungen dieser Art der Signalwertsimulation, Vor- und Nachteile, werden im Kapitel 6 gezeigt.

3.4.2. XCP

Das Kommunikationsprotokoll XCP dient zum Messen und Verstellen steuergeräte-interner Größen während der Laufzeit. Vorteil gegenüber dem Vorgängerprotokoll CCP ist die Unabhängigkeit von der physikalischen Transportschicht. Über ein Zweischichtenprotokoll trennt XCP die Protokollschicht von der Transportschicht und nutzt ein Single-MasterMulti-Slave-Konzept. [Pat07] XCP ist somit in der Lage, auf Basis verschiedener Transportschichten dieselbe Protokollschicht zu verwenden.

CCP und XCP sind beginnend mit einem Connect, Start/Stopp den entsprechenden Kommandos bis zum Disconnect in ihrem Ablauf gleich. Ein Unterschied findet sich darin, dass sich das XCP-Protokoll von dem Transport Layer und somit der CAN-Konvention löst und damit einen allgemeinen Frame definiert. Der XCP-Frame besteht immer aus XCP-Header, XCP-Packet und XCP-Tail. XCP-Header und XCP-Tail sind folglich abhängig von der Transportschicht.

Die Kommunikation über das XCP-Packet unterteilt sich in Command Transmission Object (CTO)s, in dem Kommandos versendet werden. In den s.g. DTOs werden Daten versendet bzw. ausgetauscht. Die Abbildung 3.11 zeigt mögliche Kommunikationselemente des XCP-Kommunikationsmodells.

Über die CTOs werden Kommandos an die Slaves übertragen sowie Antworten von den Slaves an den Master. Der Unterschied zum CCP zeigt sich bereits in der Begrifflichkeit von: CRO vs. CTO. Es wird nicht mehr von Command Receive Objects also Empfangs-Objekten, sondern von Command Transmission Objects also Übertragungs-Objekten gesprochen. Der Versand der Objekte ist bei XCP bidirektional.

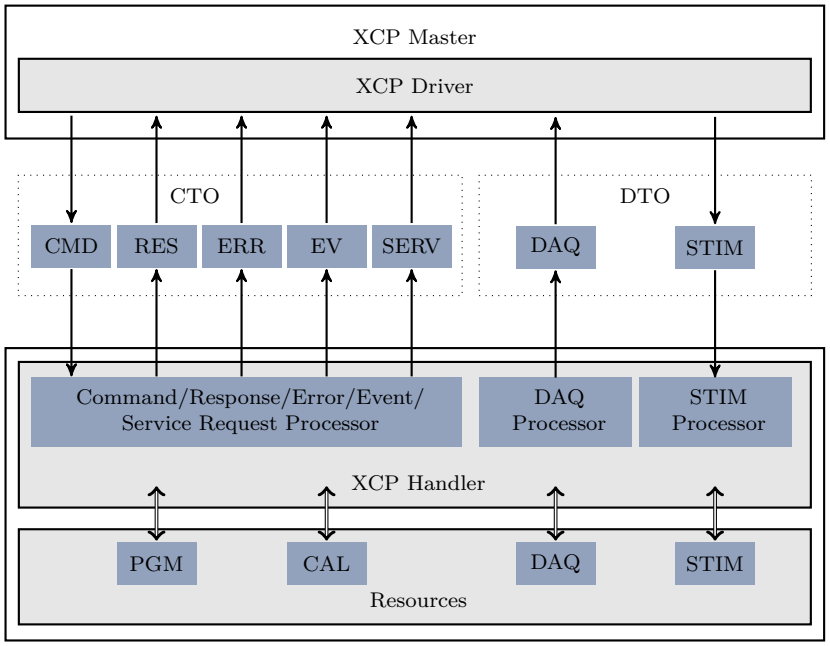


Abb. 3.11.: XCP-Kommunikation - Komponenten [Vec]

Frame-Aufbau CTO (Master → Slave)

Position	Typ	Beschreibung
0	Byte	Command Packet Code CMD
1...MAX_CTO-1	Byte	Commando specific Parameters

Über dieses CTO können nun folgende Antworten der Slaves versendet werden. Im Gegensatz zum CCP bei dem das Kommando nur bestätigt oder abgelehnt werden kann.

Frame-Aufbau CTO (Slave → Master) positive Antwort

Position	Typ	Beschreibung
0	Byte	Command Positive Response Code
1...MAX_CTO-1	Byte	Commando specific Parameters

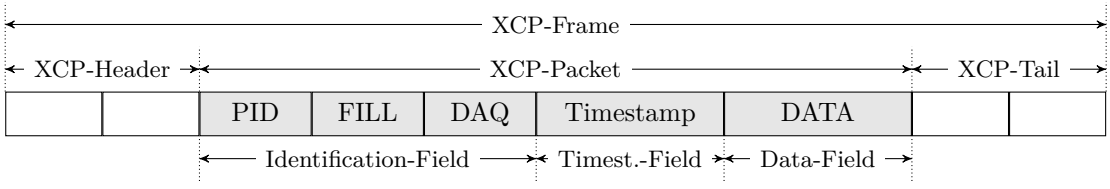


Abb. 3.12.: Aufbau XCP-Frame

Frame-Aufbau CTO (Slave → Master) negative Antwort

Position	Typ	Beschreibung
0	Byte	Error Packet Code
1	Byte	Error Code
2...MAX_CTO-1	Byte	Commando specific Parameters

Connect

Der Verbindungsaufbau beginnt mit einem *Connect*-Komando des Masters.

Beispiel: Der Master versendet einen Connect-Befehl (0xFF) mit dem Parameter Mode (0x00), dies bedeutet, dass der Master eine XCP-Kommunikation aufbauen möchte. [Vec]

Byte0	1
0xFF	0x00

Der Slave antwortet in diesem Beispiel mit einer positiven Antwort, diese kann je nachdem etwas umfangreicher ausfallen. Dabei werden Parameter, die kommunikationsspezifische Informationen enthalten an den Master gesendet. Mit optionalen Daten wie z.B. MAX_CTO und MAX_DTO teilt der Slave die jeweiligen maximalen Paketlängen mit.

Byte0	1	...	...	...	MAX_CTO-1
0xFF	0x00	...	...	...	...
		Command specific Parameters			

DTO-Objekte werden verwendet, um Messdaten vom Slave zum Master und Verstelldaten vom Master zum Slave zu übertragen. Diese Kommunikation ist in zwei Phasen aufgeteilt. Durch eine entsprechende Initialisierung teilt der Master dem Slave mit, eine Messung zu starten, ab diesem Zeitpunkt versendet der Slave die Daten an den Master, bis er ein Stopp-Kommando erhält. DTO-Pakete nutzen Zeitstempel, bei der Übertragung einer CTO-Botschaft ist dies nicht möglich. Mit dem Zeitstempel gibt der Slave den Messwerten eine Zeitinformation mit.

Per Stimulation (STIM) versendet der Master Daten an den Slave, hierbei ist die Kommunikation analog in zwei Phasen aufgeteilt. Bei der Initialisierung teilt der Master mit, welche Daten er senden will, gefolgt von dem Upload der Daten zum Slave mittels der DTOs. Durch ein entsprechendes STIM-Ereignis im Slave werden die Daten in den Speicher übertragen.

Die Zuordnung, wie die Daten in den jeweiligen Slaves wieder zusammensetzen soll, erfolgt über ODTs. Diese enthalten die Objektadresse und die Objektlänge. Genauer gesagt, referenziert ein Eintrag in einer ODT-Liste auf einen Speicherbereich im Random Access Memory (RAM) über die Adresse und die Länge des Objektes. [Vec] Je nach Transportmedium hat jedes Paket eine maximale Länge an Nutzbytes, im Falle von CAN sind das sieben Bytes.

STIM

mit diesem Kommando versendet der Master Daten zum Slave. Diese Kommunikation besteht aus zwei Phasen: In der Initialisierungsphase teilt der Master dem Slave mit, welche Daten er an den Slave senden wird. Nach dieser Phase schickt der Master die Daten an den Slave und der STIM-Prozessor speichert die Daten.

Sobald ein entsprechendes STIM-Ereignis im Slave ausgelöst wird, werden die Daten in den Speicher der Anwendung übertragen. Es existiert hierbei keine Synchronisierung zwischen dem Versenden von Daten und dem Ereignis, auf das das Steuergerät triggert. STIM verwendet DTOs für die Übertragung von Daten in das Steuergerät. Um die Verstellgrößen zu kennen, werden DAQ-Listen genutzt.

Die Adressierung der Pakete wird nun erweitert, da eine einfache Adressierung über PID nicht mehr ausreichend ist. Aus diesem Grund werden die ODT-Nummern benötigt, da diese über alle DAQ-Listen eindeutig sind.

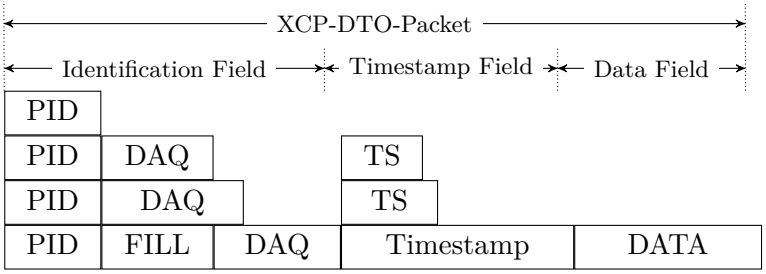


Abb. 3.13.: Aufbau XCP-DTO-Packet

XCP unter CAN

Das XCP wurde als Nachfolger von CCP entwickelt und muss daher als Nachfolger CAN bedienen können. Die Kommunikation über den BUS ist in einer Datenbank (DBC) oder AUTOSAR-Format AUTOSAR Extensible Markup Language (ARXML) beschrieben.

Um möglichst wenige CAN-IDs zu belegen, beschränkt sich XCP auf zwei CAN-IDs - ID 554 für Tx-Botschaften von Master zu Slave, ID 555 für Rx-Botschaften Slave zu Master. Diese CAN-IDs dürfen nicht anderweitig belegt sein.

Ein Standard-CAN-BUS überträgt maximal acht Nutz-Bytes pro Botschaft und ergibt daher folgenden Frame, wie er auf Abbildung 3.14 dargestellt ist. Byte eins wird für die Übertragung des Befehls bzw. Commands verwendet. Byte zwei bis sieben dienen der Nutzdatenübertragung.

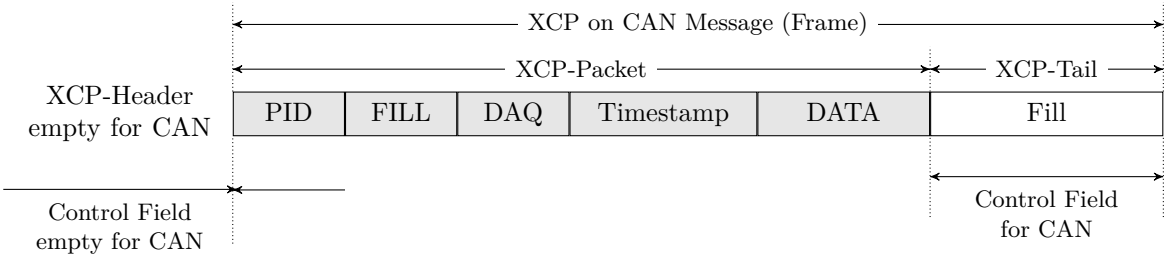


Abb. 3.14.: Aufbau XCP on CAN Frame [Vec]

Alle verfügbaren Kommandos und ausführlichere Beispiele können der Spezifikation [Vec] entnommen werden.

Unterschied XCP und simAgent

Im direkten Vergleich sind CCP und XCP methodisch gleich. Dies führt dazu, dass auch beim XCP der Kommunikationskanal während der Interaktion bzw. Verstellung von Steuergerätegrößen offen ist. XCP bietet gegenüber dem bisherigen CCP den Vorteil der Möglichkeit der verschiedenen Transport Layer. Dies ist auch für den sim-Ansatz möglich, dafür müssen auf den verwendeten Transportschichten entsprechende Signale definiert werden.

Das AUTOSAR XCP-Modul ist nach Spezifikation [AUTa] in die Communication Services zu implementieren. Die Abbildung 3.15 aus [AUTh] zeigt nochmals detailliert die Implementierung (durchgezogene Pfeile: Daten-Pfad; gestrichelte Pfeile: Control-Pfad). Damit wirkt XCP knapp unter der RTE ein und bedarf einer bereits sehr steuergerätespezifischen Implementierung. Der sim-Ansatz greift, wie in Kapitel 3.3.3 beschrieben, dagegen direkt in der Treiberschicht ein. Die Auswirkungen der XCP-Implementierung auf andere Module werden durch die Header File Struktur aus [AUTh] Kapitel 5.9.2 deutlich.

Zusammengefasst finden sich in erster Linie Unterschiede auf der Ebene der Implementierung und damit der Art der Signale, auf die eingewirkt wird. Damit stimuliert der simModul-Ansatz längere Wirkketten und mehr Schichten der AUTOSAR-BSW. Methodisch unterscheidet sich der XCP-Ansatz von simModul Ansatz durch die kontinuierliche Datenübertragung von XCP-Master zu XCP-Slave.

Hieraus ergeben sich grundlegend die gleichen Vor- und Nachteile von XCP vs. simModul-Ansatz, wie sie bereits in Kapitel 3.4.1 aufgeführt wurden.

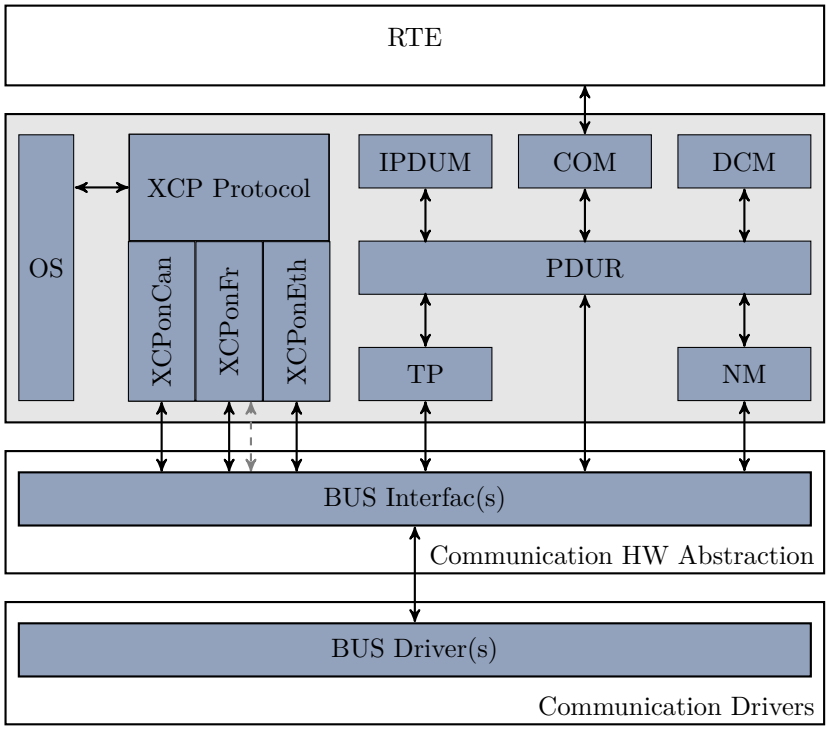


Abb. 3.15.: Funktionale Integration XCP in AUTOSAR [AUTh]

3.4.3. Diagnose Funktionen

Ursprünglich aus dem griechischen *diágnosis*, bedeutet Diagnose im Automotive Bereich, die Erkennung bzw. Befundung von Elektrik- und Elektronik-Fehlern im Fahrzeug. Diese technischen Verfahren dienen rein der Fehlererkennung und Fehleranalyse. Dabei wird grundlegend in zwei Kategorien On-Board- und Off-Board-Diagnose unterschieden.

On-Board-Diagnose Diagnosefunktionen im SUT zur Detektion und Protokollierung von Fehlern.

Off-Board-Diagnose Werkzeuge und Funktionen zur Unterstützung und Interpretation der Informationen aus der On-Board Diagnose.

Diagnose beschränkt sich nach aktuellem Stand auf die Transportschichten CAN und Eth. Zur Standardisierung der Kommunikation zwischen Fahrzeug und externem Testequipment wurde von der International Organization for Standardization (ISO) die ISO 15765 (Beschreibung der physikalischen Schnittstelle) und die ISO 14229 (Beschreibung der Services) geschaffen.

Der Aufbau der CAN Botschaften gestaltet sich wie folgt und wird in unsegmentierte Single Frame (SF) und segmentierte First Frame (FF)/Consecutive Frame (CF) unterschieden.

Single Frame — unsegmentierte Botschaften werden als SF im ersten Byte der s.g. Protocol Control Information (PCI) gekennzeichnet. Dieser Header unterteilt sich in Bit 7...4 für die Typ-Kennzeichnung (0x0) für SF und Bit 3...0 für die Angabe der Anzahl der folgenden Bytes.

FF/CF Frame — Segmentierte Botschaften werden in FF und CF unterteilt. Die erste Botschaft erhält im PCI Bit 7...4 die Kennzeichnung (0x1) für FF. Sowie in Byte 1 Bit 3...0 und Byte 2 Bit 7...0 die Anzahl der folgenden Bytes der gesamten zusammenhängenden Botschaft. (max. 4095).

Die Folgebotschaften sind in Byte0 Bit 7...4 mit CF (0x2) gekennzeichnet und haben in Bit 3...0 eine fortlaufende Sequenznummer (SN). Die Nutzdaten werden bei jeder CF auf das maximale (7 Bytes) ausgereizt.

Diagnose bietet zwei verschiedene Adressierungsverfahren an, um die ECUs im SUT zu erreichen.

physikalische Adressierung — Das Off-Board-System spricht genau ein Steuergerät im verteilten System an (1:1 Verbindung) .

funktionale Adressierung — Das Off-Board-System spricht immer alle Steuergeräte im verteilten System an (1:n Verbindung).

Funktionale Diagnose-Services können nur mit Hilfe einer Steuergeräte-Beschreibungsdatei (SGBD) erfolgen. Innerhalb einer SG-Familie müssen gleiche Größen in der SGBD auch gleiche Job- und Result-Namen haben (Standardisierte Diagnoseaufträge).

Eine SGBD-Beschreibungsdatei enthält alle Informationen über die relevanten Datenobjekte im Steuergerät wie z.B. Kenngrößen (Parameter, Kennlinien, Kennfelder), reale und virtuelle Messgrößen und Variantenabhängigkeiten. Für jedes dieser Objekte werden Informationen wie Speicheradresse, Ablagestruktur, Datentyp und Umrechnungsvorschriften zur Wandlung in physikalische Einheiten benötigt.

Grundlegende funktionale Services sind:

- ▷ Steuergeräte-Identifikation (Steuergeräte-ID)
- ▷ Software-Kontainer-Identifikation (entspricht einer SW-Versionsabfrage)
- ▷ Primärer Fehlerspeicher lesen - über die Funktion *Fehlerspeicher lesen*
- ▷ Sekundärer Fehlerspeicher lesen - ist den Entwicklern vorbehalten
- ▷ Zugriff auf Steuergeräteein- und -ausgänge (Readonly)
  (Analoggrößen sind auf SI-Einheiten zu normieren. Ausgelesene Eingangsgrößen, die an den Tester gesendet werden, dürfen keine Ersatzwerte sein. Fehlerhafte Eingangs/-Ausgangsgrößen müssen eindeutig gekennzeichnet sein (Fehlerwert festlegen). Eine

Verwechselung mit gültigen Werten muss ausgeschlossen sein. Ausgangsgrößen müssen auch dann korrekt ausgelesen werden können, wenn sie vom Tester vorgegeben werden.)

UDS ist ein Diagnoseprotokoll zur Vereinheitlichung von Diagnosekommunikation im Automotive Umfeld, entstanden ist dieses aus der ISO 14230-3, aktuell für den Automotive Bereich im ISO 14229 [Int13] spezifiziert. Ziel von UDS ist es, alle Steuergeräte im Fahrzeug kontaktieren zu können. Dabei ist UDS unabhängig von der Transportschicht und spielt sich im fünften und siebten Layer des Open Systems Interconnection (OSI)-Schichtenmodells ab.

Die UDS-Kommunikation ist im Standard [Int13] wie folgt definiert: Das ServiceID-Feld klassifiziert die Funktionsgruppe, gefolgt von einem Parameter- und Datenfeld. Ein Dienst wird durch den Client initiiert und wird durch eine positive oder negative Antwort beendet.

UDS bietet auch eine Möglichkeit auf die Steuergeräteantwort zu verzichten. Dies erfolgt normalerweise bei Broadcast Nachrichten.

Drei Service IDs, die im UDS spezifiziert sind ermöglichen den schreibenden Zugriff im Steuergerät und somit die Signalmanipulation.

- `0x2E` Write Data By Identifier, ermöglicht das Schreiben von Werten im Steuergerät. Das Steuergerät prüft hierbei Format und Länge.
- `0x3D` Write Memory By Address, ermöglicht das Schreiben kleinerer Datenmengen direkt in den Speicher. Über diesen Dienst ist auch ein Löschen des Speichers durch Überschreiben möglich.
- `0x2F` Input Output Control by Identifier, ermöglicht ein Eingreifen auf Signale bzw. Ein- oder Ausgänge des Steuergeräts.

Diese Dienste können für eine Manipulation von Steuergeräte Ein- und Ausgangssignalen verwendet werden. Wird ein zeitlich längerer Zugriff benötigt, muss zyklisch eine Botschaft `0x3E` Tester Present an die Komponenten verschickt werden, um die Verbindung aufrecht zu erhalten. [Int13]

Diagnose vs. simAgent

Diagnose Funktionen erfordern nach ihrem methodischen Ansatz eine funktionale Umsetzung im Steuergerät. Es sind daher sehr spezifische Funktionen, die im Falle von UDS einem einheitlichen Kommunikationsprotokoll folgen. Diese implementierten Funktionen müssen jeweils dem Anwendungsfall angepasst oder erweitert werden.

Der simAgent Ansatz verfolgt das Ziel einer universellen Schnittstelle und differenziert sich somit stark. Der simAgent Ansatz verwendet nur gewisse Elemente der Diagnose, wie z.B. einen ähnlichen Ansatz der Steuergeräteidentifikation oder ein Multiplexverfahren für die Übertragung von umfangreicheren Datenmengen.

Betrachtet man UDS, ist dieses Protokoll schon sehr naheliegend für eine Signalmanipulation mit Hilfe des `0x2F` Input Output Control by Identifier Dienstes. UDS nutzt eine standardisierte Kommunikation, ein standardisiertes Modul sowie einheitliche Services. Im Gegensatz zur normalen Diagnose, bei der eigentlich nur immer eine steuergerätespezifische Send and Response Kommunikation stattfindet, müsste man beim Nutzen des UDS-Protokolls die Datenverbindung zum Steuergerät offen halten, um ganze Signalfolgen darzustellen.

Der simAgent Ansatz verfolgt somit bezüglich der Kommunikation sehr ähnliche Ansätze zu UDS, da nur durch Standardisierung und Vereinheitlichung eine Simulation auf mehreren Steuergeräten stattfinden kann. UDS bestätigt in den Aspekten der Kommunikation und der Art der Umsetzung in den Steuergeräten als Standardmodul den simAgent Ansatz.

4. Integration

Die Integration in der Softwaretechnik beschreibt die Verknüpfung und das Zusammenspiel der einzelnen Anwendungen. [SZ13] In diesem Fall beschreibt die Integration die Verknüpfung der Applikation-SWCten der beiden SW-Architekturen (verteiltes System Fahrzeug und verteiltes System der simAgents) inklusive der Schnittstelle im Treiber und der Off-Board Komponente simMaster.

4.1. Systemarchitektur des verteilten Systems

AUTOSAR strebt eine Standardisierung der Schnittstellen und Komponenten sowie der Service-Routinen an. Die Systemarchitektur beschreibt die Integration der funktionalen Software-Architektur und der Hardware-Architektur. Die beiden funktionalen Architekturen, das Fahrzeug als vermaschte Topologie, das simModul als BUS-Topologie, ergeben in Kombination eine vermaschte Topologie. Die abstrakte Sicht der BUS-Topologie des simModuls löst sich nach der Integration scheinbar auf.

Die Abbildung 4.1 zeigt einen Ausschnitt aus der Gesamttopologie ohne den simMAster. Dieser ist über das Gateway (GW) mit der Architektur verbunden.

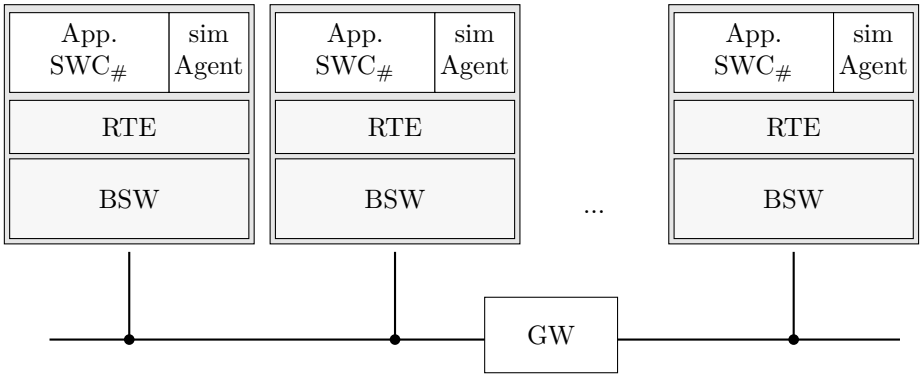


Abb. 4.1.: Generische Systemarchitektur - AUTOSAR inkl. BUS

4.2. Umsetzungsaspekte

Die Integration von simModulen in ein bestehendes Gesamtsystem erfordert ein strukturiertes Vorgehen. Hierbei werden die einzelnen Module in das jeweilige ECU integriert und nutzen dann die bestehenden Kommunikationspfade der ECUs. Im Folgenden werden verschiedene Aspekte Bottom-up aufgeführt, die letztendlich zur Auswahl der ZBE für die Integration geführt haben.

Die Integration in die AUTOSAR Software – stellt sich als Herausforderung dar, da keine AUTOSAR-BSW eines Seriensteuergeräts für die Integration beim Original Equipment Manufacturer (OEM) zur Verfügung stand. Dies ist der aktuellen Produktentwicklungsstrategie geschuldet, bei der die ECUs in der Regel von einem Lieferanten an den OEM geliefert werden, die bereits die entsprechenden konfigurierten AUTOSAR-BSW integriert haben. Der OEM kann folglich mit den abstrahierten Signalen aus der RTE seine Software entwickeln.

Für die Umsetzung muss daher eine Komponente gefunden werden, die mit einer alternativen Hardware ersetzt und auf der die in Kapitel 3 genannte Methodik umgesetzt werden kann. Im Folgenden werden die minimalen Anforderungen identifiziert, die für eine Umsetzung der Methodik gefordert werden, dies beinhaltet unter anderem die Auswahl der Kriterien an Funktionen je Testebene.

Anforderungen, die die ECU erfüllen muss:

- ▷ Prozessor — Das Steuergerät verfügt über eine AUTOSAR (Version 4.0.x oder höher) kompatible CPU.
- ▷ Schnittstellen — Digital bzw. Analog HW-Eingänge und eine Kommunikationsschnittstelle, idealerweise CAN.
- ▷ Basissoftware — Die AUTOSAR-BSW ist als Sourcecode verfügbar und frei konfigurierbar (inkl. Tooling bzw. Compiler).

Die Integration in ein Teilsystem soll die Funktionswirkketten der verteilten Funktion zwischen der minimalen Anzahl an Steuergeräten aufzeigen. Diese Wirkketten richten sich stark nach der gewählten Komponente bzw. Funktion bezüglich ihrer Ausprägung in Bezug auf Länge und Vernetzung.

Anforderungen, die die Funktion im Teilsystem erfüllen muss:

- ▷ Funktion — statische deterministische Funktionsumfänge mit begrenzten Umfang.
- ▷ Auswirkung — keine Funktionsumfänge mit direkten Auswirkungen auf z.B. höhere elektrische Spannungen.

Die Integration in ein Gesamtsystem soll die Wechselwirkungen im verteilten System unter realen Bedingungen zeigen.

Anforderungen die die Funktion im Gesamtsystem erfüllen muss:

- ▷ Funktion — hoher Grad an Vernetzung mit anderen Funktionen mit direktem und indirektem Steuerungsumfang.
- ▷ Auswirkung — keine sicherheitskritischen Funktionsumfänge, wie z.B. höhere elektrische Spannungen.
  keine fahrfreigaberelevanten Funktionsumfänge, wie z.B. Licht- oder Antriebsfunktionen.

Werden alle aufgeführten Anforderungen berücksichtigt, grenzt sich die Auswahl an möglichen ECUs stark ein. Aufgrund der Sicherheits- und Fahrfreigabeanforderungen bleibt an sich nur die Domäne Infotainment- und Karosseriefunktionen (IuK). Diese Domäne beheimatet den gesamten Anzeigeverbund, die Kommunikationssysteme und die zugehörigen Bedienelemente. Für die Implementierung, Darstellung und Validierung der Methodik genügt eine relativ einfache ECU die den Vorteil der einfacheren Implementierung mit sich bringt. Folglich wurde aus der genannten Domäne IuK die ZBE ausgewählt, da sie folgende Vorteile mit sich bringt.

- ▷ Funktion — Einfache Funktionslogik (keine Regelungsfunktionen) mit einem Anteil an Funktionen des verteilten Systems, wie z.B. Steuerung des Audiosystems, des Navigationssystems, sowie die Möglichkeit der Steuerung von Klima und Hochvolt-Ladefunktionen.
- ▷ Schnittstellen — Direkte Kundenschnittstelle (nur System-Input-Funktionalität) sowie eine CAN BUS-Schnittstelle zum verteilten System.
- ▷ Auswirkung — Ermöglicht das Steuern von Kundenfunktionen mit z.B. direkter Rückmeldung über die HMI. Keine direkten Auswirkungen auf fahr- oder sicherheitsrelevante Funktionsumfänge.

Mit der Auswahl der ZBE als Zielkomponente ergibt sich ein sehr anschaulicher Funktionsumfang im Infotainment mit den Möglichkeiten der direkten Interaktion z.B. über das Central Information Display (CID) (siehe Kapitel 4.4). Im Folgenden wird die Hardware für die Implementierung bestimmt. Vernetze Funktionen, wie die Steuerung von Klimafunktionen können im Gesamtsystem in allen Fahrzeugzuständen (Parken und Fahren) ausgeführt werden. Ein direkter Vergleich von realen und simulierten Kundeneingaben ist über die Methode sehr anschaulich möglich.

4.3. Übertrag auf generisches Steuergerät

Um eine Validierung der vorgestellten Methode zu ermöglichen wurde das ZBE-Seriensteuergerät aus dem aktuellen Systemverbund eines BMW 7er[1] aufgrund seines Funktionsumfangs ausgewählt (Kapitel 4.2) und auf ein generisches Steuergerät transferiert. Dies ist erforderlich, da die Integration des simModuls auf der Serienhardware aufgrund von fehlender Verfügbarkeit der AUTOSAR-BSW nicht möglich war (Kapitel 4.2). Durch die Portierung der Application-SWC-ZBE auf die generische ECU entsteht die Möglichkeit die AUTOSAR-BSW zu modifizieren und das simModul zu integrieren.

4.3.1. Beschreibung des generischen Steuergeräts

Das gewählte System besteht aus einem Personal Computer (PC) mit präemptiven Echtzeit-Linux Kernel und einem AUTOSAR Basic Software Stack der Firma Mentor Graphics. Das System erfüllt die unter Kapitel 4.2 aufgeführten Anforderungen an eine Hardware.

Um eine Verbindung zu den Fahrzeugschnittstellen zu ermöglichen, werden PC-Karten (wie z.B. CAN- oder FlexRay-Karten) eingesetzt. Die Implementierung von Micro Controller Abstraction Layer (MCAL)s gewährleistet, dass diese Karten auch im AUTOSAR-Umfeld eingesetzt werden können. Die Konfiguration von MCAL und BSW wird über eine gelieferte Entwicklungsumgebung realisiert. [MBt]

Als Entwicklungsplattform wurde eine F19P - 3U CompactPCI® PlusIO Intel® Core™ 2 Duo CPU Board Karte verwendet.

Prozessor	Intel® Core™ 2 Duo SP9300, 2.26 GHz 1066 MHz System bus frequency
Memory	Up to 6 MB L2 cache integrated in Core 2 Duo 16 Mbits boot Flash

Das System, auch Virtuelle Absicherungs Plattform (VAP) genannt, bietet ein AUTOSAR-kompatibles Runtime Environment (RTE) und die Integration des VSTAR Basic Software Stack von Mentor Graphics und Elektrobit's Tresos ACG (AutoCore Generic) mit jeweiliger Unterstützung von AUTOSAR 3.1.4, 3.2.2 und 4.0.3. Die vollständigen technischen Daten können dem Datenblatt [MBt] entnommen werden.

[1] Entwicklungsbaugruppe 35up / Modell G1x / Produktanlauf 11/2016

4.3.2. Aspekte Pro/Contra

Die Implementierung auf das in Kapitel 4.3.1 genannte generische Steuergerät bietet den entscheidenden Vorteil des internen Knowhows. In einer Gegenüberstellung wird schnell klar, dass zwar beide Lösungen, die originale ZBE-Hardware genannt *BMW-Preh* und die VAP, technisch machbar sind, aber für den Forschungszweck ein generisches Steuergerät besser geeignet ist. Das generische Steuergerät ist für den Zweck der Variabilität entwickelt worden und kann im Gegensatz zur ZBE für verschiedene Anwendungsfälle eingesetzt werden.

Kriterium	BMW-Preh	VAP
HW verfügbar	✓	✓
MCAL	✓	✓
BSW-Quellcode verfügbar	✓	✓
Debuging	✓	✓
Know-How verfügbar	×	✓
SDK verfügbar	✓	✓
SDK-Komplexität	mittel	niedrig

Tab. 4.1.: Vergleich HW-Lösungen

Ein entscheidender Vorteil für die Umsetzung auf der VAP ist die einfache Debugging-Möglichkeit gegenüber der originalen ZBE, die eine Implementierung und Validierung überhaupt erst effektiv ermöglicht. Weitere Vorteile dieser gewählten Lösung zeigen sich bei der Validierung in Kapitel 6.

Nachteile der VAP sind unter anderem die deutlich größere Baugröße gegenüber der ZBE-HW sowie das fehlende HMI für die Eingabe welches im Fall der VAP nur mittels einer I/O-Karte verfügbar ist. D.h., dass ein weiterer Hardware-Baustein in Form einer elektronischen Platine für die Realisierung der Kundeneingabe umzusetzen ist. Hierfür musste für die HW der VAP Steckkarte eine entsprechende Schaltung entwickelt werden. Diese Lösung ist mit einer Physical-Computing Plattform – Arduino realisiert, in Kapitel 5 erläutert, und kann aufgrund seiner Umsetzung manuell und automatisiert bedient werden.

Anmerkung: Eine Implementierung der Methode in die originale ZBE-HW wurde versucht, scheiterte jedoch an der Konfiguration und Implementierung der AUTOSAR-BSW. Der Hersteller Preh bezieht die BSW von der Firma Elektrobit, die nur eine unkonfigurierte AUTOSAR-BSW zur Verfügung stellte, ohne Toolunterstützung. Dies machte es nahezu unmöglich ohne Support durch Elektrobit, der nicht zu Verfügung stand, eine lauffähige BSW zu kompilieren.

4.4. Implementierung

Der Begriff Implementierung beschreibt die Umsetzung des Softwareentwurfs bzw. die festgelegten Strukturen und Prozessabläufe aus Kapitel 3 in einem System. Das folgende Kapitel beschreibt die spezifische Implementierung in das, in Kapitel 4.3.1 ausgewählte, generische Steuergerät mit dem Ziel eine ZBE nachzubilden und ein simModul zu integrieren.

Abbildung 4.2 zeigt die für den Kunden sichtbaren HMI-Elemente ZBE (2) und CID (1). Dabei ist die ZBE eine reine Systeminput-Komponente, das CID nur ein Display, über das die Funktionen und die Funktionssteuerung visualisiert werden. Verbunden sind diese beiden Komponenten über die HU, die die Funktionslogik für die Anzeige sowie Teilfunktionen der Systemsteuerung integriert hat, d.h. die HU beinhaltet eine Menüführung, durch die der Kunde mit Hilfe der ZBE navigieren und folglich Funktionen steuern kann. Dadurch können mit Hilfe der ZBE domänenübergreifend Funktionen im Fahrzeug gesteuert werden (Details siehe [BMW15a], verwendete Funktionsbeispiele im Kapitel 6).

Die physikalische Systemarchitektur mit ihren Bestandteilen kann der Abbildung 6.6 in Kapitel 6.2.3 entnommen werden.

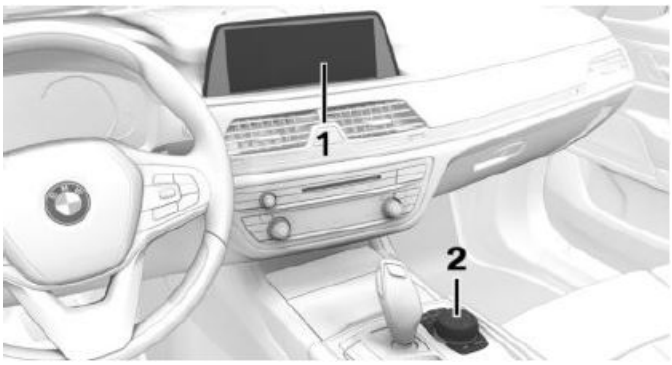


Abb. 4.2.: HMI Elemente - (1) Central Information Display, (2) Zentrale Bedieneinheit [BMW15a]

Das Stellglied der ZBE hat eine Dreh-, Drück- und Kippfunktion, sowie sieben Funktionstasten mit einer Drückfunktion implementiert. Dreh- und Kippbewegung sind über Lichtschrankensensoren, die Drückbewegungen über Taster realisiert und stellen die primäre Eingabeschnittstelle dar. Die Systemreaktion kann über die Schnittstelle des CID über den visuellen Kanal wahrgenommen werden, zudem besteht die Möglichkeit die Systemreaktion per Diagnose aus der HU auszulesen.

4.4.1. Deployment simModul

Das Deployment beschreibt das Mapping der in der VFB definierten SWCs aus Kapitel 3.3 auf die verfügbare Hardware-Topologie. Berücksichtigt werden hierbei die Konfigurationen aus der System-Description sowie Signaldefinitionen. Die Abbildung 4.3 zeigt die für das Mapping relevanten Inputs für das System Mapping, aus dem dann die entsprechenden ECU-Descriptions ausgeleitet werden können.

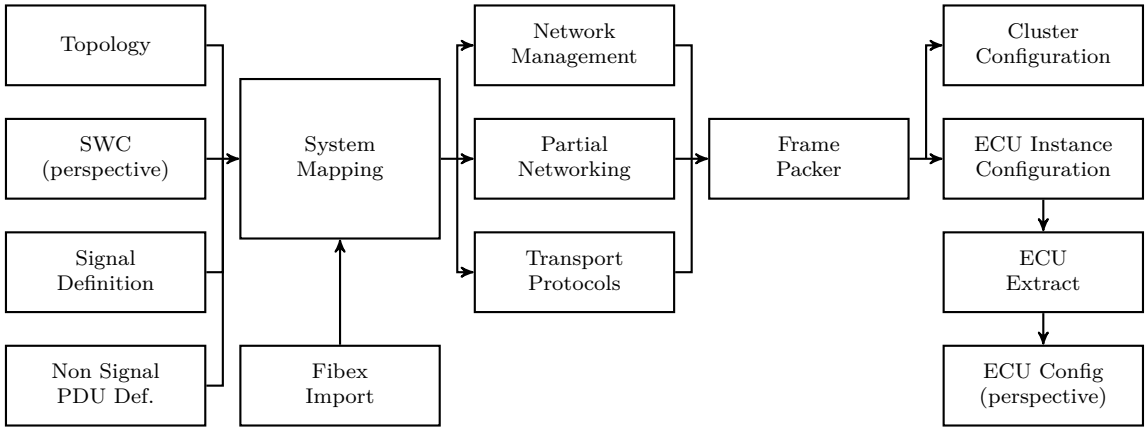


Abb. 4.3.: Prozess Generierung ECU-Configuration [Men14]

Die Topologie, die SWC sowie die Signaldefinition sind der Input für das Deployment, bei dem im Rahmen des System Mapping die Konfiguration bzw. Regeln für die Verteilung von SWCs auf die ECUs berücksichtigt werden. Mit der Hilfe dieser Informationen wird dann je ECU Kommunikationsschicht, Netzwerkmanagement und Protokolle speziell für die ECU mit ihren implementierten SWCs generiert. Ergebnis ist letztendlich die ECU-Config bzw. ECU-Description.

Die ECU-Description `ecu_extract.arxml` ist die Basis für den Software Build der entsprechenden ECU. Beim Deployment entstehen zudem die Beschreibungsdateien für die Module der AUTOSAR-BSW wie z.B. die `EcucModConfig_Dio.arxml`-Datei, die unter anderem das Port-Mapping bzw. Channel-Mapping enthalten. Die Toolumgebung erzeugt alle notwendigen Beschreibungsdateien, die auch die Partitionierung berücksichtigen, so dass der Compiler im Anschluss eine für die ECU spezifische Software inkl. spezifischer BSW und RTE baut.

Die beiden generierten Files `ecu_extract.arxml` und `EcucModConfig_Dio.arxml` ermöglichen im Bereich der Anwendung die automatische Erzeugung des Channel-Mapping für die Testfälle. (siehe Kapitel 5)

4.4.2. Architektur der Implementierung

Die Abbildung 4.4 zeigt die implementierten Elemente eines simModuls aus Kapitel 3.3.3 in die AUTOSAR-Schichtenarchitektur, die sich nach dem Deployment ergibt. Die einzelnen Module aus Liste 4.2 sind entsprechend der AUTOSAR-Spezifikation [AUTa] in den vorgegebenen Schichten der BSW implementiert. Die Kommunikation und Schnittstellen der implementierten Module untereinander sind in dem AUTOSAR Layered Software Architecture [AUTa] spezifiziert und entsprechend den Vorgaben implementiert.

Die grundlegenden Kommunikationspfade der Bausteine des simModuls sind auf der Architekturdarstellung auf Abbildung 4.4 dargestellt. Durchgezogene Linien zeigen den Signalverlauf durch die AUTOSAR-Schichten, gestrichelte Linien zeigen simModul-spezifische Kommunikationspfade. Die Architektur zeigt auch den Off-Board Anteil simMaster, der seinerseits über die normalen bereits vorhandenen Kommunikationspfade mit dem simAgent kommuniziert.

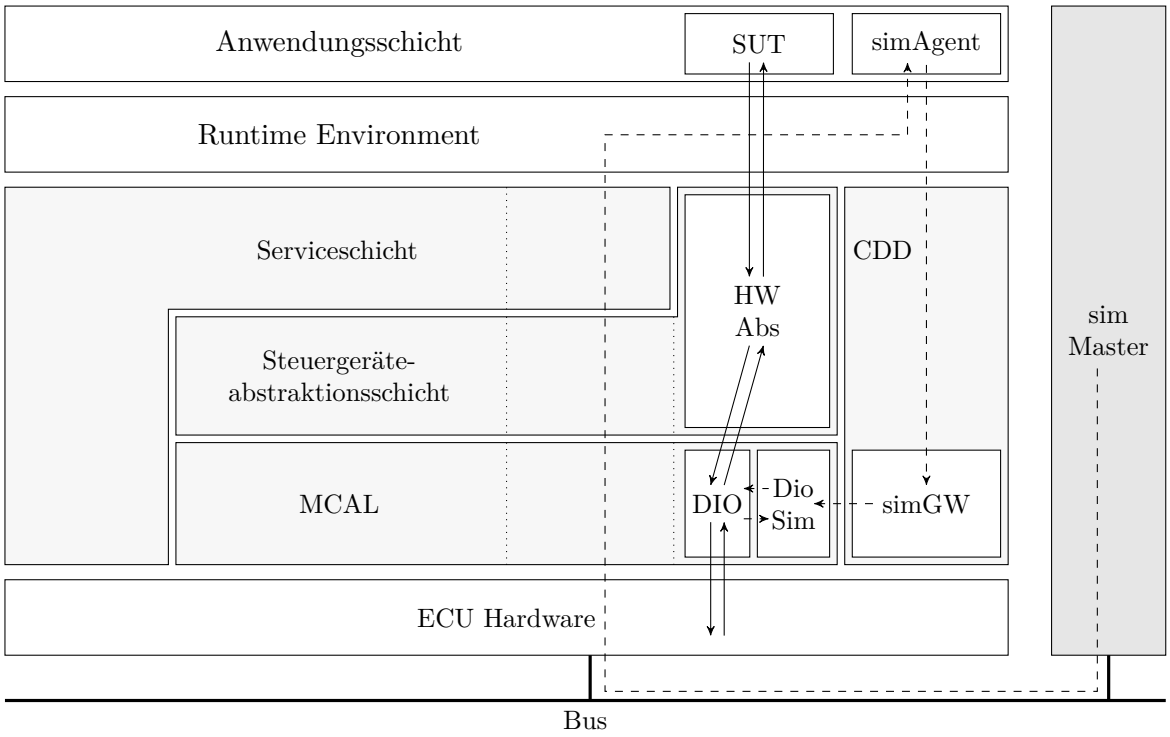


Abb. 4.4.: Implementierte Architektur

Anmerkung: Eine Zuordnung der Module aus Tabelle 4.2 in die Schichten kann mit Hilfe der Abbildung 2.8 oder der AUTOSAR-Spezifikation [AUTa] vorgenommen werden. Abbildung 4.4 zeigt diese aufgrund dem Fokus auf die Bestandteile des simModuls nicht.

Name	Typ	Beschreibung
BswM	System Service	Basis Software Mode Manager
CanIf	Abstraction	Can Interface Abstraktion
CanSM	Com Service	Can State Manager
Com	Com Service	Communication
ComM	System Service	Communication Manager
Dcm	Com Service	Diagnostic Communication Manager
Dem	System Service	Diagnostic Event Manager
Det	System Service	Default Error Tracer
EcuM	System Service	ECU State Manager
Fee_MD	Abstraction	Flash EEprom Emulation
Fls_MD	FlashDriver	Internal/External Flash Driver
HwIoAb_DIO	EcuAbstraction	HW-Abstraktion
NvM	Memory Service	NVRAM Manager
SwcZBE	Application	Application Software Component
simAgent	Application	Application Software Component
simGW	ComplexDeviceDriver	Complex Device Driver

Tab. 4.2.: Implementierte Module

4.4.3. Basissoftware der ZBE

Die AUTOSAR-BSW der nachgebauten ZBE (VAP) beinhaltet in erster Linie die in Tabelle 4.2 aufgelisteten Module, die nicht vom Typ *Application* sind. Hinzu kommen die Treiber für die Interaktion mit der VAP-Hardware sowie die ECU Abstraktion. [AUTc] Implementiert wurden der Treiber für Digital Input/Output (DIO)- und ein CAN-Treiber, entsprechend der originalen ZBE-HW. Durch die Integration des simModuls ist noch ein Complex Device Driver (simGW) erforderlich, um den Zugriff auf den DIO-Treiber bzw. DioSim zu ermöglichen (Tabelle 4.2).

Der DIO hat nach AUTOSAR Standard [AUTg] für den Service `Dio_ReadChannel` folgende Spezifikation. (Funktionsaufruf siehe Auflistung 4.1)

Service Name	Dio_ReadChannel	

Syntax	`Dio_LevelType Dio_ReadChannel(`
	`  Dio_ChannelType ChannelId`
	`)`	

Parameter (in)	ChannelId	ID of DIO Channel
Return Value	Dio_LevelType	STD_HIGH STD_LOW
Beschreibung	Gibt den Wert des spezifizierten Channels zurück.	

Tab. 4.3.: Spezifikation `Dio_ReadChannel`

Auflistung 4.1: Auschnitt `Dio.c - Read ChannelID`

```
233 FUNC(Dio_LevelType, DIO_CODE) Dio_ReadChannel(Dio_ChannelType ChannelId)
```

Im Folgenden wird die entscheidende Änderung für die Realisierung der software-basierten Simulation von Signalen erläutert.

Die Simulation von Eingangssignalen wird durch die Implementierung des `DioSim_GetSim`-Service (Tabelle 4.4) nach dem Vorbild des in AUTOSAR spezifizierten `Dio_ReadChannel`-Service (Tabelle 4.3) ermöglicht. Für die Realisierung einer Rückfallebene werden zusätzliche Informationen benötigt, so dass im Falle des Ausfalls der Simulation auf den realen HW zurückgegriffen wird. Aus diesem Grund wurde der Service mit einem Pointer realisiert, dadurch wird in erster Linie das Vorhandensein eines Signalwerts geprüft.

Service Name	DioSim_GetSim	

Syntax	`DioSim_ReturnType Dio_GetSim(`
	`  Dio_ChannelType ChannelId, Dio_LevelType *Value`
	`)`	

Parameter (in) Parameter (out)	ChannelId *Value	ID of DIO Channel Pointer auf Adresse des simWerts
Return Value	DioSim_ReturnType	DIOSIM_SUCC_CHANNEL_FOUND DIOSIM_ERR_CHANNEL_NOT_FOUND
Beschreibung	Gibt zurück ob an der Adresse ein Wert für den Channel gefunden wurde.	

Tab. 4.4.: Spezifikation `DioSim_GetSim`

Durch die Umbenennung des Aufrufs `Dio_ReadChannel()` in `DioOrigin_ReadChannel()` mittels eines Makros, siehe Auflistung 4.2 `#define` wird im Treiber der Aufruf der Services vertauscht. Dies hat den Vorteil, dass angrenzende bzw. verbundene Module in der BSW nicht angepasst werden müssen.

Auflistung 4.2: Auschnitt `Dio.c - Read Sim ChannelID`

```
221 #define Dio_ReadChannel DioOrigin_ReadChannel
```

Bei Aufruf der Funktion `Dio_ReadChannel()` wird eine `if`-Abfrage (Auflistung 4.3 Zeile 610) durchgeführt. Meldet die Funktion `DioSim_GetSimState()` eine aktive Simulation wird über den in Tabelle 4.4 spezifizierten Service der Wert aus dem simGW ausgelesen. In nächster Instanz wird durch eine weitere `if`-Abfrage (Auflistung 4.3 Zeile 613) das Vorhandensein eines Simulationswerts für den geforderten Channel geprüft. Scheitert die Abfrage des simulierten Werts oder ist die Simulation nicht aktiv wird immer der Service `DioOrigin_ReadChannel()` ausgeführt. Hierdurch wird die Rückfallebene bei fehlenden Daten sichergestellt.

Im normalen Betrieb der Software ohne aktiver Simulation ist der Eingriff durch eine Codezeile der `if`-Abfrage minimal.

Auflistung 4.3: Auschnitt `Dio.c` - `DioSim_GetSimState`

```
609 FUNC(Dio_LevelType, DIO_CODE) Dio_ReadChannel(Dio_ChannelType ChannelId){
610   if (DIOSIM_SIMULATION_ENABLED == DioSim_GetSimState()){
611     Dio_LevelType ReturnValue;
612     DioSim_ReturnType ReturnCode = DioSim_GetSim(ChannelId, &ReturnValue);
613     if (DIOSIM_SUCC_CHANNEL_FOUND == ReturnCode){
614       return ReturnValue;
615     }
616   }else if (DIOSIM_SIMULATION_LEARNING == DioSim_GetSimState()){
617     /* TODO SEND THE VALUE TO THE DIAG OR ANYWHERE VIA CAN OUT */
618   }
619   /** Cases:
620    * - Simulation Disabled
621    * - No SimulationValue for ChannelId
622    * - Simulation Learning sent Message
623    */
624   return DioOrigin_ReadChannel(ChannelId);
625 }
```

Die ECU-Hardware Abstraktions Schicht IoHwAbs_DIO (Anhang C.1.3) wandelt die spezifischen Channel einer ECU auf abstrakte Pointer. Durch diese Abstraktion können in den höheren Schichten der AUTOSAR-BSW abstrakte Namen verwendet werden, die die Programmierung und Lesbarkeit des Codes vereinfachen. Entscheidend ist, dass die Channeladressen in der `Dio_Cfg.h` deklariert sind (Ausschnitt Auflistung 4.4). D.h., eine HW-Änderung oder eine Integration auf eine andere HW kann durch eine einfache Anpassung der Channeladressen erfolgen.

Auflistung 4.4: Auschnitt `Dio_Cfg.h` - Channel Adressen

```
59 #define DioChannel_D0_BtnMain (0x300)
```

Diese Auflistung 4.4 der Channeldefinition aus der `Dio_Cfg.h` ist ein entscheidendes Element der Automatisierung und findet sich wieder in Kapitel 5. Die Konfigurations-Files `*_Cfg.h` sind aus der AUTOSAR-BSW Toolumgebung generiert und entsprechen der ECU-Description. Diese Daten werden im Rahmen der Anwendung (Kapitel 5) wiederverwendet, da sie das Channelmapping für die Off-Board Automatisierung enthalten.

Für die Drehbewegung ist die Abstraktion in der `IoHwAbs_DIO.c` aufwändiger, da in der Abstraktionsschicht die Signale der Gabellichtschranke (Funktionsprinzip [Köb15b]) interpretiert werden müssen. In der Abstraktion wurde folglich eine Drehrichtungserkennung sowie eine Schrittzählung, entsprechend [Köb15b], implementiert. An die RTE wird der s.g. `RotationCount`, der bei einer Drehrichtung rechts um die Anzahl der gedrehten Schritte inkrementiert oder bei einer Drehbewegung links um die Anzahl der Schritte dekrementiert wird, weitergegeben.

4.4.4. RTE der Komponente ZBE

Die RTE stellt die Kommunikationsschicht dar, aus der sich die SWCs bedienen und die Ausgangssignale an die BSW zurückgeben. Die RTE mit der `RTE.c` sowie alle `RTE_*.h`-Files werden vollständig entsprechend der Konfiguration in der Toolumgebung, generiert. Die `RTE.c` clustert alle Header-Files der angrenzenden Schichten bzw. Komponenten entsprechend der AUTOSAR-Spezifikation [AUTi], [AUTf].

4.4.5. Applikations SWC - ZBE

Die Applikation-SWC der ZBE ist weitestgehend aus dem Original übernommen und auf das generische Steuergerät übertragen. Diese SWC erfüllt die normalen Kundenfunktionen, die auch in der Serienkomponente verfügbar sind. Hierbei gibt es folgende funktionale Anforderungen:

Eine Systemeingabe kann über Dreh-, Drück- und Kippbewegung über das Stellglied sowie die Funktionstasten erfolgen. Bei jeder gültigen Systemeingabe wird, event-gesteuert beim nächsten Taskzyklus der Applikation-SWC, eine CAN-Botschaft mit der entsprechenden Signal-Änderung versendet.

Zu beachten ist, dass bis auf die Drehbewegung diverse Zeitkriterien die Interpretation der Eingabe bestimmen. Die Drück- und Kippbewegung ist für andere Komponenten bzw. die Empfänger der Nachricht nur gültig für *Short* und *Long*. *Too Long* und *Invalid* werden von der Komponente ZBE zwar an die Empfänger versendet, führen aber zu einem Rücksetzen der Funktionen im verteilten System. Implementiert wurde dieses Zeitverhalten in der Header-Datei `SwcZBE.h`, welche über einen `#include` in die Application-SWC eingebunden wird. Die Tabelle 4.5 listet die möglichen bzw. die verwendeten Intervalle für die Interpretation der Kundeneingabe durch die SWC-ZBE auf. Die Auflistung 4.5 zeigt die Definition der Intervalle in der `SwcZBE.h`.

Code	Intervall [ms]	Beschreibung
Short	$]0, 2000]$	Eingabe kurz
Long	$]2000, 40000]$	Eingabe lang
Too Long	$]40000, 60000]$	Eingabe zu lang
Invalid	$]60000, \infty[$	Eingabe ungültig

Tab. 4.5.: Timing-Verhalten der ZBE

Die Rotationseingabe am Drehgeber kann über die mechanische Realisierung immer nur zu einem vollständigen Zustandswechsel in der Gabellichtschranke führen. Zwischenzustände sind nicht möglich, der Drehgeber springt aufgrund seiner Hardware-Umsetzung immer in eine Raste vor oder zurück.

Auflistung 4.5: Auschnitt `SwcZBE.h`

```
8  #ifndef INC_SWCZBE_H_
9  #define INC_SWCZBE_H_
10
11 #include "Rte_SwcZBE.h"
12
13 #define SQN_OVERFLOW_THRESHOLD 254u
14
15 #define ROTATION_MIDDLE 32767u
16
17 #define BTN_PUSH_NO 0u
18 #define BTN_PUSH_SHORT 1u
19 #define BTN_PUSH_LONG 2u
20 #define BTN_PUSH_TOO_LONG 3u
21 #define BTN_PUSH_INVALID 15u
22 #define BTN_PUSH_THRESHOLD_LONG 2000 /* in ms = 2secs */
23 #define BTN_PUSH_THRESHOLD_TOO_LONG 40000 /*in ms = 40secs */
24 #define BTN_PUSH_THRESHOLD_INVALID 60000 /*in ms = 60secs */
```

Die SWC ZBE liest die einzelnen Sensorwerte zyklisch, in einem Taskcykle = 60 ms, aus der RTE und triggert bei jeder Sensorwertänderung eine CAN-Botschaft mit dem Status aller interpretierten Systemeingaben. Die Abfrage der Ports für die HW-Eingänge ist als Client/Server Schnittstelle (1:n) umgesetzt. Hier fungiert die SWC ZBE als Client (Requested Port Type) und erwartet die Bereitstellung der Signalwerte aus der RTE. Vorzugsweise wird eine Client/Server Schnittstelle für die Anbindung von SWCs an die RTE verwendet, da mehrere SWCs auf die Daten der RTE zugreifen können sollen [AUTd].

Auflistung 4.6: Auschnitt `SwcZBE.c` - Read RTE

```
52   // read values from HwIoAb_DIO
53   Rte_Call_CSIfZBE_ReadZBESensors_GetSensorData(
54     &(actValues.RotationCount),
55     &(actValues.DeflectionWest),
56     &(actValues.DeflectionEast),
57     &(actValues.DeflectionNorth),
58     &(actValues.DeflectionSouth),
59     &(actValues.BtnMain),
60     &(actValues.BtnMenu),
61     &(actValues.BtnBack),
62     &(actValues.BtnOption),
63     &(actValues.BtnMedia),
64     &(actValues.BtnMap),
65     &(actValues.BtnCom),
66     &(actValues.BtnNav),
67     &(actValues.RotationOverflow)
```

Die Bereitstellung (`Rte_Write_IfTxST_ZBE_FRT_TxST_ZBE_FRTData(&canData)`) der Statuswerte für das Versenden der CAN-Botschaft ist als eine Sender Receiver Schnittstelle (1:1) umgesetzt. Diese 1:1 Anbindung stellt sicher, dass nur eine bestimmte SWC eine Botschaft versenden kann.

Die Implementierung der CAN-Anbindung wurde entsprechend der Anforderungen aus dem BMW-Group-Lastenheft [BMWc] konfiguriert und generiert. Die Nachrichten gelangen nach AUTOSAR-Spezifikation durch die BSW-Schichten Communication Services, Communication HW-Abstraction und Communication Drivers auf den BUS. [AUTi]

Auflistung 4.7: Auschnitt `SwcZBE.c` - Write RTE

```
96   if (canData.SQN_NR != actValues.sqn){
97     // we have a change!
98     // correct the sqn-nr to the actual increment
99     if (actValues.sqn + 1 > SQN_OVERFLOW_THRESHOLD){
100      actValues.sqn = 0u;
101    }
102    canData.SQN_NR = actValues.sqn + 1u;
103    // send it via CAN
104    Rte_Write_IfTxST_ZBE_FRT_TxST_ZBE_FRTData(&canData);
105    // update the sqn
106    actValues.sqn = canData.SQN_NR;
107  }
```

4.4.6. Treibererweiterung - DioSim

DioSim ist die Erweiterung des Dio-Treiber und stellt die Schnittstelle zum Complex Device Treiber. `DioSim_GetSim()` setzt einen Pointer `*Value` auf den gefundenen Wert und gibt den Status `DIOSIM_SUCC_CHANNEL_FOUND` zurück.

Auflistung 4.8: Auschnitt `DioSim.c` - Implementierung `DioSim_GetSim`

```
76 FUNC(DioSim_ReturnType, DIO_CODE) DioSim_GetSim(Dio_ChannelType ChannelId,
       Dio_LevelType *Value){
77   int ArrayIndex;
78   DioSim_ReturnType ReturnCode = DioSim_GetIndex(ChannelId, &ArrayIndex);
79   if (DIOSIM_SUCC_CHANNEL_FOUND == ReturnCode){
80     *Value = DioSim_Values[ArrayIndex].Value;
81   }
82   return ReturnCode;
83 }
```

Das Simulationsdaten-Array im `DioSim` wird bei einer aktiven Simulation mittels dem Aufruf `DioSim_UpdateSim(Dio_ChannelType ChannelId, Dio_LevelType Value)`, durch den simAgent, zyklisch über das SimGW aktualisiert. Im Fall der synchronisierten Simulation, dargestellt in Kapitel 4.4.10, wird die Aktualisierung des Arrays durch den simAgent verzögert bis zum nächsten Taskzyklus der Applikation-SWC.

4.4.7. Complex Device Treiber - SimGW

Der Complex Device Treiber beinhaltet das simGW (Simulations Gateway), um den Zugriff auf den Treiber zu ermöglichen. Wie bereits in Kapitel 2.4.3 beschrieben, erlaubt AUTOSAR den Zugriff aus dieser Ebene. Das simGW ist hier, wie es der Name sagt, nur ein Gateway und routet die Daten vom simAgent mittels `DioSim_UpdateSim(ChanId, Value)` an die Treiberschnittstelle DioSim weiter (4.9).

Auflistung 4.9: Auschnitt `SimGW.c` - Implementierung `DioSim_UpdateSim`

```
26   PRP_INFO(RED("SimGw_AddSimChan - called with %u and %u\n"), ChanId, Value);
27   DioSim_UpdateSim(ChanId, Value);
28 }
```

4.4.8. Applikation SWC - simAgent

Die Abbildung 4.5 zeigt das implementierte Zustandsmanagement der Application SWC simAgent, in der die aus Kapitel 3.3.4 beschriebenen Zustände implementiert bzw. repräsentiert sind.

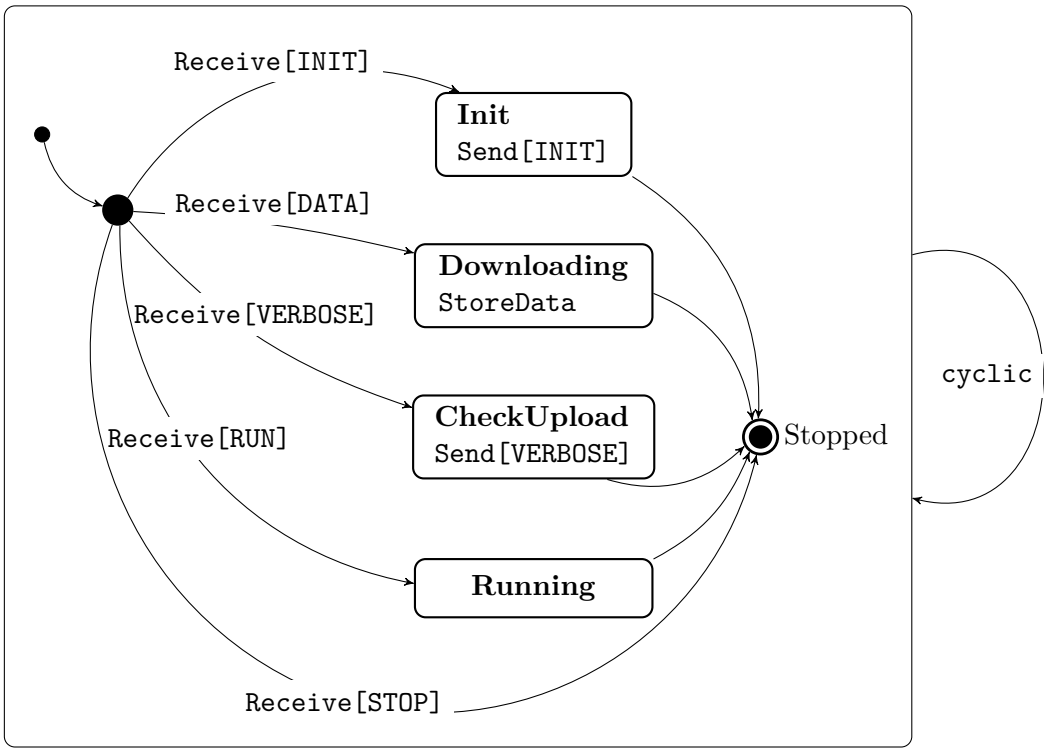


Abb. 4.5.: Implementiertes Zustandsmanagement – simAgent

Die Programmiersprache C bietet keine Möglichkeit Statemachines native zu definieren. Der Zustandsautomat auf Abbildung 4.5 zeigt das Verhalten der mit `if`-Anweisungen (Anhang C.2) umgesetzten Logik in der Application SWC simAgent. Mit der implementierten Logik kann die ursprüngliche Logik aus Kapitel 3.3.4 auf Abbildung 3.7 dargestellt werden. Unterschied ist, dass die „Zustände“ in globalen Variablen gespeichert werden und die implementierten `if`-Abfragen zyklisch durchlaufen werden. Ein weiterer Unterschied ist, dass Zustände übersprungen werden können und somit die Off-Board-Logik des simMaster aus Kapitel 4.6 die „Zustände“ des simAgent in der korrekten Reihenfolge bedienen muss, dies ist in Kapitel 4.4.9 erläutert.

Alternative Implementierungen wären über `switch - case` Anweisungen oder über eine Zustandstabelle möglich. Umsetzung siehe Anhang simAgent.c.

4.4.9. Off-Board Automatisierung - simMaster

Abbildung 4.6 zeigt das zum simAgent komplementäre Zustandsmanagement des simMasters bzw. der Off-Board Testautomatisierung. Dieses Zustandsmanagement ist vollständig in den Testfällen implementiert. D.h., die Testfälle sind in die jeweiligen Zustände (Kap. 3.1.5) *Init*, *Charging*, *CheckUpload* und *waitforFinish* strukturiert. Diese Implementierung des Zustandsmanagement in den Testfall schafft eine Entkoppelung vom Tooling. Dies hat zur Folge, dass die Anforderungen an das Tooling stark reduziert werden.

Die Umsetzung der Zustände im Testfall (im Tool ECU-Test 6.4) stellt sich in verschiedenen Modulen bzw. Services dar, die in der korrekten Ausführung das Zustandsmanagement des simMasters wiedergeben und mit dem des simAgent interagieren.

InitializeECU entspricht dem Zustand *Init* aus Abbildung 4.6. Der simMaster versendet die Botschaft(en) `SIM_INIT_MASTER` und verlässt den *Init* Zustand bei Empfang der (aller) Botschaft(en) `SIM_INIT_AGENT` oder nach Timeout.

Service Name	InitializeECU	
Parameter (in)	EcuId	ID der ECU bzw. des simAgent
Konstante(n)	Discover	`TRUE` = Antwort des simAgents
	Initialisation	`TRUE` = Init-Befehl des simMasters
Return Value	EcuId	ID der ECU bzw. des simAgent
Beschreibung	Triggert die Initialisierung des simAgents mit der ID# der mit der eigenen ID quittiert.	

Eine Sonderform der Initialisierung ist die mit der ECU-ID `0x255` und dient dem Rücksetzten aller im verteilten System befindlichen simAgents. (siehe auch Zustandsmanagement des simAgent aus Kapitel 3.3.4 auf Abbildung 3.7)

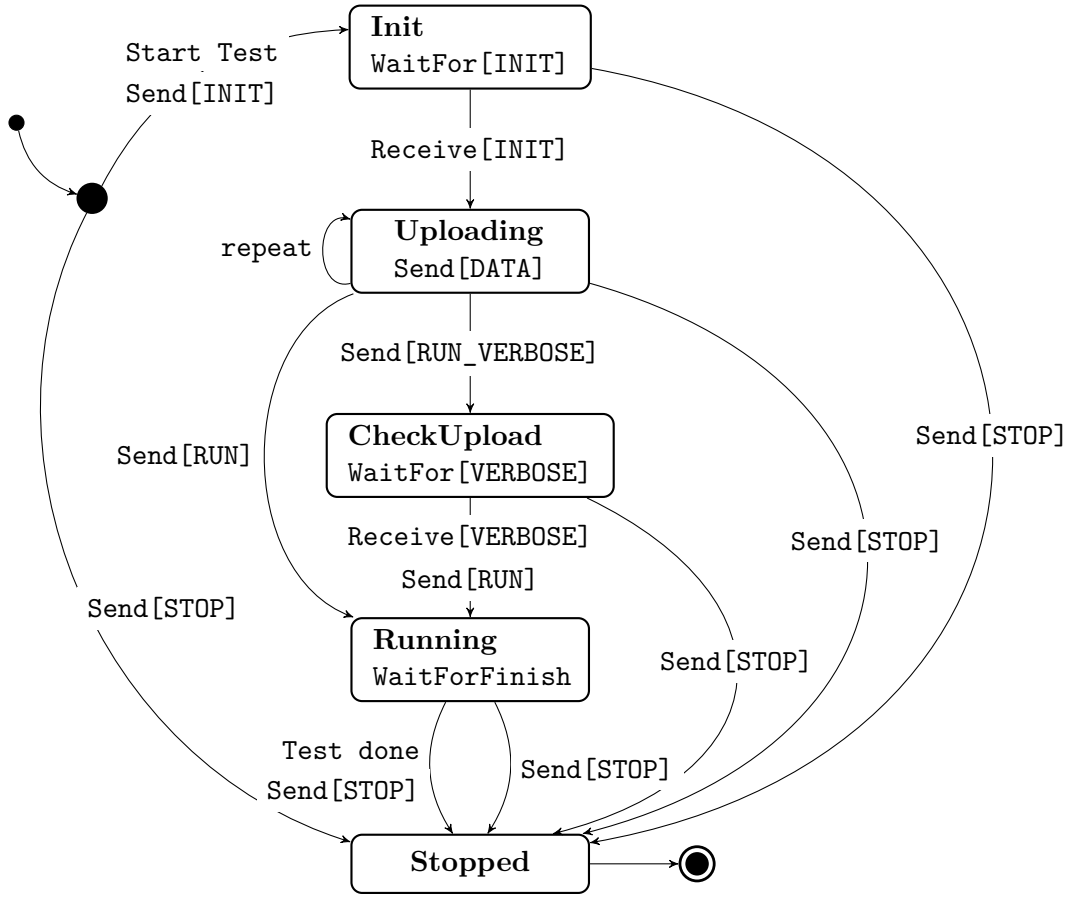


Abb. 4.6.: Implementiertes Zustandsmanagement – simMaster

SendTestData entspricht dem Zustand *Upload* und hat zu der in Kap. 3.3.5 deklarierten Nachrichtenspezifikation äquivalente Inhalte. Dieser Zustand kann nur nach erfolgreichem Verlassen des Zustands *Init* eintreten. In diesem Zustand versendet der simMaster alle Testschritte an die simAgents mittels `SIM_DATA`. Testsequenzen werden, wie in Kapitel 5.3 beschrieben in Signaltupeln abgelegt, daher wird für jeden Schritt eine Botschaft versendet. Der Zustand *Upload* wird nach dem erfolgreichen Upload verlassen.

Service Name	SendTestData	
Parameter (in)	EcuId	ID der ECU bzw. des simAgent
	ChanId	ChannelID des zu stimulierenden Kanals
	Value	Wert bzw. Pegel für den zu stim. Kanal
	Delay	Zeit in ms bis das der Wert/Pegel anliegt
Return Value	none	
Beschreibung	Versendet Datenpakete an den durch die ECU-ID adressierten simAgent.	

Anmerkung: Ein Clustern von zwei SendTestData-Services verbessert die Übersichtlichkeit und die Lesbarkeit der Testfälle. So ist aus Sicht des Use-Cases z.B. eine Betätigung eines Tasters ein Signalwechsel LOW - HIGH - LOW. Werden nun zwei SendTestData-Services geclustert, kann ein solches Verhalten innerhalb eines Packages bzw. Services dargestellt werden.

StartTest entspricht dem Zustand *StartTest* aus Kap. 3.3.5. Hierbei wird über einen Broadcast-Nachricht (`SIM_RUN`) ein Signal an alle simAgents, für den Start der Simulation versendet.

Service Name	StartTest	
Parameter (in)	StopOrGo	Stop/Go/GoSync
	Offset	Übergreifender Offset für den Start
	VerboseID	siehe CheckUpload
	Verbose	siehe CheckUpload
Return Value	none	
Beschreibung	Starten der Simulation durch die Broadcast Nachricht `SIM_RUN` oder CheckUpload.	

Anmerkung: Das Verbose-Flag muss auf `FALSE` stehen, dass ein Simulationsstart erfolgt.

CheckUpload ist ein optionaler Zustand, der ein Prüfen der Vollständigkeit der Daten in den einzelnen ECUs bzw. simAgents ermöglicht. Dieser Zustand ist als paralleler Zustand des Zustands *StartTest* implementiert und verhindert den Start der Simulation. Der Übergang in den Zustand wird getriggert durch die Nachricht `SIM_RUN` mit dem Verbose-Flag.

Service Name	CheckUpload	
Parameter (in)	ID der ECU bzw. des simAgent	
Konstante(n)	StopOrGo	0
	Verbose	`TRUE`
Return Value	ERROR	$\lvert\text{Teststeps}_{\text{simMaster}}\rvert \neq \lvert\text{Teststeps}_{\text{simAgent}}\rvert$
	SUCCESS	$\lvert\text{Teststeps}_{\text{simMaster}}\rvert \equiv \lvert\text{Teststeps}_{\text{simAgent}}\rvert$
Beschreibung	Bewertet die Anzahl der Testschritte die in einem ECU gespeichert wurden.	

EndTest entspricht dem Zustand *Stopped* und ist der Endzustand, in dem nochmal optional ein *Stop*-Befehl versendet werden kann. Von hier aus kann wiederum ein Test erneut ausgeführt werden oder der Prozess von vorne beginnen.

4.4.10. Scheduling und Tasks

Das Scheduling beschreibt den Ablaufplan bzw. die Reihenfolgeplanung der einzelnen Tasks im embedded System. Implementiert wurden zwei zyklische Tasks OsT_BSW und OsTask_Automap1. Die Reihenfolge der implementierten Runnable Entity (RE)s entspricht auch der Ausführungsreihenfolge.

OsT_BSW clustert alle REs der BSW-Schicht.

Name	Period [ms]	Komponente
OsT_BSW	5.0	
▷ Evt_BswM_MainFunction	10.0	BswM
▷ Evt_ComM_MainFunction_0	20.0	ComM
▷ Evt_Com_MainFunctionRx	10.0	Com
▷ Evt_Com_MainFunctionTx	10.0	Com
▷ Evt_CanSM_MainFunction	10.0	CanSM
▷ Evt_CanIf_MainFunction	10.0	CanIf
▷ Evt_NvM_MainFunction	10.0	NvM
▷ Evt_Fee_MainFunction	5.0	Fee_MD
▷ Evt_Fls_MainFunction	5.0	Fls_MD
▷ Evt_Dcm_MainFunction	5.0	Dcm
▷ Evt_Dem_MainFunction	100.0	Dcm
▷ Evt_EcuM_MainFunction	10.0	Dcm

OsTask_Automap1 clustert alle SWCs der Application Schicht.

Name	Period [ms]	Komponente
OsTask_Automap1_BSW	10.0	
▷ TE_SimAgent_Starter_cyclic	60.0	SimAgent
▷ TE_SimAgent_UpdateCDD_cyclic	10.0	SimAgent
▷ TE_SimGW_cyclic	10.0	SimGw
▷ TE_runZBE_Wakeup_cyclic	10.0	HwIoAb_DIO
▷ TE_SwcZBE_cyclic	60.0	SwcZBE

Die BSW und simAgent laufen in einem 10 ms, die Application-SWC ZBE in einem 60 ms Task. Aus Scheduling Gründen wurde daher der TE_SimAgent_Starter_cyclic vor die anderen Tasks so platziert, dass TE_SimAgent_UpdateCDD_cyclic und TE_SimGW_cyclic vor der RE TE_SwcZBE_cyclic ausgeführt werden. Dadurch wird die Möglichkeit geschaffen, dass der simAgent die Ausführungszeitpunkte der SWC ZBE kennt. So kann sich der simAgent mit einem Taskzyklus $T = 10\,\text{ms}$ auf die ZBE mit einem Taskzyklus von $T = 60\,\text{ms}$ synchronisieren und die Signale zum exakt gewollten Zeitpunkt einsteuern.

Die Auswirkungen zwischen synchronisiert und nicht synchronisiert werden im Kapitel 6.2.3 anschaulich dargestellt.

Die Abbildung 4.7 zeigt die implementierte Reihenfolge der einzelnen REs die im OSTask_Automap1_BSW laufen.

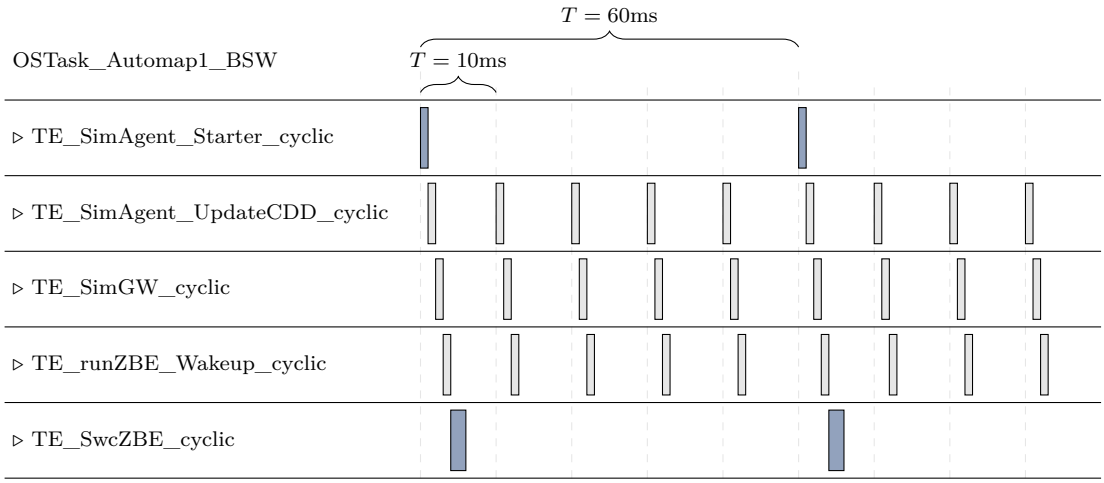


Abb. 4.7.: Exemplarische Darstellung - Implementiertes Scheduling

Die Ausführungszeiten der einzelnen REs TE_SimAgent_Starter_cyclic, TE_SimAgent_UpdateCDD_cyclic, TE_SimGW_cyclic und TE_runZBE_Wakeup_cyclic müssen in der Summe kürzer sein als die Ausführungszeit der RE TE_SwcZBE_cyclic, so dass diese im Taskzyklus OSTask_Automap1_BSW noch ausreichend Zeit vor dem nächsten Aufruf hat. Das Kapitel 6.2.2 zeigt hier im Rahmen der Worst Case Execution Time (WCET) die zeitlichen Grenzen.

4.4.11. Kommunikation und Protokoll

Für die gesamte Kommunikation, zwischen simMaster und simAgent werden insgesamt fünf Botschaften verwendet. Die CAN-IDs sind frei gewählt und werden in den aktuellen BMW-Nachrichtenkatalogen zum aktuellen Zeitpunkt noch nicht verwendet. Dies ist für die spätere Validierung an Fallstudien im Gesamtsystem ausschlaggebend, um keine unerwünschten Nebeneffekte mit anderen Funktionen hervorzurufen. Die höchste Nachrichtenpriorität hat die `SIM_RUN`-Botschaft, da sie den kleinsten Identifier aufweist. Somit ist eine Priorisierung dieser Nachricht bei hohen Buslasten, da sie auch einen Testabbruch herbeiführen kann, gewährleistet.

Initialisierung

`SIM_INIT_MASTER` mit der CAN-ID 0x255 ist nach dem Kommunikationsprotokoll die erste Botschaft, die eine Abfrage der im Gesamtsystem implementierten sim-

Agents durchführt. Hierdurch können auch alte Testdaten im Speicher der jeweiligen ECU gelöscht und die simAgents zurückgesetzt werden.

`ECUID` Signal beinhaltet die Adresse des simAgents, der die Botschaft empfangen soll.

`Init` Signal gibt an, ob der simAgent in den Zustand *Init* wechseln soll.

`CMD` Signal gibt an, ob der simAgent eine Antwort/Bestätigung versenden soll.

Code	Name	Beschreibung
0	keine Aktion	simAgent führt keine Aktion aus
1	Discover	simAgent soll eigene ID versenden

`SIM_INIT_AGENT` mit der CAN-ID 0x258 ist die Antwort des simAgents und somit die Bestätigung der Betriebsbereitschaft bzw. Initialisierung abgeschlossen.

`ECUID` Signal beinhaltet die Adresse des simAgents als Bestätigung, dass der korrekte simAgent geantwortet hat.

Code	Name	Beschreibung
0...254	ECUID	Adresse des simAgents
255	ECUID	nicht vergeben

Datenaustausch

Der Datenaustausch zwischen simMaster und simAgent findet in einer Schleife statt. Hierbei wird eine Signalsequenz in ihre Datenpunkte separiert und in einzelnen Nachrichten (`SIM_DATA`) an den simAgent versendet.

`SIM_DATA` mit der CAN-ID 0x256 dient der Übertragung der Testschritte vom simMaster zu dem über die ECU-ID adressierten simAgent.

`ECUID` Signal beinhaltet die Adresse des simAgents, der die Botschaft empfangen soll.

Code	Name	Beschreibung
0...254	ECUID	Adresse des simAgents
255	ECUID	nicht vergeben

`ChanID` Signal beinhaltet die Adresse des simAgents, der die Botschaft empfangen soll.

Code	Name	Beschreibung
0...65535	ChanID	ID des Channels (HW)

`Value` Signal beinhaltet die Adresse des simAgents, der die Botschaft empfangen soll.

Code	Name	Beschreibung
0	Low	Logikpegel = 0
1	High	Logikpegel = 1
2	ungültig	außerhalb definierten Bereich

`StartOffset` Signal beinhaltet die Adresse des simAgents, der die Botschaft empfangen soll.

Code	Name	Beschreibung
$0...2^{32}$	Offset [ms]	Offset bis zu sim. Wert

Start/Stopp/Detail-Daten

`SIM_RUN` mit der CAN-ID 0x254 beinhaltet die Start bzw. Stopp Kommandos. Des Weiteren wird die Nachricht für eine Prüfung der Datenübertragung verwendet, indem das Kommando Verbose an den entsprechenden simAgent mit der ECU-ID, auch VID genannt, versendet wird.

`StopOrGo` Signal beinhaltet das Start bzw. Stopp Kommando an den simAgent.

Code	Name	Beschreibung
0	Beenden	Simulation beenden
1	Start	Simulation beginnen
3	Sync.-Start	Sync. Simulation beginnen

`VER` Signal beinhaltet das Verbose-Flag, dies löst eine Antwort des simAgents mit detailierten Daten aus, beinhaltet gespeicherte Testdaten.

Code	Name	Beschreibung
0	keine Aktion	
1	Verbose	Anforderung detaillierte Daten

`RunOffset` Signal beinhaltet den Offset für den Start der Simulation nach Erhalt des Go Signals. Wenn nicht benötigt, wird dieser auf null gesetzt.

Code	Name	Beschreibung
$0 \ldots 2^{32}$	Offset [ms]	Zeitlicher Beginn der Simulation

`VID` Signal beinhaltet die Verbose-ID, also die Adressierung, welcher simAgent detaillierte Daten über gespeicherte Testdaten versenden soll.

Code	Name	Beschreibung
0 . . . 255	ID	simAgent Adresse

`SIM_VERBOSE` mit der CAN-ID 0x259 ist eine detailreiche Antwortnachricht des simAgents, in der er z.B. die Anzahl der Testschritte zur Verifikation der Übertragung an den simMaster schicken kann.

`ECUID` Signal beinhaltet die Adresse des simAgents, der die Botschaft versendet.

Code	Name	Beschreibung
0 . . . 254	ECUID	Adresse des simAgents
255	ECUID	nicht verwendet

`TestLength` Signal beinhaltet Anzahl der Testschritte, die vom simAgent gespeichert wurden.

Code	Name	Beschreibung
0 . . . 65535	Anzahl	Anzahl Testschritte

`LastChanID` Signal beschreibt die letzte Channel-ID, in der ein Wert simuliert wurde. Dient der Verifikation einer korrekt abgeschlossenen Simulation.

Code	Name	Beschreibung
0 . . . 65535	ChanID	ID des Channels (HW)

`VER` Signal beinhaltet ein Flag, ob eine Simulation beendet wurde.

Code	Name	Beschreibung
0	Simulation nicht abgeschlossen	
1	Simulation abgeschlossen	

`LastSimVal` Signal beschreibt den letzten Logikpegel, der an der letzten Channel-ID anlag.

Code	Name	Beschreibung
0	Low	Logikpegel = 0
1	High	Logikpegel = 1
2	ungültig	außerhalb definierten Bereich

Um nach einer Simulation von HW-Signalen die simAgents zurückzusetzen, besteht die Möglichkeit über das Versenden einer Nachricht `SIM_INIT` alle Daten aus den simAgents zu löschen.

Beispiel: Der simMaster versendet einen Init-Befehl (0x01) an alle simAgents (ECU-ID 0x255).

Byte 0	1
0xFF	0x01

Dieses Vorgehen kann vor bzw. nach einer Simulation erfolgen, um fehlerhafte oder unvorhergesehen Simulationen von simAgents zu verhindern.

Codemultiplexverfahren

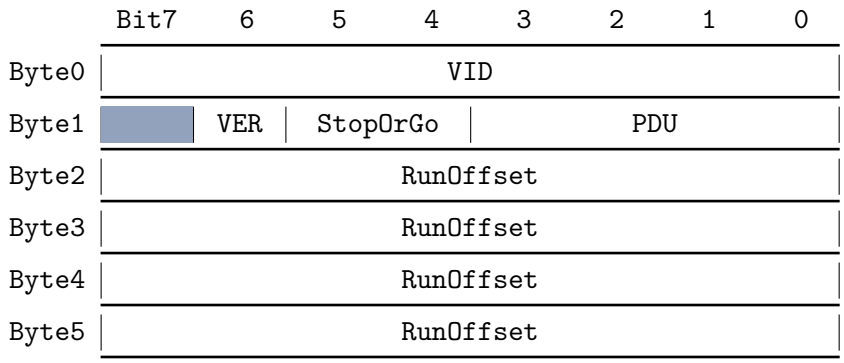
Multiplexing dient der Reduktion von CAN-Nachrichten durch die Verwendung eines weiteren Identifikators innerhalb der Nachricht. Für diesen zusätzlichen Identifikator würden in jeder der fünf Nachrichten vier Bit erforderlich sein, um eine Identifikation der hier genannten Protocol Data Unit (PDU)-ID zu ermöglichen. Durch den Einsatz von vier Bit wird ein Multiplexen einer Nachricht, also auf je CAN-ID auf $2^4 = 16$ Varianten, ermöglicht. Um eine Identifikation anhand der PDU zu erkennen, müssen die vier Bit in allen Tx-Frames sowie Rx-Frames an der jeweils identischen Stelle im CAN-Frame platziert sein. Multiplexing geht zu Lasten des Nachrichteninhalts, da in jeder Nachricht Bits für die zusätzliche Identifikation belegt werden.

Der Aufbau der CAN-Frames ändert sich wie folgt. In Byte 1 Bit 0 bis 3 ist die PDU bzw. Identifikation verankert, dadurch verschieben sich die Inhalte in den CAN-Nachrichten. Die Summe der Inhalte je CAN-Nachrichte verändert sich nicht, da in dem in der Methodik vorgestellten Protokoll ein Multiplexing von Beginn an vorgesehen wurde.

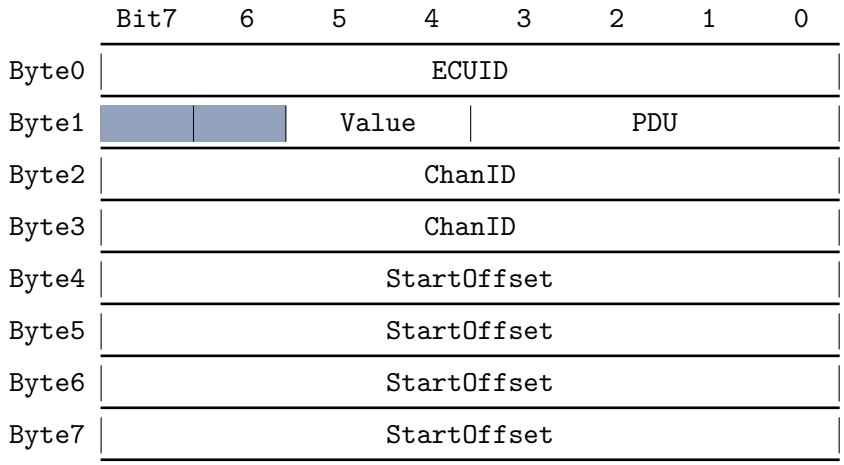
Rx-Frames Codierung der Frames in Byte1 Bit0...3 durch den s.g. PDU-Identifier.

`SIM_INIT_MASTER` wird durch die vier Bit PDU erweitert, hierdurch verschieben sich Init und CMD um vier Bit.

`SIM_RUN` wird in seinem Aufbau stark verändert. VID wird vom Ende der Nachricht auf den Anfang verschoben, wodurch der restliche Inhalt ein Byte plus die vier Bit PDU-ID verschoben werden.



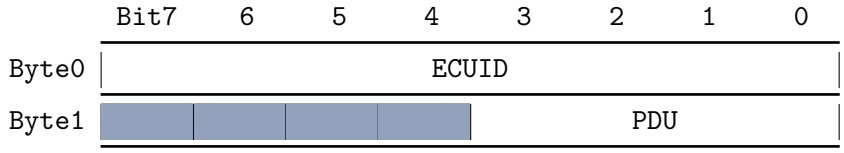
`SIM_DATA` wird in Byte 1…3 verändert. ChanID verschiebt sich ein Byte nach hinten und schafft dafür Platz für die PDU-Kennzeichnung.



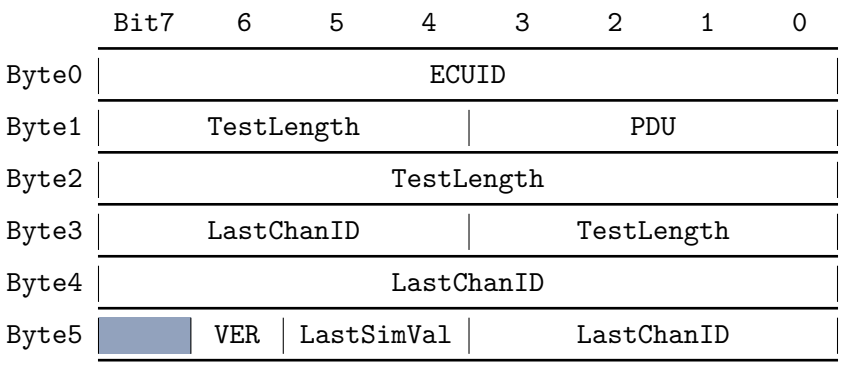
Tx-Frames Codierung der Frames ebenfalls in Byte1 Bit0…3.

Code	Nachricht
0	`SIM_INIT_AGENT`
1	`SIM_VERBOSE`

`SIM_INIT_AGENT` wird durch Multiplexing vier Bit länger. PDU ist der Identifier für die Frameaufteilung.



SIM_VERBOSE Im Frame verschieben sich Signale jedoch bleibt die Länge identisch.



Durch eine solche Multiplexing-Maßnahme reduziert sich die Anzahl der verwendeten CAN-IDs auf zwei, analog der in Kapitel 3.4.2 beschriebenen XCP on CAN Implementierung. Ein weiterer Grund für das Multiplexing ist die Möglichkeit der Erweiterung des Value-Frames zur Übertragung von z.B. analogen oder komplexeren Signalwerten zu den simAgents, analog der Methode aus Kapitel 3.4.3.

Eine beispielhafte Implementierung für die Übertragung komplexerer Simulationsdaten mittels eines Multiplexing für die vorgestellte Methode zeigt folgendes Beispiel:

Durch die PDU-ID-Codierung wird die Nachricht SIM_DATA in drei Varianten aufgeteilt. Variante eins, Singleframe (SF) wie bereits in Kapitel 3.4.3 spezifiziert. Variante zwei FF als First Frame, hierbei werden nur Channel-ID und ein optionaler Parameter übergeben. Dieser Parameter dient der Möglichkeit der Umrechnung (Skalierung, Verschiebung) der übertragenen Signalwerte im simAgent. Gefolgt von der dritten Variante der Nachricht, einem Consecutive Frame CF, welcher dann nur noch Signalwerte für den bereits in der FF-Nachricht versendeten Channel beinhaltet.

Eine Überprüfung der übertragenen Daten im Gesamten erfolgt analog wie bereits in Kapitel 3.3.5 beschrieben. Das Sequenzdiagramm 3.8 beschreibt das generische Protokoll, welches auch für diese Art von Multiplexing geeignet ist. Im Rahmen der *Datenübertragung* werden die Frames übertragen.

PDU-ID-Kodierung erweitert:

Code	Nachricht
0	SIM_INIT_MASTER
1	SIM_DATA (SF)
2	SIM_DATA (FF)
3	SIM_DATA (CF)
4	SIM_RUN

`SIM_DATA (FF)` mit der Kennzeichnung (0x02) in der PDU.

	Bit7	6	5	4	3	2	1	0
Byte0	ECUID							
Byte1	Parameter				PDU			
Byte2	ChanID							
Byte3	ChanID							
Byte4	StartOffset							
Byte5	StartOffset							
Byte6	StartOffset							
Byte7	StartOffset							

`SIM_DATA (CF)` mit der Kennzeichnung (0x03) in der PDU.

	Bit7	6	5	4	3	2	1	0
Byte0	ECUID							
Byte1	Value				PDU			
Byte2	Value							
Byte3	Value							
Byte4	StartOffset							
Byte5	StartOffset							
Byte6	StartOffset							
Byte7	StartOffset							

4.4.12. Routing Verfahren

Die vorgestellte Architektur und Kommunikation der verteilten Simulation zeichnet sich durch eine Master-Multi-Slave Architektur aus. In dieser Architektur, nur den simMaster und die simAgents betrachtet, werden für die Kommunikation zwei Routing-Verfahren verwendet:

Broadcast bezeichnet in Computernetzwerken eine Nachricht, die von einem Punkt an alle Teilnehmer des Netzwerks versendet wird.

`SIM_RUN` wird als Broadcast versendet, um einerseits alle simAgents synchron starten oder stoppen zu können.

`SIM_INIT_MASTER` mit der ECU-ID (0x255) wird vom simMaster als Broadcast versendet, um gezielt alle simAgents zurücksetzten zu können.

`SIM_INIT_AGENT` wird vom simAgent als Broadcast versendet, da die Agents die Adresse des Masters nicht kennen.

`SIM_VERBOSE` wird vom simAgent als Broadcast versendet, da die Agents die Adresse des Masters nicht kennen.

Unicast bezeichnet in Netzwerken die direkte Adressierung einer Nachricht an eine Komponente, hier Master (simMaster) an den Slave (simAgent).

`SIM_INIT_MASTER` mit einer ECU-ID ($\neq$ 0x255) wird als Unicast versendet, um gezielt einzelne simAgents zu erreichen.

`SIM_DATA` wird als Unicast versendet, um die Testdaten gezielt an einen simAgent zu versenden.

Routing on CAN – Auf der Abstraktionsebene CAN der Kommunikation werden alle Botschaften in Form eines Broadcasts versendet, da CAN in dieser Betrachtung in From eines gleichberechtigten Multi-Master-Systems agiert. Jeder CAN-Teilnehmer entscheidet für sich, ob er eine Nachricht verwendet.

4.4.13. Sicherheit - Risiken/Aspekte/Lösungen

Die Implementierung dieser Simulationsmethodik birgt gewisse Security-Risiken, da durch die einfache und Speicherplatz optimierte Lösung keine funktionelle Logik, außer der für die Simulation notwendigen Funktionen, integriert wurde. Aus der Anforderung heraus, dass die Software im Serieneinsatz äquivalent zu der getesteten Software sein soll, bleibt das simModul in dem Serien-Steuergerät implementiert und wäre somit theoretisch auch in der an den Kunden ausgelieferten Software verfügbar.

Um eine Simulation oder einen außerplanmäßigen Eingriff des simAgents z.B. im Kundenbetrieb zu verhindern, wurde eine Sicherheitsmaßnahme in die Implementierung integriert. Über eine zusätzliche Konfigurationsdatei `Sim_Cfg.h` wurde ein Precompiler-Makro implementiert, über das die Funktionalität des simAgents aktiviert bzw. deaktiviert werden kann. Der dafür implementierte Schalter ist in Auflistung 4.10 dargestellt.

Über `STD_ON` in der Header-File `Sim_Cfg.h` wird das simModul aktiviert, über `STD_OFF` deaktiviert. Dieses Makro wird durch den C-Präprozessor verarbeitet und ermöglicht eine sichere Deaktivierung des simAgents. Diese Konfiguration könnte auch in die toolgestützte Generierung der AUTOSAR-BSW-Dateien integriert werden und somit das Makro ablösen.

Auflistung 4.10: Include des on / off Makros

```
9 #define SIMULATION_MODULE_ACTIVE STD_OFF
```

Diese Art der Umsetzung bedeutet, dass die Software bei einer Änderung, also Aktivierung oder Deaktivierung des simModuls, aufgrund dieses Makros neu kompiliert werden muss. Diese Umsetzung schließt eine nachträgliche Aktivierung der simModul-Komponenten und somit eine Ausführung einer Simulation im Kundenbetrieb aus. Es besteht jedoch der Nachteil, dass am Seriensteuergerät mit der endgültigen Software keine Testfälle mehr simuliert werden können. Die auf Abbildung 4.8 dargestellte Include-Architektur zeigt, wie die `Sim_Cfg.h` eingebunden ist.

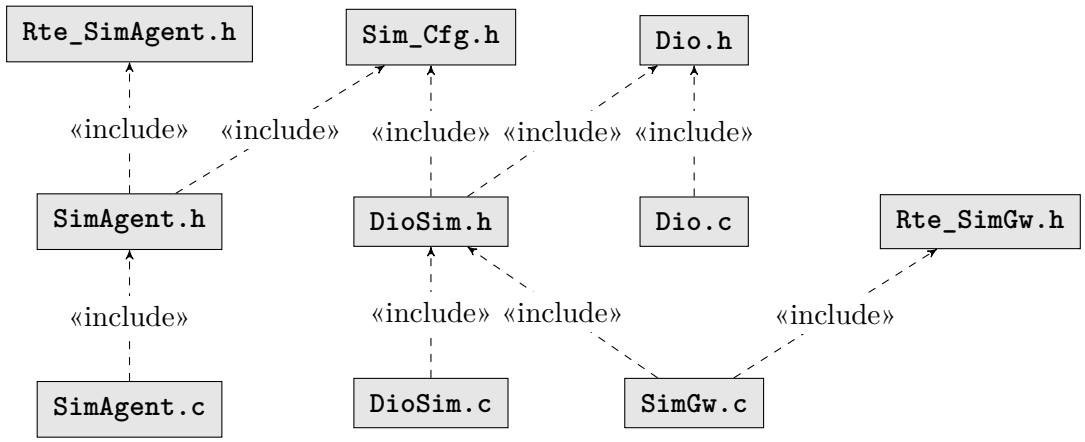


Abb. 4.8.: Include-Hierachie des simModuls

In der SWC simAgent wird durch die Umkonfiguration das Zwischenspeichern der Simulationsdatenpakete verhindert. Die Auflistung 4.11 zeigt die durch das Makro ergänzte Funktion. Die Funktion `SimAgent_AddTestcase_Entry()` kann nur Datenpakete speichern, wenn der Konfigurationsparameter aus Auflistung 4.10 auf `SIMULATION_MODULE_ACTIVE==STD_ON` steht.

Auflistung 4.11: `simAgent.c` - Deaktivierung `SimAgent_AddTestcase_Entry()`

```
134 FUNC(void, SimAgent_CODE) sym_SimAgent_DataHandler_rbl(void){
135
136   if (ECU_ID == data->ecuId){
137     PRP_INFO(GREEN("SimAgent_DataHandler_rbl - Processing Data for this ECU\n
          "));
138 #if (SIMULATION_MODULE_ACTIVE == STD_ON)
139     SimAgent_AddTestcase_Entry(channelId, value, startOffset);
140 #endif
141   }
142 }
```

In der `DioSim.c` (Auflistung 4.12) ist die Sicherheitsfunktion wie folgt implementiert. Ist die Konfiguration `SIMULATION_MODULE_ACTIVE==STD_OFF` und eine `SIM_Run_Rx` empfangen, wird diese abgefangen und der Status deaktiviert gesetzt.

Auflistung 4.12: `DioSim.c` - Deaktivierung `DioSim_EnableSimulation()`

```
85 FUNC(void, DIO_CODE) DioSim_EnableSimulation(DioSim_SimulationState enable){
86 #if (SIMULATION_MODULE_ACTIVE == STD_OFF)
87   DioSim_Enabled = DIOSIM_SIMULATION_DISABLED;
88   PRP_INFO(RED("DioSim_EnableSimulation - Disable simulation due to
         deactivated sim module \n"));
89 #else
90   DioSim_Enabled = enable;
91 #endif
92 }
```

Zusätzlich wird redundant im DioSim mittels des Makros die Funktion `DioSim_GetSimState` deaktiviert. Auflistung 4.13 zeigt die implementierte Lösung.

Auflistung 4.13: `DioSim.c` - Deaktivierung `DioSim_GetSimState()`

```
94 FUNC(DioSim_SimulationState, DIO_CODE) DioSim_GetSimState(void){
95 #if (SIMULATION_MODULE_ACTIVE == STD_OFF)
96   return DIOSIM_SIMULATION_DISABLED;
97 #else
98   return DioSim_Enabled;
99 #endif
```

Die Abbildung 4.9 zeigt in der Form eines Sequenzdiagramms den Ablauf der Sensorwertabfrage einer SWC bei aktivem und deaktiviertem simModul.

Weitere Möglichkeiten der Deaktivierung des Moduls sind möglich über das Entfernen der Komponente aus der Implementierung. Hierbei entsteht aber im Steuergerät eine andere Speicherauslastung und hätte einen erneuten Softwaretest zur Folge, um sicher zu gehen, dass die SWC auf der ECU korrekt funktioniert.

Letzteres könnte die Ausführung über einen Security-Mechanismus wie z.B. ein Passwort sichern, um unerlaubten Zugriff auf die SWC simAgent zu verhindern. Ein weiterer Mechanismus wäre die Funktionsaktivierung oder Deaktivierung über z.B. temporäre Freischaltcodes, wie bereits bei Navigationssystemen eingesetzt. Weitere Methoden sind durchaus denkbar, werden aber nicht weiter untersucht, da sie den Zielen, welche die vorgestellte Methodik verfolgt, nicht direkt dienen.

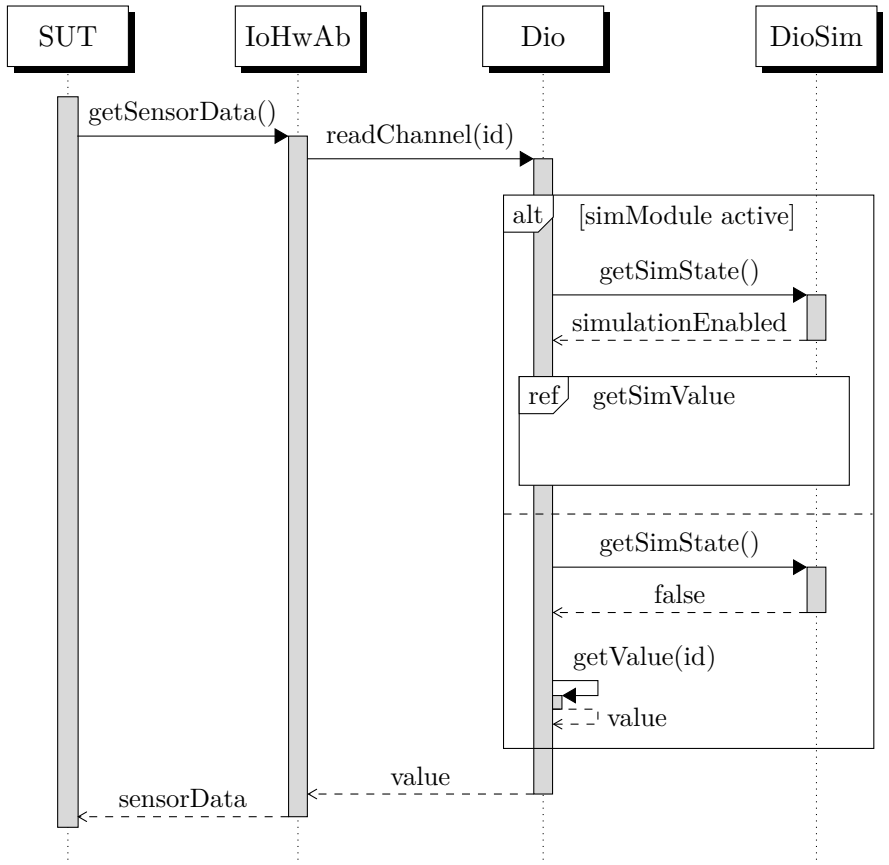


Abb. 4.9.: Sequenzdiagramm - simModul aktiviert / deaktiviert

5. Anwendung

Nach der Implementierung der Methodik in ein generisches Steuergerät kann die Anwendung der Methode gezeigt werden. Die einzelnen Elemente der Methodik, entsprechend dem verwendeten modellbasierten Testansatz von Schieferdecker, wurden dafür umgesetzt und werden im folgenden Kapitel näher erläutert. Die Softwarekomponente simAgent wurde in die s.g. ZBE implementiert sowie in einen Arduino mit dem Ziel der Automatisierung realer Kundeneingaben für die Validierung der Methodik (Kapitel 6). D.h., der Arduino bedient die reale Hardwareschnittstelle der ZBE und ermöglicht hierdurch den exakten Vergleich beider Varianten, real und simuliert, durch die Möglichkeit der Verwendung der identischen Tests.

Die Umsetzung der Modellanalyse und des simMasters erfolgte in einer Toolumgebung namens ECU-Test der Firma TraceTronic. Die Implementierung in diese Umgebung ist aufgrund des internen Supports und der Möglichkeit der Einflussnahme auf Module und Skripte gewählt worden. Die Modellanalyse selbst wurde weitgehendst von der entwickelten Methodik von Christian Saad [Saa14] abgeleitet bzw. übernommen.

Zielbild ist die vollautomatisierte Generierung von Testsequenzen aus verschiedenen Testmodellen und die Einsteuerung in das Embedded System zur Validierung von Kundenfunktionen im verteilten System. Für die Validierung wurden explizit sehr hochgradig vernetzte Funktionen aus verschiedenen Domänen gewählt, um möglichst aussagekräftige Ergebnisse zu erhalten.

Für die Bewertung der Qualität der Kundenfunktionen werden etablierte Testmethoden verwendet, auf die im Folgenden nicht im Detail eingegangen wird. Grundlegend wird anhand von CAN-Botschaften und über das Auslesen von Diagnosedaten die Systemreaktion des verteilten Systems validiert (Kapitel 6).

In den folgenden Abschnitten werden die einzelnen Prozessschritte von der Generierung von Testfällen (Kapitel 5.3) durch die Modellanalyse über die Prüfung der errechneten Pfade bis hin zur Simulation (Kapitel 5.5) erläutert. Die Pfadanalyse wird hier nicht näher erläutert, da diese in [Saa14] ausführlich beschrieben wird. Auf das Scheduling wird unter Kapitel 5.6 eingegangen, da dies im Rahmen der Validierung aus Kapitel 6 einen maßgebenden Einfluss hat.

5.1. Prozess

Prozess aus dem lateinischen procedere = vorwärts gehen beschreibt hier den Vorgang bzw. den Ablauf der Anwendung der Methodik. Das Prozessschaubild (Abbildung 5.1) zeigt anschaulich, wie die Methodik angewendet wird.

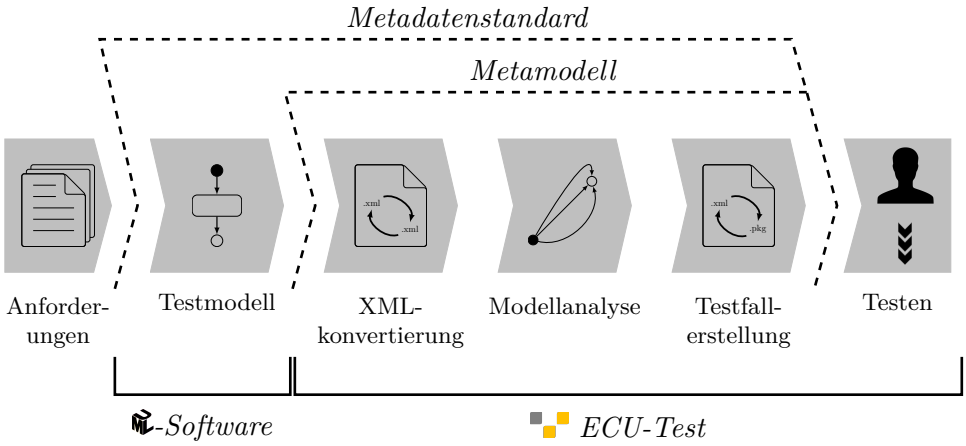


Abb. 5.1.: Prozess - Automatisierte Simulation von Kundeneingaben [Mar16]

Basis für die Automatisierung ist das Testmodell, welches einerseits aus textuellen Anforderungen erstellt, oder aber auch aus einem Anforderungsmodell abgeleitet werden kann. Dieses Testmodell kann nach einer Transformation auf ein Metamodell durch eine Modellanalyse, je nach Testmodell, in endlich oder auch unendlich viele Wirkketten (Pfade) umgewandelt werden. Aus diesen automatisiert erstellten Pfaden werden die Signalsequenzen für die sim-Agents mit Hilfe des Mappings (aus dem Deployment) extrahiert und somit in Testfälle für die entwickelte Methodik für das verteilte System umgewandelt. Diese werden dann einzeln oder in ermittelter Reihenfolge bzw. Funktionsclustern über den simMaster in das verteilte System eingespielt. In Abbildung 5.1 dargestellt durch den Block Testen.

Manuelle Tätigkeiten sind aktuell nur in der Modellerstellung sowie in der Auswahl der Funktionscluster für das Testing erforderlich. Der Prozess zeigt auch die Unabhängigkeit von Tools in der Modellierung sowie der Testautomatisierung, solange Schnittstellen korrekt bedient werden.

Um den Input für die automatisierte Simulation von Kundeneingaben zu darzustellen, wurde die Methodik zur Generierung von Testfällen für die Domänen Energiebordnetz und Infotainment umgesetzt.

5.2. Testfälle und Use Cases

Entscheidend für eine effiziente Automatisierung der Testabläufe von verteilten Softwarefunktionen sind die Testfälle. Entsprechend dem modellbasierten Testansatz von Schieferdecker aus Kapitel 2.2, welcher als Ausgangspunkt zugrunde gelegt wurde, ist hier die Verbindung von Use Cases auf den Testebenen Akzeptanz- und Systemtest (Kapitel 2.2.4 - Abbildung 2.4) ausschlaggebend.

Der Fokus im Rahmen der entwickelten Methode liegt darin, die Testfälle vollautomatisiert aus einem Testmodell, wie in Kapitel 5.3 beschrieben, auf den Testebenen Akzeptanz- und Systemtest zu generieren. Durch die neue Schnittstelle der in Kapitel 3 beschriebenen Automatisierungsmethodik besteht die Möglichkeit Tests in großer Zahl automatisiert abzuarbeiten.

Konventionelle Testfälle sind einfach gesagt Handlungsanweisungen an eine Person, die eine gewisse Systemreaktion hervorrufen und diese bewerten. Im klassischen Testing geschieht das durch Testlisten oder Testfälle in tabellarischer bzw. textueller Form. Von dieser Konvention soll das Testmodell (Kapitel 2.2.1) Abhilfe schaffen und auf Use Cases überleiten. Use Cases bündeln mehrere Tests, um einen Anwendungsfall bzw. die dem Anwendungsfall zu Grunde liegende Anforderung zu bestätigen. Diese Herangehensweise nach Use Cases, oder auch Szenarien genannt, ist für die Modellierung entscheidend. Modellierung schafft Potentiale zur Automatisierung, lässt komplexe Sachverhalte einfacher wirken und ermöglicht einfacher die Berechnung von z.B. Wechselwirkungen von verteilten Funktionen. Die Modelle sind in Form von Use Cases einfacher zu verstehen, und für die automatisierte Berechnung eröffnet sich die Möglichkeit, je Use Case alle notwendigen Tests zu ermitteln. Dies beinhaltet Positiv-Tests, die die korrekte Funktionalität der Funktion und des Gesamtsystems bestätigen, sowie Negativ-Tests, die z.B. eine Fehlbedienung oder das Erreichen bzw. Überschreiten zeitlicher Grenzen prüfen.

5.2.1. Terminologie

Für das eindeutige Verständnis der Anwendung der Methode sind Begriffsklärungen erforderlich.

Testfall	bezeichnet eine Abfolge von einer oder mehreren Funktions- bzw. Systemeingaben mit mindestens einer Funktions- bzw. Systemreaktion, die eine Bewertung der Reaktion beinhaltet.
Use Case	bezeichnet den Kundenfall, in dem dieser eine gewisse Funktion direkt, durch eine Interaktion mit dem System oder auch indirekt durch sein Zeit-Verhalten auslöst bzw. bedient.

Signalsequenz bezeichnet die extrahierte Sequenz des Pegelverlaufs eines Channels für die Systemein- oder -ausgabe und kann z.B. in Form von Tupeln oder Splines vorliegen.

Digital Input bezeichnet nach Spezifikation HW-Eingänge, die nur Boolesche Werte an die Applikation liefern können. Datentyp (Boolean)

Analog Input bezeichnet nach Spezifikation HW-Eingänge, die quasi Fließkomma-Werte an die Applikation liefern können. Dies beinhaltet die Digitalisierung und somit Diskretisierung von Signalen. Datentyp (Integer).

Weitere Definitionen sind im Glossar im Anhang A zu finden.

5.2.2. Tools

Tools sind Werkzeuge die gewisse Aufgaben innerhalb des Prozesses ausführen und dienen zur Unterstützung der Durchführung einer Methodik. Ziel im Rahmen der Arbeit war es eine toolunabhängige Methodik zu schaffen und die einzelnen Prozessschritte so zu gestalten, dass beliebige Tools eingesetzt werden können.

Testmodell Testmodelle können, spezifisch bezogen auf ihr Ziel in verschiedener Form erstellt werden. Hierunter fallen Struktur-, Verhaltens- und Interaktionsdiagramme. Naheliegend für die Funktionsbeschreibung sind vor allem Verhaltensdiagramme wie z.B. das Zustands-, Aktivitäts- oder Anwendungsfalldiagramm. Für die Modellierung von z.B. Strukturen werden Strukturdiagramme wie z.B. Komponenten- oder Klassendiagramme verwendet. Die Architektur eines Fahrzeugs bzw. eines verteilten Systems ist hierbei für die Funktionsmodellierung nicht von entscheidender Bedeutung. [Mar16]

Diese Verhaltensdiagramme können toolunabhängig modelliert werden, einzige Anforderung ist, dass die Annotation der Zustände und Transitionen bekannt ist und dadurch auf ein konform zu einem standardisierten Metamodell konvertiert werden kann.

Modellanalyse bezeichnet die Analyse des Modells mittels Algorithmen aus der Graphentheorie. Ziel ist die Berechnung der möglichen Pfade durch das Testmodell oder Teile davon. Anwendung fand eine datenflussbasierte Modellanalyse von Christian Saad [Saa14], umgesetzt in Java, implementiert in Eclipse.

Diese Methodik wurde in das Testautomatisierungstool ECU-Test der Firma Tracetronic integriert, so dass die Testmodelle direkt in der

Test-Ausführenden Toolumgebung importiert werden können. Andere Algorithmen und Implementierung der Modellanalyse-Methodik sind durchaus denkbar. Ausschlaggebend für die Verwendung bzw. Implementierung der genannten Methodik war die Quelle und der damit verbundenen Knowhow Transfer. Eine ausführliche Herleitung der Modellanalyse-Methodik ist in der Dissertation von Christian Saad [Saa14] zu finden.

Testautomatisierung

genannt simMaster ist das Tool für die Durchführung der Testfälle. Es stellt die Schnittstelle zum verteilten System (SUT) dar. Für diese Aufgabe wurde, wie bereits erwähnt ECU-Test verwendet, da es eine offene Implementierung z.B. der Modellanalyse ermöglicht oder auch eine Abstraktion der Schnittstelle zum SUT.

5.3. Generierung von Testfällen

Die automatisierte Generierung von Testfällen aus den unterschiedlichen Typen von Testmodellen bedarf eines Metamodells. Ziel dieses Metamodells ist die Abstraktion der Vielzahl an Modellen auf eine einheitliche Struktur und Annotation.

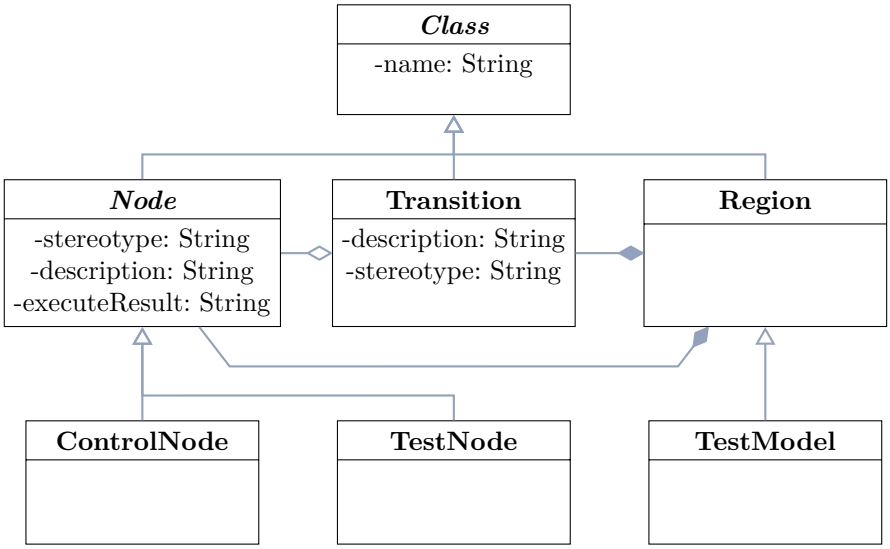


Abb. 5.2.: Metamodell Klassendiagramm - Vereinfachte Darstellung [Mar16]

Die Abbildung 5.2 zeigt einen kleineren Ausschnitt des ganzheitlichen Metamodells M1 basierend auf der Modellanalysemethodik aus [Saa14]. Die abstrakte Klasse *Node* und die Klasse Transition basieren auf der abstrakten Klasse *Class*. Die Assoziationen definieren die semantischen Beziehungen zwischen den Klassen.

Zusammengefasst: Nodes können Control-, Region- oder Testnode sein. Jeder Node muss mindestens eine ausgehende Transition haben, wenn er sourceNode (Beginn der Pfadanalyse) ist. Ebenfalls muss jeder Node der targetNode (Zielnode der Pfadanalyse) ist, mindestens eine eingehende Transition aufweisen. Regions bestehen aus [0..*] Nodes und [0..*] Transitions und sind somit geclusterte Modelle. [Saa14]

Diese auf Abbildung 5.2 dargestellten grundlegenden Klassen und Assoziationen beschreiben die elementaren Bestandteile, die für die automatisierte Generierung der Pfade erforderlich sind. Diese generierten Pfade zeichnen sich durch eine Folge von Knoten (Nodes) aus, welche durch jeweils eine Kante (Transition) verbunden sind.

Für die Pfade P gilt $P = (V, E)$ mit einer endlichen Menge $V \neq \emptyset$ und einer endlichen Menge $E \subseteq (u, v)$ mit $u, v \in V, u \neq v$. V = Menge aller Knoten (engl. vertex) von P. E = Menge aller Kanten (engl. edge) von P. Die Kante $e \in E$ ist also von der Form $e = (u, v)$ mit gewissen $u, v \in V, u \neq v$, wobei u als Anfangsknoten und v als Endknoten bezeichnet wird.

Da in der Generierung der Pfade diese soweit herunter gebrochen werden, dass keine Schleifen oder Mehrfachkanten entstehen, gilt für die Adjazenzmatrix A der Pfade P mit Knotenmenge $V = \{v_1, \ldots, v_n\}$ und der Kantenmenge $E = \{e_1, \ldots, e_{n-1}\}$ der Größe $n \times n$:

$a_{j,k}$ = Anzahl der Kanten mit Anfangspunkt v_j und Endpunkt v_k wobei gilt $j, k \in \mathbb{N}, k = j + 1$.

Die Inzidenzmatrix der Pfade P ist die $n \times n$ Matrix I mit den Einträgen

$$i_{j,k} = \begin{cases} 1 & \text{falls } v_j \text{ Anfangspunkt der Kante } e_k \text{ ist,} \\ -1 & \text{falls } v_j \text{ Endpunkt der Kante } e_k \text{ ist,} \\ 0 & \text{sonst.} \end{cases}$$

$$A = \begin{pmatrix} 0 & 1 & & \cdots & 0 \\ 0 & 0 & 1 & & \vdots \\ & & & \ddots & \\ \vdots & \vdots & \vdots & & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix} \text{ und } I = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

Zusammengefasst werden aus einem ungerichteten Graphen bzw. dem Testmodell gerichtete Graphen extrahiert, die keine Schleifen und Mehrfachkanten in ihrer Form aufweisen. Jedoch kann ein Zustand aus dem ursprünglichen Graphen bzw. Testmodell wiederholt in einem Pfad auftreten, hat jedoch eine neue eineindeutige Bezeichnung. Die Inzidenzmatrix I muss eine Einheitsmatrix ergeben, dies ist der Nachweis, dass nur Vorwärtskanten die aufeinander folgenden Knoten verbinden.

Diese Summe an Pfaden wird im Folgenden mittels einer Semantik in drei verschiedene Testmethoden, *manuell Gesamtsystem*, *automatisiert Teilsystem* und *automatisiert Gesamtsystem* bzw. für den simAgent, überführt. Hierbei dürfen die Informationen aus dem Testmodell bezüglich Kunden- und Systemverhalten bei der Pfadberechnung nicht verloren gehen und müssen zusätzlich mit Informationen, wie z.B. einem Tastendruckverhalten angereichert werden.

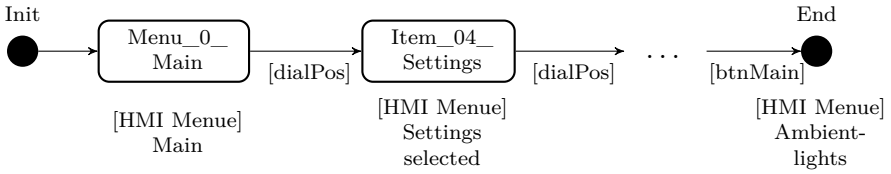


Abb. 5.3.: Beispielpfad - allgemein

manuell Gesamtsystem – Systemstimulation erfolgt über manuelle Bedienung an dem HMI. Systemreaktion wird in der Regel visuell, akustisch oder mit Hilfe von Kommunikationsanalysen bewertet. Daher beinhaltet der Testfall Informationen, welche Sensoren stimuliert werden müssen und woran die Systemreaktion erkannt werden kann.

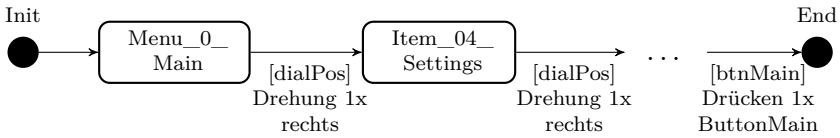


Abb. 5.4.: Beispielpfad - manuell Gesamtsystem

automatisiert Teilsystem – Die Systemstimulation erfolgt über eine Off-Board Testautomatisierung. Systemschnittstellen sind in der Regel die Kommunikationspfade oder HW-Eingänge/Ausgänge von ECUs. Testfälle sind bereits spezifischer und enthalten Informationen über Signale und Botschaften.

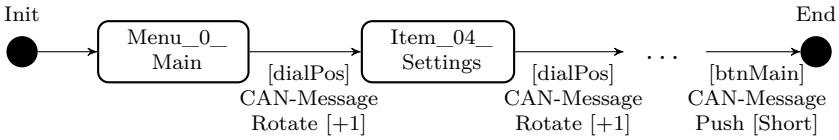


Abb. 5.5.: Beispielpfad - automatisiert Teilsystem

automatisiert Gesamtsystem – Die Gesamtsystem-Stimulation erfolgt über eine integrierte SW-Lösung. Die Systemreaktion wird über die Kommunikationspfade oder z.B. Diagnose bewertet. Hierbei finden konventionelle Methoden ihren Einsatz.

Testmodelle können generische bzw. Derivat übergreifende Funktionen beschreiben, deswegen wird mittels einer Syntax die spezifische Adaption auf eine Funktionsarchitektur und Kommunikation bzw. Signalen erreicht. Diese Abstraktion bringt den entscheidenden Vorteil, dass

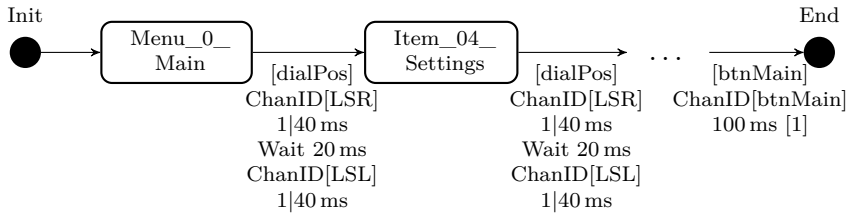


Abb. 5.6.: Beispielpfad - automatisiert Gesamtsystem

die Anzahl der Pfade reduziert wird und mehrfach verwendet werden kann. Des Weiteren können derivat-spezifische Änderungen wie z.B. Signalnamensänderungen unabhängig von den Pfaden erfolgen.

5.4. Simulation im verteilten System

Die Abbildung 5.7 stellt, in der Form eines Sequenzdiagramms den Ablauf einer softwarebasierten Kundeneingabe dar. Den parallel ablaufenden Part, der `simulateChannel`, ist in Kapitel 5.5 separat dargestellt. Im Folgenden wird vom simAgent im Singular gesprochen, dies soll nur der Lesbarkeit und Verständlichkeit des Ablaufes dienen. Des Weiteren stellt das Sequenzdiagramm einen Fall dar, bestehend aus nur einem simAgent, da der Ablauf für mehrere simAgents identisch ist.

Nach dem Start eines Tests `startTest()` durch den Benutzer werden alle noch vorhandenen Simulationsdaten bzw. gespeicherten Werte im simMaster, sofern erforderlich, durch `resetInteralData()` zurückgesetzt.

Anschließend erfolgt das Löschen der noch gespeicherten Simulationsdaten in den simAgents aus vorangegangenen Simulationen über die Nachricht `initBroadcast(init)` mit der `ECUID = 255`. Da über diesen `Init` mittels Broadcast alle im verteilten System befindlichen simAgents angesprochen werden, ist dadurch sichergestellt, dass kein simAgent vergessen wird.

Gefolgt vom Löschen der Daten im verteilten System wird über die gleiche `Init`-Nachricht der für den Testablauf erforderliche simAgent angesprochen, mit der Parametrierung, dass der betreffende simAgent eine Rückmeldung in Form seiner `ECUID` an den simMaster sendet.

Ist der erforderliche simAgent im verteilten System vorhanden, so werden die Test Case Daten über `SendData()` an den simAgent versendet. Dies läuft über die Adressierung in der `SendData()` mittels `ECUID`. Der simAgent speichert (`addData()`) die empfangenen Daten temporär im Arbeitsspeicher. Die Simulationsdaten werden gemäß dem in Kapitel 3.3.5 vorgestellten Protokoll in einzelnen Datenpunkten übertragen. Folglich läuft die Schleife `loop [test steps exists]` in der Anzahl der Datenpunkte und speichert diese Anzahl in der Variable`testLength` ab.

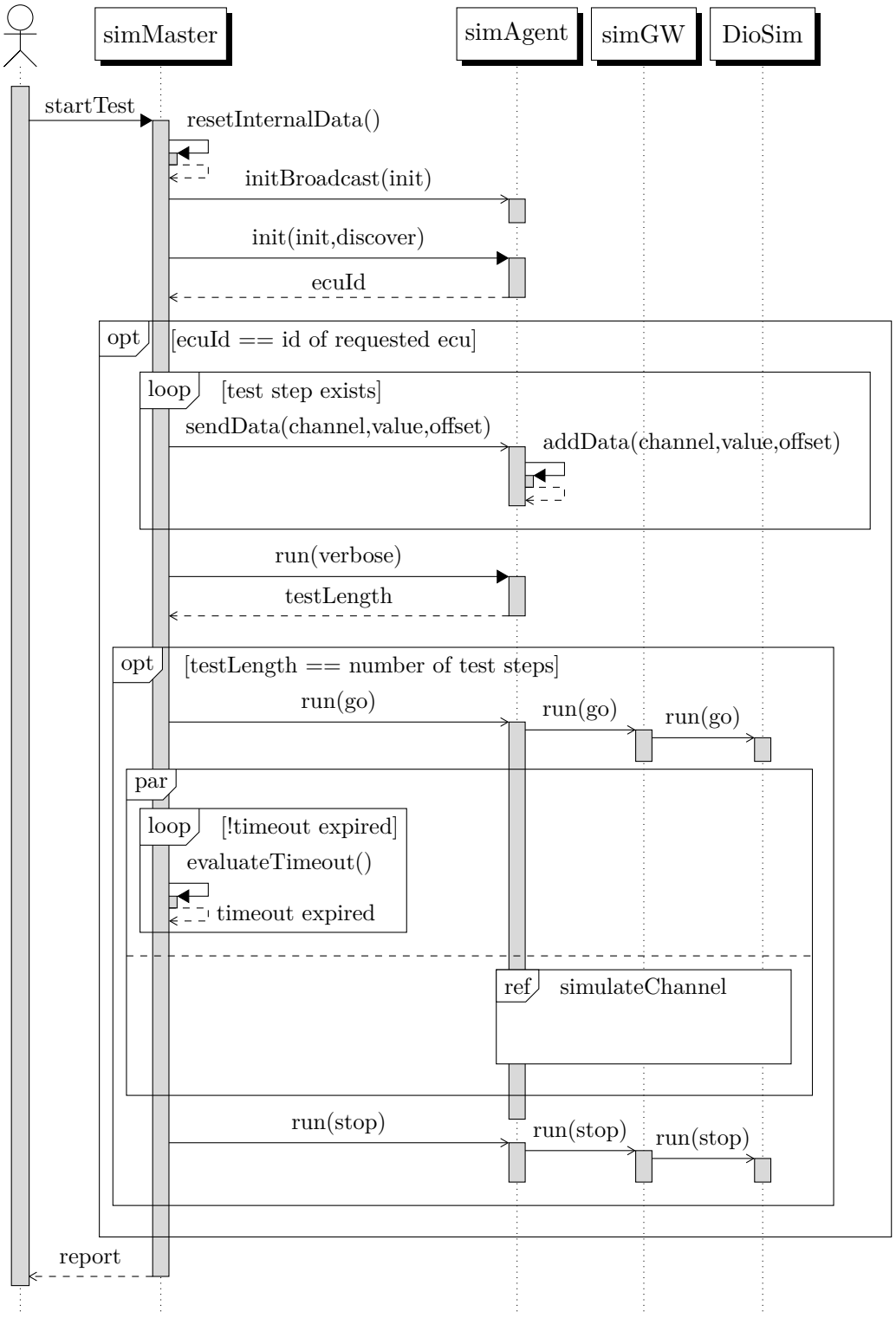


Abb. 5.7.: Sequenzdiagramm - software-basierte Simulation

Die im simAgent gespeicherte Testlänge (`testLength`) kann durch den simMaster über eine `run(verbose)`-Nachricht abgefragt werden. Dies dient dem Master der Überprüfung des Test Case Uploads. Stimmt die `testLength == number of test steps`, gespeichert im simMaster, überein, kann der Testablauf mittels `run(go)` gestartet werden.

Ab dem Befehl `run(go)` laufen der simMaster und simAgent parallel und tauschen bis zum `run(stop)` keine Daten mehr aus. Der simMaster betrachtet in seinem Ablauf die Systemreaktion, der simAgent führt strikt seine Simulation aus. Der steuergeräteinterne Ablauf ist im Kapitel 5.5 dargestellt. Die nicht unterbrochene Kommunikation zwischen simAgent und simMaster ist grundlegend gewollt, da dieser Fall seitens der Lasten auf den Kommunikationspfaden im Simulationsfall näher dem realen Kundenfall ist.

Am Ende der Simulation wird noch über den `run(stop)` jegliche Ausführung in den simAgents gestoppt. Der simMaster kann anhand der Test Case Daten bestimmen, wie lange ein Testfall im simAnget laufen wird. Dies dient auch der Sicherheit, um noch evtl. laufende oder ein verzögertes Abspielen der Simulationsdaten, z.B. in einem Fehlerfall, zu unterbrechen. Dieses Kommando kann auch jederzeit in Simulationsablauf getriggert werden, um den Testablauf abbrechen zu können.

5.5. Interne Simulation des simModuls

Die Abbildung 5.8 zeigt in Form eines Sequenzdiagramms den internen Ablauf im simModul. Dieser ist, wie bereits beschrieben, sehr einfach gehalten und besteht nur aus einem zyklischen Update des DioSim ab Aktivierung des simAgents.

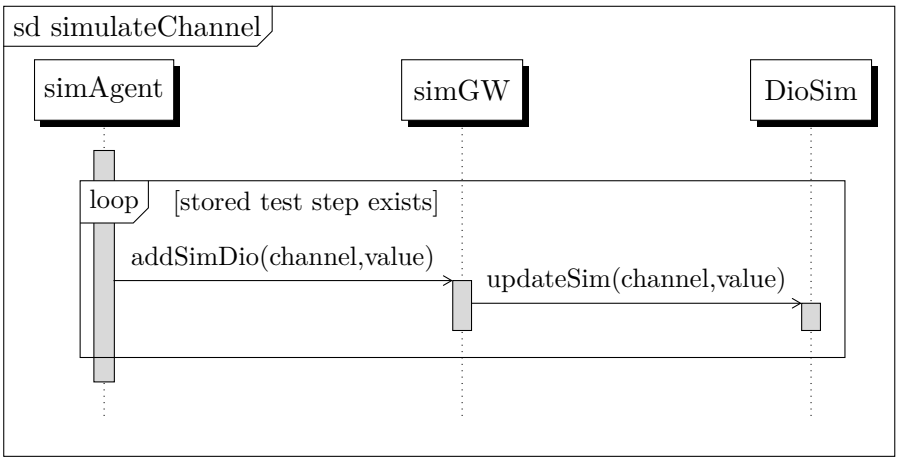


Abb. 5.8.: Sequenzdiagramm - simModul interne Signal Simulation

5.6. Scheduling des simAgent

AUTOSAR verfügt über zwei verschiedene Taskmodelle für das Scheduling bzw. die zeitliche Reihenfolge, in der die einzelnen Tasks abgearbeitet werden. *Basic Tasks* können die Zustände *running*, *ready* und *suspended* einnehmen. *Extended Tasks* können neben den Zuständen, die *Basic Task* einnehmen können, noch den Zustand *waiting* einnehmen. Der Scheduler im AUTOSAR arbeitet nach Prioritäten und hat für jede der Prioritätsstufen eine eigene First in First out (FIFO)-Schlange. Zusätzlich lässt sich auch eine Reihenfolge der Tasks festlegen, um Abhängigkeiten zwischen Tasks bereits bei der Konfiguration berücksichtigen zu können.

Um eine Reproduzierbarkeit der Simulationswerte in Bezug auf die Funktionen zu erreichen, muss die Reihenfolge der Tasks in der Komponente *simAgent vor SUT* sein. Hinzu kommt, dass der Taskzyklus t_z des simAgent ein Vielfaches des SUT sein sollte, um hier eine Drift der Simulation zu verhindern.

$$t_z(simAgent) = \frac{t_z(SUT)}{i} \qquad | \, i \in \mathbb{N}$$

Das RE `sym_SimAgent_UpdateCDD_cyclic` des simAgents muss daher in der Taskreihenfolge richtig verankert werden. Diese RE des simAgents wird alle 10 ms auf der Hardware ausgeführt, im Gegensatz zu der RE des SUT `SwcZBE_cyclic` die nur alle 60 ms ausgeführt wird. An dieser Stelle ist ein statisches Scheduling erforderlich, bei der der simAgent in der Reihenfolge vor dem SUT ausgeführt wird. Die folgende Auflistung zeigt die Reihenfolge der Tasks.

1. sym_SimAgent_Starter_cyclic
2. sym_SimAgent_UpdateCDD_cyclic
3. sym_SimGw_cyclic
4. HwIoAb_DIO_ReadZBERotation_cyclic
5. SwcZBE_cyclic

Zudem hat der simAgent dadurch die Möglichkeit mit seiner Ausführung der Simulation auf den richtigen Zyklus des SUT zu warten. Sequenzdiagramm 5.9 zeigt, wie der simAgent auf SUT wartet (`wait`).

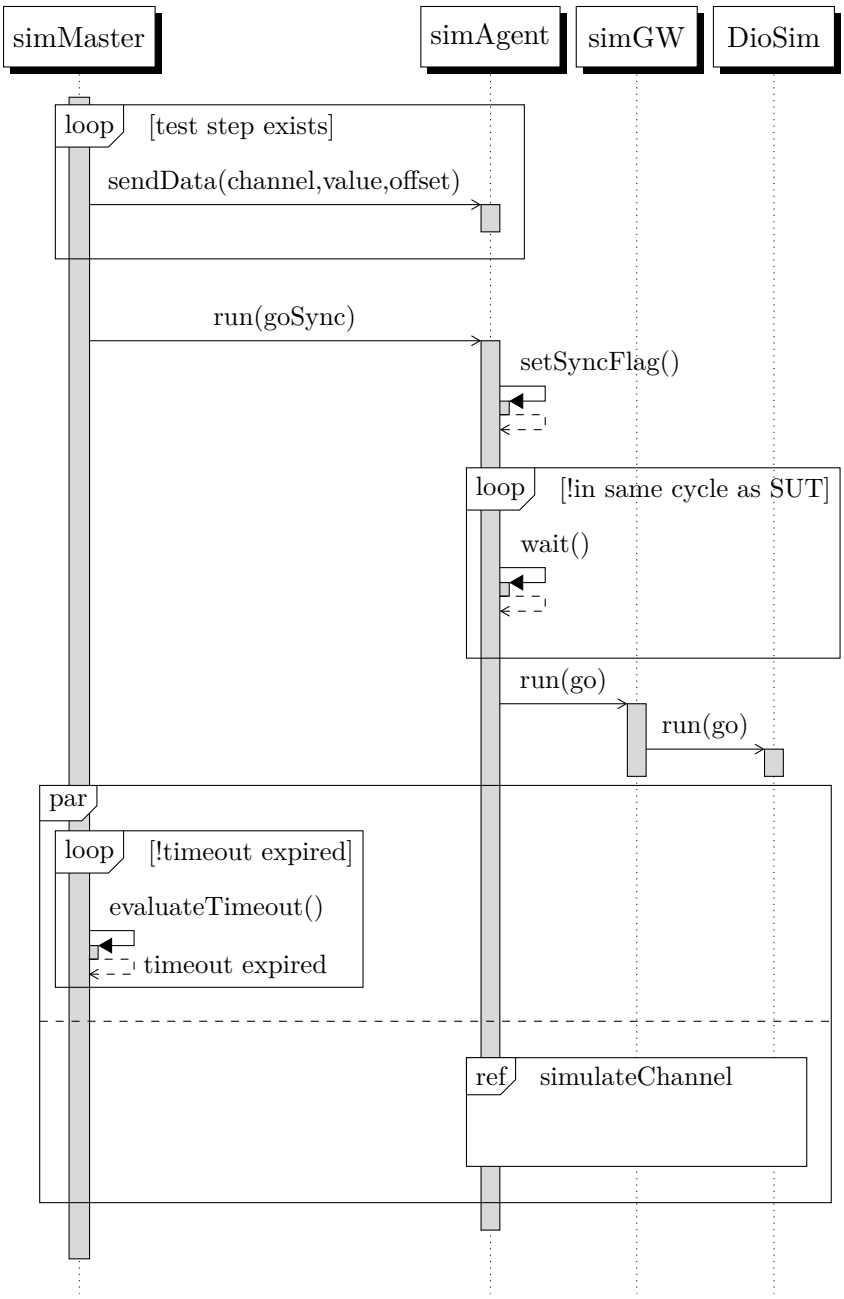


Abb. 5.9.: Sequenzdiagramm - synchronisierte Simulation

6. Validierung der Methode

Für die Validierung der Methode werden Softwarevalidierungsmethoden, beschrieben von Patricia B. Powell [Pow86], eingesetzt. Von besonderer Bedeutung sind Methoden wie *Algorithm Analysis, Software Monitors* und *Peer Review*. Die Validierung der Methode erfolgt Bottom-up von der Codeanalyse bis zur Analyse an Fallstudien bzw. bestimmten Use Cases in realen verteilten System. Die Validierung soll die folgenden Punkte aus Kap. 1.4 bestätigen:

Komplexität	Beherrschung der Komplexität von verteilten Funktionen über einen modellbasierten Ansatz.
Automatisierung	Ablösen von wiederkehrenden manuellen Tätigkeiten durch den Einsatz von einem software-basierten Ansatz.
Qualität	Verbesserung der Produktqualität (Softwarequalität) durch den Einsatz von software-technischen Systemen.

6.1. Validierungsmethoden

Algorithm Analysis – Algorithmenanalyse wird in zwei Aktivitäten unterteilt.

A-priori-Analyse	wird vor der Ausführung von Algorithmen durchgeführt. Hierbei wird ermittelt welche Zeit und Ressourcen ein Algorithmus benötigt, bis er zum Ergebnis kommt. Beispiel: WCET-Analyse [BEL05].
A-posteriori-Analyse	analysiert die berechneten Kennzahlen während Laufzeit des Algorithmus.

Mit Hilfe der beiden Analysen kann eine Aussage über die Performanz des Algorithmus getroffen werden.

Software Monitors – Die Analyse mittels Software Monitors Methode soll ineffizient arbeitende Stellen eines Programms aufzeigen. Während der Laufzeit werden Kennzahlen des implementierten Algorithmus aufgezeichnet und im Nachgang bewertet. Elementare Kennzahl ist wiederum die Ausführungszeit, die die CPU benötigt, ebenso wie Arbeitsspeicherauslastung während der Ausführung.

Peer Review – ist die manuelle Untersuchung des Programmcodes von Personen mit gleicher Ausbildung und Erfahrung. Folgende Formen des Peer Reviews sind etabliert: *Code Reading Review, Round- Robin Review, Walkthroughs* und *Inspections*.

6.2. Analyse an Fallstudien

Eine Fallstudie ist eine wissenschaftliche Untersuchung eines einzelnen Sachverhalts, die beispielhaft Probleme aufzeigen soll [Dud17]. Die ausgewählten Szenarien zeigen Situationen in denen Probleme auftreten können. Die, den Fallstudien zugrundeliegenden Probleme, beziehen sich auf die genannten Schwerpunkte aus Kapitel 1.3.

Die ausgewählten Fallstudien bzw. Use Cases dienen der Veranschaulichung der Validierung und stellen einen besseren Bezug zu den verwendeten Validierungsmethoden dar.

Use Case – Navigationssystem. Die Steuerung des fahrzeugeigenen Navigationssystems über die zentrale Bedieneinheit. Ziel ist es, hierbei eine Zieleingabe für das Navigationssystem durchzuführen und die Auswirkungen einer reproduzierten Eingabe auf Komponenten des verteilten Systems zu analysieren. Dieser Use Case stellt eine mit der ZBE minimal verteilte Funktion dar. Beteiligte Komponenten sind nur die ZBE sowie die HU, andere Komponenten sind optional und werden nicht benötigt.

Bedienablauf für den Kunden mit der ZBE: Hauptmenü → Navigation → neues Ziel eingeben → Ort und Adresse eingeben → Navigation starten [BMW15a]

Der Fokus liegt hier auf der Reproduzierbarkeit der Kundenfunktion von dynamischen Systemen.

Use Case – Lichtfunktionen. Steuerung von Lichtfunktionen in der HU über die ZBE. Dieser Use Case stellt auch eine verteilte Funktion dar, bei der der Fokus auf dem Wechsel der Transportschicht liegt. Dieser Use Case betrachtet die Komponenten ZBE, HU, BDC und die LIN-Lichtmodule. Hierbei wird über die ZBE die Lichtfunktion in der HU bzw. BDC parametriert und die Parameter an die einzelnen Lichtmodule weitergeleitet.

Bedienablauf für den Kunden mit der ZBE: Hauptmenü → Mein Fahrzeug → Fahrzeugeinstellungen → Licht → Innenbeleuchtung → Ambiente-Beleuchtung [BMW15a]

Der Fokus liegt auf der Reproduzierbarkeit der Kundenfunktionsreaktion im statischen System mit dem Wechsel der Kommunikationsschicht von CAN auf LIN über diverse SWCs.

Im Folgenden wird Bottom-up nach den aufgezeigten Validierungsmethoden aus 6.1 vorgegangen und je nach Methode auf die genannten Use Cases im Detail eingegangen.

6.2.1. Algorithmen und Datenstrukturen

Code Analyse bzw. Peer Review ist ein statisches Software-Testverfahren, eingeordnet in das White-Box Testverfahren. Der Code wird hierbei einer Reihe von formalen Prüfungen unterzogen, um inhaltliche Fehler und Auswirkungen zu identifizieren und anschließend bewerten zu können. Die Validierung auf der untersten Ebene folgt der Methode eines Peer Reviews und soll ineffiziente Datenstrukturen und Codesegmente aufzeigen. Diese Validierung bedarf keines Bezugs auf die Fallstudien und erfolgt durch reines Codereview.

Simulationsdaten

Die SWC simAgent nutzt ein Array (Auflistung 6.1) für die Zwischenspeicherung der Simulationsdatensätze. Dieses Array ist wie folgt definiert:

Auflistung 6.1: `simAgent.h` - Array `SimAgent_Testcase_Entry`

```
45 typedef struct {
46   ImpDTuint32 StartOffset;
47   ImpDTuint16 ChannelId;
48   ImpDTuint2 Value;
49 } SimAgent_Testcase_Entry;
```

Die Größe eines Elements beträgt acht Byte. Die Ausrichtung der Datenelemente im Speicher ermöglicht einen Typen für das Feld `Value`, der bis zu 16 Bit groß ist ohne Auswirkungen auf den Arbeitsspeicher. Dies hat zur Folge, dass auch z.B. analoge Kanäle simuliert werden können, die mit 2^{16} diskreten Werten auskommen. Wird der Wert 2^{16} überschritten, erhöht sich der Speicherbedarf des gesamten Arrays. Abbildung 6.1 zeigt den linearen Zusammenhang zwischen Elementgröße und Speicherbedarf.

Bereitstellen der Simulationswerte

Im Dio-Sim Modul werden die Daten in einem Array von Kanal und Werte-Paaren bereitgestellt. Dies ermöglicht für den Treiber einen schnellen Zugriff auf die Simulationsdaten. Auflistung 6.2 zeigt die Definition des Arrays und ergibt aufgrund seiner Typen `unsigned char` bzw. `unsigned int` für `Dio_ChannelType` und `Dio_LevelType` eine Elementgröße im RAM von vier Byte.

Auflistung 6.2: `DioSim.h` - Array `DioSim_Entry`

```
14 typedef struct {
15     Dio_ChannelType ChannelId;
16     Dio_LevelType Value;
17 } DioSim_Entry;
```

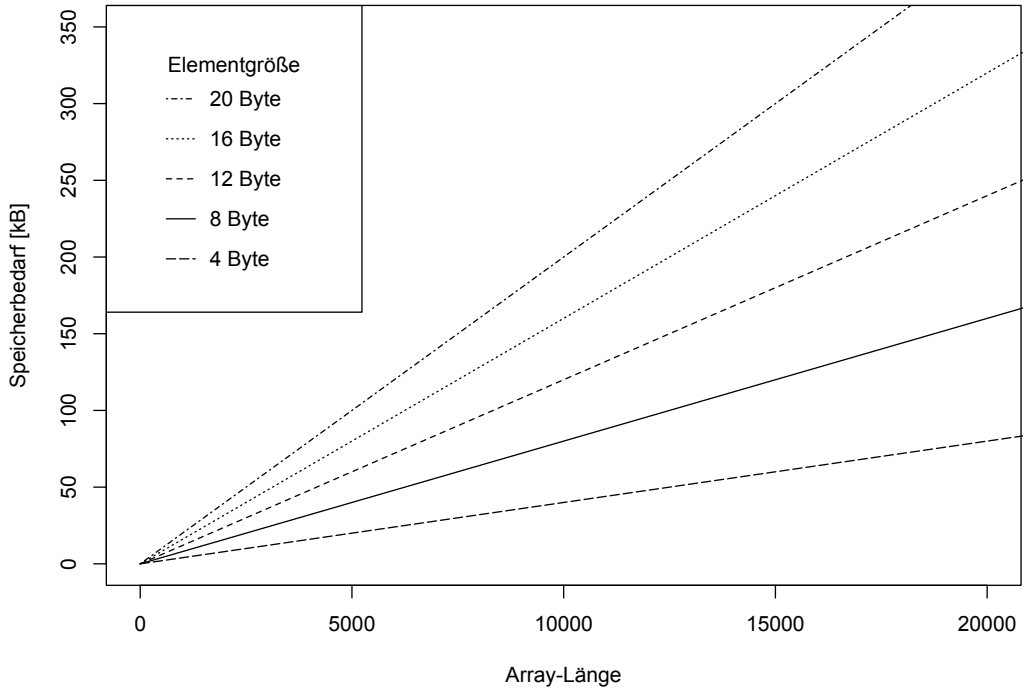


Abb. 6.1.: Speicherbedarf in Abhängigkeit der Elementgröße

Bei einer implementierten Arraylänge von 254 entspricht dies einem Speicherbedarf von 1.016 Byte im Steuergerät. Die Größe des Arrays verändert sich nicht, da die Datenstruktur des `DioSim` nur digitale Werte vorhalten muss. Für andere I/O-Treiber muss jeweils ein eigenes Array definiert werden, hierbei entstehen aber auch je nach Implementierung nur geringe Speicheranforderungen an das Steuergerät.

6.2.2. Software in the Loop

Nach der Analyse der Auswirkungen von Algorithmen und Datenstrukturen steht im Folgenden die Softwarekomponente auf dem Steuergerät im Fokus der Analyse. Es wird in erster Linie eine Methode der *A-posteriori-Analyse*, also der Ermittlung von Kennzahlen während Laufzeit, angewendet.

Die folgenden Analysen beziehen sich auf den Use Case – Lichtfunktionen, beschrieben in Kapitel 6.2.

Auswirkungen auf die CPU Last

Um die Auswirkungen einer aktiven Simulation auf die CPU-Last zu bestimmen, ist ein Shell-Skript erstellt worden, welches sekündlich Statistiken des Linux Betriebssystems sammelt. Aus diesen Statistiken kann die Rechenzeit der einzelnen Tasks ermittelt werden. Die Präzision der Daten ist abhängig von der Anzahl der Clock Ticks, welche bei der verwendeten VAP bei 100 liegt. Hierdurch ist eine Auflösung von einem Prozentpunkt möglich.

Für die Erstellung der Messreihen wurde die Anzahl gleichzeitig zu simulierender Kanäle, von einem auf bis zu 32, stetig erhöht (Tabelle 6.1). Aufgrund der implementierten Grenze von 10000 Datensätzen für die Summe der Kanäle limitiert es die mögliche Datensatzlänge je Kanal. Daraus ergab sich eine Reduzierung der Datensatzlänge je Kanal ab einer Simulation von mehr als 16 Kanälen und eine dadurch reduzierte Dauer der Tests. Jeder Datensatz wurde bis zu 300 mal in das System eingespielt (Tabelle 6.1).

Kanäle	Ausführungen	Datensätze	Dauer [s]
1	300	600	12
2	300	1200	12
4	300	2400	12
8	300	4800	12
12	300	7200	12
16	300	9600	12
20	250	10000	10
24	200	9600	8
28	175	9800	7
32	150	9600	6

Tab. 6.1.: Kennzahlen der Messreihen für SIL-Tests

Die Abbildung 6.2 zeigt den zeitlichen Verlauf der Prozessorauslastung bei steigender Anzahl an zu simulierender Kanäle. Der Abfall der Kurven in den Randbereichen lässt sich mit der Abtastverschiebung zu Simulations-SWC Takt erklären, sowie durch Effekte des internen Scheduling der Tasks, da im ersten und letzten Taskzyklus die Anzahl an simulierender Kanäle ansteigt bzw. abfällt. Dies erzeugt ein gewisses Anlaufen bzw. Abschwächen der CPU-Last.

Tabelle 6.2 zeigt die gemittelten Prozessorlasten, über die Anzahl an Ausführungen, bezogen auf die Anzahl der simulierten Kanäle. Der Anstieg bezieht sich immer auf den unbelasteten Zustand, Anzahl simulierter Kanäle = 0. Auffällig ist der starke Anstieg der CPU-Last, der bei den Messungen mit 28 und 32 Kanälen zu einem Ausfall des Steuergeräts führte. Aus diesem Grund sind hier keine Messungen möglich gewesen. Dieses nicht erwünschte Verhalten hatte eine genaue Untersuchung der einzelnen Elemente zur Folge. Anmerkung: Die scheinbar genauere Auflösung ergibt sich aus der Mittelung der Messwerte.

Kanäle	CPU-Last [%]	Anstieg [%]
0	22,0	-
1	24,2	2,2
2	26,2	4,2
4	28,0	6,0
8	32,0	10,0
12	37,6	15,6
16	41,8	19,8
20	46,4	24,4
24	51,0	29,0
28	-	-
32	-	-

Tab. 6.2.: Ergebnisse CPU-Last

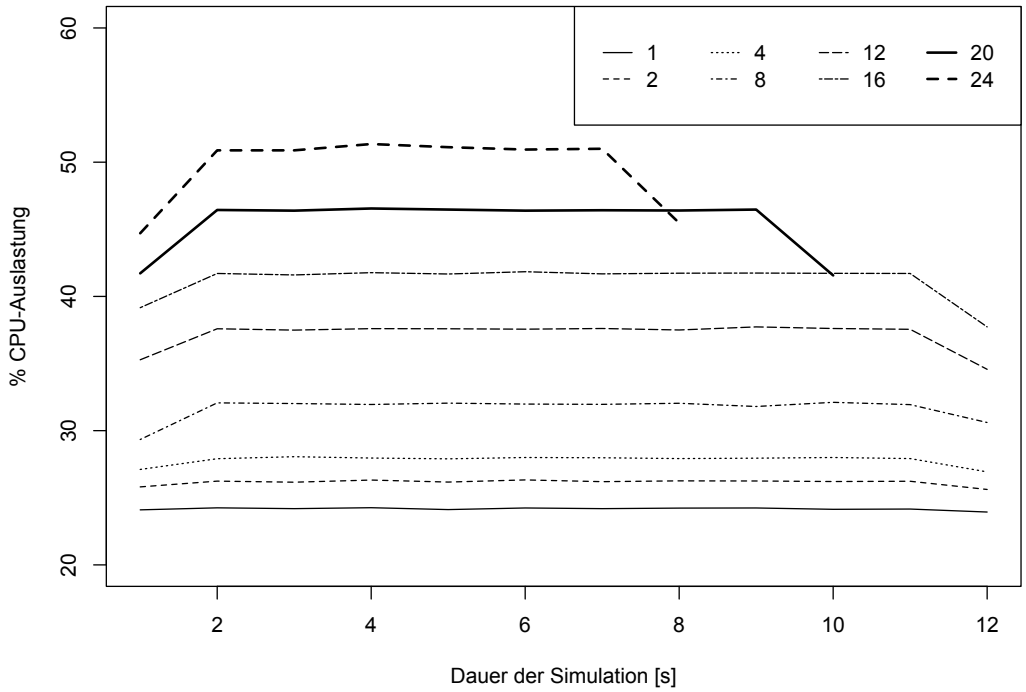


Abb. 6.2.: CPU-Last über Simulationsdauer

Durch eine intensive Analyse der einzelnen Elemente des simModuls wurde die Client/-Server-Verbindung von RTE und simGW als Auslöser für die Ineffizienz ermittelt. Der Aufruf der simGW-Funktion wurde als Ausführungswunsch deklariert und hatte zur Folge, dass sich die Ausführungswünsche in einer Warteschlange wiederfanden. Durch die Verwaltung der Warteschlange kommt es zu dem auf Abbildung 6.2 dargestellten Effekt.

Dieser Effekt kann nahezu vollständig durch die in der AUTOSAR-Spezifikation [AUTi] beschriebene Möglichkeit [SWS_Rte_07409] eliminiert werden. Hierdurch verändert sich der Client/Server-Aufruf in einen direkten Funktionsaufruf. Diese Umstellung auf direkten Funktionsaufruf wird in diesem Fall nur für das simModul angewendet, andere Software-Komponenten im Steuergerät können, wenn erforderlich, auf anderen Verbindungstypen bestehen bleiben.

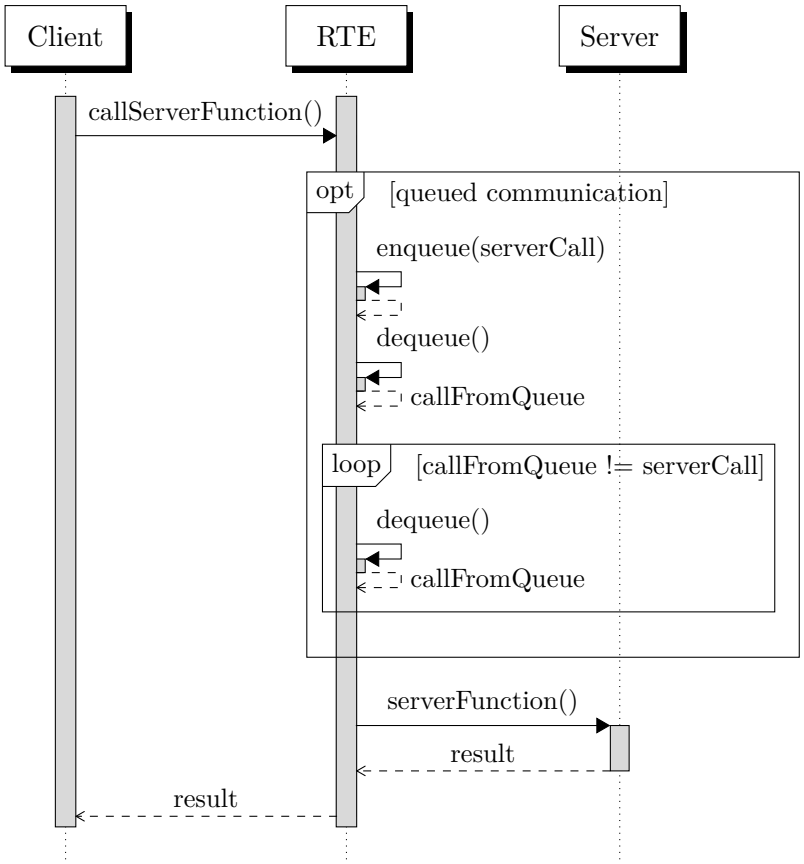


Abb. 6.3.: Sequenzdiagramm unterschiedliche Client/Server-Aufrufe

Auf Abbildung 6.3 sind die unterschiedlichen Typen der Aufrufe dargestellt: direkter Funktionsaufruf und Client/Server-Kommunikation. Nach einer Optimierung der Client/Server-Kommunikation zu einem direkten Funktionsaufruf zeigt sich folgendes verbessertes Verhalten der CPU-Last, dargestellt auf Abbildung 6.4 und in Tabelle 6.3. Diese Ergebnisse wurden mit den identischen Datensätzen durchgeführt, die zu Ergebnissen wie auf Abbildung 6.2 führten.

Auf Abbildung 6.4 ist eine nachvollziehbare Tendenz zu einer höheren CPU-Last zu sehen, die entsteht, je mehr Kanäle simuliert werden. Diese optimierte Lösung mit einem direkten Funktionsaufruf zeigt generell sehr geringe Auswirkungen auf die CPU-Last. Die CPU-Last Mehrung beschränkt sich auf $< 2\,\%$.

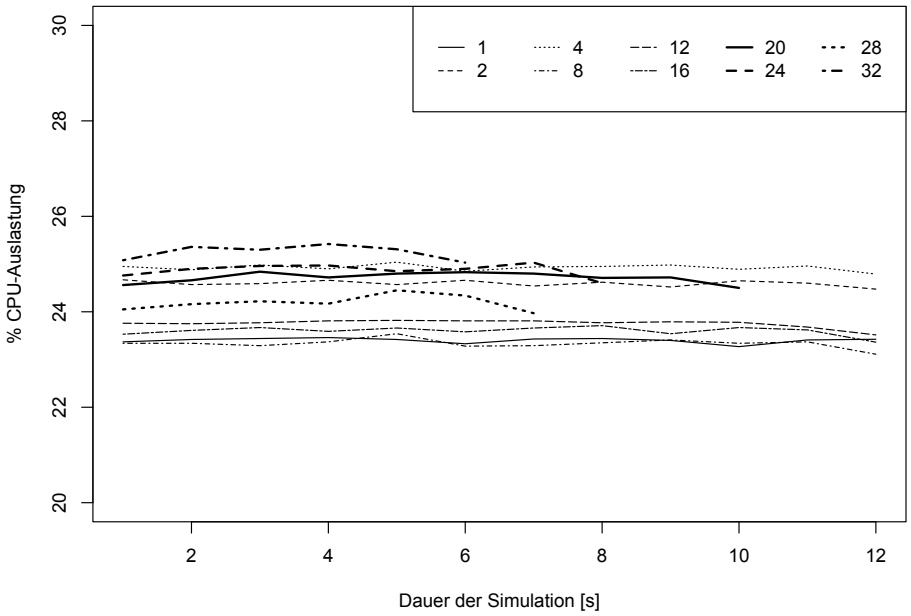


Abb. 6.4.: CPU-Last über Simulationsdauer - RTE-Kommunikation optimiert

Kanäle	CPU-Last [%]	Anstieg [%]
0	24,0	-
1	23,4	-0,6
2	24,6	0,6
4	24,9	0,9
8	23,4	-0,6
12	23,8	-0,2
16	23,6	-0,4
20	24,8	0,8
24	24,9	0,9
28	24,2	0,2
32	25,3	1,3

Tab. 6.3.: Ergebnisse CPU-Last optimiert

Abbildung 6.5 zeigt den Vergleich der CPU-Last Mehrung bei steigender Anzahl zu simulierender Kanäle der beiden Varianten der Client/Server-Kommunikation. Die optimierte Variante des direkten Funktionsaufrufs zeigt bei einem Anstieg der simulierten Kanäle ein deutlich verbessertes Verhalten.

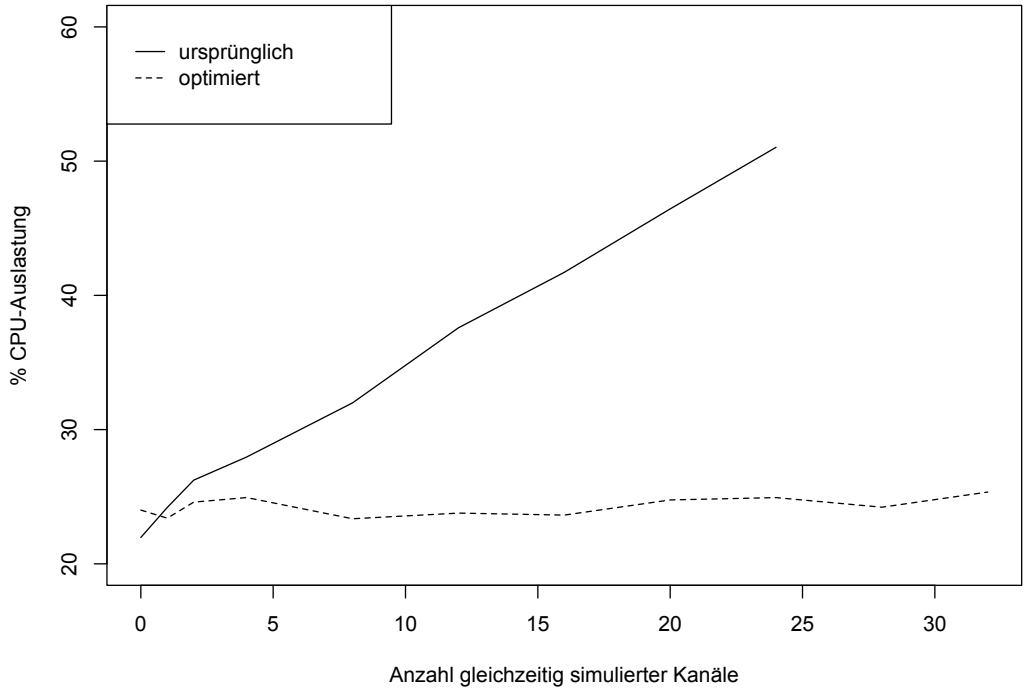


Abb. 6.5.: CPU Auslastung im Vergleich

Speicherbedarf

Der Speicherbedarf, der durch die Integration des simModuls entsteht, wird in dem statischen und dynamischen Bedarf getrennt betrachtet. Statischer Bedarf ist der Bedarf für die SWC simModul. Dynamischer ist der Bedarf des RAM des Steuergeräts, dieser ist für die Ausführung und Testfalllänge entscheidend.

Statischer Speicherbedarf – Durchgeführt wurde eine Differenzbetrachtung der kompilierten Steuergeräte Software in drei Szenarien. Hierfür wurde die SWC in den drei Szenarien wiederholt kompiliert.

- Steuergeräte Software ohne simModul bzw. originale Seriensteuergerätsoftware. Die Steuergerätesoftware hat einen Speicherbedarf von 5.264 kB.

- Steuergeräte Software mit aktivem simModul. Durch das simModul kommt ein Speicherbedarf von 25 kB hinzu, somit hat die Software eine Gesamtgröße von 5.289 kB.

- Steuergeräte Software mit integriertem simModul, aber wie in Kap. 4.4.13 beschrieben deaktiviertem simModul. Der zusätzliche Speicherbedarf beträgt 24 kB.

Diese Analyse zeigt einen zusätzlichen statischen Speicherbedarf von bis zu 25 kB für das integrierte simModul.

Dynamischer Speicherbedarf – ist der Bedarf an Speicher während der Ausführung. Einfluss auf den Arbeitsspeicherbedarf haben die Datenstrukturen, die zum Speichern der Simulationsdaten verwendet werden. Wie bereits in Kapitel 6.2.1 beschrieben bestehen für die Datenhaltung zwei Datenstrukturen im simModul.

Im simAgent dient das Array (siehe Kapitel 6.2.1, Abbildung 6.1) für das Zwischenspeichern der Simulationsdatensätze. Die Größe eines Elements beträgt acht Byte und die Implementierung ermöglicht eine maximale Array-Länge von 10.000 Elementen. Daraus errechnet sich ein maximaler dynamischer Speicherbedarf von 80 kB. Zudem wird im DioSim ein Array mit der Elementgröße von vier Byte und einer Anzahl von 254 Elementen bereitgehalten. Dieses benötigt einen Speicherplatz von 1.016 Byte bzw. 1,02 kB im Steuergerät.

In Summe ergibt sich ein dynamischer Speicherbedarf von 81,02 kB für diese spezifische Implementierung. Eine Reduktion z.B. der Array-Längen ist durchaus möglich und muss je ECU entschieden werden.

Statische Worst Case Execution Time Analyse

Die WCET steht für die maximal mögliche Laufzeit eines Programms oder Programmteils. Für die Analyse wird ein Szenario gewählt, welches eine maximale Auslastung erreicht. Die WCET ist hier von zentraler Bedeutung, da die Summe der einzelnen REs die minimale Zykluszeit der SWC nicht unterschreiten darf. Hierfür wird ein *Worst Case* Szenario erstellt, in dem alle REs die maximale Laufzeit benötigen. Dieses Verfahren ist eine statische Analyse und basiert auf der abstrakten Interpretation von Laufzeiten ohne eine Ausführung dieser.

Für die Ermittlung der WCET sind drei Schritte notwendig. Die Programmanalyse, die Low-Level-Analyse und die eigentliche Berechnung. [Sta04]

Programmflussanalyse – analysiert den Kontrollfluss eines Programms, bzw. dessen Quellcode. Durch eine semantische Analyse der einzelnen Anweisungen des Verhaltens des Programms können die einzelnen Schritte interpretiert werden. Die Programmflussanalyse ist erforderlich, da es durch das Scheduling der fünf zyklischen REs zu Verzögerungen im Programmablauf führen kann oder Informationen gar verloren gehen können.

1 `sym_SimAgent_Starter_cyclic` Triggert bei synchroner Ausführung die Simulation.

2 `sym_SimAgent_UpdateCDD_cyclic` Fügt dem DioSim Modul über das simGW Simulationsdatensätze hinzu.

3 `sym_SimGW_cyclic` ohne Funktion

4 `HwIoAb_DIO_ReadZBERotation_cyclic` Liest die Lichtschrankenzustände aus

5 `HwIoAb_DIO_ReadZBEValues_rbl` Liest die Zustände der Hardware-Kanäle und gibt sie an `SwcZBE_cyclic` zurück.

Eine Programmflussanalyse ergibt folgende Programmflüsse der REs 1-5:

```
1 sym_SimAgent_Starter_cyclic
  ↪ startSimulation
    ↪ Rte_Call_CSControlSimulation_CSOpEnableSimulation
      ↪ sym_SimGw_Switch_Simulation
        ↪ DioSim_EnableSimulation
2 sym_SimAgent_UpdateCDD_cyclic
  ↪ Rte_Call_CSChangeSimEntryDio_CSOpAddSimDio
    ↪ sym_SimGw_AddSim
      ↪ DioSim_UpdateSim
        ↪ DioSim_GetIndex
        ↪ DioSim_ShiftRight
  ↪ Rte_Call_CSChangeSimEntryDio_CSOpRemoveSimDio
    ↪ sym_SimGw_RemoveSim
      ↪ DioSim_RemoveSim
        ↪ DioSim_GetIndex
        ↪ DioSim_ShiftLeft
3 sym_SimGW_cyclic
4 HwIoAb_DIO_ReadZBERotation_cyclic
  ↪ Dio_ReadChannel
    ↪ DioSim_GetSimState
    ↪ DioSim_GetSim
      ↪ DioSim_GetIndex
    ↪ DioOrigin_ReadChannel
5 HwIoAb_DIO_ReadZBEValues_rbl
  ↪ Dio_ReadChannel
    ↪ DioSim_GetSimState
    ↪ DioSim_GetSim
      ↪ DioSim_GetIndex
    ↪ DioOrigin_ReadChannel
```

Die Programmflüsse berücksichtigen nur die simModul Anteile, da nach dem Aufruf von `HwIoAb_DIO_ReadZBERotation_cyclic` bzw. `HwIoAb_DIO_ReadZBEValues_cyclic` der originale Programmcode der BSW ausgeführt wird.

Low-Level-Analyse – verwendet ein Instruktionsschema der Programmiersprache C basierend auf [Köb15a, Dit16]. Dieses Schema (Tabelle 6.4) impliziert bzw. basiert auf den folgenden Annahmen.

Eine atomare Anweisung (<ATOM>) kann nicht weiter zerlegt werden und ergibt eine Instruktion. Eine Boolsche Instruktion (<BOOL>), die nicht weiter zerlegt werden kann, ergibt ebenfalls nur eine Instruktion. Ein (<IF>) Konstrukt besteht aus drei Instruktionen mit je einer (<BOOL>)-Anweisung. Für eine Funktion (<FUN>) werden zehn Instruktionen je Eintritt und Austritt in bzw. aus der Funktion erstellt. Um die Register auf dem Stack zu sichern, werden je zwei weitere Instruktionen für das Einfügen und Löschen erstellt. Eine Schleife (<WHILE>) enthält ein (<IF>) als Schleifeninvariante. Diese (<IF>) wird vor jedem Schleifendurchlauf geprüft und gegebenenfalls der Schleifenrumpf (<RUMPF>) ausgeführt. Sind alle Schleifendurchläufe beendet, ist die Schleifeninvariante false, und somit kommt noch eine (<IF>)-Anweisung hinzu.

Anweisung	Anzahl an Instruktionen
<ATOM>	1
<BOOL>	1
<IF>	3 ∗ <BOOL>
<FUN>	20 + (<ATOM> ∗ 2)
<RUMPF>	<ATOM> (+ <RUMPF>) | <IF> (+ <RUMPF>)
<WHILE>	(<IF> + <RUMPF>) + Schleifendurchläufe + <IF>

Tab. 6.4.: Instruktionsschema WCET

Die Auflistung 6.5 zeigt, basierend auf der vorangegangenen Programmflussanalyse, die Anzahl an Instruktionen, die für die Funktionen nötig sind. Das Instruktionsschema Tabelle 6.4 dient hier als Berechnungsgrundlage. Log-Ausgaben wurden von der Berechnung ausgenommen.

Die Ermittlung der WCET – verwendet die zuvor ermittelten Daten der Programmflussanalyse und der Low-Level-Analyse. Hierfür gibt es drei grundsätzliche Ansätze für die Berechnung: baumbasiert, Implicit Path Enumeration Technique (IPET) und pfadbasiert.

Für die Ermittlung der WCET müssen die vier in Tabelle 6.5 angegebenen Berechnungen für die Anzahl an Instruktionen der Funktionen `sym_SimAgent_UpdateCDD_cyclic`, `DioSim_GetIndex`, `DioSim_ShiftRight` und `DioSim_ShiftLeft` aufgelöst werden. Diese sind von dem jeweilig gewählten Szenario abhängig. Für die maximale Länge an Datenstrukturen kann ein Wertebereich von $]0; 10000]$ angenommen werden. Für die Komponente ZBE ist ein Wert von 14 sinnvoll, da aufgrund der Anzahl an Kanälen $c = 14$ eine maximale Anzahl von 14 gleichzeitig zu simulierenden Werten möglich ist. Folglich muss $n \leq c$ sein. Somit ergibt sich eine Anzahl von Instruktionen für `sym_SimAgent_UpdateCDD_cyclic` $47 * 14 + 14 = 672$ und für `DioSim_GetIndex` $= 10 * 14 + 6 = 146$. `DioSim_ShiftRight` verschiebt alle Elemente in dem Array nach rechts und wird nur aufgerufen, wenn noch nicht die gesamte Anzahl an Kanälen simuliert wird. Daraus ergibt sich für $k = m - 1$, da mindestens ein Kanal bereits simuliert wurde. In Zahlen heißt das für `DioSim_ShiftRight` eine Anzahl von Instruktionen von $24 + k + 10 = 24 * (m-1) + 10 = 24 * (14 - 1) + 10 = 322$. `DioSim_ShiftLeft` kann im Gegensatz zu `DioSim_ShiftRight` auch bei einem vollen Array geschehen und ergibt hiermit eine Instruktionsanzahl von $15 * l + 5 = 15 * 14 + 5 = 215$.

Funktion	Instruktionen
sym_SimAgent_Starter_cyclic	23
sym_SimAgent_UpdateCDD_cyclic	$47 * n + 14$
startSimulation	26
Rte_Call_CSControlSimulation_CSOpEnableSimulation	36
Rte_Call_CSChangeSimEntryDio_CSOpAddSimDio	38
Rte_Call_CSChangeSimEntryDio_CSOpRemoveSimDio	36
sym_SimGW_cyclic	0
sym_SimGw_Switch_Simulation	28
sym_SimGw_AddSim	24
sym_SimGw_RemoveSim	22
DioSim_GetSimState	1
DioSim_GetSim	32
DioSim_GetIndex	$10 * c + 6$
DioSim_EnableSimulation	1
DioSim_UpdateSim	62
DioSim_RemoveSim	54
DioSim_ShiftRight	$24 * k + 10$
DioSim_ShiftLeft	$15 * 1 + 5$
DioOrigin_ReadChannel	53

Tab. 6.5.: Low-Level-Analyse - Anzahl Instruktionen je Funktion

Runnable Enteties	Instruktionen
sym_SimAgent_Starter_cyclic	$23 + 26 + 36 + 28 + 1 = 144$
sym_SimAgent_UpdateCDD_cyclic	$672 + 14 * (38 + 24 + 62 + 146 + 322) = 8960$
sym_SimGw_cyclic	0
HwIoAb_DIO_ReadZBERotation_cyclic	$2 * (53 + 1 + 32 + 146) = 464$
SwcZBE_cyclic	$12 * (53 + 1 + 32 + 146) = 2784$

Tab. 6.6.: Low-Level-Analyse - Anzahl Instruktionen im Worst Case

Durch das festgelegte Scheduling der zyklischen Tasks steht die Reihenfolge dieser fest. Die Tasks der REs sym_SimAgent_Starter_cyclic und SwcZBE_cyclic weisen einen 60 ms Taskzyklus auf, alle anderen einen 10 ms Taskzyklus. Für die Betrachtung der WCET wird davon ausgegangen, dass alle fünf REs ausgeführt werden, um den Worst Case zu treffen. Die Tabelle 6.7 zeigt die Summe der Instruktionen für die fünf REs. Zu berücksichtigen ist hierbei, dass bei HwIoAb_DIO_ReadZBERotation_cyclic und bei SwcZBE_cyclic nur die durch die Integration des simModuls veränderten Stellen gezählt werden. Alle anderen Codesegmente sind Teil des originalen Programmaufrufs DioOrigin_ReadChannel und finden daher keine Berücksichtigung.

Für die Funktion `sym_SimAgent_UpdateCDD_cyclic` wird nur der relevante Programmcode des untergeordneten Aufrufs `Rte_Call_CSChangeSimEntryDio_CSOpAddSimDio` gewertet, da sich diese Funktion und die dazu inverse Funktion `Rte_Call_CSChangeSimEntryDio_CSOpRemoveSimDio` gegenseitig ausschließen. Die längere Ausführungszeit weist die Funktion `Rte_Call_CSChangeSimEntryDio_CSOpAddSimDio` auf und ist daher bei der Berechnung der WCET berücksichtigt.

Zuzüglich zu den bereits genannten Instruktionen kommen, wie in Auflistung 6.7 dargestellt, Instruktionen zur Verwaltung der RTE hinzu.

Runnable Enteties	Instruktionen in der RTE
`sym_SimAgent_Starter_cyclic`	72
`sym_SimAgent_UpdateCDD_cyclic`	66
`sym_SimGw_cyclic`	66

Tab. 6.7.: Low-Level-Analyse - Anzahl Instr. zur Verwaltung der REs in der RTE

Die Summe der Instruktionen aus Tabelle 6.6 und 6.7 ergibt 12.526 im Worst Case. Unter der Annahme, dass jeder Kanal durch die Application-SWC in einem Betrachtungszyklus genau einmal gelesen wird, kann eine Funktion $I(k)$ angegeben werden, mit der sich die Anzahl der Instruktionen in Abhängigkeit von der Anzahl der zu simulierenden Kanäle c berechnen lässt.

Diese Funktion lautet wie folgt:

$$I(k) = \underbrace{114}_{\text{SimAgent\_Starter}} + \overbrace{34k^2 + 163k + 14}^{\text{SimAgent\_UpdateCDD}} + \underbrace{0}_{\text{SimGw}} + \overbrace{10k^2 + 92k}^{\text{SUT}} + \underbrace{204}_{\text{RTE}} = 44k^2 + 255k + 332$$

Der absolute Wert der maximal auftretenden Verzögerung ergibt sich folglich unter Bezug auf die Hardware. Projiziert auf die originale HW, verfügt die ZBE über einen 80 MHz Prozessor mit einer Zykluszeit von 12,5 ns. Hieraus ergibt sich aus der Anzahl der Instruktionen und der Kanäle, multipliziert mit der Zykluszeit, eine Verzögerung bzw. Ausführungsdauer von maximal 156.575 ns. Bezogen auf die minimale Zykluszeit der implementierten Tasks von 10 ms, ergibt das einen Anteil von 1,57 %.

Die sich ergebende maximale Verzögerung $D_{max}(k, f_{mcu})$ kann, wie folgt, in Abhängigkeit der Anzahl zu simulierender Kanäle k und der jeweils entsprechenden CPU-Taktfrequenz f_{mcu}, angegeben werden.

$$D_{max}(k, f_{mcu}) = I(k) \cdot \frac{1}{f_{mcu}}$$

Die Ergebnisse der WCET zeigen unter anderem, dass bei dem in Kapitel 4.4.10 definierten Scheduling keine Konflikte auftreten und die Ausführung der RE des TE_SwcZBE_cyclic auch im worst case in den definierten Zeitkriterien möglich ist.

AUTOSAR Konformität

Wie bereits in Kapitel 4.4.3 beschrieben, wird der ursprüngliche AUTOSAR-Service von `Dio_ReadChannel` [AUTc] in `DioOrigin_ReadChannel` umbenannt und in dem Aufruf des `Dio_ReadChannel` eine Abfrage nach einer aktiven Simulation hinzugefügt. Dies ist die eigentliche Abweichung vom verwendeten AUTOSAR-Standard 4.0.x, entspricht aber der AUTOSAR-Philosophie.

Die entstandene Architektur des simModuls ist konform zu dem AUTOSAR Standard entwickelt worden. Besondere Beachtung wurde den Modulen `simAgent`, `simGW` geschenkt und entsprechend dem AUTOSAR Standard entwickelt. Ausnahme ist das `DioSim`-Modul, das, wie beschrieben, den Treiber mit einer zusätzlichen Schnittstelle erweitert. Auch die in 2.4.3 beschriebene Layer Interaction Matrix wurde entsprechend dem Standard beachtet.

Die Kommunikation zwischen dem simMaster und simAgent wird, wie in der AUTOSAR-Spezifikation [AUTd, AUTe, AUTi] beschrieben, über den Kommunikationsstack und die RTE abgewickelt. SimAgent und simGW kommunizieren standardkonform über die RTE. Auch die Implementierung des simGW als Complex Driver ist dem Standard entsprechend die optimale Lösung, da dieser die Nutzung des Complex Driver Stacks für die Umsetzung neuer oder nicht im Standard enthaltenen Funktionen vorgesehen ist.

Abweichend vom AUTOSAR-Standard sind das `DioSim`-Modul und die Erweiterung des `Dio`, welches das `DioSim`-Modul integriert. Trotzdem wurde darauf geachtet, die Realisierung dieser zusätzlichen Schnittstelle bzw. des Services so zu gestalten, dass es der AUTOSAR-Philosophie entspricht. Dies hat den Grund einer möglichen Aufnahme in einen zukünftigen AUTOSAR-Standard vorzugreifen (Kapitel 7.2).

6.2.3. Gesamtsystem/verteiltes System

Für die Validierung der Simulationsmethodik werden im Gesamtsystem Analysen, angelehnt an die *A-posteriori-Analyse* d.h. während Laufzeit, angewendet. Hierfür werden entsprechend die notwendigen Daten ermittelt, die eine Bewertung der Methodik in den verschiedenen Bereichen Kommunikation, Reproduzierbarkeit und Einsatzmöglichkeiten ermöglichen.

Für alle Analysen wird im Folgenden der Use Case – Lichtfunktionen *Aktivierung ambientes Innenlicht herangezogen*. Hierdurch sind die Analysen vergleichbar und erlauben Schlussfolgerungen. Der Use Case, vorgestellt in Kapitel 6.2, zeigt eine Kundeneingabe, die eine

Aktivierung bzw. Deaktivierung des ambienten Innenlichts bewirkt. Der Kunde navigiert hier mit Hilfe der ZBE durch die Menüstruktur in der HU und setzt im Zielmenü die entsprechende Option für die Lichtfunktion.

Für die Analyse auf Gesamtsystemebene wird eine Domänenspezifische Architektur, wie auf Abbildung 6.6 dargestellt ist, verwendet. Diese Architektur beinhaltet alle für das Infotainment relevanten ECUs. Die statischen Informationen, die das Infotainmentsystem (HU und BDC) benötigt, werden über eine spezielle Restbus-Simulation bereitgestellt. Der Arduino wurde im Rahmen des Testaufbaus verwendet, um den Menschen bei der wiederholten Eingabe von Systemeingaben abzulösen und durch diesen eine reproduzierbarere Input Simulation der ZBE zu erreichen.

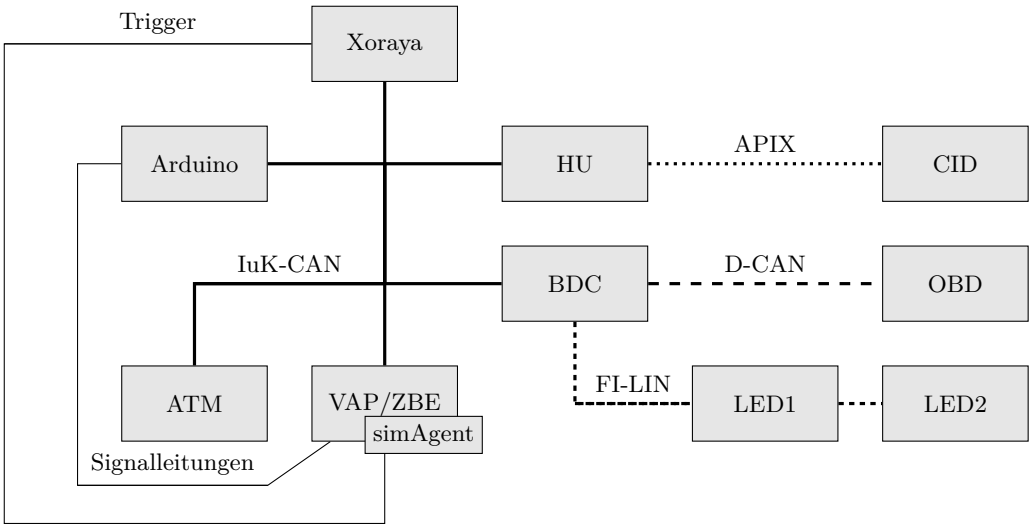


Abb. 6.6.: Use Case - Simulationsarchitektur

Die ZBE ermöglicht die Eingabe von sieben Funktionstasten, einem Drehgeber, einer Auslenkung in vier Richtungen (Nord, Ost, Süd und West) sowie einen Druck auf den Drehgeber. Technisch ist die Dreherkennung über eine Gabellichtschranke realisiert und bedarf der Auswertung von zwei digitalen Inputs. Die Drehrichtungserkennung läuft dann über die Erkennung, welche Lichtschranke als Erstes ausgelöst wird. LSR steht für Lichtschranke rechts, LSL für Lichtschranke links.

Folgende Eingabe aus dem Startmenü muss hierfür erfolgen: Vier Rotationen rechts - Druck auf den Drehgeber - Druck auf den Drehgeber - eine Rotation rechts - drei Betätigungen des Drehgebers. Die Kundeneingabe an der ZBE lässt sich in drei Signalverläufe zerlegen, die auf Abbildung 6.7 dargestellt sind.

Konstruktionsbedingt ist der Drehgeber rastend ausgeführt, d.h. die Drehung um eine Raste in eine Drehrichtung bewirkt das Unterbrechen bzw. Schließen der Lichtschranke. Somit ist ein Kippen des Signalpegelwerts äquivalent zu einer Drehung um eine Raste. Im Falle der

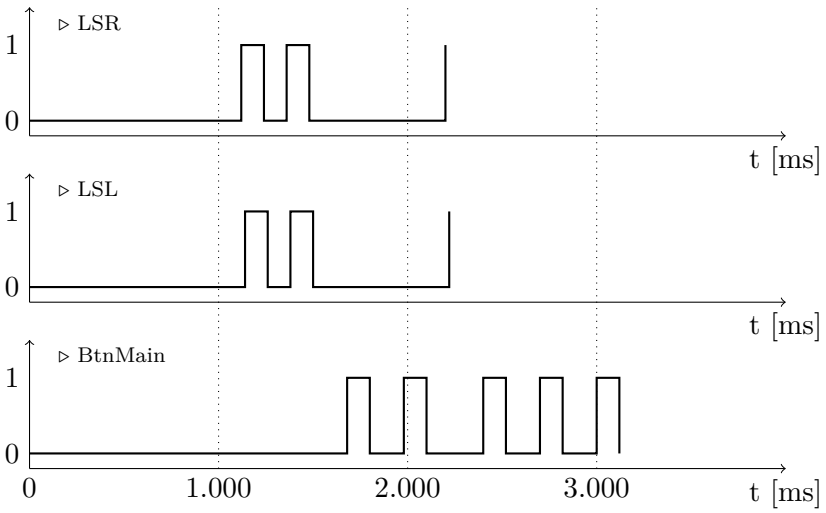


Abb. 6.7.: Use Case - Simulation Data

Abbildung 6.7 wird die LSR immer 20 ms vor der LSL ausgelöst. Hier wurde im Rahmen der Fallstudien eine konstante Drehgeschwindigkeit festgelegt. Hinzu kommt, dass ein Druck auf den Drehgeber einheitlich mit 120 ms festgelegt wurde. Durch diese Signalfolgen an den jeweiligen HW-I/Os werden im idealen Fall exakt 15 Nachrichten von der ZBE auf den BUS gelegt.

Um die Kundeneingabe für die Analyse möglichst einfach zu gestalten, wurde die Application-SWC simAgent, in identischer Form, auf dem Arduino implementiert und eine dem AUTOSAR ähnliche Abstraktionsschicht erstellt. Die Abstraktionsschicht mappt die Simulationdaten auf die HW-Output-Ports des Arduino, die mit dem HW-Eingang der VAP bzw. ZBE verbunden sind. Somit kann über einen Arduino mit der identischen SWC simAgent auch eine Kundeneingabe von außen an das System angelegt werden. Vorteil dieses Vorgehens ist, dass die identischen Testfälle aus dem simMaster verwendet werden können und damit eine Vergleichbarkeit von „manuell“ eingegebenen Systeminputs mit über die Simulationsmethodik eingespielten erreicht wird.

Kommunikationsprotokoll

Die Zustandsautomaten aus Kapitel 3.3.4 sowie die Sequenzdiagramme 5.7 und 5.9 zeigten bereits, dass die Datenübertragung und die Ausführung der Simulation entkoppelt sind. Dieser Ansatz bringt hier gegenüber den Methoden CCP (Kapitel 3.4.1) und XCP (Kapitel 3.4.2) den Vorteil der Buslast wie bei normaler Funktionsausführung. Da hier nach Spezifikation keine zusätzliche Kommunikation während der Simulation ausgeführt wird, wurden keine expliziten Messungen durchgeführt.

Bei den Messungen zur Reproduzierbarkeit zeigte die Erhöhung der Buslast in Bezug auf die Reproduzierbarkeit, welchen Einfluss diese hat. Teilweise waren Messungen nicht mehr möglich, da die Controller die Menge der Daten nicht mehr verarbeiten konnten. Dadurch war aber auch keine Kundenfunktion mehr darstellbar. Alle anderen Messungen wurden je Testebene unter realen Buslasten aufgenommen und zeigten das die entwickelte Methode diese nicht beeinflusst.

Seitens Kommunikationsprotokoll ergeben sich noch zu lösende Aufgaben bezüglich BUS-Systemen ohne dynamischen Kommunikationsanteil. D.h., bei BUS-Systemen wie z.B. LIN muss vor Implementierung bereits ein Übertragungsslot in der Konfigurationstabelle bedacht werden. Alternativ muss für die Übertagung der Simulationsdaten zum simAgent der BUS in eine andere Kommunikationstabelle umgeschaltet werden. Solch ein Umschalten ist technisch möglich, muss aber in den Komponenten zusätzlich implementiert werden.

Reproduzierbarkeit

Die Reproduzierbarkeit eines Systemverhaltens (Soll- und Fehlverhalten) ist eine der Kernherausforderungen beim Testen und ist mit den bereits genannten aktuellen Methoden CCP und XCP nahezu nicht möglich, vor allem fällt dies auf, wenn die SWCs genauer untersucht werden.

Für die Bewertung der Reproduzierbarkeit einer Wirkkette sind folgende Messwerte erforderlich: der Zeitpunkt, an dem das SUT eine Kanalwertänderung registriert, der Zeitpunkt, an dem die BSW meldet, dass eine Nachricht versendet wurde, sowie der Zeitpunkt, an dem eine Nachricht auf dem BUS registriert wurde.

Der Zeitstempel der Kanalwertänderung T_k, wird über eine SUT interne Log-Ausgaben ermittelt. Diese Log-Ausgaben bewirken keine Veränderung des Sourcecodes, sondern sind fest in der BSW integriert. Der zweite Zeitstempel T_v, wann eine Nachricht versendet wird, ist seitens der BSW bereits umgesetzt und wird ebenfalls in die Log-Ausgabe umgeleitet. Sowie ein Zeitstempel T_e, wann eine Nachricht für die anderen BUS Teilnehmer auf dem BUS liegt, wird durch einen Datenlogger (Modell Xoraya) ermittelt. Über eine externe Triggerleitung wird eine spätere Synchronisation der beiden Log-Files ermöglicht. Grund für diese Art der Lösung war, dass Zeitsynchronisation über Network Time Protocol (NTP) oder Precision Time Protocol (PTP) mit dem verwendeten Logger nicht möglich war.

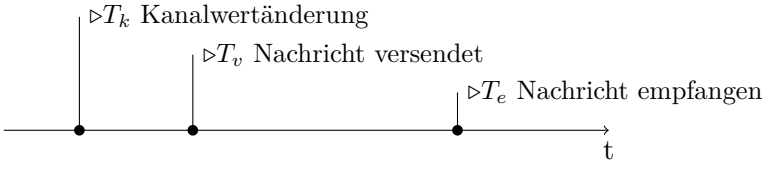


Abb. 6.8.: Reproduzierbarkeit - (Sollverhalten) Zeitstempel

Die Reproduzierbarkeit wird anhand der Differenz T_v - T_k also der Rechenzeit für das SUT und der BSW, sowie der Übertragungszeit auf dem BUS T_e - T_v gemessen.

Anmerkung: Grundlegend kann eine Aussage über Reproduzierbarkeit von Funktionsreaktionen vorweggenommen werden. In dynamischen Systemen kann eine Reproduzierbarkeit mit dem Fokus auf exaktes Timing nur ECU intern erreicht werden, da je nach verwendetem Kommunikationsmedium der ECUs aufgrund Technologieeinsatz untereinander ein Timing nicht mehr vorhersehbar ist. D.h., z.B. bei einer Kommunikation über CAN, der auf einem Broadcast basiert, kann sich das Versenden einer Nachricht verzögern, da die Kommunikationsleitung bereits durch einen anderen CAN Teilnehmer belegt ist. Zur weiteren Untersuchung wurde der Faktor dynamisches System in Bezug auf die Registrierung der Nachricht am Logger minimiert, indem alle dynamischen Anteile deaktiviert wurden.

Reproduzierbarkeit innerhalb des Testfalls

Betrachtet wird das Zeitverhalten innerhalb des Testfalls, gemessen von der ersten Reaktion auf eine Kanalwertänderung T_k zu der erwarteten bzw. errechneten Kanalwertänderung. Der Boxplot auf Abbildung 6.9 zeigt, dass die Testabläufe innerhalb des Betrachtungsraums Testfall einen hohen Grad an Reproduzierbarkeit aufweisen.

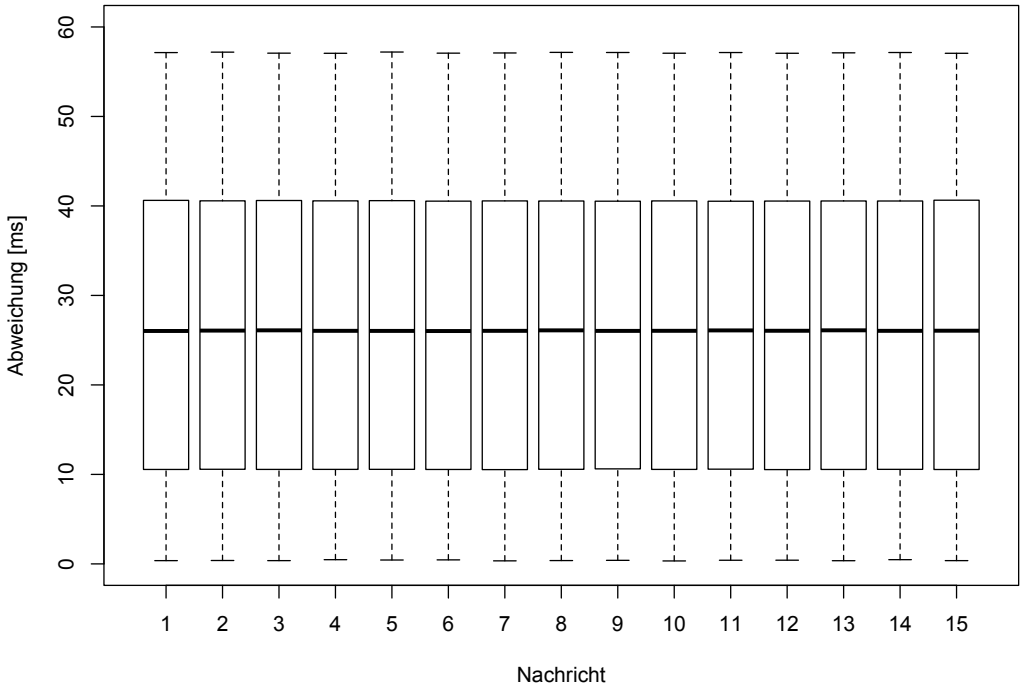


Abb. 6.9.: Timing-Verhalten innerhalb eines Testfalls

Der Boxplot 6.9 zeigt die Verteilung der Zeitstempel T_k für alle 15 Signalwertänderungen über 100 Messungen.

Reproduzierbarkeit der Testfälle

Für die Bewertung der Reproduzierbarkeit wird das Zeitverhalten ab Testbeginn T_k bzw. Ausführungstrigger durch den simAgent betrachtet. Die Abbildung 6.10 zeigt die Streuung des Eintreffens der Botschaften T_e auf dem BUS, registriert durch den Logger. Die Abweichungen sind auf Unstimmigkeiten in der Taskausführung zurückzuführen.

Eine statistische Auswertung der Daten ergibt für die mittlere Abweichung 26,5 ms bei einer Standardabweichung von 17,7 ms. Die Messreihe über die reale Stimulation der HW-Eingänge unter Zuhilfenahme des Arduino ergibt eine mittlere Abweichung von 20,8 ms bei einer Standardabweichung von 16,8 ms.

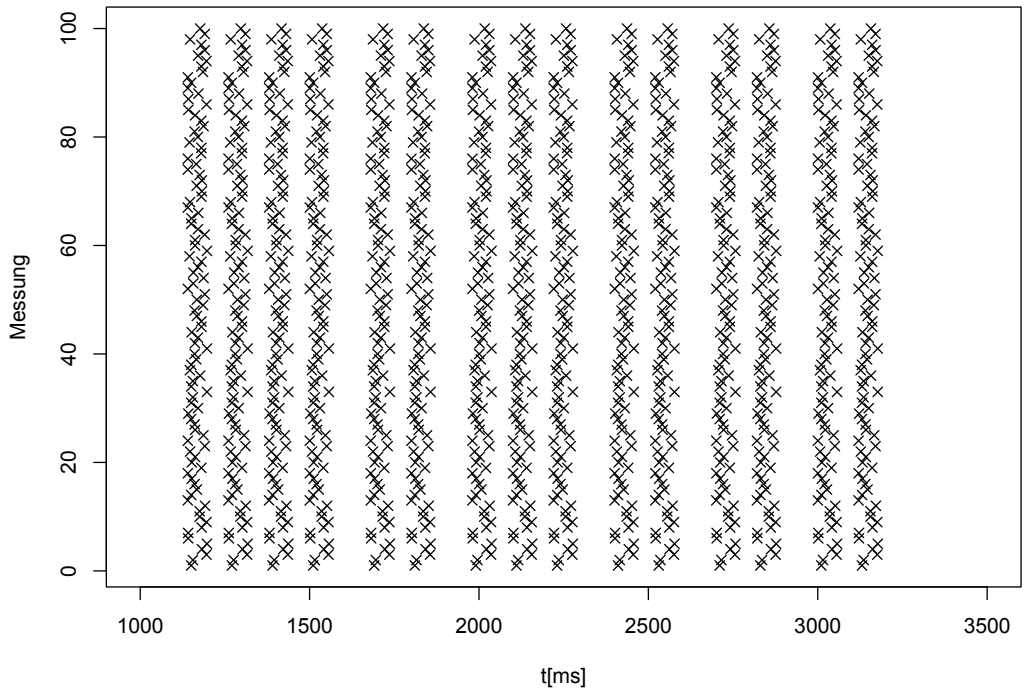


Abb. 6.10.: Timing-Verhalten noSync

Aus dem Grund, dass die BSW-Treiber Schicht mit einem 10 ms Taskzyklus läuft und die SWC ZBE mit einem 60 ms Zyklus, kann sich das Erkennen der Kanalwertänderung (Zeitpunkt T_k) in der Applikation-SWC stark verzögern. Auf Abbildung 6.11 ist dargestellt, wie das simulierte Signal von 100 ms (Abbildung 6.11 unten) in den verschiedenen Abtastungen durch die Applikation-SWC interpretiert werden kann. Der Fall (a) zeigt das Signal in der

Applikation-SWC, wenn diese 10 ms vor dem simulierten Signal abtastet. Hierdurch verschiebt und verkürzt sich im Fall (a) das simulierte Signal auf 60 ms. Im Fall (b) verlängert dieses Abtastproblem die Interpretation des simulierten Signals auf 120 ms.

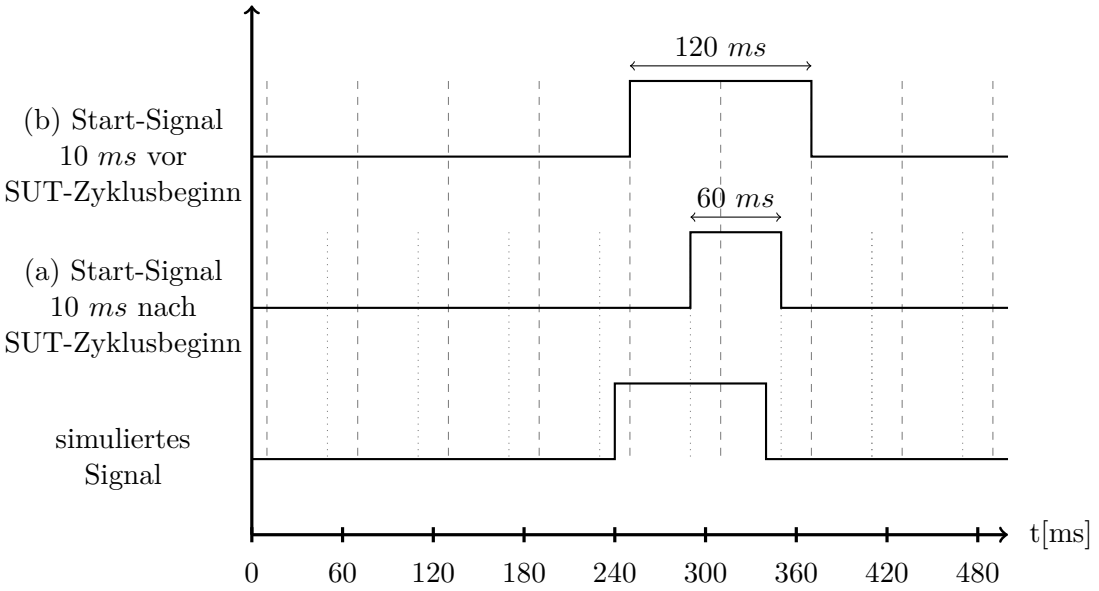


Abb. 6.11.: Verschiedene Interpretation eines 100 ms Signals

Vergleicht man die Verteilung der Signale im simulierten Fall mit dem realen Kundennutzen, so ergibt sich ein identisches Verhalten. D.h., diese Art der Simulation stellt ein Kundenverhalten in ihrer besten Weise dar. Betrachtet man die Reproduzierbarkeit wiederholter Testfälle, kann die Aussage getroffen werden, dass diese keineswegs reproduzierbar sind.

Um die Testfälle exakt reproduzierbar zu machen, wird das Scheduling bzw. *Sync* aktiviert, d.h. der simAgent kennt die Zyklen der Applikation-SWC. Über die Nachricht `SIM_Run_Rx` wird im Feld `StopOrGo` das Signal `GoSync` an den simAgent versendet. Dies führt dazu, dass der simAgent auf den nächsten Taskzyklus der Applikation-SWC wartet und ab Erkennen des Zyklus seine Ausführung des Testfalls beginnt. In dem Fall *Sync* ist das Testdesign entscheidend, wenn ein reproduzierbarer Testfall entstehen soll.

Abbildung 6.12 zeigt die gleichen Testfälle mit der *Sync* Option. Hier werden die Testfälle immer mit der exakt gleichen Verzögerung auf den Taskzyklus der Applikation-SWC ausgeführt. Eine statistische Auswertung der Messreihen ergibt für die mittlere Abweichung einen Wert von 0,4 ms bei einer Standardabweichung von 0,1 ms.

Das Sequenzdiagramm 6.13 zeigt nochmal anschaulich wie der simAgent auf den Zyklus der Applikation-SWC wartet, bis er mit der Ausführung der Simulation beginnt. Der Task `sym_SimAgent_Starter_cyclic` und `SwcZBE_cyclic` werden beide mit 60 ms ausgeführt. In der Schedulingtabelle sind diese Tasks entsprechend Tabelle 6.8 eingeordnet.

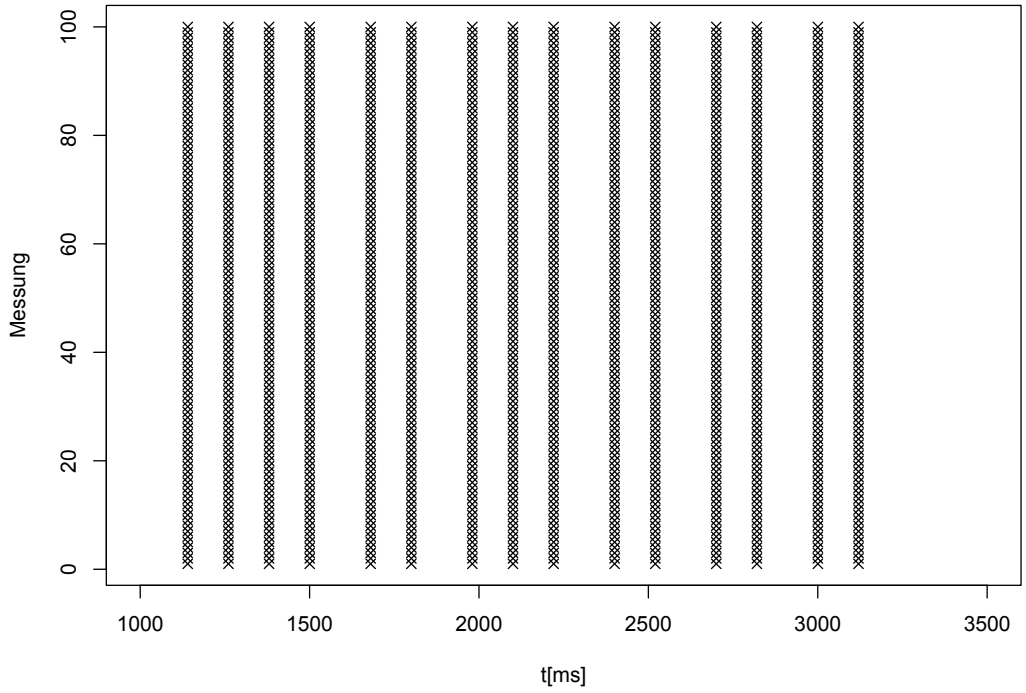


Abb. 6.12.: Timing-Verhalten Sync

Die Validierung zeigt die zwei grundsätzlich verschiedenen Anwendungsszenarien der Methodik. Einerseits das *Nachstellen eines realen Kundenverhaltens*, bei dem das Gesamtsystem als Blackbox betrachtet und die Systemreaktion auf die Kundeneingabe ungeachtet des Scheduling bewertet wird. Andererseits bietet diese Methodik für das *Reproduzieren eines Systemverhaltens* die Option, das Scheduling der Applikation-SWC zu berücksichtigen und auf das höchste Maß die Systemeingabe wiederholt an das SUT anzulegen.

Weitere Fallstudien zum Timing-Verhalten nach gleichem Schema sind in Anhang 6.4 zu finden. Durchgeführt wurden die Use Cases: Geschwindigkeitswarnung, Sofort Laden, Günstig Laden, Standklima aktivieren.

Priorität	Task
1.	`sym_SimAgent_Starter_cyclic`
2.	`sym_SimAgent_UpdateCDD_cyclic`
3.	`sym_SimGw_cyclic`
4.	`HwIoAb_DIO_ReadZBERotation_cyclic`
5.	`SwcZBE_cyclic`

Tab. 6.8.: Schedulingtabelle der Tasks

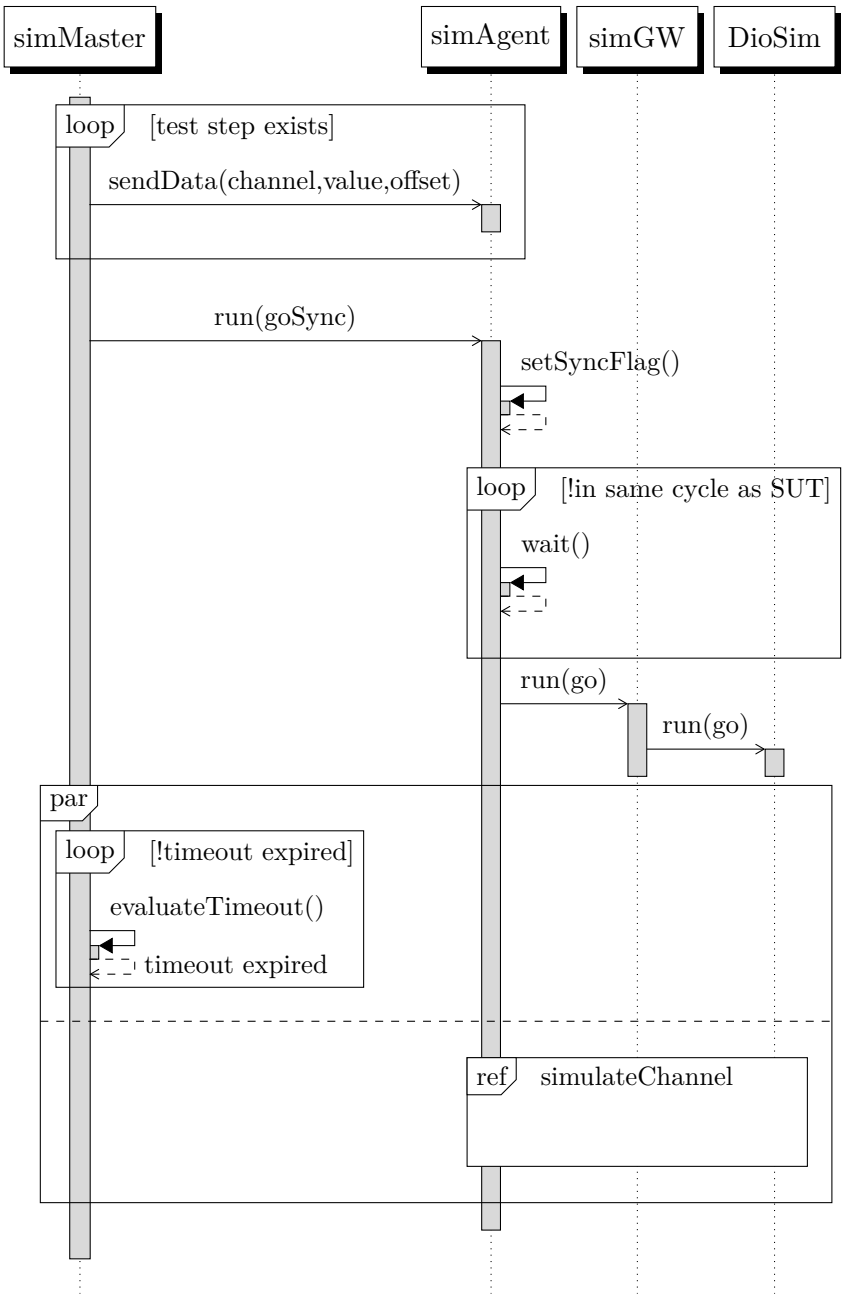


Abb. 6.13.: Sequenzdiagramm - Sync Simulation

Anwendungsszenarien

Die Entwicklung dieser Methode wurde in erster Linie für die Anwendung in der Produktentwicklung der Automotive Branche ausgelegt. Im Speziellen galt der Anspruch ein software-basiertes Testing in Gesamtfahrzeugen zu ermöglichen und möglichst viele verschiedene Kundenverhalten einspeisen zu können. Berücksichtigung bei der Entwicklung der Methodik fanden auch Funktionen, die eine Backendkommunikation erforderten, wurden aber nicht explizit untersucht. In Verbindung mit den etablierten Methoden der Testanalyse und Auswertung ergeben sich aufgrund der Verwendung eines modellbasierten Ansatzes auch weitere Anwendungsszenarien im Rahmen der Produkt- und Softwareentwicklung.

Diese in Kapitel Kapiteln 3 entwickelte Methode kann daher auch im Bereich SIL und HIL sinnvolle Anwendung finden. Für die Anwendung im SIL wird über die Kommunikation zum simAgent z.B. über die simulierte Schnittstelle eingespeist. Dabei kann mit Hilfe der neuen Methode die entwickelte Software in ihrer jeweils aktuellen Ausprägung auf ihr logisches Verhalten untersucht werden.

Eine grundlegende Anwendung findet sich im Bereich HIL, weil die dargestellte Methode hier ihre Potentiale auf der Zielkomponente (dem realen Steuergerät) zeigen kann. Besonders unter den Rahmenbedingungen der realen HW in Verbindung mit Steuergerätekommunikation stellt die Methode im Bereich der Reproduzierbarkeit neue Möglichkeiten für das Testing.

Im Bereich des MIL bringt dieser Ansatz keine Vorteile gegenüber konventioneller Simulation einer Umgebung, da MIL-Methoden grundlegend das Ziel haben die Funktionslogik einzelner Modelle nachzuweisen.

Bezogen auf die den PEP bzw. das V-Modell findet sich eine Einsatzmöglichkeit vom Komponententest bis zum Akzeptanztest. Der entscheidende Vorteil der neuen Methode ist, dass sich die software-basierte Methode bzw. das Modul (simAgent) für die Simulation in der Komponente über die verschiedenen Entwicklungsphasen nicht ändert und eine Möglichkeit schafft, diese Tests in den verschiedenen Testebenen des V-Modells zu wiederholen, ohne dass diese einer Anpassung bedürfen.

Anwendungsbeispiele finden sich folglich in allen Varianten von Systemen. Eine Anpassung der Testfälle erfolgt durch den modellbasierten Ansatz, in dem der entsprechende Teil des Testmodells freigeschnitten wird. Bei Teilsystemen, also Gruppen von embedded Systems, ist der Aufwand in der Praxis relativ gering zu bewerten, da in der Regel domänenspezifisch getestet wird. Entscheidend ist letztendlich in den verschiedenen Anwendungsszenarien, dass das Verhalten aus dem Testmodell identisch ist und auch identisch bzw. reproduzierbar in das SUT eingespielt werden kann.

6.3. Ergebnisse

Dieses Kapitel fasst die Ergebnisse aus den vorangegangen Abschnitten des Kapitel 6 zusammen und setzt sie in Bezug zu den Erkenntnissen aus den Kapiteln 3 - Methodischer Ansatz, Kapitel 4.4 - Implementierung und Kapitel 5 - Anwendung. Die nachfolgenden Abschnitte nehmen Bezug auf die ursprünglich in Kapitel 1.3 gesetzten Ziele und stellen klar in welchen Umfängen diese erreicht wurden. Kapitel 6.4 zeigt im Anschluss, dass sich die vorangegangenen Ergebnisse bzw. Systemverhalten in weiteren Fallstudien in gleicher Form nachweisen lassen.

Das Ergebnis dieser Arbeit ist eine modellbasierte Methodik, die eine Simulation von Systemeingaben bzw. Kontextverhalten im verteilten System ermöglicht. Unter Verwendung eines software-basierten Ansatzes konnten der Implementierungsaufwand und der Aufwand für die Anwendung der Methode stark reduziert werden. Mit der Hilfe modellbasierter Entwicklungsmethoden, die an die modellbasierten Testmethoden gekoppelt werden, kann ein stimmiges Vorgehen dargestellt werden, um Systemverhalten software-basiert automatisiert bewerten zu können.

6.3.1. Bestätigung modellbasierter Ansatz

Der modellbasierte Ansatz wird durch die Integration des MBT Modells von Schieferdecker (Abbildung 2.3) bestätigt. Die Verwendung von Testmodellen zu Generierung von Tests bildet auf der Testing Seite die eingehende Schnittstelle zu der software-basierten Methodik. Die Verwendung von AUTOSAR BSW Deployment Informationen aus dem Systemmodell, bzw. der Partitionierung des Systemmodells bildet den Input für die Off-Board Automatisierung.

Genutzte Methoden, wie die erfolgte Integration in die modellbasierte Entwicklung, wurden bewusst nur erwähnt, da die entwickelte Methode diese nur als Input nutzt. In diesem Bereich wurden keine neuen Erkenntnisse gewonnen, da in erster Linie ein bestehender Modellanalyse-Ansatz adaptiert wurde. Die Informationen aus der modellbasierten Entwicklung, die dem Deployment entnommen werden können, werden verwendet, um die Schnittstellendefinition für die implementierten Module der Off-Board Testautomatisierung parametrieren zu können.

6.3.2. Bestätigung in verteilten Systemen

Die dargestellte Methode kann in einzelnen Komponenten auch in verteilten Systemen Anwendung finden. Der Ansatz ist aufgrund seines Anspruches, eine AUTOSAR BSW Implementierung zu werden, als ein generisches Softwaremodul realisiert worden. Jedes der identischen Module kann durch eine komponenten-individuelle Adressierung im verteilten System angesprochen werden und erlaubt somit eine einfache Implementierung in jede

Komponente des verteilten Systems. Die Logik der entwickelten Methodik, welcher simAgent welche Sequenzen im verteilten System ausführen soll, liegt im simMaster, also außerhalb des SUT. Diese Umsetzung war ein Schlüsselbaustein für eine effiziente und minimal invasive Methodik für die verteilte Simulation von Signalen.

Grundlegende Erkenntnis ist hierbei, dass für bestimmte Signalverläufe enorme Vorteile geschaffen wurden, wenn diese über eine software-basierte Methode in das verteilte System eingespielt werden. Schwieriger wird es bei langen oder hochdynamischen Signalverläufen, da hierbei der ECU Speicher die limitierende Größe ist. Lange oder hochdynamische Signalverläufe sind im Detail betrachtet Signalverläufe, die viele Datenpunkte beinhalten. Hierunter fallen besonders komplexe hochfrequente Verläufe.

Für den Großteil von Anwendungsfällen, die sich in der automobilen Produktentwicklung ergeben, das manuelle Betätigen von HMI-Schnittstellen stellt diese Methode eine ideale Automatisierungsmethodik für verteilte Systeme dar. Wie bereits in Kapitel 6 - Validierung herausgestellt, kann die Funktionslogik im verteilten System sehr gut reproduziert und auch ideal auf andere Systeme portiert werden und stellt neue Möglichkeiten des Vergleichens von Systemverhalten verteilter Funktionen.

6.3.3. Bestätigung der Automatisierungsmethodik

Die dargestellte Methode bringt elementare Vorteile im Bereich der Automatisierung, durch ihren grundlegenden Aufbau. Entscheidend war für diesen Punkt, dass die Implementierung möglichst generisch wird und die Module eine Schnittstelle in der AUTOSAR BSW eröffnen. Hinzu kommt, dass die Logik der Module nahezu vollkommen nach Off-Board ausgelagert wurde. Durch diese Art der Umsetzung ermöglicht die integrierte software-basierte Methodik, Systemeingaben wiederholt in verschiedenen Testebenen (*Komponenten-*, *Integrations-*, *System-* und *Akzeptanztest*) einspeisen zu können und eine Vergleichbarkeit der Tests über die genannten Testebenen. Hierbei schafft die gezeigte Methodik in erster Linie Potentiale bei der Ablösung manueller Systemeingaben direkt an dem HMI.

Grundlegend kann aus dem Kapitel 6 der Validierung entnommen werden, dass ein sehr hoher Automatisierungsgrad Im Bezug auf Systemeingaben, möglich wird. Die Aussagekraft der Testergebnisse variiert je nach Testebene. Im Rahmen der Komponenten kann eine exakte Reproduzierbarkeit von Logik und Timing-Verhalten erreicht werden. Sobald der Testfokus auf verteilte Systeme ausgedehnt wird, kann in erster Linie nur noch die Funktionslogik reproduzierbar getestet werden. Eine Bestätigung des Timing-Verhaltens im verteilten System kann nur mit zeitscheiben-basierter Kommunikation erreicht werden.

Die Testqualität durch die Reproduzierbarkeit von Systemverhalten innerhalb der Komponente ist durch das verwendete Scheduling unvergleichbar hoch. Diese Art der Simulation ermöglicht daher ein vorhersehbares Verhalten, welches durch die Simulation ausgelöst wird.

6.4. Weitere Fallstudien

Der in Kapitel 3 entwickelte Ansatz mit dem in Kapitel 5.6 beschriebenen Scheduling zeigt den größten Mehrwert in der Reproduzierbarkeit von Systemeingaben in einem embedded System. Hierzu wurden noch weitere Use Cases, zu der in Kapitel 6 detailliert analysierten Fallstudie, untersucht. Ziel der weiteren Fallstudien ist zu zeigen, dass das Verhalten auch in verschiedenen Use Cases identisch ist. Es wurde dabei explizit darauf geachtet, jeweils andere Systeme bzw. Systemkombinationen zu betrachten.

Untersucht wurden vor allem Fälle im Gesamtsystem (an einem realen Fahrzeug) im normalen Betriebszustand. Dies heißt, dass kein Einfluss auf das Systemverhalten genommen wurde und daher eine sich dynamisch ändernde Buslast vorherrscht. Somit waren je Messung nicht immer die gleichen Vorbedingungen im Gesamtfahrzeug vorhanden. Ziel war es immer Funktionen des verteilten Systems zu steuern und hierbei darauf zu achten, dass die notwendigen Vorbedingungen für einen erfolgreichen Testablauf vorliegen. Somit wurde die Betrachtungsgrenze auf die für die Funktion relevanten Umgebungsbedingungen eingeschränkt. Das Gesamtsystem hingegen verändert seinen Zustand, welcher sich aber in den gewählten Use Cases nicht auf die getestete Funktionslogik auswirkt. Beispiel: Wird über einen Testfall die Geschwindigkeitswarnung im Fahrzeug gesetzt, wurde diese nach einem Testdurchlauf wieder deaktiviert.

Im Rahmen der weiteren Fallstudien wurden die gleiche Untersuchung wie in Kapitel 6.2.3 angestrebt und bei den Use Cases immer die beiden Szenarien reale Kundeneingabe und simulierte Eingabe gegenübergestellt. Dies soll hier nochmal den Mehrwert der Reproduzierbarkeit eines Use Cases durch die neue Methodik bekräftigen.

Anmerkung: Das Ergebnis z.B. 100 von 100 bedeutet, dass die Tests gemäß ihrer Spezifikation die Kundenfunktion vollständig und korrekt ausgeführt haben. Es trat somit kein Fehler bei der Durchführung des Testfalls auf. Die Funktion wurde vollumfänglich wie erwartet ausgeführt, die möglichen Veränderungen im Gesamtsystem hatten keine Auswirkungen.

6.4.1. Use Case Geschwindigkeitswarnung

Das Setzen einer *Geschwindigkeitswarnung* durch den Kunden um z.B. eine Warnung bei der Nutzung von Winterreifen einzustellen. Hierbei kann ein Geschwindigkeitslevel gesetzt werden, sowie die Funktion grundlegend ein- bzw. ausgeschaltet werden.

Use Case	Aktivierung der Funktion *Geschwindigkeitswarnung* mit dem letzten gesetzten Geschwindigkeitswert der Funktion.
Ziel	Funktion *Geschwindigkeitswarnung* aktiviert.
Vorgang	Kunde aktiviert über Infotainment Menü -> Mein Fahrzeug -> Fahrzeugeinstellungen -> Geschwindigkeitswarnung -> Geschwindigkeitswarnung aktivieren.
Testplattform	Gesamtfahrzeug
Komponenten	Dynamic Stability Control (DSC) Headunit Kombiinstrument Motorsteuerung Zentralsteuergerät (BDC)
Anzahl Messungen	100 / 100 (noSync / Sync)
Auswertung	Reproduzierbarkeit -> Kundeneingabe Abbildung 6.14 -> über Simulationsmethodik Abbildung 6.15
Ergebnis	100 von 100 Tests ohne Synchronisation waren erfolgreich. 100 von 100 Tests mit Synchronisation waren erfolgreich.

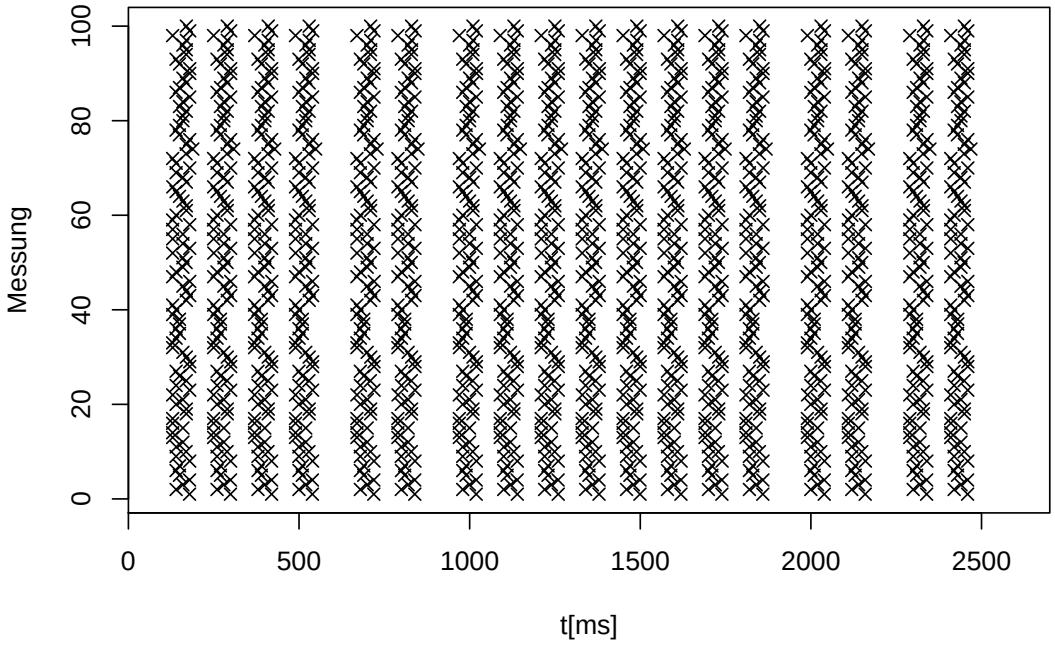


Abb. 6.14.: Timing-Verhalten Geschwindigkeitsbegrenzung Kundeneingabe

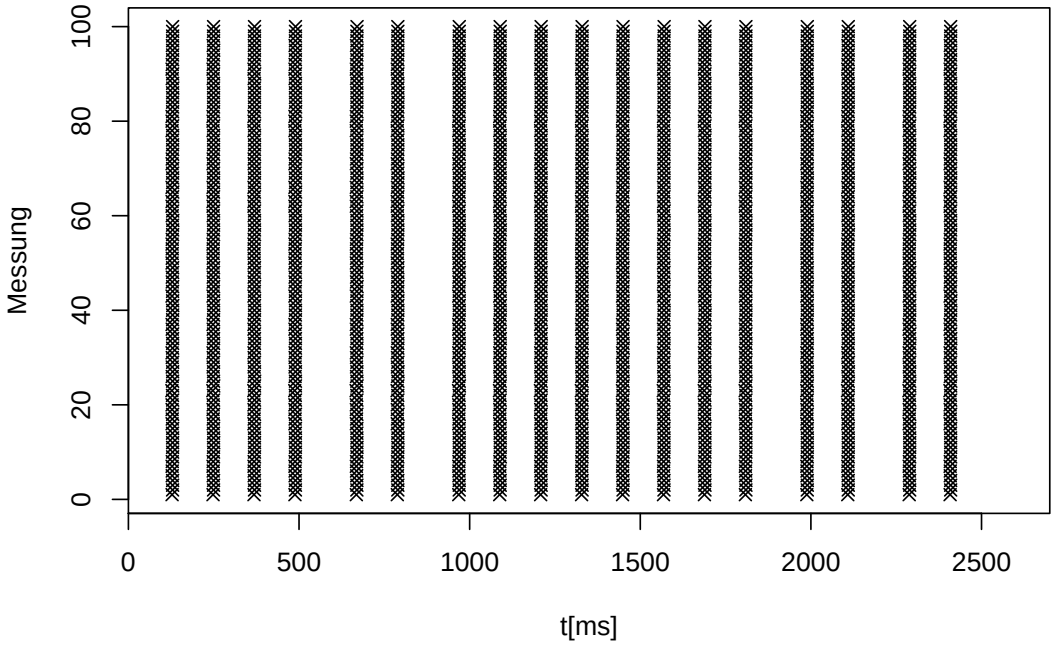


Abb. 6.15.: Timing-Verhalten Geschwindigkeitsbegrenzung Simuliert (sync)

6.4.2. Use Case Sofort Laden

Die Funktion *Sofort Laden* ist eine Teilfunktion des intelligenten Ladens des Hochvoltspeichers. Hierbei hat der Kunde die Möglichkeit das Fahrzeug z.B. zeitgesteuert zu laden. Ist solch ein zeitgeseuertes Laden aktiv, so muss der Kunde bei Bedarf des sofortigen Ladens manuell umschalten. Hierzu kann er über die Ladeeinstellungen im HMI diese Funktion auslösen.

Use Case	Aktivierung der Funktion *Sofort Laden* des Fahrzeuges durch den Kunden bei gestecktem Ladestecker und zeitgesteuertem Laden. Ladezustand kleiner 50%.
Ziel	Ladevorgang der Hochvoltbatterie gestartet.
Vorgang	Kunde aktiviert über Infotainment Menü -> Mein Fahrzeug -> Fahrzeugeinstellungen -> Laden/Klima planen -> Sofort laden.
Testplattform	Gesamtfahrzeug
Komponenten	E-Maschinen-Steuerung (Hochvoltpowermanagement) Headunit Hochvoltspeicher Kombiinstrument Standardladeelektronik Zentralsteuergerät (BDC)
Anzahl Messungen	100 / 100 (noSync / Sync)
Auswertung	Reproduzierbarkeit -> Kundeneingabe Abbildung 6.16 -> über Simulationsmethodik Abbildung 6.17
Ergebnis	100 von 100 Tests ohne Synchronisation waren erfolgreich. 100 von 100 Tests mit Synchronisation waren erfolgreich.

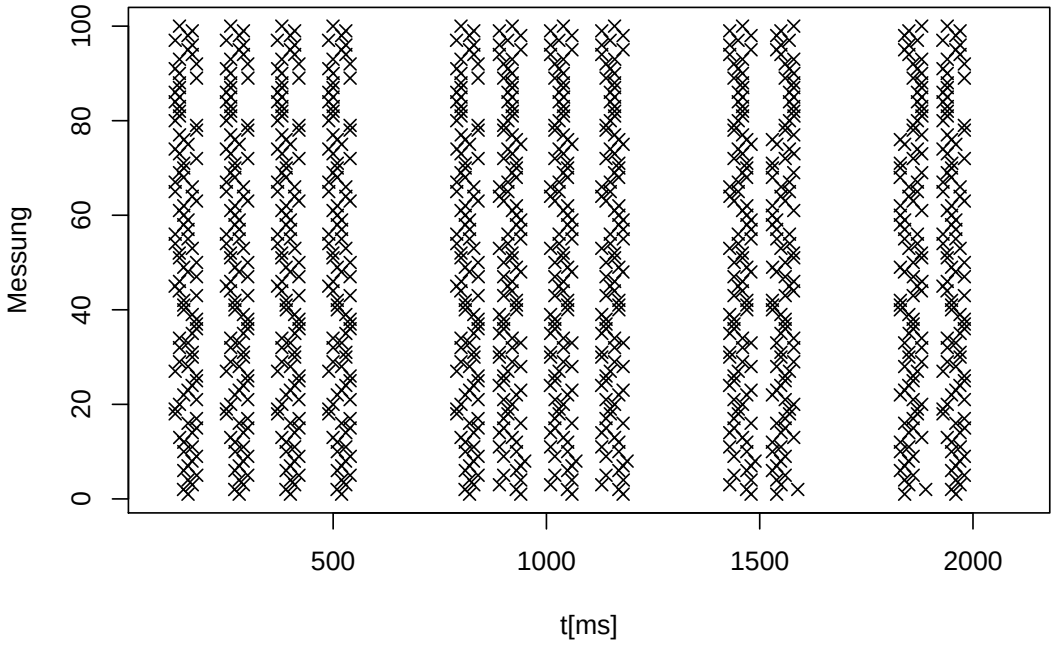


Abb. 6.16.: Timing-Verhalten Sofort Laden - Kundeneingabe

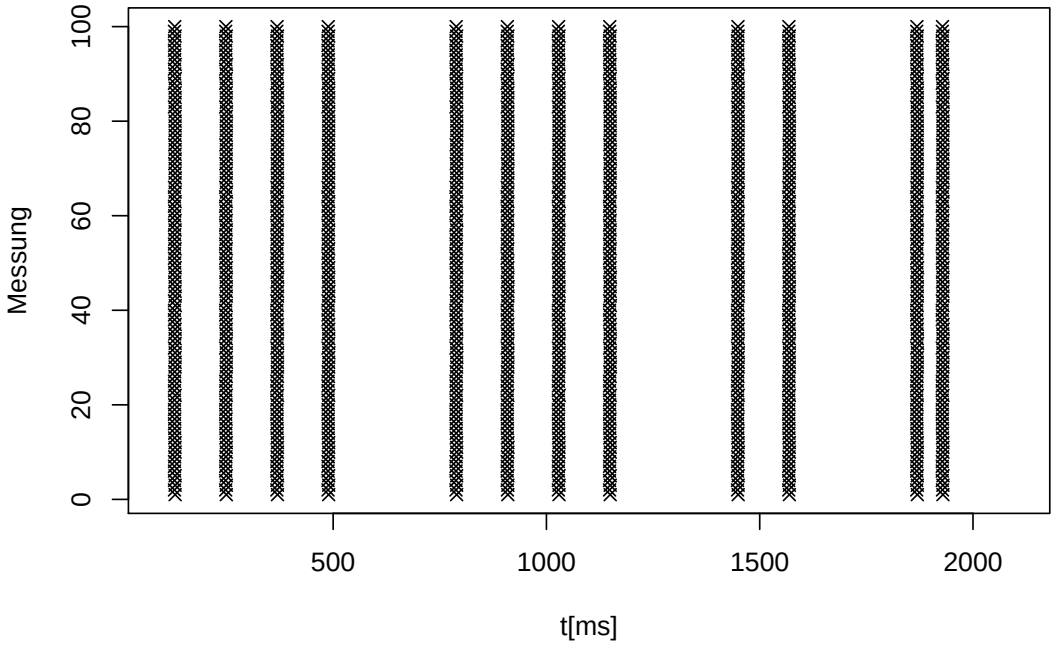


Abb. 6.17.: Timing-Verhalten Sofort Laden - Simuliert (sync)

6.4.3. Use Case Günstig Laden

Die Funktion *Günstig Laden* ist eine Teilfunktion des intelligenten Ladens des Hochvoltspeichers. Hierbei hat der Kunde die Möglichkeit das Fahrzeug z.B. zeitgesteuert in einem anderen Stromtarif zu laden. In dem verwendeten Fahrzeug muss der Kunde hierzu das sogenannte Ladefenster (frühester Ladebeginn und spätestes Ladeende, sowie eine Abfahrtzeit) einstellen und die Funktion Günstig Laden aktivieren. Das Fahrzeug berechnet die Ladedauer und versucht das Ladefenster maximal auszunutzen, um das Fahrzeug zum gewählten Abfahrtszeitpunkt vollständig aufgeladen zu haben.

Use Case	Aktivierung der Funktion *Günstig Laden* des Fahrzeuges durch den Kunden bei gestecktem Ladestecker mit voreingestelltem Ladefenster. Ladezustand kleiner 50%.
Ziel	Ladevorgang der Hochvoltbatterie wird unterbrochen.
Vorgang	Kunde aktiviert über Infotainment Menü -> Mein Fahrzeug -> Fahrzeugeinstellungen -> Laden/Klima planen -> Günstig Laden aktivieren.
Testplattform	Gesamtfahrzeug
Komponenten	E-Maschinen-Steuerung (Hochvoltpowermanagement) Headunit Hochvoltspeicher Kombiinstrument Standardladeelektronik Zentralsteuergerät (BDC)
Anzahl Messungen	100 / 100 (noSync / Sync)
Auswertung	Reproduzierbarkeit -> Kundeneingabe Abbildung 6.18 -> über Simulationsmethodik Abbildung 6.19
Ergebnis	100 von 100 Tests ohne Synchronisation waren erfolgreich. 100 von 100 Tests mit Synchronisation waren erfolgreich.

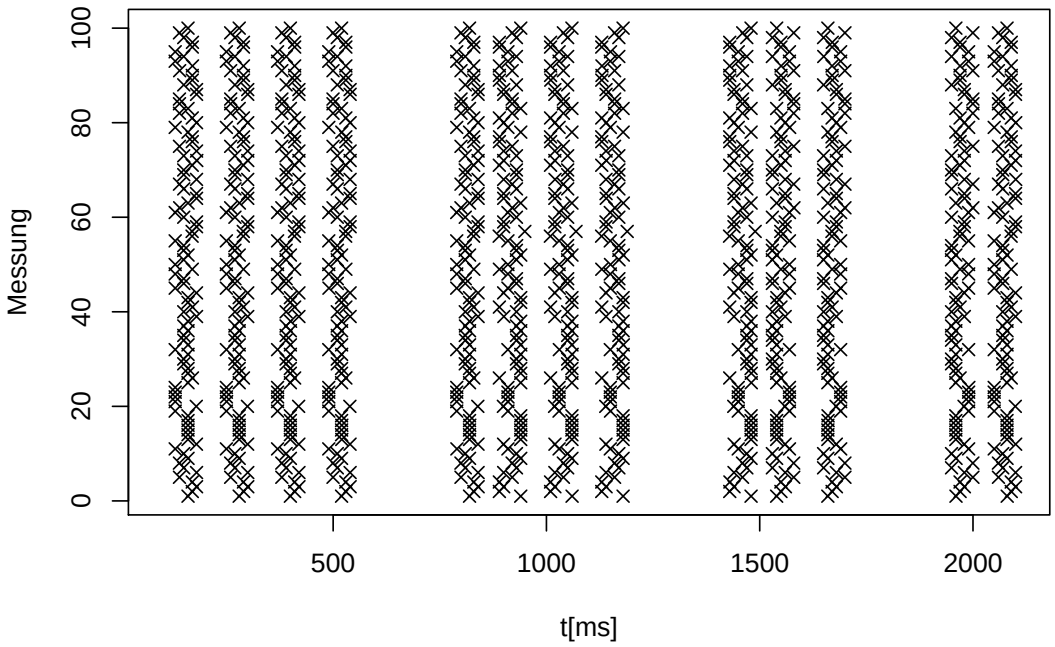


Abb. 6.18.: Timing-Verhalten Günstig Laden - Kundeneingabe

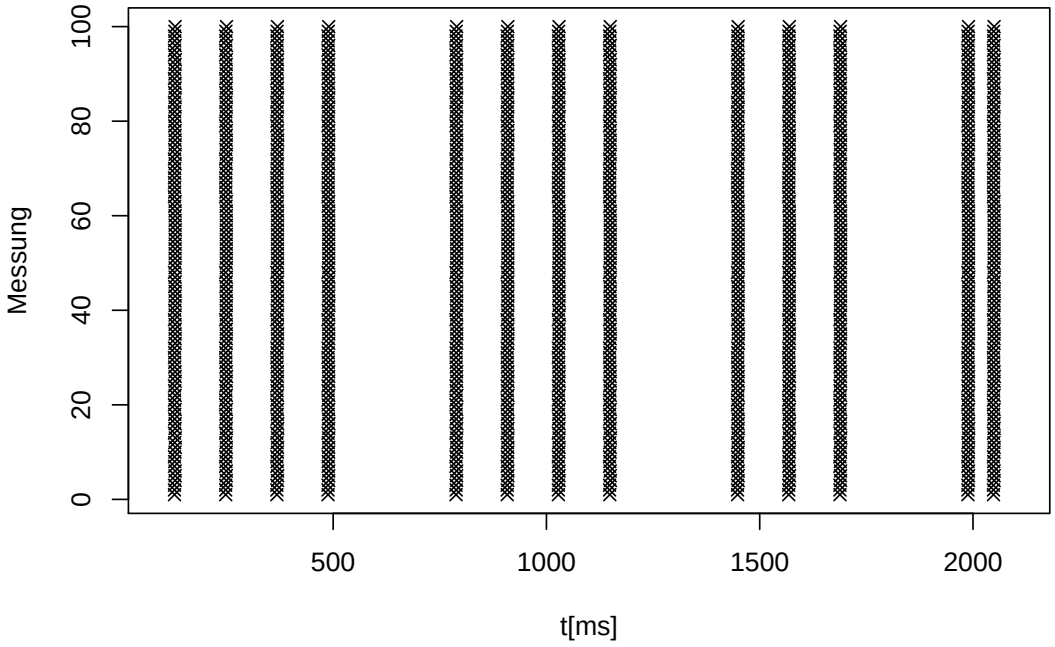


Abb. 6.19.: Timing-Verhalten Günstig Laden - Simuliert (sync)

6.4.4. Use Case Standklima aktivieren

Die *Standklimatisierung* in einem BMW Hybrid Fahrzeug nutzt einen elektrischen Klimakompressor und bezieht die Energie aus dem Hochvoltspeicher. Ist das Fahrzeug zeitgleich mit der Ladeinfrastruktur verbunden, wird automatisch ein Ladevorgang gestartet, um die Energie aus dem Stromnetz zu beziehen. Dies stellt sicher, dass dem Kunden auch bei Abfahrt die maximale Reichweite des Fahrzeugs zu Verfügung steht.

Use Case	Aktivierung der *Standklimatisierung* des Fahrzeuges durch den Kunden bei gestecktem Ladestecker.
Ziel	*Standklimatisierung* ist aktiviert, Ladevorgang der Hochvoltbatterie gestartet.
Vorgang	Kunde aktiviert über Infotainment Menü -> Mein Fahrzeug -> Fahrzeugeinstellungen -> Laden/Klima planen -> Standklima aktivieren -> Sofort aktivieren.
Testplattform	Gesamtfahrzeug
Komponenten	E-Maschinen-Steuerung (Hochvoltpowermanagement) Headunit Hochvoltspeicher Klimakompressor Klimasteuergerät Kombiinstrument Standardladeelektronik Zentralsteuergerät (BDC)
Anzahl Messungen	100 / 100 (noSync / Sync)
Auswertung	Reproduzierbarkeit -> Kundeneingabe Abbildung 6.20 -> über Simulationsmethodik Abbildung 6.21
Ergebnis	100 von 100 Tests ohne Synchronisation waren erfolgreich. 100 von 100 Tests mit Synchronisation waren erfolgreich.

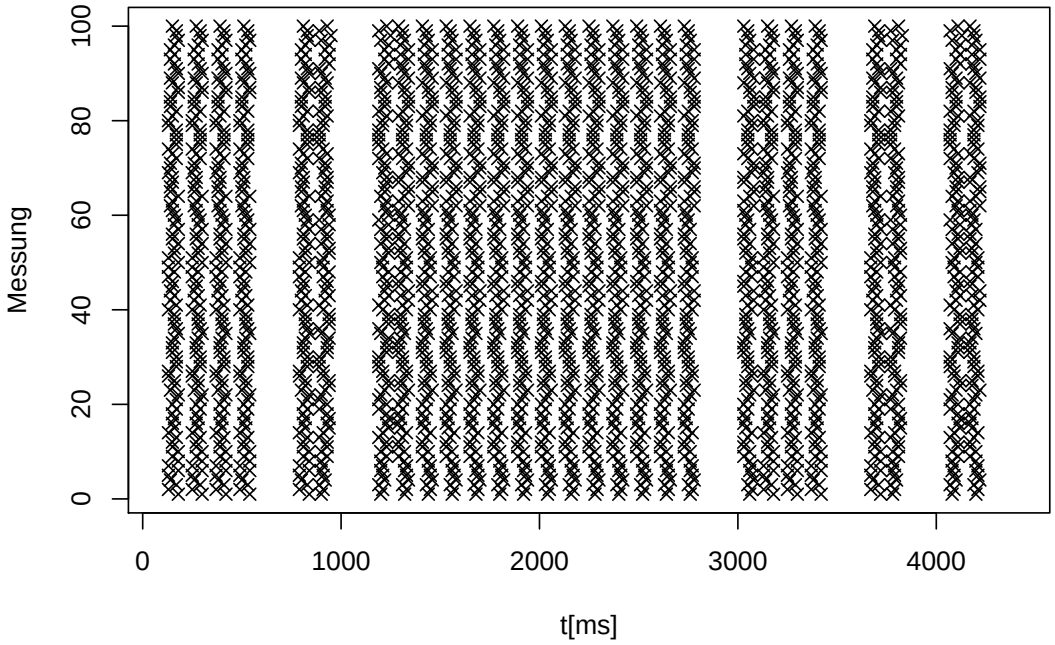


Abb. 6.20.: Timing-Verhalten Standklima aktivieren - Kundeneingabe

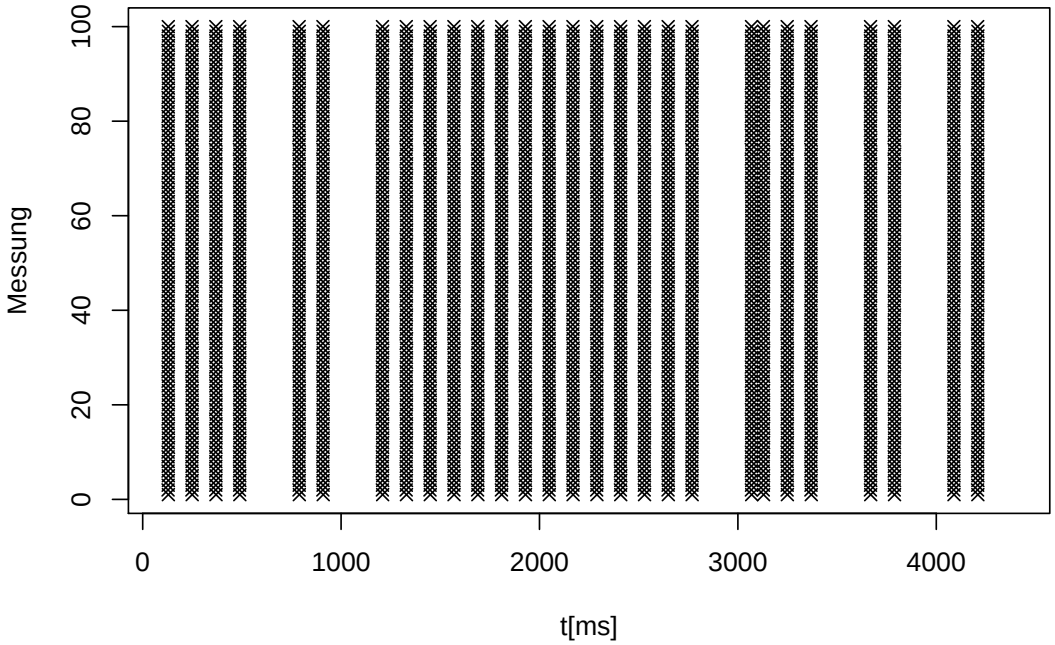


Abb. 6.21.: Timing-Verhalten Standklima aktivieren - Simuliert (sync)

6.5. Zusammenfassung und Diskussion der Evaluation

Das folgende soll kurz die Aussagekraft der Messungen beleuchten. Hierbei werden die Punkte Auswahl der Fallstudien, Logging und Zeitsynchronisation bewertet.

Die gewählten Fallstudien bzw. Use Cases spiegeln das Kundenverhalten sehr exakt wieder. Durch die Vergleichsmessungen bei manueller Systemeingabe, durch die Variante Arduino wurde ein äquivalentes Funktionsverhalten bezüglich nicht synchronisierter Simulation erreicht. Die ausgewählten Szenarien führten in den automatisierten Testabläufen zu Gesamtsystemreaktionen, die durch manuelle Eingabe nicht möglich waren. Zusammengefasst sind die Szenarien kundenrealistisch bezüglich ihrer Funktion, führten in allen Fällen zu einer für den Kunden richtigen Funktionsreaktion, konnten aber nur durch die neue Methodik reproduziert werden.

Der Einfluss Loggingausgaben für die Erzeugung von internen Logdateien haben zwar einen Einfluss auf die Prozessorlaufzeit, da diese aber auch in den Fällen der manuellen Systemeingabe aktiv waren, wird ihr Einfluss nicht negativ gewertet sondern als normale zusätzliche CPU-Last. Das Steuergerät hatte zu jeder Zeit genug Ressourcen verfügbar, um eine vollständige Prozessorauslastung durch die zusätzliche Last zu verhindern.

Die Zeitsynchronisation zwischen Datenlogger und Steuergerät war eine Behelfslösung da eine andere Technologie nicht zur Verfügung stand, mit der eine NTP Synchronisation möglich geworden wäre. Die Zeitverzögerung von ca. 3,5 ns durch den Trigger über die Signalleitung mit einer Länge von 80 cm kann bei einer mittleren Übertragungsgeschwindigkeit von 225000 m/s (bei Kupferleitungen) sowie in Bezug auf die Taskintervalle von 10 ms der Treiberschnittstelle vernachlässigt werden. Die Signallaufzeit wird hier als permanente Unschärfe bzw. Offset betrachtet, die/der aber auch im Rahmen der Messungen auf dem CAN auftritt. Hierbei sind die Leitungslängen des CAN auch auf ca. 80 cm festgelegt wurden, um somit dem Offset entgegen zu wirken.

Die o.g. Aspekte ergeben für die Messungen einige Potentiale, wie das Thema der Zeitsynchronisation oder der Verwendung von Loggingausgaben, die aber unter den gegebenen Bedingungen vernachlässigt werden können. Folglich besitzen die Messungen auf Grund genannter Auswahl an Use Cases und der Art der Messung genug Aussagekraft, um die unter Kapitel 6 genannten Aspekte nachzuweisen bzw. zu bestätigen.

7. Fazit

In dieser Arbeit wurde eine Methode erarbeitet, die das Ziel hat, die Qualität und den Grad der Automatisierung beim Testen von verteilten Funktionen auf System- und Akzeptanztestebene essentiell zu verbessern. Die Grundvoraussetzung war die Umsetzung in Form eines software-basierten Ansatzes. Nachfolgend werden im Rahmen der Zusammenfassung und in der darauf folgenden Diskussion die Themen kurz beschrieben und die sich ergebenden Problemstellungen andiskutiert.

7.1. Zusammenfassung

Aus der Idee, der stetig wachsenden Komplexität von Automotive Systemen unter dem wachsenden Kosten und Zeitdruck wurde in einem eingegrenzten Rahmen nach einer software-basierten Lösung für die Verbesserung der Testqualität und des Automatisierungsgrades gesucht. Eine verbesserte Testqualität sowie ein erhöhter Automatisierungsgrad reduzieren den Kosteneinsatz. Durch den Einsatz einer software-basierten Lösung kann die gezeigte Methode einfach auf andere Architekturen adaptiert werden und identische Ergebnisse bringen.

Nachfolgend werden die in Kapitel 1.3 aufgeführten Ziele aus den Bereichen *Komplexität*, *Automatisierung* und *Qualität* mit den im Rahmen der Arbeit entstandenen Ergebnisse zusammengefasst.

Komplexität

Ziel ist eine Verbesserung bei der Beherrschung der Komplexität von verteilten Funktionen. Durch die Verwendung eines modellbasierten Ansatzes und der Integration einer Modellanalyse kann die Beherrschung der Komplexität verbessert werden. Dies äußert sich in erster Linie dadurch, dass Funktionen in einem Testmodell modelliert werden und daher visuell leichter erfasst werden können. Zum anderen ermöglicht die Modellanalyse die Testfälle automatisiert zu generieren und erleichtert damit die aufwändigen Wirkkettenanalysen, bei denen versucht wird Abhängigkeiten und Auswirkungen von Funktionen zu identifizieren.

Die Methode mit einer Interaktion auf der Ebene der Treiber-Schicht reduziert zudem die Signaltypen und vereinfacht somit die Lesbarkeit von Tests.

Automatisierung — Ziel ist das Ablösen manueller Tätigkeiten und die Reduktion von zeitraubenden Wirkkettenanalysen. Die vorgestellte Methode erreicht einerseits durch den modellbasierten Ansatz die Möglichkeit dass Tests generiert werden können, andererseits über die Integration einer software-basierten Simulation das vollautomatisierte Einspielen und Ausführen von Tests in verschiedene Systeme.

Qualität — Ziel ist eine Reduktion von manuellen Systemeingaben, um die Reproduzierbarkeit sowie Vergleichbarkeit von Tests zu erhöhen. Die in den Methodenvorschlag vorgestellte Softwarelösung bietet hierfür eine sehr einfache Integration. Diese kann aufgrund der Interaktion am Treiber und der Synchronisation mit den Application-SWCen eine nahezu ideale (siehe Validierung) Reproduzierbarkeit darstellen.

Betrachtet man die unter Kapitel 1.3.4 aufgezeigten Beiträge so können diese in allen ihren Punkten bestätigt werden. Die Qualität des Testings konnte signifikant verbessert werden, wie die Validierungsergebnisse zeigen. Die Automatisierung wurde durch die Reduktion der Signaltypen vereinfacht.

Grundlegend kann ein in dieser Arbeit beschriebenes methodisches Vorgehen den Fokus des Testers auf das Modellieren eines umfangreichen Testmodells sowie eines wiederverwendbaren Umweltmodells lenken. Je besser die verwendeten Modelle desto besser kann mittels der gezeigten Methodik reproduzierbar und vergleichbar getestet werden. Eine software-basierte Simulationsmethodik ermöglicht dabei das methodische Vorgehen möglichst vollständig zu automatisieren.

Die Möglichkeiten der Integration sind vielzählig und relativ einfach, da die Methodik von Anfang an vorsieht, verschiedene Transport Layer oder Signaltypen zu unterstützen, ähnlich wie es ein Diagnose oder XCP Ansatz verwendet.

Die Erweiterung des DIO-AUTOSAR-Treibers ist eine Notwendigkeit und führt zu einem nicht mehr AUTOSAR konformen ECU. Um eine vollständige Konformität herzustellen, müsste der AUTOSAR Standard um den in Kapitel 4.4.3 dargestellten Teil ergänzt, sowie die Logik und das Sicherheitskonzept standardisiert werden.

Limitierungen für den Einsatz der Methodik ergeben sich durch Ressourcenknappheit in den Steuergeräten, sowie in verwendeten BUS-Technologien. Begrenzter Speicher (Kapitel 6.2.2) sowie zu langsame CPUs (Kapitel 6.2.2) können die Testlänge beschränken, eine Testdurchführung oder gar eine Implementierung verhindern.

7.2. Diskussion

Die dargestellte Methode stößt bei der Realisierung an die Grenze der AUTOSAR Konformität. Der in Kapitel 4.4.3 dargestellte neue Service (`DioSim_GetSim`) ist aktuell nicht im AUTOSAR Treiber Layer des aktuellen AUTOSAR Standard enthalten. Aus diesem Grund kann die Methodik nur in einem Entwicklungs- bzw. Forschungsbereich eingesetzt werden, da die Automobilhersteller aktuell keine eigenen BSW erstellen. Diese BSW wird über die Lieferanten der Steuergeräte bezogen. Einer Standardisierung sollte technisch nichts im Wege stehen, da darauf geachtet wurde, den Service analog zu bereits bestehenden auszubauen.

Folgend kann man sagen: Erst wenn dieser neu entwickelte Service im AUTOSAR Standard enthalten ist, kann diese Funktion effizient, kostengünstig und im verteilten System den Einsatz finden.

Die Dauer der Standardisierung für AUTOSAR beträgt ca. ein Jahr. Diese Aussage basiert auf dem Release Zyklus der neuen AUTOSAR-Adaptive Plattform [AUT17]. Nach einer erfolgreichen Standardisierung ist der Grundstein für die software-basierte Simulation von Kundeneingaben gelegt. Die Nutzung des Services ist dann den OEM überlassen und kann ohne die Lieferanten erfolgen. Durch die Integration der SWC simAgent in Verbindung mit dem simGW ist der methodischen Ansatz fahrzeugseitig bzw. in den Steuergeräten dargestellt.

Die dargestellte Methodik zeigt in erster Linie Potentiale bei der Simulation von digitalen Systemeingaben. Für die Simulation von analogen Werten ist die Methode ebenfalls geeignet, doch aufgrund der erhöhten Anzahl an Datenpunkten reduziert sich bei gleicher Speicherbelastung die mögliche Testfallzeit. Daher gilt es Lösungen und Ansätze zu finden, die Datenpunkte zu realisieren oder sich auf spezielle Anwendungsfälle zu beschränken. Ein denkbares Szenario für die Verwendung der Methodik für analoge Werte sind Ruhestromverläufe. Diese sind Testfälle bei denen eine enorm lange Testzeit von ca. 24 Stunden vorherrscht, aber die Dynamik verschwindend gering ist. Hierbei könnte die Methodik durchaus Anwendung finden. Erste Untersuchungen bzw. Berechnungen haben gezeigt, dass dies unter Verwendung des gleichen Speichers möglich ist. Interessant ist die erstmalige Chance, das Systemverhalten bei Ruheströmen und Ruhestromverletzungen in den Vergleich stellen zu können.

Ein weiteres Szenario ist z.B. Das Systemverhalten bei Startspannungen, also dem Start des Verbrennungsmotors oder dem Zuschalten von Hochstromverbrauchern. Hierbei ist im Gegensatz zu den Ruheströmen die Testzeit minimal, kleiner als 3 Sekunden, aber die Dynamik ist extrem hoch, lässt sich aber auch mit der Anzahl an Datenpunkte realisieren.

Die Anwendung der Methodik für analoge Werte ist durchaus interessant, da die Reproduzierbarkeit eines Systemverhaltens in Bezug auf Strom- und Spannungsverhalten im realen Fahrzeug nicht möglich ist. Die Chemie einer Batterie ist hierbei das entscheidende Hindernis. Temperatur, Strombezug und Ladeströme verändern die Batterie nachhaltig und ein Zustand wie bei einem bereits durchgeführten Test ist nicht wieder herstellbar. Durch den Einsatz der

gezeigten Methodik kann dieses Problem umgangen werden. Zusammengefasst kann gesagt werden, dass die Methodik für analoge Werte durchaus einsetzbar ist, solange die Eingrenzung durch Zeitdauer oder Dynamik berücksichtigt wird.

7.3. Ausblick

Die dargestellte Methodik wurde mit dem Ziel entwickelt ein möglichst kunden-äquivalentes Verhalten zu erreichen. Aus diesem Grunde wurde eine Übertragung der Testdaten vor Ausführung des Testfalls präferiert. Diese Herangehensweise hat den Nachteil des Ressourcenbedarfs im Steuergerät. Bei einem Verzicht auf den Vorteil, dass die Buslast nicht beeinflusst wird, könnte die Methodik ähnlich wie CCP oder XCP um eine Möglichkeit der permanenten Datenkommunikation erweitert werden. Dadurch könnten vor allem lange, speicherintensive Tests bzw. Tests mit viele Datenpunkte ermöglicht werden.

XCP ist bereits im AUTOSAR Standard enthalten und findet daher weitreichenden Einsatz in der Automobilindustrie. Eine Integration des Service aus Kapitel 4.4.3 in den AUTOSAR Standard würde dazu führen, dass alle Steuergeräte mit AUTOSAR eine Schnittstelle im Treiber bereitstellen würden. Alle anderen Implementierungen, z.B. des simAgents könnten herstellerspezifisch sein. Hierzu wurden bereits erste Gespräche geführt und stießen großteils auf Zustimmung für eine Einbringung in den Standard stießen.

Durch die Weiterentwicklung der AUTOSAR Plattform zu AUTOSAR-Adaptive reduziert sich der Aktualisierungszyklus und somit erhöht sich die Chance auf ein mögliches zeitnahes Umsetzen der hier gezeigten Methodik. [AUT17]

Eine Standardisierung der Methodik in AUTOSAR sollte zudem eine Standardisierung der Kommunikationsnachrichten der simModule beinhalten. Diese würde eine einheitliche Kommunikation, herstellerübergreifend mit allen Modulen sicherstellen.

A. Deklarationen und Technische Begriffe

Abnahmetest Formales Testen hinsichtlich der Benutzeranforderungen und -bedürfnisse bzw. der Geschäftsprozesse.

Agile Softwareentwicklung Eine auf iterativer und inkrementeller Entwicklung basierende Gruppe von Softwareentwicklungsmethoden, wobei sich Anforderungen und Lösungen durch die Zusammenarbeit von sich selbstorganisierenden, funktionsübergreifenden Teams entwickeln.

APIX Automotive Pixel Link ist eine High Speed Gigabit Multikanal-Verbindung von Displays, Kameras und Steuergeräten.

AUTOSAR (AUTomotive Open System ARchitecture) Das Wort AUTOSAR und das AUTOSAR Logo sind geschützte Marken der AUTOSAR-Entwicklungspartnerschaft. Die Rechtsform der AUTOSAR-Partnerschaft ist eine GbR.

Automotive Bedeutet im Zusammenhang mit AUTOSAR: motorgetriebene Landfahrzeuge, die nicht auf Schienen fahren und primär einem Transportzweck dienen.

BAC Der BMW AUTOSAR Core, besteht aus einem AUTOSAR Stack und der BMW Systemsoftware, diese wird gemäß den Anforderungen des jeweiligen Steuergeräts konfiguriert. [BMW15b]

Codereview Ist eine systematische Bewertung des Quellcodes, welcher dazu dient Verbesserungspotentiale in der Software aufzudecken. Reviews erfolgen in verschiedenen Formen z.B. informale Walk-throughs oder formale Untersuchungen.

Continious Integration Kontinuierliche Integration bezeichnet die fortlaufende Zusammenführung von sich weiterentwickelnden Softwarekomponenten zu einer Anwendung. Ziel ist die Steigerung der Softwarequalität.

Deployment Deployment oder auch Softwareverteilung bezeichnet die Verteilung und Installation von Software auf Komponenten.

FIFO (First in First out) FIFO bedeutet *zuerst rein, zuerst raus* und bezeichnet ein Ablageverfahren, bei dem diejenigen Elemente, die zuerst abgelegt werden auch wieder zuerst entnommen werden.

Gateway Ein Gateway ermöglicht die Kommunikation zwischen Netzwerken, die auf unterschiedlichen Protokollen basieren.

Hardware in the Loop bezeichnet ein Verfahren bei dem ein eingebettetes System über seine Ein- und Ausgänge an ein angepasstes Gegenstück (HIL-Simulator) gekoppelt wird.

Infotainment Der Begriff Infotainment umfasst Audio- und visuelle Unterhaltungs- und Informationselektronik.

Integrationstest Testen mit dem Ziel, Fehlerzustände in den Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten aufzudecken.

Komponententest Testen einer einzelnen Komponente mit der für sie bestimmten Software und Funktionalität unter möglichst realen Umgebungsbedingungen.

Metamodell beschreibt ein übergeordnetes Modell. Das Metamodell beschreibt modellhaft einen bestimmten Aspekt der Erstellung von konzeptuellen oder formalen Beschreibungsmodellen.

modellbasiert ist der Überbegriff für die Begriffe modellorientiert, modellgetrieben und modellzentrisch.

Model in the Loop Modellbasierte Simulation eines eingebetteten Systems, in der frühen Entwicklungsphase.

Pfad Ein Weg durch einen Graphen entlang miteinander durch Kanten verbundenen Knoten.

Refinement Unter Refinement versteht man den Prozess der Verbesserung eines Produktes in Detailaspekten.

Scrum Ein iterativ inkrementelles Vorgehensmodell für das Projektmanagement, das im Allgemeinen bei agiler Softwareentwicklung verwendet wird.

Sequenzdiagramm Verhaltensdiagramm der UML, mit dem bestimmte Aspekte des dynamischen Verhaltens eines Systems ausgedrückt werden. Es zeigt die an einer Kommunikation beteiligten Entitäten und die zwischen ihnen ausgetauschten Nachrichten an.

Signalsequenz bezeichnet die extrahierte Sequenz des Pegelverlaufs eines Channels für die Systemein- oder -ausgabe. Diese kann z.B. in Form von Tupeln oder Splines vorliegen.

simAgent ist ein Begriff für die Application-SWC mit der Funktion des Bereitstellens von

simulierten Sensorwerten. “sim” steht hier als Abkürzung für Simulation. Nach der Definition von Jennings und Wooldridge ist ein Agent “An agent is an encapsulated computer system that is situated in some environment, and that is capable of fexible, autonomous action in that environment in order to meet its design objectives.”
▷ Code C.2

simMaster ist die Off-Board Komponente der Simulationsmethodik, die den simAgent mit den Signalen versorgt.

Simulation Untersuchung eines Modells durch experimentelle Ausführung des darin modellierten Verhaltens.

Softwarearchitektur Eine Softwarearchitektur ist die oberste Strukturebene eines komplexen Softwaresystems.

Software in the Loop ist ein Verfahren, bei dem die Software in einer simulierten Umgebung getestet wird.

Systemmodell Modell des zu entwickelnden System, insbesondere seiner Struktur und seines Verhaltens.

Systemtest Testen eines integrierten Systems, um sicherzustellen, dass es die spezifizierten Anforderungen erfüllt.

Testautomatisierung Einsatz von Softwarewerkzeugen zur Durchführung oder Unterstützung von Testaktivitäten.

Testfall Umfasst alle notwendigen Angaben, wie z.B. Vorbedingungen, eine Menge an Eingaben, sowie entsprechende Systemreaktionen bzw. Ergebnisse und Nachbedingungen. Testfälle haben das Ziel einen bestimmten Programmpfad auszuführen, um die spezifizierten Anforderungen zu prüfen.

Testfallgenerator Computerprogramm, das aus einer abstrakten Beschreibung bzw. Modell Testfälle in formaler Sprache generiert.

Testmodell Modell des Tests, insbesondere seiner Struktur und seines Verhaltens.

Time to Market Unter Time to Market versteht man die Dauer von dem Beginn der Produktentwicklung bis zur Platzierung des Produktes im Markt.

Unified Diagnostic Services ist ein Diagnose-Kommunikationsprotokoll in der Automobil Elektronik, welches in dem Standard ISO 14229 [Int13] spezifiziert ist.

Umgebungsmodell Modell der Umgebung des zu entwickelnden Systems, insbesondere seiner Struktur und seines Verhaltens.

Use Case bezeichnet den Kundenfall in dem dieser eine gewisse Funktion direkt, durch Interaktion mit dem System oder auch indirekt durch sein Zeit-Verhalten auslöst bzw. bedient.

V-Modell Vorgehensmodell für die Softwareentwicklung, um die Aktivitäten des Softwareentwicklungslebenszyklus von der Anforderungsspezifikation bis zur Wartung zu beschreiben.

Zustandsautomat Ist ein Modell des Systemverhaltens, das Zustände des Systems und Zustandsübergänge darstellt.

Quellen: [RBGW10, SBB11, KF09]

B. Anwendungsszenarien

Der in dieser Arbeit entwickelte Ansatz wurde mit dem Ziel entwickelt, möglichst breit gestreut in verschiedenen Szenarien Anwendung zu finden. Das folgende Kapitel gibt hier einen Einblick, in welchen Bereichen für welche Use Cases der Ansatz diskutiert wurde und einen Nutzen bringen würde. Es werden die wesentlichen Stationen der Produktentwicklung bis zur Serienbetreuung betrachtet.

B.1. Entwicklung

Die Anwendungsfälle für eine wie hier gezeigte Methodik in der Entwicklung sind vielseitig. Hier können Anwendungsfälle auf nahezu allen Ebenen des V-Modells (siehe Abbildung 2.4) gefunden werden, um das Testing der Software zu automatisieren. Besonders interessant wird der Einsatz der Methodik auf dem Level der System- oder Akzeptanztests, da hier aktuell der größte Anteil manueller Tätigkeiten erfolgt.

B.2. Produktion

Anwendungsfälle in der Produktion finden sich in erster Linie in den s.g. Bandendekontrollen. Hier müssen aktuell Funktion durch Bandarbeiter manuell ausgeführt werden, um den korrekten Verbau und die bestimmungsgemäße Funktion eines Fahrzeugs mit allen seinen Funktionen sicherzustellen. Hier könnte durch die neu entwickelte Methodik ein Großteil der manuellen Tätigkeiten durch das Einspielen der entsprechenden Testfälle abgelöst werden. Folglich könnten damit Fehler in den manuellen Abläufen vermieden werden.

B.3. Handelsorganisation

In der Händlerorganisation könnte die Methodik in erster Linie bei Reparaturen Verwendung finden. D.h., nach Reparaturen an Fahrzeugen könnten die bereits in der Entwicklung verwendeten Tests wieder in das SUT eingespielt werden, um hier die korrekte Funktion nach Reparatur bestätigen zu können. Dies würde die Qualität der durchgeführten Reparaturen sichern.

C. Code

Folgend ist die Programmierung bzw. Umsetzung in Form von Source Code aufgelistet. Beginnend mit der AUTOSAR-BSW und den in der BSW (Abschnitt C.1) umgesetzten simModul gefolgt von der Applikations-SWC des simAgent (Abschnitt C.2).

C.1. BSW Module

C.1.1. Source Code DIO - `Dio.c`

```
#include "Dio.h"
#include "VAP_Features.h"

 /* DioSim Include */
#include "DioSim.h"

#if(DIO_DEV_ERROR_DETECT == STD_ON)
#include "Det.h"
#endif

#ifdef DIO_MULTIPLE_APP_SUPPORT
#include "miniHAL.h"
#endif


/*************AR Version*********************/
#define DIO_C_AR_MAJOR_VERSION  (4u)
#define DIO_C_AR_MINOR_VERSION  (0u)
#define DIO_C_AR_PATCH_VERSION  (3u)
/*************SW Version*********************/
#define DIO_C_SW_MAJOR_VERSION  (0u)
#define DIO_C_SW_MINOR_VERSION  (4u)
#define DIO_C_SW_PATCH_VERSION  (3u)
/************AR Version Check*****************/
#if(DIO_C_AR_MAJOR_VERSION != DIO_AR_RELEASE_MAJOR_VERSION)
  #error "Autosar major version of Dio.c and Dio.h are not matching."
#endif
#if(DIO_C_AR_MINOR_VERSION != DIO_AR_RELEASE_MINOR_VERSION)
  #error "Autosar minor version of Dio.c and Dio.h are not matching."
#endif
#if(DIO_C_AR_PATCH_VERSION != DIO_AR_RELEASE_PATCH_VERSION)
  #error "Autosar patch version of Dio.c and Dio.h are not matching."
```

```
33 #endif
34 /***********SW Version Check*****************/
35 #if(DIO_C_SW_MAJOR_VERSION != DIO_SW_RELEASE_MAJOR_VERSION)
36   #error "Software major version of Dio.c and Dio.h are not matching."
37 #endif
38 #if(DIO_C_SW_MINOR_VERSION != DIO_SW_RELEASE_MINOR_VERSION)
39   #error "Software minor version of Dio.c and Dio.h are not matching."
40 #endif
41 #if(DIO_C_SW_PATCH_VERSION != DIO_SW_RELEASE_PATCH_VERSION)
42   #error "Software patch version of Dio.c and Dio.h are not matching."
43 #endif
44
45 //#include "MemMap.h" //Commented out because of error: nO valid section
       defined.
46
47 #include "medriver/medriver.h"
48
49 #include <stdio.h>
50 #include <sys/stat.h>
51 #include <fcntl.h>
52 #include <unistd.h>
53 #include <string.h>
54 #include <errno.h>
55
56 #define DIO_PORT_ADDR_MASK (4)
57
58 //Virtual Channel Address (32Bit)   DEV  1Byte PORT 2Byte BIT_POS 1Byte
59 #define DIO_GET_DEVICE_ADDR_FROM_CHID(ChannelId) ( ChannelId < 0x400 ? 0 :
       1 )
60 #define DIO_GET_PORT_ADDR_FROM_CHID(ChannelId) ( ((ChannelId >> (8u)) % 4 )
        )
61 #define DIO_GET_CHANNEL_MASK_FROM_CHID(ChannelId) ( (1u) << ((ChannelId) &
       (0xFFu)) )
62 //Virtual Port Address (32Bit)   DEV  1Byte PORT 2Byte
63 #define DIO_GET_DEVICE_ADDR_FROM_PORTID(PortId) ( PortId <
       DIO_PORT_ADDR_MASK ? 0 : 1  )
64 #define DIO_GET_PORT_ADDR_FROM_PORTID(PortId) ( PortId % DIO_PORT_ADDR_MASK
        )
65
66 #define DIO_IO_MIRROR
67 #ifdef DIO_IO_MIRROR
68 //#define DIO_MAX_PORTS 8
69 static uint32 lport[DIO_MAX_PORTS];   //All STATIC changed to static.
70 #endif
71
72 #define Dio_PRP_INFO STD_OFF
73
74 #include "prp_logging.h"
75
76 //static P2CONST(Dio_ConfigType, AUTOMATIC, DIO_PBCFG) Dio_CurrConfigRef;
77
78 /*
79  * Local Functions
80  */
```

```
81 static FUNC(sint32, DIO_PRIVATE_CODE) Dio_meReadPort(int dev, int subdev,
       int ch,int *value)
82 {
83   int rv;
84   meIOSingle_t single;
85   single.iDevice=dev;
86   single.iSubdevice=subdev;
87   single.iChannel=ch;
88   single.iDir=ME_DIR_INPUT;
89   single.iValue=0x00;
90   single.iTimeOut=ME_VALUE_NOT_USED;
91   single.iFlags=ME_IO_SINGLE_TYPE_NO_FLAGS;
92   single.iErrno=0;
93
94 #ifdef DIO_MULTIPLE_APP_SUPPORT
95   uint8 u8SharedDev = (uint8)0;
96   uint8 u8SharedSubDev = (uint8)0;
97 #endif
98
99 #ifdef DIO_MULTIPLE_APP_SUPPORT
100   if (mHAL_MultiAppSupport() == TRUE)
101   {
102       mHAL_GetDIOResource(((dev*4)+subdev), &u8SharedDev, &u8SharedSubDev);
103       single.iDevice=u8SharedDev;
104     single.iSubdevice=u8SharedSubDev;
105   }
106 #endif
107   if ((rv = meIOSingle(&single, 1, ME_IO_SINGLE_NO_FLAGS/*
          ME_IO_SINGLE_NONBLOCKING */) != ME_ERRNO_SUCCESS)) {
108     char txtError[1024] = "";
109     meErrorGetMessage(rv, txtError, 1024);
110     PRP_DEBUG("%s(): meIOSingle: read port failed error: %s(rv=%d)\n",
           __FUNCTION__, txtError,rv);
111     return rv;
112   }
113   if (value != NULL) {
114     *value = single.iValue;
115   }
116 #ifdef DIO_IO_MIRROR
117   lport[dev + subdev] = *value;
118 #endif
119   //PRP_DEBUG("%s(): Dev[%d] SubDev[%d] Channel[%d] read value 0x%X\n",dev,
          subdev,ch,*value);
120   return ME_ERRNO_SUCCESS;
121 }
122
123 static FUNC(sint32, DIO_PRIVATE_CODE) Dio_meWritePort(int dev, int subdev,
       int ch,int value)
124 {
125   int rv;
126   meIOSingle_t single;
127   single.iDevice=dev;
128   single.iSubdevice=subdev;
129   single.iChannel=ch;
130   single.iDir=ME_DIR_OUTPUT;
```

```
131   single.iTimeOut=ME_VALUE_NOT_USED;
132   single.iFlags=ME_IO_SINGLE_TYPE_NO_FLAGS;
133   single.iErrno=0;
134
135   single.iValue = value;
136
137 #ifdef DIO_MULTIPLE_APP_SUPPORT
138   uint8 u8SharedDev = (uint8)0;
139   uint8 u8SharedSubDev = (uint8)0;
140 #endif
141
142 #ifdef DIO_MULTIPLE_APP_SUPPORT
143   if (mHAL_MultiAppSupport() == TRUE)
144   {
145       mHAL_GetDIOResource(((dev*4)+subdev), &u8SharedDev, &u8SharedSubDev);
146       single.iDevice=u8SharedDev;
147     single.iSubdevice=u8SharedSubDev;
148   }
149 #endif
150
151     if ((rv = meIOSingle(&single, 1, ME_IO_SINGLE_NO_FLAGS/*
          ME_IO_SINGLE_NONBLOCKING */) != ME_ERRNO_SUCCESS)) {
152     char txtError[1024] = "";
153     meErrorGetMessage(rv, txtError, 1024);
154     PRP_DEBUG("%s(): meIOSingle: write port failed error: %s (rv=%d)\n",
          __FUNCTION__, txtError,rv);
155     return rv;
156   }
157 #ifdef DIO_IO_MIRROR
158   lport[dev + subdev] = value;
159 #endif
160   //PRP_DEBUG("%s(): Dev[%d] SubDev[%d] Channel[%d] write value 0x%X\n",dev
        ,subdev,ch,value);
161   return ME_ERRNO_SUCCESS;
162 }
163
164 static FUNC(sint32, DIO_PRIVATE_CODE) Dio_meFindSubdeviceType(int dev, int
      subdev, int val, int *type, int *subtype)
165 {
166   int rv = 0;
167 #ifdef DIO_MULTIPLE_APP_SUPPORT
168   uint8 u8SharedDev = (uint8)0;
169   uint8 u8SharedSubDev = (uint8)0;
170 #endif
171
172   PRP_DEBUG("%s:%u -- The &type = %d and &subtype = %d\n", __FUNCTION__,
        __LINE__,type,subtype);
173
174 #ifdef DIO_MULTIPLE_APP_SUPPORT
175   if (mHAL_MultiAppSupport() == TRUE)
176   {
177       mHAL_GetDIOResource(((dev*4)+subdev), &u8SharedDev, &u8SharedSubDev);
178       dev=u8SharedDev;
179     subdev=u8SharedSubDev;
180   }
```

```
181 #endif
182
183     if ((rv = meQuerySubdeviceType(dev, subdev, type, subtype) !=
            ME_ERRNO_SUCCESS))
184     {
185         char txtError[1024] = "";
186         meErrorGetMessage(rv, txtError, 1024);
187         PRP_DEBUG("%s(): meQuerySubdeviceType: read port failed error: %s(
                rv=%d)\n", __FUNCTION__, txtError,rv);
188         return rv;
189     }
190     else
191     {
192         PRP_DEBUG("Dio_meFindSubdeviceType:type = %X and subtype = %X.\n",*
                type,*subtype);
193         if (*type == ME_TYPE_DIO && *subtype == ME_SUBTYPE_SINGLE)
194         {
195             //int ifWriteFail = 0;
196             PRP_DEBUG("Inside is successful.\n");
197             rv  = Dio_meWritePort(dev,subdev,0,val);
198             if(rv != 0)
199             {
200                 return rv;
201             }
202             else
203             {
204                 return ME_ERRNO_SUCCESS;
205             }
206         }
207         else
208         {
209             return ME_ERRNO_INVALID_ERROR_NUMBER;
210         }
211     }
212 }
213
214 /*
215  * Global Functions
216  */
217
218 /**DioSim SignatureChanges BEGIN
219  * Signature must be Changed
220  */
221 #define Dio_ReadChannel DioOrigin_ReadChannel
222 /*****************************************************************
223  * Function Name:   Dio_ReadChannel()
224  *
225  * Description:     The function returns the value of specified DIO channel
        .
226  *
227  * Parameter[In]:   ID of DIO Channel
228  *
229  * Return Value:    Dio_LevelType = Value of specified DIO Channel
230  *
231  *****************************************************************/
```

```
232 //#if (DIO_READ_CHANNEL_API == STD_ON)
233 FUNC(Dio_LevelType, DIO_CODE) Dio_ReadChannel(Dio_ChannelType ChannelId)
234 {
235 #if(DIO_DEV_ERROR_DETECT == STD_ON)
236     if (ChannelId > MAX_DIO_CHANNEL_ID)
237     {
238         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID, DIO_READCHANNEL_ID
                , DIO_E_PARAM_INVALID_CHANNEL_ID);
239         return E_NOT_OK;
240     }
241 #endif
242     int val;
243   int dev = DIO_GET_DEVICE_ADDR_FROM_CHID(ChannelId);
244   int subdev = DIO_GET_PORT_ADDR_FROM_CHID(ChannelId);
245   int mask = DIO_GET_CHANNEL_MASK_FROM_CHID(ChannelId);
246
247   Dio_meReadPort(dev,subdev,0,&val);
248
249     if ((val & mask) > 0) {
250
251 #if (STD_ON == Dio_PRP_INFO)
252         PRP_INFO("%s(): Channel(%d)  Dev[%d] SubDev[%d] mask:0x%X read val
                =0x%X => STD_HIGH\n", __FUNCTION__,ChannelId,dev,subdev,mask,
                val);
253 #endif
254         return (Dio_LevelType) STD_HIGH;
255     } else {
256 #if (STD_ON == Dio_PRP_INFO)
257         PRP_INFO("%s(): Channel(%d) Dev[%d] SubDev[%d] mask:0x%X read val=0
                x%X => STD_LOW\n", __FUNCTION__,ChannelId,dev,subdev,mask,val);
258 #endif
259         return (Dio_LevelType) STD_LOW;
260     }
261
262 }
263 //#endif
264
265 #undef Dio_ReadChannel
266 /* DioSim SignatureChanges END */
267
268 /************************************************************************
269  * Function Name:   Dio_WriteChannel()
270  *
271  * Description:     The function sets the specified level for specified DIO
       channel.
272  *
273  * Parameter[In]:   ID of DIO Channel, Value to be written
274  *
275  * Return Value:    None
276  *
277  ************************************************************************/
278 //#if (DIO_WRITE_CHANNEL_API == STD_ON)
279 FUNC(void, DIO_CODE) Dio_WriteChannel(Dio_ChannelType ChannelId,
    Dio_LevelType Level)
280 {
```

```
281 #if(DIO_DEV_ERROR_DETECT == STD_ON)
282     if (ChannelId > MAX_DIO_CHANNEL_ID)
283     {
284         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID,
                DIO_WRITECHANNEL_ID, DIO_E_PARAM_INVALID_CHANNEL_ID);
285         return;
286     }
287 #endif
288   int dev = DIO_GET_DEVICE_ADDR_FROM_CHID(ChannelId);
289   int subdev = DIO_GET_PORT_ADDR_FROM_CHID(ChannelId);
290   int mask = DIO_GET_CHANNEL_MASK_FROM_CHID(ChannelId);
291
292   int val;
293 #ifdef DIO_IO_MIRROR
294   val = lport[dev+subdev];
295 #else
296   Dio_meReadPort(dev,subdev,0,&val);
297 #endif
298
299   if (STD_LOW != Level) {
300     val |= mask;
301 #if (STD_ON == Dio_PRP_INFO)
302     PRP_INFO("%s(): Channel(%d) Dev[%d] SubDev[%d] mask:0x%X Level:0x%X
          write val=0x%X => STD_HIGH\n", __FUNCTION__,ChannelId,dev,subdev,
          mask,Level,val);
303 #endif
304   } else {
305     val &= ~mask;
306 #if (STD_ON == Dio_PRP_INFO)
307     PRP_INFO("%s(): Channel(%d) Dev[%d] SubDev[%d] mask:0x%X Level:0x%X
          write val=0x%X => STD_LOW\n", __FUNCTION__,ChannelId,dev,subdev,
          mask,Level,val);
308 #endif
309   }
310   Dio_meWritePort(dev,subdev,0,val);
311   return;
312 }
313 //#endif
314
315 /************************************************************************
316  * Function Name:   Dio_ReadChannelGroup()
317  *
318  * Description:     The function reads a subset of the adjoining bits of a
       port (channel group).
319  *
320  * Parameter[In]:   Pointer to ChannelGroup
321  *
322  * Return Value:    Dio_PortLevelType = Level of a subset of the adjoining
       bits of a port
323  *
324  ************************************************************************/
325 //#if (DIO_READ_CHL_GROUP_API == STD_ON)
326 FUNC(Dio_PortLevelType, DIO_CODE) Dio_ReadChannelGroup(P2CONST(
      Dio_ChannelGroupType, AUTOMATIC, DIO_CONST) ChannelGroupId)
327 {
```

```
328 #if(DIO_DEV_ERROR_DETECT == STD_ON)
329     if (NULL_PTR==ChannelGroupId)
330     {
331         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID,
                DIO_READCHANNELGROUP_ID, DIO_E_PARAM_POINTER);
332         return E_NOT_OK;
333     }
334 #endif
335     int PortId = ChannelGroupId->port;
336     int dev = DIO_GET_DEVICE_ADDR_FROM_PORTID(PortId);
337     int subdev = DIO_GET_PORT_ADDR_FROM_PORTID(PortId);
338     PRP_DEBUG("ChannelGroupId->port:%d ChannelGroupId->mask:%d
            ChannelGroupId->offset:%d \n",
339             ChannelGroupId->port, ChannelGroupId->mask,ChannelGroupId->
                offset);
340
341     int val;
342   Dio_meReadPort(dev,subdev,0,&val);
343     PRP_DEBUG("%s(): Dev[%d] SubDev[%d] ChannelGroupId->mask:%d
            ChannelGroupId->offset:%d val:%d\n",
344             __FUNCTION__, dev,subdev,ChannelGroupId->mask,ChannelGroupId->
                offset,val);
345     return (Dio_PortLevelType) ((val & ChannelGroupId->mask) >>
            ChannelGroupId->offset);
346 }
347 //#endif
348
349 /************************************************************************
350  * Function Name:   Dio_WriteChannelGroup()
351  *
352  * Description:     The function sets a subset of the adjoining bits of a
        port (channel group)
353  *                  to a specified level.
354  *
355  * Parameter[In]:   Pointer to ChannelGroup, Value to be written
356  *
357  * Return Value:    None
358  *
359  ************************************************************************/
360 //#if (DIO_WRITE_CHL_GROUP_API == STD_ON)
361 FUNC(void, DIO_CODE) Dio_WriteChannelGroup(P2CONST(Dio_ChannelGroupType,
        AUTOMATIC, DIO_CONST) ChannelGroupId, Dio_PortLevelType Level)
362 {
363 #if(DIO_DEV_ERROR_DETECT == STD_ON)
364     if (NULL_PTR==ChannelGroupId)
365     {
366         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID,
                DIO_WRITECHANNELGROUP_ID, DIO_E_PARAM_POINTER);
367         return;
368     }
369 #endif
370     int PortId = ChannelGroupId->port;
371     int dev = DIO_GET_DEVICE_ADDR_FROM_PORTID(PortId);
372     int subdev = DIO_GET_PORT_ADDR_FROM_PORTID(PortId);
373
```

```
374   int val;
375   int valToWrite;
376     int mask = 0xFF;
377
378 #ifdef DIO_IO_MIRROR
379   val = lport[dev+subdev];
380 #else
381   Dio_meReadPort(dev,subdev,0,&val);
382 #endif
383
384   PRP_DEBUG("Value saved in Port val:0x%X\n", val);
385
386     mask = ChannelGroupId->mask & (mask << ChannelGroupId->offset);
387     PRP_DEBUG("mask = ChannelGroupId->mask & (mask << ChannelGroupId->
          offset); mask:0x%X\n", mask);
388   valToWrite=(Level << ChannelGroupId->offset) & mask;
389   PRP_DEBUG("Level << ChannelGroupId->offset) & mask; valToWrite:0x%X\n",
        valToWrite);
390   val = (valToWrite) | (~mask & val);
391   PRP_DEBUG("(valToWrite=0x%X) | (~mask=0x%X & val=0x%X)\n", valToWrite, ~
        mask, val);
392
393   PRP_DEBUG("%s(): Dev[%d] SubDev[%d] ChannelGroupId->mask:0x%X
        ChannelGroupId->offset:0x%X Level:0x%X write ChannelGroup val:0x%X\n"
        ,
394       __FUNCTION__, dev,subdev,ChannelGroupId->mask,ChannelGroupId->offset,
            Level,val);
395   Dio_meWritePort(dev,subdev,0,val);
396   return;
397 }
398 //#endif
399
400 /**********************************************************************
401  * Function Name:   Dio_ReadPort()
402  *
403  * Description:     The function returns the level of all channels of that
        port.
404  *
405  * Parameter[In]:   ID of DIO Port
406  *
407  * Return Value:    Dio_PortLevelType
408  *
409  **********************************************************************/
410 //#if (DIO_READ_PORT_API == STD_ON)
411 FUNC(Dio_PortLevelType, DIO_CODE) Dio_ReadPort(Dio_PortType PortId)
412 {
413 #if(DIO_DEV_ERROR_DETECT == STD_ON)
414     if (PortId > 7 || PortId < 0)
415     {
416         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID, DIO_READPORT_ID,
              DIO_E_PARAM_INVALID_PORT_ID);
417         return E_NOT_OK;
418     }
419 #endif
420   int dev = DIO_GET_DEVICE_ADDR_FROM_PORTID(PortId);
```

```
421   int subdev = DIO_GET_PORT_ADDR_FROM_PORTID(PortId);
422
423   int val;
424   Dio_meReadPort(dev,subdev,0,&val);
425   PRP_DEBUG("%s(): Dev[%d] SubDev[%d] val=0x%X\n", __FUNCTION__,dev,subdev,
          val);
426   return (Dio_PortLevelType) val;
427 }
428 //#endif
429
430 /*********************************************************************
431  * Function Name:   Dio_WritePort()
432  *
433  * Description:     The function sets the specified value for the specified
          port.
434  *
435  * Parameter[In]:   ID of DIO Port, Value to be written
436  *
437  * Return Value:    None
438  *
439  *********************************************************************/
440 //#if (DIO_WRITE_PORT_API == STD_ON)
441 FUNC(void, DIO_CODE) Dio_WritePort(Dio_PortType PortId,Dio_PortLevelType
        Level)
442 {
443 #if(DIO_DEV_ERROR_DETECT == STD_ON)
444     if (PortId > 7 || PortId < 0)
445     {
446         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID, DIO_WRITEPORT_ID,
              DIO_E_PARAM_INVALID_PORT_ID);
447         return;
448     }
449 #endif
450   int dev = DIO_GET_DEVICE_ADDR_FROM_PORTID(PortId);
451   int subdev = DIO_GET_PORT_ADDR_FROM_PORTID(PortId);
452     int val = Level;
453 //  if(STD_HIGH==Level)
454 //  {
455 //    val=0xFF;
456 //  }
457 //  else
458 //  {
459 //    val=0;
460 //  }
461   PRP_DEBUG("%s(): Dev[%d] SubDev[%d] Level=val=0x%X\n", __FUNCTION__,dev,
          subdev,val);
462   Dio_meWritePort(dev,subdev,0,val);
463   return;
464 }
465 //#endif
466
467 /*********************************************************************
468  * Function Name:   Dio_GetVersionInfo()
469  *
```

```
470  * Description:      The function Dio_GetVersionInfo shall return the
         version
471  *                   information of this module. The version information
         includes:
472  *                       - Module Id
473  *                       - Vendor Id
474  *                       - Vendor specific version numbers (BSW00407).
475  *
476  * Parameter[Out]:  Pointer to where to store the version information of
         this module.
477  *
478  * Return Value:    None
479  *
480  ***************************************************************/
481 #if (DIO_VERSION_INFO_API == STD_ON)
482 FUNC(void, DIO_CODE) Dio_GetVersionInfo(P2VAR(Std_VersionInfoType,
        AUTOMATIC, DIO_APPL_DATA) versioninfo)
483 {
484 #if(DIO_DEV_ERROR_DETECT == STD_ON)
485     if (NULL_PTR==versioninfo)
486     {
487         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID,
            DIO_GETVERSIONINFO_ID, DIO_E_PARAM_POINTER);
488         return;
489     }
490 #endif
491     versioninfo->vendorID         = ( DIO_VENDOR_ID );
492     versioninfo->moduleID         = ( DIO_MODULE_ID );
493     versioninfo->sw_major_version = ( DIO_SW_RELEASE_MAJOR_VERSION );
494     versioninfo->sw_minor_version = ( DIO_SW_RELEASE_MINOR_VERSION );
495     versioninfo->sw_patch_version = ( DIO_SW_RELEASE_PATCH_VERSION );
496     return;
497 }
498 #endif
499
500 /*****************************************************************
501  * Function Name:   Dio_Init()
502  *
503  * Description:     The function initializes all global variables of the
         DIO module.
504  *
505  * Parameter:       Pointer to configuration set.
506  *
507  * Return Value:    None
508  *
509  ***************************************************************/
510 FUNC(void, DIO_CODE) Dio_Init(P2CONST(Dio_ConfigType, AUTOMATIC, DIO_CONST)
        ConfigPtr)
511 {
512   PRP_DEBUG("-- entering %s:%u --\n", __FUNCTION__, __LINE__);
513 #if(DIO_DEV_ERROR_DETECT == STD_ON)
514     if (NULL_PTR==ConfigPtr)
515     {
516         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID, DIO_INIT_ID,
            DIO_E_PARAM_CONFIG);
```

```
517         return;
518     }
519 #endif
520     Dio_ConfigPtr = ConfigPtr;
521 }
522
523 /************************************************************************
524  * Function Name:   Dio_FlipChannel()
525  *
526  * Description:     The function inverts the level of the channel and
        returns ineverted level
527  *                  if specified channel is an output channel.
528  *
529  * Parameter[in]:   ID of DIO channel
530  *
531  * Return Value:    Dio_LevelType
532  *
533  ************************************************************************/
534 #if(DIO_FLIPCHANNEL_API == STD_ON)
535 FUNC(Dio_LevelType, DIO_CODE) Dio_FlipChannel(Dio_ChannelType  ChannelId)
536 {
537
538 #if(DIO_DEV_ERROR_DETECT == STD_ON)
539     if (ChannelId > MAX_DIO_CHANNEL_ID)
540     {
541         Det_ReportError (DIO_MODULE_ID, DIO_INSTANCE_ID, DIO_FLIPCHANNEL_ID
            , DIO_E_PARAM_INVALID_CHANNEL_ID);
542         return E_NOT_OK;
543     }
544 #endif
545
546     int rv;
547     int dev = DIO_GET_DEVICE_ADDR_FROM_CHID(ChannelId);
548     PRP_DEBUG("dev = %d\n",dev);
549     int subdev = DIO_GET_PORT_ADDR_FROM_CHID(ChannelId);
550     PRP_DEBUG("subdev = %d\n",subdev);
551     int mask = DIO_GET_CHANNEL_MASK_FROM_CHID(ChannelId);
552     int type = 0;
553     int subtype = 0;
554     int val = 0;
555
556     Dio_meReadPort(dev,subdev,0,&val);
557     //PRP_DEBUG("%s:%u -- The &type = %d and &subtype = %d\n", __FUNCTION__
        , __LINE__,&type,&subtype);
558     rv = Dio_meFindSubdeviceType(dev, subdev, val, &type, &subtype);
559     //PRP_DEBUG("rv = %d\n",rv);
560     if( rv != 0)
561     {
562         Dio_meReadPort(dev,subdev,0,&val);
563         if ((val & mask) > 0)
564         {
565             PRP_DEBUG("%s(): Dev[%d] SubDev[%d] mask:0x%X read val=0x%X =>
                STD_HIGH\n", __FUNCTION__,dev,subdev,mask,val);
566             return (Dio_LevelType) STD_HIGH;
567         }
```

```
568         else
569         {
570             PRP_DEBUG("%s(): Dev[%d] SubDev[%d] mask:0x%X read val=0x%X =>
                STD_LOW\n", __FUNCTION__,dev,subdev,mask,val);
571             return (Dio_LevelType) STD_LOW;
572         }
573         PRP_DEBUG("Channel is configured as Input...So can not be flipped
                ...returns existing Level.\n");
574
575     }
576     else
577     {
578         Dio_meReadPort(dev,subdev,0,&val);
579         PRP_DEBUG("Inside else of the FlipChannel\n");
580         int check = (val & mask);
581         PRP_DEBUG("check = %d\n",check);
582         if (check > 0)
583         {
584             //val |= mask;
585             val &= ~mask;
586             Dio_meWritePort(dev,subdev,0,val);
587             PRP_DEBUG("%s(): Dev[%d] SubDev[%d] mask:0x%X Existing Level:
                    STD_HIGH write val=0x%X => STD_LOW\n", __FUNCTION__,dev,
                    subdev,mask,val);
588             return (Dio_LevelType) STD_LOW;
589
590         }
591         else
592         {
593             //val &= ~mask;
594             val |= mask;
595             Dio_meWritePort(dev,subdev,0,val);
596             PRP_DEBUG("%s(): Dev[%d] SubDev[%d] mask:0x%X Existing Level:
                    STD_LOW write val=0x%X => STD_HIGH\n", __FUNCTION__,dev,
                    subdev,mask,val);
597             return (Dio_LevelType) STD_HIGH;
598
599         }
600
601      }
602 }
603 #endif
604
605 /* DioSim Additions BEGIN */
606 #ifdef Dio_ReadChannel
607 #error Dio_ReadChannel in Dio.c still defined!
608 #endif
609 FUNC(Dio_LevelType, DIO_CODE) Dio_ReadChannel(Dio_ChannelType ChannelId){
610   if (DIOSIM_SIMULATION_ENABLED == DioSim_GetSimState()){
611     Dio_LevelType ReturnValue;
612     DioSim_ReturnType ReturnCode = DioSim_GetSim(ChannelId, &ReturnValue);
613     if (DIOSIM_SUCC_CHANNEL_FOUND == ReturnCode){
614       return ReturnValue;
615     }
616   }else if (DIOSIM_SIMULATION_LEARNING == DioSim_GetSimState()){
```

```
617     /* TODO SEND THE VALUE TO THE DIAG OR ANYWHERE VIA CAN OUT */
618   }
619   /** Cases:
620    * - Simulation Disabled
621    * - No SimulationValue for ChannelId
622    * - Simulation Learning sent Message
623    */
624   return DioOrigin_ReadChannel(ChannelId);
625 }
626 /* DioSim Additions END */
```

C.1.2. Source Code DIO Sim - DioSim.c

```
1  /*
2   * DioSim.c
3   *
4   *  Created on: 27.04.2015
5   *      Author: q358209, q394119
6   */
7
8  #include "DioSim.h"
9  #include <stdlib.h>
10
11
12 static DioSim_Entry DioSim_Values[DIOSIM_MAXVALUES];
13
14 /**
15  * TODO: Define Datatype for DioSim_ValuesCount in AUTOSAR as it is defined
        in CONST!
16  */
17 static int DioSim_ValuesCount = 0;
18 static DioSim_SimulationState DioSim_Enabled = DIOSIM_SIMULATION_DISABLED;
19
20 FUNC(void, DIO_CODE) DioSim_UpdateSim(Dio_ChannelType ChannelId,
      Dio_LevelType Value){
21   int ArrayIndex;
22   DioSim_ReturnType ReturnCode = DioSim_GetIndex(ChannelId, &ArrayIndex);
23   if (DIOSIM_SUCC_CHANNEL_FOUND == ReturnCode){
24     //We have the correct Ptr in the ArrayEntry.
25     //Replace the value!
26     DioSim_Values[ArrayIndex].Value = Value;
27   }else if(DIOSIM_ERR_CHANNEL_NOT_FOUND == ReturnCode){
28     //No entry found! We have to insert it at the given position
29     //We have the position where it has to be inserted
30     //Look if the valuecount is already at its limit
31     if (DIOSIM_MAXVALUES <= DioSim_ValuesCount + 1){
32       //implicite devensive programming... no check needed for out of
           bounds
33       //If there are as much or more (only as much could happen)
34       //just return the insertionerror const
35       //FIXME VOID-Method! BUT WE HAVE TO RETURN AN ERROR!
36       //return DIOSIM_ERROR_ADD_WOULD_EXCEED;
```

```
37      }
38      //So there is space for one more value...
39      //Add it!
40      //To do so shift (or try to) the elements first to the right
41      DioSim_ShiftRight(ArrayIndex);
42      //Now we can write it at ArrayIndex
43      DioSim_Values[ArrayIndex].ChannelId = ChannelId;
44      DioSim_Values[ArrayIndex].Value = Value;
45      DioSim_ValuesCount++;
46    }
47  }
48
49  FUNC(void, DIO_CODE) DioSim_RemoveSim(Dio_ChannelType ChannelId){
50    if (DioSim_ValuesCount==0){
51      //FIXME ERROR EMPTY LIST!
52      //return DIOSIM_ERROR_REMOVE_LIST_EMPTY;
53    }
54    int ArrayIndex;
55    DioSim_ReturnType ReturnCode = DioSim_GetIndex(ChannelId, &ArrayIndex);
56    if (DIOSIM_SUCC_CHANNEL_FOUND == ReturnCode){
57      //We found the Index, so just shift to the left and we got it!
58      DioSim_ShiftLeft(ArrayIndex);
59      DioSim_ValuesCount--;
60    }else if(DIOSIM_ERR_CHANNEL_NOT_FOUND == ReturnCode){
61      //This is an error!
62      //FIXME WE HAVE TO RETURN AN ERROR
63      //return DIOSIM_ERROR_REMOVE_NOT_FOUND;
64    }
65  }
66
67  FUNC(void, DIO_CODE) DioSim_FlushSim(void){
68    //we do not really delete the values and overwrite them with 0...
69    //We only set the value to zero
70    DioSim_ValuesCount=0;
71  }
72
73  /**
74   * TODO: RETURN VALUE NEEDS TO BE DEFINED IN AUTOSAR
75   */
76  FUNC(DioSim_ReturnType, DIO_CODE) DioSim_GetSim(Dio_ChannelType ChannelId,
       Dio_LevelType *Value){
77    int ArrayIndex;
78    DioSim_ReturnType ReturnCode = DioSim_GetIndex(ChannelId, &ArrayIndex);
79    if (DIOSIM_SUCC_CHANNEL_FOUND == ReturnCode){
80      *Value = DioSim_Values[ArrayIndex].Value;
81    }
82    return ReturnCode;
83  }
84
85  FUNC(void, DIO_CODE) DioSim_EnableSimulation(DioSim_SimulationState enable)
       {
86  #if (SIMULATION_MODULE_ACTIVE == STD_OFF)
87    DioSim_Enabled = DIOSIM_SIMULATION_DISABLED;
88    PRP_INFO(RED("DioSim_EnableSimulation - Disable simulation due to
         deactivated sim module \n"));
```

```
89 #else
90   DioSim_Enabled = enable;
91 #endif
92 }
93
94 FUNC(DioSim_SimulationState, DIO_CODE) DioSim_GetSimState(void){
95 #if (SIMULATION_MODULE_ACTIVE == STD_OFF)
96   return DIOSIM_SIMULATION_DISABLED;
97 #else
98   return DioSim_Enabled;
99 #endif
100 }
101
102 /** LOCAL FUNCTIONS **/
103
104 /**
105  * This local function searches the ChannelId in the DioSim_Values Array
       and stores the ArrayIndex by Ref of the searched Index
106  * Returns:
107  * - DIOSIM_SUCCESS and the correct set ArrayIndex if ChannelId was found
108  * - DIOSIM_NOTFOUND and the ArrayIndex where it has to be inserted
109  * TODO: Define Datatype for DioSim_ValuesCount and use it as param for
       ArrayIndex in AUTOSAR as it is defined in CONST!
110  * TODO: Analyze how indizes are searched and replace it by better search
       algorithms (e.g. Binary-Search or implement Hash-map)
111  */
112 DioSim_ReturnType DioSim_GetIndex(Dio_ChannelType ChannelId, int *
      ArrayIndex){
113   int i=0;
114   //We have 3 cases
115   //1. The searched one is being found
116   //|--> ez, just return it...
117   //2. the searched one is not on one of the peeked ones but smaller than
        the i'one
118   //|--> return the i position to shift right
119   //3. the searched one is not found and the while block exceeded and
        bigger than the i'one
120   //|-- the searched one is not found in the whole while block means it
        must be bigger than any i one
121   //|--> return the DioSim_ValuesCount as Index
122   //One special case is if the list is empty. Because of the while it will
        end in case 3
123   while(i<DioSim_ValuesCount){
124     //Case 1.
125     if (ChannelId == DioSim_Values[i].ChannelId){
126       //We found it!
127       //Set the arrayindex
128       *ArrayIndex = i;
129       return DIOSIM_SUCC_CHANNEL_FOUND;
130     //Case 2.
131     //is the searched ChannelId smaller than the [i] one?
132     //Than it must be inserted at position i!
133     }else if(ChannelId < DioSim_Values[i].ChannelId){
134       *ArrayIndex = i;
135       return DIOSIM_ERR_CHANNEL_NOT_FOUND;
```

```
136     }
137     i++;
138   }
139   //Case 3 - Return the last index
140   *ArrayIndex = DioSim_ValuesCount;
141   return DIOSIM_ERR_CHANNEL_NOT_FOUND;
142 }
143
144 /**
145  * This local function shifts all elements to the right so a new element
        fits at the given index.
146  */
147 void DioSim_ShiftRight(int ArrayIndex){
148   //It is easy.
149   //count backwards
150   //Exit after we moved the ArrayIndex
151   int toShift=DioSim_ValuesCount; //How many do we have to shift at most?
152
153   //Cases:
154   //1. Array is empty
155   //|--> shift nothing
156   //2. ArrayIndex is out of bounds
157   //|--> shift nothing
158   //3. ArrayIndex is in the middle of the field
159   //|--> shift from the beginning of the right (valuecount) until we match
        the arrayindex
160   while (toShift > 0 && ArrayIndex < DioSim_ValuesCount){
161     DioSim_Values[toShift].ChannelId  = DioSim_Values[toShift-1].ChannelId;
162     DioSim_Values[toShift].Value    = DioSim_Values[toShift-1].Value;
163     if (toShift-1 == ArrayIndex){
164       toShift = 0;
165     }else{
166       --toShift;
167     }
168   }
169 }
170
171 /**
172  * This local function shifts all elements to the left so the given
        ArrayEntry is overwritten
173  */
174 void DioSim_ShiftLeft(int ArrayIndex){
175   //DioSim_ValuesCount is defenetley > 0!
176   int actIndex = ArrayIndex;
177   while (actIndex < DioSim_ValuesCount-1){
178     DioSim_Values[actIndex].ChannelId = DioSim_Values[actIndex+1].ChannelId
          ;
179     DioSim_Values[actIndex].Value = DioSim_Values[actIndex+1].Value;
180     ++actIndex;
181   }
182 }
```

C.1.3. Source Code HwIoAb - HwIoAb_DIO.c

```
#include "HwIoAb_DIO.h"

static ImpDTuint16 HwIo_Ab_DIO_ZBE_RotationCount = ROTATION_MIDDLE;
static ImpDTuint16 HwIo_Ab_DIO_ZBE_RotationOverflow = 0u;

FUNC(void, HwIoAb_DIO_CODE) HwIoAb_DIO_ReadZBEValues_rbl(P2VAR(ImpDTuint16,
     AUTOMATIC, RTE_APPL_DATA) RotationCount,
    P2VAR(ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) DeflectionWest, P2VAR(
        ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) DeflectionEast,
    P2VAR(ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) DeflectionNorth, P2VAR(
        ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) DeflectionSouth,
    P2VAR(ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnMain, P2VAR(
        ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnMenu,
    P2VAR(ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnBack, P2VAR(
        ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnOption,
    P2VAR(ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnMedia, P2VAR(
        ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnMap,
    P2VAR(ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnCom, P2VAR(
        ImpDTBoolean, AUTOMATIC, RTE_APPL_DATA) BtnNav,
    P2VAR(ImpDTuint2, AUTOMATIC, RTE_APPL_DATA) RotationOverflow){

  /* Use the static rotatoinvariables to update the values */
  *RotationCount    = HwIo_Ab_DIO_ZBE_RotationCount;
  *RotationOverflow = HwIo_Ab_DIO_ZBE_RotationOverflow;

  /* is the overflow now noticed? then reset it */
  if (OVERFLOW_NO != HwIo_Ab_DIO_ZBE_RotationOverflow){
    HwIo_Ab_DIO_ZBE_RotationOverflow = OVERFLOW_NO;
  }

  /* Read all values */
  /* Read the Deflection right of the Lightsensors */
  *DeflectionWest   = Dio_ReadChannel(DioChannel_C5_Deflection_West);
  *DeflectionEast   = Dio_ReadChannel(DioChannel_C3_Deflection_East);
  *DeflectionNorth  = Dio_ReadChannel(DioChannel_C2_Deflection_North);
  *DeflectionSouth  = Dio_ReadChannel(DioChannel_C4_Deflection_South);

  /* Read the Buttonstates */
  *BtnMain          = Dio_ReadChannel(DioChannel_D0_BtnMain);
  *BtnMenu          = Dio_ReadChannel(DioChannel_D1_BtnMenu);
  *BtnBack          = Dio_ReadChannel(DioChannel_D2_BtnBack);
  *BtnOption        = Dio_ReadChannel(DioChannel_D3_BtnOption);
  *BtnMedia         = Dio_ReadChannel(DioChannel_D4_BtnMedia);
  *BtnMap           = Dio_ReadChannel(DioChannel_D5_BtnMap);
  *BtnCom           = Dio_ReadChannel(DioChannel_D6_BtnCom);
  *BtnNav           = Dio_ReadChannel(DioChannel_D7_BtnNav);
}

FUNC(void, HwIoAb_DIO_CODE) HwIoAb_DIO_ResetRotationCount_rbl() {
  HwIo_Ab_DIO_ZBE_RotationCount = ROTATION_MIDDLE;
  HwIo_Ab_DIO_ZBE_RotationOverflow = 0u;
```

```
45   PRP_DEBUG(YELLOW("HwIoAb_DIO_ResetRotationCount_rbl - rotation count was
         reset\n"));
46 }
47
48 FUNC(void, HwIoAb_DIO_CODE) HwIoAb_DIO_ReadZBERotation_cyclic(void){
49
50   /** Initializing the variables with its real states is not allowed
         because static init must be constant
51    * This circumstance could lead to one cursorjump after this part booted
52    * But it would be nearly impossible because the old state is 00 and if
          the new state is 11 then no action happens
53    */
54   static ImpDTBoolean oldRotation_LSL = FALSE;
55   static ImpDTBoolean oldRotation_LSR = FALSE;
56
57   static ImpDTBoolean LeftRotIncoming = FALSE; /* buffers the recent LS
         hits */
58   static ImpDTBoolean RightRotIncoming = FALSE; /* because we need to see
         if Right rotation also happened */
59
60   /** TODO Check for Wakeup handle and call
         Rte_Call_HwIoAb_DIO_CSIfZBE_Wake_CSOpWakeup if happened
61    * But do not only check if the Button gets pressed! Also check if the
          acutal State is Sleep!!
62    * else if the State is Awake then check if any Rotation happened
63    * TODO insert check on States
64    */
65
66   /* We have to look which LightBarrier gets changed first */
67   ImpDTBoolean actRotation_LSL = Dio_ReadChannel(DioChannel_C0_Rotation_LSL
         );
68   ImpDTBoolean actRotation_LSR = Dio_ReadChannel(DioChannel_C1_Rotation_LSR
         );
69
70   /** But What can happen with two Sensors
71    *
72    * In Cycle 1 we see following states
73    * State 0 LSL followed by 1 while LSR remains same --> left rotation
          incoming (wait for LSR change)
74    * State 0 LSL followed by 0 while LSR remains same --> nothing happened
          or too fast
75    * State 1 LSL followed by 0 while LSR remains same --> left rotation
          incoming (wait for LSR change)
76    * State 1 LSL followed by 1 while LSR remains same --> nothing happened
          or too furious
77    *
78    * So the important states can be reduced to
79    * LSL changed and LSR remains the same --> left rotation incoming
80    * all other states do not remain in statechanges
81    *
82    * In Cycle 2 we see following states
83    * State 1 LSL and LSR 0 followed by 1 --> left rotation done (LSR is now
           LSL) (clear incomingflags)
84    * State 1 LSL and LSR 1 followed by 0 --> right rotation incoming (LSR
          is unequal LSL) (cycle 1 analog)
```

```
85    * State 0 LSL and LSR 0 followed by 1 --> right rotation incoming (LSR
          is unequal LSL) (cycle 1 analog)
86    * State 0 LSL and LSR 1 followed by 0 --> left rotation done (LSR is now
          LSL) (clear incomingflags)
87    *
88    * So the important states can be reduced to
89    * LSL remains the same and LSR changed to LSL --> left rotation done if
          flag set otherwise not valid
90    *
91    * all other states do not produce statechanges
92    *
93    */
94   if (actRotation_LSL != oldRotation_LSL && actRotation_LSR ==
          oldRotation_LSR){ /* Cycle 1*/
95     /* the if-statements need to be read bottom-up */
96
97     /* lets see if LSL changed to LSR and if the RightrotIncFlag is true */
98     if (actRotation_LSL == actRotation_LSR && TRUE == RightRotIncoming){
99       /* first clear the flags */
100      LeftRotIncoming = FALSE;
101      RightRotIncoming = FALSE;
102
103      /* one Step to the Right */
104      /* but check for overflow */
105      if (ROTATION_OVERFLOW == HwIo_Ab_DIO_ZBE_RotationCount){ /* caution!
             Overflow is NOT 655335!! */
106        HwIo_Ab_DIO_ZBE_RotationCount = ROTATION_MIDDLE; /* set it to the
               middle */
107        HwIo_Ab_DIO_ZBE_RotationOverflow = OVERFLOW_YES; /* and set the
               overflow */
108      }else{
109        HwIo_Ab_DIO_ZBE_RotationCount = HwIo_Ab_DIO_ZBE_RotationCount + 1;
110      }
111
112    }else if(actRotation_LSL == actRotation_LSR && FALSE ==
          RightRotIncoming){
113
114      /* we have to check if the rotation is only hit by the left but not
             the right and it falls back into middle pos */
115      /* if so just delete both flags! */
116      LeftRotIncoming = FALSE;
117      RightRotIncoming = FALSE;
118      PRP_INFO(RED("HwIo_Ab_DIO_ReadZBERotation_cyclic - no valid rotation
             - reset flags\n"));
119    }else if (actRotation_LSL != actRotation_LSR){
120
121      /* set the left rotation incoming flag if now LSL != LSR */
122      LeftRotIncoming = TRUE;
123      RightRotIncoming = FALSE;
124    }
125
126    /* else remains no change */
127
128  }else if (actRotation_LSL == oldRotation_LSL && actRotation_LSR !=
          oldRotation_LSR){
```

```
129     /* the if-statements need to be read bottom-up */
130
131     /* lets see if LSR changed to LSL and if the LeftRotIncFlag is true */
132     if (actRotation_LSL == actRotation_LSR && TRUE == LeftRotIncoming){
133
134       /* then we have our rotation done */
135       LeftRotIncoming = FALSE;
136       RightRotIncoming = FALSE;
137
138       /* but first check if a underrun is caused */
139       if (ROTATION_UNDERRUN == HwIo_Ab_DIO_ZBE_RotationCount){
140         HwIo_Ab_DIO_ZBE_RotationCount = ROTATION_MIDDLE; /* set it to the
              middle */
141         HwIo_Ab_DIO_ZBE_RotationOverflow = OVERFLOW_YES; /* and set the
              overflow */
142       }else{
143         HwIo_Ab_DIO_ZBE_RotationCount = HwIo_Ab_DIO_ZBE_RotationCount - 1;
144       }
145
146     }else if(actRotation_LSL == actRotation_LSR && FALSE == LeftRotIncoming
          ){
147
148       /* if so just delete both flags! */
149       LeftRotIncoming = FALSE;
150       RightRotIncoming = FALSE;
151       PRP_INFO(RED("HwIo_Ab_DIO_ReadZBERotation_cyclic - no valid rotation
            - reset flags\n"));
152     }else if (actRotation_LSL != actRotation_LSR){
153
154       /* set the right rotation incoming flag if now LSL != LSR */
155       LeftRotIncoming = FALSE;
156       RightRotIncoming = TRUE;
157     }
158
159     /* else remains no change */
160
161   }
162   /* else case is: both remained the same AND if we spinned too fast (or
        too furious) */
163
164   /* in any way update the last values for the next cycle */
165   oldRotation_LSL = actRotation_LSL;
166   oldRotation_LSR = actRotation_LSR;
167 }
```

C.1.4. Source Code simGW - `simGW.c`

```
1 #include "Rte_SimGw.h"
2
3 #include "DioSim.h"
4
5 FUNC(void, SimGw_CODE) sym_SimGw_cyclic(void){
```

```
  /** Here we can implement the learning algorithm */
}

FUNC(void, SimGw_CODE) sym_SimGw_SwitchSimulation(ImpDTBoolean enable){
  if (DIOSIM_SIMULATION_ENABLED == enable){
    PRP_INFO(RED("SimGw_EnableSimulation - Enabling Simulation\n"));
  }else if(DIOSIM_SIMULATION_LEARNING == enable){
    PRP_INFO(RED("SimGw_EnableSimulation - Learning activated\n"));
  }else{
    PRP_INFO(RED("SimGw_EnableSimulation - Disabling Simulation\n"));
  }
  DioSim_EnableSimulation(enable);
}

/*
 * Adds the Channeldata to the simlist and stores the simulated value
 * Refreshes the Simlist @ Dio.c
 * FUTUREWORK: Store splines for the by time values
 */
FUNC(void, SimGw_CODE) sym_SimGw_AddSim(ImpDTuint16 ChanId, ImpDTBoolean
    Value){
  PRP_INFO(RED("SimGw_AddSimChan - called with %u and %u\n"), ChanId, Value
      );
  DioSim_UpdateSim(ChanId, Value);
}

/*
 * Removes the Channeldata from the simlist.
 * Refreshes the Simlist @ Dio.c
 */
FUNC(void, SimGw_CODE) sym_SimGw_RemoveSim(ImpDTuint16 ChanId){
  PRP_INFO(RED("SimGw_RemoveSimChan - called with %u\n"), ChanId);
  DioSim_RemoveSim(ChanId);
}

FUNC(void, SimGw_CODE) sym_SimGw_FlushSim(void){
  PRP_INFO(RED("SimGw_FlushSim - called\n"));
  DioSim_FlushSim();
}
```

C.2. Application SWC - simAgent

```
#include "SimAgent.h"
#include <time.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>


static SimAgent_Testcase_Entry SimAgent_Testcase[SIMAGENT_MAXVALUES]; //
    how much data can one testcase handle
```

```
9  static ImpDTuint16 SimAgent_Testcase_ValuesCount = 0; // this is the actual
       count of testdata
10 static ImpDTuint16 SimAgent_Testcase_Counter = 0; // this is the actual
     progress in the testcase
11 static ImpDTBoolean SimAgent_ReadyToStart = SimAgent_ReadyToStart_NOT_READY
     ; // this flag signals the start of the simulation in the next
     sym_SimAgent_Starter_cyclic invocation
12 static ImpDTuint2 SimAgent_StopOrGo = SimAgent_StopOrGo_STOP; // this is
     the acutal state of the simulation
13 static ImpDTuint32 SimAgent_Cycle_Counter = 0u; // Represents the time
     since Teststart in ms
14
15 #if (SIMULATION_PRINT_TIMESTAMPS == STD_ON)
16 FUNC(void, SimAgent_CODE) getSubSecondTimeString(char *timeString, int size
     ) {
17   struct tm *tm;
18   struct timeval tv;
19   char format[64];
20
21   gettimeofday(&tv, NULL);
22   tm = localtime(&tv.tv_sec);
23
24   strftime(format, 64, "%Y-%m-%d %H:%M:%S.%%06u", tm);
25     snprintf(timeString, size, format, tv.tv_usec);
26 }
27 #endif //SIMULATION_PRINT_TIMESTAMPS
28
29 FUNC(void, SimAgent_CODE) startSimulation(void){
30   // Added counter resets so that multiple test runs can be executed
       without reinitializing (q394119)
31   SimAgent_Cycle_Counter = 0;
32   SimAgent_Testcase_Counter = 0u;
33   SimAgent_ReadyToStart = SimAgent_ReadyToStart_NOT_READY;
34   SimAgent_StopOrGo = SimAgent_StopOrGo_GO;
35   Rte_Call_CSControlSimulation_CSOpEnableSimulation(TRUE);
36 }
37
38 /* sym_SimAgent_UpdateCDD_cyclic is triggered every 10ms */
39 FUNC(void, SimAgent_CODE) sym_SimAgent_UpdateCDD_cyclic(void){
40   // TODO define Constants in AUTOSAR for StopOrGo
41   if (SimAgent_StopOrGo_GO==SimAgent_StopOrGo){
42     // We are in GO! let the cycle_counter run
43     // the time is uint32 encoded with the baseunit milliseconds
44     // add only the updatesequence so we keep the same time base in
         milliseconds
45     SimAgent_Cycle_Counter = SimAgent_Cycle_Counter +
         RTE_TimingEventPeriodMS_TE_SimAgent_UpdateCDD_cyclic;
46     // Check if the time of the current SimAgent_Testcase_Counter is now
         smaller than the Cycle_Counter
47     // do the Add or Remove Sequence while the Time of the incrementing
         testdata is smaller
48     // With this instruction we can simulate channels simultaniously
49     // TODO refactor while condition if we have abort criteria for
         Simulationend
50     while(SimAgent_Testcase_Counter < SimAgent_Testcase_ValuesCount
```

```
51          && SimAgent_Testcase[SimAgent_Testcase_Counter].StartOffset <
                SimAgent_Cycle_Counter){
52        //check if we have to add the simDio or remove the simDio for the
              given ChannelId
53        if (2u > SimAgent_Testcase[SimAgent_Testcase_Counter].Value){
54          // 0 or 1 is binary value of the dio channel that needs to be
                simulated...
55          Rte_Call_CSChangeSimEntryDio_CSOpAddSimDio(
56              SimAgent_Testcase[SimAgent_Testcase_Counter].ChannelId,
57              SimAgent_Testcase[SimAgent_Testcase_Counter].Value);
58        }else if (2u == SimAgent_Testcase[SimAgent_Testcase_Counter].Value){
59          //2 means remove the channel from simulation (bin 11)
60          Rte_Call_CSChangeSimEntryDio_CSOpRemoveSimDio(SimAgent_Testcase[
                SimAgent_Testcase_Counter].ChannelId);
61        }else{
62          // 3 means learn
63          // TODO - refactoring because we do not make progress in the
                testcase, or do we? - question of concept!
64        }
65        // We have progress in the testcase the value, so increment it!
66        SimAgent_Testcase_Counter = SimAgent_Testcase_Counter + 1;
67      }
68    }
69  }
70
71  /* sym_SimAgent_Starter_cyclic is triggered every 60ms */
72  FUNC(void, SimAgent_CODE) sym_SimAgent_Starter_cyclic(void){
73    if (SimAgent_ReadyToStart_READY == SimAgent_ReadyToStart) {
74      startSimulation();
75      PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_Starter_cyclic - Started
            Simulation");
76    }
77  }
78
79  /**
80   * This method reads CAN-Messages and stores changes into local variables
81   */
82  FUNC(void, SimAgent_CODE) sym_SimAgent_InitHandler_rbl(void){
83    ImpDTBoolean init = FALSE;  // is the init flag set?
84    ImpDTEcuId_Type ecuId = 0u;
85    ImpDTBoolean discover = FALSE; //is discover active? so yes, then
          initialize yourself and send your ECU_id
86
87    ImpDTRxInit *data = Rte_IRead_SimAgent_InitHandler_rbl_RxInit_RxInitData
          ();
88
89    ecuId = data->ecuId;
90    init = data->initialization;
91    discover = data->discover;
92
93    PRP_DEBUG(GREEN("SimAgent_InitHandler_rbl - EcuId %u, Discover %u, Init %
          u\n"), ecuId, discover, init);
94
95    if (ECU_ID != data->ecuId && BROADCAST_ECU_ID != data->ecuId){
```

```
96      PRP_INFO(GREEN("SimAgent_InitHandler_rbl - EcuId %u was unequal
            required %u\n"),ecuId, ECU_ID);
97      return;
98    }
99
100   if (TRUE == init){
101
102     // initialize SimAgent
103     SimAgent_Testcase_ValuesCount = 0u;
104     SimAgent_Testcase_Counter = 0u;
105     SimAgent_Cycle_Counter = 0;
106
107     // initialize SimGw
108     Rte_Call_CSChangeSimEntryDio_CSOpRemoveAllEntries();
109     Rte_Call_CSControlSimulation_CSOpEnableSimulation(FALSE);
110     SimAgent_StopOrGo = SimAgent_StopOrGo_STOP;
111     SimAgent_ReadyToStart = 0u;
112
113     if (BROADCAST_ECU_ID == ecuId) {
114
115       PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_InitHandler_rbl -
              Initialized SimAgent from broadcast");
116 #if (SIMULATION_TRIGGER == STD_ON)
117       system("/home/vap/MEiDS-1.3.64/examples/di_test 1150 &");
118       PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_InitHandler_rbl - Trigger
              sent");
119 #endif //SIMULATION_TRIGGER
120     } else {
121       PRP_INFO(GREEN("SimAgent_InitHandler_rbl - Initialized SimAgent\n"));
122     }
123   }
124   if (TRUE == discover && BROADCAST_ECU_ID != ecuId){
125     // Send an Update with your ECU_ID via CAN
126     ImpDTTxInitDiscovery initDiscoveryData;
127     initDiscoveryData.ecuId = ECU_ID;
128
129     PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_InitHandler_rbl - Send init
            discovery data: ecuId %u", initDiscoveryData.ecuId);
130     Rte_Write_TxInitDiscovery_TxInitDiscoveryData(&initDiscoveryData);
131   }
132 }
133
134 FUNC(void, SimAgent_CODE) sym_SimAgent_DataHandler_rbl(void){
135   ImpDTuint8 ecuId = 0x0;
136   ImpDTuint16 channelId = 0x0;
137   ImpDTuint2 value = 0x0;
138   ImpDTuint32 startOffset = 0x0;
139
140   ImpDTRxDataDio *data =
          Rte_IRead_SimAgent_DataHandler_rbl_RxDataDio_RxData();
141
142   ecuId = data->ecuId;
143   channelId = data->channelId;
144   value = data->value;
145   startOffset = data->startOffset;
```

```
146
147   PRP_DEBUG(GREEN("SimAgent_DataHandler_rbl - EcuId %u, Channel %u, Value %
          u, StartOffset %u\n"), ecuId, channelId, value, startOffset);
148
149   if (ECU_ID == data->ecuId){
150     PRP_INFO(GREEN("SimAgent_DataHandler_rbl - Processing Data for this ECU
            \n"));
151 #if (SIMULATION_MODULE_ACTIVE == STD_ON)
152     SimAgent_AddTestcase_Entry(channelId, value, startOffset);
153 #endif
154   }
155 }
156
157 FUNC(void, SimAgent_CODE) sym_SimAgent_RunHandler_rbl(void){
158   ImpDTuint2 stopOrGo = 0x0;
159   ImpDTBoolean verbose = FALSE; // is the verbose flag set?
160   ImpDTuint32 offset = 0u;
161   ImpDTuint8 verboseEcuId = 0x0;
162
163   ImpDTRxRun *data = Rte_IRead_SimAgent_RunHandler_rbl_RxRun_RxRunData();
164
165   stopOrGo = data->stopOrGo;
166   verbose = data->verbose;
167   offset = data->offset;
168   verboseEcuId = data->verboseEcuId;
169
170   PRP_DEBUG(GREEN("SimAgent_RunHandler_rbl - StopOrGo %u, Verbose %u,
          Offset %u, VerboseEcuId %u\n"), SimAgent_StopOrGo, verbose, offset,
          verboseEcuId);
171
172   if (1u == verbose) {
173     if (ECU_ID == verboseEcuId) {
174       ImpDTTxVerbose verboseData;
175       verboseData.ecuId = ECU_ID;
176       verboseData.testLength = SimAgent_Testcase_ValuesCount;
177       if (SimAgent_Testcase_ValuesCount == 0) {
178         verboseData.lastSimulatedChannel = 0x0;
179         verboseData.lastSimulatedValue = 0x0;
180         verboseData.simulationDone = 0x0;
181       } else {
182         verboseData.lastSimulatedChannel = SimAgent_Testcase[
              SimAgent_Testcase_Counter - 1].ChannelId;
183         verboseData.lastSimulatedValue = SimAgent_Testcase[
              SimAgent_Testcase_Counter - 1].Value;
184         verboseData.simulationDone = SimAgent_Testcase_ValuesCount ==
              SimAgent_Testcase_Counter;
185       }
186       PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_RunHandler_rbl - Send
            verbose data: test length %u, last simulated channel %u, last
            simulated value %u, simulation done %u",
187         verboseData.testLength, verboseData.lastSimulatedChannel,
              verboseData.lastSimulatedValue, verboseData.simulationDone);
188       Rte_Write_SimAgent_TxVerbose_TxVerboseData(&verboseData);
189     } else {
```

```
190       PRP_INFO(GREEN("SimAgent_RunHandler_rbl - EcuId %u was unequal
              required %u\n"), verboseEcuId, ECU_ID);
191     }
192   } else {
193     if (SimAgent_StopOrGo_STOP == stopOrGo){
194
195       SimAgent_StopOrGo = SimAgent_StopOrGo_STOP;
196       SimAgent_ReadyToStart = SimAgent_ReadyToStart_NOT_READY;
197       Rte_Call_CSControlSimulation_CSOpEnableSimulation(FALSE);
198
199       PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_RunHandler_rbl - Stopped
              Simulation");
200     } else if (SimAgent_StopOrGo_GO == stopOrGo){
201
202       startSimulation();
203       PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_RunHandler_rbl - Started
              Simulation");
204     } else if(SimAgent_StopOrGo_LEARN == stopOrGo) {
205
206       SimAgent_StopOrGo = SimAgent_StopOrGo_STOP;
207       SimAgent_ReadyToStart = SimAgent_ReadyToStart_NOT_READY;
208       Rte_Call_CSControlSimulation_CSOpEnableSimulation(FALSE);
209
210       PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_RunHandler_rbl - LEARNING
              MODE NOT IMPLEMENTED YET! Stopped Simulation");
211     } else if(SimAgent_StopOrGo_GO_SYNC == stopOrGo) {
212
213       SimAgent_ReadyToStart = SimAgent_ReadyToStart_READY;
214       PRP_TIMESTAMP(PRP_INFO, GREEN, "SimAgent_RunHandler_rbl - Set
              ReadyToStart flag");
215     }
216   }
217 }
218
219 void SimAgent_AddTestcase_Entry(ImpDTuint16 ChannelId, ImpDTuint2 Value,
       ImpDTuint32 StartOffset){
220   SimAgent_Testcase[SimAgent_Testcase_ValuesCount].ChannelId = ChannelId;
221   SimAgent_Testcase[SimAgent_Testcase_ValuesCount].Value = Value;
222   SimAgent_Testcase[SimAgent_Testcase_ValuesCount].StartOffset =
          StartOffset;
223   PRP_INFO(YELLOW("SimAgent_AddTestcase_Entry called - ChannelId=%u, Value
          =%u, StartOffset=%u\n")
224       , ChannelId, Value, StartOffset);
225   SimAgent_Testcase_ValuesCount = SimAgent_Testcase_ValuesCount + 1; //
          increment at the end to use the zero index
226 }
```

Literaturverzeichnis

[ATv07] Andrew S. Tanenbaum & Maarten van Steen ; Tanenbaum, Andrew S.; van Steen, Maarten: *Distributed Systems: Principles and Paradigms // Distributed systems: Principles and paradigms*. 2. ed. Upper Saddle River NJ : Pearson Prentice Hall, 2007. – ISBN 0–13–239227–5

[ATZ13] ATZagenda: „Es gibt zunehmend kostensensitive Kunden“. 2 (2013), Nr. 1, S. 60–63. http://dx.doi.org/10.1007/s40357-013-0018-y. – DOI 10.1007/s40357–013–0018–y

[AUTa] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *AUTOSAR Layered Software Architecture*. V3.0.4 (Release 4.1)

[AUTb] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Complex Driver design and integration guideline*. V1.1.0 (Release 4.1)

[AUTc] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Requirements on DIO Driver*. V2.2.0 (Release 4.1)

[AUTd] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Requirenments on Runtime Environment*. V2.3.1 (Release 4.1)

[AUTe] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Specification of Communication*. V5.1.1 (Release 4.1)

[AUTf] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Specification of Compiler Abstraction*. V3.3.1 (Release 4.1)

[AUTg] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Specification of DIO Driver*. V2.7.0 (Release 4.1)

[AUTh] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Specification of Module XCP*. V2.3.0 (Release 4.1)

[AUTi] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Specification of RTE*. V3.5.0 (Release 4.1)

[AUTj] AUTOSAR ; AUTOSAR Partnership (Hrsg.): *Virtual Functional Bus*. V3.2.0 (Release 4.1)

[AUTk] AUTOSAR Partnership ; AUTOSAR Partnership (Hrsg.): *AUTOSAR - VFB: Virtual Function Bus*

[AUT14] AUTOSAR Partnership ; AUTOSAR Partnership (Hrsg.): *AUTOSAR Methodology*. 2014

[AUT16] AUTOSAR Partnership ; AUTOSAR Partnership (Hrsg.): *AUTomotive Open System ARchitecture: Enabling Innovation*. 2016

[AUT17] AUTOSAR Partnership ; AUTOSAR Partnership (Hrsg.): *Adaptive Platform: General Information*. `https://www.autosar.org/standards/adaptive-platform/`. Version: 2017

[Bal92] Bal, Henri E.: *Programming distributed systems*. 2. [print.]. Summit, NJ and New York, NY : Silicon Press and Prentice Hall, 1992. – ISBN 0–13–722083–9

[BBT10] Berns, Karsten von (Hrsg.) ; Bernd, Schürmann von (Hrsg.) ; Trapp, Mario (Hrsg.): *Eingebettete Systeme*. Wiesbaden : Vieweg+Teubner, 2010. `http://dx.doi.org/10.1007/978-3-8348-9661-2`. `http://dx.doi.org/10.1007/978-3-8348-9661-2`. – ISBN 978–3–8348–0422–8

[BE08] Balzert, Helmut ; Ebert, Christof: *Lehrbuch der Sofwaretechnik*. 2. Heidelberg: Spektrum Akademischer Verlag, 2008. – ISBN 978–3–8274–1161–7

[Bei90] Beizer, Boris: *Software testing techniques*. 2. New York : International Thomson Computer Press, 1990. – ISBN 9781850328803

[BEL05] Byhlin, Susanna (Hrsg.) ; Ermedahl Jan Gustafsson, Andreas (Hrsg.) ; Lisper, Björn (Hrsg.): *Applying Static WCET Analysis to Automotive Communication Software*. 2005

[Bel07] Beltran, Juan Carlos F.: *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Frankfurt am Main : Entwickler.press, 2007. – ISBN 9783939084112

[BMWa] BMW AG ; BMW AG (Hrsg.): *Funktionslastenheft ZBE Touch: Lastenheft: PDM-Dokumentnummer: 10363004 - 000 - 065*

[BMWb] BMW AG ; Bayerische Motoren Werke Aktiengesellschaft (Hrsg.): *Geschäftsbericht der BMW Group 2014*. München,

[BMWc] BMW AG ; BMW AG (Hrsg.): *Zentrales Bedienelemente ZBE Touch/high: Lastenheft: PDM-Dokumentnummer: 10363004 - 000 - 06*

[BMW15a] BMW AG ; BMW AG (Hrsg.): *Der BMW 7er. Betriebsanleitung.* 2015

[BMW15b] BMW AG: BMW AUTOSAR CORE 4 REL. 2: Booklet - Short Guide AUTOSAR REL. 4 based Platform Software. (2015)

[Bru15] Bruns, Axel: *Die Geschichte des Computers: Wie es bis zur Form des heutigen 'PC' kam.* München : neobooks Self-Publishing, 2015. – ISBN 9783738021455

[CDK05] Coulouris, George F. ; Dollimore, Jean ; Kindberg, Tim: *Distributed systems: Concepts and design.* 4th ed. Harlow : Addison-Wesley, 2005 (International computer science series). – ISBN 978–0–321–26354–4

[CI] CANtech, Vector ; Inc ; Vector Informatik GmbH (Hrsg.): *Introduction to the CAN Calibration Protocol: Version 3.0*

[Deu07] Deutsches Institut für Normung e.V: *Leittechnik - Prozeßautomatisierung - Automatisierung mit Prozeßrechensystemen, Begriffe.* 1998-07

[Dit16] Dittrich, Alexander: *Validierungsmethoden für eine integrierte softwarebasierte Simulationsmethodik.* Augsburg, Universität Augsburg, Masterarbeit, 2016

[DNT09] Dias-Neto, Arilo C. ; Travassos, Guilherme H.: Model-based Testing Approaches Selection for Software Projects. In: *Inf. Softw. Technol.* 51 (2009), Nr. 11, 1487–1504. http://dx.doi.org/10.1016/j.infsof.2009.06.010. – DOI 10.1016/j.infsof.2009.06.010. – ISSN 0950–5849

[Dud17] Dudenredaktion: *Definition "Fallstudie".* http://www.duden.de/node/757360/revisions/1084491/view. Version: 2017

[Dud18] Dudenredaktion: *Definition "Prozess".* https://www.duden.de/node/659973/revisions/1195613/view. Version: 2018

[EAS] EAST-ADL Association ; EAST-ADL Association (Hrsg.): *EAST-ADL WhitePaper: Version 2.1.12*

[ERZ14] Eigner, Martin ; Roubanov, Daniil ; Zafirov, Radoslav: *Modellbasierte virtuelle Produktentwicklung.* Berlin : Springer Vieweg, 2014. http://dx.doi.org/10.1007/978-3-662-43816-9. http://dx.doi.org/10.1007/978-3-662-43816-9. – ISBN 978–3–662–43815–2

[ES13] Eigner, Martin ; Stelzer, Ralph: *Product-Lifecycle-Management.* 2. Berlin, Heidelberg : Springer, 2013 (VDI). – ISBN 3540684018

[Ger14] Gert Reiling ; automotiveIT (Hrsg.): *Agile Methoden in der Automobilindustrie - automotiveIT.* http://www.automotiveit.eu/agile-methoden-in-der-automobilindustrie/entwicklung/id-0046081. Version: 2014

[Han06] Hanser Verlag (Hrsg.): *Model Transformers for Test Generation from System Models.* 2006

[Hof08] Hoffmann, Dirk W.: *Software-Qualität.* Berlin, Heidelberg : Springer, 2008 (EXamen.press). – ISBN 3540763228

[IEE90] IEEE: *IEEE Standard Glossary of Software Engineering.* 1990

[Int13] International Organization for Standardization: *Road vehicles — Unified diagnostic services (UDS).* 03.2013

[Joh13] Johnson, Steven: *The Connected Lives of Ants, Brains, Cities, and Software Emergence.* 2013. – ISBN 978–0684868769

[KF09] Kindel, O. ; Friedrich, M.: *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis.* dpunkt-Verlag, 2009 https://books.google.de/books?id=VLRmPgAACAAJ. – ISBN 9783898645638

[Kle] Kleinknecht, H. ; ASAP (Hrsg.): *Can Calibration Protocol Specification: Version 2.1*

[Kle09] Kleuker, Stephan: *Formale Modelle der Softwareentwicklung: Model-Checking Verifikation Analyse und Simulation.* 1. Aufl. Wiesbaden, 2009 (Datenbanken und Softwareentwicklung). – ISBN 978–3–8348–9595–0

[Köb15a] Köberl, Marcel: *Integration einer Softwarebasierten Kundeneingabe.* Augsburg, Universität Augsburg, Seminararbeit, 2015

[Köb15b] Köberl, Marcel: *Integration softwarebasierter Automatisierungsmethoden in eine Test-ECU.* Augsburg, Universität Augsburg, Masterarbeit, 2015

[Mar16] Markthaler, Matthias: *Metamodellbasierte Methodik zur automatisierten Testfallerstellung für verteilte Systeme aus domänenspezifischen Testmodellen.* München, Hochschule München, Masterarbeit, 2016

[MBt] MBtech Group GmbH & Co. KGaA ; MBtech Group GmbH & Co. KGaA (Hrsg.): *Virtuelle Absicherungsplattform VAP: Die innovative Lösung für Rapid Prototyping*

[Men14] Mentor Graphics: *Volcano VSX: Vehicle Systems Solutions: Version:*

2014.2.3.20150309-0402, Supported AUTOSAR Versions: 4.0.3, 4.0.2, 3.2.1. 2014

[MLD+09] Müller, Tobias C. ; Lochau, Malte ; Detering, Stefan ; Saust, Falko ; Garbers, Henning ; Märtin, Lukas ; Form, Thomas ; Goltz, Ursula: Umsetzung eines modellbasierten durchgängigen Entwicklungsprozesses für AUTOSAR-Systeme mit integrierter Qualitätssicherung - A comprehensive Description of a Model-based, continuous Development Process for AUTOSAR Systems with integrated Quality Assurance. (2009)

[MSB12] Myers, Glenford J. ; Sandler, Corey ; Badgett, Tom: *The art of software testing*. 3. Hoboken, N.J : John Wiley & Sons, 2012. – ISBN 1119202485

[MW08] Marwedel, Peter ; Wehmeyer, Lars: *Eingebettete Systeme*. Korrigierter Nachdr. Berlin : Springer, 2008 (EXamen.press). http://dx.doi.org/10.1007/978-3-540-34049-2. http://dx.doi.org/10.1007/978-3-540-34049-2. – ISBN 978–3–540–34048–5

[Pat07] Patzer, Andreas: Steuergeräte-Kalibrierung in der Produktentwicklung. In: *Elektronik automotive* (2007)

[Pow86] Powell, Patricia B.: Software Validation, verification, testing and documentation. In: Andriole, S. J. (Hrsg.): *Software Validation, verification, testing and documentation*. Princeton, N.J. : Petrocelli Books, 1986. – ISBN 0–89433–269–4, S. 187–312

[PP02] Pretschner, Alexander; Philipps, Jan: Szenarien modellbasierten Testens. (2002)

[Prö14] Pröll, Reinhard: *SW-Basierte Kundenfunktionseingabe für die Automatisierung*. Augsburg, Universität Augsburg, Masterarbeit, 2014

[Pu] Pretschner, Alexander; und E. T. H. Informationssicherheit: *Zur Kosteneffektivität modellbasierten Testens*

[PY08] Pezzè, Mauro ; Young, Michal: *Software testing and analysis*. [Hoboken, N.J.]: Wiley, 2008 (Process, principles, and techniques). – ISBN 0471455938

[RBGW10] Rossner, Thomas ; Brandes, Christian ; Götz, Helmut ; Winter, Mario: *Basiswissen modellbasierter Test*. 1. Heidelberg : dpunkt.verl., 2010. – ISBN 9783898645898

[Saa14] Saad, Christian: *Data-flow based model analysis: Approach, Implementation and Applications*, Diss., 2014

[SBB11] Seidl, Richard ; Baumgartner, Manfred ; Bucsics, Thomas: *Basiswissen Testautomatisierung*. 1. Heidelberg, Neckar : dpunkt, 2011. – ISBN 9783898647243

[Sch] Schieferdecker, Ina: Interoperabilität: Modellbasiertes Testen. In: *OBJEKT-Spektrum 3/07*

[Sch05] Scholz, Peter: *Softwareentwicklung eingebetteter Systeme: Grundlagen Modellierung Qualitätssicherung*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005 (SpringerLink : Bücher). http://dx.doi.org/10.1007/3-540-27522-3. http://dx.doi.org/10.1007/3-540-27522-3. – ISBN 978–3–540–27522–0

[Sta04] Stappert, Friedhelm: *From Low-Level to Model-Based and Constructive Worst-Case Execution Time Analysis.*, Dissertation, 2004

[SVEH12] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 1. Aufl. s.l.: dpunkt.verlag, 2012. – ISBN 978–3–89864–448–8

[SZ13] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. 5., überarb. u. ak. Aufl. 2013. Wiesbaden : Springer, 2013 (ATZ / MTZ-Fachbuch). http://dx.doi.org/10.1007/978-3-8348-2470-7. http://dx.doi.org/10.1007/978-3-8348-2470-7. – ISBN 3834824704

[Vec] Vector Informatik GmbH ; Vector Informatik GmbH (Hrsg.): *XCP – Das Standardprotokoll für die Steuergeräte-Entwicklung: Grundlagen und Einsatzgebiete*

[Vec13] Vector Informatik GmbH: *AUTOSAR Configuration Process - How to handle 1000s of parameters*. 01.01.2013 (Webinar 2013-04-19)

[Web15] Weber, Sergej: Agile in Automotive – State of Practice 2015. (2015)

[ZG10] Zöller-Greer, Peter: *Software-Architekturen: Grundlagen und Anwendungen ; mit einer Einführung in Architekturbeschreibungssprachen (ADLs), UML, Object-Z, OCL, CORBA, IDL, Entwurfsmuster, Architektursichten, Architekturmuster, DDD, Architektur-Dokumentation, Komplexitätsprobleme, Standard-Architekturen, SOA, TOGAF, RM-ODP, Software-Factories*. 3. Aufl. Wächtersbach : Composia-Verl., 2010 (Reihe Wissen & Praxis "kompakt"). – ISBN 978–3–9811639–3–3

Abkürzungsverzeichnis

ABK	Anzeige-Bedien-Konzept
Abk.	Abkürzung
ADL	Architecture Description Language
API	Application Programming Interface
APIX	Automotive Pixel Link
ARXML	AUTOSAR Extensible Markup Language
ATM	Advanced Telecomunication Module
AUTOSAR	Automotive Open System Architecture
BC	Business Case
BAC	BMW AUTOSAR Core
BDC	Body Domain Controller
BMW	Bayerische Motoren Werke
BSW	Basic Software
CAN	Controller Area Network
CCP	Can Calibration Protocol
CDD	Complex Device Driver
CF	Consecutive Frame
CID	Central Information Display
CRO	Command Receive Object

CRM	Command Return Message
CMD	Command Code
CO	Kohlenstoffmonoxid
CO_2	Kohlenstoffdioxid
CPU	Central Processing Unit
CTR	Command Counter
CTO	Command Transmission Object
D	Diagnose
DAQ	Data Acquisition
DSC	Dynamic Stability Control
DTO	Data Transmission Object
DIN	Deutsche Industrie Norm
DIO	Digital Input/Output
EAST	Electronics Architecture and Software Technology
ECU	Steuergerät (Electronic Control Unit)
ERR	Error
Eth	Ethernet
EV	Event *Packet*
E/E	Elektrik Elektronik
FAA	Functional Analysis Architecture
FDA	Functional Design Architecture
FF	First Frame
FI	Fahrzeug Interieur

FIFO	First in First out
GW	Gateway
GWS	Gangwahlschalter
HDA	Hardware Design Architecture
HIL	Hardware in the Loop
HMI	Human Machine Interface
HU	Headunit
HW	Hardware
HWM	Hardware Modul
IEEE	Institute of Electrical and Electronics Engineers
ID	Identifikator
IPET	Implicit Path Enumeration Technique
ISO	International Organization for Standardization
IT	Informationstechnik
ItO	Idea to Offer
IuK	Infotainment- und Karosseriefunktionen
LED	Light-Emitting Diode
LIN	Local Interconnection Network
LOC	Lines of Code
LSL	Lichtschranke links
LSR	Lichtschranke rechts
MCAL	Micro Controller Abstraction Layer
MBT	Modellbasiertes Testen

MIL Model in the Loop

MMI Mensch Maschine Interface

MOST Media Oriented System Transport

NTP Network Time Protocol

OBD On-board Diagnose

ODT Object Descriptor Table

OEM Original Equipment Manufacturer

OS Operating System

OSI Open Systems Interconnection

PC Personal Computer

PCI Protocol Control Information

PDU Protocol Data Unit

PEP Produktentwicklungsprozess

PID Packageidentifier

PTP Precision Time Protocol

RAM Random Access Memory

RE Runnable Entity

RES *Command* Response Packet

ROM Read Only Memory

RTE Runtime Environment

RTTI Real Time Traffic Information

SDK Software Development Kit

SERV Service Request *Packet*

SF	Single Frame
SGBD	Steuergeräte-Beschreibungsdatei
SIL	Software in the Loop
STIM	Stimulation
SUT	System under Test
SW	Software
SWC	Softwarekomponente (Software Component)
SZL	Schaltzentrum Lenksäule
TFM	Technical Feature Model
UDS	Unified Diagnostic Services
UML	Unified Modeling Language
VAP	Virtuelle Absicherungs Plattform
VFB	Virtual Functional BUS
VID	Verbose Identifier
WCET	Worst Case Execution Time
XCP	Universal Measurement and Calibration Protocol
ZBE	Zentrale Bedieneinheit

Abbildungsverzeichnis

Tabellenverzeichnis