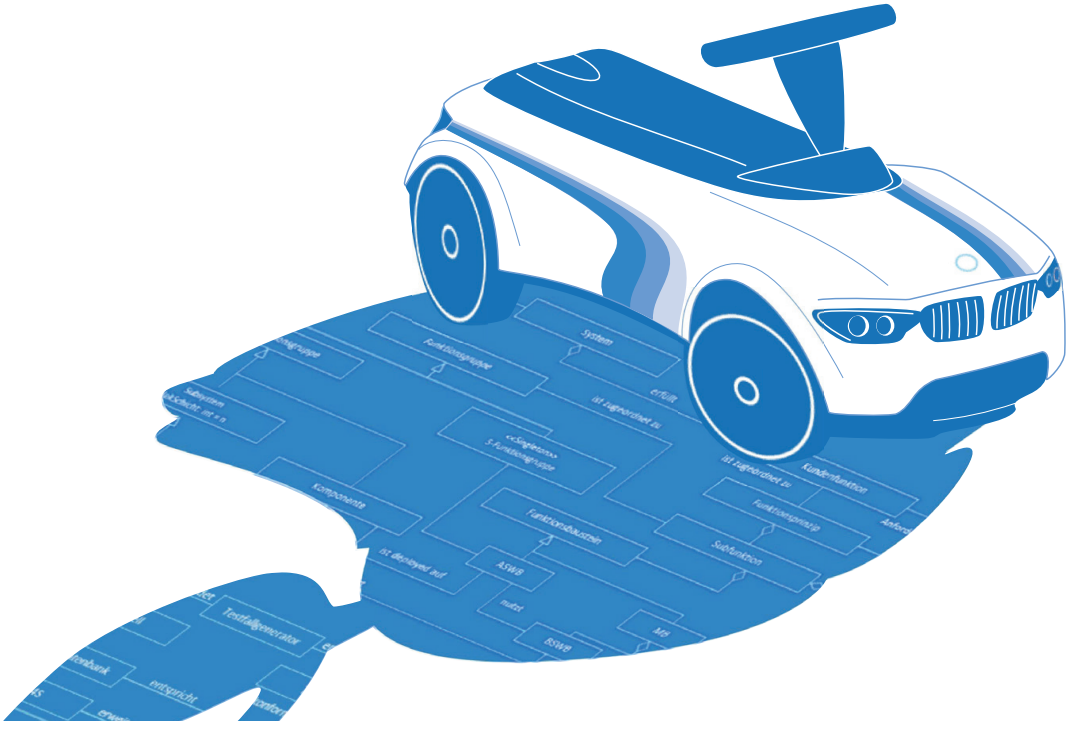
SE | RWTH AACHEN UNIVERSITY

Matthias Markthaler

Modellbasierte Methode für die automatisierte Testfallerstellung in der Automobilindustrie auf der Grundlage eines durchgängigen Systems Engineering Ansatzes



Aachener Informatik-Berichte,
Software Engineering

Band 52

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Modellbasierte Methode für die automatisierte Testfallerstellung in der Automobilindustrie auf der Grundlage eines durchgängigen Systems Engineering Ansatzes

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

M. Sc.
Matthias Markthaler
aus Fürstenfeldbruck

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Matthias Tichy

Tag der mündlichen Prüfung: 22. Juni 2022

[Mar22] M. Markthaler:
Modellbasierte Methode für die automatisierte Testfallerstellung in der Automobilindustrie auf der Grundlage eines durchgängigen Systems Engineering Ansatzes.
Aachener Informatik-Berichte, Software Engineering, Band 52
ISBN 978-3-8440-8845-8, Shaker Verlag, Nov. 2022.
www.se-rwth.de/publications

Eidesstattliche Erklärung

I, Matthias Markthaler

erklärt hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;

2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;

3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;

4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;

5. Alle wesentlichen Quellen von Unterstützung wurden benannt;

6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;

7. Teile dieser Arbeit wurden zuvor veröffentlicht und zwar in:

   [KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas-Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

   [DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

   [CMRP18] Mohamad Chamas, Matthias Markthaler, Bernhard Rumpe, and Kristin Paetzold. Bewertungsmethodik zur Verbesserung der Testfallerstellung auf

Basis von SysML. In *DFX 2018: 29th Symposium on Design for X*, pages 143–154, München, 2018.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. *SMArDT modeling for automotive software testing. Software: Practice and Experience*, 0(49):301–328, 2019.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

– Stefan Kriebel, Matthias Markthaler, Christian Granrath, Johannes Richenhagen, and Bernhard Rumpe. Modeling Hardware and Software Integration by an Advanced Digital Twin for Cyber-Physical Systems – Applied to the Automotive Domain. In *Handbook of Model-Based Systems Engineering*. Springer. ist derzeit zur Veröffentlichung akzeptiert und befindet sich in der Endkontrolle.

München, den 08.07.2022

Matthias Markthaler

Erklärung zur Wahrung von Betriebsgeheimnissen

Matthias Markthaler

erklärt hiermit, dass diese Dissertation durch ihre Veröffentlichung keine bestehenden Betriebsgeheimnisse verletzt.

Hiermit versichere ich, dass die in Zusammenarbeit mit der *BMW Group (Petuelring 130, 80809 München)* entstandene Dissertationsschrift durch ihre Veröffentlichung keine bestehenden Betriebsgeheimnisse verletzt.

München, den 08.07.2022

Matthias Markthaler

Kurzfassung

Die heutige und zukünftige Komplexität in großen Cyber-Physischen Systemen ist ohne systematische und digitalisierte Herangehensweisen kaum noch beherrschbar. Ein aktuelles Paradebeispiel solcher Cyber-Physischen Systeme sind die Produkte der Automobilindustrie, die einer noch nie da gewesenen Komplexität gegenüberstehen. Die Komplexität in der Automobilindustrie steigt stetig mit den Kundenansprüchen wie Individualität, Nachhaltigkeit, autonomes Fahren und Sicherheit bei hohen Qualitätsansprüchen zu günstigen Preisen. In der Informatik konnte in den letzten Jahrzehnten die Komplexität, Qualität und Effizienz mit agilen Methoden und einer durchgängigen Verwendung von Modellen verbessert werden. Während agile Methoden in der Automobilindustrie bereits ein fester Begriff ist, steht eine durchgängige Verwendung von Modellen in diesem Bereich noch aus. Auch wenn eine Reihe erfolgversprechende modellbasierter Ansätze aufgesetzt wurden, konnte sich eine modellbasierte Entwicklung noch nicht vollständig etablieren.

Im Rahmen dieser Dissertation wird eine durchgängige Entwicklungsmethode unter Verwendung von Modellen in Verbindung mit einem kompatiblen Testfallgenerator in der Industrieanwendung vorgestellt. Die Methode ist eine Weiterentwicklung von der modellbasierten Spezifikationsmethode für Anforderungen, Design und Test und erlaubt eine modellgetriebene Entwicklung für die automatisierte Erstellung von Artefakten wie beispielsweise Testfälle.

Die wichtigsten Ergebnisse dieser Arbeit lassen sich wie folgt zusammenfassen:

- Eine modellgetriebene Spezifikationsmethode für Anforderungen, Design und Test in einer Automobilindustrie-Anwendung.
- Eine Statusanalyse des modellbasierten Testens in der Automobilindustrie.
- Kriterien für die Testbarkeit der spezifizierten Modelle und für den Nutzen der erstellten Testfälle.
- Eine domänenspezifische Sprache auf der Basis einer entwickelten MontiCore-Grammatik, die eine Maschinenlesbarkeit der Modelle gewährleistet.
- Ein Testfallgenerator, der diese domänenspezifischen abstrakten Modelle für die automatisierte Erstellung von Testfällen nutzt.
- Gewonnene aufgearbeitete Erkenntnisse aus der angewendeten Forschung in der Industrie.

Die Erkenntnisse aus der Anwendung der Methoden und der Tools in der Automobilindustrie dienen als Leitlinien für die Übertragung weiterer Kenntnisse aus der Informatik in die Praxis von traditionellen maschinenbau-geprägten Industrien. Die vorgestellte modellbasierte Spezifikationsmethode wurde fest in der Entwicklung von elektrischen Antrieben in einem Automobilkonzern verankert. Die Spezifikationsmodelle dienen den testfallerstellenden Personen als Basis zur manuellen und automatisierten Absicherung der Systeme und ermöglichen eine breite, tiefe und infolgedessen qualitativ hohe Absicherung.

Abstract

The current and future interdisciplinary complexity in large cyber-physical systems can hardly be managed without systematic and digitized approaches. A popular example of cyber-physical systems are automobiles, which are facing unprecedented complexity. The complexity in automobiles is constantly increasing with its customer demands such as individuality, sustainability, autonomous driving and safety at high-quality standards at reasonable prices. In software engineering, complexity, quality, and efficiency have been improved over the last decades with agile methods and consistent use of models. While agile methods are already established in the automotive industry, the consistent use of models is still pending. Even though a number of promising model-based approaches have been set up, model-based development has not yet fully been established. This dissertation presents a consistent model-driven development method and a compatible test case generator. The method is a further development of the model-based Specification Method for Requirements, Design, and Test and allows model-driven development for the automated creation of artifacts such as test cases.

The main contributions of this thesis are:

- A model-driven Specification Method for Requirements, Design, and Test used in the automotive industry,
- A status analysis of model-based testing in the automotive industry,
- A criteria catalog for the testability of the specified models and the usability of created test cases,
- A domain-specific language based on a developed MontiCore grammar, which provides machine readability of models,
- A test case generator that creates automated test cases out of these domain-specific abstract models, and
- Lessons learned from the applied research in the industry.

The lessons learned serve as guidelines for the transfer of computer science methods to traditional mechanical engineering industries. The presented model-based specification method was firmly established in the electric drive development of a multinational automotive company. The specification models provide a basis for manual and automated testing and enable broad, deep, and high-quality safeguarding.

Danksagung

Ich bedanke mich bei all denen, die mich während meiner Promotion begleitet, unterstützt, mir den nötigen Rückhalt gegeben oder für passende Ablenkungen gesorgt und dadurch zu dem Erfolg meiner Promotion beigetragen haben. Mein besonderer Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die Betreuung meiner Promotion, für die spannende und lehrreiche Zeit, für die konstruktiven Diskussionen und für die Möglichkeit im industriellen Umfeld promovieren zu können. Als Nächstes möchte ich Prof. Dr. Matthias Tichy für die Bereitschaft, das Zweitgutachten dieser Arbeit zu übernehmen, danken. Darüber hinaus bedanke ich mich bei Prof. Dr. Erika Ábrahám für die Leitung meines Prüfungskomitees und Prof. Dr. Stefan Kowalewski für die Mitarbeit in diesem Komitee. Des Weiteren bedanke ich mich bei der BMW Group, der Exida GmbH und der FEV GmbH für die konstruktive Zusammenarbeit während der im Kontext dieser Arbeit durchgeführten Industriekooperationen. In dieser Hinsicht gilt mein Dank Dr. Michael Blum, Dr. Mohamad Chamas, Jörg Ebeling, Dr. Marc-Thomas Eisele, Arne Haas, Kevin Heinen, Jonas Karlsson, Dr. Stefan Kriebel, Thomas Lachner, Dr. Philipp Orth, Dr. Johannes Richenhagen, Karin Samira Salman, Dr. Georg Strobl, Dr. Björn Weigold und besonders Dr. Rolf Ebert für die Betreuung seitens der BMW Group.

Herzlich bedanken möchte ich mich auch bei allen Kolleg*innen, die mich während meiner Anwesenheit am Lehrstuhl und über die Distanz begleitet haben: Kai Adam, Vincent Bertram, Marita Breuer, Arvid Butting, Manuela Dalibor, Robert Eikermann, Sylvia Gunder, Arkadii Gerasimov, Robert Heim, Gabriele Heuschen, Dr. Katrin Hölldobler, Nico Jansen, Oliver Kautz, Jörg Christian Kirchhof, Dr. Evgeny Kusmenko, Achim Lindt, Dr. Markus Look, Dr. Judith Michael, Dr. Pedram Mir Seyed Nazari, Dr. Klaus Müller, Sonja Müßigbrodt, Lukas Netz, Dr. Dimitri Plotnikov, Deni Raco, Dr. Alexander Roth, David Schmalzing, Steffi Schrader, Igor Shumeiko, Sebastian Stüber, Simon Varga, Galina Volkova, Louis Wachtmeister, Dr. Michael von Wenckstern und Dr. Andreas Wortmann. Besonderer Dank gilt Imke Nachmann geb. Drave, Dr. Timo Greifenberg, Steffen Hillemacher und Dr. Christoph Schulze die mir durch ihren Einsatz diese Arbeit ermöglicht haben. Darüber hinaus bedanke ich mich bei allen beteiligten Hiwis und Student*innen, die die Entwicklung des Testfallgenerators unterstützt haben. Auch möchte ich mich bei den Kolleg*innen und Freund*innen bedanken, die immer ein offenes Ohr für mich hatten und mich mit Rat und Tat unterstützt haben.

Abschließend möchte ich ein besonders großes Danke an meine Eltern, Bettina und Mike, sowie meinem Bruder, Simon richten. Ihr wart immer für mich da und habt mich auf meinem gesamten Lebensweg stets unterstützt. Dadurch habt ihr es mir überhaupt erst möglich gemacht diesen Weg zu gehen. Ebenso möchte ich mich bei Simon für das Sichten und Kommentieren früher Fassungen dieser Arbeit bedanken. Schließlich möchte ich mich noch ganz besonders bei meiner Partnerin Christina bedanken. Sie hat mich stets unterstützt, mich motiviert und mir den Rücken freigehalten. Ich danke dir!

München, Juli 2022
Matthias Markthaler

Disclaimer

Die Ergebnisse, Meinungen und Schlüsse dieser Dissertation sind nicht notwendigerweise die der BMW Group.

The results, opinions, and conclusions expressed in this thesis are not necessarily those of the BMW Group.

Inhaltsverzeichnis

Kapitel 1

Einführung

Eine entscheidende Herausforderung für den Erfolg der Automobilindustrie in der Gegenwart und Zukunft ist die Bewältigung der interdisziplinären Komplexität bei gleichzeitiger Wahrung der Qualität und Reduzierung der Kosten der Produkte. Heutige Cyber-Physische Systeme (CPSs) wie das Automobil sind in der Entwicklung nicht mehr gemäß der physischen Komponenten modularisierbar, da Fahrzeugfunktionen hoch vernetzt über mehreren hierarchischen fahrzeuginternen und externen Systemen verteilt sind. Wird versucht die Fahrzeugfunktionen mittels einer rekursiven Systemzerlegung isoliert zu betrachten, können viele nicht lokale, auftauchende Muster nicht beobachtet werden. Zusätzlich stehen die Automobilhersteller unter einem enormen Innovationsdruck, um den heutigen Kundenansprüchen wie neue Technologien, Nachhaltigkeit und Individualitätsansprüchen bei gleichbleibender Qualität gerecht zu werden. Abbildung 1.1 fasst zwei gegenwärtige Trends aus der Automobilindustrie zusammen und gibt einen Ausblick auf die zukünftige zu bewältigende Komplexität.

Nach [Hit21] definiert sich *Komplexität* über die Konnektivität, die Vielfalt und den Grad der Verflechtung von Teilen eines Systems. Je komplexer ein System ist desto mehr Aufwand wie Zeit, Kosten und Rechenleistung wird benötigt, um die Verflechtungen zu entwirren, die vielfältigen Systeme und deren Interaktionen zu identifizieren und zu charakterisieren [Hit07].

In heutigen und zukünftigen Fahrzeugen werden neue Funktionalitäten durch die steigende technische Leistungsfähigkeit von Prozessoren und deren zunehmende fahrzeuginterne und -externe Vernetzung ermöglicht (vgl. Abbildung 1.1). Auf dieser Basis lassen sich Softwareinnovationen umsetzen, die einige Jahre zuvor noch undenkbar schienen. Insbesondere die Vernetzung von Fahrzeugen untereinander und der Umwelt ermöglichen es, neue software-basierte Funktionsinnovationen umzusetzen. Zusätzlich werden traditionelle Lösungen aus dem Maschinenbau zunehmend durch geeignetere Softwarelösungen verbessert. Dies erklärt auch, dass obwohl die Anzahl an Steuergeräten abnimmt, die Anzahl an Funktionen pro Fahrzeug und pro Steuergerät offenbar exponentiell zunimmt [SZ16]. Überdies bringt die größer werdende Vielfalt an systemübergreifenden Funktionen stark steigende Komplexität mit sich. Diese Komplexität ist intellektuell[1] nicht mehr greifbar. Dies wird zusätzlich durch eine wachsende Variantenvielfalt verstärkt.

[1] In dieser Arbeit wird das Adjektiv „intellektuell“ verwendet, um sich von „maschinellen“ Eigenschaften abzugrenzen. *Intellektuell* verweist demnach auf die geistige, verstandesmäßige Erschließung oder die Aktion eines Menschen während der Entwicklung, während *maschinell* auf ein mit einer Maschine durchgeführtes Verfahren hinweist.

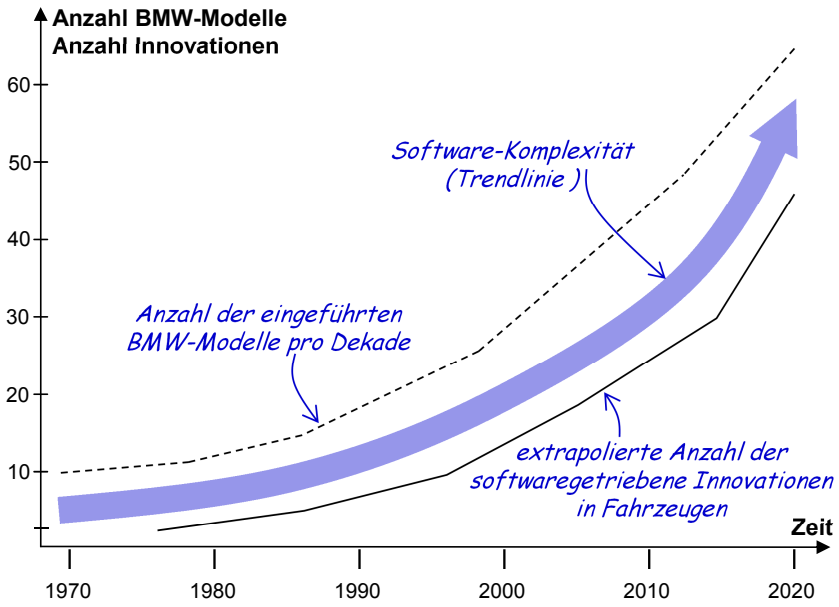


Abbildung 1.1: Trend-Zusammenhang der Anzahl der neu eingeführten BMW-Modelle [LvO15, BMW19], der extrapolierten Anzahl von eingeführten Software-Innovationen [EF17] und der resultierenden Komplexität.

Mit dieser Variantenvielfalt versprechen sich die Automobilhersteller, den individuellen Kundenwünschen gerecht zu werden (vgl. Abbildung 1.1 am Beispiel der BMW Group). Darüber hinaus werden die Produkt- und Entwicklungszyklen in der Automobilindustrie kürzer, da die softwareintensiven Systeme mit den Ansprüchen der schnelllebigen Telekommunikationsbranche Schritt halten müssen. Gleichzeitig müssen die Innovationen effektiv für bestehende langlebige Produkte und Systeme übertragen und integriert werden. Selbstverständlich kostengünstig für einen steigenden Absatz und Umsatz.

Um die Komplexität zu bewältigen, hat die Automobilindustrie mehrere Normen, Lösungen und Plattformen entwickelt und übernommen. Diese Normen dienen nicht nur als interne Leitlinien und Empfehlungen für Partner*innen, sondern werden in Rechtsfragen auch von rechtlichen Instanzen auf ihre Beachtung geprüft, wie beispielsweise Richtlinien in der Medizin. Aus diesem Grund werden in dieser Arbeit verstärkt bestehende Normen herangezogen. Ein Beispiel ist die Norm für die funktionale Sicherheit von Straßenfahrzeugen, ISO 26262 [ISO18b, ISO18a]. Abseits der rechtlichen Aspekte steht jedoch die Beherrschung der Komplexität im Vordergrund. Darunter fallen prozess- als auch produktbezogene Maßnahmen [HBP+17] wie die ISO 26262, die standardisierte Softwarearchitektur wie AUTOSAR [AUT19], Vorgehensmodelle wie das V-Modell [Koo19] und Referenzmodellen wie das CMMI [CMM19, CKS11] oder SPICE [ISO12].

Die steigende Bedeutung von Software in der Automobilindustrie legt nahe, schon etablierte Methoden und Tools aus der Informatik zu übernehmen. In der Informatik verbesserten in den letzten Jahren vor allem die agilen Methoden und die durchgängige

Verwendung von Modellen die Qualität und Effizienz in der Softwareentwicklung [Rum17]. Die agilen Methoden werden von der Automobilindustrie bereits erfolgversprechend verwendet [Sch19]. Allerdings steht eine durchgängige etablierte Verwendung von Modellen, beginnend mit der Anforderungserhebung bis hin zur Implementierung in der Automobilindustrie noch aus.

1.1 Beitrag

Die Fragen, zu dessen Antwort diese Arbeit Beiträge in methodischer und technologischer Hinsicht leistet, sind die folgenden:

- *Wie können etablierte Erkenntnisse aus der Informatik in die Praxis von traditionell maschinenbau-geprägten Industrien eingerichtet werden, um effektiv, effizient und nachhaltig die Qualität der Produkte zu gewährleisten?*
- *Welche Herausforderungen muss sich eine durchgängige Verwendung von Modellen im Tagesgeschäft der Industrie stellen?*
- *Wie sieht ein mögliches Szenario einer Methode und einem Tool für die Gewährleistung eines Qualitätsanspruchs aus?*

Bevor die Beiträge und Methodik dieser Arbeit skizziert werden, erfolgt eine Präzisierung der Herausforderung und einzelner Begriffe.

1.1.1 Herausforderungen

Viele traditionell maschinenbau-geprägte Industrien wie die Automobilindustrie mussten sich in der Vergangenheit komplizierten Herausforderungen stellen. Klassisch konzentriert sich ein Automobilhersteller auf die Integration und Absicherung ausgewählter Einzelteile von Zulieferern [PBKS07]. Infolgedessen liefen die Entwicklungen der Einzelteile und deren Funktionen nahezu autark ab [Bro06]. Die Vergabe von Entwicklungsumfängen begünstigte ebenfalls die individuelle Entwicklung der verschiedenen Disziplinen[2] Maschinenbau, Elektrotechnik oder später Informatik. Komplizierte Probleme der Automobilkonzerne bei der Integration und Absicherung konnten mittels einer rekursiven Zerlegung in Einzelteile beziehungsweise einzelne Systeme verstanden und gelöst werden. Mit dem Einzug von Software als essenzieller Bestandteil der Produkte konnten teile-übergreifende Funktionen ermöglicht werden. Traditionell unverbundene und unabhängige Funktionen wie Beschleunigen, Bremsen oder Lenken interagierten zusammen und es entstanden Abhängigkeiten der Funktionen [PBKS07]. Diese wachsende Zahl an teile- und systemübergreifenden Funktionen erhöht die Komplexität auftretender Probleme. Eine Lösung der Probleme mithilfe einer rekursiven Systemzerlegung in Einzelteile ist mit systemübergreifenden Funktionen nicht ohne Ausnahme möglich, da Querwirkungen der Funktionen und Systeme nicht immer abgebildet werden können. Die isolierte Entwicklung der einzelnen

[2] Eine Disziplin ist ein Wissenschaftszweig beziehungsweise ein akademisches Fachgebiet [Dud20].

Disziplinen, in sogenannten „Silos“ stellt die Integration und Absicherung dieser Systeme und Funktionen vor große Herausforderungen.

Um diesen Herausforderungen zu begegnen, verwendet die Automobilindustrie Methoden aus dem *Systems Engineering*. Das Systems Engineering ist ein ganzheitlicher, interdisziplinärer Ansatz, um die Ziele der verschiedenen Disziplinen in einer gemeinsamen Lösung zu realisieren und während des Lebenszyklus zu unterstützen [ISO17b, CKS11, Wei08]. Mit der Hilfe dieser Methoden und der Erfahrungen aus dem Systems Engineering werden die Entwicklungsprozesse und das Produktmanagement kontinuierlich verbessert. Das Systems Engineering vernetzt die einzelnen Disziplinen und unterstützt dabei, keine wichtigen Punkte zu übersehen. Gleichzeitig lässt das Systems Engineering Freiheiten bei einer für die Organisation adäquaten Umsetzung. Ein Ansatz, damit die Entwicklungen der Disziplinen nicht auseinanderlaufen, ist die Verwendung von modellbasierten Methoden. Ein Modell erlaubt es, relevante Eigenschaften und Aspekte eines zu analysierenden und zu entwickelnden Systems gezielt abstrakt darzustellen [Rum16]. Infolgedessen lässt sich die wahrgenommene Komplexität eines Systems reduzieren, um einen intellektuellen Zugang zu schaffen. Ansätze wie die modellbasierte Entwicklung werden insbesondere in dem *Software Engineering* erfolgreich angewendet [Rum16, Rum17]. Das Software Engineering ist eine Ingenieursdisziplin, die mittels systematischer, disziplinierter und quantifizierbarer Ansätze, Konzepte, Prozesse und Werkzeuge, die Entwicklung, den Betrieb und die Wartung von Software umsetzt [BF14, Rum16].

In dem (modellbasierten) Software Engineering haben sich in den letzten vier bis fünf Jahrzehnten fundamentale Prinzipien herauskristallisiert, wie etwa die Unterteilung komplexer Softwaresysteme in Abstraktionsebenen, der „Teile-und-herrsche-Ansatz“, das Verbergen nicht relevanter Informationen, die modulare Entwicklung, die schrittweise Verfeinerung oder Schichtenarchitekturen [BR07]. Diese etablierten Prinzipien wurden in dem Software Engineering über die letzten Jahrzehnte kontinuierlich verfeinert und weiterentwickelt [BR07]. Infolgedessen kann die Komplexität der weit umspannenden Softwaresysteme mit Softwaretools und Methoden beherrscht werden. In maschinenbau-geprägten Industrien haben sich diese modellbasierten Ansätze aus dem Software Engineering noch nicht durchgesetzt. Auch mit dem zunehmenden Anteil von Software ist die Automobilindustrie traditionell von verschiedenen Ingenieurdisziplinen geprägt, im Wesentlichen durch den Maschinenbau und die Elektrotechnik. Eine Methode für modellbasierten Ansätze muss das tiefgehende Verständnis der Ingenieurdisziplinen aufnehmen und in der Lage sein, sie in geeignete Modelle zu transformieren. Die Methoden und Erfahrungen aus dem Systems Engineering können bei dieser Transformation unterstützen. Zusammen mit den Erkenntnissen, Methoden und Tools aus dem Software Engineering wird infolgedessen die gegenwärtige und zukünftige Komplexität in der Automobilindustrie versucht zu beherrschen, um effektiv, effizient und nachhaltig die Qualität zukünftiger Produkte zu gewährleisten.

1.1.2 Methode

Inhalt dieser Arbeit ist ein systematischer Testansatz, der Methoden und Tools aus dem Systems und Software Engineering in einem bestehenden Seriengeschäft eines Automobil-

konzerns integriert. Dieser Ansatz fokussiert sich auf das Testen in der Automobilindustrie, insbesondere von elektrischen Antrieben.

Als Fundament der Methodik dient ein Ansatz aus dem System und Software Engineering für die Spezifikation und Entwicklung mithilfe von Modellen. Diese Modelle umfassen die im elektrischen Antrieb auftretenden Disziplinen wie Maschinenbau, Elektrotechnik und Informatik. Die Disziplinen arbeiten infolgedessen zusammen, um übergreifend die geeignetste Lösung zu finden. Zu diesem Zweck enthalten die Modelle, die in einem Gesamtmodell zusammengeführt sind, neben funktionalen Elementen auch Elemente der Hardware und der Organisation. Zusätzlich werden bereits etablierte Prinzipien aus dem Software Engineering wie Abstraktionsebenen, Schichtenarchitekturen, modulare Entwicklung und schrittweise Verfeinerung verwendet.

Auf dem Fundament dieser (modellbasierten) Systementwicklung können weitere Methoden und Tools digitalisiert[3] werden. In dieser Arbeit werden auf Basis von abstrakten Modellen, die der Spezifikation dienen, automatisiert Testfälle für die funktionale Integration in Hardwareumgebungen erstellt. Für diese modellbasierte Testfallerstellung wird ein Testfallgenerator vorgestellt. Dieser Testfallgenerator kann aus abstrakten Modellen, in denen keine Informationen über konkrete Signale wie Bordnetzsignale[4] vorliegen, Testfälle erstellen und vervollständigt die vorgestellte durchgängige Methodik von der Spezifikation bis zur Verifizierung. Diese Digitalisierung der Systementwicklung unterstützt bei der Testfallerstellung, erlaubt ein breites Prüfen von Funktionen und ermöglicht schnellere Test- und Entwicklungszyklen.

Zusätzlich zu dem methodischen und technologischen Inhalt steht in dieser Arbeit die Anwendbarkeit der Forschung im Vordergrund. Aus diesem Grund werden Praxiserfahrungen wissenschaftlich aufgearbeitet und an geeigneten Stellen eingebracht. Diese Exkurse aus der Praxis verstehen sich als Brücke zwischen Theorie und Praxis und sind mit dem Ausdruck **Theorie und Praxis** gekennzeichnet. Auf Grundlage dieser Arbeit können Modelle somit nicht nur methodisch, sondern praxisnaher gestaltet und vermittelt werden. Die Arbeit richtet sich also gleichermaßen an Theoretiker*innen und Praktiker*innen. Die theoretische Basis wird neben den wissenschaftlich unterlegten Exkursen aus der Praxis, ebenfalls mit Fallstudien anhand eines Fragebogens und eines Interviews untermauert und evaluiert.

1.1.3 Einordnung

Thema dieser Arbeit ist die modellbasierte Testfallerstellung für CPS. Hierfür ist es nötig, einerseits das System und dessen Funktionen adäquat zu beschreiben und andererseits geeignete Testfälle für eben dieses System zu generieren. Die Methodik zur Beschreibung des Systems lässt sich dem Gebiet der Technischen Lösung [CKS11] beziehungsweise der Technischen Prozesse [ISO15a, WRF+15] des Systems und Software Engineering

[3] Digitalisierung: „Digitalisierung ist der Einsatz digitaler Technologien, um ein Geschäftsmodell zu verändern und neue Umsatz- und Wertschöpfungsmöglichkeiten zu schaffen; es ist der Prozess des Übergangs zu einem digitalen Geschäft.“ nach [GR15, Gar20]

[4] Das physikalische Bordnetz ist für die Energie- und Signalverteilung zwischen Komponenten und Steuergeräten zuständig [PS16].

zuordnen. Diese Spezifikationsmethodik basiert auf digitalen Modellen, den sogenannten Spezifikationsmodellen, die eine Abhängigkeit von der Systementwicklung und deren Prozesse mit sich bringen. Darauf aufbauend können dann mit Hilfe dieser Spezifikationsmodelle geeignete Testfälle für das besagte System erzeugt und ausgeführt werden. Zur Validierung dieser Tests muss die modellbasierte Testfallerstellung stets bezogen auf die Anforderungen an das System geprüft werden. Aus diesem Grund schneidet diese Arbeit auch Elemente des Anforderungsmanagements an. Zusätzlich sollten in einem iterativen Prozess die Ergebnisse und Erkenntnisse der Tests in das Spezifikationsmodell zurückgespielt werden. Infolgedessen werden organisatorische und prozessuale Elemente der modellbasierten Testfallerstellung angeschnitten.

Technisch ergeben sich für den vorgestellten Testfallgenerator Überschneidungen mit der Technik der Modellprüfung (engl. „Model Checking“), auf deren Vergleich und Beschreibung in dieser Arbeit bewusst verzichtet wird, da es den Rahmen dieser Arbeit sprengen würde. Die Implementierung des Testfallgenerators erfolgt mit einer vollwertigen Sprach-Workbench für den Entwurf und die Realisierung von textuellen domänenspezifischen Sprachen.

Um Missverständnisse aufgrund disziplintypischer Semantik auszuschließen, werden relevante Begrifflichkeiten anfangs nochmals definiert. Die definierten Begriffe sind im Anhang B in einem Glossar zusammengeführt. Dennoch würde eine ausführliche Einführung in alle berührten Domänenspezifika den Umfang dieser Arbeit sprengen. Aus diesem Grund verweisen Literaturangaben auf relevante Referenzen.

1.2 Aufbau der Arbeit

Diese Arbeit besteht aus vier Teilen. Der erste Teil, Kapitel 1, 2 und 3, beinhaltet die relevanten Begriffsdefinitionen und Grundlagen aus der modellbasierten Entwicklung und dem Gebiet des Testens in der Automobilindustrie. Der zweite Teil, Kapitel 4 und 5, beinhaltet die in dieser Arbeit verwendete Entwicklungsmethodik und die daraus resultierenden Anforderungen an die technische Umsetzung der Testfallerstellung. Der dritte Teil, Kapitel 6 und 7, beinhaltet die technologische Umsetzung des Testfallgenerators für Spezifikationsmodelle. Der vierte Teil, Kapitel 8, beinhaltet die aus der Praxis gewonnenen Erkenntnisse und evaluiert diese.

Im Einzelnen gliedert sich die Arbeit wie folgt:

Kapitel 2 behandelt die Grundlagen der modellbasierten Entwicklung, die zum Verständnis dieser Arbeit notwendig sind. Zu diesem Zweck werden allgemeine Begrifflichkeiten aus der modellbasierten Entwicklung festgelegt. Darauf aufbauend wird auf die Entwicklung und Vorgehensnormen in der Automobilindustrie eingegangen. Des Weiteren werden die im Fokus stehende Modellierungssprache und das verwendete Werkzeug, MontiCore-Framework, erläutert. Schließlich wird noch die Basis, auf dem die Entwicklungsmethodik dieser Arbeit aufsetzt, erläutert. Am Ende von Kapitel 2 wird diese Methodik anderen ähnlichen Entwicklungsmethoden gegenübergestellt.

Kapitel 3 stellt die Testfallerstellung in der Automobilindustrie vor. Dazu werden Begrifflichkeiten aus dem Bereich des Testens und der Automobilindustrie und die Verwendung im Rahmen dieser Arbeit festgelegt. Aufbauend auf einer Beschreibung der Testumgebung wird der derzeitige Forschungsstand der Testfallerstellung in der Automobilindustrie erläutert. Anschließend werden existierende modellbasierte Testansätze aus der Automobilindustrie und verwandten Industrien einsortiert und vorgestellt.

Kapitel 4 stellt die entwickelte modellbasierte Methodik vor. In diesem Zuge werden erst die verwendete Systemzerlegung mittels Schichten, dann die Verwendung von Abstraktionsebenen und anschließend die gemeinsame Verwendung erläutert. Darauf aufbauend werden die Rolle und der Zusammenhang dieser Methodik mit dem modellbasierten Testen verdeutlicht. Am Ende von Kapitel 4 wird ein Ausblick auf die evolutionäre Entwicklung mit der beschriebenen Methodik gegeben.

Kapitel 5 stellt die Kriterien, Anforderungen an die Testmethode und den Testfallgenerator vor. Diese Anforderungen betreffen insbesondere die in Kapitel 2 vorgestellten Modelle. Zusätzlich wird der aktuelle Status quo des modellbasierten Testens anhand einer Studie präsentiert. Auf der Basis der behandelten Anforderungen an die Modelle und des aktuellen Status quo wird der Verwendungszweck der zu erstellenden Testfälle und die dafür notwendigen Eigenschaften erläutert. Am Ende von Kapitel 5 werden die Erwartungen an eine automatisierte modellbasierte Testfallerstellung nochmals zusammengefasst.

Kapitel 6 stellt die verwendete domänenspezifische Sprache vor. Vorweg werden die Herausforderungen an eine domänenspezifische Sprache in dem Kontext dieser Arbeit erläutert. Danach wird die Sprache mit ihrer Grammatik und Kontextbedingungen vorgestellt. Anschließend werden die für die Praxis notwendigen Modellierungsrichtlinien und ihr Verhältnis zum Testfallgenerator erläutert.

Kapitel 7 stellt die Transformation und technische Umsetzung des Testfallgenerators vor. Zu diesem Zweck werden die einzelnen Transformationsschritte von einem Aktivitätsdiagramm bis zum Testfall erläutert. Ergänzend werden die Transformation und der Umgang mit Zustandsdiagrammen vorgestellt. Anschließend werden die erzeugten Artefakte des Testfallgenerators beschrieben.

Kapitel 8 stellt in einer Evaluation die in dieser Arbeit vorgestellten Ergebnisse vor. Die Evaluation umfasst zum einen eine technische Gegenüberstellung der vom Testfallgenerator erzeugten Testfälle mit äquivalenten manuell erstellen Testfällen. Zum anderen umfasst die Evaluation eine Studie über die Anwendbarkeit der Modellierungsrichtlinien für die Modellierenden. Am Ende von Kapitel 8 werden gewonnene Erkenntnisse aus der Einführung des modellbasierten Testens und der notwendigen Methode präsentiert.

Kapitel 9 fasst diese Arbeit abschließend zusammen und gibt einen Überblick über mögliche Anknüpfungspunkte, die aufbauend auf den Ergebnissen dieser Arbeit bearbeitet werden können.

1.3 Publikationen

Die Ergebnisse dieser Arbeit sind in mehreren Jahren Forschung und Projekten entstanden. Aus diesem Grund sind einige Teile bereits vor Veröffentlichung dieser Arbeit publiziert worden. Im Folgenden werden diese Veröffentlichungen kurz erläutert.

- *Status der Testfallerstellung in der Automobilindustrie:* Die Studie über den aktuellen Status der Testfallerstellung in der Automobilindustrie wurde in [KMS+18, DGH+19] veröffentlicht. Zu beachten ist, dass die Veröffentlichung [DGH+19] einen Bearbeitungsfehler beinhaltet. Der Fehler wurde beim Übertrag der automatisch ausgewerteten Daten in [DGH+19] nicht festgestellt und wäre ansonsten behoben worden. Es handelt sich nach [Sta20] um einen geringen inhaltlichen Fehler, deren Größenordnung so gering ist, dass die Aussage unverändert bleibt. In dieser Arbeit wurde der festgestellte Fehler korrigiert veröffentlicht.

- *Testen mit einer Spezifikationsmethode für Anforderungen, Design und Test:* Teile der Spezifikationsmethode für Anforderungen, Design und Test wurde in [KMS+17, KMS+18, DGH+18, DGH+19] vorgestellt. Ein Teil dieser Methode ist die automatisierte Erstellung von Testfällen, die in [KMS+18, DGH+19] vorgestellt wurde.

- *Kennzahlen für die Bewertung von Modellkomplexität:* In [CMRP18] wird ein Verfahren vorgestellt, dass Aktivitätsdiagramme aufgrund bestimmter Kennzahlen bewertet, um die Eindeutigkeit und damit die Verständlichkeit und Testfallquote zu verbessern.

- *Effizientes Testen:* Für ein effizientes Testen muss der Einsatz der erstellten Testfälle organisiert werden. Ein möglicher Ansatz in der Form einer Produktlinien-Testmethodik wird in [EJK+19] vorgestellt.

Darüber hinaus wurden mit studentische Arbeiten Teile der Ansätze und Konzepte der vorliegenden Arbeit evaluiert beziehungsweise die Umsetzbarkeit der Ansätze und die Anwendbarkeit der Konzepte nachgewiesen [Mil17, Koh19, Wes19].

Kapitel 2

Modellbasierte (Software-) Entwicklung

Im vorangegangenen Kapitel wurden die Motivation, der Beitrag und die Einordnung dieser Arbeit beschrieben. In diesem Kapitel werden die Grundlagen für die modellgetriebene Entwicklung zum Verständnis dieser Arbeit erläutert. Zu diesem Zweck wird nach einem Überblick und der Einordnung der Arbeit in die modellgetriebene (Software-) Entwicklung eine Einführung in die modellbasierte Entwicklung in der Automobilindustrie gegeben. Die Grundlagen von Modellierungssprachen und die Spezifikationsmethode für Anforderung, Design und Test (SMArDT) wird anschließend erläutert. Darüber hinaus werden alle relevanten Begriffe definiert, um eine einheitliche Semantik zu gewährleisten.

2.1 Grundlagen der modellbasierten Entwicklung

Die in dieser Arbeit verwendeten Begriffe sowie eingesetzte Software und Hardware richten sich nach der ISO 247965 „Systems and Software Engineering“ [ISO17b]. *Hardware* sind physische Geräte [ISO17b]. Diese physischen Geräte umfassen einen Teil oder ein komplettes Informationssystems [ISO17b]. Das Informationssystem dient der Verarbeitung, Speicherung von Computerprogrammen und deren Datenaustausch. Die Eigenschaften des Informationssystems werden in der Software definiert. Die *Software* ist ein Programm oder ein Satz von Programmen, Verfahren und Regeln, die ebenfalls zugehörige Dokumentation und Daten eines Computersystems umfassen [ISO17b]. Werden Softwarekomponenten, und/oder Hardwarekomponenten, zu einem System kombiniert, spricht man von einer *Integration* [ISO17b]. Ein *System* ist die interagierende Kombination von Elementen, die ein definiertes Ziel verfolgen [ISO17b, BF14, Pat82]. Die Elemente eines Systems können abhängig vom Kontext Hardware, Software, Menschen, Informationen, Techniken, Einrichtungen, Dienste und andere Unterstützungselemente sein [ISO17b, BF14, Pat82]. Zum Beispiel ist der Antrieb eines Fahrzeugs ein (Sub-) System des Systems „Fahrzeug“. Das Fahrzeug andererseits ist ein Teil eines Mobilitätssystems.

Systeme werden insbesondere bei interdisziplinären Projekten, wie beispielsweise in der Automobilindustrie, komplexer [Bro06, PBKS07, SZ16]. Ein mittel diese Komplexität zu bewältigen ist das Systems Engineering. Die Methoden aus dem Systems Engineering werden in dieser Arbeit mit Verfahren dem Software Engineering kombiniert. Diese Verfahren des Software Engineering passen sich den individuellen Projekten an [Rum16]. Ein Ansatz und Konzept des Software Engineering ist die Nutzung von Modellen. Mit Modellen kann von der komplexen Realität abstrahiert werden, um einzelne Aspekte

intellektuell zu beherrschen.

Nach Stachowiak [Sta73] besitzt ein *Modell* drei Merkmale.

- Ein Modell repräsentiert, und/oder bildet ein Original ab, wobei dieses Original selbst wieder ein Modell sein kann.
- Ein Modell ist eine Abstraktion seines repräsentierten Originals und umfasst nur relevant („scheinende“) Eigenschaften des Originals.
- Ein Modell erfüllt mit dieser Abstraktion des Originals immer einen Zweck.

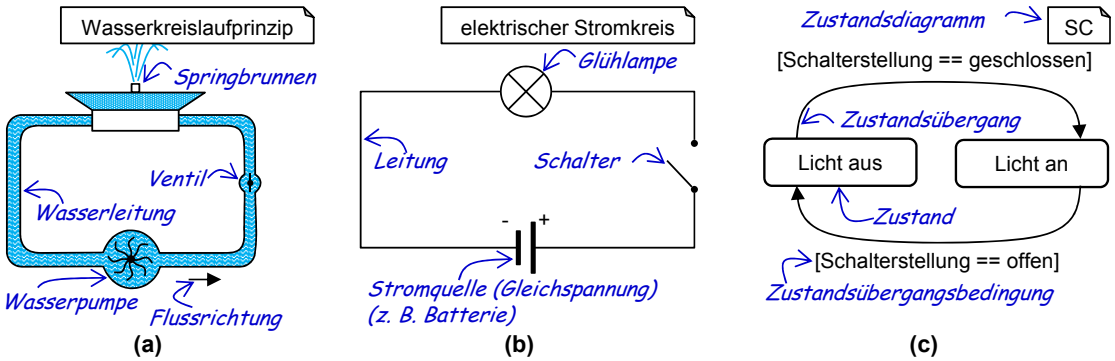


Abbildung 2.1: Drei Modelle eines Schaltkreises mit einer Glühlampe, Batterie und Schalter. Jedes Modell dient einem anderen Zweck: Modell (a) ist eine vereinfachte Erklärung des Stromflusses, Modell (b) ist der elektrischer Stromkreises und Modell (c) ist ein Zustandsdiagramm des Schaltkreises

Abbildung 2.1 illustriert verschiedene Modelle eines Originals. Das Original ist ein Schaltkreis aus einer Glühlampe, die über Kabel mit einem Schalter an einer Batterie angeschlossen ist. Mit dem Schalter ist es möglich die Glühlampe an und auszuschalten. Modell (a) aus Abbildung 2.1 dient dem Zweck der Erklärung des Stromflusses mittels eines Wasserkreislaufes. Hierfür wurde vom Original in einer Weise abstrahiert, um anhand der Repräsentanten vereinfacht das Prinzip des Stromflusses und der elektrischen Spannung erklären zu können. Modell (b) abstrahiert ebenfalls vom Original in der Form eines elektrischen Stromkreises und illustriert die Verschaltung und Art der elektrischen Baugruppen. Modell (c) bildet die generelle Funktionsweise mit einem Zustandsdiagramm (engl. „State Chart“) (SC). Das Modell (a) dient nur dem Zweck der erleichterten Erklärung, wohingegen die Modelle (b) und (c) neben einem beschreibenden Charakter auch weitere Verwendungszwecke, wie zum Beispiel der Simulation, eröffnen.

Mit Modellen kann je nach Absicht ein anderer Zweck verfolgt werden. Bei der Verwendung von Modellen wird in dem Software Engineering zwischen modellbasierten und modellgetriebenen Ansätzen unterschieden [BWC12, Sch12]. In der *modellbasierten Entwicklung* spielen Modelle eine wichtige Rolle für die Entwicklung eines Produktes, sind aber nicht notwendigerweise die primären Artefakte für die Entwicklung [BWC12]. Zum Beispiel dienen die Modelle oft nur der Dokumentation, als Vorlage für die Entwicklung

oder für die Testfallerstellung. Für diese Zwecke müssen die Modelle nicht zwangsläufig auf vorgegebenen Regeln und Richtlinien beruhen.

Nach [Gre19] ist ein *Artefakt* „eine individuell speicherbare Einheit mit einem eindeutigen Namen, die einem bestimmten Zweck im Kontext des Software-Engineering-Prozesses dient.“ Artefakte können folglich in der Software individuell speicherbare Einheiten, wie ein Unified Modeling Language (UML)-Modell, ein Systems Modeling Language (SysML)-Modell, eine .java-Datei oder eine .txt-Datei sein.

Im Gegensatz zur modellbasierten Entwicklung sind Modelle in der modellgetriebenen Entwicklung die maßgeblichen, wichtigsten und wesentlichen (primären) Artefakte und werden nicht allein zu Dokumentationszwecken erstellt werden [BWC12]. Die Modelle werden genutzt, um die Absichten von Entwickler*innen präzise zu erfassen, anstatt diese nur informal auszudrücken [GSCK04]. Diese Modelle können dann genutzt werden, um durch Verfeinerungen (teilweise) automatisiert Implementierungen zu erzeugen. Für diesen Zweck müssen die Modelle abstrakt und formal zugleich sein [SV06]. Wobei in diesem Zusammenhang der Begriff *Abstrakt* nicht für Vagheit, sondern für Kompaktheit und Reduktion auf das Wesentliche steht [SV06], wie in der bereits eingeführten Definition des Modells. Das Ziel der modellgetriebenen Entwicklung ist, die Modelle für effektive und effiziente Ansätze zu nutzen. Häufig werden für die Effizienzsteigerung maschinenlesbare Modelle verwendet, um aus diesen Modellen (semi-)automatisiert Artefakte zu erstellen [BWC12]. Notwendige Eigenschaften für das Verarbeiten maschinenlesbarer Modelle sind die *Wohlgeformtheit* und *Wohldefiniertheit*. Nach [Grö10] ist ein Modell wohlgeformt, falls es syntaktisch korrekt ist und insbesondere die Kontextbedingungen erfüllt. Die *Kontextbedingungen* für eine Sprache schränken die Menge der erlaubten Modelle ein und ergänzen damit die Definition der Syntax [Grö10]. Die Wohlgeformtheit ist demzufolge eine hinreichende Voraussetzung, aber keine notwendige, damit ein Modell maschinenlesbar ist. In dieser Arbeit ist ein Modell wohldefiniert, falls die formale Semantik eines Modells nicht leer ist und alle verwendeten Elemente der Modell-Sprache eindeutig definiert sind. Die Wohldefiniertheit ist demzufolge eine hinreichende aber nicht notwendige Voraussetzung damit ein Modell eine Bedeutung hat, die gleich interpretiert wird, sowohl von Menschen als auch von Maschinen. Ein Modell, das wohlgeformt und wohldefiniert ist, wird in dieser Arbeit als *formales Modell* bezeichnet.

Für die Definition von Modellen dienen in dieser Arbeit Modellierungssprachen. Es werden Universelle Modellierungssprachen (engl. „General Purpose Modeling Languages“) (GPMLs) und domänenspezifische Sprachen (engl. „Domain Specific Languages“) (DSLs) beziehungsweise domänenspezifische Modellierungssprachen (engl. „Domain Specific Modeling Languages“) (DSMLs) unterschieden [BWC12, FP11]. Wie der Name impliziert sind GPMLs nicht auf eine Domäne[1] spezialisiert. DSLs hingegen sind auf eine bestimmte Domäne fokussiert und weisen begrenzte Ausdrucksmöglichkeiten auf [BWC12, FP11]. Geläufige Beispiele für eine GPML sind die UML [OMG17b] und deren Ableger die SysML [OMG17a]. Der Übergang zwischen DSMLs und GPMLs ist oft fließend, da GPMLs zum Beispiel von Profilen für bestimmte Domänen und Bereiche angepasst werden

[1] Eine Domäne, auch Produktdomäne, ist ein Spezialgebiet, in dem Expert*innen mit Fachwissen arbeiten und sich besonders gut auskennen [Dud20]. Beispiele sind die Domänen Automobil, IT, Medizin etc.

können [Höl18]. Zum Beispiel wurde mit der Modellierungssprache UML/Programmiergeeignet (UML/P) [Rum16, Sch12] die UML reduziert, um sie für die Programmierung zu eignen. Sowohl die UML als auch die SysML umfassen Modellierungssprachelemente zur Struktur- und Verhaltensmodellierung. In dieser Arbeit liegt der Fokus auf der SysML und deren Verhaltensmodellierung.

Nachdem in diesem Abschnitt die allgemeinen Grundlagen der modellgetriebenen Softwareentwicklung kurz erläutert wurden, werden in den folgenden Abschnitten die modellbasierte Entwicklung in der Automobilindustrie in Abschnitt 2.2, relevante Modellierungssprachen in Abschnitt 2.3, die Grundlagen von SMArDT in Abschnitt 2.4 und verwandte modellbasierte Ansätze in Abschnitt 2.5 erläutert.

2.2 Entwicklung in der Automobilindustrie

Traditionell wird in der Automobilindustrie das System Fahrzeug in die Subsysteme Antriebsstrang, Fahrwerk, Karosserie, Multi-Media sowie Fahrerassistenz unterteilt [SZ16]. Die Subsysteme können erneut in weitere untergeordnete Subsysteme bis hin zu Komponenten aufgegliedert werden. Diese traditionell größtenteils entkoppelten (Sub-) Systeme ermöglichen eine Arbeitsteilung, parallele Entwicklung und Vergabe an externe Zulieferer [SZ16]. Die unterteilten Subsysteme können entweder wiederverwendet oder mittels Simulationen und Rapid-Prototyping[2] frühzeitig validiert werden [SZ16]. Ein *Prototyp* ist ein Muster einer geplanten großen Serie von Produkten [SZ16]. Häufig wird in der Fahrzeugtechnik zwischen Hardware- und Software-Prototypen unterschieden, da unter anderem die Vervielfältigung von Software einen weniger technischen Aufwand als die Vervielfältigung von Hardware mit sich bringt [SZ16]. Über Prototypen können schnell Ergebnisse erbracht werden, um so ein frühzeitiges Feedback zu erhalten. Anschließend werden die validierten Subsysteme mit den anderen Subsystemen in das System integriert.

Mit einer steigenden Kopplung, Kommunikation und systemübergreifenden Funktionen der Subsysteme wird die Integration komplizierter, da mehr Schnittstellen und Funktionalitäten abgeglichen werden müssen. Eine *Funktionalität* wird durch einzelne Funktionen oder durch semantisch zusammengehörige Funktionsgruppen wie Klassen, Schnittstellen oder Komponenten beschrieben [Bal09]. In dieser Arbeit wird die Begriffsdefinition der Funktion aus der Mathematik verwendet. Eine *Funktion* beschreibt eine Relation zwischen der Eingabemenge (engl. „Input“) und der sich daraus ergebenden Ausgabemenge (engl. „Output“) [Bal09]. Mit anderen Worten bewirkt eine Funktion in dieser Arbeit eine Veränderung des Inhalts oder der Struktur auf der Basis des spezifischen Inputs wie Informationen, Material oder Energie. Eine Funktion kann für Benutzer oder andere Funktionen und Systeme nutzbar und/oder sichtbar sein. Die Funktionalität steht folglich für andere Funktionen eines Systems oder einer Systemelementes, unabhängig von der Sichtbarkeit von Informationen, Material oder Energie zur Verfügung.

[2] Rapid-Prototyping bezeichnet die Ausführung von Software-Funktionen auf einem Rechner mit Schnittstellen zum Fahrzeug oder Komponenten [SZ16]. Bei der Hardware und der Software der Fahrzeuge und deren Komponenten handelt es sich in der Entwicklungsphase häufig um Prototypen mit begrenzten Funktionalitäten.

Mit der steigenden Komplexität der Systeme und deren Funktionalitäten erhöht sich neben der Systemkommunikation während der Laufzeit auch der Bedarf an Kommunikation zwischen den involvierten internen und externen Entwicklungsparteien. Die klassische Vergabe mittels Anforderungen in Form eines *Lastenhefts*[3] und das Integrieren der Systeme wird mit der systemübergreifenden Komplexität ebenfalls erschwert. Die Beherrschung dieser Komplexität erfordert einen Start eines Umdenkprozesses weg von einer Bottom-up hin zu einer systematischen Top-down Entwicklung mit einer ständigen Reflexion und Kommunikation [Bro06, SZ16].

Gründe sind, dass ein nicht koordiniert unabhängiges Aufteilen und Entwickeln von übergreifenden Funktionalitäten und Systemen erheblichen Abstimmungs- und Mehraufwand bei der Integration verursachen und oft nicht alle Benutzeranforderungen von Anfang an vollständig bekannt sind. Die gewünschten Systemfunktionen sind in CPS mehr als die Summe ihrer Teile (emergentes Verhalten) und können nicht mehr effektiv Bottom-up entwickelt werden, sondern müssen mittels des Top-down-Prozesses ausgehend vom gesamten Fahrzeug spezifiziert werden. Auch nachträglich mit aufgenommene Benutzeranforderungen, die häufig eine systemübergreifende koordinierte Funktion nach sich ziehen, können so kompensiert werden. Theorie und Praxis 2.1 konkretisiert die Begriffe Bottom-up und Top-down für diese Arbeit.

Theorie und Praxis 2.1 (Bottom-up und Top-Down in der Automobilindustrie)**.** *In der Automobilindustrie werden traditionell vor allem die einzelnen Hardwarekomponenten auf der Basis eines Lastenheftes Top-Down intern oder extern entwickelt. Anschließend werden die entwickelten Komponenten Bottom-up integriert. Ein komponentenübergreifender Top-down Entwicklungsansatz ohne Lücken und Unterbrechungen über das komplette Fahrzeug hat sich allerdings noch nicht etabliert (vgl. Abschnitt 2.5). Aufgrund dessen erhöht sich der Aufwand der Bottom-up Integration der Komponenten in das Fahrzeug, da Lücken in der Spezifikation zu Fehlern führen können. Da der Fokus der Aufwände auf den Komponenten und deren Integration statt auf der vorab Definition der Anforderungen und der Architektur liegt, wird in dieser Arbeit die traditionelle Entwicklung der Automobilindustrie als Bottom-up charakterisiert.*

Zwei wichtige Faktoren für die (simultane) interdisziplinäre Entwicklung in der Automobilindustrie sind ein gemeinsames Verständnis und eine Standardisierung von Entwicklungsmethoden. Als Grundlage für ein gemeinsames interdisziplinäres Verständnis wie Maschinenbau, Elektrotechnik, Elektronik und Informatik können grafische Modelle dienen. Besonders in der Softwareentwicklung lösen zunehmend modellbasierte Software-Entwicklungsmethoden die Spezifikationen in natürlicher Sprache („Prosa“) ab [Rum16, Rum17, SZ16]. Auf der Basis der Modelle können weitere standardisierte Vorgehensmodelle wie das V-Modell [Koo19], die CMMI [CMM19, CKS11], die SPICE [ISO12], die ISO 26262 [ISO18a, ISO18b] und eine standardisierte Softwarearchitektur wie AUTOSAR [AUT19] aufgebaut werden.

[3] Das (Produkt) Lastenheft ist die Grundlage für die externe oder interne Vergabe von zu entwickelnden Systemen und enthält „alle“ an das System gestellte Anforderungen [Koo19].

2.2.1 Das V-Model

Das V-Modell 97 [Boe79, BD95] ist ein Vorgehensmodell und definiert einen Rahmen für den organisatorischen Prozess der Entwicklung von Software [Koo19, Bal09]. Das V-Modell 97 wurde im Jahr 2005 vom V-Modell XT [Koo19] abgelöst. Während das V-Modell 97 den Fokus auf die Aktivitäten im Prozess richtet, ist das V-Modell XT produktzentriert. Produktzentriert bedeutet, dass die auf das Produkt bezogenen Projektergebnisse im Vordergrund stehen und folglich den Zusammenhang und die Abhängigkeiten der Ergebnisse beschreiben. Neben einer Fokussierung auf das Produkt wurde im V-Modell XT zudem eine Modularisierung mittels einzelner Bausteine geschaffen (so genanntes „tailoring“) und die zeitliche Abfolge der zu erledigenden Aufgaben gelockert. Im Folgenden wird das V-Modell XT Version 2.3 [Koo19] als V-Modell bezeichnet, da es sich um die aktuelle Version des V-Modells handelt und alle Anforderungen des V-Modells 97 erfüllt.

Für die Entwicklung von Software hat das V-Modell eine hohe Relevanz, da es der Standard für Softwareentwicklungsprojekte in Deutschland ist und die Bundesverwaltung „dringend empfiehlt“ das V-Modell anzuwenden [Koo19]. Im V-Modell stehen die Produkte im Fokus. *Produkte* sind Elemente, die Zwischen- oder auch Endergebnisse des Projekts sein können. Abbildung 2.2 illustriert eine Übersicht über den Entwicklungsprozess mit dem V-Modell, wobei hier der Fokus auf der Entwicklung von Funktionalitäten liegt.

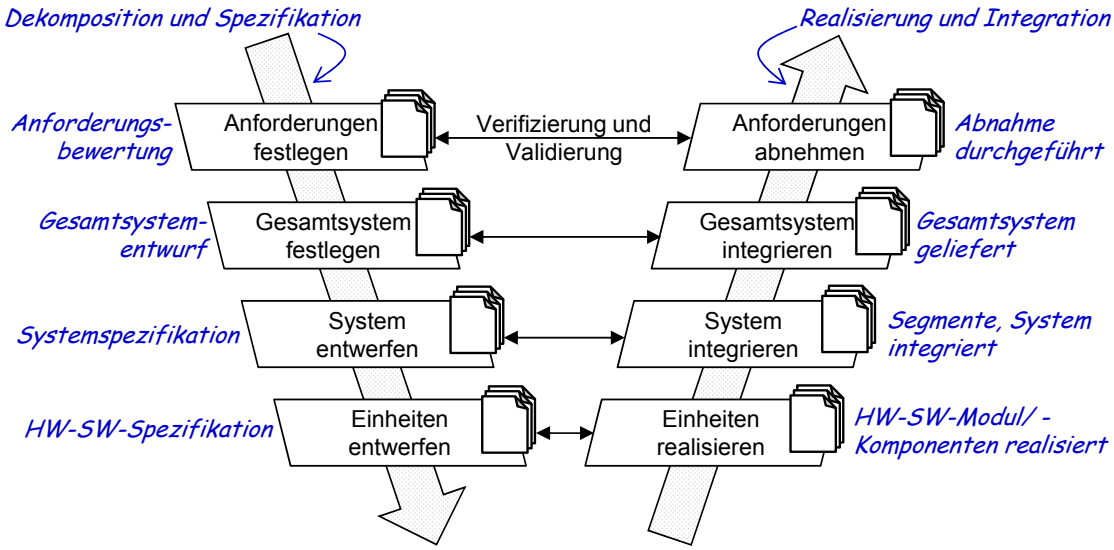


Abbildung 2.2: Übersicht des V-Modell-Entwicklungsprozesses nach [Koo19]

Auf der linken absteigenden Seite des „Vs“ findet eine Zerlegung, auch *Dekomposition* genannt, und Spezifizierung des Systems statt [Koo19]. Ein System kann in weitere Subsysteme unterteilt werden bis auf eine unterste Ebene aus Hardware- und Software-Einheiten. Wird die unterste Ebene eines Systems aus Hardware- und Software-Einheiten zusammen gesetzt, spricht man von einem *eingebetteten System* (engl. „embedded system“).

Die Ziele eines Systems werden in den Anforderungen definiert. Diese Anforderun-

gen werden in der ersten Ebene des V-Modells bewertet und festgelegt. *Anforderungen* sind qualitative und/oder quantitative Bedingungen oder Fähigkeiten, die ein System, Subsystem oder Produkt erfüllen und/oder besitzen muss oder soll [ISO17b]. Anforderungen beinhalten folglich Bedürfnisse und Erwartungen der Kund*innen und anderer Interessengruppen, wie zum Beispiel Endkund*innen oder andere Systeme [DIN17].

Nach der Bewertung und Definition der Anforderung kann entschieden werden, ob die Anforderungen, von Arbeitgeber*innen beauftragt und an Arbeitnehmer*innen in der Form eines Lastenhefts übergeben oder selbst weiter spezifiziert werden. In einer *Spezifikation* werden die Anforderungen und die Funktionsweise von Systemelementen und deren logistische Unterstützung detailliert beschrieben [Koo19]. Die Spezifikation dient der Entwicklung oder Validierung des Systems, Subsystems und/oder des Produkts [ISO17b]. Neben Anforderungen und der Schnittstellenbeschreibung sind folglich ebenfalls die interne Schnittstellenrealisierung und das interne Verhalten dargelegt.

Die Spezifikation kann nach der ersten Ebene unabhängig vom Arbeitgeber oder Arbeitnehmer auf der nächsten Ebene, der Gesamtsystemebene, weitergeführt werden. Dabei handelt es sich um einen ersten Entwurf des Gesamtsystems. Das *Gesamtsystem* ist ein System, das aus einem Verbund heterogener Einzelsysteme (Subsystemen) besteht. Der abstrakte Entwurf kann erste Sicherheits- und Testkonzepte enthalten. Diese Konzepte dienen als Basis für die Systemspezifikation in der nächsten dritten Ebene. Zusätzlich zu dem Umfang der zweiten Ebene kann die dritte Ebene neben einer Systemarchitektur, unter anderem ein Implementierungs- und Integrationskonzept umfassen. Auf der folgenden vierten Ebene der Dekompositions- und Spezifikation-Seite werden die Software- und Hardware-Einheiten entworfen. Neben dem Umfang der dritten Ebene werden hier explizit die Hardware- und Software-Funktionalitäten spezifiziert. Das gewünschte System ist nun spezifiziert und kann implementiert werden.

Auf der rechten aufsteigenden Seite des „Vs“ findet die Realisierung und Integration des spezifizierten Systems statt. Die *Realisierung* ist die Verwirklichung beziehungsweise die Umsetzung der Spezifikation [Koo19]. Sie beinhaltet Eigenschaften und/oder einen Algorithmus, der die Spezifikation umsetzt [ISO17b]. Die Ebenen der Realisierung und Integration entsprechen der Ebenen der Dekomposition und Spezifikation. Auf der vierten untersten Ebene werden die Komponenten und Module der Hardware und Software realisiert und geprüft. Nach [Koo19] sind *Module*, die atomaren Elemente der Hardware und Software. Beispiele sind für die Hardware ein Interface-Element und für die Software die Klasse „Elektrischer Antrieb“. Ein Modul ist ein Teil einer Komponente. Eine *Komponente*[4] ist ein Element eines Systems[5] [ISO17b]. Komponenten erfüllen eine

[4]Nach [Koo19] ist die Komponente ein Element, das ausschließlich aus Hardware oder Software besteht. In dieser Arbeit wird der Begriff „Komponente“ beziehungsweise Bauelement aus der Norm des Systems und Software Engineering [ISO17b], der Norm für elektrische Betriebsmittel [DIN18a] und der Fahrzeugtechnik [PS16] verwendet, da dies sich in der Domäne der Automobilindustrie etabliert hat. Falls es sich um eine reine Hardware- oder Software-Komponente nach [Koo19] handelt, wird dies explizit als Hardware- oder Software-Komponente ausgeschrieben.

[5]Nach [Koo19] ist die Komponente ein Teil einer Einheit. *Hardware-* beziehungsweise *Software-Einheiten* sind die am hierarchisch höchsten stehenden reinen Hardware- oder Software-Elemente. Die der Komponente übergeordnete Kategorie „Einheit“ wird bewusst ausgelassen.

oder mehrere bestimmte Funktionen, die für das Element höherer Ordnung von Bedeutung ist [ISO17b, DIN18a]. Komponenten können hierarchisch in weitere Komponenten unterteilt werden und sind zum Beispiel ein Steuergerät für Koordination der Energieversorgung im Fahrzeug. Diese realisierten Komponenten werden in der dritten Ebene des V-Modells zu Segmenten beziehungsweise Subsystemen zusammengefasst. Die Segmente werden zusammen mit externen Modulen in das System integriert und geprüft. Auf der folgenden zweiten Ebene wird das System in das Gesamtsystem aufgenommen. Falls die Entwicklung von einem Arbeitgeber an einen Arbeitnehmer beauftragt wurde, entspricht der Vorgang auf der zweiten Ebene einer Lieferung des Systems. Auf Basis der festgelegten Anforderungen wird auf der obersten ersten Ebene geprüft, ob die Anforderungen erfüllt werden. Im Falle einer Arbeitnehmer-Beauftragung wird auf dieser Ebene das System abgenommen.

Auf jeder der beschriebenen horizontalen Ebenen findet eine Verifizierung und Validierung (V&V) statt. Die *Verifizierung* oder Verifikation ermittelt die Übereinstimmung der Spezifikation mit dem einwickelten (Software-) Produkt [Boe79, ISO17b]. Die Verifizierung beantwortet die Frage „Baue ich das Produkt richtig?“ [Boe79]. Die *Validierung* ermittelt die Eignung beziehungsweise den Wert eines (Software-) Produktes für den Betrieb [Boe79, ISO17b]. Die Validierung beantwortet die Frage „Baue ich das richtige Produkt?“ [Boe79].

Das „V“ des V-Modells kann mehrfach iterativ durchlaufen werden. Diese Vorgehensweise ermöglicht eine iterative Verfeinerung der horizontalen Abstraktionsebenen.

2.2.2 ISO 26262

Die ISO 26262 [ISO18a, ISO18b] ist eine Normenreihe und definiert ein Vorgehensmodell, das auf dem V-Modell basiert. Sie passt die Normenreihe IEC 61508 [DIN11] an die Bedingungen der Automobilindustrie an. Die Anpassung umfasst insbesondere die Gefahreneinstufung, da nicht jede Gefährdung in einem Fahrzeug zu einem Unfall führt. In diesem Zuge wurde die in der IEC 61508 relevante, probabilistische Sicherheitsanforderungsstufe (engl. „Safety Integrity Level“) durch die qualitative Automotive Sicherheitsanforderungsstufe (engl. „Automotive Safety Integrity Level“) (ASIL) ersetzt.

Mit dieser Herangehensweise werden Gefahren und gefährliche Ereignisse, die verhindert oder kontrolliert werden müssen, identifiziert. Die identifizierten Ereignisse werden in fünf verschiedene Sicherheitsanforderungsstufen, QM, ASIL A, ASIL B, ASIL C und ASIL D eingeteilt, wobei QM ohne in der Norm vorgesehene Maßnahmen auskommt und ASIL D die höchsten Anforderungen und Maßnahmen nach sich zieht. Neben der Einteilung in Stufen werden die Ereignissen mit Sicherheitsstufen A bis D und dementsprechend Sicherheitszielen zugeordnet, denen mittels funktionalen oder technischen Sicherheitskonzepten begegnet werden muss.

Ein wichtiges Element der Normenreihe und in dieser Arbeit fokussiertes Element der ISO 26262 ist die Verifizierung. Diese Verifizierung kann durch Techniken wie Rezensionen, Simulation, Analyse oder Testen erfolgen. In dieser Arbeit steht die Technik des Testens im Fokus, welches in Kapitel 3 näher erläutert wird.

Für die Erstellung von (Sicherheits-)Anforderungen wird ein Systemdesign empfohlen,

welches von der Aufgabendefinition bis hin zum Systemarchitekturdesign kontinuierlich (weiter-)entwickelt wird. Ein solches durchgängiges Systemdesign wird in Kapitel 4 vorgestellt. Wie bei dem V-Modell ist die Anwendung der ISO 26262 freiwillig, doch in Zeiten von komplexer werdenden automobilen Systemen rückt die ISO 26262 mit ihren Sicherheitsthemen mehr in den Blickfeld der Automobilindustrie [Ros16].

2.3 Verwendete Modellierungssprachen

Eine für diese Arbeit elementare GPML ist die SysML. Die SysML [OMG17a] ist von der Object Management Group (OMG) [OMG19] veröffentlicht und erweitert eine Teilmenge der UML 2.5 [OMG17b] für die Anforderungen physischer Systeme. Insbesondere die „Trace-“ und die Verfeinerungsstereotypen für die Rückverfolgbarkeit von Anforderungen wurden erweitert [OMG17a]. Abbildung 2.3 klassifiziert die einzelnen Diagramme der SysML und stellt sie in Zusammenhang mit der UML 2.5 und dieser Arbeit.

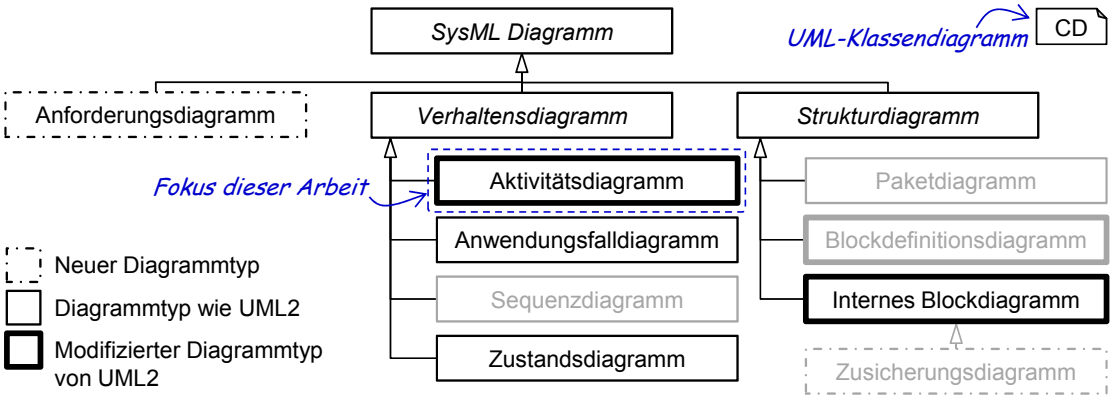


Abbildung 2.3: Klassendiagramm der SysML Diagramme nach [OMG17a] (für diese Arbeit nicht relevante Diagramme in grau)

Die Menge der Modellierungssprachen der SysML dienen zur Struktur-, Verhaltens- und Anforderungsmodellierung. In dieser Arbeit liegt der Fokus auf der Verhaltensmodellierung und insbesondere auf dem Aktivitätsdiagramm (AD)[6]. Neben einer Spezifikation der SysML-Diagramme ist in [OMG17a] vermerkt, dass die SysML an domänenspezifische Modellanwendungen wie Automobil, Luft- und Raumfahrt, Kommunikation und Informationssysteme angepasst wird. Ein Werkzeug für eine solche domänenspezifische Anpassung ist die MontiCore Language Workbench.

Im Folgenden wird in Unterabschnitt 2.3.1 die für diese Arbeit relevanten Elemente des ADs erläutert und anschließend in Unterabschnitt 2.3.2 die zum Verständnis dieser Arbeit notwendigen Elemente der MontiCore Language Workbench vorgestellt.

[6]Die geläufigen Abkürzungen wie AD und SC sind in der SysML [OMG17a] als ATC für das Aktivitätsdiagramm und STM für das Zustandsdiagramm spezifiziert. Da sich die Abkürzungen AD und SC bereits etabliert haben, werden AD und SC in dieser Arbeit verwendet. Dies gilt auch für das Anwendungsfalldiagramm (engl. „Use Case Diagram“) (UCD).

2.3.1 SysML-Aktivitätsdiagramm

Für eine umfassende Erläuterung des ADs und der anderen Struktur- und Verhaltensdiagramme, wie SCs, Internen Blockdiagrammen (IBDs) und UCD ist auf [Rum16, OMG17a, OMG17b] zu verweisen.

Das AD ist ein Verhaltensdiagramm, beschreibt den Ablauf von Aktivitäten und entspricht einem gerichteten Graphen [GD18] mit Knoten und Kanten. Eine *Aktivität* modelliert das Verhalten eines Systems mittels Aktionen und deren Vernetzung mittels Kontroll- und Datenflüssen, Entscheidungen und synchronen Abläufen [OMG17a].

Abbildung 2.4 illustriert beispielhaft ein AD, das den Ladevorgang eines Elektrofahrzeuges modelliert. Der Ablauf der Aktivität, hier `Fahrzeug laden`, startet mit dem

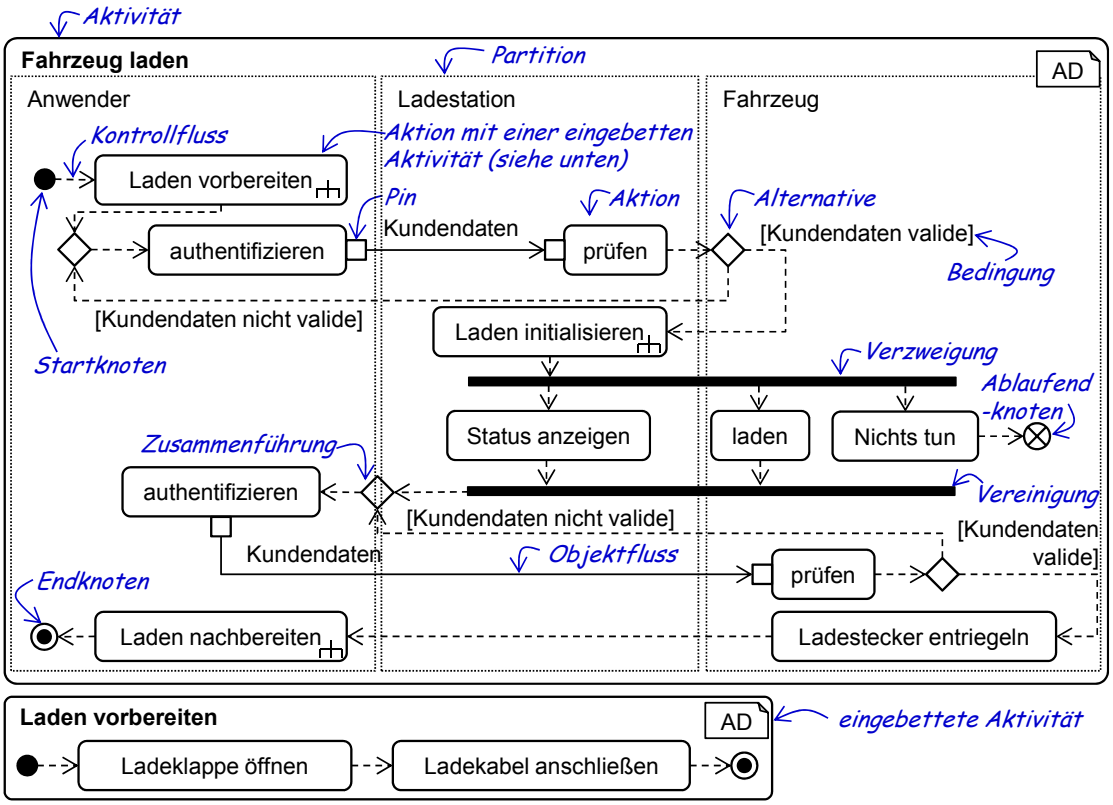


Abbildung 2.4: Vereinfachte Darstellung eines Ladevorganges mit den für diese Arbeit relevanten AD-Elementen

Startknoten (●-->). Ausgehend von dem Startknoten spezifizieren *Kontrollflüsse* (-->[7]) die Reihenfolge der ausführbaren Aktionen über ausgehende Kanten. Kontrollflüsse können Bedingungen als Beschriftung enthalten. Diese *Bedingungen* sind nach Alternativen relevant, um eine Entscheidung des Ablaufs treffen zu können. Bedingungen werden mittels

[7] Kontrollflüsse können in SysML ADs als gestrichelte Pfeile modelliert werden, wohingegen sie in UML-ADs als durchgezogene Pfeile modelliert werden [OMG17a].

eckigen Klammern `,[‘<Bedingung>,]‘` ausgedrückt und können entweder wahr oder falsch sein. *Aktionen*, wie zum Beispiel `authentifizieren`, sind ausführbare Knoten und die elementaren Schritte in einer Aktivität. Jede Aktion führt das für sie modellierte und spezifizierte Verhalten aus. Ist eine Aktivität in eine Aktion *eingebettet* (⏄), wie zum Beispiel `Ladeklappe öffnen` in `Laden vorbereiten`, wird bei einem Aufruf der Aktion die darin eingebettete Aktivität ausgeführt. *Objektflüsse* (□→□) sind, wie Kontrollflüsse, Kanten und modellieren den Fluss von Informationen zwischen Objekten. Sie verlaufen immer von einem Ausgangspin zu einem Eingangspin, was sie explizit von Kontrollflüssen unterscheidet. *Pins* (□) sind Objektknoten, die Objekte (Daten) enthalten. Im Beispiel aus Abbildung 2.4 werden die `Kundendaten` über den Objektfluss übertragen. In der SysML können sowohl diskrete als auch kontinuierliche Ströme, entweder von Material, Energie oder Informationen in Objektflüsse übertragen werden [OMG17a]. Entscheidungen des Ablaufs werden mittels Alternativen modelliert. *Alternativen* (⇢◇⇢) sind Knoten, die sich für einen der ausgehenden Kanten entscheiden. Eine Alternative besitzt genau eine eingehende Kante[8] und mindestens eine ausgehende Kante. Ist die eingehende Kante ein Kontrollfluss, so müssen alle ausgehenden Kanten ebenfalls Kontrollflüsse sein. Ist die eingehende Kante ein Objektfluss, so müssen auch alle ausgehenden Kanten Objektflüsse sein[9]. Die aus den Alternativen ausgehenden Kanten haben Bedingungen, wie zum Beispiel die Bedingung `Kundendaten valide`. Ist die Bedingung einer Kante erfüllt, entscheidet sich die Alternative für die Kante, die den Wert der Bedingung erfüllt. Erfüllen mehrere Bedingungen einen Wert, wird eine Kante festgelegt. Welche Kante bei zahlreicher Übereinstimmung gewählt wird, ist in der [OMG17a] nicht vorgegeben. Diese Unterspezifikation (beziehungsweise Nichtdeterminismus) kann von methodischem Interesse sein, beispielsweise falls das Verhalten eines Objekts noch nicht genau festgelegt werden kann oder soll [Rum16]. Allerdings ist diese eine Art von nichtdeterministischen Verhalten und es sollte durch nur eine Bedingung pro Wert oder zusätzliche Object Constraint Language (OCL)-Bedingungen[10] vermieden werden [Rum16]. Infolgedessen darf auch die Bedingung `sonst` (engl. „`else`“) verwendet werden. Die `sonst`-Bedingung wird dann akzeptiert, wenn keine andere Bedingung zutrifft. Das Gegenstück der Alternative ist die Zusammenführung. Eine *Zusammenführung* (⇢◇⇢) führt die Abläufe der Alternative zusammen. Eine Zusammenführung hat genau eine ausgehende Kante und mindestens eine eingehende Kante. Analog zu der Alternative müssen alle Kanten entweder alle Kontrollflüsse oder alle Objektflüsse sein und eine Vermischung ist nicht möglich. Im Beispiel Abbildung 2.4 wird, nachdem der `Anwender` in der Aktion `authentifizieren` seine `Kundendaten` übermittelt hat, diese von der `Ladestation` in der Aktion `prüfen` validiert. Sind die `Kundendaten nicht valide` muss sich der Anwender

[8] Die Möglichkeit von zwei eingehenden Kanten, von denen die zweite als Entscheidungs-Eingangs-Fluss (Objektfluss) zu identifizieren ist, wird in dieser Arbeit vernachlässigt.

[9] Die SysML verbietet nicht explizit die Kombination von Kontroll- und Objektflüssen an einem Knoten-Element. Allerdings können in physischen Systemen keine Objekte aus dem nichts entstehen oder einfach verschwinden. Infolgedessen „müssen“ im Laufe dieser Arbeit bei Knoten und deren ein- und ausgehenden Kanten eine identische Kantenart besitzen.

[10] Die OCL ist eine formale textuelle Sprache, mit deren Hilfe notwendige Kontextbedingungen bei der Modellierung festgelegt werden können [OMG14].

erneut authentifizieren. Sind die `Kundendaten valide` kann die Ladestation das `Laden initialisieren`. Wer oder was die einzelnen Aktionen ausführt, wie in diesem Beispiel der `Anwender` oder die `Ladestation`, wird mittels *Partitionen* modelliert. Partition, dienen zur Identifizierung und Gruppierung von Aktionen, die individuell festgelegte Eigenschaften gemeinsam haben. Sie werden verwendet, um Merkmale oder Ressourcen diesen gruppierten Aktionen zuzuordnen. Parallele Abläufe, wie die Aktionen `Status anzeigen` und `laden` werden mittels Verzweigungen modelliert. *Verzweigungen*(⇢|<) teilen den Kontroll- oder Objektfluss in mehrere parallele Abläufe. Eine Verzweigung hat genau eine eingehende Kante und mindestens eine ausgehende Kante. Analog zu der Alternative und der Zusammenführung müssen alle Kanten entweder alle Kontrollflüsse oder alle Objektflüsse sein. Eine Vermischung der Kantenarten wird im Rahmen dieser Arbeit nicht verwendet. Anders als bei der Alternative wird der ankommende Kontroll- oder Objektfluss kopiert und an alle ausgehenden Kanten verteilt. Das Korrelat der Verzweigung ist die Vereinigung. *Vereinigungen* (⇢|>) sind Knoten, die mehrere Abläufe der Verzweigung zusammenführen. Anders als bei der Zusammenführung werden alle ankommenden Kontroll- und Objektflüsse verschmolzen. Analog zu der Verzweigung wird eine Vermischung der Kantenarten nicht verwendet. Handelt es sich bei der eingehenden Kante um einen Objektfluss mit verschiedenen Objekten, werden diese Objekte weitergereicht. Identische Objekte werden vereinigt und nur einmal weitergereicht. Für eine Weitergabe eines Kontroll- oder Objektflusses müssen alle Flüsse eingegangen sein (implizite „UND“ Semantik)[11]. Für den gewollten Abbruch eines Flusses in einer Aktivität gibt es den *Ablaufendknoten* (⇢⊗). Bei einem Ablaufendknoten wird die Aktivität nur dann beendet, wenn keine weiteren Flüsse in der Aktivität aktiv sind. Im Gegensatz zu einem Ablaufendknoten stoppen mit dem Endknoten alle Flüsse der zugehörigen Aktivität und der eingebetteten Aktivitäten. Der *Endknoten* (⇢◉) ist infolgedessen der finale Knoten einer Aktivität. Die einzelnen Abläufe beziehungsweise Folgen von Knoten und gerichteten Kanten durch das Diagramm nennt man *Pfade*. Pfade beginnen immer im Startpunkt einer Aktivität und enden in Abbruch-Knoten, die alle aktiven Flüsse beenden, oder Endknoten. In Abbildung 2.4 ist ein Pfad durch das Diagramm `Laden vorbereiten` → `authentifizieren` → `prüfen` → `Laden initialisieren` → `Status anzeigen` &[12] `laden` → `authentifizieren` → `prüfen` → `Ladestecker entriegeln` → `Ladestecker nachbereiten`.

Alle Verzweigungen, Vereinigungen, Alternativen und Zusammenführungen müssen korrekt verschachtelt sein. Um dies zu gewährleisten, sollten jede Alternative und jede Zusammenführung eins zu eins abgestimmt werden. Darüber hinaus sollte jede Verzweigung mit einer Vereinigung zusammengeführt werden.

2.3.2 Domänenspezifische Sprachen mit dem MontiCore Framework

Die *MontiCore Language Workbench* dient der schnellen und effektiven Entwicklung von DSMLs mittels Grammatiken. Die Grammatik dient zur Definition der textuellen

[11] Die Möglichkeit einer Spezifikation für Vereinigungen um Bedingungen zu definieren [OMG17a, OMG17b], welche das Zusammenführen beschreiben, ist für den Fokus dieser Arbeit nicht relevant.

[12] & symbolisiert eine Vereinigung der Kontroll- und Objektflüsse.

Modellierungssprache. Auf der Basis dieser Grammatik generiert MontiCore ein DSL-Tool für das effiziente Engineering von Textsprachen und deren Infrastruktur wie Parser, Symboltabellen, Transformationen und Codegeneratoren. In diesem Abschnitt wird auf die für diese Arbeit relevanten Elemente der MontiCore Language Workbench eingegangen. Für eine umfassende Erläuterung der MontiCore Language Workbench ist auf [GKR+08, KRV08, KRV10, Kra10, HR17, Höl18, HKR21] zu verweisen.

MontiCore nutzt kontextfreie Grammatiken zur Definition von abstrakter und konkreter Syntax. Eine *MontiCore-Grammatik* besitzt einen Namen und eine durch die Grammatik spezifizierte Syntax der Sprache. Die Syntax wird mittels Erweiterte Backus Naur Form (EBNF)-Produktionsregeln [ISO96] definiert [HR17]. Die linke Seite der Produktionsregel, beziehungsweise die linke Seite des Gleichheitszeichens (`=`), besteht aus einem Nichtterminal. Auf der rechten Seite ist der Körper der Produktion, die das Nichtterminal ersetzt. Dieser Körper besteht aus Terminalen und Nichtterminalen. Terminale sind in Anführungszeichen eingeschlossen, definieren die konkrete Syntax und sind die atomaren Elemente der Grammatik. Ist die interne Repräsentation der Terminale von Bedeutung so können sie zusätzlich in eckige Klammern eingeschlossen werden.

In den Produktionsregeln können Alternativen durch `|` und Kardinalitäten durch `*` für beliebig viele Vorkommen, durch `+` für beliebig viele aber mindestens ein Vorkommen und durch `?` für maximal ein Vorkommen, definiert werden.

In Abbildung 2.5 ist ein für die Erläuterung vereinfachtes Beispiel der MontiCore-Grammatik für ADs abgebildet. Die Produktionsregel für das Nichtterminal `Activity` definiert den Namen `activity` (Terminal) und das beliebig viele Vorkommen von Knoten (`Node`), Kontrollknoten (`ControlNode`) und alternativ von Kanten (`Edge`).

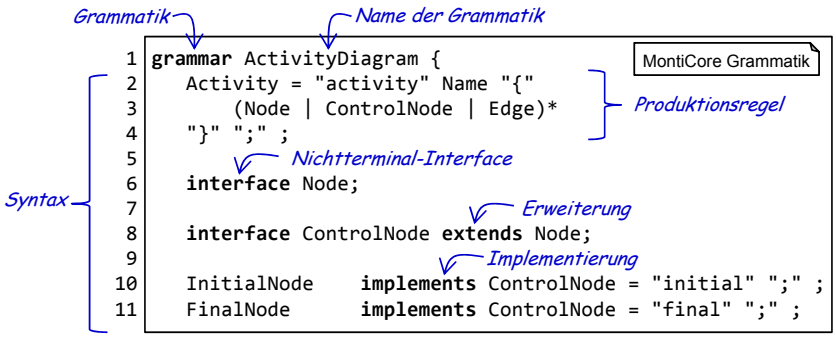


Abbildung 2.5: Vereinfachter Auszug aus der MontiCore-Grammatik für ADs

Für wichtige Nichtterminale wie Ausdrücke oder Aussagen lohnt es sich bestimmte Strukturen der Sprache zu erweitern. Diese Nichtterminale werden mittels `interface` eingeführt. In Abbildung 2.5 wird das Nichtterminal `Node` in Zeile 6 eingeführt. In Zeile 8 wird der bereits definierte `Node` um die zusätzliche Alternative, das Nichtterminal `ControlNode` mittels `extends` erweitert, ohne die ursprüngliche Definition von `Node` zu verändern. Der `ControlNode` wird mittels den (abstrakten) Nichtterminalen `InitialNode` (Startknoten) und `FinalNode` (Endknoten) mittels (`implements`) implementiert.

Für die Generierung der abstrakten Syntax für Parser, Lexer, Symboltabellen, Transformationen und Codegeneratoren nutzt MontiCore die Nichtterminalstruktur der Grammatik. Die generierte, interne Repräsentation des Modells der abstrakten Syntax wird abstrakter Syntaxbaum (engl. „Abstract Syntax Tree “) (AST) genannt. Abbildung 2.6 zeigt eine Übersicht über die templatebasierte Generierung eines DSL-Tools mit MontiCore.

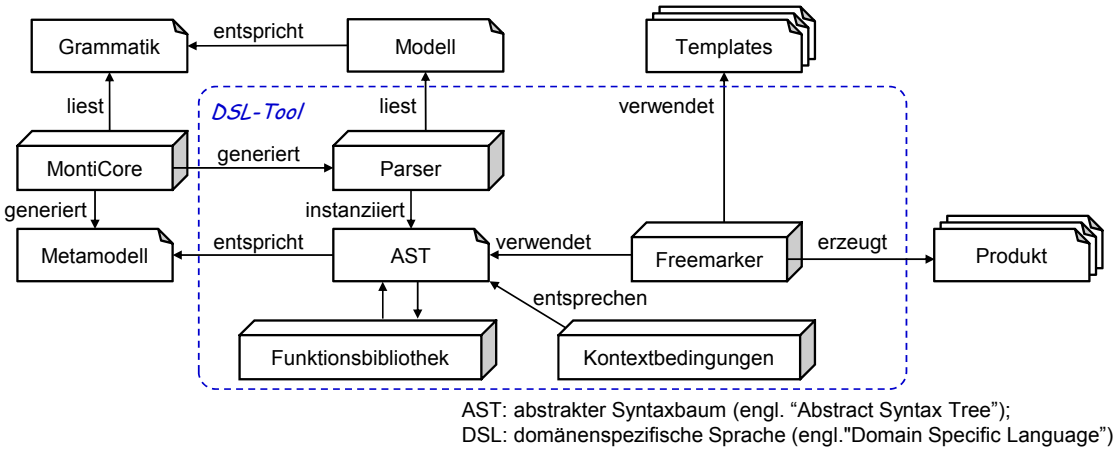


Abbildung 2.6: Übersicht über die templatebasierte Generierung eines DSL-Tools mit MontiCore angelehnt an [HR17, Höl18]

Der von MontiCore generierte Parser liest Modelle, die der definierten Grammatik entsprechen ein. Durch den Parser werden ein AST und eine Funktionsbibliothek erzeugt. Der AST kann, falls erforderlich, erweitert und modifiziert werden. Die Funktionsbibliothek ermöglicht den Zugriff auf die MontiCore-Bibliothek, die unter anderem den Aufbau von Symboltabellen, das Anordnen von Transformationen und das Festlegen von spezifischen Vorlagenkonfigurationen umfasst. Auch Bibliotheks-Funktionen können durch zusätzliche Funktionen erweitert werden.

Für die Generierung des gewünschten Produkts setzt MontiCore auf die Template-Engine Freemarker [For13, Fre19]. Diese erzeugt auf Basis von definierten Templates das Produkt. Das Template umfassen neben dem Code des gewünschten Produkts auch Anweisungen der Templatesprache wie zum Beispiel Kontrollstrukturen.

2.4 Grundlagen der Spezifikationsmethode für Anforderungen, Design und Test (SMArDT)

In diesem Abschnitt wird die Basis von SMArDT erläutert und die für diese Arbeit relevanten Begriffe definiert. Diese Basis wird in Kapitel 4 näher spezifiziert und erweitert.

Mit der Einführung der ISO 26262 (siehe Unterabschnitt 2.2.2) und komplexer werdenden Fahrzeugfunktionen wurde der Bedarf nach einer Spezifikationsmethode für (sicherheitsrelevante) Fahrzeugfunktionen größer [KMS+18]. Infolgedessen wurde SMArDT initi-

al von der BMW Group begründet und in Zusammenarbeit mit der exida GmbH [Exi20], der FEV Europe GmbH [Fev20] und dem Lehrstuhl für Software Engineering an der RWTH Aachen [SE-20] sukzessive weiterentwickelt [KMS+17, KMS+18, KKRvW18, HKK+18, DGH+18, KRGK18, Kri18, DGH+19]. SMArDT ordnet sich in das Systems Engineering ein. Der Fokus von SMArDT liegt auf dem technischen Prozess [ISO15a] beziehungsweise der technischen Lösung [CKS11]. Einen besonderen Wert legt SMArDT auf die Durchgängigkeit der Anforderungen über die Spezifikation der Funktionen bis hin zur Realisierung des Systems. SMArDT nutzt die Herangehensweise des V-Modells (siehe Unterabschnitt 2.2.1) insbesondere als Anordnungsrahmen für die Artefakte und deren Erstellung. Zum Beispiel werden Anforderungen an die Spezifikation von Funktionen über die jeweiligen Abstraktionsebenen verfeinert. Abbildung 2.7 zeigt die grundlegende funktionale Abstraktion in SMArDT.

In dieser Arbeit ist darauf zu achten, dass die verwendete Semantik der deutschen und englischen Begriffe „Ebene“ (engl. „level“) und „Schicht“ (engl. „layer“) nicht äquivalent ist. In der vorhandenen Literatur zu SMArDT [KMS+17, KMS+18, KKRvW18, HKK+18, DGH+18, KRGK18, DGH+19] werden diese zwei Begriffe allerdings als Synonym verwendet. In dieser Arbeit ist eine (Abstraktions-)Ebene wie folgt definiert.

Definition 2.1 (Ebene)**.** *Die Ebene (engl. „level“) ist in SMArDT ein Niveau der funktionalen Abstraktion. Es gibt vier Ebenen: Ebene A (Betrachtungseinheit), Ebene B (Funktionales Konzept), Ebene C (Technische Lösung) und Ebene D (Realisierung).*

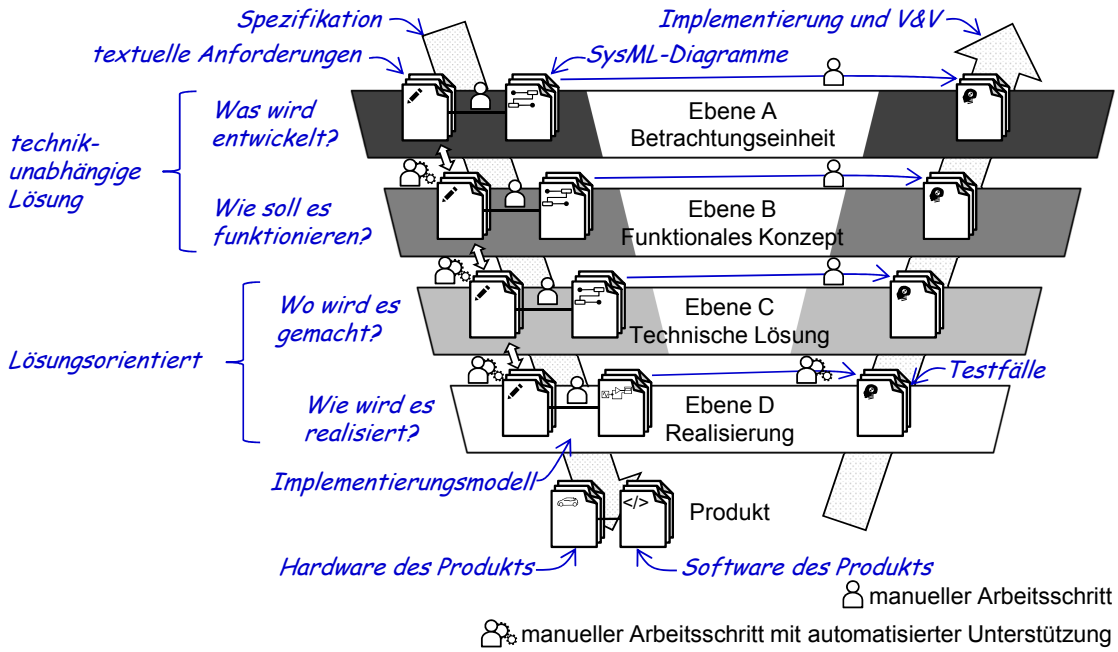


Abbildung 2.7: Übersicht über die funktionale Abstraktion in den Ebenen A bis D nach SMArDT (Erweiterung von [Kri18, DGH+19])

Die Abstraktion in SMArDT ist in vier Abstraktionsebenen strukturiert.

Ebene A, die Betrachtungseinheit, klärt die Frage: „Was wird entwickelt?“ [Kri18]. Ebene A enthält eine erste Beschreibung des betrachteten Objekts und grenzt dieses aus einer Kundensicht ab. Ebene *A* interpretiert und beschreibt die Anforderungen und Anwendungsfälle mittels UCDs für jede Kundenfunktionalität. [DGH+19]

Ebene B, das funktionale Konzept, klärt die Frage: „Wie soll es funktionieren?“ [Kri18]. Ebene *B* spezifiziert die Anforderungen aus Ebene *A* in ein Funktionskonzept. Diese Spezifikation abstrahiert von einer technisch/physikalischen Umsetzung und lässt insofern die Lösung offen. Zu diesem Zweck werden SysML-Verhaltensdiagramme verwendet. Für eine erste Gruppierung der Funktionen dient das SysML Strukturdiagramm, IBD. [DGH+19]

Ebene C, die technische Lösung, klärt die Frage: „Wo wird es gemacht?“ [Kri18]. Ebene *C* spezifiziert das funktionale Konzept von Ebene *B* weiter und setzt dieses in eine technische Lösung um. In dieser Ebene kommen ebenfalls die SysML Verhaltens- und Strukturdiagramme zum Einsatz. [DGH+19]

Ebene D, die Realisierung, klärt die Frage: „Wie wird es realisiert?“ [Kri18]. Ebene *D* setzt das Verhalten gemäß der spezifizierten Funktion in Software und/oder Hardware um. Aus den Artefakten dieser Ebene wird unter anderem der Quellcode des Produktes generiert und mechanische Bauteile mit Hilfe von computergestützter Konstruktion (engl. „Computer-Aided Design“) (CAD) entworfen und umgesetzt Diese Softwareartefakte können zum Beispiel UML, Matlab/Simulink [Mat19] oder Modelica [Mod19] Modelle sein, aus denen Code generiert wird. [DGH+19]

Ebene *A* und Ebene *B* verstehen sich als abstrakte Repräsentationen beziehungsweise als „technikunabhängige“ Lösung der Funktionalität. Ebene *C* und Ebene *D* modellieren hingegen eine direkte Darstellung (Lösungsorientiert) der Implementierung. Die „technikunabhängige“ Lösung bewegt sich in einem definierten Rahmen, lässt aber die technische Lösung eines Problems und die sich daraus ergebenen Rahmenbedingungen offen. Zum Beispiel werden in der Spezifikation eines Autos definierte Rahmenbedingungen wie vier Räder, die von einem oder mehreren Drehmomentgebern angetrieben werden, als gesetzt angenommen. Wie dieses (jeweilige) Drehmoment erzeugt wird und ob ein Drehmoment an einem Rad oder mehreren Rädern anliegt, bleibt offen. In den Ebenen *C* und *D* kann über quantifizierbare Kriterien wie dem gewünschten Fahrverhalten, dem Verbrauch, den Herstellungskosten oder der benötigten Entwicklungszeit entschieden werden, welche Lösung sich am besten eignet.

Im Vergleich zu vorangegangenen Veröffentlichungen aus weicht das in dieser Arbeit erläuterte Verfahren zum Teil ab. In [KKRvW18, HKK+18, DGH+19] kann Ebene *C* zur Generierung des Produktcodes mittels Komponenten- und Konnektor (engl. „Component and Connector“) (C&C) Modellen verwendet werden. In dieser Arbeit werden auf Ebene *C* abstrakten Elementen konkrete Elemente zugewiesen und über eine Realisierung in Hardware und/oder Software entschieden. Der Ansatz mit C&C Modellen geht demzufolge

einen sinnvollen Schritt für Softwarerealisierungen weiter. Des Weiteren ist in Bezug auf die vorangegangenen Veröffentlichungen [KMS+18, KKRvW18, HKK+18, DGH+18, KRGK18, Kri18, DGH+19] anzumerken, dass diese sich auf die softwaretechnischen Aspekte konzentrieren. In dieser Arbeit werden neben der softwaretechnischen Perspektive von SMArDT ebenfalls die mechatronischen Hardwareaspekte verstärkt berücksichtigt, da diese Aspekte gleichermaßen ein elementarer Bestandteil von SMArDT sind.

SMArDT nutzt nicht nur das Prinzip des V-Modells, sondern gleicht Defizite des oft manuellen Vorgehens aus [DGH+19]. Beispiele für Defizite sind unter anderem, dass funktionale Erweiterungen und Änderungen oft nur in der Hardware- und Software-Spezifikationsebene (vgl. Unterabschnitt 2.2.1) dokumentiert und darüberlegende obere Ebenen nicht entsprechend aktualisiert werden [DGH+19]. SMArDT nutzt eine Teilmenge der SysML um natürlichsprachliche textuelle Anforderungen zu entwickeln (vgl. Abbildung 2.7). Diese textuellen Anforderungen sind Teil der SysML-Modelle und werden für das Anforderungsmanagement beziehungsweise das Lastenheft exportiert. Demzufolge sind nur die textuellen Anforderungen verbindlich, es sei denn die SysML-Modelle werden ebenfalls in das Lastenheft integriert. Die Informationen der textuellen Anforderungen sind demnach redundant zu den Informationen der SysML-Modelle und erweitern die Informationen um nicht im SysML-Modell dargestellte Anforderungen wie zum Beispiel Anforderungen der Temperaturbeständigkeit. Redundanzen dieser Art können zu Inkonsistenzen führen und sollten vermieden werden. Ein Ansatz zur Vermeidung der Redundanzen wird in Kapitel 4 vorgestellt.

In den textuellen Anforderungen können Elemente des SysML-Modells verwendet werden, um eine Kohäsion sicherzustellen beziehungsweise Inkonsistenzen zu vermeiden. Beispielsweise können Systemelemente im SysML-Modell, wie ein Signal, als Element in eine textuelle Anforderung integriert werden. Ändert sich das Signal im SysML-Modell wird das Signal auch automatisch in der textuellen Anforderungen geändert und das Anforderungsmanagement informiert. Weiterhin tragen die SysML-Modelle zum Verständnis der textuellen Anforderungen bei und erleichtern eine lückenlose Anforderungsentwicklung mit Hilfe von manueller und automatisierter Verfahren. Beispielsweise können testfallerstellende Personen notwendige Informationen und Zusammenhänge der textuellen Anforderungen oder Modellelemente zur Verifizierung der Anforderungen aus dem Modell gewinnen.

Verglichen mit dem Entwicklungsprozess des V-Modells (vgl. Abbildung 2.2) ist erkennbar, dass Abstraktions- und Dekompositionsschritte kombiniert sind. In SMArDT sind diese Schritte bewusst separiert, stehen orthogonal zueinander und erlauben deshalb es die Komplexität des Systems in kleinere und weniger komplexe Elemente zu zerteilen [DGH+19]. Mit dieser orthogonalen Zerteilung wird versucht, die wachsende System- und Funktionskomplexität intellektuell beherrschbar zu machen [Pre03]. Diese Separation der Schritte ist allerdings nicht verbindlich. Die oft nahezu gleich bleibenden Funktionalitäten und die sich schnell ändernden technischen Systeme und hardwarespezifischen Strukturen werden somit entkoppelt. Infolgedessen kann eine Funktionalität überarbeitet und verbessert werden, ohne alle vergangenen und aktuellen technischen Aspekte, der

Dekomposition[13], berücksichtigen zu müssen. Die Dekomposition erlaubt es Aufgaben in kleinere, potenziell unabhängige, voneinander lösbare Aufgaben aufzuteilen [Fey10]. Es wird somit ermöglicht, über eine (Teil-) Parallelisierung von Aufgaben den Entwicklungsprozess zu beschleunigen [Fey10, SZ16]. Diese Strategie wird als „teile-und-herrsche“ (engl. „devide and conquer“) bezeichnet und hat sich über die letzten vier bis fünf Jahrzehnte in der Automobilindustrie und Software Engineering etabliert [BR07, SZ16]. Die Struktur der Dekomposition wie beispielsweise die Organisationsstruktur, Komponentenstruktur oder funktionalen Struktur ist nicht vorgegeben. Abbildung 2.8 stellt die sich ergänzende Abstraktion und Dekomposition vereinfacht dar. Im Prinzip umfasst eine Dekompositionsschicht mehrere Funktionalitäten, die mittels der funktionalen Abstraktionen in weitere Funktionen oder Funktionalitäten zerlegt und/oder auf Systemelemente abgebildet werden. Zum Beispiel werden in der funktionalen Abstraktion die Fahrzeugfunktionen

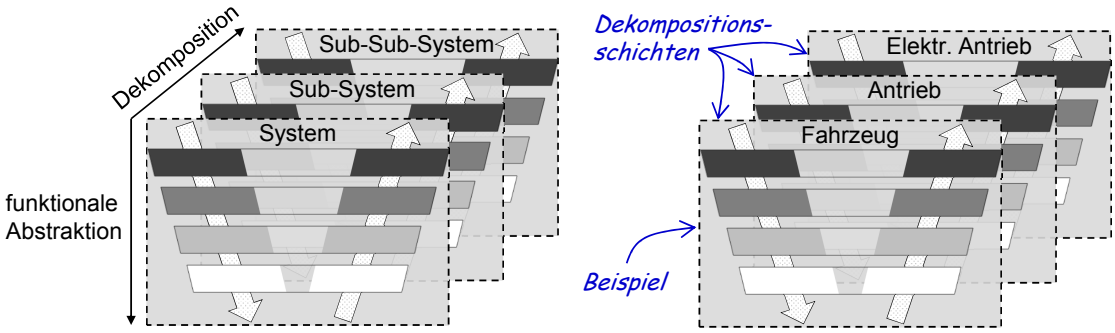


Abbildung 2.8: Vereinfachte Darstellung der orthogonal stehenden funktionalen Abstraktion und Dekomposition in SMArDT

der Dekompositionsschicht *Fahrzeug* in die Funktionalitäten der Antrieb, Fahrwerk, Karosserie, Multi-Media sowie Fahrerassistenz partitioniert. Diese werden in der nächsten Dekompositionsschicht Antrieb sukzessiv funktional zerlegt, bis sie auf konkrete Systemelemente abgebildet werden können. In dieser Arbeit ist eine (Dekompositions-)Schicht wie folgt definiert.

Definition 2.2 (Schicht)**.** *Die Schicht (engl. „layer“) ist in SMArDT eine Stufe der Dekomposition. Die Zerlegung und Aufteilung der Schichten ist in SMArDT nicht vorgegeben und kann an das jeweilige System und den Kontext angepasst werden.*

SMArDT nutzt nicht alle Potenziale einer modellbasierten Entwicklung. In SMArDT dient das Modell für die Entwicklung von textuellen Anforderungen, die mit den grafischen SysML-Modellen verknüpft werden. Diese textuellen Anforderungen sind rechtlich

[13] Nach [Fey10] gibt es nicht immer eine zwangsläufige disjunkte Aufteilung eines Problemraums in bestimmte Kategorien (hier Abstraktion und Dekomposition). Im Rahmen dieser Arbeit wird aufgrund der bereits etablierten Aufteilung in der Automobilindustrie der Begriff Dekomposition verwendet. In [KRGK18] wird anstatt einer Dekomposition von einer „lateralen Dekomposition“ gesprochen. In der lateralen Dekomposition wird jedoch auf die Synchronisation der (Sub-)Systeme verzichtet und nur eine gemeinsame Wertschöpfung sicherstellt [BNHZ18]. Dies trifft nicht auf die (Sub-)Systeme der hier behandelten Domäne zu.

verbindlich und werden anschließend in etablierten Prozessen und Tools weiter verwendet. Insofern bilden die textuellen Anforderungen die primären Artefakte (textuelle Dokumente) und nicht die Modelle, die für diese Zwecke nicht formal sein müssen. Die Potenziale des Modells beziehungsweise eines modellgetriebene Entwicklungsansatzes wie der Präzision und der Automatisierung von Aufgaben und Artefakterstellung bleiben somit ungenutzt. Aus diesem Grund wird in dieser Arbeit eine Weiterentwicklung von SMArDT vorgestellt, die eine modellgetriebene Entwicklung unterstützt.

2.5 Modellbasierte Entwicklungsmethoden in der Automobilindustrie

SMArDT reiht sich in die Kategorie modellbasierter (Software-) Methoden mit anderen Ansätzen ein. Dieser Absatz leitet die Diskussion über einige der Alternativen ein und erklärt, warum diese nicht übernommen und erweitert wurden. Ebenfalls bietet dieser Absatz die Möglichkeit, den Einfluss der existierenden Forschung auf dem Gebiet anzuerkennen. Trotz mehrerer Studien, die über die weite Verbreitung von modellbasierten Methoden in der Automobilindustrie und deren Vorteile berichten, deutet die aktuelle Literatur darauf hin, dass natürlichsprachliche Artefakte nach wie vor noch den Status quo in der Fahrzeugentwicklung bilden [LTK19, LTK$^+$18, VFW16, GRS16, BBH$^+$14, PSP09, PBKS07, WW03]. Ungeachtet dessen hat die Industrie erkannt, dass bei komplexen Systemen modellbasierte Ansätze Vorteile bieten [Bro06, PBKS07, WGM$^+$11, WHR14]. Theorie und Praxis 2.2 erläutert noch einmal die Eigenschaften und den Umgang von natürlichsprachlichen Anforderungen in der Industrie, um die Relevanz von formalen (modellbasierten) Ansätzen zu verdeutlichen.

Theorie und Praxis 2.2 (Natürlichsprachliche Anforderungen in der Industrie)**.** *Artefakte, wie Anforderungen in natürlichsprachlich textueller Sprache sind einfach zu schreiben, flexibel und universell einsetzbar [Bal09]. Allerdings ist ein großer Nachteil die lexikalische, syntaktische, semantische Mehrdeutigkeit und vage Begriffe [Bal09]. Diese Mehrdeutigkeiten werden oft zum Problem bei Domänen und Bereichen, die mit diesen natürlichsprachlichen Anforderungen arbeiten müssen, wie zum Beispiel Tester*innen. Eine Möglichkeit, Mehrdeutigkeiten von natürlichsprachlichen Anforderungen zu vermeiden, sind formale textuelle Anforderungen. In der Industrie wird erfahrungsgemäß versucht, die Nachteile natürlichsprachlichen Anforderungen mit Glossaren und der Benutzung von Anforderungsschablonen abzuwehren. Häufig sind das Ergebnis semiformale natürlichsprachliche Anforderungen, die nicht für effiziente Methoden des Software Engineering geeignet sind und Mehrdeutigkeiten enthalten.*

Es existieren mehrere Ansätze für modellbasierte Methoden zur Spezifikation von Anforderungen und Architekturen von verteilten Systemen. Bekannte Beispiele sind EAST-ADL [EAS13] und SPES 2020 [BBK$^+$21, PBDH16, PHAB12]. Die Gemeinsamkeit aller Ansätze ist der Versuch, mittels Abstraktion der Komplexität zu begegnen. Einige Ansätze konzentrieren sich auf spezielle Aspekte wie der Architektur [EAG$^+$05], fokussieren sich auf einen kleinen Teil des gesamten Systems [Piq14] oder nutzen eine geringe Anzahl

an Abstraktionsebenen [JR04, Mar02]. Eine weitere Gemeinsamkeit der angesprochenen Ansätze und [ZBF+05] ist, dass diese die Schritte der Abstraktion und hierarchische Dekomposition kombinieren und somit bei der Anwendung an großen komplexen Systemen schwer umzusetzen sind, obwohl sie in kleineren (Teil-) Domänen gelingen. Dies kann unter anderem ein Grund dafür sein, dass obwohl diese Ansätze wichtige Aspekte in der Entwicklung in der Automobilindustrie adressieren, die Akzeptanz der Industrie verhalten ist. Weitere Ansätze wie [EAS13, BBRS03, KMP+07, PHAB12, BBK+21] sind initial von der Industrie und Forschungsallianzen getrieben worden und umfassend dokumentiert.

Ein bekannter Ansatz ist OOSEM (engl. „Object-Oriented Systems Engineering Method“), der eine Top-down gerichtete und modellbasierte Entwicklung propagiert [Est20]. Der objektorientierte Ansatz beschreibt mit der SysML die verschiedenen Entwicklungsartefakte [Est20]. Diese Entwicklungsartefakte werden rekursiv auf Basis des V-Modells in den Systemhierarchien durchlaufen [Est20]. In den Systemhierarchien werden die Kundenbedürfnisse analysiert [Est20]. Darauf folgen die Analyse der Systemanforderungen, die Definition einer logischen Architektur und zuletzt die Zuweisung an die Hardwarearchitektur [Est20]. Diese vierstufigen Prozessschritte müssen anwendungsspezifisch individuell definiert werden [Est20]. Daher gibt es für den nicht domänenspezifischen Ansatz keine detaillierte Beschreibung der daraus resultierenden Artefakte. Ein weiterer Unterschied besteht darin, dass die vier Prozessschritte die Schritte der Abstraktion und Dekomposition kombinieren.

Ein weiterer modellbasierter Ansatz ist RUP SE (engl. „Rational Unified Process for Systems Engineering“), der in vier Abstraktionsebenen („Phasen“) aufgeteilt ist [Kru03]. In der Konzeptionsphase werden das System und Akteur*innen beschrieben [Kru03]. In der Entwurfsphase wird das System partitioniert [Kru03]. In der Konstruktionsphase werden die Systempartitionen realisiert und in der Umsetzungsphase implementiert und konfiguriert [Kru03]. Jede Phase außer Konzeption wird normalerweise in mehreren Iterationen durchgeführt [Kru03]. Auch dieser Ansatz kombiniert die Abstraktions- und Dekompositionsschritte, ist nicht domänenspezifisch und beschreibt die resultierenden Artefakte nicht detailliert [Kru03].

Im Gegensatz zu den nicht domänenspezifischen Ansätzen OOSEM und RUP SE bietet East-ADL [EAS13] (engl. „Electronics Architecture and Software Technology - Architecture Description Language“) eine DSL für die Spezifikation von Elektrik und Elektronik (E/E)-Fahrzeugarchitekturen in der Automobilindustrie [EAS13]. East-ADL basiert unter anderem auf den Konzepten der UML und SysML [EAS13]. Über die vier Abstraktionsebenen Fahrzeug-, Analyse-, Design und Implementierungsebene werden die Aufgaben und die Kommunikation des Systems beschrieben [EAS13]. Diese Abstraktionsebenen werden um Anforderungen, Variabilität, Timing und Abhängigkeiten erweitert [EAS13]. Diese Struktur der Abstraktionsebenen ist fest und nicht anwendungsspezifisch anpassbar. East-ADL fokussiert nicht das Verhalten, sondern die Struktur in separaten Software und E/E-Hardwaresichten und kombiniert die funktionale Abstraktion und hierarchische Dekomposition. Allerdings schließt der E/E-Fokus andere Disziplinen, wie den Maschinenbau, bei der Lösungssuche aus.

Die Automotive Modeling Language (AML) [FBBR03, BBRS03] ist eine weitere DSL

und Ansatz für die Analyse und Synthese von eingebetteten Systemen in der Automobilindustrie [RBBS02]. Ähnlich wie bei East-ADL spezifiziert AML den Software- und Hardwareanteil der Systemarchitektur mittels Konzepten der UML [FBBR03]. Der Ansatz von AML gliedert sich in fünf Abstraktionsebenen, Signale, Funktionen, logische Architektur, technische Architektur und Implementierung [RBBS02, BBRS03]. Diese fünf Abstraktionsebenen abstrahieren Signale, Funktionen, elektronische Steuergeräte, Echtzeitbetriebssysteme, Kommunikationsinfrastruktur und Prozessoren [RBBS02, BBRS03]. Die fünf klar beschriebenen Abstraktionsebenen helfen, die Abstraktionen intellektuell verständlich abzugrenzen. Die Abstraktionshöhe von AML mit dem Fokus auf E/E und Software konzentriert sich auf die Architektur und Funktionen von eingebetteten elektronischen Steuergeräten (vgl. Abstraktionsebenen C und D in Abschnitt 2.4). Die Schritte der Abstraktion und Dekomposition werden in AML ebenfalls kombiniert und Disziplinen wie der Maschinenbau stehen nicht im Fokus.

Ein weiterer Ansatz ist REMsES (Requirements Engineering und Management softwareintensiver Eingebetteter Systeme) [KMP+07, BBH+14] für die Spezifikation mittels Konzepten der UML. Ein System wird in drei Sichten der Systemebene, der Funktionsgruppenebene und der Hardware- und Software-Ebene zerlegt und trennen die Lösungs- und Problemebene [BBH+14]. Orthogonal zu dieser Zerlegung wird eine weitere Trennung in die Artefaktklassen *Kontext*, *Anforderungen* und *Entwurf* vollzogen [KMP+07, BBH+14]. Diese Artefaktklassen lassen sich mit den SMArDT-Abstraktionsebenen A, B und C vergleichen. In der Artefaktklasse *Kontext* werden die Anforderungen gesammelt und in den Kontext des Systems gesetzt, in der Artefaktklasse *Anforderungen* die Funktionsanforderungen spezifiziert und in der Artefaktklasse *Entwurf* Anforderungen an die Komponenten festgelegt [BBH+14]. Die Systemebene, Funktionsgruppenebene und Hardware- und Software-Ebene von REMsES ähneln hingegen den Dekompositionsschichten von SMArDT. In der festen orthogonalen Struktur wird das System als Ganzes betrachtet und die Dekomposition erfolgt mit Funktionsgruppen. Diese werden in einer späteren Phase einzelner Hardware- und Softwarekomponenten zugewiesen. Die Anzahl der Abstraktions- und der Dekompositionssichten sind fest vorgegeben. Folglich lässt sich Dekomposition nicht individuell auf Systeme anpassen. Die ganzheitliche Sicht des Systems stellt stets die zahlreichen festgelegten Abhängigkeiten der Artefakte dar und kann bei großen komplexen Systemen zu intellektuellen Verständnisproblemen [BBK+17]. Ebenfalls ist zu erwähnen, dass REMsES die Inhalte der Software fokussiert und andere Disziplinen nicht im Fokus stehen.

Eine weitere Methodik für die Entwicklung eingebetteter Systeme ist der Ansatz SPES (engl. „Software Platform Embedded Systems“) [PHAB12, PBDH16, BBK+21]. Ähnlich dem Ansatz von REMsES sind die Dekopmositionsschichten von SPES orthogonal zu den Sichten *Anforderungen*, *Funktionen*, *Logik* und *Technik* angeordnet. Diese vier Sichten gleichen den vier Abstraktionsebenen von SMArDT. Die drei Dekopmositionsschichten von SPES ähneln den Dekompositionsschichten von SMArDT. Die Anzahl der Dekopmositionsschichten ist in SPES 2020 [PHAB12] allerdings auf drei festgelegt. Die Sichten von SPES werden intellektuell verständlich getrennt, wobei SMArDT die technikunabhängige Lösung zur Wiederverwendung und deren Artefakte in den Abstraktionsebenen A und B

stärker von dem Lösungsraum in den Ebenen C und D abgrenzt (vgl. Abbildung 2.7). Der Vorteil der Trennung von SPES ist, dass der Abstraktionsgrad mit weniger Aufwand korrigiert werden kann, da keine technikunabhängige Lösung vorausgesetzt wird. Die Abstraktions- und Dekompositionsschritte sind in SPES nicht getrennt. Wie bei auch REMsES, fokussiert SPES die Inhalte der Software und die Abhängigkeiten der Artefakte können bei großen komplexen Systemen zu intellektuellen Verständnisproblemen führen [BBK+17].

Zusammenfassend fokussieren die domänenspezifischen Ansätze überwiegend die Disziplinen E/E und Informatik. SMArDT verspricht, die Domänenexperten aller Disziplinen in der Entwicklung mit einzubinden. Zu diesem Zweck werden die technikunabhängig Lösung und die lösungsorientierten Abstraktionsschritte getrennt, um keine spezifische Lösung einer Disziplin vorwegzunehmen. Bisher lässt SMArDT eine Trennung von Abstraktions- und Dekompositionsschritten zu, aber schreibt diese Trennung nicht vor. Die in dieser Arbeit vorgestellte Weiterentwicklung von SMArDT in Kapitel 4 schreibt die Schritte der Abstraktion und Dekomposition vor, welche in keinem der vorgestellten Ansätze klar unterschieden werden. Zusätzlich werden die Abstraktionsschritte, Dekompositionsschritte und klar definierte Elemente beschrieben.

Kapitel 3

Absicherung in der Automobilindustrie

Im vorangegangenen Kapitel wurde die modellgetriebene Entwicklung in der Automobilindustrie, deren Methoden und relevante Modellierungssprachen beschrieben. In diesem Kapitel werden die Grundlagen für modellbasiertes Testen (MBT) zum Verständnis dieser Arbeit erläutert. Zu diesem Zweck wird einerseits ein Überblick zum Testen im Allgemeinen gegeben. Darüber hinaus werden die Eigenschaften der Domäne und der Testfallerstellung erläutert. Zudem werden die in diesem Zusammenhang auftretenden Begrifflichkeiten definiert.

Ein *Fehler* (engl. „failure“) ist nach [ISO17b] die Nichterfüllung einer Spezifikation oder Anforderung. Diese Nichterfüllung kommt zustande, wenn das Ist-Verhalten vom Soll-Verhalten abweicht [SL12]. Das Ist-Verhalten tritt zur Laufzeit eines Systems auf. Das Soll-Verhalten ist in der Spezifikation oder den Anforderungen definiert. In der Literatur und der Norm wird der Begriff „Fehler“ weiter differenziert [ISO17b, SL12]. Der *innere Fehler* (engl. „fault“ oder „bug“) ist nach [ISO17b] ein inkorrektes Produkt. Es handelt sich demnach um einen Fehler, der durch einen *Mangel* (engl. „defect“) eines (Software-) Produkts hervorgerufen wird. Der innere Fehler ist prinzipiell für den Benutzer erst einmal nicht sichtbar und kann zu einem äußeren Fehler führen [SL12]. Der *äußere Fehler* (engl. „failure“) ist die äußerlich sichtbare Abweichung von der Spezifikation oder den Anforderungen des Systems [ISO17b]. Der äußere Fehler kann auftreten, wenn ein innerer Fehler aufgetreten ist. Oft wird unter dem allgemeinen Begriff Fehler der äußere Fehler verstanden. Wird ein Fehler durch menschliches Handeln hervorgerufen, spricht man von einer *Fehlhandlung* (engl. „error“) [ISO17b, SL12]. Im Vergleich zu einem Mangel oder einem äußeren Fehler ist eine Fehlhandlung zum Beispiel mit einem Auto bei angezogener Handbremse versuchen loszufahren.

Menschen machen Fehler. Besonders wer neue Wege beschreitet, neue Technologien einsetzt und/oder unter Zeitdruck steht, wird früher oder später Fehler machen [SM91]. Im Allgemeinen ist anerkannt, dass es derzeit sehr unwahrscheinlich ist ein fehlerfreies (Software-) System zu entwickeln, sobald dieses System einen gewissen Grad an Komplexität und Umfang erreicht [ISO13a, SL12]. In komplexen und umfangreichen Systemen, wie einem Fahrzeug, können Mängel zu Fehlern und unter Umständen zu Unfällen mit Verletzen oder Toten führen. Zum Beispiel, wenn die Bremsen versagen.

Deshalb ist es wichtig, Fehler schon während der Entwicklung aufzuspüren und deren Ursache, die inneren Fehler beziehungsweise Mängel, zu beseitigen. Ein Verfahren zur Lokalisierung von Fehlern ist das Testen. Das *Testen* beziehungsweise ein *Test* ist eine Reihe von Aktivitäten, die unter bestimmten Bedingungen auf einem System oder einer

Komponente ausgeführt werden [ISO13a, ISO17b]. Bestimmte Aspekte und Ergebnisse werden bei Tests beobachtet oder aufgezeichnet und daraufhin eine Bewertung dieser Aspekte des Systems oder der Komponente vorgenommen [ISO13a, ISO17b]. Ziel des Testens ist es Fehler aufzudecken und somit die Anwesenheit von Fehlern zu demonstrieren [SL12], jedoch nicht deren Abwesenheit [Dij72]. Die Tests haben das übergeordnete Ziel eine Qualität des betrachteten Testobjektes zu bestimmen, um das Vertrauen in eben dieses zu erhöhen [SL12, ISO18b].

In der Softwareentwicklung werden zwei Testmethoden unterschieden, das statische Testen und das dynamische Testen. Ein *statischer Test* dient der Ermittlung von Fehlern und Verstößen gegen Standards oder Spezifikationen, ohne dass ein System oder eine Komponente ausgeführt wird [ISO13a, SL12, Lig09]. Ein Beispiel für statische Tests sind zum Beispiel Reviews, Inspektionen oder statische Analysen. Allerdings können bei der reinen Analyse nicht alle Informationen, die während der Ausführung eines Systems oder einer Komponente erzeugt werden, betrachtet werden [Lig09]. Deshalb kann die statische Analyse keine vollständige Aussage über die Qualität des Testobjektes treffen [Lig09]. Eine weitere Testmethode ist der in der Softwareentwicklung weit verbreitete dynamische Test. Der *dynamische Test* führt Systeme und Komponenten mit konkreten Eingabewerten aus [SL12, ISO13a, Lig09]. Im Gegensatz zum statischen Test umfassen dynamische Tests reale Betriebsumgebungen und schließen während der Ausführung erzeugte Informationen mit ein. Diese dynamischen Tests sind allerdings stichprobenartig, da in der Praxis nicht alle möglichen Situationen getestet werden können [Dij72, Lig09]. Stichprobenartig, da nicht alle möglichen Situationen getestet werden können, sind alle praktisch relevanten dynamischen Testtechniken Stichprobenverfahren. Folglich beweisen dynamische Tests keine vollständige Korrektheit des Systems oder der Komponente [Dij72, Lig09, SL12]. In dieser Arbeit liegt der Fokus auf dynamischen Tests.

Um Unsicherheiten der Qualitätsaussage [ISO11] auf Basis von dynamischen Tests so gering wie möglich zu halten, wird in dieser Arbeit ein systematischer Ansatz empfohlen. Für diesen systematischen Ansatz dienen Testfälle mit einem konkreten Testziel und einem bestimmten Testumfang auf einem bestimmten Testobjekt. Ziel ist es Fehler zu identifizieren und Mängel zu beheben.

Definition 3.1 (Absicherung)**.** *Die Absicherung (engl. „safeguarding“) umfasst die Fehleridentifikation und Mängelbeseitigung.*

Nachdem in diesem Abschnitt die allgemeinen Begrifflichkeiten des Testens kurz erläutert wurden, werden in den folgenden Abschnitten einzelne Themen weiter vertieft. Im Folgenden werden in Abschnitt 3.1 die Eigenschaften des Testobjektes und der Automobilindustrie in Bezug auf das Testen erläutert. Anschließend wird der Testfallerstellungsprozess in Abschnitt 3.2 vorgestellt. Abschließend wird in Abschnitt 3.3 das modellbasierte Testen in der Automobilindustrie erläutert.

3.1 Testobjekt und Testumgebung

Das in dieser Arbeit behandelte Testobjekt ist das (Sub-) System „elektrischer Antrieb“, wie in einem konventionellen Elektroauto (engl. „Battery Electric Vehicle“) (BEV) oder einem Plug-In-Hybrid Electric Vehicle (PHEV). Beispiele hierfür sind der BMW i3 (BEV) und der BMW 330e (PHEV). Abbildung 3.1 illustriert eine vereinfachte Übersicht über die Komponenten (blau markiert oder blau schraffiert) des Subsystems elektrischer Antrieb.

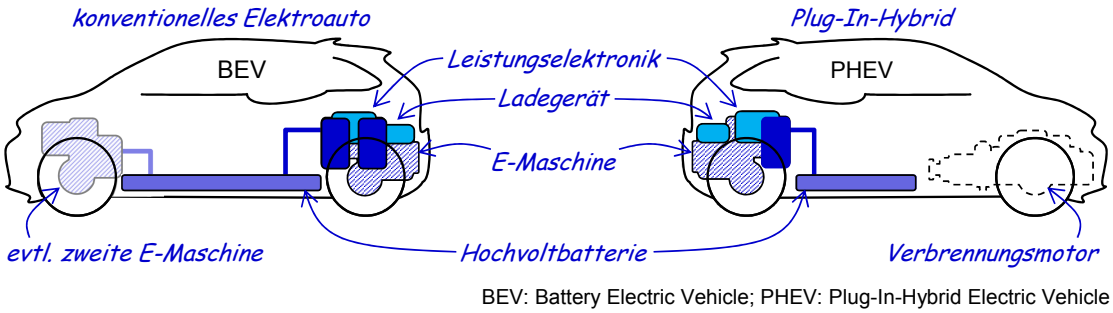


Abbildung 3.1: Vereinfachte Übersicht über das Subsystem E-Antrieb (farblich markiert oder schraffiert) abgeleitet aus [MSWD15]

Im Prinzip nimmt das Subsystem elektrischer Antrieb, elektrische Energie auf, speichert diese und gibt die Energie in Form eines Drehmoments weiter. Die Energieumwandlung wird mittels verschiedener Komponenten gewährleistet. Diese Komponenten sind neben der elektrischen Maschine, der sogenannten E-Maschine, auch das Ladegerät, die Hochvoltbatterie und die Leistungselektronik zur Steuerung der Energieverteilung. Auch diese Komponenten können wiederum in kleinere Einzelteile gegliedert werden. Je nach Sicht beziehungsweise Detaillierungsgrads werden die Testfälle ausgelegt [ISO18a]. Abbildung 3.2 gibt einen Überblick über funktionsorientierte Fahrzeugtests mit eingebetteten Systemen. Das Beispiel veranschaulicht verschiedene Testobjekte - Software und Hardware mit teils komponentenübergreifende Funktionen - und die daran angepasste Testmethodik. *Funktionsorientierte Tests* (engl. „functional testing“) beziehungsweise funktionsorientierte Testfälle (engl. „functional test cases“) prüfen, ob die Anforderungen (Spezifikation) an das zu prüfende Objekt erfüllt sind [ISO15b, UL07, Lig09, Wit16]. Da die funktionsorientierten Testfälle der Soll-Funktionalität aus der Spezifikation und nicht aus dem Programmcode abgeleitet werden, handelt es sich um *Black-Box-Tests* [Lig09, ISO17b, UL07]. Abgeleitete Tests aus dem Programmcode werden als *White-Box-Tests* bezeichnet [ISO15b, UL07, Lig09, Wit16]. Werden Testfälle ohne einen vorliegenden implementierenden Programmcode aber mit ausreichenden Informationen über das Verhalten einer Funktion erstellt, wird von einem *Grey-Box-Test* gesprochen [Wit16, ISO15b]. Grey-Box-Tests basieren auf der Verhaltens- und Strukturspezifikation und bilden eine Mischform von Black-Box- und White-Box-Tests. Abbildung 3.2 zeigt ein eingebettetes System. Auf dem System laufen drei Funktionen, die zum Teil über mehrere Softwarebausteine und Hardwarekomponenten verteilt sind. Ein Softwarebaustein ist ein abgegrenztes programmiertechnisches Artefakt [BF14]. Softwarebausteine entsprechen

Softwaremodulen aus [Koo19] und werden in dieser Arbeit auch als allgemeiner Begriff für Datenbank oder Softwarekomponente verwendet.

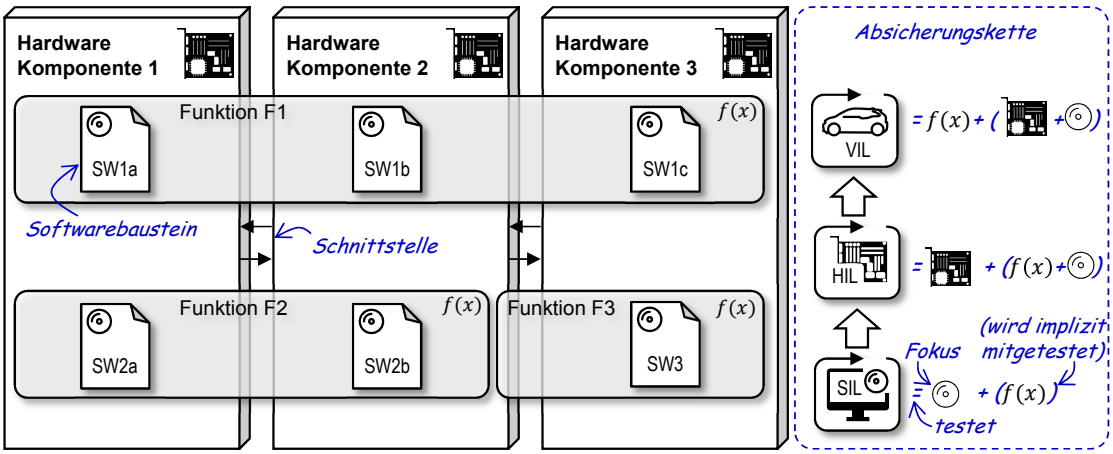


Abbildung 3.2: Überblick über ein eingebettetes System mit den verteilten Softwaremodule und die automatisierte Testumgebung der Absicherungskette am Beispiel eines Fahrzeugs

Bei *Komponententests* kann zwischen reinen Hardware-/Softwarekomponenten und den Hardware-Software-Verbund-Komponenten unterschieden werden. Das Testziel, ob die in der Spezifikation festgelegte Funktionalität korrekt und vollständig ausgeführt realisiert wurde, ist bei reinen Hardware- und Softwaremodulen und eingebetteten Systemen gleich [SL12]. Allerdings werden Softwaremodule wie zum Beispiel `SW1a`, `SW2a` und `SW3` in Abbildung 3.2 verstärkt in einer Software in the Loop (SIL) Testumgebung auf der Basis von Programmcode getestet. Eine *Testumgebung* ist eine Umgebung, die das eigentliche System möglichst realitätsnah abbildet. Somit können Tests auf einer verhaltensähnlichen Konfiguration ausgeführt werden. Die Testumgebung kann Hardware, Firmware, Instrumentierung, Simulatoren, Softwarewerkzeuge und andere unterstützende Hilfsmittel umfassen, die für die Durchführung von Tests vorgesehen sind oder dazu verwendet werden[IST20, ISO13a]. Im Gegensatz zu einer SIL Testumgebung für das Testen von Programmcodes wird für Komponententests mehr Hardware benötigt. Aus diesem Grund werden für Komponententests verstärkt Hardware in the Loop (HIL) ausgeführt. HIL-Umgebung sind Prüfstände, die eine reale Umgebung eines Systems nachbilden. Zu diesem Zweck werden teilweise reale Hardware und Hardwareprototypen und Software genutzt, während der Rest der Umgebung simuliert wird. Diese HIL-Umgebung wächst/entwickelt sich mit dem Projektfortschritt und die simulierten Testumgebungen können nach und nach durch reale Module und Komponenten ergänzt werden. Für Komponententests werden neben Informationen aus dem Programmcode, insbesondere Testfälle aus der Spezifikation beziehungsweise den Anforderung gewonnen.

Sind die Komponenten größtenteils realisiert können die Integrations- beziehungsweise

(Sub-) Systemtests beginnen. Das Ziel der *Integrationstests* ist die komplette Funktionalität im Zusammenspiel aller Einzelteile zu testen und zu prüfen, ob die Anforderungen an das (Sub-) System erfüllt werden (*anforderungsbasiertes Testen*) [SL12, ISO13a]. Im Beispiel aus Abbildung 3.2 könnte Funktion `F3` allein von Komponententests abgesichert werden. Allerdings müssen die komponentenübergreifenden Funktionen `F1` und `F2` aufgrund ihres verteilten Charakters einem Integrationstest unterzogen werden. Die Testfälle der Integrationstest werden vor allem an Produkt nahen HIL-Umgebungen und dem Fahrzeug selbst (Vehicle in the Loop (VIL)) durchgeführt [DGH+19]. Bei VIL-Umgebungen handelt es sich, um voll ausgestattete Fahrzeuge beziehungsweise Fahrzeugprototypen.

Nach den Integrationstests wird mittels *Systemtests* das integrierte System geprüft. Der Fokus dieser Prüfung liegt darin, das Produkt und die Anforderungen aus der Sicht der Kund*innen zu erfüllen [SL12]. Im Beispiel aus Abbildung 3.2 prüft ein Systemtest alle verbauten Hardware-Komponenten und der kompletten Umgebung mit entsprechenden Prototypen.

Die *Prüfkosten*[1] für den Betrieb der verschiedenen Testumgebungen unterscheiden sich erheblich. Während die Kosten eines Testfalls in der SIL-Testumgebung relativ gering sind, fallen in der HIL- und VIL-Umgebung wesentlich höhere Kosten an. Auch die Prüfzeit pro Testfall kann in der SIL-Testumgebung verringert werden, da man nicht an Echtzeit gebunden ist. HIL-Testumgebungen können auf eine Komponente ausgerichtet sein (*Komponenten-HIL*) oder die Integration wie einem *Verbund-HIL* mit mehreren Komponenten fokussieren. Die Erstanschaffungskosten (vgl. Prüfkosten) einer HIL-Umgebung können bis zu den Kosten eines Fahrzeugprototypen ansteigen. Allerdings sind in der HIL-Umgebung das Testen beziehungsweise die Absicherung mehrerer Fahrzeugtypen, -varianten und deren Komponenten möglich. Im Gegensatz dazu kann mit einem Prototypen (VIL) oft nur ein Fahrzeugtyp getestet werden und der Umbau, um weitere Komponenten in diesem Fahrzeug testen zu können, ist kostspielig und zeitaufwendig. Allerdings ist bei VIL-Tests keine Simulation der Umgebung nötig und bestimmte elektromagnetische Effekte und Temperaturauswirkungen können allein in VIL-Tests evaluiert werden, da diese oft erst in real verbauten Systemen auftreten. Abbildung 3.3 stellt die Testumgebungen schematisch dar und setzt sie in einen Zusammenhang mit allgemeinen *Fehlerkosten*[2] und Projektphasen.

Am Allgemeinen verursachen später auftretende Fehler oder neue Anforderungen im Projekt wesentlich höhere Kosten als frühzeitig erkannte Fehler [Wit16, Boe79, SL12]. Je nach Projektphase können sich die Kosten verdoppeln bis verzehnfachen [Wit16, SL12]. Ein Meilenstein dieser Kostenentwicklung in der Automobilindustrie ist der Start of Production (SOP) (vgl. Abbildung 3.3). Dieser wird am Anfang der Projektplanung festgelegt und ausschließlich im Notfall „geopfert“ [Sch08]. Ab dem SOP nehmen die Kosten für Änderungen rasant zu [Sch08]. Deshalb muss die Entwicklung und Fehlerbehebung abgeschlossen sein. Dennoch kommt es zum Beispiel aufgrund von Lieferverzögerungen oder Än-

[1] Prüfkosten umfassen die Anschaffungskosten, die Betriebskosten, die Personalkosten und deren Organisation [DIN08]

[2] Fehlerkosten sind die Kosten, die durch einen Fehler verursacht wurden [DIN08]. Neben der Linderung oder Beseitigung des Fehlers müssen ebenfalls Kosten der Verschrottung, Ausfallzeiten, Gewährleistung, Produkthaftung und der Imageverlust in Betracht gezogen werden [DIN08].

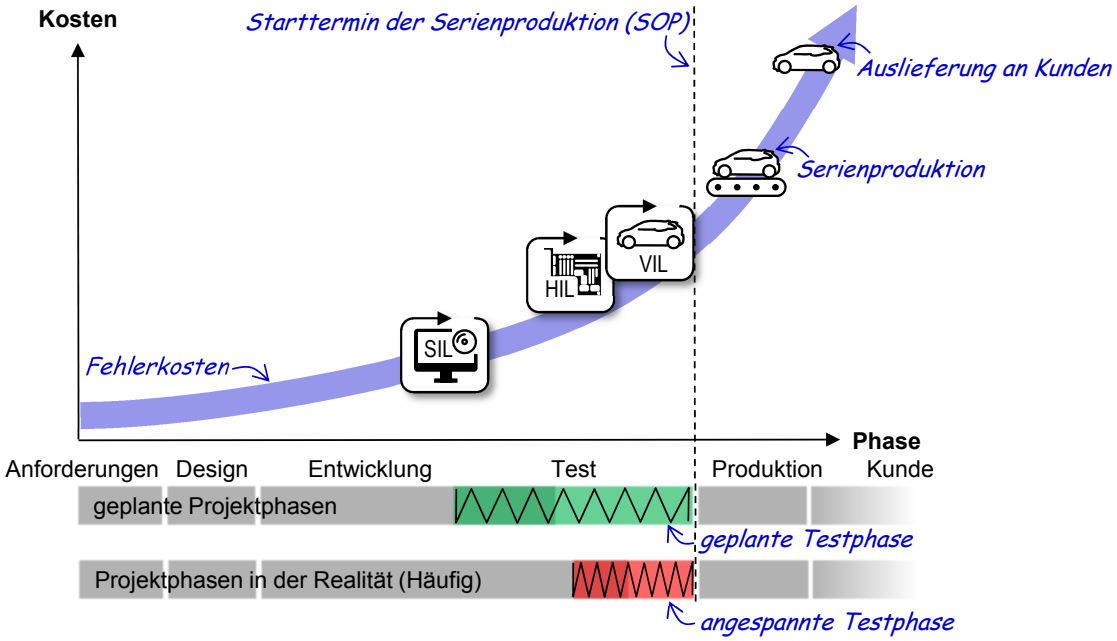


Abbildung 3.3: Vereinfachte Darstellung der Fehlerkosten [Wit16, Boe79] in Abhängigkeit von ihren Projektphasen[4] [HW17, Boe79] und initialen Starts der Prüfverfahren SIL, HIL und VIL

derungen in letzter Sekunde häufig zu einer verlängerten Entwicklungszeit [Sch08, SZ16]. Die Zeit für die Entwicklung von Testfällen und das Testen für Änderungen verkürzt sich infolgedessen erheblich (vgl. Abbildung 3.3).

Als letzte Glieder in der Absicherungskette vor SOP sind die Systemtests betroffen. Neben dem Termindruck des SOPs, müssen Freigaben für Fahrzeug-Prototypen, Zulassungen von neuen Modellen, Zertifizierungen und Serienproduktion erteilt werden. Alle Fehler, die in dieser Phase nicht mehr gefunden werden, können für das jeweilige Unternehmen teuer zu stehen kommen. Diese Arbeit konzentriert sich auf die anforderungsbasierte Testfallerstellung für HIL und VIL, um die Tester*innen bei der Testfallerstellung zu unterstützen.

3.2 Testfallerstellung

Ein *Testfall* ist eine Reihe von Testeingaben und Ausführungsbedingungen, die erwartete (Test-) Ergebnisse hervorrufen sollen [ISO17b, ISO13a]. Die in dieser Arbeit im Fokus ste-

[4]Die Projektphasen sind durch einen Phasen- und Meilensteinplan klar (vertraglich) definiert [HW17]. Allerdings können die Termine aufgrund der Projektsituation abweichen, wie zum Beispiel weitere Anforderungen die während der Entwicklung anfallen. Insbesondere die Testaktivitäten sind heutzutage breite und kontinuierliche Aktivitäten während des gesamten Entwicklungsprozesses [Ber07]. Die maximale Konzentration der Testaktivitäten ist erfahrungsgemäß in der Phase vor dem SOP.

henden Testfälle bestehen aus einer *Vorbedingung* (engl. „precondition“), Eingabewerten, deren eventuell auszuführenden Aktionen, den erwarteten Ergebnissen [ISO13a, ISO18b] und einer Nachbedingung (engl. „postcondition“). Auszuführende Aktionen des Testfalls werden mit dem Begriff *Steuern* und vergleichende, prüfende Aktionen werden mit *Status* gekennzeichnet. Soll ein *Status* sichergestellt hergestellt werden, unabhängig davon, ob die Aktion *Steuern* funktioniert, wird eine *Herstellen*[3]-Aktion verwendet. Abbildung 3.4 stellt einen Testfall zur Prüfung der Glühlampe im Schaltkreis auf Basis der Modelle aus Abbildung 2.1 dar.

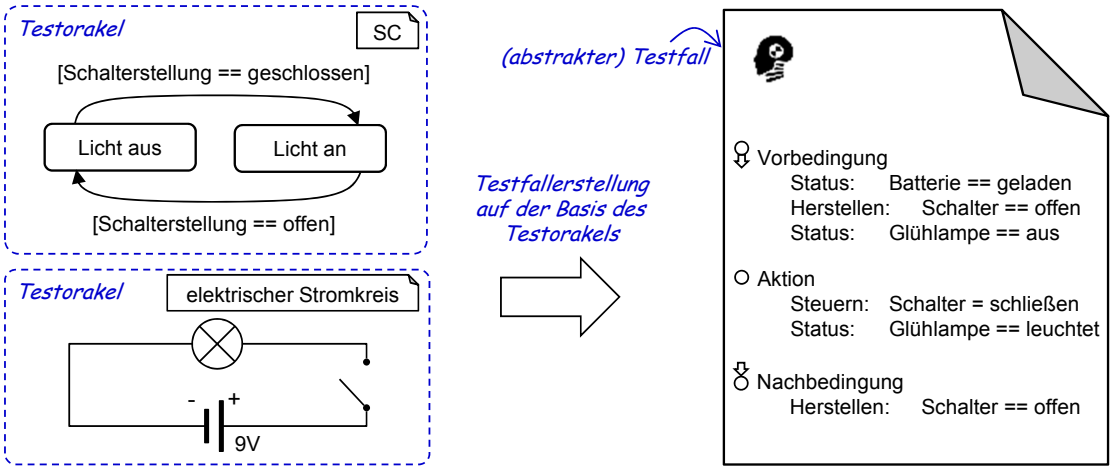


Abbildung 3.4: Modelle eines Schaltkreises und ein Testfall zur Überprüfung der Funktionsfähigkeit einer Glühlampe

Die Modelle dienen dem Testfall als Testorakel. Das *Testorakel* ist die Informationsquelle zur Ermittlung der erwarteten Ergebnisse eines Testfalls. Es stellt unter anderem textuelle Anforderungen oder Modellspezifikationen wie in Abbildung 3.4 dar. Die Vorbedingung (♀) für den Testfall „Lampenschaltung prüfen“ ist, dass die Stromquelle genug Energie liefert, um die Glühlampe zum Leuchten zu bringen (`Status: Batterie == geladen`) und der Schalter offen ist (`Status: Schalter == offen`). Sind die Vorbedingungen nicht erfüllt, wird der Testfall aufgrund einer Fehlhandlung nicht ausgeführt. Es ist möglich die Vorbedingungen eines Testfalls nicht nur passiv zu überprüfen, sondern auch aktiv herzustellen, zum Beispiel `Herstellen: Schalter == offen` um die Aktion des Testfalls ausführen zu können. In diesem Fall wird der Begriff *Herstellen* verwendet. `Herstellen` kann einen `Status` mit einem mehrmaligen Aufruf von `Steuern`, mehreren Eingaben oder das Signal direkt über eine Manipulation des Signals manipulieren. Die auf die Vorbedingung folgende Aktion (○) des Testfalls in Abbildung 3.4 ist das Schließen des Schalters (`Steuern: Schalter = schließen`) und das erwartete Ergebnis, die leuchtende Glühlampe (`Status: Glühlampe == leuchtet`). Die Aktion ist das Kernelement des Testfalls und prüft die zu untersuchende Funktionalität. Wird die Aktion nicht

[3]Dieses Verfahren wird unter anderem in von der TraceTronic GmbH [Tra20] angewendet.

komplett ausgeführt, liegt ein äußerer Fehler vor. Andernfalls läuft der Testfall weiter. Abschließend wird ein gewünschter Zustand des Testobjekts in einer Nachbedingung (8) wieder hergestellt, um das Testobjekt für eventuelle weitere Testfälle vorzubereiten. Wird der Schritt nicht erfolgreich ausgeführt, wird dies wieder als eine Fehlhandlung interpretiert.

Die einzelnen Bausteine eines Testfalls werden *Testschritte* genannt. Diese Schritte werden nacheinander ausgeführt. Kann ein Schritt wie zum Beispiel `Steuern: Schalter = schließen` nicht korrekt ausgeführt oder entspricht das eingetroffene Ergebnis nicht dem erwarteten Wert, wird entsprechend der Testfallgliederung eine Fehlhandlung oder ein äußeren Fehler gemeldet. Ein Testfall deckt äußere Fehler auf, behebt diese allerdings nicht. Zum Beispiel könnte ein Testfall den äußeren Fehler, dass die Glühlampe nicht wie erwartet leuchtet aufdecken. Aufgrund des gefundenen äußeren Fehlers kann die innere Fehlersuche und deren Behebung beginnen. Dieser verursachende innere Fehler kann mit weiteren Tests oder Analysen der Testergebnisse identifiziert werden.

Im Folgenden werden in Unterabschnitt 3.2.1 das Konzept des Testens mit Schlüsselwörtern erläutert und anschließend der anforderungsbasierte Testfallerstellungsprozess in der Automobilindustrie in Unterabschnitt 3.2.2 vorgestellt.

3.2.1 Schlüsselwortgetriebenes Testen

Für ein gemeinsames und vereinfachtes Verständnis von Testfällen mit ihren Testschritten können Schlüsselwörter verwendet werden. Ein *Schlüsselwort* besteht aus einem oder mehreren Wörtern, die auf eine bestimmte Aktion oder mehrere definierte Aktionen verweisen [ISO16]. Diese Aktion/en der Schlüsselwörter werden von einem oder mehreren Testfällen durchgeführt [ISO16]. Da es sich um Aktionen handelt, werden Schlüsselwörter mit mindestens einem Verb benannt [ISO16].

Abbildung 3.5 illustriert einen Ausschnitt des Testfalls aus Abbildung 3.4 in einer Variante mit und einer ohne schlüsselwortgetriebenes Testen.

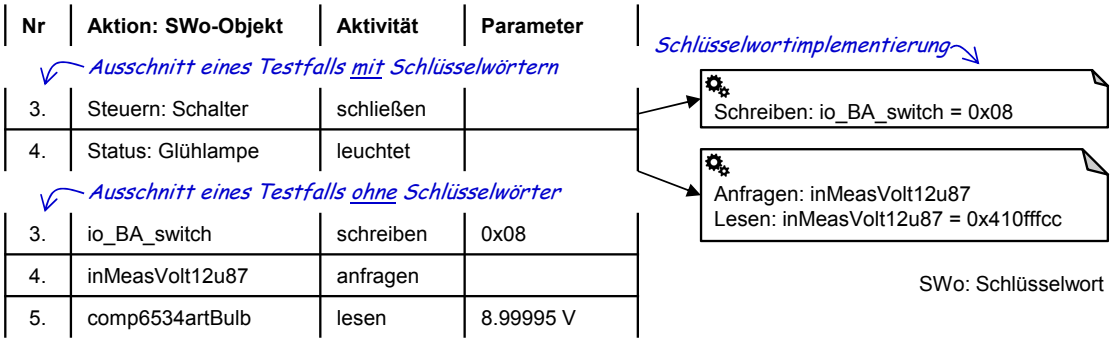


Abbildung 3.5: Ausschnitt des Testfalls aus Abbildung 3.4, oben mit den Schlüsselwörtern „Steuern: Schalter schließen“ und „Status: Glühlampe leuchtet“ sowie unten ohne Schlüsselwörter

Aus Abbildung 3.5 ist zu erkennen, dass der Ablauf eines schlüsselwortgetriebenen

Testfalls intellektuell verständlicher ist als der Testfall mit exakten Signalnamen[4]. Neben einem verbesserten Verständnis mittels Abstraktion, unterstützt schlüsselwortgetriebenes Testen die Wiederverwendbarkeit von Testfällen und vereinfacht die Testfallautomatisierung [ISO16]. Beides wird über einen Implementierungsschritt zwischen dem erstellten Testfall mit Schlüsselwörtern und dem Ausführen des Testfalls in der Testfallumgebung erreicht. In diesem Schritt werden die abstrakten Schlüsselwörter in der entsprechenden Testumgebung automatisiert und die abstrakten Aktionen den konkreten Signalen des Systems beziehungsweise der Testumgebung zugeordnet. Am Beispiel von Abbildung 3.5 wird das Schlüsselwort `Steuern: Schalter = schließen` in der Testumgebung durch die Schlüsselwortimplementierung `Schreiben: io_BA_switch = 0x08` ersetzt. Das Schlüsselwort `Status: Gluehlampe = leuchtet` wird außerdem durch `Anfragen: inMeasVolt12u87` und `Lesen: inMeasVolt12u87 = 0x410fffcc` abgebildet, da diese Nachricht nur in dieser Form in der Testumgebung gelesen werden kann. Schlüsselwörter können auch auf andere Schlüsselwörter und/oder Implementierungen verweisen. Zum Beispiel könnte `Status: Gluehlampe = leuchtet` wieder auf eine Implementierung verweisen, die wiederum auf Implementierungen und Schlüsselwörter referenzieren. Dieses Prinzip erlaubt den Aufbau von Hierarchien von Schlüsselwörtern mit verschiedenen Abstraktionsstufen.

Abbildung 3.6 beschreibt das Schlüsselwort mit Relationen zu anderen Elementen wie der Implementierung mithilfe eines Referenzmodells.

Definition 3.2 (Referenzmodell)**.** *Ein Referenzmodell (engl. „reference model“) ist ein spezifisches Modell innerhalb einer Anwendungsdomäne [ANT14]. Es dient als Dokumentation für bestehende spezifische Anwendungen oder als Ausgangspunkt für die Entwicklung eines Schemas für bestimmte Anwendungen. In dieser Arbeit werden Referenzmodelle mithilfe der Modellierungstechnik des Klassendiagramms (engl. „Class Diagram“) (CD) [OMG17b] modelliert.*

Das Referenzmodell wird in dieser Arbeit genutzt, um die Struktur und den Zusammenhang von Elementen zu beschreiben und verständlicher zu machen. Auffallend ist das Element beziehungsweise die CD-Klasse `Schlüsselwortaktion` in kursiver Schreibweise. Kursiv geschriebene Elemente im Referenzmodell sind abstrakt und bedeuten, dass es keine Instanzen der Elemente geben kann. Diese abstrakten Elemente werden zur Strukturierung genutzt, zum Beispiel um gemeinsame Eigenschaften in eine Oberklasse zu gruppieren. Die `Schlüsselwortaktion` ist die abstrakte Oberklasse der drei konkreten Ausprägungen `Steuern`, `Herstellen` und `Status`, die nicht abstrakt sind. In der Modellierung beinhaltete die abstrakte Oberklasse („Oberelement“) alle Informationen, die ihre konkreten Unterklassen gemeinsam haben. Jede konkrete Unterklasse beinhaltet dann spezifische Informationen, die nur sie hat. Außerdem kann eine abstrakte Oberklasse mit konkreten Unterklassen modelliert werden, dass es genau nur die Instanzen der Unterklassen geben kann und keine anderen. In Abbildung 3.6, kann eine `Schlüsselwortaktion` genau nur eine Instanz der drei Unterklassen sein.

[4] Signalnamen sind in ihrer Umgebung eindeutig und für nicht involvierte Personen oft schwer verständlich.

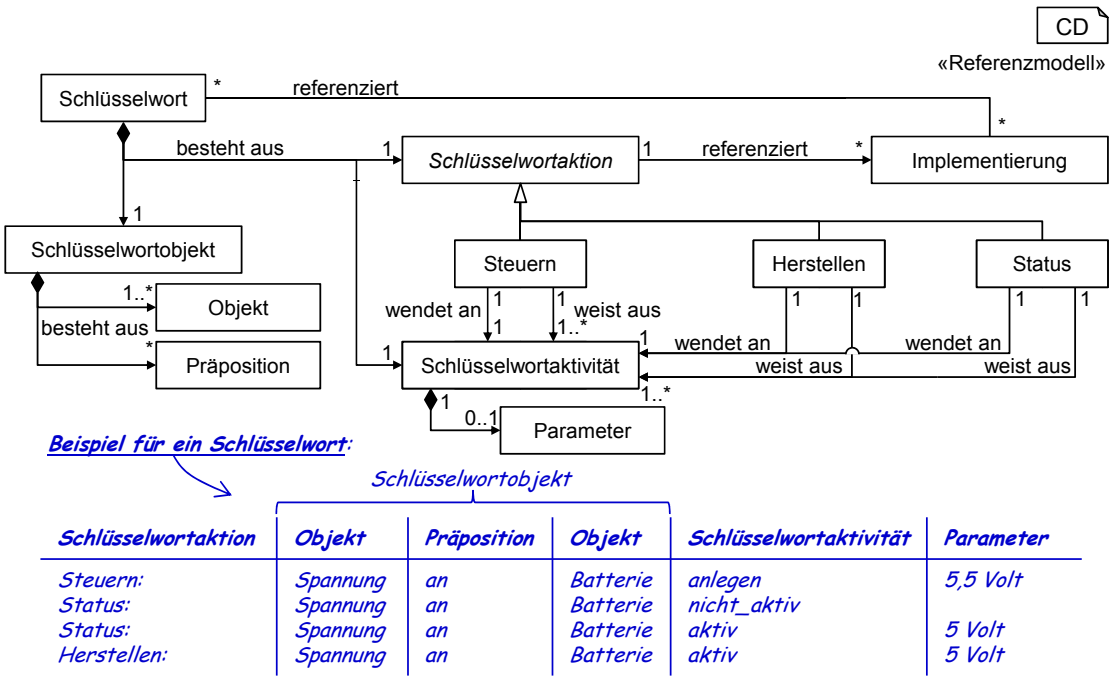


Abbildung 3.6: Vereinfachtes Referenzmodell des Schlüsselwortes mit Beispiel

Ein `Schlüsselwort` besteht immer genau aus einem `Schlüsselwortobjekt`, einer `Schlüsselwortaktion` und einer `Schlüsselwortaktivität`. Das Schlüsselwortobjekt setzt sich aus mindestens einem im Fokus stehenden `Objekt` und beliebig vielen `Präpositionen` für eine detailliertere Beschreibung zusammen. Die Schlüsselwortaktion eines Schlüsselwortes kann austauschbar entweder `Steuern`, `Status` oder `Herstellen` sein. Alle drei Schlüsselwortaktionen weisen mindestens eine Schlüsselwortaktivität für den Gebrauch in einer Liste aus. In einem Testschritt mit einem Schlüsselwort, kann immer genau eine dieser Schlüsselwortaktivitäten der Liste von der jeweiligen Schlüsselwortaktionen angewendet werden. Bestimmten Schlüsselwortaktivitäten können auch `Parameter` zugeordnet werden, wie *Steuern: Spannung an Batterie anlegen* 5,5 *Volt* in Abbildung 3.6.

Neben der Anwendung von Schlüsselwortaktivitäten referenzieren das Schlüsselwort je nach Schlüsselwortaktion eine spezifische `Implementierung`. Die Implementierung kann neben der konkreten Umsetzung in der Testumgebung auch wieder auf Schlüsselwörter und andere Implementierungen referenzieren. Zu beachten ist, dass es durchaus sinnvoll sein kann, die Implementierung von `Herstellen` auf die Implementierungen von `Steuern` und `Status` zu referenzieren, aber nicht umgekehrt. Zum Beispiel kann `Herstellen` so implementiert werden, dass in einer Schleife `Steuern` ausgeführt wird und anschließend mit `Status` geprüft wird, ob der erwartete Zustand des Testobjektes eingetroffen ist. Die Referenz des Schlüsselwortes auf die Implementierung kann je nach Testumgebung über eine Testumgebungskonfiguration angepasst werden.

Mit den umfangreichen und individuellen Implementierungsmöglichkeiten kann folglich

das schlüsselwortgetriebene Testen auf allen Prüfstufen (zum Beispiel SIL, HIL, VIL, Komponententests, Integrationstests, Systemtests) angewendet werden [ISO16]. Abbildung 3.7 gibt eine Übersicht über die Zuordnung von Schlüsselwörtern zu deren Implementierung in den jeweiligen Testumgebungen.

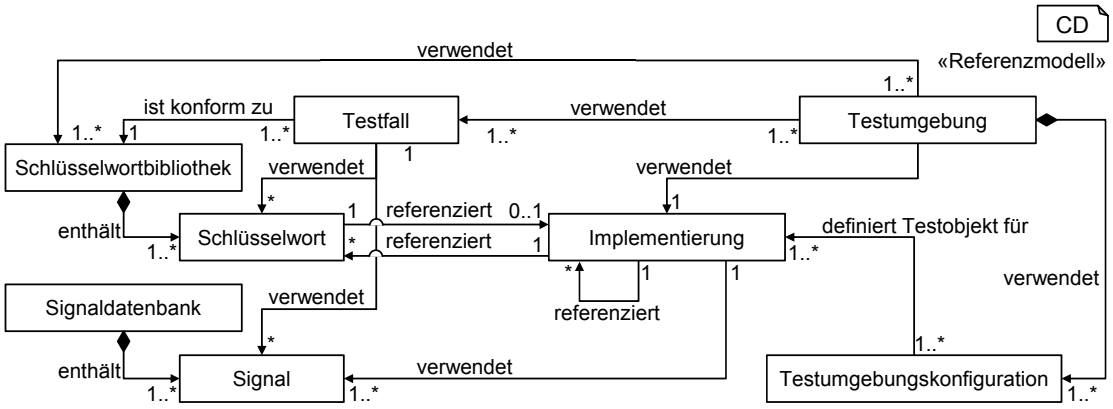


Abbildung 3.7: Übersicht über eine mögliche Zuordnung von Schlüsselwörtern und deren Implementierung in den Testumgebungen

Die `Testumgebung` kann mehrere `Schlüsselwortbibliotheken`, `Testfälle`, `Implementierungen` und `Testumgebungskonfigurationen` verwenden. Während eines Tests ist allerdings nur eine passende Gruppe (`Schlüsselwortbibliothek`, `Testfall`, `Implementierung`, `Testumgebungskonfiguration`) aktiv. Die gewählte Testumgebungskonfiguration definiert das Testobjekt für die Implementierung. In einer allumfassenden Testumgebung, die funktional gleiche SIL, HIL, VIL, Komponententests, Integrationstests und Systemtests unterstützt, kann über die Testumgebungskonfiguration gewählt werden, mit welcher Implementierung und demnach welche Prüfstufe getestet wird. Auch gleichwertige Testumgebungen können die gleichen Testfälle verwenden. Auf diese Weise können Ausfälle oder Unsicherheiten von Testumgebungen kompensiert, Testfälle wiederverwendet und Testfälle verglichen werden[5].

Ein schlüsselwortbasierter Testfall verwendet immer `Schlüsselwörter` einer spezifischen Schlüsselwortbibliothek. Der schlüsselwortbasierte Testfall ist folglich immer konform zu einer Schlüsselwortbibliothek. Diese bilaterale Konformität kann nur dann erweitert werden, wenn eine andere Schlüsselwortbibliothek exakt die gleichen Schlüsselwörter beinhaltet, die der Testfall verwendet. Ein Testfall ist weiterhin konform zu eben dieser Schlüsselwortbibliothek, wenn direkt `Signale` aus der `Signaldatenbank` verwendet werden, da die Schlüsselwörter über die `Implementierung` auf eben diese `Signale` der `Signaldatenbank` referenzieren. Diese direkte Verwendung von Signalen, kann sinnvoll sein, wenn die gleichen/selben Signale aus verschiedenen Testumgebunden verwendet

[5]Der Vergleich von Testfällen ist vor allem für HIL-Testumgebungen relevant. Falls ein Testfall einen Fehler wirft, kann vereinzelt nicht erkannt werden, ob der innere Fehler am System, an der Testfallumgebung oder am Testfall liegt. Mit einem zweiten Lauf in einer anderen Testumgebung kann eine falsche Interpretation der Testergebnisse weiter eingegrenzt beziehungsweise ausgeschlossen werden.

werden. Dadurch wird die Schlüsselwortbibliothek nicht mit unnötigen Schlüsselwörtern überladen. Grundsätzlich referenziert ein im Testfall verwendetes Schlüsselwort auf eine Implementierung. Welche Implementierung referenziert wird, ist in der Testumgebungskonfigurationen definiert. Die Implementierung führt anschließend die definierten automatisierten Operationen, unter der direkten Verwendung von Signalen oder mit der Referenz auf andere Schlüsselwörter, aus. Referenziert eine Implementierung andere Schlüsselwörter, muss darauf geachtet werden, dass keine Schleifen entstehen.

Die *Schlüsselwortbibliothek* ist ein Repository, das eine Reihe von Schlüsselwörtern enthält [ISO16]. Die Schlüsselwörter einer Schlüsselwortbibliothek spiegeln alle die Sprache und den Grad der Abstraktion wider, der beim Schreiben von entsprechenden Testfällen verwendet wird [ISO16]. Alle Testfälle auf Basis von Schlüsselwörtern aus einer Schlüsselwortbibliothek besitzen folglich den gleichen Abstraktionsgrad. Zusätzlich benötigt die testfallerstellende Person keine detaillierten Kenntnisse von der Programmierung und den Testumgebungen [ISO16]. Demzufolge kann der modulare und strukturierte Testfall mit einheitlichen natürlichen Sprache von vielen Personen verstanden werden [ISO16]. Bei Änderungen der Signale oder der Testumgebungen werden nur die jeweilige Implementierung und nicht alle betroffenen Testfälle angepasst. Diese Eigenschaften schirmen die Testfälle vor kurzfristigen Änderungen ab und erleichtern die Wartung, da nur bei Änderungen der Systemfunktionalität die Testfälle angepasst werden müssen. Abbildung 3.7 stellt dar, wie die Schlüsselwörter über die Testumgebungskonfiguration mit den Signalen der Testumgebung verknüpft werden. Die Testumgebungskonfiguration enthält neben einer Definition des Testobjekts und deren Schnittstellen ebenfalls die Zuweisung für die Referenz der Schlüsselwörter auf deren Implementierung.

3.2.2 Anforderungsbasierter Testfallerstellungsprozess

Im Bereich des bereits vorgestellten Subsystems zum elektrischen Antrieb in Abschnitt 3.1 werden Testfälle zur Prüfung der Funktionalität erstellt. Hierbei liegt der Schwerpunkt auf Intergrations- beziehungsweise Systemtests auf Basis der Anforderungen und Spezifikationen. Der in dieser Arbeit vorgestellte Testfallerstellungsprozess basiert auf der ISO 29119 Software Testing [ISO13a, ISO13b, ISO13c, ISO15b, ISO16] und detailliert diese für den Bereich des elektrischen Antriebs.

Am Anfang der Testfallerstellung wird meist auf der Grundlage der Anforderungen ein funktionales Testkonzept beziehungsweise ein Testplan erstellt [DGH+19, ISO13b, ISO13c]. In diesem *Testkonzept* wird detailliert beschreiben, welche Anforderungen getestet werden, wie diese Anforderungen getestet werden, welche Testumgebungen notwendig sind und wann, welche Anforderungen getestet werden sollen[6] [ISO11, ISO13b]. Anschließend können die Testfälle erstellt werden. Abbildung 3.8 stellt die einzelnen Aktionen der Testfallerstellung dar.

[6] „Wann welche Anforderungen getestet werden sollen“ ist vor allem von Relevanz in der evolutionären Prototypentwicklung mit Entwicklungssteuergeräten [SZ16]. In dieser Art der Entwicklung werden mehrere Prototypen von Steuergeräten und Komponenten in der Entwicklung angefertigt. Diese Hardware- und Software-Prototypen erfüllen entsprechend des Entwicklungsstands (Integrationsstufe) bestimmte Anforderungen und verfügen infolgedessen nur über eine eingeschränkte Funktionalität.

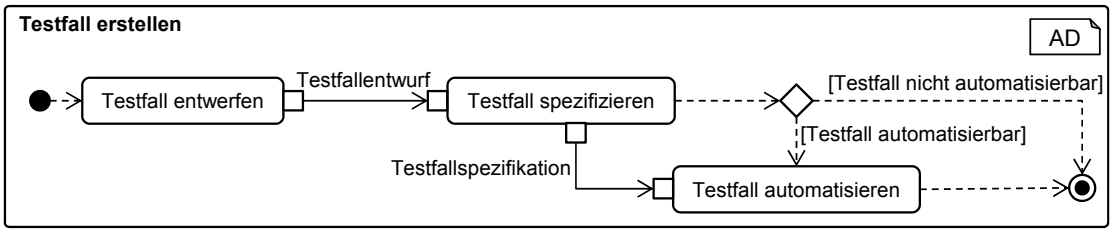


Abbildung 3.8: Vereinfachte dargestellte Aktivität des Testfallerstellungsprozesses

In einem ersten Schritt wird ein Testfall entworfen. Hierbei wird der Testfall angelegt und beschrieben. Außerdem wird bestimmt, welche Anforderung/en mit diesem Testfall getestet werden. Die Testbarkeitsbewertung von Anforderungen, wird mit der Person abgestimmt, die diese Anforderungen stellt. Die *Testbarkeit* gibt die Effizienz beziehungsweise den Aufwand an, mit dem die Funktionalität und andere Eigenschaften des (Sub-) Systems[7] getestet werden können [ISO17b, ISO11, SL12]. Ist eine Anforderung testbar, kann ein Testfall entworfen werden. Dieser *Testfallentwurf* umfasst eine grobe Vorbedingung, Anforderungen an die gewünschte Testumgebung und eine Verknüpfung für die Rückverfolgbarkeit zu der erfüllenden Anforderung der Spezifikation.

Als Nächstes folgt die Spezifikation des Testfalls. Für die *Testfallspezifikation* werden die genauen Testschritte spezifiziert. Diese Testschritte können Schlüsselwörter oder direkte Signalmanipulationen sein. Ist der Testfall spezifiziert, wird dem Testfall eine Priorität zugeteilt, weitere Eigenschaften in Attributen definiert und eine konkrete Zielumgebung für den Testfall benannt. Zusätzlich wird bestimmt, wann dieser Testfall ablaufen soll.

Auf der Basis der entwickelten Testfallspezifikation wird anschließend entschieden, ob dieser Testfall automatisierbar ist (vgl. Unterabschnitt 3.2.1). Diese „Automatisierbarkeit“ wird mit der Person abgestimmt, die für die Testumgebung verantwortlich ist. Ist eine Automatisierung möglich und gewünscht, werden in der *Testfallautomatisierung* die Eingabe von Daten und die Abarbeitung der Testschritte automatisiert. Im Fall von schlüsselwortgetriebenen Testfällen umfasst die Testfallautomatisierung, die Automatisierung der (noch nicht automatisierten) Schlüsselwörter eines Testfalls. Kann ein spezifizierter Testfall nicht automatisiert werden[8], wird dieser nicht als automatisierter Testfall gekennzeichnet und bleibt folglich nur manuell ausführbar. Ein manueller Testfall dient als Skript für einen Test, der von einer Tester*in ausgeführt wird. Für diesen Zweck enthält der manuelle Testfall Testdaten über durchzuführende Aktionen und zu erwartende Ergebnisse. Der manuelle Test kann durch eine Toolführung unterstützt werden aber im Gegensatz zum automatisierten Testfall nicht ohne Tester*innen automatisch laufen.

Ändert sich die Funktionalität eines Systems, wird dies in den Anforderungen vermerkt.

[7] Teilweise hängt die Testbarkeit von der betrachteten Sicht auf ein System (Teststufe) ab [Wit16]. Bestimmte Anforderungen an Funktionen, wie ein Selbstschutz der Komponente, sind testbar in Bezug auf Komponenten- oder Integrationstest, jedoch nicht unbedingt auf der Systemteststufe.

[8] Gründe für einen manuellen Testfall sind im Normalfall eine Anforderung an eine Testumgebung, die zu diesem Zeitpunkt oder aufgrund des Automatisierungsaufwands von der Testumgebung nicht erfüllt werden kann oder der Aufwand in keinem Verhältnis zum Nutzen steht.

Über die Rückverfolgbarkeit der Tests zu den Anforderungen werden die Tester*innen informiert und können kontrollieren, ob die Anforderungen von den zugeordneten Testfällen noch geprüft werden. Ist dies nicht der Fall, müssen die Testfälle nach und nach korrigiert werden wie in Abbildung 3.8 dargestellt.

3.3 Ansätze für modellbasiertes Testen (MBT) in der Automobilindustrie

Sind die zu testenden Anforderungen eines Systems als (abstraktes) Modell modelliert, wird von MBT gesprochen [BF14, BWC12, Lig09, UL07]. Bei diesen Modellen kann es sich um Modelle von Domänen, Umgebungsmodelle, Verhaltensmodelle oder abstrakte Modelle eines Testfalls handeln [UL07]. Im Allgemeinen werden Verhaltensmodelle für das modellbasierte Testen herangezogen [Lig09, UL07]. Verwendet man in den Modellen eine schlüsselwortbasierte Notation, können automatisiert ausführbare Testfälle entstehen (vgl. Unterabschnitt 3.2.1). Nach [Lig09] gibt es die Möglichkeit Testfälle direkt aus dem Systemmodell abzuleiten oder ein unabhängiges Testmodell parallel für die Erstellung von Testmodellen zu nutzen. Abbildung 3.9 stellt die beiden Ansätze dar.

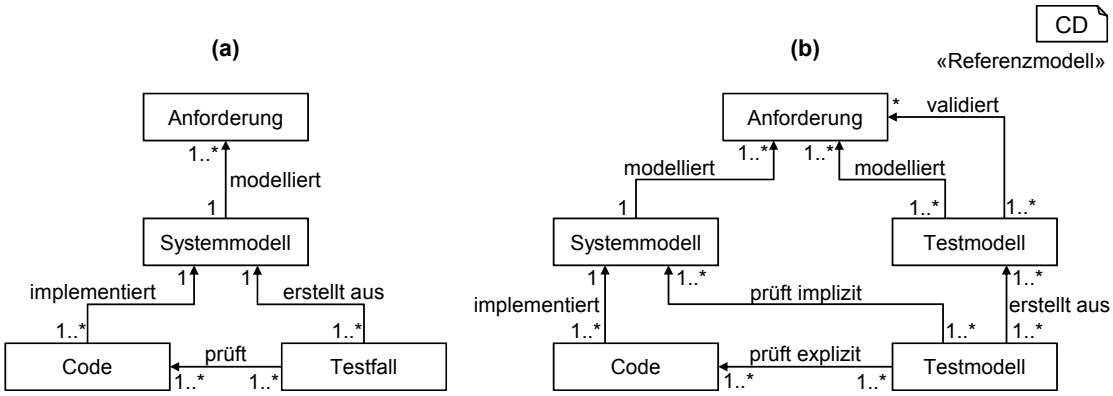


Abbildung 3.9: MBT mit (a) einem Systemmodell und (b) einem unabhängigen Testmodell nach [Lig09]

Im Rahmen dieser Arbeit beschreibt das *Systemmodell* ausgewählte Aspekte der Struktur, des Verhaltens, des Zustands und der Interaktion von Objekten und Konzepten des Systems [BCGR09b, BCGR09c]. Die Spezifikation des Systems ist über das Systemmodell definiert. Das *Testmodell* umfasst testrelevante Aspekte und stellt einen Testauftrag dar, der für die Testfallerstellung verwendet wird [ISO15b]. Für spezifische Aspekte kann es sinnvoll sein, mehrere Testmodell zu erstellen wie Testmodelle für funktionale und/oder sicherheitskritische Aspekte.

Nach [Lig09] basieren beide Ansätze auf Anforderungen, aus denen ein Systemmodell und gegebenenfalls ein oder mehrere Testmodelle erstellt werden. Bei einer Ableitung von Testfällen aus einem Systemmodell wird der manuell oder automatisiert abgeleitete

Code getestet [Lig09]. Werden unter einem Mehraufwand Testmodelle je nach Testaspekt modelliert, kann mittels der Testfälle das Systemmodell implizit und/oder der Code explizit gegen die Anforderungen geprüft werden. Es kann erwartet werden, dass je abstrakter die Anforderungen beziehungsweise die Modelle sind, desto abstrakter sind auch die abgeleiteten Testfälle. Im abstrakten Fall müssen Testfälle gegebenenfalls noch verfeinert und präzisiert werden, um diese direkt in der Testumgebung ausgeführt werden zu können [Lig09]. Im Folgenden wird der Testfokus dieser Arbeit in der bestehenden Literatur eingeordnet.

Modellbasiertes Testen in der Literatur

Die Idee von MBT ist nicht neu[9] und dennoch ist die modellbasierte automatisierte Testfallgenerierung in der Automobilindustrie gering vertreten [DGH+19, AWS14, KPM13, SVWH11, GJM+10, Bro06]. Ein Problem der Verhaltensmodelle für die Testfallerstellung ist, dass sie abstrakt und konkret zugleich sein sollen [PSAK04]. Die Modelle sollten abstrakt sein, da sie verständlich und handhabbar sein müssen. Andererseits sollten sie konkret sein, um eine Generierung von Testfällen zu ermöglichen [PSAK04].

Ein großer Teil der Ansätze für automatisiertes MBT generiert Black-Box-Testfälle auf der Basis von konkreten Testmodellen. In der Regel modellieren die Parteien, die nicht direkt für die Entwicklung zuständig sind, ein Testmodell, das auf den natürlichsprachlichen Anforderungen basiert und mit den Entwickler*innen abgesprochen ist. Abbildung 3.10 ordnet bestehende MBT-Ansätze aus der Automobilindustrie (und zum Teil anderen verwandten Industrien) in den Kontext der SMArDT Abstraktionsebenen und in System- und Testmodellen ein. Die Aufteilung nach System- und Testmodellen erfolgt nach Abbildung 3.9. Wird ein unabhängiges Modell für das Testen modelliert, ist dieses Modell einem Testmodell zuzuordnen. Das Einordnen in die SMArDT Abstraktionsebenen richtet sich nach den verwendeten Elementen in den Modellen und den abgeleiteten Elementen. Zum Teil sind die Abgrenzungen für die Abstraktionsebenen nicht eindeutig und es wurde, die in den Ansätzen verwendeten Beispiele als Indikator verwendet.

In Abbildung 3.10[10] ist zu erkennen, dass sich die MBT-Ansätze vermehrt konkreten Testmodellen (vgl. Ebene C und D) zuordnen lassen. Auf der abstrakten Betrachtungseinheit (Ebene A) gibt es Ansätze, wie zum Beispiel [JTH18a, JTH18b, JTH18c, Juh21], jedoch sind diese nicht als MBT identifizierbar. Zum einen ist die Ebene so abstrakt (Kundensicht), dass es schwierig ist, konkrete Testfälle automatisiert abzuleiten, oder der Aufwand steht in keiner wirtschaftlichen Relation zum Nutzen. Zum anderen beinhaltet die Ebene A kein eindeutig funktionales Verhalten des (Sub-) Systems, um für die Generierung geeignete Ergebnisse zu messen. Infolgedessen werden Testfälle dieser Ebene manuell erstellt.

[9] Schon im Jahr 1956 wurden Überlegungen bezüglich der Forschung der Testfallerzeugung auf der Basis von endlichen Automaten publiziert [Moo56].

[10] Die in Abbildung 3.10 erfassten Ansätze schließen Methoden der Modellprüfung (engl. „model checking“) und Testfälle für das autonome Fahren aus. Im Allgemeinen werden bei der Methodik der Modellprüfung Modelle und beim Testen die Implementierung geprüft [PSAK04]. In dieser Arbeit wird die Implementierung getestet.

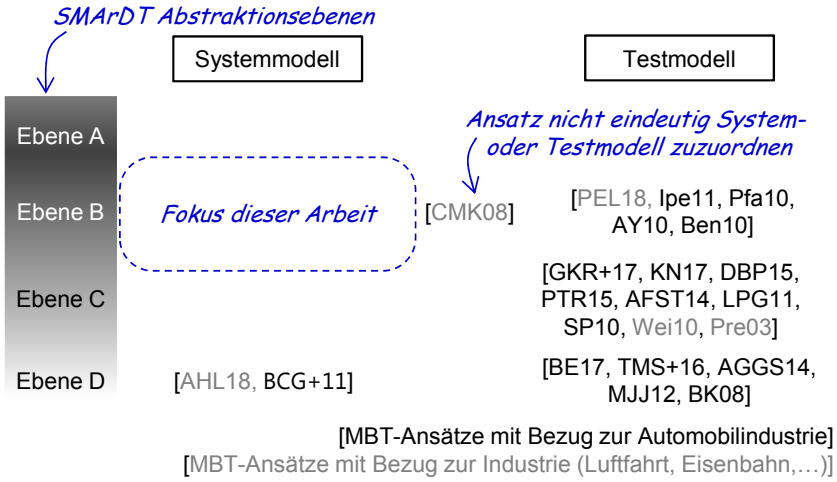


Abbildung 3.10: Einordnung von Ansätzen der modellbasierten Testfallerstellung aus dem Automobilbereich (und verwandten Industrien in grau) für dynamische Tests

Die Ansätze [CMK08, Pel18, Ipe11, Pfa10, AY10, Ben10] sind der SMArDT Abstraktionsebene *B* zuzuordnen. In [CMK08] werden abstrakte UML-Modelle mit Testkriterien angereichert, um anschließend Testfälle zu erstellen. Es wird nicht explizit darauf eingegangen, ob das initiale Modell ein für die Testfallerstellung geeignetes Systemmodell oder ein Testmodell ist. In [Pel18] werden neben einfachen Anforderungen im SysML-Testmodell, komplexere Anforderungen als Formel verfasst und ermöglichen somit die anforderungsbasierte Testfallerstellung für die Luftfahrtindustrie. In [Ipe11] wird aus Anforderungen eine DSL auf der Basis eines UML Testmodells vorgestellt. Die vorgestellte formale Testfall-Spezifikationssprache wird weiterhin zur Testfallerstellung genutzt. Aus der abstrakten DSL können konkrete Testfälle abgeleitet werden, da die Sprache ebenfalls Timings und konkrete Komponenten und zum Teil Nachrichten umfasst. In [Pfa10] wird das abstrakte Testmodell ebenfalls weiter präzisiert und ergänzt, um eine anforderungsbasierte Testfallerstellung zu gewährleisten. Die Testfälle können mithilfe der Erfahrung der Tester*innen weiter priorisiert werden. Der Ansatz [AY10] konzentriert sich auf Software-Komponenten und erstellt auf Basis von Anforderungen einen endlichen Automaten. Dieser Automat wird mit weiteren Informationen, die für die Tests während der Laufzeit wichtig sind, angereichert, um automatisiert Testfälle zu erstellen. Mittels eines Testmodells werden in [Ben10] Interaktionen zwischen Benutzer*innen, Umgebung und System getestet. Die Testfälle sind auf die Ausführung in einer SIL-Umgebung ausgerichtet.

Die meisten MBT-Ansätze sind dem konkreten technischen Konzept (Ebene *C*) von SMArDT zuzuordnen. Im Ansatz von [GKR+17] werden mittels Component & Connector Modellen Testfälle für eine Simulationsumgebung generiert. In [KN17] werden Testfälle inklusive der automatischen Zuweisung von Testprioritäten aus formalen Modellen abgeleitet. Der Ansatz [DBP15] stellt einen Vorgang für das MBT von sicherheitskriti-

schen eingebetteten Systemen vor. In [PTR15] werden Testfälle auf der Basis von Zustandsdiagrammen erstellt. Die automatisierte Testfallgenerierung kann von Tester*innen angepasst werden. In [AFST14] werden mittels Matlab und Matlab/Simulink [Mat19] funktionsorientierte Modell- und Softwaretetests durchgeführt. Der Ansatz [LPG11] verwendet UML-Testmodellen für die Testfallgenerierung. Da die konkreten Testfälle noch zu abstrakt für die Matlab/Simulink-Testumgebung sind, werden diese Zeitvorgaben angereichert. In [SP10] werden Testfälle aus SCs generiert. Die konkreten Testfälle werden mit einem Mapping der Nachrichten für die HIL-Umgebung ausführbar. In [Pre03] wird umfangreich und ausführlich das Thema modellbasiertes funktionales Testen auf Basis von Testmodellen behandelt. Der vorgestellte Ansatz betrachtet ausschließlich Softwaretests und demonstriert den Testfallgenerator anhand einer Fallstudie mit Chipkarten.

Für die Testfallgenerierung auf der Basis der SMArDT Ebene D werden konkrete Modelle der Realisierung verwendet. Da sich die vorliegende Arbeit auf eine andere Abstraktionsebende konzentriert, wird nur eine exemplarische Auswahl an Ansätzen betrachtet. Der Ansatz [AHL18] nutzt Systemmodelle für die automatisierte Testfallerstellung in der Luftfahrt. In [TMS+16] werden Matlab/Simulink-Modelle von Steuergeräten und der Umwelt verwendet, um damit Tesfälle für Modelle, Software und Hardware abzuleiten. In [BCG+11] werden auf Basis eines Systemmodells Code und Testfälle ausgeleitet. Die Testauswahl wird mittels Testfallabdeckungskriterien optimiert. In [BEK17] werden über ein semiformales UML-Modell mit zusätzlichen Notationen und Theoremen, Testfälle für die Verifizierung der ISO 26262 erstellt. Der Ansatz aus [AGGS14] setzt auf Stichproben der erstellten funktionsorientierten Tests, die speziell für die Domäne der Automobilindustrie zugeschnitten sind. In [MJJ12] wird mittels Testfällen Steuerungssoftware für Komponenten verifiziert. Der Ansatz [BK08b] generiert Testfälle aus Matlab/Simulink-Testmodellen für verschiedene Testumgebungen.

Die hier vorgestellten Ansätze stellen eine automatisierte Generierung von Testfällen in der Automobilindustrie oder ähnlichen Industrien vor. Durchaus lassen sich Testfälle auch manuell aus den Modellen ableiten [Lig09]. Die System- und Testmodelle werden in der Regel aus Anforderungen erstellt. Diese Anforderungen werden größtenteils automatisch verknüpft, um später eine Anforderungsabdeckung transparent darstellen zu können. Die automatisierte Anforderungsverknüpfung der Testfälle reduziert den „Verknüpfungsaufwand“ enorm und ist ein klarer Vorteil der automatisiert modellbasierten Testfallerstellung [UL07].

Der Großteil der MBT-Ansätze nutzt Testmodelle für die Testfallerstellung (vgl. Abbildung 3.10). Ein Grund ist, dass häufig kein (aktuelles) formales, abstraktes und technisches Konzept vorliegt[11]. Sind die Elemente vorhanden, sind es in der Regel nur Skizzen oder nicht ausführbare Diagramme, die zudem auch für die Tester*innen überwiegend nicht verfügbar sind. Ein weiterer Grund ist, dass die Testaktivitäten auch an externe Parteien vergeben werden [Lig09, Wit16]. Diesen externen Parteien liegt häufig kein Systemmodell sondern nur ein Lastenheft vor.

Ein Testmodell bringt durchaus Vorteile. So kann das Systemmodell geprüft, Modellfehler behoben und Inkonsistenzen oder Unvollständigkeiten erkannt werden [Lig09]. Der

[11] Anwendungsfälle des Kunden und die Realisierung sind generell vorhanden.

Mehraufwand für die initiale Erstellung und das Instandhalten eines Testmodells ist für komplexe und größere Systeme wie dem elektrischer Antrieb enorm. Im Rahmen dieser Arbeit wird eine Methode vorgestellt die, die manuelle und automatisierte Ableitung von Testfällen aus abstrakten Systemmodellen (Verhaltensmodellen) unterstützt. Diese funktionsorientierten Testfälle testen Fahrzeugfunktionen des elektrischen Antriebs. Die Testfälle können den Integrations- beziehungsweise Systemtests zugeordnet werden.

Kapitel 4

Modellgetriebene Entwicklung mit SMArDT

In den vorangegangenen Grundlagenkapiteln wurden die Testfallumgebung, die Testfallerstellung und das modellbasierte Testen in der Automobilindustrie beschrieben. Zudem wurden relevante Begrifflichkeiten definiert und SMArDT vorgestellt. SMArDT ist modellbasiert, weißt aber keine modellgetriebene Entwicklung auf. In diesem Kapitel wird eine modellgetriebene Ausprägung/Weiterentwicklung von SMArDT vorgestellt, deren Artefakte zu mehr als nur der Dokumentation genutzt werden. Diese Weiterentwicklung wurde im Kontext der Automobilindustrie für den elektrischen Antrieb entwickelt.

Die vorgestellte „modellbasierte“ SMArDT-Methodik aus Abschnitt 2.4 bringt zwei wesentliche Punkte mit sich, die in dieser Arbeit weiterentwickelt werden. Der erste Punkt ist die Redundanz der natürlichsprachlichen textuellen Anforderungen und der SysML-Modelle. Diese Redundanz kann potenziell zu Inkonsistenzen führen. Insbesondere bei den iterativen und evolutionären Entwicklungsschritten ist dies zu erwarten, da bestehende komplexe Funktionen und Systeme überarbeitet werden und bei einer manuellen Verwaltung schnell der Überblick verloren geht. Der zweite Punkt ist, dass SMArDT keine Richtlinien für die Syntax und Semantik der Modelle vorgibt. Sind die textuellen oder grafischen Modelle nicht wohlgeformt, wohldefiniert und einheitlich können Missverständnisse aufgrund von Mehrdeutigkeiten entstehen und effiziente automatisierte Verfahren sind schwer anzuwenden. Effiziente Methoden aus dem Software Engineering wie die automatisierte Testfallerstellung oder Konsistenzchecks sind insofern nicht anwendbar.

In diesem Kapitel wird auf der bestehenden modellbasierten SMArDT-Ausprägung aus Abschnitt 2.4 aufgebaut, notwendige Definitionen für die modellgetriebene Entwicklung erweitert und eine Funktion aus der Automobilindustrie entwickelt. Diese modellgetriebene SMArDT-Methodik dient als Basis für weitere automatisierte Verfahren, wie beispielsweise die automatisierte Testfallerstellung, dessen Methodik in Kapitel 5, Kapitel 6 und Kapitel 7 vorgestellt wird.

Zu diesem Zweck wird in dieser Arbeit eine Einführung in die hierarchische Dekomposition eines Automobils in Abschnitt 4.1 und die funktionale Abstraktion anhand der Funktion „Anhalten“ in Abschnitt 4.2 gegeben sowie die beiden Zerlegungen in Abschnitt 4.3 separiert vollzogen. Danach wird in Abschnitt 4.4 auf die V&V mit SMArDT 2.0 und abschließend auf die evolutionäre Entwicklung mit SMArDT 2.0 in Abschnitt 4.5 eingegangen. Bei SMArDT 2.0 handelt es sich um einen aus der Praxis motivierten Ansatz, der für die modellgetriebene Anwendung in der Automobilindustrie weiterentwickelt wurde. Im Folgenden wird SMArDT 2.0 als SMArDT bezeichnet, da es sich um eine Ausprägung von SMArDT handelt.

4.1 Hierarchische Dekomposition und Integration

In diesem Abschnitt wird eine Verschärfung der in Abschnitt 2.4 beschriebenen Dekomposition, die hierarchische Dekomposition für die modellgetriebene Verwendung vorgestellt. Die hierarchische Dekomposition steht orthogonal zur funktionalen Abstraktion. Diese Orthogonalität hilft, ein System zu zerlegen und gleichzeitig die Komplexität dieses Systems intellektuell beherrschbar zu machen (vgl. Abschnitt 2.4). Infolgedessen können Unsicherheiten bei der Entwicklung identifiziert, dokumentiert, analysiert und berücksichtigt werden [BBK+21]. Das orthogonale Konzept ist deshalb bei Ansätzen für die Entwicklung komplexer CPS wie [BBH+14, BBK+21, DGH+19] verbreitet und wird auch in dieser Arbeit angewendet (vgl. Abschnitt 2.5). Unterschiede ergeben sich insbesondere im Aufbau und der Anwendung der Abstraktion und der Dekomposition.

Der Zweck der hierarchischen Dekomposition ist Aufteilung der Funktionalität eines Systems auf weitere Subsysteme[1] [Fey10]. Das Ziel ist, die Teilfunktionalitäten unterschiedlich tief in „Teilbäume“ zu gruppieren, um die sich aus der Funktionalität resultierenden Aufgaben intellektuell beherrschbar zu machen.

Im Gegensatz zur Dekompositionen aus Abschnitt 2.4 und Abschnitt 2.5 gibt die hierarchische Dekomposition in dieser Arbeit eine definierte Rangordnung für Funktionen und Systeme vor [Bal09]. Die hierarchische Dekomposition teilt Systeme und ihre Subsysteme in über- und untergeordnete Elemente ein und gibt dem Gesamtsystem somit eine geordnete Struktur. Auch wenn die Hierarchie die Systemstruktur auf definierte gerichtete Strukturen beschränkt, überwiegt der Vorteil der Ordnung [Bal09]. Das Prinzip der Hierarchie wird in dieser Arbeit nicht nur für die Dekomposition verpflichtend angewendet, sondern auch für die daraus resultierende Architektur und die Kommunikation der Architekturelemente. In Folge der strikten hierarchischen Ordnung der Elemente wird eine Modularität der Systeme und deren Subsysteme erleichtert. Vorteile der Modularisierung sind unter anderem ein leichterer Austausch der Elemente und Erweiterbarkeit, eine verbesserte der Wartbarkeit und Überprüfung und eine erleichterte Arbeitsorganisation und Arbeitsplanung [Bal09]. Der vorgestellte SMArDT-Ansatz aus Abschnitt 2.4 schließt eine hierarchische Dekomposition und die Modularisierung zwar nicht aus, fordert diese allerdings auch nicht explizit, wie in dieser Arbeit.

Besonders bei komplexen Systemen, wie den heutigen Automobilen, ist eine hierarchische Dekomposition hilfreich. Die Systeme sind deshalb komplex, da eine hohe Anzahl an Funktionen und deren komplexe Vernetzung in Hardware und Software vorliegen [SZ16, EF17]. In vielen Fällen werden für das Ausführen einer Funktion mehrere vernetzte Hardwarekomponenten benötigt. Zum Beispiel ist die Beschleunigung eines Elektrofahrzeugs ein komplexes Zusammenspiel des Fahrwerks, des Antriebsstrangs, der Leistungselektronik, der E-Maschine, des Hochvoltspeichers und des Ladegeräts. Aus diesem Grund werden in einer hierarchischen Dekomposition die Funktionen auf die einzelnen Dekompositionsschichten herunter gebrochen und geordnet. Abbildung 4.1 gibt eine exemplarische Übersicht über die hierarchische Dekomposition mit SMArDT der Drehmomentfunktionalität eines elektrischen Antriebs.

[1] In [ISO15a, WRF+15] werden Subsysteme als Systemelemente bezeichnet.

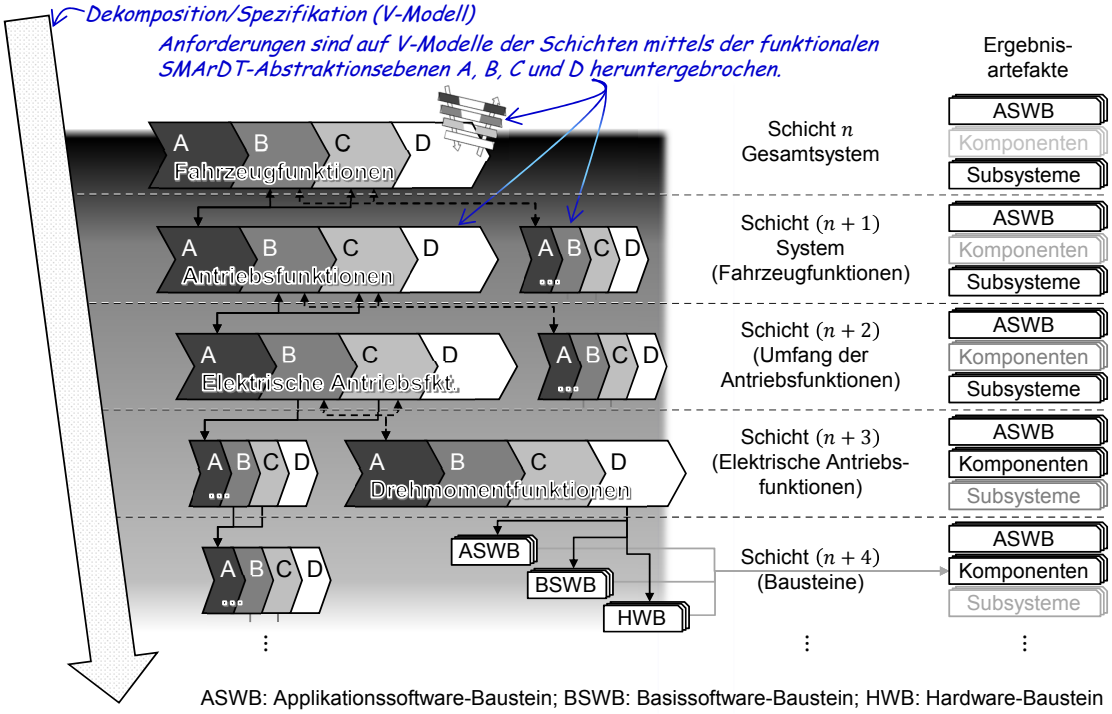


Abbildung 4.1: Exemplarische Übersicht über die hierarchische Dekomposition in Schichten von SMArDT mit einer Zerlegung nach [SZ16, MSWD15] und einem Vergleich mit der Dekomposition/Spezifikation des V-Modells

Spezifikation und Dekomposition

In der hierarchischen Dekomposition werden die Funktionen des Gesamtsystems, hier die Fahrzeugfunktionen, zunächst auf die Subsysteme wie Antriebsfunktionen, Fahrwerksfunktionen, Karosseriefunktionen, Multi-Media-Funktionen sowie Fahrerassistenzfunktionen nach [SZ16] partitioniert. Diese Subsysteme sind historistisch gewachsen, haben sich unternehmensübergreifend etabliert, können meist Steuergeräten zugewiesen werden [SZ16] und sind häufig in der Organisationsstruktur widergespiegelt. Diese Subsysteme werden anschließend in weitere Subsysteme partitioniert bis, hin zu reinen Hardware- und Softwarebausteinen. Der Begriff „Funktion“ ist bewusst gewählt, betont die funktionsorientierte Entwicklung/ Spezifikation. Folglich wird der Fokus, wie in Abschnitt 2.4 nicht auf die Hardwaresysteme des Fahrzeugs gelegt, wie traditionell üblich [SZ16], sondern auf die Funktionen des Fahrzeugs[2]. Allerdings werden dei Funktionen, im Gegensatz zu Abschnitt 2.4, nicht aus bestehenden Hardwarefunktionen gewonnen, sondern abstrakt über Systeme hinweg betrachtet. Die (Sub-)Systeme definieren sich folglich über ihre Funktionen. Die funktionsorientierte Spezifikation fördert eine interdisziplinäre (me-

[2]Auch in anderen Industrien gibt es vergleichbare Ansätze. So wird in einem Ansatz aus der Prozesstechnik bei der Auslegung von Prozessen der Fokus nicht mehr auf die Prozesskomponenten, sondern auf die Prozessfunktionen gelegt [FS08].

chatronische) Betrachtungsweise des Systems Engineerings. Infolgedessen können die Funktionen unabhängiger von bestehenden Hardwaresystemen entwickelt werden, als das bei traditionellen Methoden der Fall ist. Somit kann domänenübergreifend nach der geeignetsten Lösung gesucht. Darüber hinaus ist im Fall eines nicht zufriedenstellenden Ergebnisses in einer Domäne nur eine geringere Konzeptarbeit für die Entwicklung in einer anderen Domäne nötig. Der interdisziplinäre Produktentwurf hilft das synergetische Zusammenwirken der Domänen zu berücksichtigen und ein „Silodenken“ [TF17] der einzelnen Disziplinen zu vermeiden. Dieses synergetische Zusammenwirken kann mit der richtigen Organisationsstruktur unterstützt werden. Theorie und Praxis 4.1 reflektiert gewonnene Erfahrungen mit System- und Organisationsstrukturen und gibt Tipps für eine Umsetzung von SMArDT.

Theorie und Praxis 4.1 (Strukturbeziehungen zwischen System und Organisation)**.** *Homomorphismus oder auch Spiegelungshypothese bezeichnet die strukturerhaltende Beziehung zwischen zwei Dingen, wie der Struktur eines Systems und der Struktur der Arbeitsteilung von Unternehmen [Con68, CB16]. Die hierarchische Dekomposition bezieht sich nicht explizit auf die Struktur eines Unternehmens, sondern spiegelt eine Struktur des entworfenen Systems mit deren Funktionen wider. Allerdings herrscht zwischen der Unternehmensstruktur und deren entworfenen Systemen eine sehr enge Beziehung, sodass sie in einigen Fällen identisch ist [Con68]. Nach [Con68, HC90] entwerfen Unternehmen Systeme, die eine Organisation und den Kommunikationsbedarf des Systemdesigns widerspiegeln. Diese Spiegelung findet sich in der Organisation und umgekehrt im System wider [FP08, CB16]. Wird diese strukturerhaltende Beziehung während der Entwicklung nicht beachtet, neigen Strukturen großer Systeme dazu, sich während der Entwicklung aufzulösen [Con68, CH05]. Die in Abbildung 4.1 abgebildete Struktur ist historisch mit der Struktur beziehungsweise Architektur des Automobilsystems gewachsen [SZ16].*

Strukturen wie die in der Automobilindustrie können auch Auswirkungen auf den Endpunkt der Definition von Subsystemen und Teilfunktionen haben. Zum Beispiel ist es möglich, Funktionen zu dekomponieren, bis bekannte oder bereits existierende Hardware- oder Softwareelemente getroffen werden. Alternativ, falls die Systeme schon vorher bekannt sind, können Teilfunktionen nach der Organisationsstruktur ausgerichtet sein. Eine weitere Möglichkeit ist, Funktionen soweit zu dekomponieren, bis reine Hardware- oder Softwareelemente vorliegen. Da eine vorzeitige zu schnelle Modularisierung Gefahren birgt [CB16], wird in dieser Arbeit empfohlen, die häufig komponentenorientierte Arbeitsteilung iterativ auf die funktionsorientierte Arbeitsteilung auszurichten. Die funktionsorientierte Entwicklung ist ein wichtiger Bestandteil der modellbasierten Entwicklung mit SMArDT. Da sowohl die Struktur einer Arbeitsteilung als auch die Struktur eines Systems „lebt“ und flexibel ist, kann sich die hierarchische Dekomposition eines Systems über die Zeit ändern.

In Abbildung 4.1 werden in der Schicht (n) die Fahrzeugfunktionen mittels der funktionalen Abstraktion konzeptionell entwickelt und gruppiert. Das Ergebnis der Ebene B, der funktionalen Abstraktion sind Anforderungen an die Funktionalität der tiefer liegenden Subsysteme der Schicht $(n+1)$. Die auf Ebene C erstellten Modelle stellen formale Anforderungen an die Schnittstellen der tiefer liegenden Subsysteme der Schicht $(n+1)$. Diese

Anforderungen werden mit den jeweiligen Verantwortlichen der Subsysteme abgeklärt und falls nötig korrigiert. Dieser Prozess ist iterativ und wird über ein gemeinsames Releasemanagement abgestimmt. Das heißt, falls sich in der Verfeinerung, der Subfunktion in Schicht $(n+1)$, widersprüchliche Anforderungen ergeben, werden diese mit den Verantwortlichen der übergeordneten Funktion, der Schicht (n), erneut abgestimmt. Das Ziel ist eine modulare, austauschbare und funktionale Spezifikation des Gesamtsystems zu erhalten, um schnell und agil auf technische und funktionale Änderungen reagieren zu können.

Alle Funktionalitäten, die nicht an die tieferen Schichten weitergereicht werden, werden von den Verantwortlichen selbst spezifiziert. In den „höheren“ Schichten, wie (n) und $(n+1)$, sind dies vorwiegend Applikationssoftwarebausteine. Ein Beispiel für eine Applikationssoftware des Gesamtsystems ist eine allgemeine Betriebsstrategie des Fahrzeugs, nach der sich alle anderen Funktionalitäten richten müssen. In den „tieferen“ Schichten werden hingegen die mechatronischen Funktionsbausteine, Applikationssoftware-, Basissoftware- und Hardwarebausteine vollständig spezifiziert (vgl. Drehmomentfunktionen in Abbildung 4.1). Diese werden konkreten Komponenten zugeordnet. Nach diesem Prinzip werden konfigurierbare und wiederverwendbare Funktionsbausteine spezifiziert. Diese Funktionsbausteine und andere Funktionsumfänge werden intern spezifiziert und entwickelt oder können an externe Partner vergeben werden.

Aufgrund dieser Besonderheiten werden die Ergebnisse der höheren Schichten mit systemübergreifenden Applikationssoftwarebausteinen, partitionierten Subsystemen und ohne konkrete Komponenten, assoziiert. Die Ergebnisse der tieferen Schichten werden andererseits mit allen Funktionsbausteinen und konkreter Hardware, den Komponenten assoziiert. Je tiefer die Schicht und je weiter die Ebene, desto detaillierter ist die Spezifikation. Infolgedessen kann es dazu kommen, dass in tieferen Schichten keine Ebene A notwendig ist, da die Spezifikation der darüber liegenden Schicht ausreichend ist. Analog kann es in höheren Schichten dazu kommen, dass keine Realisierung (Ebene D) benötigt wird.

Jede Dekompositionsschicht kann die übergeordnete Schicht $(n-1)$ als Kund*in und die unterordnete Schicht $(n+1)$ als Auftragsnehmer*in sehen. Ob Kund*innen oder Auftragsnehmer*innen intern, extern oder innerhalb der eigenen Gruppe liegen, ist in SMArDT nicht vorgegeben und kann je nach Projekt entschieden werden . Ist entschieden, wo und wie die spezifizierten Funktionsbausteine umgesetzt werden, können diese realisiert und in die jeweiligen Komponenten, (Sub-) Systeme $(n+1)$ bis hin zum Gesamtsystem integriert werden.

Integration der dekomponierten Subsysteme

Abbildung 4.2 stellt die Integration der realisierten Funktionsbausteine schematisch dar. Gestartet wird „unten“ mit den bereits spezifizierten und realisierten Funktionsbausteinen. Diese werden auf die vorgesehenen Komponenten integriert. Applikationssoftwarebausteine der höheren Schichten ohne Hardwarekomponenten werden mit Methoden der Softwarearchitektur optimiert und auf den Komponenten installiert. So wird zum Beispiel der Applikationssoftwarebaustein der Betriebsstrategie für das Laden der Schicht (n) auf

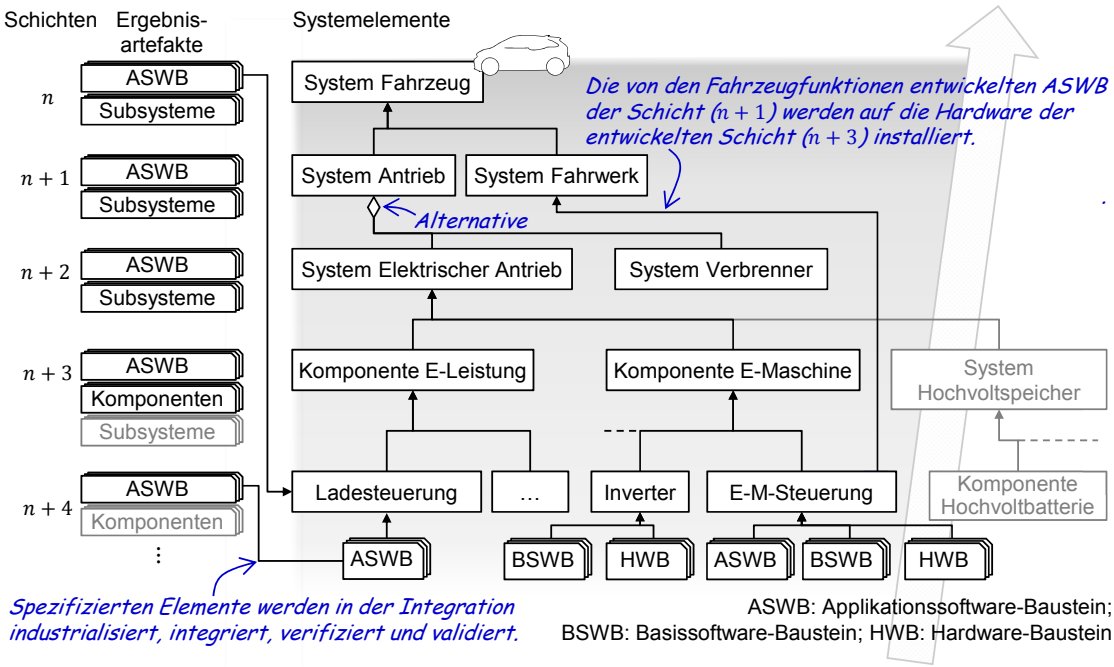


Abbildung 4.2: Exemplarische Übersicht über die Realisierung und Integration mit SMArDT und einem Vergleich mit der Integration des V-Modells

der Ladesteuerung der E-Leistungskomponente der Schicht $(n+3)$ installiert, falls es kein zentrales Hardwaresteuergerät gibt. Auch zeitkritische Softwarefunktionen des Fahrwerks können analog direkt auf der zugehörigen Komponenten (hier Komponente E-Maschine) installiert werden. Die Komponenten und Subsysteme werden analog in ihre zugehörigen Systeme integriert. Das System elektrischer Antrieb integriert in Abbildung 4.2 die Komponenten E-Leistung, E-Maschine und das Subsystem Hochvoltspeicher. Da es sich bei dem System elektrischer Antrieb um eine Alternative beziehungsweise Variante des Systems Antrieb handelt, kann dieses modular entweder allein als BEV oder in Verbindung mit einem Verbrenner zusammen als PHEV in das System Antrieb integriert werden. Aufgrund der Modularität werden diesen Systemen in SMArDT eigenständige Schichten (hier $n+2$) zugesprochen, die in dem übergeordneten System (hier Antrieb) integriert werden. Das System Antrieb stellt in diesem Zusammenhang übergeordnete, übergreifende und verknüpfende Funktionen der Subsysteme bereit. Das Gesamtsystem Fahrzeug der Schicht (n) integriert zuletzt alle Systeme der Schicht $(n+1)$.

Die in diesem Abschnitt vorgestellte Methodik von SMArDT gibt einen Rahmen für die Dekomposition und Integration der Funktionen und der Subsysteme beziehungsweise der Komponenten vor. Diese Methodik erleichtert sowohl die Integration als auch die V&V der einzelnen Funktionen und Subsysteme. Zusätzlich werden klare Schnittstellen für die funktionale Abstraktion der einzelnen Elemente geschaffen.

4.2 Funktionale Abstraktion der Funktion Anhalten

Die in Abschnitt 4.1 beschriebene hierarchische Dekomposition steht orthogonal zur funktionalen Abstraktion. Abbildung 4.3 ist die Basis für eine modellgetriebene Erweiterung von Abbildung 2.7 und zeigt die grundlegende funktionale Abstraktion in SMArDT. Des Weiteren sind die essenziellen Elemente der Spezifikation auf der linken Seite des V-Modells und exemplarischen Elemente der rechten Seiten des V-Modells dargestellt.

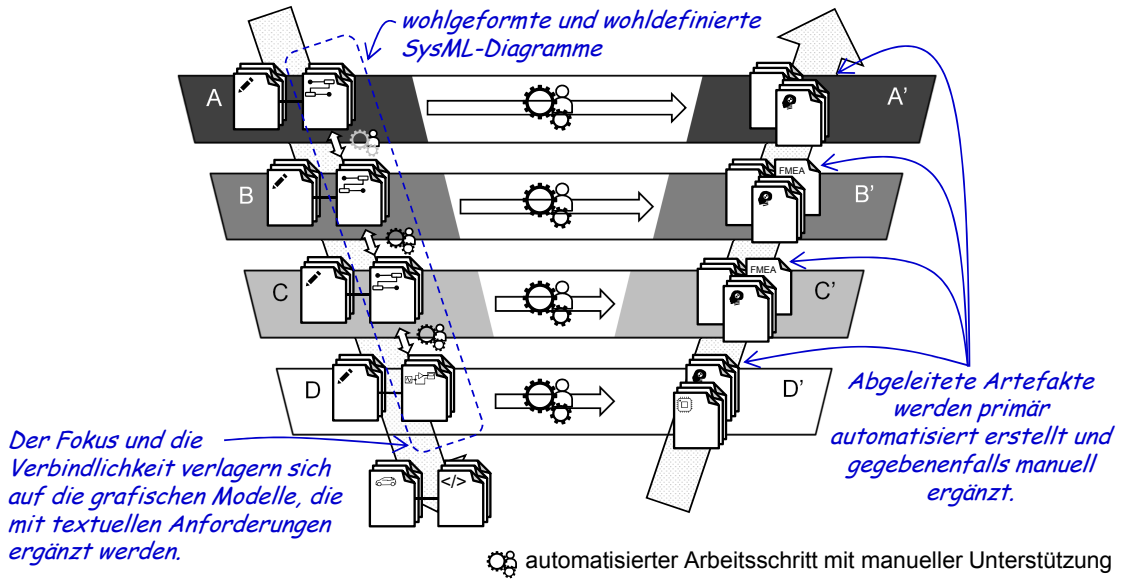


Abbildung 4.3: Übersicht über die funktionale Abstraktion auf Ebenen mit der SMArDT (Erweiterung von Abbildung 2.7)

Im Fokus stehen nicht mehr die textuellen Anforderungen, die auf der Basis der SysML Modelle entwickelt werden, sondern die SysML Modelle selbst. Diese grafischen (formale) Modelle entsprechen in den Modellierungssprachen verankerten Wohlgeformt- und Wohldefiniertheitsbedingungen und werden mit textuellen Anforderungen sinnvoll ergänzt. Die Informationen der Kernfunktionalität sind in den grafischen Modellen abgebildet und nicht redundant in den textuellen Anforderungen enthalten. Die Gesamtheit der grafischen Modelle mit ihren textuellen Anforderungen wird als Gesamtmodell bezeichnet. Für rechtlich verbindliche Dokumente wie Lastenhefte wird folglich das komplette Gesamtmodell exportiert.

Auf der Basis dieser formalen grafischen Modelle können automatisierte Verfahren über Ebenen und Schichten übergreifend angewendet werden. Die effiziente automatisierte Ableitung von Elementen wie Testfällen oder einer Fehlermöglichkeits- und Einflussanalyse (FMEA)[3] ist möglich und kann alle Beteiligten unterstützen. Die modellgetriebene Spezifikation wird von der Betrachtungseinheit bis hin zum Implementierungsmodell verfeinert und ermöglicht eine transparente durchgängige Modellierung sowie Konsistenzchecks.

[3]Die automatisierte Ableitung einer FMEA ist nicht Teil dieser Arbeit.

Die Aufteilung und der Verwendungszweck der Abstraktionsebenen dieser Arbeit entsprechen den vorgestellten Abstraktionsebenen in Abschnitt 2.4. Die Überschneidungen und Abgrenzungen zu anderen Abstraktionsmethoden entsprechen folglich den in Abschnitt 2.5 vorgestellten Aspekten. Im Unterschied zu den Abstraktionsebenen aus Abschnitt 2.4 sind die Ebenen mit ihren Modellierungselementen und Beziehungen in einem Referenzmodell eingeordnet. Außerdem sind der Zweck, die Elemente und die Ergebnisse jeder Ebene klar definiert und festgelegt. Diese feste Definition der Ebenen und ihrer Elemente schränkt die individuellen Modellierungsansätze für Subsysteme ein. Aus der Sicht des Gesamtsystems überwiegen allerdings die Vorteile wie

- Vergleichbarkeit einzelner Lösungskonzepte wie auch deren Entwicklungsstatus,
- erleichterte Analyse der Wechselbeziehungen der Elemente [Hit07],
- Effizienzsteigerungen aufgrund zuverlässiger Ergebnisse sowie wiederverwendbarer Tools und Prozesse und
- Gleichgewicht der Abstraktionsebenen, da die gleichen Ebenen die gleichen Informationen liefern [DIN15].

Begonnen wird die funktionale Abstraktion immer „oben“ in der Ebene A über die Ebenen B und C bis hin zur Ebene D. Die abgeleiteten Hardware- und Softwareelemente der Ebene D bilden das jeweils realisierte Produkt. Das jeweilige Produkt ist für jedes `System` der Dekompositionsschicht ein anderes. Entscheidend ist, dass über die Integration aller relevanten Produkte in das Gesamtsystem das finale Gesamtprodukt entsteht. Das Referenzmodell dieser Arbeit in Abbildung 4.4 stellt die Elemente der funktionalen Abstraktion und deren Beziehungen in den Zusammenhang mit den Abstraktionsebenen von SMArDT.

Folgende Kontextbedingungen gelten für das Referenzmodell in Abbildung 4.4:

- Eine `Subfunktion` in der Dekompositionsschicht n bedient sich entweder aus `Funktionsbausteinen` oder stellt Anforderungen an die `Kundenfunktion` oder an das `Funktionsprinzip` der nächsten Dekompositionsschicht $n + 1$.
- Eine `Subfunktion` in der Dekompositionsschicht n kann nur Anforderungen an `Funktionsprinzipien` der gleichen Dekompositionsschicht n stellen.
- Ein `Funktionsbaustein` kann nur aus einer der folgenden Kombinationen bestehen:
  - Einem `Applikationssoftwarebaustein`
  - Einem `Applikationssoftwarebaustein`, einem `Basissoftwarebaustein` und einem `Hardwarebaustein`
  - Einem `Basissoftwarebaustein` und einem `Hardwarebaustein`
  - Einem `Hardwarebaustein`

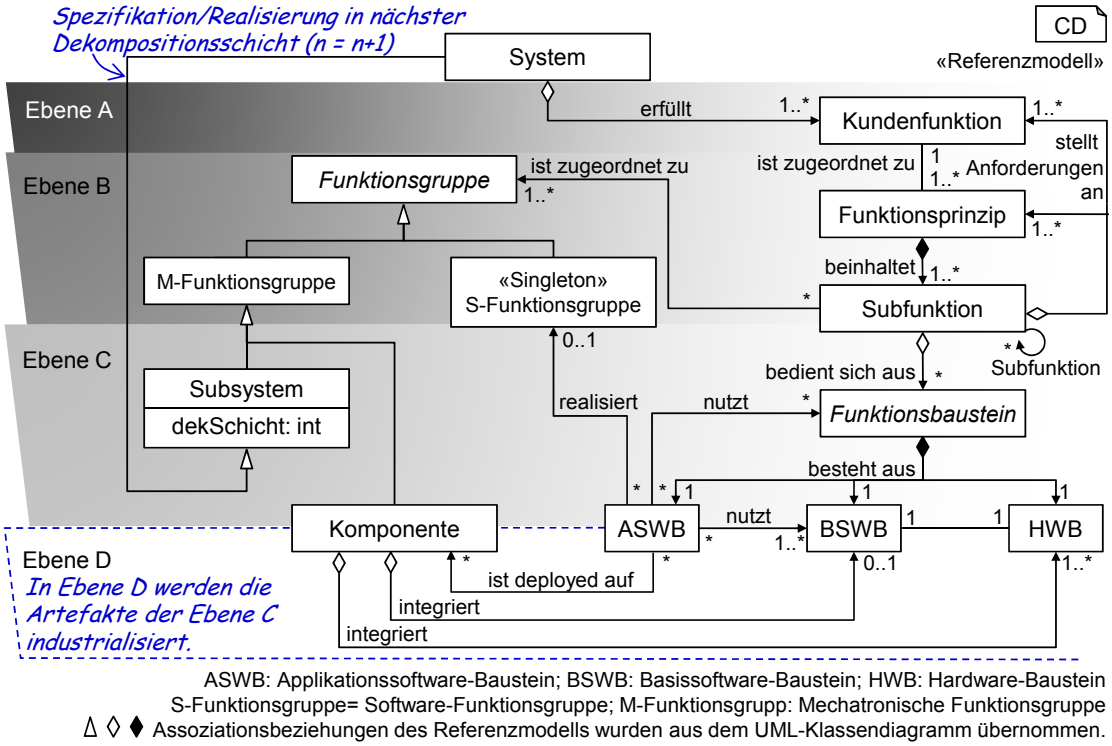


Abbildung 4.4: Referenzmodell der Abstraktion hinterlegt mit den SMArDT Ebenen

- Ein `Applikationssoftwarebaustein` realisiert entweder einen Teil der `S-Funktionsgruppe`[4] oder ist über „`ist deployed auf` → `Komponenten` einer `M-Funktionsgruppe`“ zugeordnet.
- Eine `M-Funktionsgruppe`[5] wird durch ein `Subsystem` oder eine `Komponente` realisiert.
- Ein `Subsystem` der Dekompositionsschicht n ist ein `System` der nächsten Dekompositionsschicht $n + 1$.
- Ein `System` der Dekompositionsschicht n kann nur `Funktionsgruppen` der eigenen Dekompositionsschicht n verwenden.

Die linke Seite des Referenzmodells in Abbildung 4.4 umfasst die Elemente/Modelle, die den Entwicklungsprozess in SMArDT beschreiben. Zu diesen gehören die abstrakte `Funktionsgruppe`, die `M-Funktionsgruppe`, die `S-Funktionsgruppe` und das `Subsystem`. Das `System` und die `Komponente` kann sowohl den systembeschreibenden technischen

[4]Eine S-Funktionsgruppe (Software-Funktionsgruppe) gruppiert in SMArDT Funktionen, die ausschließlich aus Applikationssoftware(-bausteinen) bestehen (siehe Anhang B).

[5]Eine M-Funktionsgruppe, auch mechatronische Funktionsgruppe genannt, beschreibt in SMArDT ein mechatronisches System, dessen Umsetzung noch nicht (vollständig) spezifiziert ist (siehe Anhang B).

Elemente als auch dem Entwicklungsprozess zugeordnet werden. Die rechte Seite in Abbildung 4.4 umfasst die in SMArDT systembeschreibenden technischen Elemente beziehungsweise Modelle. Zu diesen gehören die `Kundenfunktion`, das `Funktionsprinzip`, die `Subfunktion`, der `Funktionsbaustein`, der `Applikationssoftwarebaustein`, der `Basissoftwarebaustein` und der `Hardwarebaustein`.

Zu beachten ist, dass hier die einzelnen `Funktionsbausteine` vereinfacht modelliert sind, um die Komplexität zu reduzieren. Die abgebildeten `Applikationssoftware-`, `Basissoftware-` und `Hardwarebausteine` bestehen aus mehreren Elementen, die eindeutig referenzierbar sind. Die Funktion der `Applikationssoftware-`, `Basissoftware-` und `Hardwarebausteine` werden in der Spezifikation in SysML beschrieben. Im Fall von `Applikationssoftwarebausteinen` wird die Funktionsspezifikation mit deren Schnittstellen entweder direkt und/oder über weitere Verfeinerungsschritte mit Modellen in Code wie Java oder C-Code realisiert. Im Fall von `Basissoftware-` und `Hardwarebausteinen` wird die Funktionsspezifikation in entsprechenden CAD Umgebungen modelliert, simuliert und über mehrere Iterationsschritte optimiert [DRW+20]. Anschließend wird die qualifizierteste Lösung realisiert. Diese realisierte Lösung entspricht der Verhaltens- und Strukturbeschreibung, da diese Eigenschaften der Spezifikation erfüllen.

4.2.1 Ebene A, die Betrachtungseinheit

Zweck: Der Zweck der Ebene A ist, Anforderungen der Interessenvertreter*innen an ein `System` zu definieren. Im Zuge dessen werden die Umweltbedingungen und der Kontext bestimmt. Die Umweltbedingungen beschreiben die unmittelbar einwirkende Umgebung auf das `System` oder auf die Funktion. Der Kontext beschreibt den Sach- und Situationszusammenhang des Systems und/oder der Funktion. Auf diese Weise wird identifiziert, welche von Interessenvertretern*innen benötigten Funktionen in der definierten Umgebung bereitgestellt werden können. Folgende Elemente werden zur Modellierung auf dieser Ebene verwendet:

Folgende Elemente werden zur Modellierung auf dieser Ebene verwendet:

- **UCD** beschreibt die Schnittstellen des Systems zur Umwelt und anderen Systemen, den Kontext und identifiziert Interessenvertreter*innen.
- **Textuelle Anforderung** ergänzen das UCD und bestimmen gegebenenfalls weitere Details.
- **Anforderungsdiagramm** stellen die Zusammenhänge von textuellen Anforderungen dar.
- **Blockdefinitionsdiagramm (BDD)** [optional] illustriert die Beziehungen zwischen den geforderten Funktionen und der Systeme.

Ergebnisse:

- Die Umweltbedingungen sind bestimmt.

- Der Lösungsraum ist charakterisiert.
- Der Kontext ist bestimmt.
- Die Interessenvertreter*innen sind identifiziert.
- Die Einschränkungen des `Systems` sind herausgearbeitet.
- Die Anforderungen an ein `System` sind identifiziert und in Zusammenhang gesetzt.

In der ersten Ebene A wird entschieden, was entwickelt werden soll (vgl. Abschnitt 2.4). Ein `System` erfüllt mindestens eine `Kundenfunktion` (vgl. Abbildung 4.4). Diese `Kundenfunktion` wird von Interessenvertreter*innen (kurz Kund*in) angefragt. Die Interessengruppen können andere Systeme, Normen, Standards, Gesetze, weitere Bereiche einer Organisation oder Personen sein. Um zu bestimmen, ob diese `Kundenfunktion` erfüllt werden kann, werden die Umweltbedingungen, der Lösungsraum, der Kontext, die Interessenvertreter*innen und gegebenenfalls Einschränkungen des Systems identifiziert. Falls das `System` die Anforderung erfüllen kann, wird diese Anforderung angenommen und mit anderen Anforderungen in Zusammenhang gesetzt. Der visuell dargestellte Zusammenhang erleichtert die intellektuelle Identifikation von widersprüchlichen, abhängigen und verwandten Anforderungen. Zu diesem Zweck werden die gewünschten `Kundenfunktionen` der Interessengruppen auf Ebene A definiert oder falls Anforderungen aus anderen Systemen der Schicht n gestellt wurden, ausgewertet und gegebenenfalls ergänzt. Zusätzlich werden in der Ebene A die gegebenen Schnittstellen aus Schicht $n-1$ zur Umwelt und anderen Systemen festgehalten. Das Ergebnis der Ebene sind Anforderungen an das betrachtete `System` aus der Sicht einer Kund*in. Abbildung 4.5 illustriert ein UCD der Ebene A in der Dekompositionsschicht Fahrzeug.

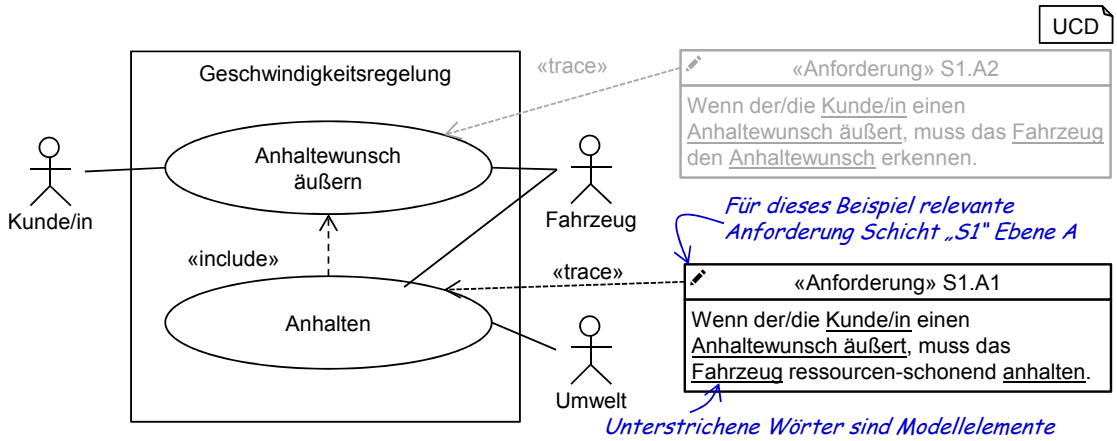


Abbildung 4.5: UCD mit exemplarischen Anforderungen der Schicht Fahrzeug in der Ebene A, der Funktion „Anhalten“ (nebensächliche Elemente in Grau)

Abbildung 4.5 ist der Anwendungsfall (engl. „use case“) `Anhalten` und deren involvierte Akteure, `Kund*in`, `Fahrzeug` und `Umwelt`. Das UCD wurde aus den SysML Diagrammen

ausgewählt, da es sich für die Darstellung von Konzepten, Kontexten und Interaktionsmöglichkeiten zwischen den Akteuren wie Kund*innen und Systemen eignet [OMG17a]. Der Systemkontext des Anwendungsfalls in Abbildung 4.5 befasst sich mit der `Geschwindigkeitsregelung`.

Die textuelle ≪`Anforderung`≫ `S1.A1` ist mit den Elementen des UCDs verknüpft und spezifiziert diese weiter. Auf der rechten Seite des „Vs“ (vgl. Abbildung 4.3), der Integration (A') wird betrachtet, ob und wo eine Aussage über die Qualität des spezifizierten Systems getroffen werden kann. Zum Beispiel unter welchen Bedingungen kann eine Funktion eines Systems überhaupt getestet werden und was wird für die Freigabe benötigt. Die aus der Spezifikation abgeleiteten Artefakte sind eine Teilmenge der Ebene A und werden mittels A' gekennzeichnet. Ein typisches abgeleitetes Element der Ebene A' ist das funktionale Testkonzept. Die entstandenen Modellelemente und Ergebnisse der Ebene A werden der nächsten Ebene B zur Verfügung gestellt.

4.2.2 Ebene B, das funktionales Konzept

Zweck: Der Zweck der Ebene B, dem funktionalen Konzept, liegt darin, zu spezifizieren, wie das System funktionieren soll. Hierzu werden die kundenorientierten Ergebnisse auf die abstrakten Merkmale, Attribute, Funktions- und Leistungsanforderungen des Systems abgebildet. Soweit es der Kontext, die Umweltbedingungen und der Lösungsraum zulassen, wird keine spezifische Implementierung impliziert. Um eine Rückverfolgbarkeit zu gewährleisten, werden die Artefakte aus der Ebene B mit den zugehörigen Ergebnissen aus Ebene A verknüpft. Ist eine Ebene B und Ebene C aus der Dekompositionsschicht $n-1$ vorhanden, werden die Elemente wie Schnittstellen, Aktionen, Zustände, Blöcke etc. auf der Ebene B der Schicht n übernommen. Die abstrakten Funktionszusammenhänge lassen eine erste funktionale Architekturbeschreibung mit Abhängigkeiten zu.

Elemente:

- **AD** beschreibt den genauen Ablauf des Anwendungsfalls.
- **SC** stellt die Zustände des betrachteten Systems beziehungsweise der Funktionen dar.
- **Sequenzdiagramm (SD)** spezifiziert den Ablauf eines Anwendungsfalls mit dem Fokus auf Interaktionen.
- **BDD** illustriert Zusammenhänge der verwendeten Funktionsgruppen.
- **IBD** beschreibt die Schnittstellen und Informationsflüsse der Funktionen.
- **Textuelle Anforderung** ergänzen die Diagramminformationen.
- **Anforderungsdiagramm** stellen die Zusammenhänge von textuellen Anforderungen dar.

Ergebnisse:

- Die abstrakte Funktionsbeschreibung einschließlich des Funktionsablaufs, der groben Funktionsarchitektur, der Systemschnittstellen und der Informationsflüsse[6] ist abgestimmt und modelliert.
- Die funktionalen Anforderungen inklusive Schnittstellen stehen fest.
- Die Funktionsanforderungen erfüllen die Anforderungen der Interessengruppen aus Ebene A und sicherheitsrelevanten Informationen und Elemente nach ISO 26262 sind identifiziert.

Die Hauptaufgabe der Ebene B ist die Beschreibung des `Funktionsprinzips`. Beschreibt das `Funktionsprinzip` den Zusammenhang von physikalischen Effekten sowie geometrischen und stofflichen Merkmalen, so ist dann auch von einem *Wirkprinzip* die Rede, ähnlich dem Wirkprinzip in [PBFG07]. Im weiteren Verlauf der Arbeit wird ausschließlich von `Funktionsprinzipien` gesprochen, da Wirkprinzipien eine echte Teilmenge von `Funktionsprinzipien` sind. Das `Funktionsprinzip` impliziert nach Möglichkeit keine technische Lösung, da sich die (technischen) Anforderungen während der Systementwicklung häufig ändern [Pre03]. Können Entwickler*innen ihre technischen Versprechungen nicht halten, so kann auf der Basis des unabhängigen und lösungsneutralen `Funktionsprinzips` schnell eine andere technische Lösung gefunden werden. Für eine effizientere Systematisierung wird der Lösungsraum dennoch domänenspezifisch eingeschränkt, um die Ausrichtung von Beschreibungsmitteln zu limitieren. Diese Limitierung schränkt allgemeine Probleme auf domänenspezifische ein, vereinfacht die Lösungsfindung und steigert infolgedessen die Effizienz in der Systementwicklung, der Validierung und der Verifizierung. Ein Fahrzeug in der Domäne der Automobilindustrie wird beispielsweise mit einem oder mehreren Motoren angetrieben und nicht mit einem Strahltriebwerk[7]. Wie diese Motoren technisch umgesetzt werden, bleibt auf der Ebene B offen.

Bei der Erarbeitung des funktionalen Konzepts werden die `Funktionsprinzipien` den jeweiligen `Kundenfunktionen` zugeordnet. Die `Funktionsprinzipien` sind den jeweiligen `Kundenfunktionen` zugeordnet und umgekehrt. In tieferen Dekompositionsschichten $n + m$ werden in den `Subsystemen` die `Funktionsgruppen` der höheren Dekompositionsschicht $n+m-1$ verfeinert und gegebenenfalls um spezifische Anforderungen der Ebene A ergänzt. Für die Beschreibung der `Funktionsprinzipien` werden für das Verhalten der Funktion ja nach Eignung die SysML Verhaltensdiagramme AD, SC, SD und für die Strukturbeschreibungen das BDD und IBD eingesetzt. Folgende Fragen sollen mit dem `Funktionsprinzip` beantwortetet werden:

[6] Ein Informationsfluss (engl. „Information Flow“) dient der Darstellung eines Austauschs von Informationen zwischen zwei oder mehreren Elementen auf hohen Abstraktionsebenen, wie beispielsweise der Ebene B [OMG17b, OMG17a]. Informationsflüsse erlauben, nicht vollständig spezifizierte oder weniger detaillierte Materie-, Energie- oder Signalflüsse abstrakt zu modellieren. Auf diese Weise können abstrakte Modelle das intellektuelle Verständnis erleichtern und bewusst Entscheidungen über Details in tiefere Detailebenen des Modells verschoben werden, wie beispielsweise der Ebene C und D.

[7] Bisher haben sich Fahrzeuge mit Gasturbine wie der Chrysler Turbine Car [LL10] in der Praxis nicht durchgesetzt.

- Was wird für die Funktion benötigt? (Materie, Energie, Information)
- Was liegt für die Funktion bereits vor? (Beispiel: Lenkrad, Räder, Informationen, Daten)
- Wie wird die Funktion prinzipiell umgesetzt?
- Welche anderen `Funktionsprinzipien` und `Systeme` haben auf das betrachtete `Funktionsprinzip` einen Einfluss? (Beispiel: Fahrerassistenz auf Motorsteuerung)
- Welche Einflüsse hat das betrachtete `Funktionsprinzip` auf andere `Funktionsprinzipien` und `Systeme`? (Beispiel: Betätigung der Handbremse bei Höchstgeschwindigkeit führt zu einem Ausbrechen des Fahrzeugs)

Gegebenenfalls wird auf Ebene B schon ein Lösungsprinzip beschrieben, dass von den entwickelnden Personen favorisiert wird. Auch das Beschreiben und das Diskutieren unterschiedlicher Lösungen mit disziplinenübergreifenden `Funktionsprinzipien` sind möglich. Beispielsweise können neben hybriden Antrieben auch reine elektrische Antriebe beschrieben werden. Anschließend werden die `Funktionsprinzipien` verglichen, die Entscheidungen dokumentiert und die geeignetste Lösung gewählt. Auf diese Weise kann später nachvollzogen werden, warum ein `Funktionsprinzip` gewählt wurde. Jedes `Funktionsprinzip` besteht aus mindestens einer `Subfunktion`, die das Verhalten der Funktion weiter spezifiziert. Die `Funktionsprinzipien` unterscheiden sich von den `Subfunktionen` dahin gehend, dass in ihrem Datenmodell keine Entitäten wie Größen oder Einheiten hinterlegt sind. Dadurch sind `Funktionsprinzipien` per se unterspezifiziert. Das Datenmodell der `Subfunktionen` umfasst hingegen auch Entitäten wie beispielsweise SI-Einheiten. Diese Entitäten werden den `Subfunktionen` auf der Ebene C zugeordnet.

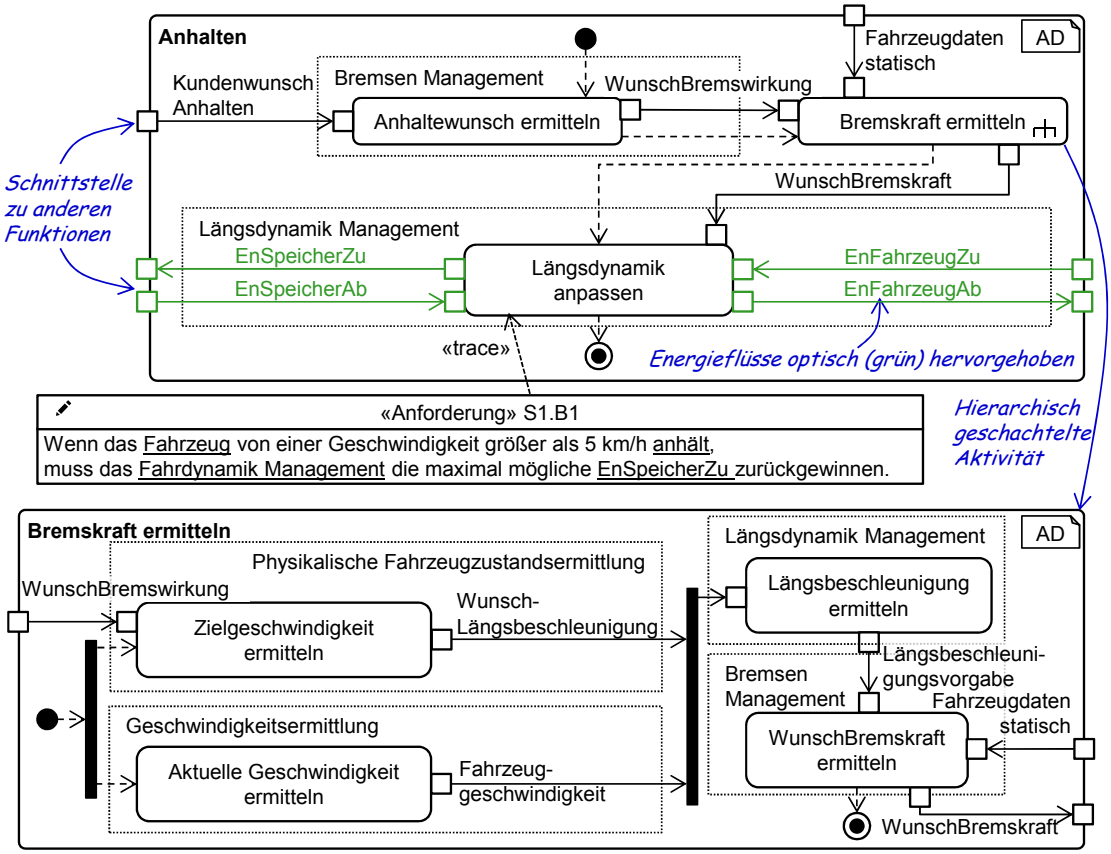
Die `Subfunktionen` werden in einem weiteren Schritt `Funktionsgruppen` zugeordnet. Falls es sich um eine Funktion handelt, die mittels reiner Applikationssoftware umgesetzt werden kann, wird sie der `S-Funktionsgruppe`[8] zugeordnet (vgl. Abbildung 4.4). Die `S-Funktionsgruppe` sammelt alle Applikationssoftwarefunktionen und deren Anforderungen, um diese auf die `Komponenten` mit ausreichend Leistung zu verteilen (engl. „deploy“). Falls die `Subfunktion` noch nicht klar als Applikationssoftware identifiziert wird, wird diese der entsprechenden `M-Funktionsgruppe` zugeordnet. Die Spezifikation der abstrakten Ebene B ist implementierungsneutral, das heißt, sie impliziert keine Lösungsumsetzung. Keine Lösungsumsetzung bedeutet, dass alle Informationsflüsse, Werte, Aufgaben und Objekte in dem Modell abstrakt sind und nur über die Zuordnung von `Funktionsgruppen` einen Bezug zu der Realisierung aufweisen. Theorie und Praxis 4.2 reflektiert Erfahrungen aus der Praxis mit der Abstraktion der Ebene B.

Theorie und Praxis 4.2 (Wahl des Abstraktionsgrads auf Ebene B)**.** *Die Wahl des richtigen Abstraktionsgrads ist für den weiteren Verlauf der funktionalen Abstraktion in tieferen Ebenen von hoher Bedeutung. Wird ein zu hoher Abstraktionsgrad gewählt, liefern*

[8] Die S-Funktionsgruppe existiert im gesamten System über alle Dekompositionsschichten $n+m; m \in \mathbb{N}_0$ nur einmal (vgl. ≪`Singleton`≫ in Abbildung 4.4).

die aus Ebene B abgeleiteten Artefakte keinen oder geringen Mehrwert (Informationen) für die weitere Entwicklung und Absicherung. In schon entwickelten Systemen und Fahrzeugfunktionen wird allerdings zum großen Teil der Abstraktionsgrad der Ebene B zu tief gewählt, nämlich dem auf Ebene C, da die Lösung und deren Umsetzung schon bekannt sind. Eine Möglichkeit, den richtigen Abstraktionsgrad zu finden, ist die schon vorhandenen Elemente in der Entwicklung und Absicherung heranzuziehen. Liegt zum Beispiel eine (aktuelle und etablierte) DSL in Form einer Schlüsselwortbibliothek vor, kann diese als Referenz für einen geeigneten Abstraktionsgrad dienen.

Abbildung 4.6 illustriert ein AD der Ebene B in der Dekompositionsschicht Fahrzeug. In



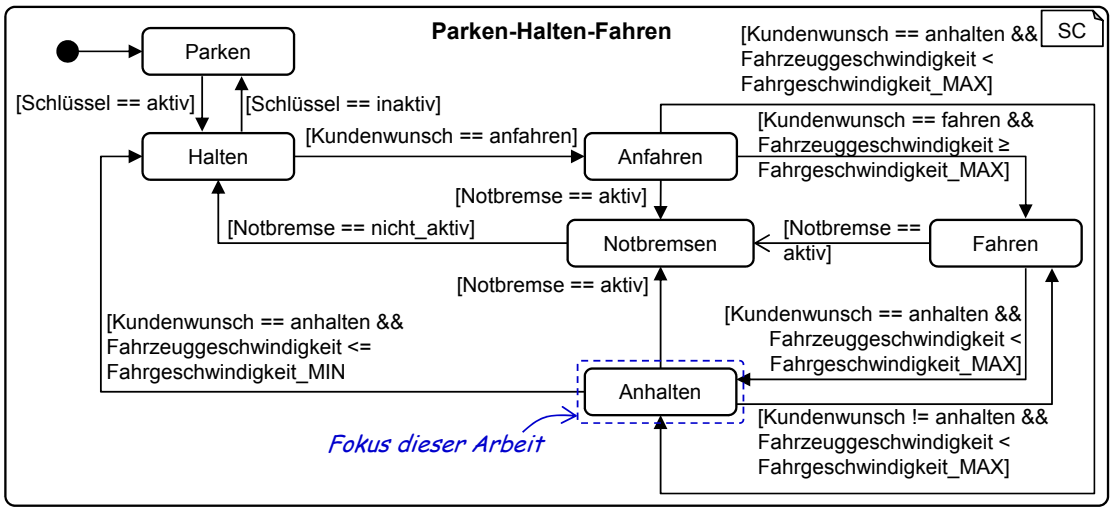
EnFahrzeugAb: Abgeführte Energie Fahrzeug; EnFahrzeugZu: Zugeführte Energie Fahrzeug;
EnSpeicherAb: Abgeführte Energie Speicher; EnSpeicherZu: Zugeführte Energie Speicher;

Abbildung 4.6: AD mit exemplarischen Anforderungen der Schicht Fahrzeug in der Ebene B der Funktion „Anhalten“

der Aktivität `Anhalten` in Abbildung 4.6 werden die UCD aus Abbildung 4.5 entwickelten Anforderungen funktional weiter spezifiziert. Anhalten ist eine Reduktion der Geschwindigkeit bis 5 km/h, wohingegen eine Reduktion der Geschwindigkeit kleiner als 5 km/h bis zum Stillstand des Fahrzeugs als „halten“ definiert ist. Der `Kundenwunsch Anhalten` wird

schließlich erkannt und eine dem Kundenwunsch entsprechende `WunschBremswirkung` für das Fahrzeug wird ermittelt. Darauf folgt die Ermittlung von `WunschBremskraft`, über die Aktion `Bremskraft ermitteln`. Aus der Differenz der `Fahrzeuggeschwindigkeit` und der `WunschLängsbeschleunigung` wird über eine prognostizierte Zeit in der Aktion `Längsbeschleunigung ermitteln` eine `Längsbeschleunigungsvorgabe` für das Fahrzeug ermittelt. Zusammen mit den `statischen Fahrzeugdaten` wie der Masse des Fahrzeugs wird danach die `WunschBremskraft` ermittelt. Außerdem muss das Fahrzeug bei einer Geschwindigkeit größer als 5 km/h die maximal mögliche (`zugeführte Energie Speicher`) zurückgewinnen (vgl. ≪`Anforderung`≫ `S1.B1`).

Neben dem `Funktionsprinzip` wird eine Übersicht aller Aktivitäten (Funktionen), deren Verhalten, deren Kommunikation und Zustandsabhängigkeiten empfohlen. Die Ansicht in Abbildung 4.7 ordnet die Zustände beziehungsweise Aktivitäten wie `Anhalten` am Beispiel eines SCs in den Funktionszusammenhang ein.

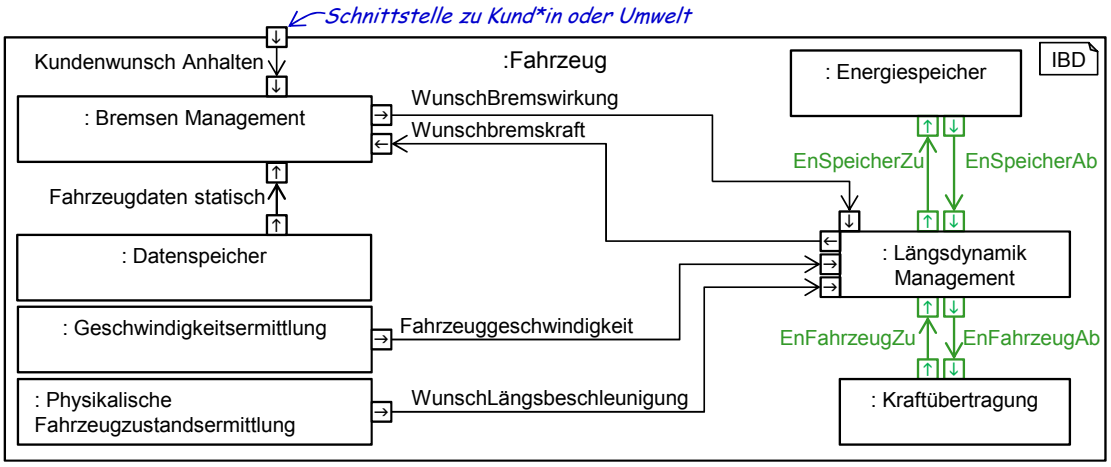


Fahrgeschwindigkeit_MAX: Geschwindigkeitsgrenze, ab der das Fahrzeug als „anfahrend/fahrend/anhaltend“ registriert wird; Fahrgeschwindigkeit_MIN: Geschwindigkeitsgrenze, ab der das Fahrzeug als „haltend“ registriert wird

Abbildung 4.7: SC mit vereinfacht dargestellten Transitionen (Aufgrund der Übersichtlichkeit ist die Notbremse in Abbildung 4.6 vernachlässigt)

Eine Übersicht auf der Basis von Verhaltensdiagrammen wie in Abbildung 4.7 hilft, die einzelnen Aktivitäten oder Zustände in der jeweiligen Schicht zu verknüpfen und funktionsübergreifende Zusammenhänge zu erkennen. Diese Zusammenhänge sind aus der übergeordneten Schicht $(n-1)$ oder in der eigenen Schicht (n) klar definiert. Auf der Grundlage der Schnittstellen der AD kann ebenfalls eine Funktionsarchitektur in der Form eines Strukturdiagrammes (automatisiert) erstellt werden. Hierbei ist es notwendig, die Abhängigkeiten zwischen den Funktionen frühzeitig zu erkennen, ganz unabhängig von den architektonischen Entwürfen [VF13, Man10]. Insbesondere in der Automobilindustrie sind hohe Abhängigkeiten von traditionell isolierten Funktionseinheiten eine Quelle für fehlerhaftes und mangelhaftes Verhalten [VF13, Ben10, Bro06, PBKS07].

Unter Berücksichtigung der Lösungsneutralität von Ebene B werden in Abbildung 4.6 erste thematisch zusammenhängende Funktionen in M-Funktionsgruppen gruppiert. Unter anderem wird die Aktion `Anhaltewunsch ermitteln` der Funktionsgruppe (Partition) `Bremsen Management` zugewiesen und die Aktion `Bremskraft ermitteln` der Funktionsgruppe `Physikalische Fahrzeugzustandsermittlung`. Die Gruppierung von Aktionen erleichtert die Identifizierung von Funktionen, die gleiche oder ähnliche Merkmale aufweisen. Auf Basis der gruppierten Aktionen kann (automatisiert) eine funktionale Systemstruktur der Funktion `Anhalten` erstellt werden. Abbildung 4.8 illustriert ein (automatisiert) erstelltes IBD der Ebene B in der Dekompositionsschicht Fahrzeug. In Ab-



EnFahrzeugAb: Abgeführte Energie Fahrzeug; EnFahrzeugZu: Zugeführte Energie Fahrzeug;
EnSpeicherAb: Abgeführte Energie Speicher; EnSpeicherZu: Zugeführte Energie Speicher;

Abbildung 4.8: IBD der Schicht Fahrzeug in der Ebene B der Sicht der Funktion „Anhalten“ (bidirektionale Kommunikation mit dem `Längsdynamik Management` aus Gründen der Übersichtlichkeit zum Teil ausgeblendet)

bildung 4.8 sind alle Schnittstellen der Funktionsgruppen auf einen Blick ersichtlich. Jede Funktion besitzt eine solche funktionsorientierte Sicht auf die Systemstruktur. Werden die Sichten zusammengeführt, ergibt dies ein komplettes Funktionsnetz der `Funktionsgruppen`, die Funktionsarchitektur. Anhand der Funktionsarchitektur können eng verzahnte `Funktionsgruppen` leichter identifiziert werden. Dies ermöglicht erste Architekturentscheidungen für den weiteren Verlauf (Ebene C) und Dekompositionsschichten $(n+1)$ von SMArDT. Die Funktionsarchitektur ist ein wichtiges Element von SMArDT und ist nicht frei wählbar, sondern wird in Rücksprache mit dem Systems Engineering entwickelt. Das Systems Engineering dient als überblickende Einheit und ermöglicht eine Methode, wie SMArDT auch auf bereits bestehende Architekturen anzuwenden. Theorie und Praxis 4.3 stellt einen möglichen Ansatz für die Einführung von SMArDT in existierende Funktionsarchitekturen dar.

Theorie und Praxis 4.3 (Funktionsgruppen - Greenfield versus Brownfield)**.** *Der hier verwendete Ansatz geht von einer Entwicklung auf der „grünen Wiese“*

(engl. „greenfield“) aus. Allerdings liegen bei großen und komplexen Systemen in der Regel „Brownfield“-Systeme und Strukturen vor [HJ08]. Diese Systeme beinhalten und interagieren mit einer Reihe anderer existierender Systeme [HJ08]. Ein möglicher Ansatz, einem Brownfield System mit SMArDT iterativ zu verbessern, ist:

1. *Schritt Die aktuelle Dekompositionsstruktur des Systems als Ausgangszustand initial übernehmen.*
2. *Schritt Die Funktionen und Subfunktionen der einzelnen (Sub-)Systeme mittels* ***Funktionsprinzips*** *abbilden und gruppieren.*
3. *Schritt Die Funktionsarchitektur evaluieren und gegebenenfalls* ***Funktionsprinzipien*** *verbessern und neu gruppieren.*
4. *Schritt (Automatisiert) prüfen, inwiefern die aktuelle funktionale Systemarchitektur und die konzeptionelle Systemarchitektur abweichen.*
5. *Schritt Die Systemarchitekturen zusammenführen und optimieren.*

Die Schritte können iterativ beliebig oft durchgeführt werden und erleichtern einem Brownfield System den Übergang von einer komponentenorientierten zu einer funktionsorientierten Entwicklung.

Die Art der Gruppierung ist von SMArDT nicht vorgegeben, so können diese `Funktionsgruppen` unter anderem funktional oder auch organisatorisch zugeordnet werden. Funktional im Sinne der Funktionalität die ein System erfüllt und organisatorisch im Sinne der Organisationseinheit die ein System entwickelt oder verantwortet. Theorie und Praxis 4.4 gibt Erfahrungen mit Gruppierung der `Funktionsgruppen` wieder.

Theorie und Praxis 4.4 (Funktionsgruppen - funktional oder organisatorisch)**.** *Bei der Erarbeitung der* ***Funktionsgruppen*** *auf Ebene B wird empfohlen, diese rein funktional und möglichst unabhängig von der vorhandenen System- und Organisationsstruktur zu betrachten. Es wird deswegen empfohlen, da sich ein etabliertes (abstraktes) Funktionsprinzip seltener ändert als die System- und Organisationsstruktur. Ein Beispiel aus einer anderen Domäne ist das System Smartphone. Die Technik von Smartphones ändert sich fortlaufend, doch die Grundfunktionen wie Telefonieren, Surfen im Internet, Fotografieren und Applikationen installieren ändern sich kaum bis gar nicht. Die Organisationsstruktur ist ebenfalls von Entscheidungen des Managements abhängig und kann sich stetig ändern. Besonders in den unteren Dekompositionsschichten wie $n+3, n+4, ...$ schafft eine eindeutig definierte Funktionsarchitektur Klarheit in Bezug auf andere (Sub-)Systemen und deren Aufgaben. Die Funktionsarchitektur kann und wird als Kommunikationsgrundlage innerhalb der Organisation genutzt [Stö17]. Nach dem initialen Aufsetzen der* ***Funktionsprinzipien*** *ist ein Abgleich wie in Theorie und Praxis 4.3 sinnvoll.*

Die aus der Spezifikation der Ebene B abgeleiteten Artefakte (B') fokussieren die Funktion, ohne die Details der Produktimplementierung zu beinhalten (vgl. Abbildung 4.3). Ein Beispiel für ein solches Element sind der Testfallentwurf und die Testfallspezifikationen. Entstandene Anforderungen und Element werden an die nächste Ebene, Ebene C, weitergegeben.

4.2.3 Ebene C, die technische Lösung

Zweck: Der Zweck der Ebene C ist es zu entscheiden, wo und wie eine Funktion umgesetzt wird. Es steht also die Entwicklung der technischen Lösung oder auch des technischen Konzepts im Vordergrund. Zu diesem Zweck werden die abstrakten Elemente aus Ebene B auf konkrete Elemente abgebildet. Dieser Vorgang liefert ausreichend detaillierte Daten und Informationen über das System, dessen Funktionen und deren Architektur, sodass eine Implementierung ermöglicht wird.

Elemente:

- **AD** beschreibt den genauen Ablauf der Funktionsprinzipien.
- **SC** stellt die konkreten Zustände des betrachteten Systems beziehungsweise der technischen Funktionen und den konkreten Signalen dar.
- **SD** spezifiziert den Ablauf eines Anwendungsfalls mit dem Fokus auf Interaktionen, konkreten Signalen und Timings.
- **BDD** illustriert Zusammenhänge der verwendeten technischen Funktionsgruppen.
- **IBD** beschreibt die Schnittstellen der Systemelemente und deren Funktionen mit konkreten Signalen.
- **Textuelle Anforderung** ergänzen die Diagramminformationen.
- **Anforderungsdiagramm** stellen die Zusammenhänge von textuellen Anforderungen dar.

Ergebnisse:

- Die Funktionen, die Schnittstellen, die Signale und deren Entitäten sind technisch, wie sie umgesetzt werden, beschrieben.
- Die Funktionen sind konkreten Systemelementen wie Applikationssoftwarebausteine, Basissoftwarebausteine oder Hardwarebausteine zugeordnet.
- Die Hardware- und Softwarearchitektur des Systems ist verbindlich konzeptioniert.
- Die Funktionsanforderungen sind Verfeinerungen der Anforderungen aus Ebene B und verknüpft.

Die Ebene C konkretisiert die entwickelten `Funktionsgruppen` der Ebene B mit zugeordneten `Subfunktionen` und ergänzt diese optional mit Hardwareanforderungen. Das heißt die Diagramme von Ebene B werden auf Ebene C verfeinert und weitere Details hinzugefügt, wobei die Konsistenz von Ebene B und C nicht verletzt wird. Die Spezifikation der `Funktionsgruppen` auf Ebene C kann einer (externen) Arbeitsgruppe[9] übergeben

[9]Eine Arbeitsgruppe kann aus einer bis endlich vielen Personen bestehen.

werden. Diese Arbeitsgruppe kann entscheiden, ob die Funktionen in einer Dekompositionsschicht $n+1$ weiter verfeinert und dekomponiert werden müssen. Ist dies nicht nötig, können die `Funktionsbausteine` mittels einer Hardware- oder einer Softwarelösung realisiert werden. Im Falle einer Softwarelösung werden die `Applikationssoftwarebausteine` spezifiziert. Im Falle einer Hardwarelösung werden die `Basissoftwarebausteine`, `Hardwarebausteine` und falls notwendig `Applikationssoftwarebausteine` spezifiziert (vgl. Abbildung 4.4). Zum Beispiel kann die Temperatur in einem Kabel über einen zusätzlichen Temperatursensor gemessen werden (Hardwarelösung) oder anhand eines Widerstandswertes des Kabels errechnet werden (Softwarelösung). Alle Bestandteile einer bestimmten `Funktionsgruppe` werden von einer Arbeitsgruppe mit der nötigen Expertise verantwortet und auf Basis der Anforderungen aus anderen Schichten und der Ebene A und B entwickelt. Abbildung 4.9 illustriert ein IBD der Ebene C in der Dekompositionsschicht Fahrzeug.

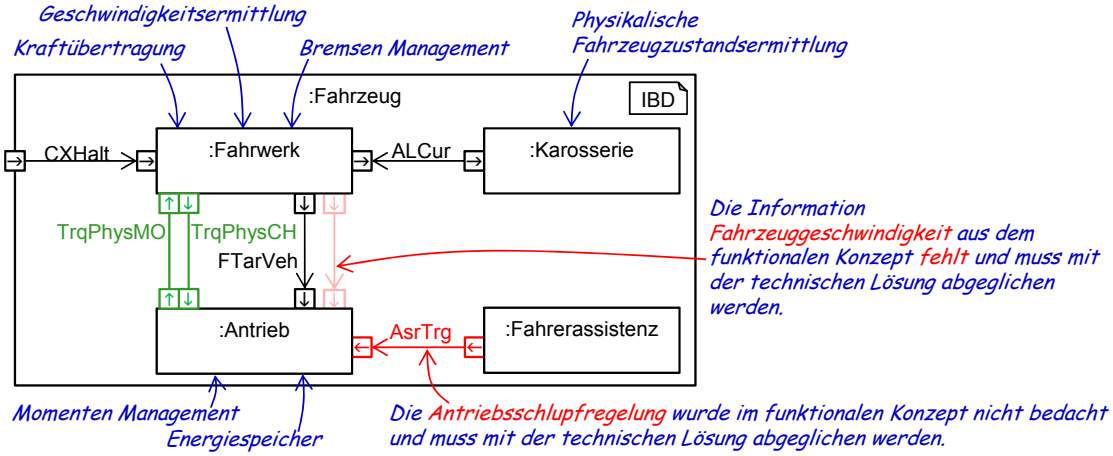


Abbildung 4.9: IBD der Schicht Fahrzeug in der Ebene C von „Anhalten“

In Abbildung 4.9 sind alle Schnittstellen der Systemelemente (hierarchische Dekomposition) auf einen Blick ersichtlich. Die auf Ebene B entwickelten Funktionsgruppen sind den Systemelementen (Blöcke) zugeordnet. Diese Systemelemente und verantwortliche Arbeitsgruppen übernehmen die mit den Funktionsgruppen verknüpften Funktionen und Anforderungen. Die abstrakten Schnittstellen der Ebene B sind den konkreten Schnittstellen der Ebene C zugeordnet (Signal-Mapping).

Für die Funktionsmodellierung der einzelnen Blöcke werden die Verhaltensdiagramme AD, SC und SD und Strukturdiagramme wie das BDD und IBD für Systembeschreibungen eingesetzt. Mittels Modellprüfung lassen sich Inkonsistenzen wie widersprüchlichen Anforderungen der Modelle der Ebene B und der Ebene C (automatisiert) erkennen [DGH+19]. Im IBD in Abbildung 4.9 fehlt zum Beispiel ein passendes Signal der `Fahrzeuggeschwindigkeit` aus Abbildung 4.8. Außerdem wurde in dem funktionalen Konzept der Ebene B die `Antriebsschlupfregelung`, die für eine technische Umsetzung der Funktion nötig ist,

nicht beachtet. Solche Inkonsistenzen können automatisiert erkannt werden und sollten in der Entwicklung so früh wie möglich angegangen werden.

Die realen Signalbezeichnungen, Wertebereiche und Elemente entsprechen auf der Ebene C der realen Umsetzung. Infolgedessen können abgeleitete Testfälle C' das realisierte Produkt prüfen. Ebenfalls kann auf Basis der Ebene C eine Hardware- und Softwarearchitektur abgeleitet werden.

4.2.4 Ebene D, die Realisierung

Zweck: Die Realisierung auf Ebene D dient dem Zweck, genügend Daten zu den verschiedenen Lösungen aus Ebene C zu generieren, die geeignetste auszuwählen und dann zu realisieren. Hierbei werden alle relevanten Hardware- und Software-`Funktionsblöcke` aus Ebene C berücksichtigt.

Elemente:

- **Textuelle Anforderung** dient gegebenenfalls zur Beschreibung der Entscheidungsfindung.
- **Anforderungsdiagramm** dient zum Verständnis der Zusammenhänge von textuellen Anforderungen.
- **Disziplinenspezifisches Modell** wie beispielsweise UML/P Modelle, Matlab/Simulink Modelle oder CAD Modelle

Ergebnisse:

- Die dokumentierte Ergebnisfindung.
- Die Auslegung von Hardwareteilen.
- Die Analyseergebnisse für die Realisierung.

Die Ebene D beinhaltet die Realisierung des Konzepts. Die Elemente der Ebene D dienen unter anderem der Generierung von Quellcode und weiteren Simulationen. Hierzu zählen Softwareelemente (UML/P Modelle, Java Code) E/E-Elemente für elektronische Regelsysteme (Matlab/Simulink Modelle) und Hardwareelemente (CAD Modelle). Neben der Hardware und Software Dokumentation werden spätestens hier eine Funktions- und eine Softwarearchitektur erstellt. Wird in der betrachteten Dekompositionsschicht keine Hardware- oder Software-Realisierung benötigt, kann es passieren, dass Ebene D keine Elemente beinhaltet (Weiteres in Abschnitt 4.3). Die Ebene D steht nicht im Fokus dieser Arbeit und wird im weiteren Verlauf nicht weiter detailliert.

Die Funktion Anhalten aus der Sicht des Systems Fahrzeug

Ist die Funktion aus der Sicht der Schicht Fahrzeug spezifiziert, wurde die linke Seite des „Vs“ (vgl. Abbildung 4.3) von SMArDT mindestens einmal durchlaufen. Die Elemente der Ebene A sind SysML Modelle mit textuellen Anforderungen und risikomindernden Maßnahmen für sicherheitsrelevante Funktionen des Systems (Funktionale Sicherheit) nach ISO 26262 (vgl. Unterabschnitt 2.2.2). Die Elemente der Ebene B sind neben den Verhaltensmodellen, den textuellen Anforderungen und der funktionalen Sicherheit auch generierte Strukturmodelle (IBDs). Die Elemente der Ebene C sind konkrete Struktur- und Verhaltensmodelle mit Anforderungen der technischen Lösung, auf deren Basis eine Hardware- und Softwarearchitektur abgeleitet werden kann. Zu guter Letzt werden in Ebene D Elemente erzeugt, die als Implementierungs- oder Simulationsmodell dienen und auf deren Basis die Hardware und Software erzeugt werden können.

Mit den Elementen aus A bis D können zudem weitere Entwicklungen vorangetrieben und Elemente für die V&V abgeleitet werden.

4.3 Hierarchische Dekomposition und funktionale Abstraktion

In diesem Abschnitt wird die Kombination der Dekompositionsschichten aus Abschnitt 4.1 und die Abstraktionsebenen aus Abschnitt 4.2 erläutert. Zu diesem Zweck wird die Abstraktion anhand der Funktion `Anhalten` aus Abschnitt 4.2 aufgegriffen und weiter konkretisiert. Für die technischen und physikalischen Daten wurde eine Rekuperationsbremse aus [RNB12] gewählt. Abbildung 4.10 gibt eine Übersicht über die Kombination der hierarchischen Dekomposition und der funktionalen Abstraktion.

Die Basis für die weitere Spezifikation ist das V-Modell der Schicht n, dem Fahrzeug aus Abschnitt 4.2. Die Funktionsgruppen und Subfunktionen der Ebene B und die Subsysteme mit ihren Schnittstellen der Ebene C dienen dem verzielten System als Ausgangsbasis. Verzielt bedeutet, dass die Funktionen und Anforderungen, die Funktionsgruppen, Subfunktionen und Subsysteme aus der Schicht $n+m$ den Systemen beziehungsweise Arbeitsgruppen von Schicht $n+m+1$, nach Absprache, zugewiesen werden. Diese müssen die Anforderungen erfüllen und können diese weiter spezifizieren (vgl. Abbildung 4.4).

In einem Subsystem $n+1$, in diesem Fall dem `Antrieb`, können die Anforderungen des Systems/Fahrzeugs als „Kunden“-Anforderungen gesehen werden. Die Anforderungen an das `Momenten Management` umfassen dabei ebenfalls die modellierten Funktionen `Geradlinige Ist-Kraft ermitteln`, `Längskräfte anpassen` und deren Schnittstellen (vgl. Abbildung 4.10). Der Antrieb als Subsystem spezifiziert und entwickelt diese Funktionen anhand der funktionalen Abstraktion weiter, um die Anforderungen der anforderungsstellenden Systeme zu erfüllen.

Im Gegensatz zu Abbildung 2.8 ist zu sehen, dass eindeutig immer von Ebene A nach Ebene D entwickelt werden muss und kein diagonaler Weg der Entwicklung genommen werden kann. Eine oft genutzte Interpretation des V-Modells in der Automobilentwicklung kombiniert die Schritte der Abstraktion und Dekomposition, da sie auf den ersten Blick effizienter erscheint. Die Trennung der Abstraktions- und Dekompositionsschritte in

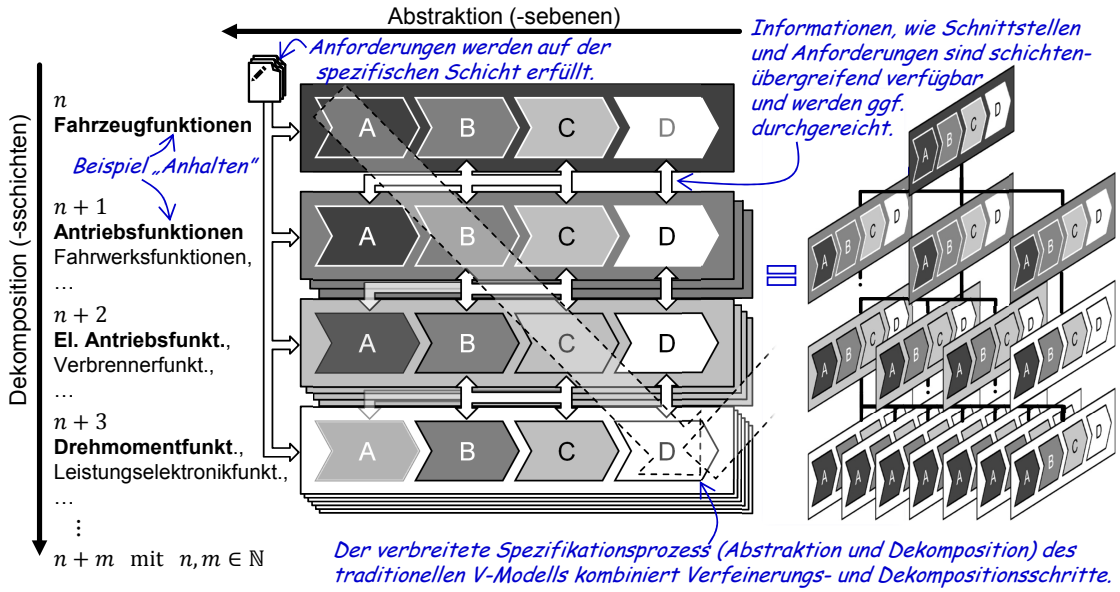


Abbildung 4.10: Übersicht über die hierarchische Dekomposition (vgl. Abbildung 4.1) und der funktionale Abstraktion (vgl. Abschnitt 4.2) am Beispiel von `Anhalten`

dieser Arbeit dienen nicht nur zur Komplexitätsreduzierung, sondern auch dafür, die Elemente für Modifikationen oder technische Ersetzungen wieder auseinanderziehen zu können. Bei der diagonalen Entwicklung ist der Aufwand einer Anpassung beziehungsweise Überarbeitung eines oder mehrerer Aspekte eines Systems hoch, da immer das komplette System betrachtet werden muss. Diesem kombinierten diagonalen Entwicklungsprozess wird auch in Abschnitt 2.4 nicht explizit vorgebeugt. Die Entwicklung einer Matrix wie in Abbildung 4.10 ermöglicht die spezifische, iterative und agile Modifikation eines Subsystems oder eines Schrittes, ohne das komplette System betrachten zu müssen. Diese Agilität wird begünstigt durch die eindeutigen Kommunikationsschnittstellen der Kacheln der Matrix und der später entwickelten Kommunikationsarchitektur, die sich deshalb für die Entwicklung in agilen Projektteams wie beispielsweise SAFe [KL19] eignen. Theorie und Praxis 4.5 verdeutlicht den Bedarf einer agilen Überarbeitung von Entwicklungselemente.

Theorie und Praxis 4.5 (Änderungen von Anforderungen während der Entwicklung)**.** *In CPS entwickelnden Industrien liegt ein Hautproblem des Software und Systems Engineering darin, dass sich die Anforderungen während der Systementwicklung ändern [Pre03]. Zum Beispiel können Zulieferer nicht wie gewünscht liefern oder Funktionalitäten werden kurzfristig geändert. Aus diesem Grund müssen das System und die Entwicklungsprozesse für evolutionäre und iterative Prozesse ausgelegt sein. Hierbei gilt, agile und schnelle Systemänderungen sind dann möglich, wenn Entwicklungselemente reibungslos iterativ überarbeitet werden können.*

In Abbildung 4.10 wird neben der Darstellung der Trennung zudem aufgeführt, dass aufgrund der Anforderungen aus Schicht $n+2$ prinizipiell auf Ebene A der Schicht $n+3$ verzichtet werden kann. Dies sollte allerdings genau geprüft und dokumentiert werden. Abbildung 4.11 stellt das UCD der Ebene A im Antrieb für das Beispiel `Längskräfte anpassen` dar.

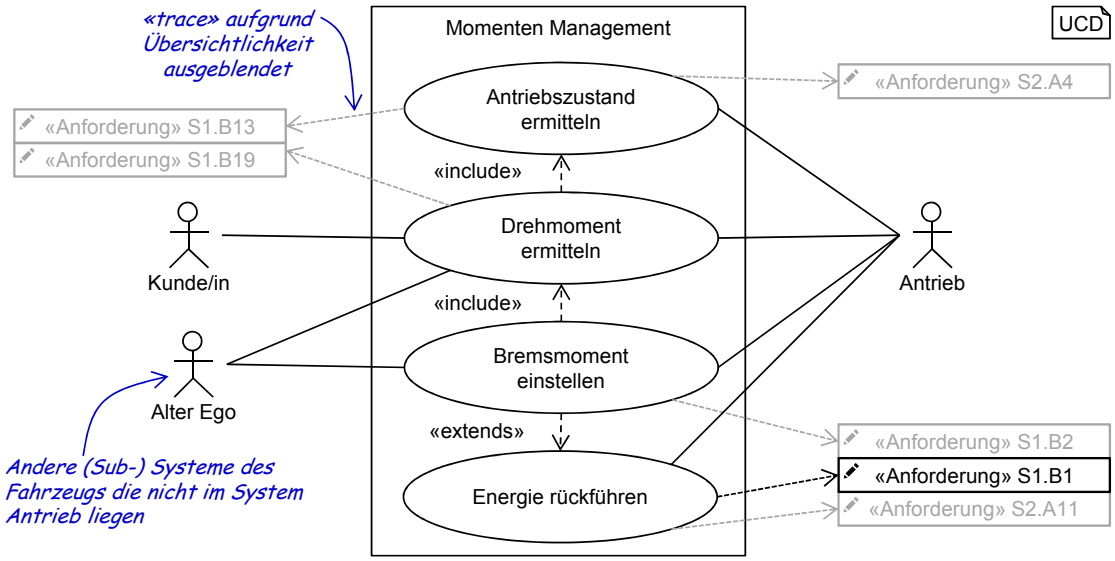


Abbildung 4.11: UCD der Schicht Antrieb in der Ebene A am Beispiel `Anhalten` beziehungsweise `Längskräfte anpassen` (graue Anforderungen sind exemplarisch eingefügt.)

In Abbildung 4.11 werden relevante textuelle Anforderungen den Anwendungsfällen zugeteilt (≪`Anforderung`≫ `S1.B1`) und weitere Anforderungen für das eigene System können hinzugefügt werden (≪`Anforderung`≫ `S2.A4`). Anforderungen, die sich aus Normen der Sicherheit und anderen technischen Normen ergeben, sind zwar auch schon den oberen Schichten vorhanden, wie beispielsweise Normen für Fahrzeugzulassungen, werden aber erst in tieferen Dekompositionsschichten und Subsystemen berücksichtigt und erfüllt. Die Anforderungen aus Normen liegen vorwiegend textuell vor. Die textuellen Anforderungen werden im Modell modelliert oder ergänzend als textuelle Anforderungen den Diagrammen beigefügt.

Theorie und Praxis 4.6 (Anforderungsmanagement in der Automobilindustrie)**.** *Das traditionelle Anforderungsmanagement in der Automobilindustrie basiert auf textuellen Anforderungen und Dokumenten (vgl. Abschnitt 2.5). Die Spezifikationsmethode für Anforderungen von SMArDT muss sich deshalb in die historisch gewachsenen und bereits etablierten textuellen Anforderungsmanagementprozesse und Tools, wie zum Beispiel dem Anforderungsmanagementtool Doors [IBM19], einfügen. Eine Möglichkeit ist die jeweiligen modellbasierten Informationen vorübergehend als Text zu exportieren, bis das Modell ausgereift ist und sich gegenüber textuellen Anforderungsmanagementprozessen*

etabliert hat. Auf keinen Fall sollten Modellinformationen in Text übersetzt und danach erneut modelliert werden, da dieses Vorgehen einen vermeidbaren Mehraufwand mit sich bringt und zu Redundanz und Informationsverlust führen kann.

Im nächsten Schritt werden die Abstraktionsebenen wie in Abschnitt 4.2 durchlaufen. Abbildung 4.12 stellt das AD der Ebene B (Antrieb) für das Beispiel `Längskräfte anpassen` dar. Die Funktion `Längskräfte anpassen` wird in Abbildung 4.12 in zwei Ak-

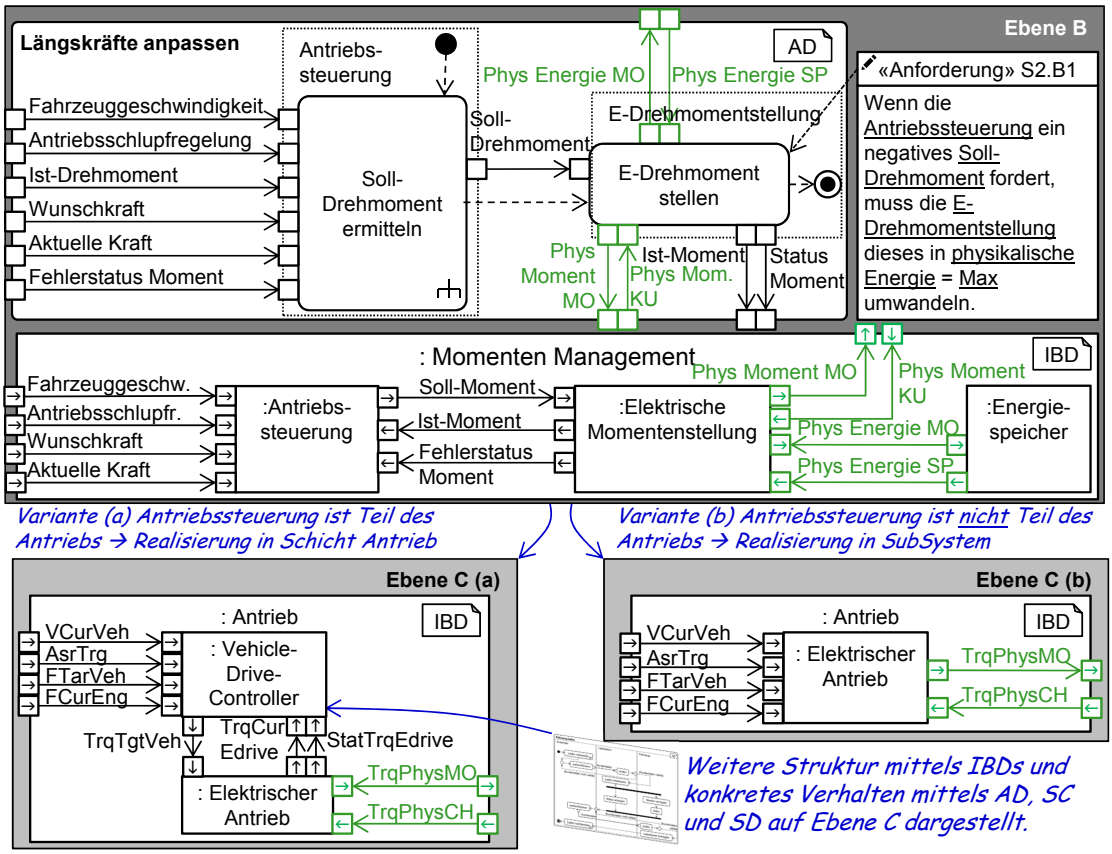


Abbildung 4.12: AD und IBD der Schicht Antrieb in der Ebene B und zwei IBD-Varianten der Ebene C am Beispiel `Längskräfte anpassen` (Variante Ebene C (a) mit einer steuernden Architekturschicht des Antrieb und Variante Ebene C (b) ohne steuernden Architekturschicht des Antriebs)

tionen, `Soll-Moment ermitteln` und `Drehmoment stellen` unterteilt und mit textuellen Anforderungen versehen (≪`Anforderung`≫ `S2.B1`). Auf Basis dieses ADs kann Ebene B das IBD `Momenten Management` mittels die Partitionen des ADs generiert werden.

Im Schritt von Ebene B nach Ebene C werden die Ebene B-Funktionsgruppen `Antriebssteuerung` und `Drehmomentstellung` auf die konkreten Systeme (Ebene C Blöcke) verzielt (vgl. Abbildung 4.4). Der Block `Vehicle-Drive-Controller` auf Ebene C (a) setzt die Steuerung oder Koordination der (verschiedenen) Antriebe des Fahrzeugs (hier

`Elektrischer Antrieb`) um. Diese Koordination kann mittels reiner Software umgesetzt werden. Die resultierenden Subfunktionen werden deshalb der S-Funktionsgruppe zugeordnet. In diesem Fall sind die realisierten Applikationssoftwarebausteine an keine konkreten Komponenten gebunden. Anhand der Softwarearchitektur der S-Funktionsgruppe und weiteren Eigenschaften, wie Performance-, Leistungs- und Speicheranforderungen, wird zentralisiert entschieden, wie Softwarefunktionen (schichtenübergreifend) verteilt werden. In diesem Beispiel übernimmt der Antrieb die Spezifikation der resultierenden Subfunktionen und Applikationssoftwarebausteine. Der Block mit den Funktionen des `Elektrischen Antriebs` wird hier an ein Subsystem des Antriebs vergeben (vgl. Abbildung 4.4). Eine andere Möglichkeit bietet Ebene *C* (b), welche das System Antrieb, die Steuerung, Koordination und die Funktionen des `Elektrischen Antriebs` komplett an ein Subsystem verzielt. In diesem Szenario werden sowohl die S-Funktionsgruppen als auch die M-Funktionsgruppe dem Subsystem elektrischer Antrieb übergeben. In diesem Fall nimmt der Antrieb keine weitere Spezifikation und Realisierung der Applikationsbausteine für die Funktion `Längskräfte anpassen` vor.

Theorie und Praxis 4.7 (Architekturentscheidungen)**.** *Bei komplexen Systemen wird empfohlen, eine Schichtenarchitektur anzustreben [BSH07]. Diese Schichtenarchitektur mit ihren Systemen und Subsystemen sind in dieser Arbeit hierarchisch strukturiert (vgl. Abschnitt 4.1). Es ist keine fest vorgegebene Anzahl an Schichten vorgegebenen, es werden keine festgelegten Aufgaben an eine spezifische Schicht verteilt und der Dekompositionsbaum des Systems lässt auch Asymmetrien zu (vgl. Abbildung 4.13). Mithilfe der Asymmetrien können die unterschiedlichen Subsysteme mit einer individuellen Anzahl von Schichten beschrieben werden.*

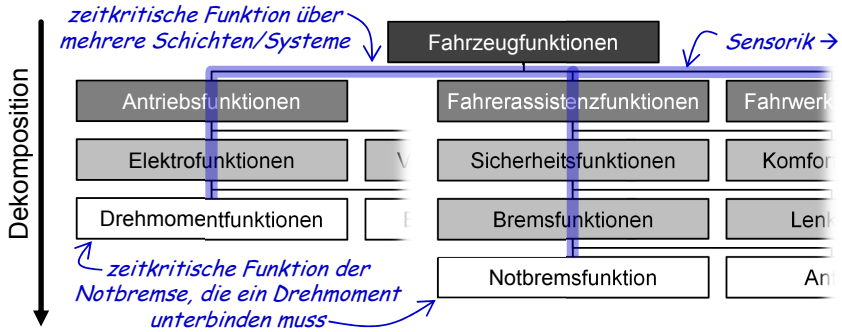


Abbildung 4.13: Verteilung der Funktionalität Notbremse über die Systeme/Dekompositionsschichten

In Abbildung 4.13 ist ein Ausschnitt eines Dekompositionsbaums abgebildet. Wie in Abschnitt 4.1 beschrieben, wird die Hierarchie des Systems auch für die Kommunikation der Architekturelemente angewendet und grenzt sich somit von dem Ansatz in Abschnitt 2.4 ab. Jedes System und jedes (Sub-) System bietet eine spezifische Funktionalität, die eine direkt darüber liegendes System nutzen kann und ist angelehnt an die Kommunikation des OSI-Modells [Zim80]. Die eigene Funktionalität des (Sub-) Systems ist intrinsisch oder

bedient sich aus den unmittelbar darunterliegenden Subsystemen entsprechend den Ästen des Baumes. Eine „horizontale„ Kommunikation der Subsysteme innerhalb einer Schicht ist folglich nicht erlaubt, was einem Verbot von Sprunganweisung („go to statements“) in der Informatik entspricht. In Abbildung 4.13 können Antriebsfunktionen demnach nur mit den Fahrerassistenzfunktionen über die Fahrzeugfunktionen kommunizieren. Die Strukturierung der Funktionalitäten ist initial aufwendiger, aber liefert im Nachhinein eine höhere Flexibilität, höhere Wiederverwendbarkeit und vereinfacht die Integration, da Abhängigkeiten und Komplexität reduziert werden (vgl. Abschnitt 4.1). In Abbildung 4.14 spricht dieser Aspekt für die Wahl der Variante `Ebene C (a)`.

Erfahrungsgemäß wird eine direkte Kommunikation der Subsysteme (unter den untersten Schichten) häufig mit zeitkritischen Funktionalitäten begründet, in denen jede Millisekunde zählt. In Abbildung 4.13 ist beispielhaft die Aufteilung der zeitkritischen Sicherheitsfunktion des Notbremsassistenten über die Dekompositionsschichten hinweg illustriert. Bei einer Geschwindigkeit von 130 $km/h \approx 36\ m/s$ *beträgt der Bremsweg einer Gefahrenbremsung circa* 85 $Meter$. *Folglich kann eine Verzögerung der Funktionalität Notbremsassistenten zu Verletzten oder auch zu Toten führen. Dennoch sollte in Abbildung 4.13 nicht ad hoc die Notbremsfunktion direkt mit den Drehmomentfunktionen kommunizieren, da es zu ungewollten Abhängigkeiten und Querwirkungen kommen kann, wenn eine Vielzahl von Funktionen auf diese Weise kommunizieren. Infolgedessen leidet die Flexibilität des Systems und die Wiederverwendbarkeit von Subsystemen, die Integration und eine nachhaltige Architektur wird erschwert.*

Mit der in Abbildung 4.13 und Abbildung 4.1 funktionalen Systemdekomposition werden systemübergreifende Funktionalitäten offengelegt. Auf dieser Basis können Systemarchitekturentscheidungen intellektuell verständlich aufbereitet und/oder maschinell identifiziert und bewertet werden. Da die Abstraktion und Dekomposition in SMArDT getrennt sind, lassen sich Funktionsprinzipien der Funktionalitäten ohne unnötige Abhängigkeiten mit der Dekomposition umziehen und wiederverwenden (vgl. Theorie und Praxis 4.5).

In Abbildung 4.14 wird von Variante Ebene C a) aus Abbildung 4.12 ausgegangen und das Drehmoment auf den elektrischen Antrieb gestellt. Es fällt auf, dass das `Soll-Moment` hier abermals überprüft wird. Dieser Fall tritt ein, wenn funktionale Sicherheitsanforderungen, einerseits vom System und andererseits von Subsystemen und Komponenten erfüllt werden müssen. Das übersichtliche Funktionsprinzip der Ebene B stellt derartige Fälle dar und ermöglicht eine Absprache der Systeme. Nachdem das Funktionsprinzip mit den aus ihm resultierenden Subfunktionen spezifiziert ist, werden die Funktionsgruppen `E-Moment-Leistung berechnen` und `Fehler melden` dem `E-Drive-Controller` der Ebene C übergeben. In dem hier verwendeten Beispiel existiert eine konkrete `E-Drive-Controller`-Hardwarekomponente, die von dem System E-Antrieb entwickelt und realisiert wird.

Sind die Funktionsbausteine in der Ebene D umgesetzt, können diese in die Komponente integriert („deployed“) werden (vgl. Abbildung 4.4). Die Realisierung der Funktionsbausteine und der Komponenten erfüllt dann den zugeordneten Anforderungen (engl. „satisfied“). Abbildung 4.15 gibt eine Übersicht über die in diesem Kapitel entwickelten textuellen Anforderungen in einem Anforderungsdiagramm.

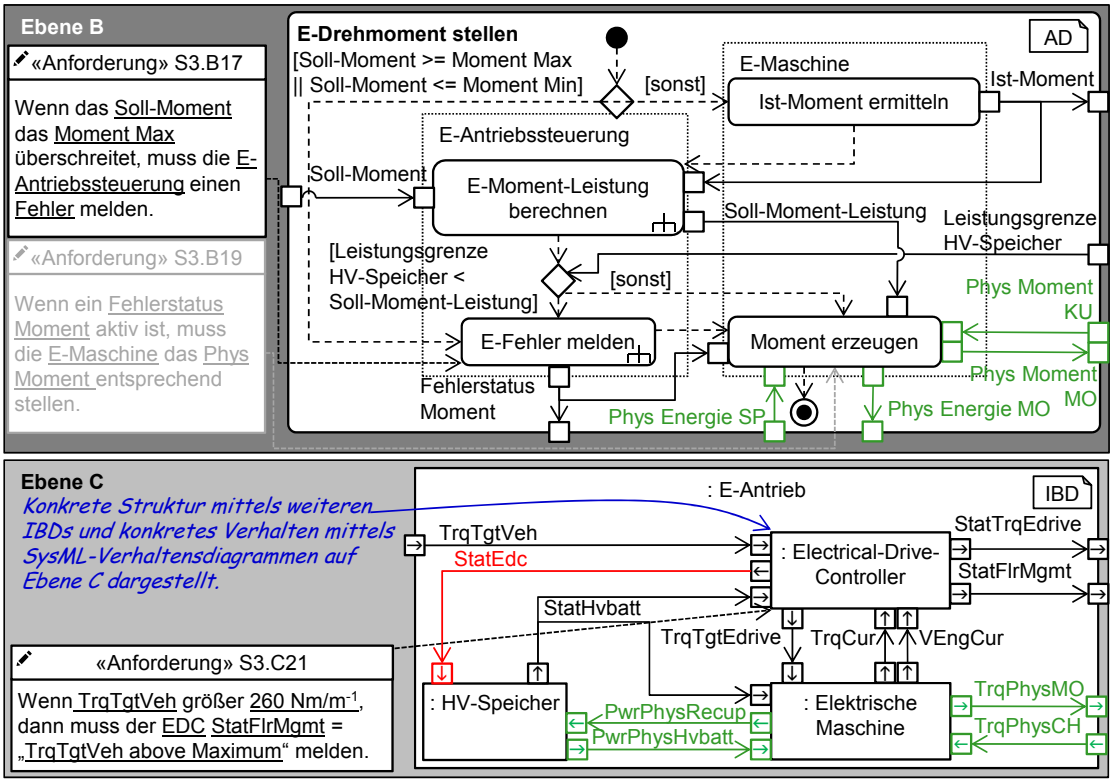


Abbildung 4.14: AD der Schicht E-Antrieb in der Ebene *B* und das IBD der Ebene *C* am Beispiel `Drehmoment stellen` (nebensächliche Elemente in Grau)

In Abbildung 4.15 ist die Ableitungsstruktur bis hin zur Erfüllung der textuellen Anforderungen dargestellt. Eine Ableitung einer Anforderung aus einer anderen bestehenden Anforderung wird mittels einer „Derive Requirement Relationship“ (≪`deriveReq`≫) modelliert. Auf diesem Weg kann nachvollzogen werden, welche Anforderung aus welcher übergeordneten Anforderung entstanden ist. Werden alle atomaren Anforderungen von einem Designelement gedeckt können auch die „darüberliegende“ Anforderung als erfüllt, angesehen werden. Im Beispiel in Abbildung 4.15 wird die atomare Anforderung von ≪`Anforderung`≫ `S3.C21` dem Softwarebaustein `SwBErrorMngEDC` über eine `SatisfiedBy`-Beziehung „erfüllt“.

Zusätzlich zu der transparenten durchgängigen Anforderungsdarstellung können mittels SMArDT vollständige Funktionsgruppen der Systeme, die als Funktionsbibliotheken[10] dienen, ausgeleitet werden. Eine *Funktionsbibliothek* beinhaltet als vollständige Funktions-

[10] Die hier genannten Funktionsbibliotheken der Systeme sind nicht zu verwechseln mit den Funktionsbibliotheken von MontiCore in Unterabschnitt 2.3.2. Beide fungieren als Bibliotheken, allerdings unterscheidet sich der Inhalt.

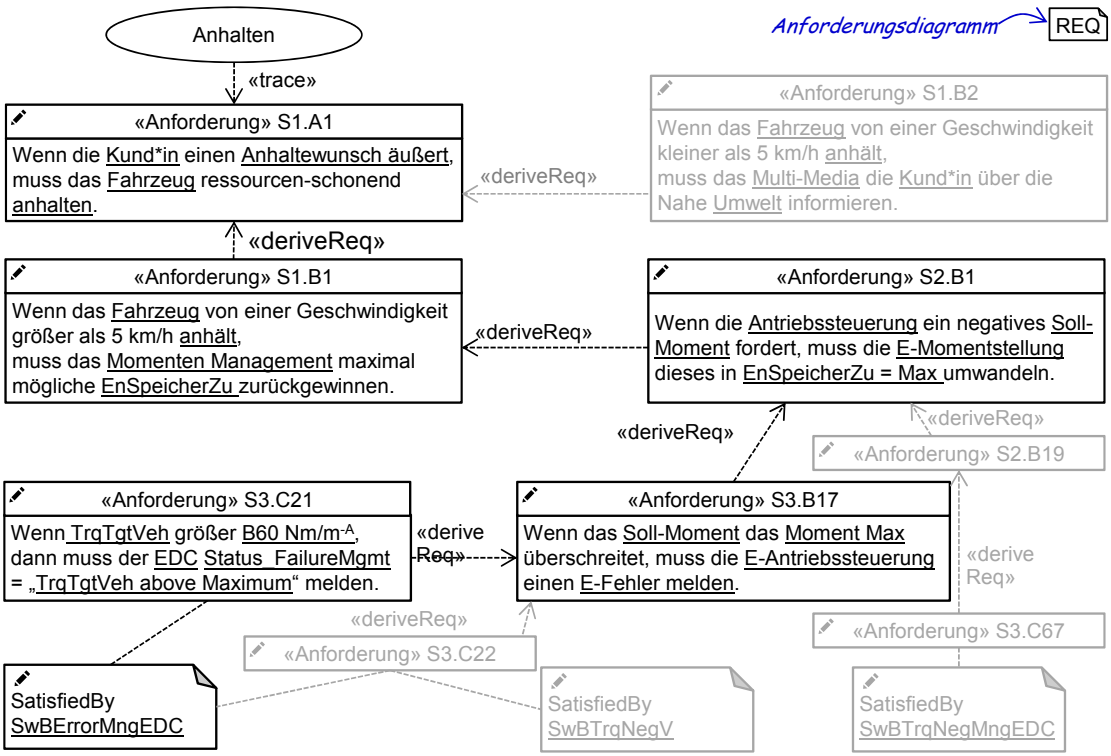


Abbildung 4.15: Anforderungsdiagramm der Schicht E-Antrieb des Beispiels `Anhalten` - `Längskräfte anpassen` - `Drehmoment stellen` - `Fehler melden`

gruppe alle von dem System bereitgestellten Funktionen. Sind diese bereits mit SMArDT entwickelt, können alle zugehörigen Elemente und Referenzen eingesehen werden. Die Funktionsbibliotheken erhöhen die Transparenz über die Darstellung der Abhängigkeitsbeziehungen und dem Bestand von Funktionen eines jeden Systems. Zusätzlich profitieren weitere Entwicklungen davon, dass die Verwendung und Wiederverwendung von Funktionen erleichtert wird (vgl. Abschnitt 4.5). Theorie und Praxis 4.8 teilt Erfahrungen mit existierenden Funktionsbibliotheken in der Einführung von SMArDT.

Theorie und Praxis 4.8 (SMArDT und Funktionsbibliotheken)**.** *In etablierten Systemen wie dem Automobil sind viele Funktionen, Funktionsbausteine und Funktionsbibliotheken bereits vorhanden (vgl. Theorie und Praxis 4.3). Wird SMArDT in einem bestehenden großen Industriekonzern eingeführt, empfiehlt es sich vorhandene Funktionen (Funktionsbibliotheken) zu identifizieren, SMArDT darauf aufzubauen und zu optimieren. Mehr zur Weiterentwicklung von Funktionen mittels SMArDT in Abschnitt 4.5.*

4.4 MBT im Zusammenhang der Abstraktion und Dekomposition mit SMArDT

Wie der Name „Spezifikationsmethode für Anforderungen, Design und *Test*“ andeutet, ist das Testen (von Anforderungen) ein wichtiges Element von SMArDT. Die Spezifikation basiert in SMArDT auf Modellen. Diese Modelle dienen dem Test als Testorakel. Abbildung 4.16 gibt eine Übersicht über die Testfallerstellung und Artefakte in Abhängigkeit der funktionalen Abstraktionsebenen in SMArDT.

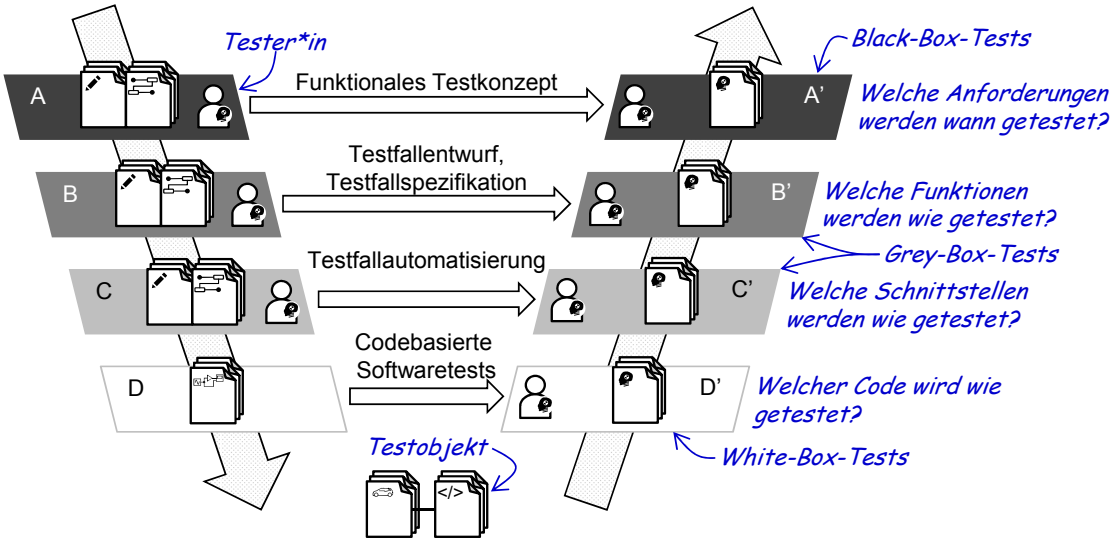


Abbildung 4.16: Übersicht über die funktionalen Abstraktionsebenen, die Beteiligung der Tester*innen, die aus der Spezifikation abgeleiteten Testartefakte und zugeordnete Testebene in SMArDT

Die Testingenieur*innen nutzen nicht nur die abgegebene Spezifikation als Artefakte, sondern sind aktiv in den Spezifikationsprozess und die Anforderungserstellung eingebunden. Hierbei wird die Spezifikation in statischen Tests, wie einem Review, geprüft und freigegeben. Dieser Prozess unterstützt eine Kommunikation zwischen Entwickler*innen und Tester*innen und stellt sicher, dass die Spezifikation und deren Anforderungen testbar ist. Außerdem kann der Beitrag der beteiligten Testgruppen helfen, in der Entwicklung des Systems beziehungsweise des Testobjekts Aspekte der Testbarkeit zu berücksichtigen. Das können zum Beispiel kleinere, leicht prüfbare Einheiten oder Schnittstellen sein.

Ebene A

Aus der Spezifikation der Ebene *A* werden Black-Box-Tests und das funktionale Testkonzept abgeleitet (vgl. Abschnitt 3.3). Da das Testobjekt von Dekompositionsschicht zu Dekompositionsschicht variiert, ergeben sich spezifische Systemtestfälle. In der Schicht Antrieb wird zum Beispiel die Teilfunktion `Längskräfte anpassen` der übergeordneten

Funktion `Anhalten` getestet (vgl. Abbildung 4.11). Das betrachtete System ist der Antrieb mit Schnittstellen zur `Kund*in` und dem Rest des Fahrzeugs, `Alter Ego`. Aus den Anwendungen und textuellen Anforderungen können manuell Testfälle aus Kundensicht und Akzeptanztests des Systems abgeleitet werden. Zusätzlich kann für System- und Integrationstest ein funktionales Testkonzept erstellt werden, wie zum Beispiel die V&V für das gewünschte Produkt beziehungsweise die gestellten Anforderungen. Da funktionale Schnittstellen und Systemgrenzen aus der überordneten Schicht bekannt sind, können nötige Testumgebung ermittelt werden. Zum Beispiel werden die Schnittstellen vom Antrieb gefordert, die `Wunschkraft` (`FTarVeh`), ... , die auf einem Verbund-HIL geprüft werden müssen.

Ebene B

Aus der Spezifikation der Ebene B werden der Testfallentwurf und die Testfallspezifikation in Form von Grey-Box-Tests abgeleitet. Wie auf Ebene A werden die Spezifikationen aktiv in einem Review auf Testbarkeit geprüft, eventuelle Unklarheiten ausgeräumt, gewonnene Erfahrungen aus früheren Projekten in die Spezifikation mit eingebracht und schließlich freigegeben. Mit dem Review wird also nicht nur das Modell validiert, sondern es werden auch Fehler identifiziert und behoben, die in der Vergangenheit aufgetreten sind. Auch Missverständnisse im Vorgängermodell können aufgedeckt und ausgeräumt werden. Im Beispiel der Schicht elektrischer Antrieb der Ebene B aus Abbildung 4.14 wurde die Vorbedingung, dass ein einstellbares `Soll-Moment` über ein Review entdeckt und nachträglich im Modell ergänzt. Auch Entscheidungen wie zum Beispiel `Soll-Moment >= Moment Max || Soll-Moment <= Moment Min` können über Reviews im Vieraugenprinzip[11] validiert werden.

Zusätzlich zur Prüfung der Validität können Vorbedingungen für die Testfälle ermittelt werden. Zum Beispiel kann die Vorbedingung `Herstellen: Soll-Moment == Moment Max` ermittelt werden, um den Grenzbereich der Funktion `E-Drehmoment stellen` zu prüfen. Außerdem können einzelne Schritte des Testfalls (Eingänge, Entscheidungen und Aktionen in Abbildung 4.14) und Messpunkte, wie zum Beispiel das `Ist-Drehmoment`, der `Fehlerstatus-Drehmoment` und weitere Ausgänge aus der Spezifikation entnommen werden. Um die Testfallerstellung zu erleichtern, sind einheitliche Abstraktionsniveaus der Spezifikation und der Tests nötig. Theorie und Praxis 4.9 reflektiert Erfahrungen aus der Praxis mit dem Abstraktionsniveau der Spezifikation und Schlüsselwörtern.

Theorie und Praxis 4.9 (Abstraktionsniveau Ebene B und Schlüsselwörter)**.** *Für eine Testfallspezifikation auf Basis von Schlüsselwörtern ist ein gleiches Abstraktionsniveau der spezifischen Ebene (hier B) und der entsprechenden Schlüsselwörter sinnvoll. Andernfalls lassen sich die einzelnen Testschritte schwer intuitiv oder automatisiert aus den Modellen ableiten und erschwert Rückschlüsse von Testschritten und Testergebnissen auf das Modell ziehen.*

[11] Das Vieraugenprinzip stellt sicher, dass ein Artefakt nicht von der gleichen Person entwickelt und geprüft beziehungsweise validiert und verifiziert wird [Lig09, Wit16].

Ebene C

Aus der Spezifikation der Ebene C werden die Schnittstellen mit den zu integrierenden Subsystemen und Signalen für die Testfallautomatisierung der Grey-Box-Tests abgeleitet. Mithilfe der Schnittstellen der Strukturdiagramme und der Verhaltensdiagramme können zusätzlich Grey-Box-Tests für die umgesetzten Komponenten abgeleitet werden. Wie auf Ebene B wird die Spezifikation aktiv in einem Review auf Testbarkeit geprüft, eventuelle Unklarheiten ausgeräumt, gewonnene Erfahrungen aus früheren Projekten in die Spezifikation mit eingebracht und schließlich freigegeben. Falls die Testfallspezifikation auf Ebene B für Integrationstests erstellt wurden und auf Schlüsselwörtern basieren, können mit den Informationen der Ebene C die Schlüsselwörter für die automatisierte Ausführung befähigt werden. Falls keine Testfallspezifikationen aus der Ebene B existieren, kann ein Testfall aus Ebene C (automatisiert) erzeugt werden[12]. Im Beispiel der Schicht elektrischer Antrieb der Ebene C aus Abbildung 4.14 können beispielsweise die konkreten Signalnamen und deren Wertebereiche der Ebene C für die Testfallautomatisierung der abgeleiteten Schlüsselwörter der Ebene B' verwendet werden. Zusätzlich können auf Basis der Struktur, der Schnittstellen und des Verhaltens die Softwaretests der Steuerungskomponente, in Abbildung 4.14 der `E-Drive-Controller`, als Black-Box oder Grey-Box-Tests abgeleitet werden.

Theorie und Praxis 4.10 (Black-Box-Test)**.** *Black-Box-Tests stellen mit ihrer Sicht die Umgebung des betrachteten Objektes in den Vordergrund. Das heißt allerdings nicht, dass die testende Person nicht wissen sollte, wie das System und deren (Sub-)Systeme funktionieren [KPB02]. Je mehr eine testende Person über das Testobjekt weiß, umso besser kann sie es testen [KPB02]. Den Domänenexpert*innen ist bewusst, dass das Wissen über das Testobjekt und dessen Rolle im Gesamtkontext essenziell für die Absicherung ist [JTH21]. Die sich ergänzende Struktur- und Verhaltensspezifikation von SMArDT und anderer modellbasierter Methoden (vgl. Abschnitt 2.5), unterstützen in diesem Punkt die Tester*innen, da auch übergreifende Themen verständlich in verschiedenen Abstraktionsebenen dargestellt sind. Neben Black-Box-Tests, gilt das Prinzip auch für Grey-Box-Tests und den nicht fokussierten Code, der für White-Box-Tests relevant ist.*

Ebene D

Aus der Spezifikation der Ebene D werden codebasierte Softwaretests beziehungsweise White-Box-Tests abgeleitet. Diese Tests werden häufig von den entwickelnden Personen während der Realisierung durchgeführt. Da es sich bei der Software um formale Artefakte handelt und Tests häufig von den entwickelnden Personen während der Realisierung durchgeführt werden, ist keine unabhängige Testperson für statische Tests vorgegeben. Allerdings wird aufgrund des Vieraugenprinzips dringend empfohlen, dass die entwickelten Artefakte der Ebene D wie auch in den anderen Ebenen von unabhängigen Tester*innen geprüft werden.

[12] Besonders in Schichten nahe der atomaren Funktionsbausteine kann es vorkommen, dass keine Ebene A oder Ebene B benötigt werden. Dies ist dann der Fall, falls in den übergeordneten Schichten, die Struktur und das Verhalten schon ausreichend spezifiziert worden sind.

Absicherung und Integration von Systemen

Die in diesem Abschnitt erläuterte Methode für MBT spiegelt sich auf allen Schichten des Produktes wieder. Abbildung 4.17 stellt die Methode für die Absicherung in den Zusammenhang mit den Teststufen des V-Modells.

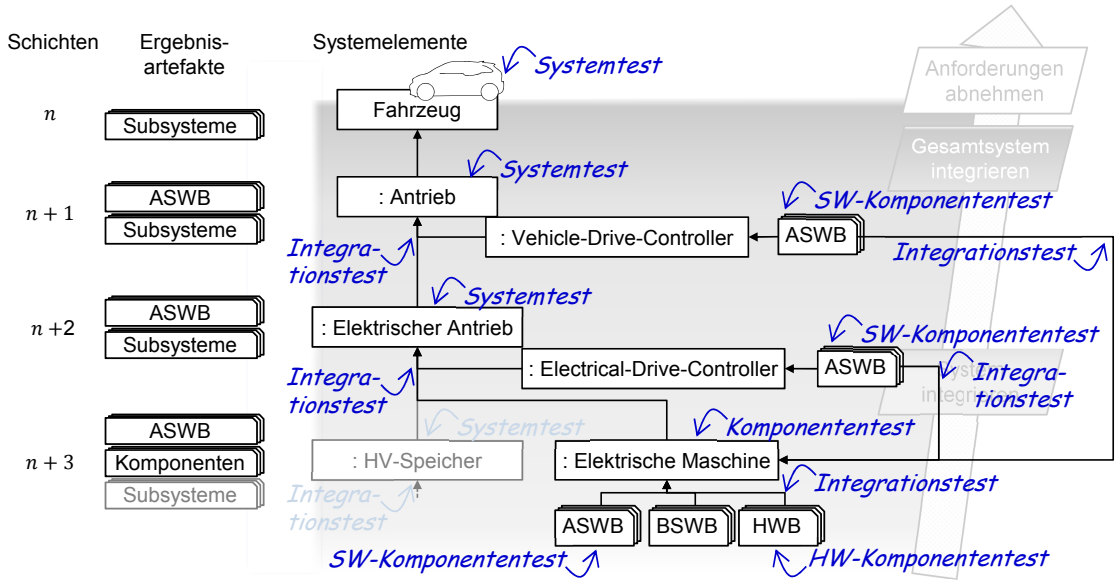


Abbildung 4.17: Übersicht über die Absicherung und Integration der (Sub-)Systeme am Beispiel „Anhalten“ im Zusammenhang mit den Teststufen des V-Modells (matt im Hintergrund)

Die Spezifikationsmodelle dienen den System-, Integrations- und Komponententests als Testorakel. Der Aufwand für die Absicherung der Integration richtet sich hierbei nach der Größe und Komplexität des Systems. Absicherungen auf der Gesamtsystemschicht umfassen System- und Integrationstests, falls es kein Systemmanagement auf dieser Schicht gibt, wie im Beispiel „Anhalten“. Umfasst eine Schicht keine (Hardware-) Komponentenanteile, sondern allein Software, die auf Komponenten anderer (Sub-)Systeme verteilt ist, wie der `Vehicle-Drive-Controller`, so werden diese Softwareanteile einem Software-Komponententest unterzogen. Ist die Rückmeldung des Software-Komponententests positiv und die integrierende Komponente einsatzbereit, werden weitere Integrationstests durchgeführt. Die Absicherung des Hardware- und Softwareverbundes erfolgt im jeweiligen (Sub-) System mit der zugehörigen Komponente. Alle Komponenten, die anderen (Sub-)Systemen zugeteilt sind, werden während den Intergrations- und Systemtests einer Schicht als Black-Box oder Grey-Box-Testkomponenten angesehen. In Abbildung 4.17 integriert der `Elektrische Antrieb`, sowohl die `Elektrische Maschine` als auch den `HV-Speicher`. Die Spezifikation des HV-Speichers ist über SMArDT zwar einzusehen, dennoch ist der HV-Speicher aus der Sicht des elektrischen Antriebs eine Grey-Box oder Black-Box. Umfasst die Schicht eine oder mehrere Hardware-Komponenten, wie die

`Elektrische Maschine`, sind ebenfalls HW-Komponententests durchzuführen.

Die Fülle an Tests, die in Abbildung 4.17 exemplarisch dargestellt sind, dienen dann der Überprüfung der Spezifikationen. Dies ist in der Regel ein iterativer Prozess, bei dem die Testergebnisse zur Überarbeitung eben dieser Spezifikationen verwendet werden. Die gewonnenen Erfahrungen und Testartefakte fließen hierbei über ein Feedback in die Spezifikation der nächsten Iteration mit ein. Abbildung 4.18 stellt die Test-Iterationen und deren Erkenntnisrückfluss in die nächste Spezifikationsüberarbeitung am Beispiel der Ebene B dar.

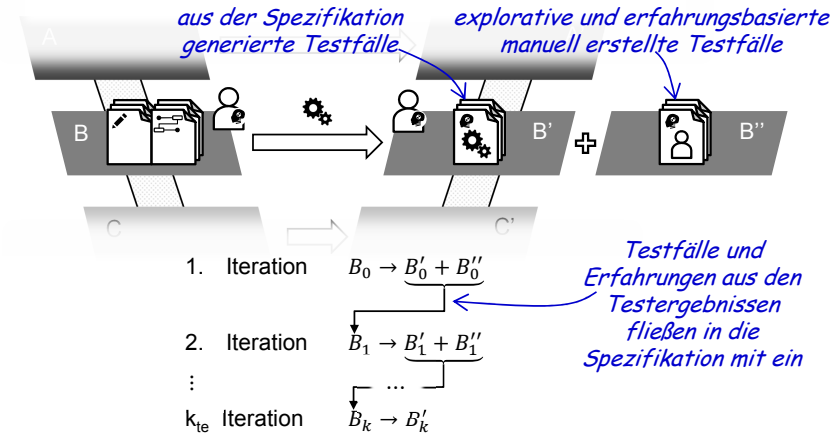


Abbildung 4.18: Darstellung des iterativen Testverfahrens bei der Prüfung der Spezifikation (B_k) durch automatisiert (B'_k) und manuell (B''_k) erstellte Teställe während k Iterationsschritten

Zusätzlich zu den anforderungsbasierten automatisiert erstellten Testfälle B' werden für die Absicherung anfangs ebenfalls explorative und erfahrungsbasierte Testfälle B'' manuell erstellt. Die Erkenntnisse der Absicherung in dieser Iteration k fließen über das Feedback der Tester*innen in die Spezifikation der nächsten Iteration $k+1$ mit ein. Infolgedessen umfassen die generierten Testfälle B'_{k+1} aus der Spezifikation B_{k+1}, sowohl die generierten Testfälle B'_k als auch die manuell erstellten Testfälle B''_k der vorherigen Iteration. Das Ziel der iterativen Vorgehensweise ist, dass irgendwann die automatisiert abgeleiteten Testfälle für eine komplette Absicherung ausreichen und kein Mehraufwand der manuellen Testfallerstellung notwendig ist ($B_k \rightarrow B'_k$).

Für jede Modelliteration existieren aus dem Modell abgeleitete Testartefakte. Da das in dieser Arbeit vorgestellte Modell formal ist, können diese Testartefakte automatisiert generiert werden. Die automatisierte Testdatengenerierung aus Modellen besitzt gegenüber rein manuell erstellten Tests zwei Vorteile. Erstens bieten diese Testfälle eine bessere Systematik für die Absicherung und zweitens können die Tests mit einer höheren Effizienz durchgeführt werden [Pre03]. Diese Systematik führt zu einer erhöhten Produktqualität [Pre03]. Außerdem ermöglicht die Trennung der Testfallerstellung und der Absicherung die Konzentration auf jeweils einen wesentlichen Teil nicht nur die intellektuelle Durchdringung des Problems.

Dieser iterative Test-first Ansatz wird ebenfalls in den anderen Abstraktionsebenen A, C und D angewendet. Der Rückfluss aus Tests der Ebene A schließt insbesondere Anforderungslücken. Die Tests auf Ebene B ermöglichen dagegen vor allem funktionale Konzeptänderungen, während die Testergebnisse aus Ebene C konzeptionelle/ technische Änderungen hervorrufen. Der Rückfluss auf Ebene D umfasst überwiegend Änderungen der Hardware- und Software- Implementierung. Da die iterativ optimierte Spezifikation neben der Entwicklung des Produktes ebenfalls in anderen Domänen wie der funktionalen Sicherheit als Grundlage dient, profitieren nicht nur die Testgruppen von diesem Verfahren. Für die Testgruppen sind wiederum die Verbesserungen und der zusätzliche Informationsgehalt in anderen Dömanen von Vorteil. Falls die Informationen der anderen Domänen nicht für die eigene Domäne relevant sind, lassen sich diese über Sichten wie zum Beispiel in [GHK+08a, MRR13] aus- oder einblenden.

4.5 Evolutionäre Entwicklungen mit SMArDT

In den Grundlagen von SMArDT in Abschnitt 2.4 wird von einer Funktionsentwicklung auf der grünen Wiese ausgegangen (vgl. Theorie und Praxis 4.3). Allerdings trifft dies nur für Funktionalitäten zu, die noch nicht entwickelt sind. Sind diese nach einer modellgetriebenen SMArDT Methode bereits spezifiziert, entwickelt, verifiziert und validiert ist ein anderes Szenario gefragt. Abbildung 4.19 illustriert eine Übersicht über eine nach SMArDT spezifizierte Funktion mit exemplarischen Artefakten.

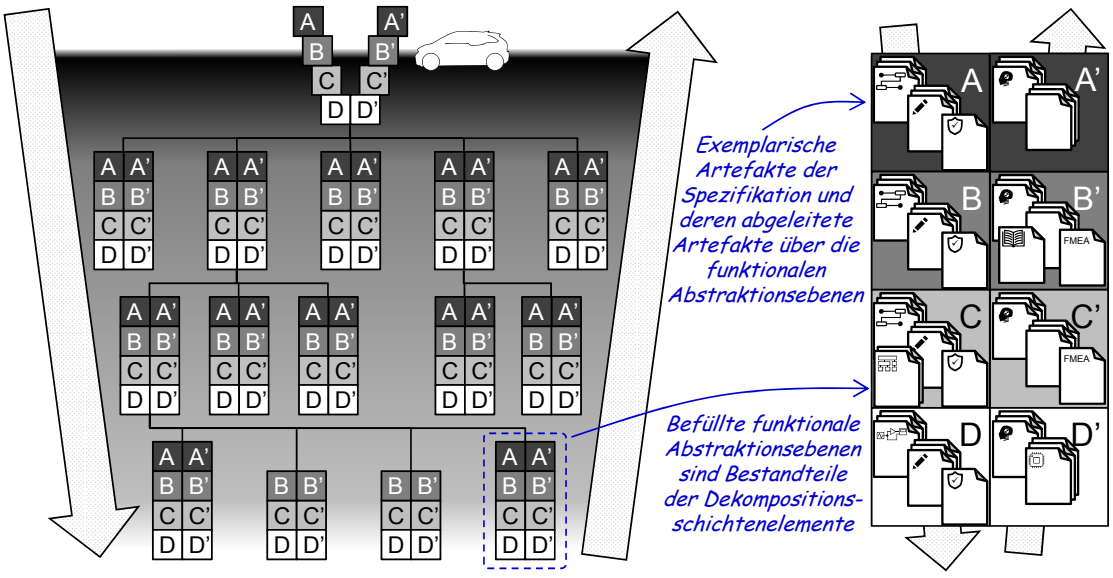


Abbildung 4.19: Exemplarische Übersicht über eine nach SMArDT spezifizierte Funktion mit exemplarischen Artefakten

Eine Funktionalität wurde über die hierarchische Dekomposition (in Abbildung 4.19 links) bereits zugeteilt und fertig entwickelt. Die einzelnen Funktionen sind über die

funktionale Abstraktion folglich spezifiziert, realisiert, integriert, validiert und verifiziert. Die Artefakte der Abstraktionsebenen A, B, C und D geben die Spezifikation des Produktes vor (Beispiel in Abbildung 4.19 rechts). Die abgeleiteten Elemente A', B', C' und D' sind automatisiert erstellt und das Ergebnis des iterativen Prozesses beschrieben. Wurde eine Funktion mittels der funktionalen Abstraktion ordnungsgemäß von A, nach B, nach C und nach D durchlaufen, entwickelt und über die Kette von D', nach C', nach B' bis A' korrekt validiert und verifiziert ist die erste Entwicklung und Prüfung des Produktes abgeschlossen. Das im übertragenen Sinn „befüllte V-Modell“, besitzt somit die Artefakte der Spezifikationsblöcke A, B, C und D und die abgeleiteten Artefakte der Integrationsblöcke A', B', C' und D'. Alle Spezifikationsblöcke, beziehungsweise befüllten V-Modelle, ergeben einen vollständig abgestimmten, verifizierten und validierten Funktionsbibliothek. Diese vollständige Funktionsbibliothek besteht mehreren kleinen Funktionsbibliotheken wie Ebene B in Schicht 3 und spezifiziert folglich das gesamte mechatronische System.

Die entwickelten Spezifikationen aus den Abstraktionsebenen B und C sind die Eingangsleistung der jeweiligen Ebene A des zugehörigen Subsystems (vgl. Abbildung 4.10). Ein entwickeltes und geprüftes Produkt des Subsystems kann in der höheren Dekompositionsschicht integriert werden. Die abgeleiteten Artefakte A', B', C' und D' umfassen unter anderem Code, Hardware, Signalbibliotheken, Testartefakte und FMEA.

Für eine Neuentwicklung, einen Tausch oder eine Modifikation einer Funktion oder einer Hardwarekomponente kann mit Hilfe der spezifizierten Architekturschnittstellen identifiziert werden, welche Maßnahmen auf welcher Ebene notwendig sind. Wird eine Funktion eines Systems neu entwickelt oder geändert, kann sich dieses System an den Funktionen der Funktionsbibliotheken oder Spezifikationsblöcken der untergeordneten Subsysteme bedienen. Ist keine Funktion vorhanden, kann das betrachtete System diese Funktion direkt umsetzen oder gibt eine neue Funktion in Auftrag. Diese Funktion kann auch eine Änderung einer bestehenden Subfunktion eines Subsystems beinhalten. Wurde zum Beispiel im `elektrischer Antrieb` die Spezifikation der Funktion `E-Drehmoment stellen` geändert, muss geprüft werden, ob diese Änderungen von den (Sub-) Funktionen noch erfüllt werden. Ist keine Anpassung oder Neuentwicklung der (Sub-) Funktionen notwendig müssen diese befüllten V-Modelle der Subsysteme nicht modifiziert werden. Ist eine Änderung der Funktion eines Subsystems (hier `E-Maschine`) notwendig, kann diese mittels SMArDT neu entwickelt oder in den jeweiligen Spezifikationsblöcken angepasst werden.

Zu beachten ist, dass eine Änderung in der nächst höheren Ebene sehr wahrscheinlich eine Änderung der untergeordneten Ebene nach sich zieht (vgl. Abbildung 4.19). Wird zum Beispiel eine Änderung in der funktionalen Ebene B geändert, muss das Ebene B entsprechende Konzept (C) und die Realisierung (D) geprüft und gegebenenfalls angepasst werden. Da die Elemente B' aus B, C' aus C und D' aus D abgeleitet sind, müssen diese ebenfalls geprüft und gegebenenfalls neu abgeleitet werden. Abbildung 4.20 illustriert eine Änderung eines Funktionsprinzips auf Ebene B in Schicht $n = 2$, kurz B_2, am Beispiel von `Zielgeschwindigkeit ermitteln` aus Abbildung 4.6.

Für die Ermittlung der Zielgeschwindigkeit soll in Abbildung 4.20 eine neue Information

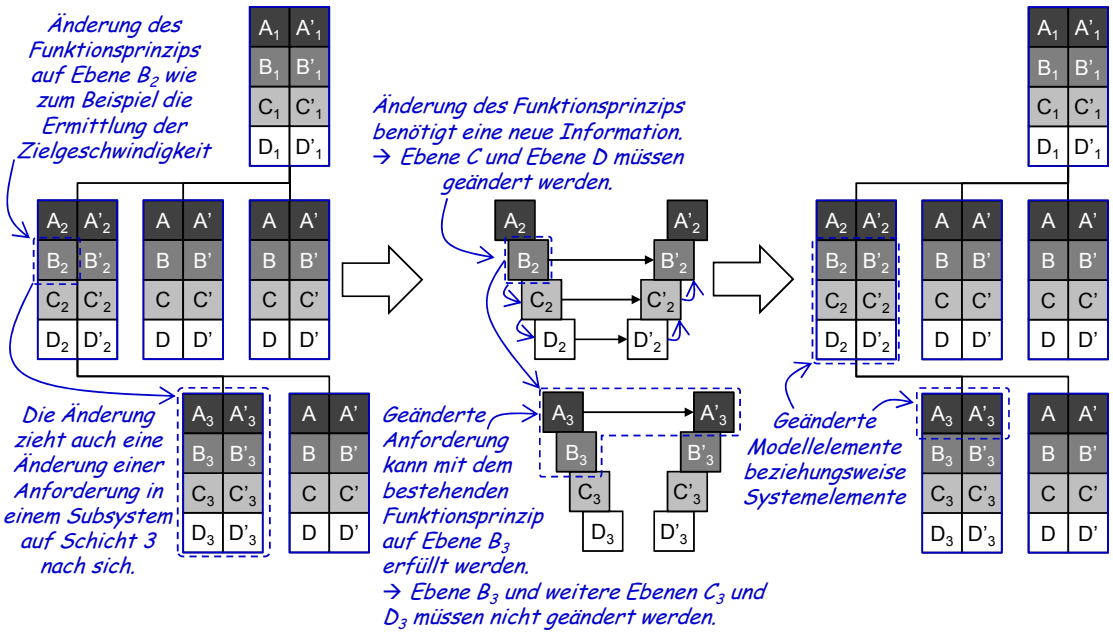


Abbildung 4.20: Exemplarische Übersicht über eine nach SMArDT spezifizierte Funktion mit exemplarischen Artefakten

beziehungsweise Informationsfluss hinzugezogen werden. Die Information existiert bereits im Gesamtsystem aber wird von `Zielgeschwindigkeit ermitteln` noch nicht verwendet. Infolgedessen muss die Subfunktion auf Ebene C_2 und die Realisierung auf Ebene D_2 ebenfalls in der Software nachgezogen werden. Mittels der neu abgeleiteten Elemente für D_2', C_2' und B_2' lassen sich die Änderungen verifizieren. Zusätzlich beeinflusst die Änderung des Funktionsprinzips auf Ebene B_2 auch eine Anforderung eines Systems in Dekompositionsschicht $n = 3$. Allerdings erfüllt im Beispiel aus Abbildung 4.20 das Funktionsprinzip auf Ebene B_3 bereits die Anforderungen und es müssen nur betroffenen Artefakte in A_3 angepasst und Artefakte in A_3' abgeleitet werden.

Ebenfalls zu beachten ist, dass eine Systemänderung in einer Dekompositionsschicht, die Funktion der anderen Systeme auf der gleichen Dekompositionsschicht nicht beeinträchtigt. Liegt eine Beeinträchtigung vor, wird vom integrierenden System $n-1$ geprüft, ob die Änderung nach wie vor der Spezifikation entspricht. Liefert die betrachtete Funktion der Leistungselektronik beispielsweise bestimmte Energiewerte für anderen Funktionen, müssen diese Werte bei einer Funktionsänderung ebenfalls wieder vorhanden sein. Ist dies nicht der Fall, muss die andere Funktion angepasst und geprüft werden, ob die Spezifikation der übergeordneten Dekompositionsebene erfüllt wird. Im Beispiel von Abbildung 4.20 ist keine Prüfung des übergeordneten Systems $n = 1$ nötig, da das System $n = 2$ nach außen hin sein verhalten nicht ändert.

Anhand dieses Verfahrens können Funktionalitäten beliebig modifiziert, weiterentwickelt oder neu entwickelt werden und fügen sich in die vollständige Funktionsbibliothek ein. Dies

impliziert, dass bewährte Funktionen auch bei erneuerter aktueller Technik übernommen werden und abgeleitete Artefakte je nach Änderung wiederverwendet werden. Die nur zum Teil modifizierte Funktionsbibliothek erleichtert auch die Integration und die V&V. Da das Vorgängermodell schon verifiziert und validiert wurde und abgeleitete Artefakte existieren, kann der Testfokus effizient auf (Sub-)Systeme deren Modelle geändert wurden gelegt werden.

Auch bei Neuentwicklungen eines Produkts kann auf die bestehende vollständigen Funktionsbibliothek zurückgegriffen werden. Zum Beispiel kann bei der Neuentwicklung einer Benutzerschnittstelle die generelle Funktionalität des existierenden Produktes (Ebene B) übernommen werden, aber die Umsetzung muss natürlich auf den neuesten Stand der Technik angepasst werden (Ebene C und D). In einer Zeit, in der man nicht sicher weiß, welche Technologie sich durchsetzen wird, die grundlegende Funktionalität jedoch gleich bleibt, kann sich ein technologieunabhängiges Vorgehen als Wettbewerbsvorteil herausstellen. Besonders mit einer wie in dieser Arbeit vorgestellten modellgetriebenen Methodik und automatisierten Verfahren ist die Neuentwicklung von Produkten agil und schnell möglich.

Zusammenfassung von Kapitel 4

Ein Unterschied der in dieser Arbeit weiterentwickelten SMArDT 2.0 Methode und der Ursprungsmethode SMArDT aus Abschnitt 2.4 ist, dass die Trennung zwischen Abstraktions- und Dekompositionsschritten verbindlich ist (vgl. Abbildung 4.10). Diese Differenzierung ermöglicht die Entwicklung einer eindeutigen und klaren Funktionsbibliothek. Der modulare Aufbau und die klare Unterscheidung der Verfeinerungsschritte vereinfachen eine Anpassung der Spezifikation und infolgedessen des Systems. Auf dieser Basis lassen sich zukünftige Modifikationen des Systems agil umsetzen und die Spezifikation um neue Funktionalitäten erweitern und integrieren (vgl. Theorie und Praxis 4.5).

Ein weiterer Unterschied zu der Ursprungsmethode SMArDT ist, dass die vorgestellte Weiterentwicklung sich auch für Brownfield-Systeme eignet (vgl. Theorie und Praxis 4.3). In diesem Zusammenhang sind die Schritte der Abstraktion und Dekomposition klar beschrieben und Unterschiede von Greenfield und Brownfield hervorgehoben. Im Gegensatz zu klassischen SMArDT ist diese Beschreibung detailliert und umfasst auch die Integration, V&V und Evolution im Detail.

Außerdem distanziert sich die weiterentwickelte Methode von der Modell-zu-Text-zu-Modell Übersetzung (vgl. Theorie und Praxis 4.6). Das heißt, wenn das Modell eines Systems vorliegt, werden die Anforderungen dieser Modellform an die Subsysteme übergeben. Sie werden folglich nicht (manuell) in Text übersetzt und über die Ebene A übergeben, um dann erneut modelliert zu werden. Auch wenn das Gesamtmodell aus Gründen der physikalischen Speichergröße, organisatorischen Aspekten oder intellektuellen Beherrschbarkeit der Komplexität in kleinere Modelle aufgeteilt wird, können die Modellzusammenhänge folglich immer eindeutig bestimmt werden. Zusätzlich werden mit einem durchgängigen Modell viele modellgetriebene Verfahren ermöglicht.

Kapitel 5

Testmethodenanforderungen

Im vorangegangenen Kapitel wurde die modellgetriebene SMArDT Entwicklungsmethodik beschrieben und die Methodik für die Absicherung der einzelnen Ebenen und Schichten erläutert. Der weitere Verlauf der Arbeit fokussiert sich auf die Absicherung beziehungsweise Verifizierung der abstrakten Ebene B.

Dieses Kapitel entwickelt ein Zielbild, das mit der automatisierten Testfallerstellung aus Spezifikationsmodellen erreicht werden soll. Hierfür wird ein Ansatz gewählt, bei dem Testfälle automatisiert aus Verhaltensmodellen erstellt werden. Diese automatisierte Testfallerstellung unterstützt testfallerstellende Personen und verbessert infolgedessen die Effizienz des Testfallerstellungsprozesses.

Zu diesem Zweck wird auf die Testbarkeit von Anforderungsmodellen in Abschnitt 5.1 eingegangen. In Abschnitt 5.2 werden eine Studie über den aktuellen Status der Testfallerstellung in der Automobilindustrie und die Erwartung an eine modellgetriebene Testfallerstellung vorgestellt. Anschließend werden die gewonnen Erkenntnisse der Studie in Abschnitt 5.3 mit SMArDT in Zusammenhang gebracht und für diese Arbeit weiter konkretisiert. In Abschnitt 5.4 werden die herausgearbeiteten Anforderungen für den weiteren Verlauf der Arbeit kurz zusammengefasst.

5.1 Kriterien für die Testbarkeit von Spezifikationsmodellen

Das in Kapitel 4 vorgestellte SMArDT Spezifikationsmodell fungiert neben der Spezifikation des Systems als Testorakel. Das Modell dient als zentrale, eindeutige und verlässliche Datenbasis (engl. „Single Point of Truth“) für alle anderen Elemente wie Systeme, Dokumente und Testfälle. Infolgedessen sollen ungewollte Redundanzen, die häufig zu Inkonsistenzen führen [Rum17], vermieden werden. Aus diesem Zusammenhang ergeben sich für das Modell drei **allgemeine Kriterien**:

Aktualität: Das Modell muss die aktuelle Spezifikation umfassen. Ein Spezifikationsmodell ist aus der Testsicht aktuell, wenn sie dem aktuellen, zu testenden Stand der Realisierung entspricht.

Vollständigkeit: Das Modell muss für den jeweiligen Testzweck vollständig sein. Die in dieser Arbeit im Fokus stehenden anforderungsbasierten Tests sind stark kontextabhängig. Ist das Modell hinsichtlich des zu testenden Testobjektes unvollständig oder nicht einheitlich spezifiziert, leiden die Praktikabilität der Testfallerstellung und folglich die Testfallqualität [ISO13a].

Verfolgbarkeit: Das Modell und deren Elemente müssen leicht zugänglich und nachvollziehbar sein[1]. Eine Verfolgbarkeit kann beispielsweise mit einem gemeinsam verwalteten Versionsmanagement[2] oder einem Repository mit Querverweisen gewährleistet werden. Vor allem die Schritte der Abstraktion und Dekomposition müssen klar definiert sein, um eine Zuweisung und Abgrenzung der Testumfänge gewährleisten zu können.

Zusätzlich zu den allgemeinen Kriterien muss für die Absicherung des Systems eine Testbarkeit gewährleistet sein. In dem Kontext dieser Arbeit gelten für die **Testbarkeit** folgende Kriterien:

Determiniert: Das Testergebnis der Testfälle auf der Basis des Modells müssen determiniert beziehungsweise eindeutig sein. Auch wenn das (Spezifikations-)Modell per Definition nicht deterministisch beziehungsweise unterspezifiziert ist [Rum16], muss bei einer Testfallerstellung ein eindeutiges Ergebnis reproduzierbar sein [Rum17].

Widerspruchsfreiheit: Das Modell muss widerspruchsfrei sein. Hierfür müssen Redundanzen und Inkonsistenzen ausgeschlossen werden, was durch Modellprüfungen zum Teil bewerkstelligt werden kann.

Technische Realisierbarkeit: Ein für die Absicherung benötigtes Testobjekt, wie das System oder die Testumgebung, muss in einem sinnvollen Zeitraum[3] verfügbar sein. Ein Modell, dass erst nach der Produktion eines Produkts den Tester*innen zur Verfügung steht, ist zum Beispiel für die Absicherung davor nutzlos.

Teststufenorientierung: Die Absicherung eines Testobjektes muss auf der richtigen Teststufe verordnet werden. Die Teststufe ist in SMArDT an die Abstraktionsebenen und Dekompositionsschichten gebunden. Auf diese Weise kann klar zwischen den verschiedenen Komponenten-, Integrations- und Systemtest unterschiedenen werden. Mit der Teststufe werden bestimmte Testziele und Projektsituationen verknüpft. Zum Beispiel ist es praktikabel bestimmte Funktionalitäten,wie das Verhalten bei Extrembedingungen (z.B. bei $+100°C$) in Software-Komponententests oder auf der Teststufe der Komponenten nachzuweisen. Solche Fälle lassen sich nämlich im Allgemeinen nicht auf der Teststufe der Hardware-Integrationstests oder Systemtests praktikabel nachstellen. Zudem ist es nicht immer praktikabel, das komplexe Zusammenspiel der Komponenten unter Realbedingungen in Software-Komponententests eins-zu-eins nachzubilden, da sich die Realbedingungen oft auch relativ effizient simulieren lassen[4].

[1] Die Verfolgbarkeit ist nicht direkt ein Kriterium, dass das Modell erfüllen muss, sondern umspannt mehr die Methodik und den Einsatz des Modells. Allerdings muss dieser Einsatz mit dem vorhandenen Modell effektiv und effizient realisierbar sein, wie beispielsweise eine versionierbare Datenbasis des Modells und die Möglichkeit von eindeutigen Querverweisen.

[2] Auf ein Versionsmanagement von SMArDT wird in dieser Arbeit nicht eingegangen.

[3] Ein „sinnvoller Zeitraum“ ist im Kontext der Entwicklung in der Automobilindustrie ein Zeitpunkt mit ausreichend Zeitreserve vor dem SOP (vgl. Abbildung 3.3).

[4] Je umfangreicher und komplexer ein System ist, desto größer ist auch der Aufwand es zu testen und desto schwieriger ist es davon zu abstrahieren.

Grundsätzlich ist die Testbarkeit von vielen Faktoren abhängig. Beispiele sind der Detaillierungsgrad der Systeminformationen[5], der Grad der Vernetzung der Funktionen und des Systems mit deren Subsysteme und die vorhandenen/benötigten Testressourcen wie Testumgebungen, Zeit und Fachpersonal. Die Testbarkeit muss infolgedessen anhand der Situation des Projekts bestimmt werden.

Neben den Allgemeinen und Testbarkeitskriterien gibt es **Prioritätskriterien**:

Sicherheit: Die ASIL-Klassifikation der Anforderungen sollte definiert sein, um die funktionalen Auswirkungen und Maßnahmen der Funktion nach der ISO 26262 in die Absicherung mit einzubeziehen.

Grundfunktionalität: Anforderungen, die eine Grundfunktionalität betreffen, müssen definiert sein, um die Planbarkeit der Testaktivitäten und anderer Domänenaktivitäten zu erleichtern.

Das Spezifikationsmodell nach SMArDT kann grob in SysML Diagramme und textuelle Anforderungen eingeteilt werden. Abbildung 5.1 stellt die Beziehungen der Modellartefakte und deren Schnittstellen dar. Das Modell spezifiziert mindestens ein System. Da im Modell auch Informationen zu mehreren Systemen enthalten sein können, ist es möglich, mehr als ein System zu spezifizieren. Ein Modell kann mehrere Varianten eines Systems mit daraus resultierenden unterschiedlichen Systemen umfassen, die mit der Hilfe von Sichten, wie beispielsweise in [GHK+08a, MRR13], weiter fokussiert werden. Das zu testende System wird gegebenenfalls in einer Testumgebung simuliert. In dieser Testumgebung wird ein Test ausgeführt. Die für den Test benötigten Testeingaben und erwarteten Ergebnisse werden von einem Testfall geliefert. Dieser Testfall wird aus einem oder mehreren Verhaltensdiagrammen erstellt und für eine Rückverfolgbarkeit mit diesem Diagramm verknüpft. Eine Verknüpfung besteht ebenfalls bidirektional zwischen den Testfällen und den zugehörigen textuellen Anforderungen. Die textuellen Anforderungen liefern zusätzliche Bedingungen und Informationen, wie beispielsweise die Funktionsweise unter bestimmten Lastbedingungen. Die Verknüpfungen zu den Elementen des Testorakels, wie Verhaltensdiagrammen und textuellen Anforderungen, dienen insbesondere dem Feedback für das Modell und der Wartbarkeit.

Da sowohl die Diagramme als auch die textuellen Anforderungen ausschlaggebend für die Testergebnisse sein können, müssen diese eng miteinander verknüpft werden. Generell können die textuellen Anforderungen und deren Beziehungen mittels eines Anforderungsdiagramms dargestellt werden (vgl. Abbildung 4.15). Die textuellen Anforderungen entsprechen den Verhaltensdiagrammen, den Strukturdiagrammen und deren Elementen. Der Begriff „entsprechen“ verdeutlicht die Widerspruchsfreiheit und Gleichwertigkeit zwischen einer Anforderung und dem zugeordneten Element einer Funktion in einem Diagramm. Abbildung 5.2 stellt die Beziehungen aus Abbildung 5.1 abstrakt dar.

[5] Besonders in der anfänglichen Testphase, in der das Testkonzept entwickelt wird, entsprechen die vorhandenen Systeminformationen nicht den benötigten Systeminformationen. Zum einen liegen die Anforderungen nicht in der benötigten Aktualität und Vollständigkeit vor. Zum anderen sind aufgrund der noch andauernden Entwicklungsänderungen die Anforderungen zum Teil nicht vollumfänglich eindeutig und widerspruchsfrei in Bezug auf das System. Hier sind ein intensiver Austausch und eine

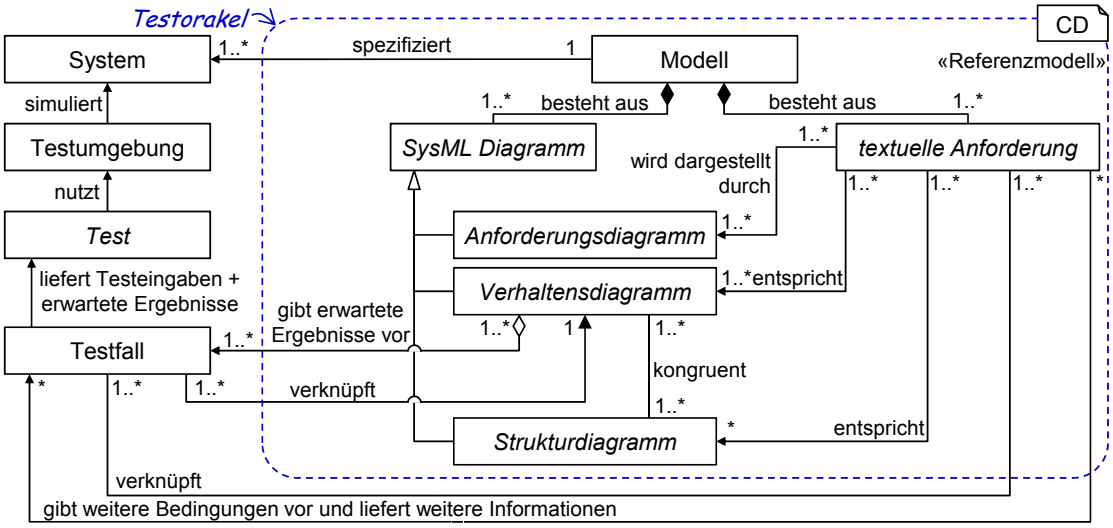


Abbildung 5.1: Beziehungen der textuellen Anforderungen und der Diagramme des Modells hinsichtlich der Testfallerstellung

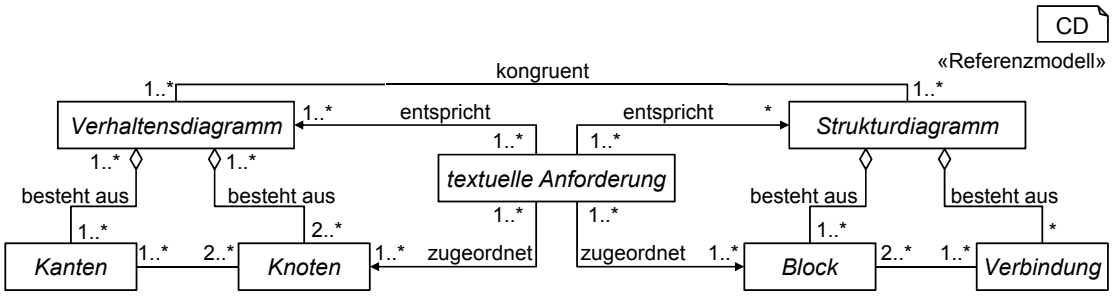


Abbildung 5.2: Zusammenhang der Anforderungen mit den Elementen der Verhaltens- und Strukturdiagramme

Die textuellen Anforderungen entsprechen ihren Diagrammen und somit implizit ihren zugeordneten Knoten und Blöcken. Die Kanten der Verhaltensdiagramme werden über Knoteneingänge und -ausgänge implizit den textuellen Anforderungen adressiert. Analog werden Verbindungen der Strukturdiagramme über die anliegende Eingänge und Ausgänge den textuellen Anforderungen zugeordnet. Folgende Invarianten gelten für das Diagramm aus Abbildung 5.2:

- Eine `textuelle Anforderung` wird mindestens einem `Knoten` zugeordnet.
- Eine `textuelle Anforderung` kann einem `Block` zugeordnet werden.
- Eine `textuelle Anforderung` entspricht mindestens einem `Verhaltensdiagramm`.

Zusammenarbeit zwischen den Entwicklungsteams und Testteams nötig.

- Eine `textuelle Anforderung` kann einem `Strukturdiagramm` entsprechen.
- Über die Beziehung „`textuelle Anforderung` → `entspricht` → `Verhaltensdiagramm`“, die Beziehung „`textuelle Anforderung` → `zugeordnet` → `Knoten`“ und die Beziehung „`Verhaltensdiagramm` → `besteht aus` → `Knoten`“ entspricht die `textuelle Anforderung` dem ihr zugeordneten `Knoten`.
- Ist eine `textuelle Anforderung` einem `Block` zugeordnet, muss diese `textuelle Anforderung` dem `Block` entsprechen.

Das Ziel ist die Informationen der Diagramme mit den textuellen Anforderungen zu ergänzen. Zu diesem Zweck sind die textuellen Anforderungen in das Modell integriert (vgl. Kapitel 4) und können falls nötig von Testfällen verknüpft werden. Oftmals ist dies in der Praxis nicht ohne Weiteres möglich, da etablierte Prozesse und Tools exisitieren (vgl. Theorie und Praxis 4.6). Theorie und Praxis 5.1 greift diesen Umstand auf und gibt eine Empfehlung, wie diesem begegnet werden kann, ohne unnötige Redundanzen zu erzeugen.

Theorie und Praxis 5.1 (Single Point of Truth von textuellen zu grafischen Anforderungen)**.** *Wie in Theorie und Praxis 4.6 beschrieben, müssen sich modellgetriebene Ansätze in bestimmten Domänen noch weiter etablieren. Eine Möglichkeit den Single Point of Truth von einem textuellen, natürlichsprachlichen Anforderungsmanagement Schritt für Schritt zu einem (formalen) Modell zu überführen, ist die funktionalen Anforderungen aus den erstellten Diagrammen automatisiert auszuleiten. Diese automatisiert erstellten Anforderungen entsprechen eins-zu-eins den formalen Diagrammen. Hierbei entsteht eine Redundanz, die allerdings nur für den Übergang gedacht ist, um den Single Point of Truth zu verschieben. Die erstellten textuellen Anforderungen dürfen weder geändert noch weiterverarbeitet werden. Somit lässt sich eine Formalität der textuellen Anforderungen sicherstellen und die Wartung erleichtern, da nur die Diagramme als Single Point of Truth dienen. Mit dieser Möglichkeit kann sichergestellt werden, dass Prozesse und Tools, die auf textuellen Anforderungen basieren, nicht beeinträchtigt werden. Eine weitere Möglichkeit ist, alle vorhandenen textuellen Anforderungen in Dokumentenform initial in ein Modell zu überführen und die Dokumente zu „verwerfen“. Anschließend kann mit diesem Modell als Single Point of Truth gearbeitet werden. Allerdings wird dieser Prozess härter in bestehende Prozesse und Tools eingreifen, da neue Prozesse etabliert und alte Prozesse auf einen Schlag obsolet werden. Beide Möglichkeiten spannen einen Raum an Restrukturierungsmöglichkeiten für ein Modell als Single Point of Truth auf. Letztendlich muss individuell entschieden werden, welche Restrukturierungsmöglichkeit sich am besten eignet.*

Zusätzlich zur Widerspruchsfreiheit und Angemessenheit ist für die Nachvollziehbarkeit der Testfälle eine Verknüpfung der textuellen Anforderung mit Diagrammelementen für die Erfüllung von Standards wie der ISO 26262 nötig (vgl. Abbildung 5.2). Somit kann evidenzbasiert eine Abdeckung der Verhaltensdiagramme und demzufolge der Erfüllung der angeforderten Funktion nachgewiesen werden. Ein solches strukturelles

Vorgehen entspricht den aktuellen Normen wie beispielsweise in [ISO17a, DIN15, ISO18a]. Die Widerspruchsfreiheit muss auch unter den Diagrammen selbst gelten. Verhaltens- und Strukturdiagramme müssen kongruent sein. Die Assoziation *kongruent* verdeutlicht die Übereinstimmung in allen Punkten bezüglich des zugeordneten Diagramms. Es ist möglich, dass in einem Verhaltensdiagramm nicht alle Elemente eines zugehörigen Strukturdiagramms dargestellt werden, allerdings stimmen die Elemente und deren Eigenschaften exakt überein. Der umgekehrte Fall ist gleichermaßen möglich. Mit diesem Prinzip können verschiedene Sichten auf Funktionen und deren Struktur erstellt werden.

Die in dieser Arbeit vorgestellte Methode nutzt die Verhaltensdiagramme des SMArDT Modells für die automatisierte Erstellung von Testfällen. Aus diesem Kontext ergeben sich weitere *Anforderungen für die Diagramme*:

Wohlgeformtheit: Die Diagramme müssen wohlgeformt sein.

Einheitliche formale Sprache: Eine einheitliche wohldefinierte Sprache.

Einheitliches Abstraktionsniveau: Diagramme der gleichen SMArDT Ebene und Schicht müssen ein gleiches Abstraktionsniveau aufweisen. Ein gleiches Abstraktionsniveau ist die Voraussetzung für eine einheitliche Teststufenorientierung.

Echte maschinenlesbare Informationen: Die Diagramme müssen Informationen über das Verhalten der betrachteten Funktion für funktionsorientierte Tests enthalten. Es reicht nicht ein formales Modell zu besitzen, welches von einer Person gelesen und interpretiert werden kann, aber maschinell interpretiert keinen Sinn ergibt.

Verknüpfung der Modellelemente und der Testfälle: Testfälle, Diagramme und deren textuelle Anforderungen müssen untereinander verknüpft sein. Wird etwas im Modell geändert, so lassen sich die die notwendigen Änderungen an Testfällen rasch identifizieren. Umgekehrt können von mit einem Testfall aufgedeckte Widersprüchlichkeiten zwischen System und Modell leichter detektiert und verbessert werden.

5.2 Statusanalyse - Wie wird traditionell im E-Antrieb getestet?

Das Testen von Fahrzeugen, deren Subsystemen und Komponenten sowie Hardware- und Softwareelementen ist ein bekanntes Gebiet auf dem Testingenieur*innen in der Regel bereits bestimmte Erfahrungen aufweisen und Strategien verfolgen. Deshalb wurde im Rahmen dieser Arbeit eine systematische Umfrage in der Automobilindustrie zum Thema Testen mit dem Fokus auf modellbasiertes Testen durchgeführt. Die aggregierten Umfrageergebnisse wurden erstmals 2018 in [KMS+18] und 2019 in [DGH+19] detailliert veröffentlicht. Dieser Abschnitt beschreibt die Forschungsfragen, das Umfragedesign und seine Ergebnisse im Detail und ergänzt die Umfrageergebnisse um die für diese Arbeit relevanten Details und Schlussfolgerungen.

Forschungsfragen zum Thema Testen

In der Vergangenheit hat es sich oft als schwierig erwiesen, Ingenieur*innen modellbasierte Technologien aus der Softwareentwicklung extrinsisch vorzugeben (vgl. Abschnitt 2.5). Deshalb und um bereits bestehende Erfahrungen und Strategien aufzugreifen, untersucht die Umfrage dieser Arbeit potenzielle Nutznießer*innen bei der BMW Group. Zudem wurde hier untersucht, inwiefern bei der Einführung bestimmte Barrieren auftreten und wie es um die Motivation der Testingenieur*innen bestellt ist. Das Ziel der Umfrage ist es, zu analysieren, ob MBT sich dazu eignet, die Testqualität zu verbessern und zuversichtliche Anwendungsszenarien zu identifizieren. Zwei übergreifende Fragen sind die der Testabdeckung[6] und die der Verwendbarkeit der einzelnen Testfälle. Des Weiteren wird der Einsatz verschiedener Testumgebungen und der zugehörigen Teststufen untersucht. Traditionell sind natürlichsprachliche textuelle Anforderungen die Grundlage für die Testfallerstellung und das Testen in der Automobilindustrie (vgl. Abschnitt 3.3).

Abbildung 5.3: Vergleich der traditionellen Testfallerstellungsmethode mit der modellbasierten automatisierten Testfallerstellung mit SMArDT [DGH+19]

Abbildung 5.3 veranschaulicht die traditionelle Testfallerstellung und vergleicht sie mit dem Konzept der modellbasierten Testfallerstellung mit SMArDT. Mit der modellbasierten Testfallerstellung wird die manuelle Aufbereitung der natürlichsprachlichen textuellen Anforderungen aufgrund der automatischen Verarbeitung modellbasierten Anforderungen hinfällig. Trotzdem sollte angesichts der Erfahrung und der Kompetenz der Testingenieur*innen weiterhin ermöglicht werden, bestimmte Testfallaspekte manuell zu konfigurieren. Daher sind die wichtigsten Forschungsfragen der Umfrage:

R1 Was ist der Hintergrund der Testingenieur*innen? Wer kann von einer (halb-) automatisch modellbasierten Testfallerstellung profitieren?

R2 Ist es möglich, die Testqualität und die Testabdeckung durch MBT zu verbessern (aus der Sichtweise von der Testingenieur*innen)?

R3 Welche Umgebungen sind für modellbasiertes Testen im Kontext der BMW Group am vielversprechendsten?

[6] Die Testabdeckung trifft eine Aussage, in welchem Verhältnis die Spezifikation durch Testfälle tatsächlich abgedeckt wird gegenüber der theoretisch möglichen Abdeckung.

Das übergeordnete Ziel der Umfrage ist es, das Potenzial der modellbasierten Testfallerstellung herauszustellen, im Vergleich mit der traditionellen Vorgehensweise. In Bezug auf Forschungsfrage **R1** untersucht der Fragebogen die Haupttätigkeit der Teilnehmer*innen, deren Arbeits- beziehungsweise Bildungshintergrund, die aktiven Jahre in der Automobilindustrie, die langjährigen Erfahrungen bei der Erstellung von Testfällen, den Fokus der Testaktivitäten und die Einstellung gegenüber MBT im Allgemeinen. Auf der Grundlage der gewonnenen Profilinformationen können Rückschlüsse auf den Hintergrund der Teilnehmer*innen gezogen werden. In Bezug auf die Forschungsfragen **R2** und **R3** untersucht der Fragebogen die Ausgangslage der Artefakte, den Ausgangspunkt für die Erstellung von Testfällen, den Automatisierungsgrad, die Testumgebung, den Testzweck, die Testfallerstellungsmethode und die aktuelle und mögliche geschätzte Testfallabdeckung.

5.2.1 Design und Durchführung der Studie

Die anonyme Umfrage umfasst 27 Fragen (13 geschlossene Fragen und 14 offene Fragen). Neben den Likert-Skalen, Multiple-Choice und nummerischen Fragetypen wurde ebenfalls Freitext eingesetzt. Alle Fragen mit vorgefertigten Antworten (Multiple-Choice) verfügen über eine Antwortoption „Sonst“ oder „Sonstiges“ in Verbindung mit einem Freitextfeld, um mögliche nicht aufgelistete Antworten abzufangen. Zusätzlich wurde Freitext verwendet, um die Umfrageteilnehmer*innen aufzufordern, ihre Antwort-Auswahl zu begründen. Somit wurde neben einem quantitativen auch ein qualitatives Feedback erhalten. Die Fragen sind in drei Abschnitte unterteilt und mit den Identifikatoren Q1 bis Q5, Q6 bis Q11c und Q12 bis Q19b versehen. Tabelle 5.4 zeigt die Fragen Q1 bis Q5 bezüglich des Hintergrunds der Teilnehmer*innen.

Tabelle 5.4: Fragen zum Hintergrund der Teilnehmer*innen, Fragetyp und Bezug zu den Forschungsfragen [DGH+19]

ID	Frage	Fragetyp	Zuordnung
Q1	Was ist Ihre Haupttätigkeit?	Multiple Choice	R1
Q2	Was haben Sie studiert? bzw. welchen Beruf haben Sie erlernt?	Multiple Choice	R1
Q3	Wie lange arbeiten Sie schon in der Automobilindustrie?	Nummerisch/Jahre	R1
Q4	Wie lange Arbeiten Sie schon im Bereich der Testfallerstellung?	Nummerisch/Jahre	R1
Q5	Was ist Ihr Testfokus?	Multiple Choice	R1, R3

Die Fragen Q1, Q2, Q3 und Q4 untersuchen den individuellen Hintergrund der Teilnehmer*innen, die in Bezug auf **R1** relevant sind. Die Frage Q5 untersucht den Testfokus. Die gewonnen Informationen aus Q5 sollen helfen den Standpunkt der Teilnehmer*innen zu

verstehen (**R1**) und geht der Frage des Zielsystems, wie Software, Komponente, Teilsystem oder Gesamtsystem (Fahrzeug) nach (**R3**). Der zweite Teil der Umfrage untersucht die wichtigsten Testfallartefakte und Testverfahren (siehe Tabelle 5.5).

Tabelle 5.5: Fragen zu den wichtigsten Testfallartefakten und -verfahren [DGH+19]

ID	Frage	Fragetyp	Zuordnung
Q6	Aus welchen Quellen leiten Sie Ihre Testfälle ab und zu wie viel Prozent leiten Sie die Testfälle der angegebenen Quellen ab?	Likert-Skala	R2
Q7	Mit welcher Ausgangslage erstellen Sie den Testfall?	Multiple Choice	R1, R2
Q8	Wie gehen Sie bei der Erstellung eines Testfalls vor?	Multiple Choice	R1, R2
Q8a	Erläutern Sie kurz die Erstellungsmethodik eines Testfalls:	Freitext	R1, R2
Q9	Welches Ergebnis erstellen Sie?	Multiple Choice	R1, R2
Q9a	Erläutern Sie kurz das Ergebnis:	Freitext	R1, R2
Q10	Welche Punkte kosten viel Zeit bei der Testfallerstellung?	Freitext	R2
Q11	Sind Ihre Testfälle automatisiert ausführbar?	Multiple Choice	R2
Q11a	Falls Ja oder in %: Wo bzw. wie sind diese Testfälle automatisiert ausführbar?	Multiple Choice	R3
Q11b	Falls Nein oder in %: Warum sind diese Testfälle nicht automatisiert?	Freitext	R2, R3
Q11c	Falls Nein oder in %: Was muss passieren, damit diese Testfälle automatisierbar werden?	Freitext	R2, R3

Um einen detaillierten Vergleich der Umfrageergebnisse hinsichtlich traditioneller und modellbasierter Testfallerstellung zu ermöglichen (vgl. Abbildung 5.3), untersucht die Umfrage die Verfahrensschritte und den Inhalt der Artefakte (Q6, Q7, Q8, Q9, Q11, Q11b, Q11c). Zusätzlich lassen sich die einzelnen Schritte bezüglich der subjektiv wahrgenommen Zeitkosten gegenüberstellen (Q10). Der Vergleich ermöglicht somit eine Aussage über Vor- und Nachteile der modellbasierten Testfallerstellung (**R1**) und der Qualität (**R2**) zu treffen. Außerdem lassen die Fragen Q7, Q8, Q8a, Q9 und Q9a Rückschlüsse auf den aktuellen Hintergrund der Teilnehmer*innen zu (**R1**). Darüber hinaus untersuchen die Fragen Q11b und Q11b die Testfallautomatisierung, mögliche Verbesserungen für Artefakte (**R2**) und Umgebungen (**R3**).

Tabelle 5.6 zeigt den dritten Teil der Umfrage. Der dritte Teil untersucht zusätzliche Themen, die nicht direkt mit dem Hintergrund und den Hauptartefakten des Testfalls in Verbindung gebracht werden konnten.

Insgesamt wurden 196 interne und externe professionelle Testingenieur*innen und

Tabelle 5.6: Teil drei der Fragen mit der Zuordnung zu den Fragetypen und Forschungsfragen [DGH+19]

ID	Frage	Fragetyp	Zuordnung
Q12	Geben Sie die Testfallabdeckung/Anforderungsabdeckung nach ihrem Wissen (Abschätzen) an?	Nummerisch/%	R2
Q13	Wie könnte die Testfallabdeckung/Anforderungsabdeckung verbessert werden?	Freitext	R2
Q14	Welche Abdeckung wäre Ihrer Meinung nach möglich? (in realistischer Zeit, Ressourcen, ...)	Nummerisch/%	R2
Q14a	Welche Ressourcen würde man hierfür benötigen?	Freitext	R2
Q15	Welches Absicherungsziel wird mit den von Ihnen erstellten Testfällen verfolgt?	Likert-Skala	R2
Q16	Was ist ein gültiger/ legitimer Testfall? (Ein Testfall der abgenommen wird)	Freitext	R2
Q17	In welchem Takt müssen Sie Testfälle liefern?	Multiple Choice	R1
Q18	In welchem Takt werden die Testfälle durchgeführt?	Multiple Choice	R1
Q19	Glauben Sie eine modellbasierte Testfallerstellung kann Sie sinnvoll unterstützen?	Likert-Skala	R1
Q19a	Begründen Sie Ihre Aussage:	Freitext	R1
Q19b	Welche Vorteile/Nachteile sehen Sie bei einer modellbasierten Testfallerstellung gegenüber der Testfallerstellung auf Basis von natürlichsprachlichen Anforderungen:	Freitext	R1

Testmanager*innen der BMW Group kontaktiert. Um die Wahrscheinlichkeit zu verringern, dass es sich bei den Ergebnissen um zufällige Ergebnisse handelt, wurde ein Signifikanzniveau von 5% festgelegt. Entsprechend diesem Niveau wurde eine erforderliche Stichprobengröße von 65 bei 196 kontaktierten Testingenieur*innen für ein statistisch signifikantes Ergebnis ermittelt. Anhand der Stichprobengröße kann zu 95% zuversichtlich davon ausgegangen werden, dass die beobachteten Ergebnisse wahr sind und keine Fehler enthalten, die durch Zufall[7] zustande gekommen sind. Ist die Stichprobengröße zu klein, zum Beispiel nur eine Person, könnte es sich um eine Testmanager*in handeln,

[7]Zufall bedeutet diesem Kontext, dass keine kausalen Zusammenhänge für ein einzelnes Ereignis oder das Zusammentreffen mehrerer Ereignisse bestehen.

die Testaktivitäten managend aber nicht aktiv Testfälle erstellt. Das zufällige Ergebnis wäre dann, das 100% der befragten Personen keine Testfälle erstellt aber Testaktivitäten koordiniert. Insgesamt wurden 70 der Fragebögen beantwortet mit 69 gültigen Fragebögen. Der ungültige Fragebogen ist ohne Inhalt.

5.2.2 Ergebnisse

Die Umfrage wurde durch eine externe Abteilung der BMW Group für Mediendienste und Online-Befragungen vom 23. August bis 22. September 2017 durchgeführt. Die Rohdaten der Befragungsergebnisse können jedoch aus Gründen der Vertraulichkeit nicht veröffentlicht werden. Die dargestellten Ergebnisse in Tabelle 5.7 verdeutlichen, dass tatsächlich mehr als 90% der Befragten in ihrer Haupttätigkeit mit Tests arbeiten (Q1). Andere gaben an, dass sie in ihrer Haupttätigkeit mit Anforderungen beschäftigt sind. Der

Tabelle 5.7: Antworten der Umfrage zum Hintergrund der Teilnehmer*innen mit dem Mittelwert ($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n) [DGH+19].

ID	**Frage** / Antwortoptionen	**Ergebnis**
Q1	**Haupttätigkeit:**	($n = 69$)
	Erstellung von Anforderungen:	8,96%
	Erstellung neuer Testfälle:	26,87%
	Automatisieren von Testfällen:	5,97%
	Testen/ Testfall-Durchführung:	13,43%
	Bewertung/ Analyse von Testergebnissen:	17,91%
	Testmanagement:	26,87%
Q2	**Beruflicher Hintergrund:**	($n = 68$)
	Elektrotechnik:	39,13%
	Informatik:	8,70%
	Maschinenbau:	23,19%
	Mechatronik:	15,94%
	Wirtschaftsingenieurwesen:	4,35%
	Sonst:	8,70%
Q3	**Jahre in der Automobilindustrie:**	($n = 69$)
	$\overline{x}$ und s in Jahren:	$\overline{x} = 10{,}16; s = 8{,}08$
Q4	**Jahre in der Testfallerstellung:**	($n = 66$)
	$\overline{x}$ und s in Jahren:	$\overline{x} = 4{,}33; s = 4{,}60$
Q5	**Testfokus?**	($n = 69$)
	Software:	15,94%
	Komponente:	24,64%
	Teilsystem:	17,39%
	Gesamtsystem (Fahrzeug):	42, 03%

berufliche Hintergrund der Teilnehmer ist in der Regel Elektrotechnik (Q2), gefolgt von Maschinenbau (Fahrzeugtechnik), Mechatronik, Informatik, Wirtschaftsingenieurwesen

und verwandten Studiengängen in den Fächern wie Mathematik, Chemie oder Wirtschaftswissenschaften. Die Teilnehmer*innen gaben an, seit einer Zeitspanne von 0,5 bis 41 Jahren in der Automobilindustrie zu arbeiten (Q3). In Bezug auf die Testfallerstellung lagen die Antworten in einem Bereich von 0,5 bis 20 Jahren (Q4). Aufgrund der großen Spannbreite in der Erfahrung in Jahren ähnelt jeder Mittelwert ($\overline{x}$) von Q3 und Q4 seiner zugehörigen Standardabweichung (s). Zum größten Teil konzentrieren die Teilnehmenden ihren Fokus (Q5) der Testumgebung auf das komplette Fahrzeug (42,03%). Die anderen Teilnehmer*innen konzentrieren sich bei der Testfallerstellung auf die Komponenten (24,64%), Teilsysteme (17,39%) oder Software (15,94%).

Laut Umfrage sind die wichtigsten Ressourcen für die Erstellung von Testfällen (Q6) die textuellen Anforderung in natürlicher Sprache, gefolgt von persönlichen Erfahrungen, vorhandenen Testfällen, Fehlerbeschreibungen und UML- und SysML-Modellen (Abbildung 5.8).

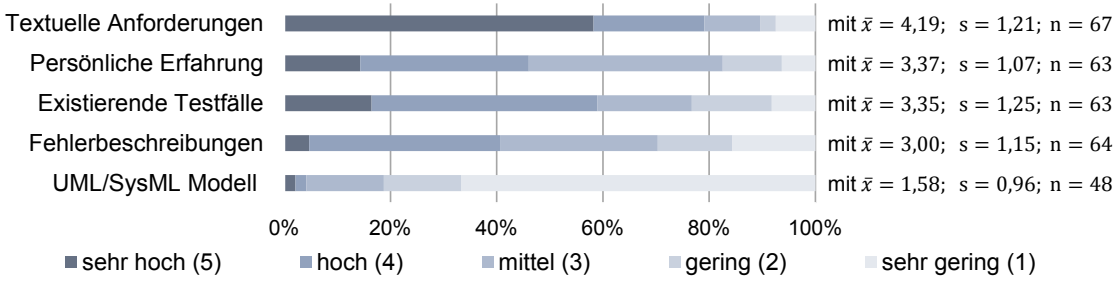


Abbildung 5.8: Antworten auf die Frage Q6, „Aus welchen Quellen leiten Sie Ihre Testfälle ab und zu wie viel Prozent leiten Sie die Testfälle der angegebenen Quellen ab?“ mit dem Mittelwert ($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n) [KMS+18].

Auf die Frage Q7 gaben 76,56% der Befragten an, dass Anforderungen als Basis für die Testfallerstellung dienen (Tabelle 5.9). Der Testfallentwurf mit 14,06% und die Testfallspezifikation[8] mit 9,38% spielen eine deutlich geringere Rolle. In Bezug auf das Testfallergebnis (Q9) gaben 51,67% der Befragten an, einen Testfallentwurf zu erstellen, während 11,67% und 36,67% der Befragten eine Testfallspezifikation beziehungsweise einen automatisierten Testfall produzieren.

Die „Rohbeschreibung des Testfalls“ aus Q7 entspricht in dieser Arbeit einem Testfallentwurf. Das „Grundgerüst des Testfalls“ aus Q7 und das „Rohe Grundgerüst des Testfalls“ aus Q9 entsprechen in dieser Arbeit der Testfallspezifikation. Um das gängigste Szenario zu ermitteln, wurden Angaben der Teilnehmer*innen bezüglich der Ausgangsla-

[8] Für die Umfrage wurde das Testkonzept bewusst entweder den Anforderungen oder dem Testfallentwurf zugeordnet. Das Testkonzept ist nicht ein Teil der Testfallerstellungskette „Testfallentwurf → Testfallspezifikation → automatisierter Testfall“ (vgl. Abschnitt 3.3), sondern umfasst eine Anforderungsanalyse und Anforderungsbewertung. Für ein besseres Verständnis aus Gründen der Eindeutigkeit wurde in Q7 das Testkonzept mit dem Punkt „Anforderungen“ abgedeckt. Im Kontext der BMW Group umfasst das Testkonzept in einigen Bereichen die Erstellung von Testfallentwürfen. Insofern wurde das Testkonzept in Q9 dem Testfallentwurf zugeordnet.

Tabelle 5.9: Antworten auf die Fragen bezüglich der Artefakte der Teilnehmer*innen mit dem Mittelwert ($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n) [DGH+19].

ID	**Frage** / Antwortoption	**Ergebnis**[9]
Q7	**Mit welcher Ausgangslage erstellen Sie den Testfall?**	($n = 64$)
	Anforderungen:	76,56%
	Rohbeschreibung des Testfalls:	14,06%
	Grundgerüst des Testfalls:	9,38%
Q8	**Wie gehen Sie bei der Erstellung eines Testfalls vor?**	($n = 64$)
	Definierte Methode (Systematisch):	56,25%
	Von Testfall abhängig:	40,63%
	Automatisch generiert:	3,13%
Q9	**Welches Ergebnis erstellen Sie?**	($n = 62$)
	Konzept/Entwurf eines Testfalls:	51,67%
	Rohes Grundgerüst des Testfalls:	11,67%
	Automatisierter Testfall:	36,67%
Q11	**Sind ihre Testfälle automatisiert ausführbar?**	($n = 67$)
	Ja:	34,34%
	Nein:	10,44%
	zu...% ($\overline{x} = 68\%; s = 26\%$):	55,22%
Q11a	**Wo/wie sind diese Testfälle automatisiert ausführbar?**	($n = 60$)
	HIL (Hardware in the Loop):	81,67%
	MIL (Model in the Loop):	1,67%
	SIL (Software in the Loop):	5,00%
	VIL (Vehicle in the Loop):	11,66%

[9] Die Stichprobengröße (n) in Tabelle 5.9 der Fragen Q7 mit $n = 64$, Q8 mit $n = 64$, Q9 mit $n = 62$ und Q11a mit $n = 60$ stimmt nicht mit den Stichprobengrößen von [DGH+19] mit $n = 69$ (Q7), $n = 69$ (Q8), $n = 69$ (Q9) und $n = 68$ (Q11a) überein. Hierbei handelt es sich um einen Bearbeitungsfehler der beim Übertrag der automatisch ausgewerteten Daten in [DGH+19] nicht festgestellt wurde und ansonsten behoben worden wäre. Es handelt sich nach [Sta20] um einen geringen inhaltlichen Fehler, deren Größenordnung so gering ist, dass die Aussage unverändert bleibt. In dieser Arbeit wurden die festgestellten Fehler korrigiert veröffentlicht.

ge der Testfallerstellung (Q7) und dem Ergebnis der Testfallerstellung (Q9) verknüpft. Abbildung 5.10 veranschaulicht den Zusammenhang der Artefakte der Ausgangslage und des Ergebnisses.

Ein Anteil von 26,56% der Teilnehmer*innen beginnt mit Anforderungen und liefert einen automatisierten Testfall. Ob dieser Prozess direkt oder mit den Zwischenschritten, Testkonzept/Testentwurf und Testfallspezifikation ausgeführt wird, lässt sich aus den Daten nicht erschließen. Ein weiterer Teil (9,37%) beginnt mit Anforderungen und erstellt Testfallspezifikationen. Auch hier können eventuelle Zwischenschritte nicht nachvollzogen werden. Der größte Teil, mehr als ein Drittel, der Teilnehmer*innen gab an mit den Anforderungen ein Testkonzept beziehungsweise Testfallentwürfe zu erstellen. In Bezug

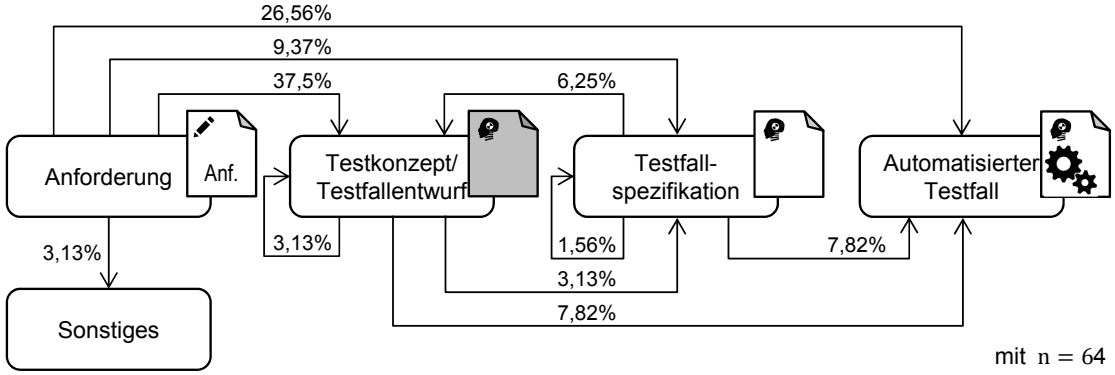


Abbildung 5.10: Zusammenhang zwischen den Ergebnissen aus Q7 und Q9 mit Stichprobengröße (n).

auf die Ausgangslage und das erstelle Ergebnis gaben 3,13% der Befragten an, aus einem Testkonzept/Testfallentwurf eine Testfallspezifikation zu entwickeln, während 7,82% einen automatisierten Testfall und 3,13% wieder ein Testkonzept/Testfallentwurf erstellen. Diese erneute Erstellung kann zum eine damit zusammenhängen, dass aus dem Testkonzept ein neuer Testfallentwurf konstruiert oder eine Überarbeitung der Artefakte vorgenommen wird. Die Testfallspezifikation überarbeiten 1,56% der Umfrageteilnehmer*innen und 7,82% erstellen einen automatisierten Testfall. Ein Anteil von 6,25% gab an, aus der Testfallspezifikation ein Testkonzept/Testfallentwurf zu erstellen. Dieses Szenario könnte mehrere Gründe haben. Der eindeutig vorherrschende Grund ist, dass die Fragen aus Q7 und Q9 nicht eindeutig gestellt und infolgedessen Raum für Interpretationen lassen (Weiteres dazu in Unterabschnitt 5.2.4). Eine weitere Möglichkeit ist, dass auf der Basis von Testfallspezifikationen das Testkonzept oder die Testentwürfe überarbeitet werden, da es neue Erkenntnisse aus der Praxis gibt. Die Durchsicht der Kommentare dieser Szenarien lieferten ebenfalls keine Rückschlüsse. Außerdem wählten 3,13% eine Antwortoption „Sonstiges“ ohne eine Beschreibung[10]. Bei der Erstellung von Testfällen (Q8) gaben mehr als die Hälfte (56,25%) der Umfrageteilnehmer*innen an, dass sie systematisch nach einer definierten Methode vorgehen. Etwas weniger als die Hälfte (40,63%) macht ihr Vorgehen bei der Testfallerstellung abhängig von dem jeweiligen Testfall. Die automatische Generierung von Testfällen spielt mit 3,13% eine geringe Rolle.

Die Freitext-Antworten auf die Frage (Q10) wurden bei der Analyse in die Kategorien: Anforderungen, Testfallerstellung, Datenermittlung, Prozess und Logistik, und Sonst zusammengefasst (Abbildung 5.11). Dem Bereich der Anforderungen und die daraus resultierenden Rücksprachen konnten 65,38% der Antworten zugeordnet werden. Insbesondere die Klärung der Anforderung aufgrund der Qualität, wie Mehrdeutigkeiten und Ungenauigkeiten wurden von den Teilnehmenden erwähnt. Der tatsächlichen Tätigkeiten

[10] Die Antwortoption „Sonstiges“ lässt keine Aussage über die Trends der Ausgangslage und des Ergebnisses zu. Aus diesem Grund wurden die einzelnen „Sonstiges“-Antworten nicht in der Tabelle 5.9 aufgenommen.

der Testfallerstellung, wie dem Testfallentwurf, der Testfallspezifikation und der Testfallautomatisierung konnten 21,15% der Antworten zugeordnet werden. In dieser Kategorie wurden neben diesen einzelnen Schritten auch die Fehlerbehebung bereits bestehender Testfälle angemerkt. Die Datenermittlung, wie die Suche nach Signallabels oder anderen notwendigen Daten empfanden 9,62% der Teilnehmenden als zeitaufwendig. Den Prozess und die Logistik, wie den Prüfstandsaufbau, formale Vorgaben, das „organisatorische Drumherum“ und Verknüpfung der Testfälle mit den Anforderungen gaben 7,69% als zeitaufwendig an. Der Rest der Teilnehmenden (5,77%) wussten keine Antwort oder erstellen aktuell keine Testfälle.

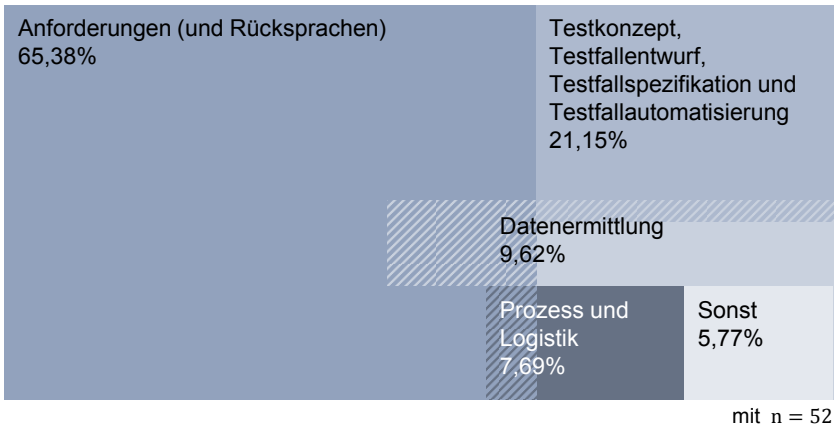


Abbildung 5.11: Antworten auf die Frage Q10, „Welche Punkte kosten viel Zeit bei der Testfallerstellung?“ mit der Stichprobengröße (n). Schraffierte Bereiche weisen auf Überschneidungen hin.

Auf die Frage zu den automatisierten Testfällen (Q11) gab ein Drittel der Teilnehmenden an, dass die Testfälle automatisiert ausführbar sind. Ein Zehntel gab an, dass die Testfälle nicht automatisiert sind und mehr als die Hälfte gab an, dass die Testfälle zum Teil ausführbar sind. Von diesen automatisierten Testfällen wird der größte Teil (81,67%) für HIL-Verfahren entwickelt. Entsprechend der Ergebnisse zu Q5 stehen MIL mit 1,67% und SIL mit 5,00% weniger im Fokus der Teilnehmer*innen. Im Gegensatz zum Testfokus der Testumgebung (Q5) ist das automatisierte Verfahren VIL mit dem Fahrzeug als weniger wichtig angegeben (11,66%). Im Fall Q11b, warum diese Testfälle nicht automatisiert seien, wurde generell geantwortet, dass das Gesamtsystem aufgrund der teilweise hohen Dynamik und den vielen Umwelteinflüssen schwer zu automatisieren sei. Für eine Automatisierung müssten die Automationssoftware und Prüfstände für eben diese dynamischen Umwelteinflüsse ausgestattet werden, die der Ergebnisinterpretation beziehungsweise Fehlerinterpretation menschlicher Intelligenz gleichen und somit den Fahrer ersetzen können (Q11c).

Um mögliche Verbesserungen der Testabdeckung zu untersuchen, wurden die Teilnehmer*innen zur aktuellen und möglichen Testabdeckung befragt (Abbildung 5.12). Aus den Daten der Schätzungen geht hervor, dass derzeit mit einem Mittelwert von 67,4%

und einer Standardabweichung von 25,34% die gegenwärtige Testabdeckung etwa 20% niedriger eingeschätzt als die für möglich geschätzte Testabdeckung, mit einem Mittelwert von 87,55% und die Standardabweichung von 14,23%. Folglich ist nach der Einschätzung der Teilnehmer*innen eine mittlere Verbesserung der Testabdeckung von 20% möglich.

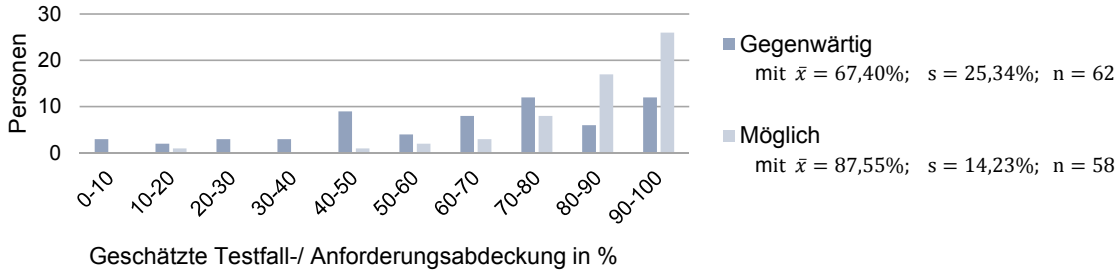


Abbildung 5.12: Ergebnisse von Q12 und Q14: Geschätzt gegenwärtige und geschätzt mögliche Testabdeckung mit dem Mittelwert ($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n) [KMS+18].

Aus den Freitextkommentaren von Q13 konnten die Kategorien Ressourcen, Anforderungen und Sonstige identifiziert werden (Abbildung 5.13). Der Prozentsatz (11,9%) der ungültig gekennzeichneten Antworten waren mit „-“ ausgestrichen oder enthielten Zahlen. Ein Anteil von 61,9% der Teilnehmer*innen gab an, dass sie mehr Ressourcen wie Zeit, Fachpersonal oder Zugang zu Testumgebungen benötigen. Im Zusammenhang mit dem Fachpersonal wurde unter anderem angemerkt, dass diese sich intensiv mit der Anforderungsanalyse auseinandersetzen sollten. Die Qualität der Anforderungen wurde, wie in der Frage Q10, bemängelt. In der Kategorie Sonst wurde vor allem eine bessere Vernetzung und Zusammenarbeit aller Domänenexpert*innen und ein höherer Automatisierungsgrad angemerkt.

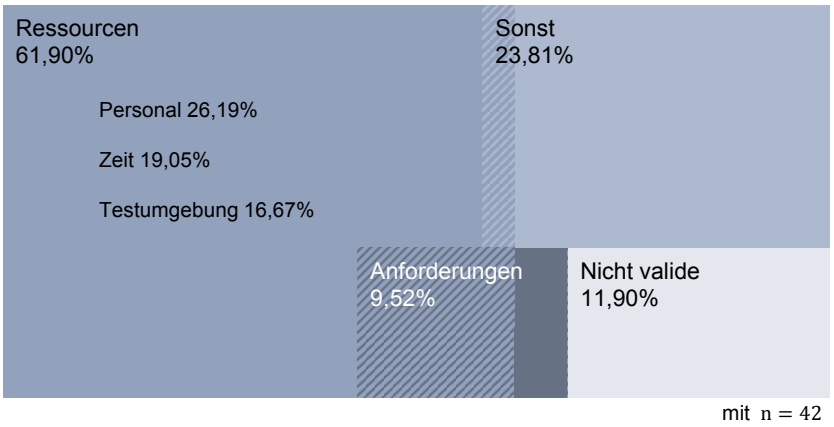


Abbildung 5.13: Antworten auf die Frage Q13, „Wie könnte die Testfallabdeckung/Anforderungsabdeckung verbessert werden?“ mit Stichprobengröße (n).

Darüber hinaus wurde mit der Frage Q15 der Qualitätsfokus untersucht. Aus Abbildung 5.14 ist ersichtlich, dass die Funktionalität als wichtigster Faktor für das Absicherungsziel der Qualitätskriterien angegeben wird. Ein weiterer Fokus ist die Robustheit mit einem ähnlichen Mittelwert wie Zuverlässigkeit und Funktionssicherheit. Danach folgen die Zuverlässigkeit und die funktionale Sicherheit. Die Qualitätskriterien der Leistung(-sfähigkeit), Wiederverwendbarkeit, Integrierbarkeit, Effizienz/Sparsamkeit und Wartbarkeit werden als weniger wichtig angegeben. In Bezug auf die Validität der Testfälle

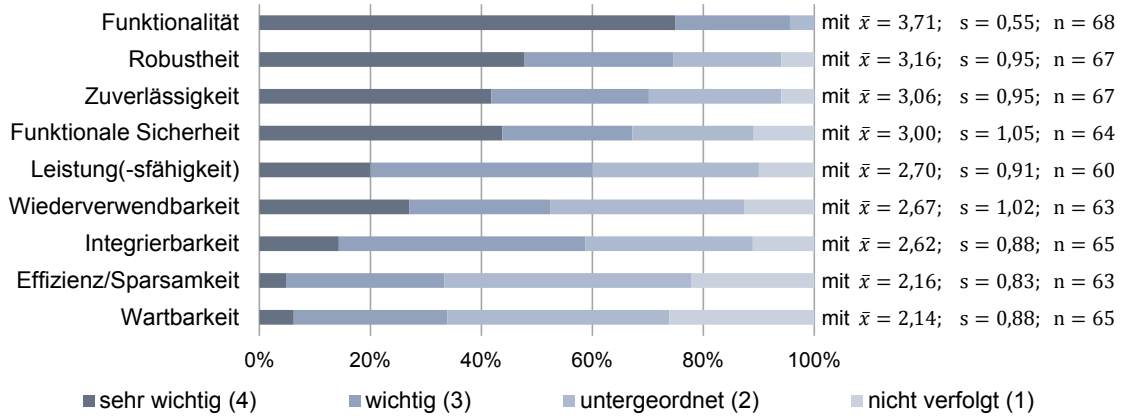


Abbildung 5.14: Ergebnisse zu Q15 „Welches Absicherungsziel wird mit den von Ihnen erstellten Testfällen verfolgt?“ mit dem Mittelwert($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n) [DGH+19].

(Q16) gaben 95,35%[11] von $n = 42$ Teilnehmer*innen an, dass der Testfall bestimmte Kriterien erfüllen muss. Zu diesen Kriterien gehören unter anderem die sinnvolle Prüfung einer aktuellen Anforderung, ein vorgegebenes Testfallformat (Struktur und Automation), Verständlichkeit und Eindeutigkeit bis hin zu einer initialen Durchführung auf der entsprechenden Testumgebung.

Um die Häufigkeit des Erstellungs- und Testprozesses zu untersuchen, wurden mit den Frage Q17 und Q18 der Takt der Testfallerstellung und -durchführung abgefragt (Tabelle 5.15). Die Frage Q17 zielt darauf ab, wie oft Testfälle erzeugt werden und Q18 gibt an, wie oft diese zuvor erzeugten Testfälle dann durchgeführt werden. Die Ergebnisse zeigen deutlich, dass Testfälle hauptsächlich in Intervallen, die länger als ein Monat dauern (Meilenstein) oder wöchentlich, für Sprints bis zu 3 Wochen, angefertigt werden. Diese Ergebnisse decken sich auch mit der Durchführung der Testfälle, obwohl die Meilensteine mit 77,19% eine größere Bedeutung zugesprochen werden kann. Die monatlichen, täglichen und einmaligen Takte spielen eine weitaus geringere bis gar keine Rolle bei der Erstellung und Durchführung der Testfälle.

Zusätzlich wurden die Teilnehmer*innen in Q19 direkt nach dem erwarteten Nutzen der modellbasierten Testfallerstellung befragt (Abbildung 5.16). Ein Anteil von 7,27% der Teilnehmer*innen erwartet sich eine sehr große Hilfe von der modellbasierten Test-

[11] Die restlichen 4,65% waren mit „-“ ausgestrichen

Tabelle 5.15: Antworten auf die Fragen bezüglich des Taktes der Testfallerstellung (Q17) und der Durchführung der Testfälle (Q18) mit der Stichprobengröße (n) [DGH$^+$19].

ID	**Frage** / Antwortoption	**Ergebnis**
Q17	**In welchem Takt müssen Sie Testfälle liefern?**	($n = 58$)
	Meilensteine (Intervalle länger als ein Monat):	46,30%
	Monatlich:	5,56%
	Wöchentlich (Sprints bis zu 3 Wochen inklusive):	35,19%
	Täglich:	3,70%
	Einmalig:	9,25%
Q18	**In welchem Takt werden Testfälle durchgeführt?**	($n = 61$)
	Meilensteine (Intervalle länger als ein Monat):	77,19%
	Monatlich:	3,51%
	Wöchentlich (Sprints bis zu 3 Wochen inklusive):	12,28%
	Täglich:	7,02%
	Einmalig:	0,00%

fallerstellung und 38,18% versprechen sich eine Unterstützung bei der Arbeit. Neutral stehen 36,36% der modellbasierten Testfallerstellung gegenüber und 18,18% sehen in dieser Herangehensweise einen Mehraufwand beziehungsweise eine Behinderung. Für die statistische Datenanalyse wurde eine gewöhnliche numerische Skala mit einer Likert-Skala versehen, von einer (die modellbasierten Testfallerstellung behindert meine Arbeit) bis zu fünf (modellbasierten Testfallerstellung hilft mir sehr). Angesichts eines Mittelwerts ($\overline{x}$) von 3,29 kann von einer leicht positiven Erwartung der modellbasierten Testfallerstellung ausgegangen werden.

Diese positive Erwartung der Teilnehmenden wird zudem mit einer Mehrzahl an positiver Erklärungen aus Q19a und Q19b untermauert. Die Freitextkommentare von Q19a und Q19b sollen vor allem das Verständnis der Indikationen in Q19 erleichtern. Da sich die Kommentare aus Q19a und Q19b zum Teil ergänzen, gegenseitig auf sich verweisen und wiederholen, wurden die Antworten zusammen betrachtet. Die Antworten aus Q19a und Q19b konnten für eine Übersicht in die Kategorien „Leer/ Weiß nicht“, „Aufwand“, „Qualität“, „Grafisches Modell reicht nicht aus“ und „Effizienz“ zugeordnet werden. Eine Mehrfachzuordnung ist möglich. Diese Kategorien wurden aus Gründen der Nachvollziehbarkeit mit den Ergebnissen aus Q19 in Zusammenhang gebracht (vgl. Tabelle 5.17).

Ein großer Anteil von 54,55% der Teilnehmer*innen kommentierten gar nicht oder wollten keine Aussage treffen, da das nötige Hintergrundwissen oder konkrete Beispiele fehlten. In der Kategorie Aufwand wurden vor allem die Themen der Wartbarkeit bezüglich des Modells und des initialen Wissensaufbaus erwähnt. Der große Teil dieser Kategorie vertritt die Meinung, dass der Aufwand in keiner Relation zum Nutzen steht. Wiederum andere wiegten ab und sahen einen Aufwand bei der Einarbeitung, der Prozessumstellung und dem Tooling, der sich aber letztendlich auszahlt. Bezüglich der Qualität versprechen sich die Kommentierenden insbesondere eindeutige und verständliche Darstellungen von Funk-

Abbildung 5.16: Antworten auf die Frage Q19 bezüglich des zu erwartenden Nutzens der modellbasierten Testfallerstellung mit dem Mittelwert ($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n) [KMS+18].

Tabelle 5.17: Zusammenhang der Antworten aus Begründungen (Q19a), der Vor- und Nachteile (Q19b), und den Ergebnissen aus Q19 (vgl. Abbildung 5.16) mit dem Mittelwert ($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n) basierend auf der Likert-Skala von Q19 und explizit in Q19b ausgeschriebene Vorteile ($+_{Q19b}$) und Nachteile ($-_{Q19b}$).

Kategorie	$\overline{x}$	s	n	$+_{Q19b}$	$-_{Q19b}$
Leer/ Weiß nicht	3,17	0,60	36	–	–
Aufwand (Einarbeitung, Wartbarkeit)	3,29	0,88	14	3	10
Qualität (Eindeutigkeit, Verständlichkeit)	3,80	0,60	10	9	1
Grafisches Modell reicht nicht aus	2,33	1,05	9	0	4
Effizienz (Automatisierbarkeit, Einheitlichkeit)	4,38	0,48	8	7	0

tionen und somit bessere und testbare Anforderungen beziehungsweise Spezifikationen. Ein Bedenken ist jedoch das Risiko, dass Anforderungen nicht mehr klar beschrieben werden. Diese Befürchtung findet sich auch im Hauptkritikpunkt der negativen Assoziationen aus Q19 wieder, dass ein (grafisches) Modell nicht ausreicht. Als Gründe im Freitextfeld wurde angegeben, dass die natürlichsprachlichen textuellen Anforderungen zu komplex seien, die Erfahrung für erfahrungsorientierte Tests nicht in den Modellen hinterlegt ist und keine Fehlerpfade im Modell abgebildet werden. Im Zusammenhang mit positiven Assoziationen zu Q19 wurde neben der Qualität die (potenzielle) Effizienz genannt. Es wurde kommentiert, dass die modellbasierte Testfallerstellung eine Automatisierbarkeit der Testfallerstellung und weitere unterstützende Prozesse ermöglichen kann, welche die Effizienz steigern können. Außerdem Versprechen sich die Kommentierenden eine einheitliche Herangehensweise und klare Strukturen bei der Testfallerstellung.

Tabelle 5.18: Korrelation zwischen dem Hintergrund (R1) und dem erwarteten Nutzen der modellbasierten Testfallerstellung (Q19) [KMS+18].

ID	Hintergrund	r	p	n
Q1	Haupttätigkeit	−0,05	0,371	53
Q2	Fachhintergrund	0,03	0,421	55
Q3	Erfahrung in der Automobilindustrie	0,19	0,088	55
Q4	Erfahrung in der Testfallerstellung	0,10	0,240	55

Zusätzlich zu den Kommentaren Q19a und Q19b wurde der Aspekt einer möglichen Korrelation zwischen dem persönlichen Hintergrund und den Erwartungen bezüglich der modellbasierten Testfallerstellung untersucht. Zu diesem Zweck wurden die Frage Q19 mit dem Profil der Teilnehmenden (R1) verknüpft. Initial wurde davon ausgegangen, dass die Meinung über den Nutzen der modellbasierten Testfallerstellung nicht von ihrem Hintergrund beeinflusst wird. Tabelle 5.18 stellt die Ergebnisse anschaulich dar. Die Korrelationskoeffizienten (r) der Fragen Q1 und Q2 sind niedriger als die Korrelationskoeffizienten der Fragen Q3 und Q4. Darüber hinaus sind die p-Werte für Frage Q1 und Frage Q2 höher. Die Korrelationskoeffizienten sind niedriger als ±0,20, was die Hypothese bekräftigt, dass nur ein schwacher Zusammenhang zwischen der Meinung der Teilnehmer*innen (Abbildung 5.16) und dem Hintergrund besteht (Tabelle 5.7). Allerdings deuten die p-Werte auf statistisch nicht signifikante Ergebnisse hin.

5.2.3 Fazit der Umfrage

In der Entwicklung von einem CPS, wie dem heutigen Automobil, ist eine interdisziplinäre Zusammenarbeit (in der Softwareentwicklung) unumgänglich [BCL+21, LTK19]. Im Rahmen dieser Umfrage in der Automobilindustrie [KMS+18, DGH+19] konnte diese Interdisziplinarität im Bereich der Testfallerstellung bestätigt werden (**R1**). Der akademische Hintergrund der Testingenieur*innen deckt sich mit anderen Studien [SVWH11, ADS18] und ist der Umfrage zufolge verstärkt Elektrotechnik, gefolgt von Maschinenbau, Mechatronik, Informatik und anderen Bereichen (**R1**). Anhand des Korrelationskoeffizienten wird davon ausgegangen, dass die positive Wahrnehmung gegenüber der modellbasierten Testfallerstellung unabhängig von Erfahrungsniveau, beruflichem Hintergrund oder Prüfungsschwerpunkt ist (Tabelle 5.18). Trotz der Tatsache, dass es noch keinen weitverbreiteten modellbasierten Ansatz gibt, erwartet mehr als ein Drittel der Teilnehmer*innen, dass sie von einer modellbasierten Testfallerstellung profitieren (vgl. Abbildung 5.16). Diese positiven Erwartungen und Effekte decken sich auch mit denen aus [LMT+18].

Wie auch schon in anderen Umfragen wie [SVWH11, LTK+18] ermittelt, stützen sich die Prüfaktivitäten funktionsorientierter Tests auf die Basis der natürlichsprachlichen textuellen Anforderungen (vgl. Abbildung 5.8). Des Weiteren zeigt die Auswertung, dass ein erheblicher Teil der Teilnehmer*innen ihre persönliche Erfahrung, alte Testfälle und Fehlerbeschreibungen als Grundlage für die Erstellung von Testfällen heranziehen. Wie

auch schon in anderen Studien beobachtet wurde, spielt die (automatisierte) Testfallerstellung in der Industrie, auf der Basis eines grafischen Modells, eine untergeordnete Rolle [AWS14, KPM13, SVWH11, GJM+10, Bro06]. Ebenfalls kann der geringe Anteil an grafischen Modellen an der Zielgruppe der Umfrage liegen. Die Zielgruppe der Testingenieur*innen verwenden traditionell Anforderungsspezifikationen in natürlicher textueller Sprache, während Softwareentwickler*innen mit Modellen arbeiten. Vor allem bei Änderungen der textuellen Anforderungen, werden existierenden Testfälle herangezogen und angepasst. Dies kann zu fehlender Transparenz führen, welcher Testfall welche Anforderung abdeckt. SMArDT aus Kapitel 4 begegnet diesem Umstand mittels einer modellgetriebenen Entwicklung (vgl. Abschnitt 4.4). Anstatt nur vorhandene Testfälle anzupassen, werden die neuen Testfälle mit dem Modell verknüpft. Daher sind Änderungen an Testfällen leicht zu verstehen und nachzuvollziehen. Ebenfalls sollen die implizit gewonnenen Erfahrungen der Tester*innen in das Modell fließen, sodass auch andere Domänen, Tester*innen und zukünftige Projekte von dem expliziten Wissen profitieren können. Infolgedessen ist der Prozess weniger anfällig für volatiles Fachwissen, das durch persönliche Änderungen verursacht wird, und stärkt darüber hinaus die Testfallerstellung und stützt die Hypothese (**R2**). Die Fehlerfälle sollten laut SMArDT ebenfalls im Modell enthalten sein und erweitern folglich die traditionellen natürlichsprachlichen Anforderungen (vgl. Abschnitt 4.4), was ebenfalls die Hypothese (**R2**) unterstützt. Laut Umfrage befinden die Teilnehmer*innen, dass die Testabdeckung mit MBT um etwa 20% verbessert werden könnte. Besonders im Bereich der funktionsorientierten Testfallerstellung (**R2**), ist aufgrund des angegebenen Fokus die potenziell größtmögliche Unterstützung zu erwarten (vgl. Abbildung 5.14). Dieser Fokus auf die Funktionalität des Systems wird vor allem in der Ebene B von SMArDT dargestellt (vgl. Abschnitt 4.1).

Damit sich der von den Teilnehmer*innen erwartete hohe Aufwand bei der Einarbeitung und Wartung des Modells lohnt, sollten effizienzsteigernde Maßnahmen eingeführt werden. Beispiele für effizienzsteigernde Maßnahmen sind die automatisierte Erstellung von Testfällen oder automatische Überprüfungen, ob und wann das Testen von Modelländerungen relevant ist (vgl. Tabelle 5.17). Durch die Automatisierung wäre zudem der Zeitbedarf für die Testfallerstellung nicht mehr primär von der Ressource Mensch abhängig, sondern von der Rechnerkapazität. Infolgedessen könnten die eingesparten Ressourcen wie Personen (vgl. Abbildung 5.13) oder Zeit für die Klärung von Anforderungen (vgl. Abbildung 5.11) an anderer Stelle genutzt werden. Theorie und Praxis 5.2 gibt Erfahrungen mit nicht eindeutigen Anforderungen aus der Paxis wieder.

Theorie und Praxis 5.2 (Eindeutige und formale Anforderungen in der Industrie)**.** *Der erhöhte Zeitaufwand für die Klärung von Anforderungen ist unter anderem darin begründet, dass die Anforderungen sich häufig ändern [TTS+19], nicht formal sind und insofern für gewöhnlich einen zu klärenden mehrdeutigen Interpretationsspielraum offenlassen [LTK19]. Ein Grund für die nicht formalen Anforderungen und die geringe Bedeutung von MBT in der Automobilindustrie ist dass (formale) Ansätze in der Regel nicht in den Entwicklungsprozess integriert sind [DSVT07]. Dies ist unter anderem zwei Aspekten geschuldet und zwar der enormen (organisatorischen) Komplexität des Systems und der absichtlichen Trennung von Entwicklung und V&V aufgrund des Vieraugenprin-*

zips [Wit16]. Besonders in der Testphase (vgl. Abbildung 3.3) werden Prüfaktivitäten an externe Abteilungen, Teams oder Dienstleister vergeben [AWS14].

Das angegebene Potenzial für die Testfallabdeckungen zeigt (vgl. Abbildung 5.12), dass die modellbasierte Testfallerstellung in der Lage sein sollte, die Testfallqualität zu verbessern und die Hypothese **R2** weiter zu konsolidieren.

Die Umfrage ergab ebenfalls, dass alle hierarchischen Dekompositionsschichten von SMArDT vertreten sind (**R3**). Im Fokus steht das gesamte Fahrzeug, gefolgt von Komponenten, Subsystemen und Software (vgl. Tabelle 5.4). Die automatisiert ausführende Testfallumgebung wird stark dominiert von HIL und VIL (vgl. Tabelle 5.5). Aus den Ergebnissen lässt sich schließen, dass Systemtests, Subsystemtests und Komponententests die vielversprechendsten Umgebungen für eine automatisierte modellbasierte Testfallerstellung in dieser Domäne sind (**R3**). Die geringen Werte für SIL und MIL in der Umfrage sind wahrscheinlich in der Zusammensetzung der Zielgruppe der Teilnehmenden begründet. Die Zielgruppe der Umfrage sind Testingenieur*innen und keine Softwareentwickler*innen, die in der Regel SIL- und MIL-Tests durchführen.

5.2.4 Validität der Studie

Um eine möglichst große Anzahl an externen Testingenieur*innen zu erreichen, wurde die Umfrage per E-Mail an die Teilnehmer*innen verteilt und weitergeleitet. Diese Tatsache könnte die Validität der Umfrage allerdings gefährden. Denn obwohl die Teilnehmer*innen durch ihre Spezialisierung im Rahmen der Testfallerstellung ausgewählt wurden, kann eine Gefahr von Fehlinterpretationen und doppelten Einreichungen nicht ausgeschlossen werden.

Da die Stichprobenzahl der Umfrage begrenzt ist, unterliegen die Ergebnisse der Studie sicherlich einer gewissen Ungenauigkeit. Ein Beispiel für die Auswirkung ist, dass die Automatisierung von Testfällen in der Regel an externe Dienstleister*innen vergeben wird. Für den Fall, dass alle externen Testingenieur*innen einbezogen würden, könnten die Werte in Tabelle 5.5 anders verteilt sein. Häufig werden aus Gründen der Wahrung der Unabhängigkeit der Entwicklung und des Testens (Vieraugenprinzip), externe Unternehmen hinzugezogen. Da keine höhere Anzahl an Teilnehmer*innen erreicht wurde, besteht die Möglichkeit, dass Ergebnisse anders ausfallen könnten. Beispielsweise könnten die Fragen Q12 und Q14, trotz des Anhangs „(in realistischer Zeit, Ressourcen,...)“, als Leistungsbeurteilung wahrgenommen werden und infolgedessen zu angepassten Schätzungen führen. Des Weiteren deuten die p-Werte Tabelle 5.18 auf keine statistische Signifikanz hin, was die Bedeutung dieser Ergebnisse mindert. Es kann mit einer Sicherheit von 95% gesagt werden, dass die Fehlermarge für das Stichprobenverfahren und deren Ergebnisse nicht mehr als ± 20 Teilnehmer*innen bei einer Gesamtstichprobe von 70 Teilnehmer*innen in dieser Umfrage beträgt. Allerdings unterschritt die Anzahl der Antworten einzelner Fragen die erforderliche Stichprobengröße von 65. Aufgrund der Fehlermarge und des der Umstandes, dass nicht alle Teilnehmer*innen alle Fragen beantwortet haben, sowie der Tatsache, dass die Ergebnisse statistisch unbedeutend sind, kann nur von Tendenzen ausgegangen werden.

Neben den Gefahren für die interne Validität ergeben sich auch Gefahren für die Konstruktvalidität durch den Einsatz einer Online-Umfrage. Auch wenn die Fragen überwiegend mit wertneutralen Antwortoptionen und freien Kommentarfeldern („Sonst“ oder „Sonstiges“) versehen sind, kann eine suggestive Wirkung der Fragen nicht ganz ausgeschlossen werden. Beispielsweise wurde, um eine konsequente Aussage zu erhalten, die Frage Q11 nur mit Auswahloption „Ja“, „Nein“ und ohne die neutrale Auswahloption „weiß nicht“ ausgestattet. Den Teilnehmenden blieb somit, bis auf die Option „weiß nicht“, allein die Möglichkeit der wertenden Auswahloptionen. Ein anderes Beispiel sind die Fragen Q7 und Q9 mit Multiple-Choice Optionen. Um Missverständnisse aufgrund unterschiedlicher Begriffsdefinitionen zu vermeiden, wurden die Artefakte kurz umschrieben. Abbildung 5.10 verdeutlicht das potenzielle Missverständnis aufgrund eines Interpretationsspielraumes von „Rohbeschreibung“, „Grundgerüst“ und „Rohes Grundgerüst“ eines Testfalls.

5.3 Verwendungszweck und Erwartungen an die Testfälle

Zusammen mit den in Abschnitt 5.1 beschriebenen Anforderungen an die Spezifikationsmodelle und den Ergebnissen aus der Umfrage in Abschnitt 5.2 werden in diesem Abschnitt der Verwendungszweck und die identifizierten Erwartungen an die Testfälle definiert.

5.3.1 Einsatz der Testfälle

Eine wichtige Frage in dieser Arbeit ist „Was soll mit den Testfällen getestet werden?“. Diese Arbeit fokussiert sich auf den Systemverbund des elektrischen Antriebs (vgl. Abschnitt 3.1). Folglich liegt der Schwerpunkt auf BEV und PHEV Elektrofahrzeugen mit deren Komponentenverbund. Diese Arbeit beantwortet demnach nicht die Frage nach einem allgemein „guten“ Testfall wie in [Pre03] oder qualitativ hochwertigen Testfallspezifikationen wie in [Juh21], sondern fokussiert sich auf aus Spezifikationsmodellen erstellte Testfälle für (mechatronische) CPS. Der Fokus auf Elektrofahrzeugen bedeutet nicht, dass die in dieser Arbeit vorgestellte Methode nicht auch für andere Domänen und Systeme angewendet werden kann, sondern stellt die Methode beispielhaft in diesem Systemverbund vor.

Der im Zentrum stehende Komponentenverbund des elektrischen Antriebs wird in erster Linie in HIL und VIL Testumgebungen abgesichert. Diese primären Testumgebungen decken sich ebenfalls mit den Ergebnissen der Umfrage aus Abschnitt 5.2 (vgl. Tabelle 5.9). Bei der Absicherung mithilfe von HIL und VIL ist zu berücksichtigen, dass diese Testumgebungen verschiedenen Prüfständen entsprechen. Diese Prüfstände können sich in der Ausstattung und Aktualität unterscheiden. Außerdem kann die Lokalität der Prüfstände extrem variieren und demzufolge auch die Tester*innen. Zum Beispiel können sich funktional gleichwertige Prüfstände je nach Zulieferer auf unterschiedlichen Kontinenten befinden. Abbildung 5.19 stellt ein Szenario für global verteilte Testumgebungen vereinfacht dar.

Die Darstellung aus Abbildung 5.19 verdeutlicht noch einmal die Relevanz von testum-

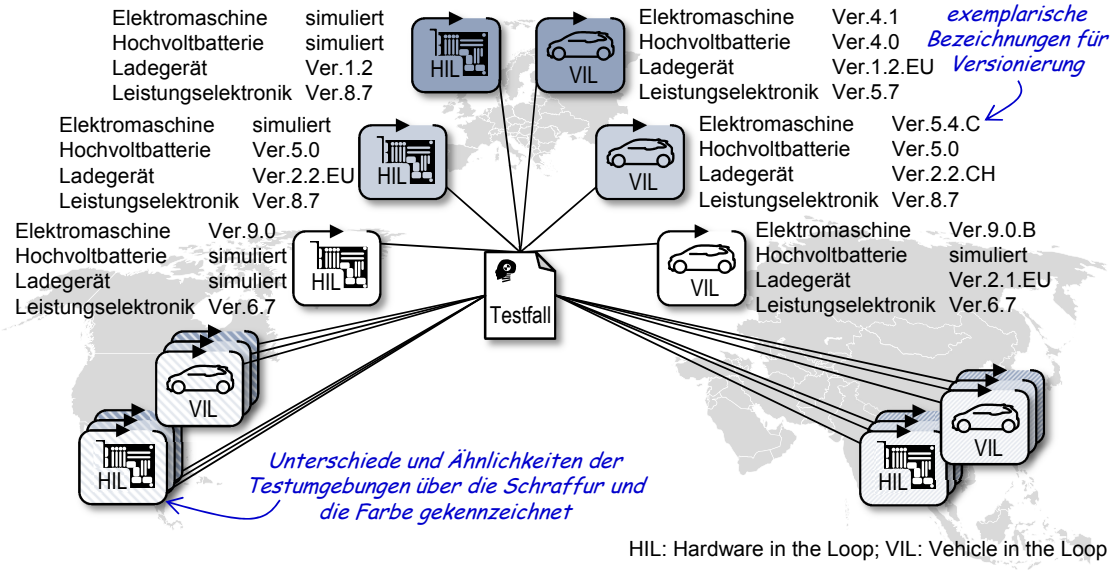


Abbildung 5.19: Veranschaulichtes Beispiel der lokal und global übergreifenden und unterschiedlichen Testumgebungen am Beispiel von HIL und VIL.

gebungsunabhängigen Testfällen. Nicht nur, dass lokal verschiedene Prüfstände vorliegen, sondern es findet auch ein übergreifender Austausch von Testfällen statt. Nur wenn eine Unabhängigkeit von der Testumgebung gewährleistet ist, können Testfälle im Fall eines Prüfstandausfalls, wie beispielsweise durch eine Wartung oder Verifizierung von Ergebnissen, übertragen und verwendet werden. Es ist wichtig diese Unabhängigkeit zwischen den Testumgebungen zu gewährleisten und ebenfalls eine übereinstimmende Funktionalität der Testfälle sicherzustellen. Aus diesem Grund ist es notwendig, dass in den Testfällen keine prüfstandsspezifische Charakteristika, wie beispielsweise spezielle Signalbenennungen oder simulationstypische Abläufe, enthalten sind. In dieser Arbeit wird dies mit einem Ansatz der Abstraktion bewerkstelligt. Ein solcher Abstraktionsgrad kann mit dem in der Automobilindustrie bereits etablierten schlüsselwortgetriebenen Testen umgesetzt werden (vgl. Abschnitt 3.2).

Nachdem der Einsatz der Testfälle näher beleuchtet wurde, wird im nächsten Abschnitt näher auf die gewünschten Eigenschaften der Testfälle eingegangen.

5.3.2 Eigenschaften der Testfälle

Für einen einwandfreien Einsatz der Testfälle sind neben der Testumgebung auch weitere Eigenschaften der Testfälle von Bedeutung. Die in dieser Arbeit im Fokus stehenden Testfälle umfassen folgende Eigenschaften:

- modellbasiert
- anforderungsbasiert

- funktionsorientiert
- automatisierbar
- abstrakt und einheitlich
- prägnant

Modellbasiert und anforderungsbasiert

Ein wichtiges Kriterium der Testabdeckung in der Automobilindustrie ist nach wie vor die Verifizierung über (textuelle) Anforderungen (vgl. Abbildung 5.8 und Tabelle 5.9). In SMArDT ist bezüglich der Testfalleigenschaften zu beachten, dass das Spezifikationsmodell alle Anforderungen umfasst, sodass „MBT gleich anforderungsbasiertem Testen“ entspricht. Im Gegensatz dazu, sind die Eigenschaften modellbasiert und anforderungsbasiert in dieser Arbeit getrennt aufgeführt, da die Testfälle auf der Grundlage von grafischen Diagrammen erstellt werden, aber die Testfallabdeckungskriterien (aktuell) nach textuellen Anforderungen bewertet werden (vgl. Theorie und Praxis 4.6). Weil die Informationen der Diagramme und der funktionsorientierten textuellen Anforderungen sich entsprechen beziehungsweise identische Informationen enthalten (vgl. Theorie und Praxis 4.6), steht dies nicht im Widerspruch (vgl. Abschnitt 5.3). Anforderungen, die nicht mittels grafischer Diagrammtypen wie UML oder SysML dargestellt werden können, werden als textuelle Anforderungen im Diagramm ergänzt und sind demnach ebenfalls ein Teil des Modells. Infolgedessen kann die Annahme getroffen werden, dass die Testfallabdeckung 100% ist, wenn alle (textuellen) Anforderungen getestet sind und umgekehrt die Testfallabdeckung 100% beträgt, wenn das Modell des Systems komplett getestet wurde.

Theorie und Praxis 5.3 (Anforderungsbasierte Testfallabdeckung)**.** *Im Fall der* 100*-prozentigen anforderungsbasierten Testfallabdeckung sind alle Anforderungen mit der aktuell vorliegenden und betrachteten Testumgebung getestet. Dies bedeutet nicht, dass alle Fehler in einem Testobjekt gefunden wurden. Es wird insofern nur eine Aussage darüber getroffen, dass die vorliegenden Anforderungen beziehungsweise spezifizierten Funktionen umgesetzt wurden. Weder kann Aussage getroffen werden, wie gut es funktioniert, noch, ob alle möglichen Umstände getestet wurden. Insbesondere bei einem CPS, das stark mit der Außenwelt interagiert, müssen immer auch die äußeren Umstände berücksichtigt werden. Die Aussagekraft über das Testobjekt der* 100*-prozentigen anforderungsbasierten Testfallabdeckung hängt somit stark von den zugrunde liegenden Anforderungen ab.*

Folglich sind die in dieser Arbeit behandelten Testfälle kein Ersatz für erfahrungsorientierte und explorative Tests, jedoch eine wichtige Basis für die Verifizierung eines Testobjekts (vgl. Bedenken der Tester*innen in Abschnitt 5.2). Theorie und Praxis 5.4 gibt Erfahrungen aus der Praxis mit erfahrungsorientierte und explorative Tests wieder.

Theorie und Praxis 5.4 (Anforderungsbasierte vs. explorative Tests)**.** *Ein sorgfältig, systematisch konzipiertes Testen kann ein hinreichend fehlerarmes Testobjekt erreichen [Lig09].*

Das anforderungsbasierte Testen ist eine Art des systematisch konzipierten Testens. Allerdings können Anforderungen trotz der in Abschnitt 5.1 definierten Kriterien unvollständig und fehlerhaft sein (vgl. Theorie und Praxis 5.3). Ein weiterer Aspekt ist, dass die Spezifikation beschreibt, wie das System funktionieren soll, wenn es richtig umgesetzt ist [KPB02]. Die Spezifikation umfasst weder alle Fehler, die zu erwarten sind[12]*, noch welche Testfälle zur Fehleridentifikation notwendig sind, um diese Fehler zu finden [KPB02]. Grundlegend verifizieren anforderungsbasierte Testfälle und Tests ein Testobjekt direkt und validieren indirekt über das Vieraugenprinzip durch die Tester*innen. Ein hinreichend fehlerarmes Testobjekt ist jedoch in der hier vorliegenden Komplexität weder mit rein erfahrungsorientierten Testfällen und explorativen Tests noch mit ausschließlich anforderungsbasierten Tests zu erreichen [Lig09]. Die Art der Testfälle und ihre resultierenden Tests schließen sich demzufolge nicht aus, sondern ergänzen sich sinnvoll, um ein Produkt hinreichend zu verifizieren und validieren.*

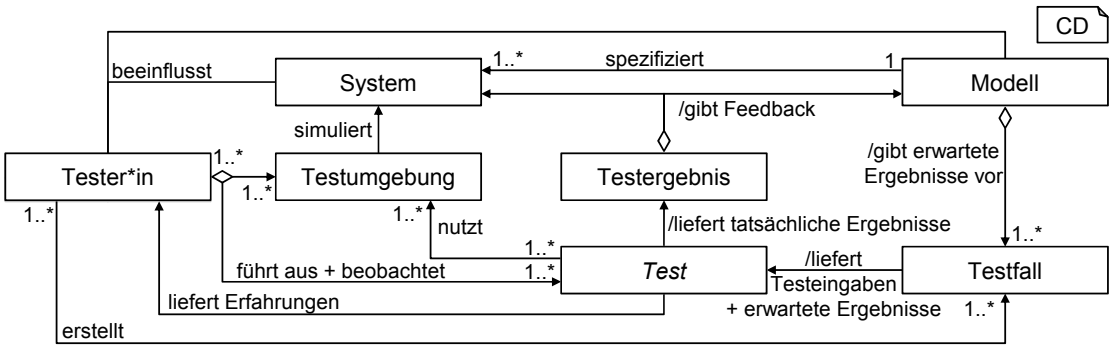


Abbildung 5.20: Vereinfachter Zusammenhang der Testdomäne, des Systems und des Modells

Abbildung 5.20 stellt den Zusammenhang der Testdomäne, des Systems und des Modells vereinfacht dar. Der Testfall liefert die Testeingaben und erwarteten Ergebnisse für die Tests, die er aus den Informationen des Modells bezieht. Diese Informationen nutzt wiederum der Test, der einerseits von der Testumgebung genutzt wird und andererseits die Testergebnisse liefert. Diese Testergebnisse werden von den Tester*innen interpretiert. Zudem geben die Testergebnisse dem Modell und dem System das Feedback, ob das System die Spezifikation des Modells erfüllt. Die Tester*innen führen die Tests auf der Testumgebung aus, die das betrachtete System simuliert. Voraussichtlich gewinnen die Tester*innen Erfahrungen aus der Testfallausführung und den Ergebnissen. Diese gewonnen Erfahrungen können die Tester*innen nutzen um das Modell und infolgedessen das System anzupassen beziehungsweise dieses zu verbessern. Zugleich wird die Tester*innen von der Modellspezifikation und dem System beeinflusst, da anhand dieser Elemente und den Erfahrungen Testfälle erstellt werden.

[12] Funktionen zur funktionalen Sicherheit, die zur Sicherstellung eines vertretbaren Risikos bei Gefährdungen und Fehlfunktionen beitragen, sind inbegriffen. Dies kann in diesem Fall deswegen erfolgen, da es sich ebenfalls um Funktionen handelt, die sicherstellen, dass ein System reibungslos funktioniert.

Funktionsorientiert

Die Testfallabdeckung ist ein wichtiges Absicherungsziel (vgl. Abbildung 5.14). Diese Absicherung/Verifizierung der Funktionalität wird über das Testen der einzelnen Funktionen und deren Subfunktionen bewerkstelligt. Zu diesem Zweck nutzen dynamische Testfälle die funktionsorientierte Spezifikation des Modells. Die Testfälle liefern demzufolge eine Aussage über das Erfüllen einer bestimmten spezifizierten Funktion bei gegebenen Eingangswerten. Aspekte, die nicht im Diagramm modelliert sind, sind aus dem Verwendungszweck der in dieser Arbeit behandelten Testfälle ausgeschlossen.

Automatisiert

Eine wichtige effizienzsteigernde Eigenschaft der Testfälle ist die der „Automatisierbarkeit“ (vgl. Tabelle 5.9). Die Eigenschaft „automatisierbar“ bezieht sich auf die Ausführung der Tests und nicht auf die Erstellung der Testfälle. Automatisierbar bedeutet in diesem Zusammenhang, dass die Testfälle in der Testumgebung automatisiert ausgeführt werden können. Diese Eigenschaft ist ein wichtiger Bestandteil für das effiziente und ressourcenschonende Testen (vgl. Abbildung 5.13 und Tabelle 5.17). Gängige eingesetzte Softwarewerkzeuge zur automatisierten V&V eingebetteter Systeme, wie beispielsweise [MAG20, Tra20, Vec20], nutzen hauptsächlich sequenzielle Testabläufe und Eingaben in Testfällen. Der Grund ist, dass die HIL und VIL Testumgebungen und deren Testautomatisierungstools, trotz mehrerer vorhandener Prozessoren, nur sequentielle Eingaben unterstützen. Das hat zur Folge, dass die Testfälle für die Automatisierung in solchen Testumgebungen einen sequentiellen Charakter haben müssen. Die Entscheidung der automatisierten Ausführbarkeit der Testfälle obliegt den Tester*innen. Diese müssen die Praktikabilität einer Automatisierung aufgrund von Kapazitäten, wie Personal, Zeit, Kosten und Testfallumgebung, und dem Testfokus entscheiden.

Theorie und Praxis 5.5 (Automatisierte und manuelle Tests)**.** *Obwohl automatisiert ausgeführte Tests viele Kapazitäten schaffen können, die für andere Aufgaben frei werden, haben manuelle Tests ihre Daseinsberechtigung [KPB02]. Besonders bei Systemtests werden bei manuellen Tests häufig Dinge von testenden Personen wahrgenommen, die nicht im Testfokus stehen, wie beispielsweise ungewöhnliche Geräusche aus einem anderen Teil des Fahrzeugs.*

Prägnant

Insbesondere für automatisierte Tests ist die Prägnanz der Testfälle ein wichtiges Kriterium. Automatisierte Testfälle werden in der Regel unbeobachtet durchgeführt und allein die Testergebnisse und Aufzeichnungen werden analysiert. Eine manuelle Analyse wird insbesondere bei Fehlern angestoßen. Diese Analyse ist umso einfacher, je prägnanter ein Testfall ist. Ein automatisierter Testfall ist genau dann prägnant, wenn er genau einen Teilaspekt einer Funktion(alität) testet und nicht viele Teilaspekte umfasst. Detektiert der prägnante Test einen äußeren Fehler, kann demnach direkt die Suche nach dem inneren Fehler beginnen. Werden hingegen viele Teilaspekte in einem Testfall getestet, muss

zuerst der auslösende äußere Fehler gefunden werden. Ein Testfall sollte infolgedessen möglichst kurz sein[13].

Abstrakt

Eine weitere Eigenschaft der Testfälle ist die der Einheitlichkeit (vgl. Tabelle 5.17). Einheitliche Testfälle sind nicht nur systemübergreifend sinnvoll, sondern auch in Bezug auf die in Unterabschnitt 5.3.1 erwähnten verteilten Testumgebungen. Die Einheitlichkeit wird in dieser Arbeit über den Ansatz der Abstraktion erreicht. Wie in Unterabschnitt 3.2.1 implizit erläutert, sollen die Testfälle auf einer gemeinsamen Sprache beruhen. Auf der Basis der gemeinsamen Sprache wird die Lesbarkeit und Verständlichkeit der Testfälle sichergestellt. Die Modelle ist von technischen Details wie konkreten Signalnamen und Testumgebungen abstrahiert. Somit wird ein Prüfen der Funktionalität und deren einzelne Funktionen, die auf verschiedenen Systemen laufen, losgelöst von bestimmten Varianten von Systemen, ermöglicht. Des Weiteren können verschiedene Methoden integriert werden, die beispielsweise Aussagen über Testobjekte treffen, ohne diese direkt zu testen und somit die Komplexität von Systemen und deren Varianten zu begegnen [EJK+19, PES20, AHLL+17].

5.4 Erwartungen an eine automatisiert modellbasierte Testfallerstellung

Im Gegensatz zur manuellen Testfallerstellung werden bei einer automatisierten Vorgehensweise vorrangig erwartet, dass diese die Effizienz steigert und dabei mindestens die gleiche Qualität der Absicherung aufweist. Die Vorgaben, die Anforderungen und die Ausgangslage für die manuell erstellten Testfälle gelten uneingeschränkt auch für die automatisiert erstellten Testfälle. Abbildung 5.21 stellt die Ausgangslage, die Anforderungen und die Vorgabe für die automatisiert erstellten Testfälle dar.

Die Anforderungen für die Eingabe der automatisierten Testfallerstellung wurde in Abschnitt 5.1 detailliert beschrieben. Gleichermaßen wurden die Erwartungen beziehungsweise die Anforderungen an die erstellten Testfälle in Abschnitt 5.3 dargelegt.

Die Erwartung der Tester*innen an einen automatisierten Prozess ist eine Effizienzsteigerung (vgl. Tabelle 5.17). Diese kann beispielsweise mit der Zeitersparnis bei der Erstellung einer fertigen Testfallspezifikation ohne einen Zwischenschritt (Testfallentwurf) erreicht werden (vgl. Zeitaufwand in Q10 in Abschnitt 5.2). Trotz der Effizienzsteigerung durch Automatisierung ist es sinnvoll, dass die Erfahrung der Tester*innen bei Bedarf in die Testfallerstellung mit einfließen kann.

[13] Neben kurzen prägnanten Tests haben auch Testfälle mit langen Testsequenzen ihre Berechtigung. Insbesondere für die Robustheit und Zuverlässigkeit wie Pufferüberläufe, Speicherlecks oder Ermüdungserscheinungen von Hardware sind einzelne Tests über Tage oder Wochen sinnvoll. Allerdings sind solche Dauer- oder Ausdauertests nicht Gegenstand dieser Arbeit, sondern Test mit einer Aussage über die Erfüllung einer bestimmten Funktionalität.

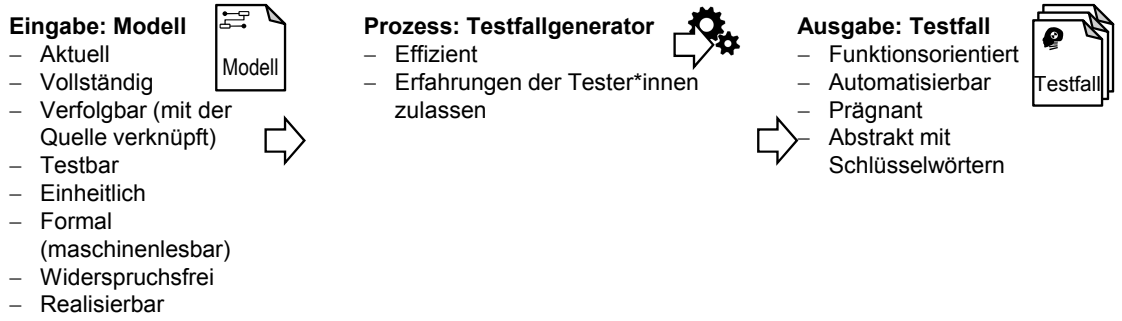


Abbildung 5.21: Übersicht über Eigenschaften der Eingabe, der Ausgabe und des Prozesses der automatisierten Testfallerstellung

Damit ein automatisiert erstellter Testfall die gleichen oder höhere Qualitätsstandards erreicht, wie ein manuell erstellter Testfall, müssen die gleichen Vorgaben erfüllt sein. Die in Abschnitt 5.2 und Abschnitt 5.3 beschriebenen Vorgaben, Einsatzbereiche, Eigenschaften und Verwendungszwecke gelten infolgedessen uneingeschränkt auch für automatisiert erstellte Testfälle. Die Gültigkeit der automatisiert erstellten und anforderungsbasierten Testfälle ist äquivalent zur Gültigkeit von manuell erstellten Testfällen. Das heißt, dass anforderungsbasierte Testfälle die Spezifikation Verifizieren aber nicht erfahrungsbasierte und explorative Testfälle ersetzen (vgl. Theorie und Praxis 5.4).

Wie auch für die manuellen Testfälle muss die Quelle, das Spezifikationsmodell, die Kriterien für die Testbarkeit erfüllen (vgl. Abschnitt 5.1). Hinzu kommt, dass das spezifizierte Verhalten nicht nur in einer formalen Form, sondern auch in einem maschinenlesbaren Format vorliegt.

Kapitel 6

Aktivitätsdiagramme für SMArDT

Die immer weiter zunehmende Verbreitung von Software in der Automobilindustrie erhöht die Notwendigkeit der interdisziplinären Zusammenarbeit von Informatiker*innen mit den Domänenexpert*innen des Maschinenbaus, der Elektrotechnik und der Elektronik (vgl. Kapitel 1). In der traditionellen Softwareentwicklung definieren diese Expert*innen, die Anforderungen an das System und formulieren dabei Lösungen auf Fachprobleme in ihrer eigenen Terminologie. Infolge der sich häufig unterscheidenden Terminologien und Herangehensweisen der Elektrotechnik, der Elektronik, des Maschinenbaus und der Informatik, kommt es häufig zu Fehlinterpretationen und Fehleinschätzungen. Zusätzlich ist eine komplette Spezifikation am Anfang eines Projekts selten möglich, da beispielsweise mechanische Anteile sich oft noch in der Entwicklung befinden und Entwicklungsprozesse miteinander verschränkt sind (vgl. Kapitel 4).

Aus diesem Grund kommt in dieser Arbeit, unterstützend zur interdisziplinären Herangehensweise von SMArDT, eine DSL zum Einsatz. Die Nutzung der fachspezifischen Notation erleichtert die Integration von Expert*innen und zwar stärker als bei einer herkömmlichen Softwareentwicklung, weil die Modelle von ihnen besser verstanden werden [FP11]. Der Einsatz der vorgestellten DSL ermöglicht die direkte Einbindung aller Expert*innen in den Entwicklungsprozess von zusammenhängenden abstrakten und detaillierten Modellen. Auf Basis einer gemeinsamen Sprache können die verschränkten Disziplinen mit ihren sich noch in der Entwicklung befindenden Anteilen besser verständigen und in Zusammenarbeit die geeignetste Lösung finden. Bei Änderungen der Spezifikation ist es zudem möglich, besser transparenter und kontrollierter auf die domänenübergreifenden Auswirkungen zu reagieren. Zusätzlich ermöglichen DSLs den Einsatz von Methoden aus dem Software Engineering wie die Prüfung von syntaktischen und semantischen Unterschieden zwischen Modellversionen [Kau21], die Identifikation von ungenauen (schwachen) Spezifikationen [KSS21] oder die Generierung von Software aus DSL-Modellen [HMR+19].

Zu diesem Zweck wird auf die domänenspezifischen Herausforderungen einer DSL in der Automobilindustrie in Abschnitt 6.1 eingegangen. Danach werden in Abschnitt 6.2 der Aufbau der Sprache anhand einer kompakten Version der Grammatik erläutert und zusätzlich die textuelle und grafische Repräsentation der Sprache gegenübergestellt und die Kontextbedingungen definiert. Die aus dieser Sprache und diesen Bedingungen hervorgehenden Modellierungsrichtlinien werden in Abschnitt 6.3 vorgestellt.

6.1 Herausforderungen einer Sprache für SMArDT

Bei der Definition und Implementierung von Modellierungssprachen gibt es zwei verschiedene Ansätze [SLH+17, Höl18]. Modellierungssprachen können entweder durch ein Metamodell [Küh04] oder durch eine Grammatik [GKR+08] definiert werden. Die in dieser Arbeit vorgestellte Modellierungssprache ist grammatikbasiert (vgl. Unterabschnitt 2.3.2).

Ziel der Sprache ist es, die einzelnen Terminologien der Disziplinen zu abstrahieren und zu bündeln, um eine gemeinsame Entwicklungsbasis zu schaffen. Außerdem sollen mittels der Modelle automatisierte Prozesse, wie die Testfallerstellung ermöglicht werden. Infolgedessen müssen die Modelle maschinenlesbar sein.

Die Ausgangslage der Sprachentwicklung ist ein interdisziplinäres Umfeld in der Automobilindustrie, das gängige Modellierungseditoren verwendet, die eine allgemeine SysML-Spezifikation unterstützen [OMG17a]. Die SysML zielt nicht auf die Bereitstellung einer eindeutigen Sprachspezifikation ab, die Mehrdeutigkeit verbietet. Daher gewährleistet die SysML weder Eindeutigkeit noch Maschinenlesbarkeit. Diese Ausgangslage stellt die Sprachentwicklung vor vier Herausforderungen, die im folgenden näher beschreiben werden:

- Benutzerfreundlichkeit für Entwickler*innen verschiedener Disziplinen
- Mehrere Zwecke innerhalb einer Domäne
- Modellierung von Verhalten der Funktionen als auch deren Interaktionen
- Abstraktion als auch Detaillierung je nach Anwendungsfall

6.1.1 Intuitive, disziplinenübergreifende Sprache

Die weit verbreiteten Modellierungseditoren (z.B. Enterprise Architect [Spa20], MagicDraw [NoM19] oder PTC Integrity Modeler [PTC19]) zielen oft darauf ab, eine große Bandbreite von Modellanwendungen zu bedienen. Daher schreiben diese Editoren nur eine sehr allgemeine Semantik der Modellierungssprache vor. Die allgemeine Semantik hat zur Folge, dass eine Wohlgeformtheit gemäß einer SysML-Spezifikation keine Maschinenlesbarkeit oder Eindeutigkeit impliziert. Die Modelle werden von Domänenexpert*innen erstellt, die in der Regel aus dem Umfeld der Elektrotechnik und des Maschinenbaus kommen (vgl. Abschnitt 5.2). Abbildung 6.1 stellt beispielhaft zwei abstrakte Diagramme der gleichen Funktion gegenüber.

Die vereinfacht dargestellte Funktion aus Abbildung 6.1 soll aufgrund der Informationen vom Energiespeicher die Leistungsaufnahme der E-Maschine beschränken. Zwischen der ersten (`Ver.1`) und der zweiten Version (`Ver.2`) der Funktion liegen ungefähr zwei Jahre, ohne dass die grundlegende Funktionalität und deren Anforderungen geändert wurden. Die erste Version lässt starke Einflüsse ausführbarer Simulationstools für kontinuierliche Systeme wie Matlab/Simulink und Modelica erkennen. Mithilfe der Simulationstools werden vorrangig Regelsysteme mit kontinuierlichen Verhalten modelliert, die eine wichtige Grundlage für die Entwicklung in der Automobilindustrie sind [SSS18]. Die von der

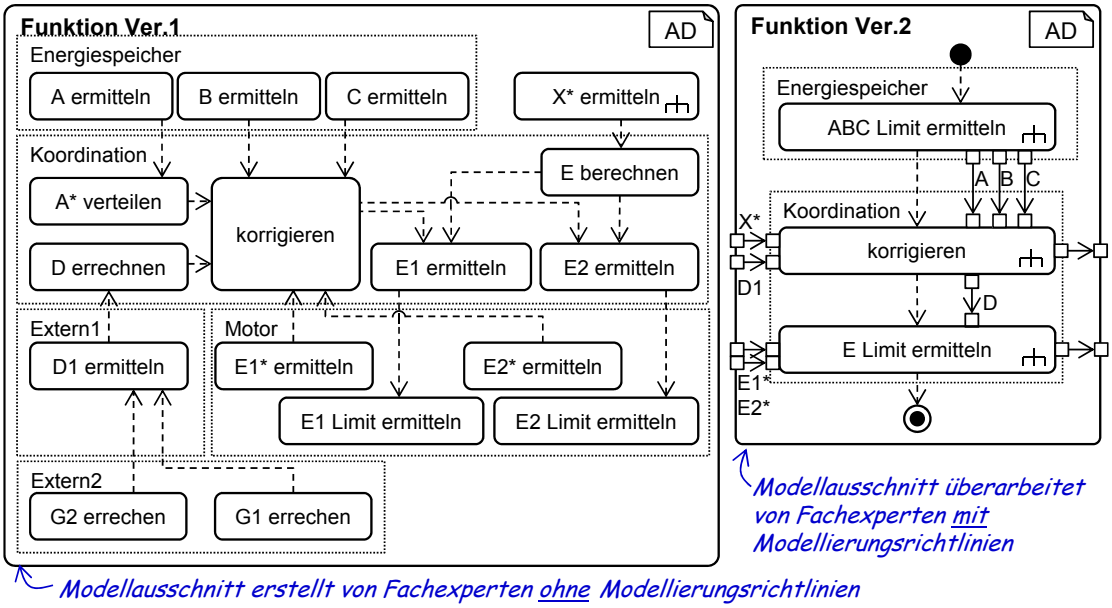


Abbildung 6.1: Vereinfacht dargestellte ADs der gleichen Funktionen mit und ohne Modellierungsrichtlinien für AD4S, ohne textuelle Anforderungen

objektorientierten Softwareentwicklung geprägten UML und deren Ableger, die SysML, eignen sich vor allem zur Darstellung diskreten Verhaltens. Die unterschiedliche kontinuierliche und diskrete Wahrnehmung spiegelt auch in gewisser Weise die zwei Ansichten von Systemen und deren Funktionen wieder: „gleichzeitig“ und „aufeinander folgend“. In der Aktivität `Funktion Ver.1` aus Abbildung 6.1 lässt sich diese „Gleichzeitigkeit“ erkennen. Zum einen hat die Funktion keinen eindeutigen Anfang und kein eindeutiges Ende. Zum anderen sind ebenfalls Aktionen modelliert, die nicht direkt zur Funktion beitragen, aber permanent benötigte Eingangsgrößen liefern, wie beispielsweise `D1 ermitteln`. In `Ver.2` ist der Einfluss einer diskreten Ansicht deutlich erkennbar. Es gibt eine klar definierte Sequenz von Anfang bis Ende, und die Funktion ist klar von anderen Funktionsteilen getrennt. Zusätzlich werden die Mehrdeutigkeit von `Ver.1`, wie beispielsweise sequenziellen Unklarheiten, von `Ver.2` eliminiert. Dies erleichtert eine maschinelle Verarbeitung.

Modelle, wie `Ver.1`, können innerhalb der Domäne größtenteils verstanden werden. Allerdings weichen die einzelnen Interpretationen ohne SysML-Modellierungsrichtlinien voneinander ab. Außerhalb der Domänen und zum Teil auch innerhalb der einzelnen Disziplinen treten allerdings häufig Verständnisprobleme aufgrund von verschiedenen Interpretationen auf. Das Fehlen eines allgemein verständlichen Modellierungskonzepts führt zu komplexen, unvollständigen oder nicht wohlgeformten Modellen, die Mehrdeutigkeiten enthalten und dadurch eine automatisierte Testfallerstellung verhindern. Ein geeignetes Modellierungskonzept muss daher kritische Aspekte der Modellierungssprache identifizieren und eine eingeschränkte Sprache definieren, um verständliche und maschinenlesbare Modelle zu erhalten.

6.1.2 Domänenspezifische Mehrzweck-Modelle

SMArDT ist eine Methodik für die interdisziplinäre Systementwicklung. Die Modelle dienen mehreren Zwecken wie beispielsweise der Dokumentation, dem Test und der Kommunikation. Die Modelle werden nicht nur von den beteiligten Entwickler*innen und Testingenieur*innen betrachtet, sondern auch von Projektkoordinator*innen, die für die Reifegradplanung[1] verantwortlich sind. Letztere besitzen meistens einen betriebswirtschaftlichen Hintergrund. Daher darf bei einer Definition der Modellierungssprache für eine automatisierte Testfallgenerierung, die Mehrfachverwendung nicht außer Acht lassen.

Eine wohlgeformte, wohldefinierte und einheitliche Sprache allein reicht nicht, wie für eine generische Modellierung verlangt. Um Probleme der Verständlichkeit über Domänen hinweg auszuräumen, ist eine Sprache mit einer einfachen Semantik nötig [Pre03]. Ein Modell sollte einfach zu erstellen sein[2]. Das gemeinsam vereinbarte Modellierungskonzept muss demzufolge in ihrer Ausdrucksfähigkeit als Beschreibungsmittel eingeschränkt werden, um die Komplexität der Sprache nicht unnötig zu steigern. Diese Einschränkung erleichtert eine intellektuelle und maschinelle Analyse.

6.1.3 Modellierung von Verhalten und Interaktionen

Im Mittelpunkt der Spezifikation mit SMArDT stehen das Verhalten und die Interaktion von Funktionen, Systemen und Komponenten (vgl. Unterabschnitt 4.2.2 und Unterabschnitt 4.2.3). Auf Ebene B werden in SMArDT Modelle für die (abstrakte) Funktionsbeschreibung - einschließlich des Funktionsablaufs - deren Schnittstellen als auch Interaktionen (Informationsflüsse) genutzt. Ebene C verfeinert diese Funktionen und modelliert die Funktionsabläufe mit Bezug auf die konkreten Systemelemente sowie deren Schnittstellen und Signalen.

Eine Modellierungssprache sollte diese Abläufe sowie die Interaktionen unabhängig von der Betrachtungsebene einfach abbilden können, ohne notwendige Details wie Funktionsablauf, Interaktionen oder Schnittstellen auszublenden. Gleichzeitig müssen die Modelle maschinenlesbar sein, um eine maschinelle Verarbeitung zu ermöglichen.

6.1.4 Vereinbarkeit von Abstraktion und Detaillierung

Die Abstraktion auf den SMArDT-Ebenen ermöglicht komplexe Sachverhalte, mit einer intellektuell handhabbaren Formalisierbarkeit darzustellen. Insbesondere in den höheren Ebenen, wie Ebene A, Ebene B und auch Ebene C kann mit der Loslösung von Hardware, Software und deren Spezifika die Komplexität gemindert werden. Zusätzlich wird eine Wiederverwendung von Modellen ermöglicht, die beispielsweise bei verschiedenen Produktlinien zum Einsatz kommt. Die Abstraktion entspricht jedoch einer Unterspezifikation im konkreten Modell [Rum16, Rum17]. Ein Modellierungskonzept muss demnach

[1] Die Reifegradplanung agiert nach einem Reifegradmodell der ISO 9004 [DIN18b]. Das fünfstufige Reifegradmodell erleichtert die objektive Beurteilung der Prozesse und dient als Leitfaden zur nachhaltigen Leistungsverbesserung eines Qualitätsmanagementsystems zu verstehen [DIN18b].

[2] Der Aspekt des Toolings wird in dieser Arbeit nicht betrachtet.

in der Lage sein, die Unterspezifikation so zu handhaben, dass sie mit den Spezifikationen der Abstraktionsschicht übereinstimmt. Der Automatismus der Testfallgenerierung auf abstrakten Schichten muss sicherstellen, dass die auf den unteren Abstraktionsschichten generierten Testfälle ihre übergeordneten Geschwister verfeinern. Das Modellierungskonzept muss daher so angepasst werden, dass der Testfallmechanismus in der Lage ist, eine solche Unterspezifikation angemessen zu behandeln.

Der im nächsten Abschnitt vorgestellte Ansatz überwindet diese Herausforderungen, indem es eine Modellierungssprache mit eindeutiger Semantik für jede Entwicklungsstufe anfordert. Durch die Spezifikation einer eindeutigen Sprache für die Modellierung funktionaler Anforderungen wird eine automatisierte Testfallgenerierung möglich. Dadurch werden sowohl maschinenlesbare Modelle als auch eine gemeinsame Verständigungsbasis geschaffen.

6.2 MontiCore-Grammatik für AD4S

Die in Abschnitt 6.1 angesprochenen Herausforderungen werden in dieser Arbeit mit einer DSL, die SysML [OMG17a] weiter einschränkt, begegnet. Eine GPML wie die SysML, ist hierfür nicht restriktiv genug. Ziel der GPML ist es, generelle Modellierungskonzepte für alle Modellierer*innen zur Verfügung zu stellen, die als deren Modellierungssprache verwendet werden kann. Ohne eine definierte Methodik zur Modellierung sind die Modelle allerdings wie Bilder, die sich frei interpretieren lassen und fernab von spezifischen Funktionsbeschreibungen, deren Abläufen, Interaktionen und Schnittstellen. Folglich kann die GPML allein nicht sicherstellen, dass die Modellierer*innen eindeutige Modelle erstellen. Zum einen erhalten bei GPMLs die Modellierenden keine Unterstützung bei der Abstraktion und müssen selbstständig Abstraktionskonzepte ausarbeiten. Die individuelle Konzepterstellung gefährdet vor allem die Herausforderungen der *generischen Modellierung* und *Mehrzweck-Modelle* (vgl. Abschnitt 6.1). Dies liegt unter anderem daran, dass kein übergreifendes generisches Modellierungskonzept sichergestellt werden kann. Die inkonsistenten Modellierungskonzepte können zu Fehlinterpretationen führen. Die Wahrscheinlichkeit der Fehlinterpretationen steigt insbesondere bei Modellen, die interdisziplinär genutzt und mehrfach betrachtet werden. Zusammenfassend erhöht die Eigenschaft einer GPML, Modellierungssprachen für sämtliche Domänen und Sachverhalte zu liefern, die Komplexität für spezifische Domänen. Aus diesem Grund wird für die in dieser Arbeit fokussierten Domäne eine DSL vorgestellt, um die Modellierungsmöglichkeiten an die Interessengruppen anzupassen. Für die schnelle und effektive Umsetzung dieser DSL wird das in Unterabschnitt 2.3.2 vorgestellte MontiCore-Framework verwendet. Für detaillierte Informationen zu dem verwendeten MontiCore-Framework wird auf [HR17] verwiesen.

Die in dieser Arbeit vorgestellte Sprache mit dem Namen Aktivitätsdiagramme für SMArDT (AD4S) dient als formale Grundlage, um den SysML-Standard so einzugrenzen, dass die Modelle eine eindeutige Semantik aufweisen. Zu diesem Zweck wurde auf das Modellierungskonzept des SysML-ADs zurückgegriffen, das sich insbesondere dazu eignet Abläufe und Interaktionen darzustellen. Infolgedessen sind die Modelle allgemein verständ-

lich und maschinenlesbar. Es wurde insbesondere darauf geachtet die Ausdrucksfreiheit der modellierenden Domänenexpert*innen zu erhalten.

Abbildung 6.2 stellt die Beschreibung der Grammatik kompakt dar. Die kontextfreie AD4S-Grammatik wurde erstmals in [DGH+19] veröffentlicht. Zugunsten einer kompakteren Darstellung werden einige technische Grammatikdetails ausgelassen. In Zeile


```
1  grammar ADforSMArDT {
2     interface Node;
3     Activity implements Node = "activity" Name Requirements? "{"
4         Parameter*
5         (Node | ControlNode | Edge)*
6     "}" ";" ;
7     Parameter = "in:" | "out:" Name ("," Name) ";" ;
8
9     Action implements Node = "action" Name Requirements?;
10
11    Requirements = "[" req:Requirement ("," req:Requirement )* "]" ";" ;
12    Requirement = "(" Name "," value:String "," security:["security"] ")" ";" ;
13
14    interface ControlNode extends Node;
15    InitialNode    implements ControlNode = "initial" ";" ;
16    FinalNode      implements ControlNode = "final" ";" ;
17    FlowFinalNode  implements ControlNode = "flowfinal" " ;" ;
18    ForkNode       implements ControlNode = "fork" Name ";" ;
19    JoinNode       implements ControlNode = "join" Name ";" ;
20    DecisionNode   implements ControlNode = "decision" Name ";" ;
21    MergeNode      implements ControlNode = "merge" Name ";" ;
22
23    Edge = "transition" id:String? source:Name ("->" guard:Guard? target:Name)+ ";"  ;
24    Guard = "["/" Variable+ | ("else" ("/" Variable+)) | (BooleanExpr ("/" Variable+)) "]" ;
25    Variable = signal:Name ("=" value:Name)? ";" ;
26 }
```

Abbildung 6.2: Kompakte kontextfreie MontiCore-Grammatik für AD4S [DGH+19]

2 von Abbildung 6.2 wird das grundlegende Nichtterminal von AD4S definiert. Hierbei handelt es sich um den Knoten (`Node`). Die Knoten ermöglichen eine hierarchische Verzweigung/Strukturierung der Aktivitätselemente (vgl. Unterabschnitt 2.3.1). Eine Aktivität (`Activity`) hat eine Liste von Knoten (Zeile 2 bis 6). Aktivitäten können erneut Knoten, Kontrollknoten (`ControlNode`) und Kanten (`Edges`) enthalten (Zeile 5). Folglich sind Knoten Schnittstellen und bilden die Möglichkeit von hierarchischen Strukturen in AD Elementen, da sie weitere Knoten beinhalten können. Die Kardinalität (`*`) signalisiert, dass Aktivitäten unbeschränkt viele Knoten, Kontrollknoten und Kanten enthalten können. Ergänzend zu Knoten können Aktivitäten eingehende (`in`) und ausgehende (`out`) `Parameter` enthalten (Zeile 4 und 7). In Zeile 9 wird die grundlegendste Art von Knoten in einem AD, die Aktion (`Action`) definiert. Aktionen und Aktivitäten können von beliebig vielen spezifizierten Anforderungen (`Requirements`) referenziert werden (Zeile 3 und 9). Die Kardinalität (`?`) signalisiert, dass Anforderungen für Aktionen und Aktivitäten optional sind. Anforderungen bestehen aus mehreren einzelnen Anforderungen (`Requirement`-Elementen) (Zeile 11). Eine Anforderung besteht aus einem Namen, einem Wert (`value:String`), der die Zeichenfolge enthält und einem Sicherheitsattribut

(`security`) (Zeile 12) (vgl. Prioritätskriterien in Abschnitt 5.1). Ein Knoten beinhaltet auch eine Liste von Kontrollknoten, die eine grundlegende Implementierung von Kontrollknoten der UML-, UML/P- beziehungsweise SysML-ADs sind (vgl. Unterabschnitt 2.3.1). Die Liste beinhaltet folgende Elemente (Zeile 14 bis 21):

- Startknoten (`InitialNode`)
- Endknoten (`FinalNode`)
- Ablaufendknoten (`FlowFinalNode`)
- Verzweigungsknoten (`ForkNode`)
- Vereinigungsknoten (`JoinNode`)
- Entscheidungsknoten (`DecisionNode`)
- Verbindungsknoten (`MergeNode`)

Die Verbindung zwischen den Knoten bilden Kanten. Die Kanten sind über einen Quellknoten (`source`) und einem Zielknoten (`target`), die auf die jeweiligen angeschlossenen Knoten zeigen, definiert (Zeile 23). Die ID (`id:String?`) dient zur Identifikation der jeweiligen Kante. Da sich Kanten immer als eine Kombination aus Start, Ende, Name und Kardinalität oder gegebenenfalls Wächterausdruck identifizieren lassen, dient die ID vor allem der Vereinbarkeit mit verschiedenen Tools über eine Darstellung der AD4S in der erweiterbaren Auszeichnungssprache (engl. „Extensible Markup Language“) (XML) [Wor21a]. Optional können Kanten eine Bedingung (`Guard`) enthalten (Zeile 23). Diese Bedingungen werden nach dem Quellknoten gefolgt von einem „`->`“ definiert. Nach einem Wächterausdruck folgt der Zielknoten. Ein Wächterausdruck ist von eckigen Klammern umschlossen und dient zwei Zwecken. Einerseits entspricht er einem booleschen Ausdruck, wenn die Quelle der Kante ein Entscheidungsknoten ist. Die Bedingung kann auch „sonst“ (`else`) sein, falls alle anderen Wächterausdrücke des Zweiges nicht erfüllt werden. Ohne solche Bedingungen in Form von booleschen Ausdrücken ist eine Pfadvalidierung nicht möglich (vgl. Unterabschnitt 7.1.8). Der zweite Zweck der Wächterausdrücke ist der, Signalzuweisungen oder -definitionen zu modellieren (`BooleanExpr (/"Variable+)`). Eine Zuordnung oder Definition von Variablen (`Variable`) wird mit einem voran gesetzten Schrägstrich modelliert. Eine Kombination von verschiedenen Wächterausdrücken ist möglich. In Zeile 25 ist abschließend das Aussehen von Variablen definiert (Zeile 25). Zum Beispiel prüft der Wächterausdruck `[var1 == true / out1 = false]`, ob die Variable `var1` den Wert `true` hat und setzt in diesem Fall die Variable `out1` auf den Wert `false`. Diese Syntax weicht zunächst minimal von der SysML Syntax aus [OMG17a] mit `trigger[guard]/activity` mit der eckigen Klammersetzung ab, verhindert diese aber nicht. Mit dieser Flexibilität können Modellierende freier agieren und Fehler in der Syntax werden verringert.

Auch wenn das textuelle Modell Vorteile bietet, hat sich die grafische Modellierung in der Praxis insbesondere bei SMArDT durchgesetzt (vgl. Kapitel 4). Aus diesem Grund werden im nächsten Abschnitt die grafische und textuelle Repräsentation von AD4S gegenübergestellt.

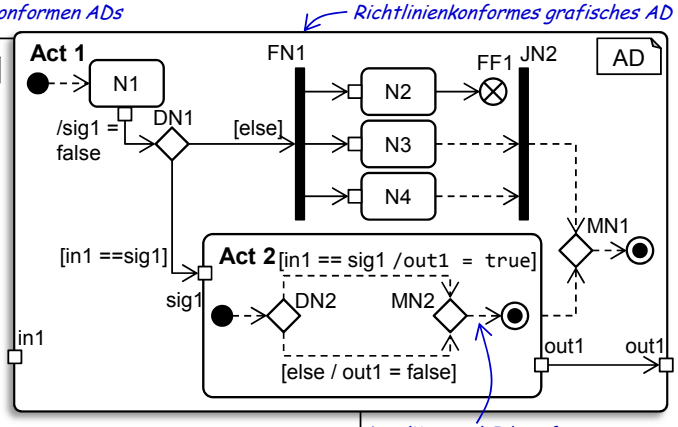
6.2.1 Grafische und textuelle Repräsentation von AD4S

Wie in Abschnitt 4.1 beschrieben, umfassen die Modelle abstrakte Funktionsbeschreibungen. Für eine maschinelle Verarbeitung der Modelle für das Testen müssen die eingeführten Unterspezifikationen in den Diagrammen bewältigt werden. Die Erstellung grafischer Modelle unter Einhaltung der grammatikalischen Bedingungen impliziert die Existenz einer äquivalenten Darstellung in AD4S. Im Folgenden wird die Sprache vorgestellt, die insbesondere für Diagramme in der SMArDT-Ebene *B* zugeschnitten ist. In Abbildung 6.3 sind ein richtlinienkonformes grafisches AD und dessen textuelle AD4S-Repräsentation abgebildet.

Textuelle AD4S-Repräsentation des richtlinienkonformen ADs

```
1  activity Act1 {
2     in: in1;
3     out: out1;
4
5     action N1, N2, N3, N4;
6     fork FN1;
7     join JN1;
8     decision DN1;
9     merge MN1;
10    //weitere Knoten
11
12    activity Act2 {
13       in: sig1;
14       out: out1;
15       decision DN1;
16       merge MN1;
17       transition „trans05" initial -> DN2;
18       transition "trans06" DN2 -> [in1 == sig1 /out1 = true] MN2;
19       transition "trans07" DN2 -> [else /out1 = false] MN2;
20       transition "trans08" MN2 -> final;
21    }
22
23    transition "trans01" initial -> N1;
24    transition "trans02" N1 -> [/sig1 = false] DN1;
25    transition "trans03" DN1 -> [else] FN1;
26    transition "trans04" N2 -> FF1;
27    //weitere Kanten
28 }
```

Richtlinienkonformes grafisches AD



Die (Kanten-) Identifier wie „trans08" werden für eine bessere Übersicht in einer grafischen Darstellung ausgeblendet.

Abbildung 6.3: Darstellung eines richtlinienkonformen grafischen ADs und dessen textuelle AD4S-Repräsentation. In der AD4S-Repräsentation sind nicht alle Kanten (`transitionen`) des grafischen ADs abgebildet.

In Abbildung 6.3 ist zu erkennen, dass eine Aktivitätsdefinition in AD4S die Eingangs- und Ausgangsparameter in der Parameterliste (Zeile 2 und 3) angegeben sind und die zwei Listen mit Semikolon getrennt werden. Die Listen enthalten mit `in:` deklarierte Eingangssignalnamen und mit `out:` deklarierte Ausgangssignalnamen. Die Knoten des ADs werden in einer Knotendefinitionsliste (Zeile 5 bis 9) deklariert. Die Knotendefinitionsliste enthält die Markierung des Knotentyps. Ein Beispiel für eine Markierung des Knotentyps ist `action` für einen Aktionsknoten gefolgt von einem eindeutigen Knotennamen. Die

Knotennamen müssen nur im Bereich der Aktivität eindeutig sein. Verschachtelte Aktivitäten können daher Knoten mit denselben Namen wie in der übergeordneten Aktivität enthalten (vgl. Zeile 3 und 14). Eine verschachtelte Aktivität wird durch eine äquivalente Aktivitätsdefinition innerhalb einer Aktivität definiert. Kanten werden in einer Kantenliste definiert (Zeile 17 bis 20 und Zeile 23 bis 26). Eine Kantendefinition beginnt mit dem Schlüsselwort `transition`, gefolgt von dem Identifier, der aus Buchstaben und ganzen Zahlen besteht. Der Rest der Übergangsdefinition enthält den Namen des Quellknotens, gefolgt von einem „`->`“, dem Namen des Zielknotens und optional Wächterausdrücke. Der Identifier erlaubt ein Gewichten der Kanten für die Testfallerstellung und eine bessere intellektuelle Nachvollziehbarkeit bei Modelltransformationen, die in Abschnitt 7.1 vorgestellt werden. In den Zeilen 18, 19 und 24 mit Signalzuweisungen ist zu erkennen, dass die AD4S-Grammatik keine Datentypdefinitionen vorgibt. Diese Unterspezifikation ist unter anderem in abstrakteren Ebenen, wie der SMArDT-Ebene B, hilfreich, da alle Variablen und Signale logisch sind. Somit ist hierfür kein zugehöriger Datentyp bekannt.

Für die textuellen ADs und deren grafisches Äquivalent kann mit bereits entwickelten MontiCore-Framework basierenden Modellprüfungstools [DGH+19] sichergestellt werden, dass Modelle in frühen Phasen eines Generierungsprozesses erkannt werden, falls diese noch nicht grammatikalisch analysiert wurden. Darüber hinaus kann mit dem Parser des MontiCore-Frameworks gewährleistet werden, dass syntaktisch korrekte Diagramme erstellt werden. Zusätzlich zur definierten Syntax kann ebenfalls geprüft werden, ob bestimmte Kontextbedingungen erfüllt sind. Die Kontextbedingungen der Grammtiksprache sorgen dafür, dass die AD4S-Grammatik wohlgeformt ist. Diese Kontextbedingungen können in [HR17] eingesehen werden und betreffen hauptsächlich Nichtterminale, deren Erweiterung oder Implementierung, Existenz und Namenskonflikte. Zum anderen existieren spezifische Kontextbedingungen für die Modelle der AD4S-Grammatik definiert. Eine Auswahl dieser spezifischen Kontextbedingungen werden im nächsten Abschnitt näher betrachtet.

6.2.2 Kontextbedingungen von AD4S

Für eine maschinelle Verarbeitung der Modelle, wie die automatisierte Testfallerstellung, ist das automatische Durchqueren von extrahierten Pfaden nötig. Informalitäten wie Verklemmungen (engl. „Deadlocks“), Endlosschleifen oder nichtdeterministische Verzweigungen verhindern die automatische Extraktion von Pfaden. Daher müssen grafische Modelle und deren textuelles Äquivalent wohlgeformt sein, bevor eine maschinelle Verarbeitung möglich ist. MontiCore generiert eine Kontextbedingungsinfrastruktur, die eine Modellprüfung für AD4S-Modelle erleichtert [HR17]. Im Folgenden werden die definierten Kontextbedingungen beschrieben, die eine maschinelle Lesbarkeit von abstrakten SMArDT-Modellen in AD4S sicherstellen. Dies ermöglicht ein umgehendes Feedback, um Fehler sofort zu erkennen und diese Fehler korrigieren zu können. Zusätzlich hilft die formale Semantik Mehrdeutigkeiten des Modellinhalts zu vermeiden und erleichtert so die interdisziplinäre Zusammenarbeit.

Jeder Entscheidungsknoten eines Aktivitätsdiagramms ist deterministisch.

Die Syntax von AD4S erlaubt das Fehlen von Wächterausdrücken auf Kanten (vgl. Abbildung 6.2 Zeile 23). Sie wird nicht benötigt, wenn eine Kante ein Kontrollfluss zwischen zwei aufeinander folgenden Knoten modelliert. Wenn die Quelle der Kante jedoch ein Entscheidungsknoten ist, muss ein Wächterausdruck vorhanden sein. Darüber hinaus muss der Satz von Wächterausdrücken an allen ausgehenden Kanten des Entscheidungsknotens so vollständig sein, dass er den gesamten logischen Wertebereich der übergeordneten Aktivität abdeckt. Ohne diesen vollständigen Satz ist eine korrekte Pfadvalidierung, ob - maschinell oder manuell - ohne Hintergrundinformationen nicht möglich. Die automatische Durchquerung des Modells und die Pfadextraktion werden aufgrund des entsprechenden nichtdeterministischen Verhaltens verhindert. Ein vereinfachtes Beispiel ist Abbildung 6.4 abgebildet.

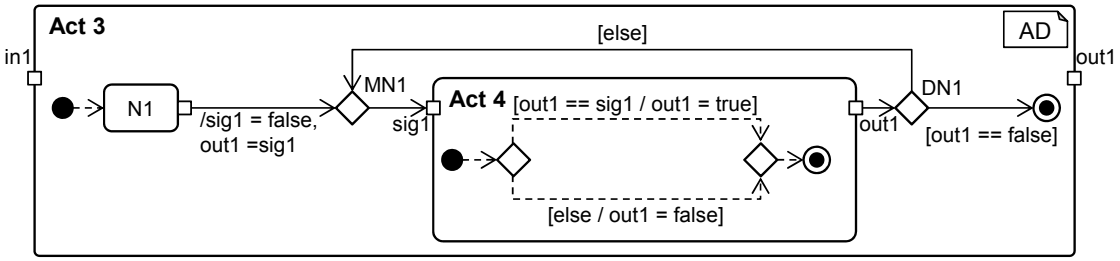


Abbildung 6.4: Beispiel-AD mit deterministischem Verhalten

Die Verzweigungen des Entscheidungsknoten `DN1` in Abbildung 6.4 ist durch die `[else]`-Bedingung vervollständigt. Das `[else]` greift alle Bedingungen des jeweiligen Entscheidungsknotens auf, die nicht zu treffen. In der Folge entspricht die `[else]`-Bedingung von `DN1` der booleschen Bedingung `out1 != false`.

Jedes Aktivitätsdiagramms ist frei von Deadlocks.

Viele Funktionen in einem Fahrzeug basieren traditionell auf Regelsystemen. Unter anderem ist dies ein Grund, dass das Verhalten mehrerer auftretender Aktionen als simultan wahrgenommen wird. In AD4S und der SysML können simultane Prozesse innerhalb von Verzweigungs- und zusammenführenden Vereinigungsknoten modelliert werden. Diese Art zu modellieren wird in den abstrakten SMArDT-Ebenen A und B ebenfalls für die Darstellung von Prozessen verwendet, für die keine Reihenfolge vorgegeben werden soll oder die Reihenfolge noch nicht klar ist. Aktionen, wie beispielsweise `N2`, `N3` und `N4` in Abbildung 6.3, könnten in abstrakten Ebenen simultan oder auch in einer nicht festgelegten Reihenfolge eintreten. Um die Maschinenlesbarkeit zu gewährleisten, muss sichergestellt sein, dass das AD keine Deadlocks enthält. Diese Bedingung entspricht der formalen Wohlgeformtheitsbedingung, dass jeder Pfad eines ADs an einem Startknoten beginnt und an einem Endknoten enden muss. In dieser Arbeit wird in AD4S zwischen zwei Formen von Deadlocks unterschieden, Deadlocks aufgrund von modellierten Endlosschleifen und

Deadlocks, wenn ein Pfad unterbrochen wird. In beiden Fällen wird der Endknoten einer Aktivität nicht erreicht. Abbildung 6.5 zeigt zwei Beispiele für Deadlocks, aufgrund von unterbrochenen Pfaden.

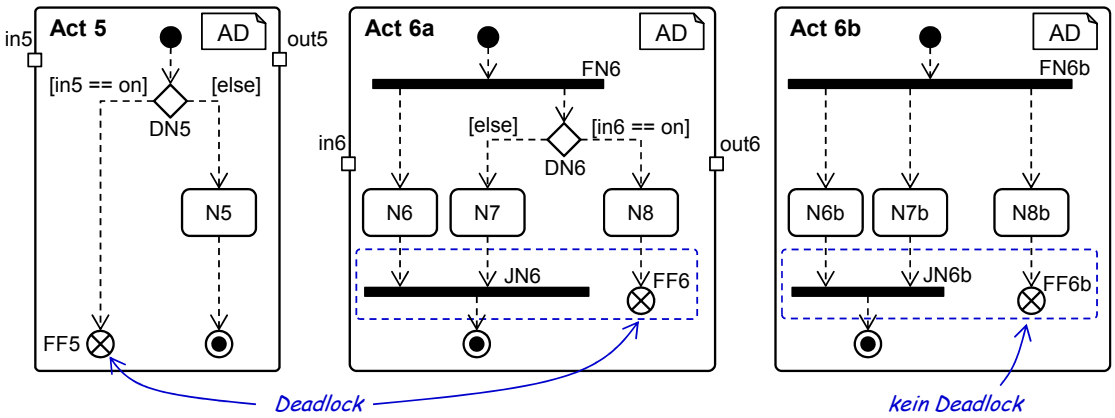


Abbildung 6.5: Unterbrochene Pfade in ADs: Sobald `[in5 == on]` beziehungsweise `[in6 == on]` zutrifft, tritt ein Deadlock ein.

In Abbildung 6.5 tritt in `Act 5` ein Deadlock auf, wenn `[in5 == on]` eintritt, da der Ablaufendknoten `FF5` verhindert, dass die Aktivität geschlossen wird. In `Act 6a` wird im Fall `[in6 == on]` der Vereinigungsknoten `JN6` nicht vollständig verknüpft, da `N7` nicht ausgeführt wird und der Ablaufabschlussknoten `FF6` die Aktivität nicht abschließt. `Act 6b` stellt zur Veranschaulichung noch einmal eine `Act 6a` ähnliche Aktivität, ohne einem Deadlock, gegenüber. Es liegt deshalb kein Deadlock in `Act 6b` vor, weil immer alle Aktionen ausgeführt werden und der Endknoten erreicht wird. Die Aktivität `Act 6b` wird infolgedessen immer geschlossen, sobald die Aktionen `N6b` und `N7b` abgeschlossen sind. Auch `N8b` wird irgendwann beendet und `FF6b` erreicht (vgl. Fairness [BK08a]). Es ist anzumerken, dass die Domänenexpert*innen nicht ausschließlich Rechenprozesse modellieren, sondern ebenfalls Funktionen aus Kundensicht. Zum Beispiel könnte `N6b` die Entriegelung des Ladesteckers und `N7b` die Anzeige auf dem Display repräsentieren, während `N8b` die Meldung auf einem Fahrzeugschlüssel oder Smartphone auslöst. `N8b` wäre infolgedessen für den Ladeprozess nicht zeitkritisch und könnte demnach auch verzögert gesendet werden.

Neben den angesprochenen Kontextbedingungen sind ebenfalls allgemeine Kontextbedingungen für AD4S definiert. Zu diesen Bedingungen gehören beispielsweise, dass bei Kanten immer eine Quelle und ein Ziel existieren müssen oder dass die Aktionsnamen eindeutig sind. Diese Bedingen werden meist von verbreiteten Modellierungseditoren gewährleistet. Zur sofortigen Rückmeldung der Modellkorrektheit an die Modellierer*innen wurde in dem verwendeten grafischen Modellierungseditor ein automatischer Richtlinienprüfer für alle definierten Kontextbedingungen integriert. Die AD4S-Kontextbedingungen wurden den Nutzer*innen über Modellierungsrichtlinien vermittelt. Die essenziellen Modellierungsrichtlinien werden im nächsten Abschnitt näher erläutert.

6.3 Modellierungsrichtlinien für Aktivitätsdiagramme

Das Arbeiten mit grafischen Modellen ist in SMArDT ein wichtiger Bestandteil. Für eine maschinelle Verarbeitung der Modelle müssen die grammatikalischen Bedingungen von AD4S eingehalten werden. Zu diesem Zweck wurden Modellierungsrichtlinien für SMArDT-Diagramme gemeinsam mit den Domänenexpert*innen entwickelt und allen beteiligten Modellanwender*innen mitgeteilt, um eine gemeinsame Verständigungsbasis zu schaffen. Die AD4S beansprucht, unabhängig von Modellierungstools anwendbar zu sein[3], die das Vorhandensein einer AD4S konformen Darstellung implizieren. Die Entwicklung mit den modellierenden Domänenexpert*innen stellt sicher, dass die Richtlinien die Benutzer*innen nicht ungewollt einschränken.

Die Modellierungsrichtlinien gewährleisten die Existenz einer wohlgeformten textuellen Äquivalenz von AD4S und legen den grafischen Modellen damit implizit die AD4S-Semantik auf. Dadurch enthalten SMArDT Modelle eine eindeutige Semantik, die ein gemeinsames Modellverständnis begründet. Die Richtlinien (**RL**s) für ADs zielen darauf ab, die Spezifikation der SysML so einzugrenzen, dass Mehrdeutigkeiten vermieden werden und deterministische Wege durch das AD gewährleistet sind. Andere Elemente und Richtlinienpunkte sind nicht durch AD4S-Richtlinien oder deren Kontextbedingungen geregelt:

- Kommentare, Notizen, verschiedene Elemente zur Beschreibung, Zuordnungsabhängigkeiten und andere beschreibende Elemente;
- Verknüpfung zwischen Modellierungselementen, wie zum Beispiel ADs und IBDs;
- Abweichungen zwischen dem intellektuellen und maschinellen Interpretation von Diagrammen;
- Richtlinien zu anderen Themenbereichen, wie beispielsweise Funktionssicherheit, Diagnose und Anforderungsmanagement

Die AD4S-Richtlinien definieren größtenteils den grafischen Anteil der ADs, der die automatische Verarbeitung des Modells ermöglicht. Im Wesentlichen schränken diese Richtlinien die Syntax der SysML AD-Sprache ein und legen die Kontextbedingungen des textuellen AD4S auf grafische ADs fest. Dies ermöglicht eine automatische Übersetzung von graphischen Modellen in AD4S und stellt sicher, dass die AD4S-Darstellung entsprechend diesen Kontextbedingungen wohlgeformt ist. Die Modellierungsrichtlinien sind:

RL1 Das Modell enthält nur die folgenden Elemente: Startknoten, Endknoten, Ablaufendknoten, Verzweigungsknoten, Vereinigungsknoten, Entscheidungsknoten, Verbindungsknoten, Partition, Kontrollfluss, Objektfluss, Pin, Aktivität und Aktion (vgl. Abbildung 2.4).

[3]Für den Fall, dass ein Modellierungstool Profile unterstützt, sind die Richtlinien auch über Profile umsetzbar.

RL2 Jede Aktivität enthält einen eindeutigen Startknoten und einen eindeutigen Endknoten.

RL3 Jede Aktivität/Aktion muss transitiv mit dem Startknoten verbunden sein.

RL4 Jeder Fluss endet am Endknoten oder einem Ablaufendknoten. Mindestens ein Fluss endet am Endknoten.

RL5 Jede Aktivität, die eine bestimmte Funktionalität modelliert, hat einen vordefinierten domänenspezifischen Stereotypen.

RL6 Wächterausdrücke sind boolesche Ausdrücke.

RL7 Verschachtelte Aktivitäten können entweder direkt als textuelles AD4S-Modell unter Verwendung eines bestimmten Eigenschaftsfeldes oder in einem AD- oder SC-Unterdiagramm, wie beispielsweise einer Aktivität, modelliert werden. Für Einschränkungen der textuellen Sprachbeschreibung siehe Richtlinie **RL8**.

RL8 Eine Unterfunktionalität mit weniger als vier Zweigen (einschließlich Unterzweigen) des Entscheidungsknotens (die zwei Wächterausdrücken entspricht) kann durch eine textuelle Liste der Wächter spezifiziert werden, die die Zweige innerhalb des Eigenschaftsfeldes der Aktivität angibt.

RL9 Wenn ein Kontrollfluss mit einem Objektfluss zusammenfällt, kann der Kontrollfluss ausgelassen werden, um die visuelle Komplexität des Modells zu verbessern.

Die Richtlinien definieren die interdisziplinäre modellbasierte Zusammenarbeit und verhindert, dass diese Zusammenarbeit der Domänenexpert*innen zu semiformalen Modellen führt [Bro06]. Die Absicht von **RL1** ist es, die Verwendung der AD-Elemente einzuschränken, um ein gemeinsames Verständnis der Domänenexpert*innen aufzubauen. So wird die Erstellung formaler Modelle erleichtert [ADS18]. Die Richtlinien **RL2**, **RL3** und **RL4** zielen zum einen darauf ab, einen eindeutig identifizierbaren Kontrollfluss des ADs sicherzustellen. Zum anderen soll gewährleistet werden, dass die Diagramme keine Deadlocks enthalten. Aufgrund der Richtlinien ist es möglich, auf einen Blick zu erkennen, wie die Funktion beginnt (Startknoten) und wo sie endet (Endknoten). Da alle Elemente miteinander verbunden sind, sind die Pfade durch das AD eindeutig. Diese Eindeutigkeit ist eine Grundvoraussetzung für die maschinelle Pfadvalidierung. Die Richtlinie **RL5** greift die Stereotypen der Aktivitäten auf und dient der Differenzierung von Funktionszugehörigkeiten. Auf der SMArDT-Ebene B sind dies zum Beispiel Stereotypen von Sicherheitsfunktionen, wie die funktionale Sicherheit, die Priorität der Aktivität für die Entwicklung oder die Zuordnung zu bestimmten Funktionsgruppen (vgl. Prioritätskriterien in Abschnitt 5.1). Mit der Richtlinie **RL6** soll eine eindeutige Formalität für ein gemeinsames Verständnis und eine Maschinenlesbarkeit sichergestellt werden. Die Richtlinie **RL7** legt die Möglichkeit einer textuellen oder grafischen Repräsentation für ein Diagramm offen. Diese Möglichkeit wird mit der Richtlinie **RL8** eingeschränkt. Die Kombination der Richtlinien führt zu einem einheitlichen Vorgehen, die auf der

grafischen Repräsentation basiert und mit textuellen AD4S-Elementen ergänzt werden kann. Diese Kombination begrenzt schließlich die visuelle Komplexität der Diagramme hinsichtlich der Größe und der Anzahl an Aktivitäten. In Abbildung 6.6 sind beispielhaft die Wächterausdrücke in einer AD4S-konformen grafischen Repräsentation und einer textuellen Liste von Wächterausdrücken dargestellt.

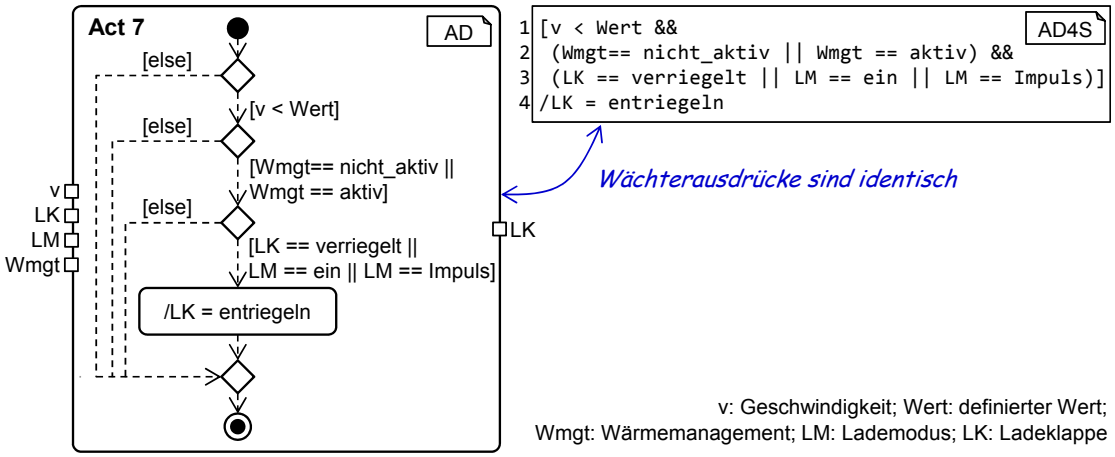


Abbildung 6.6: Beispielgegenüberstellung von Wächterausdrücken in grafischen und textuellen Repräsentationen von AD4S

Die Aktivität `Act 7` in Abbildung 6.6 `entriegelt` die `Ladeklappe (LK)` abhängig vom Zustand des Fahrzeugs. In diesem Fall ist auf der textuellen Repräsentation mit wenigen Blicken ersichtlich, welchen Status und welche Signalwerte die Aktion `Ladeklappe = entriegeln` hervorrufen. In der grafischen AD-Repräsentation ist das Erkennen dieses Zusammenhangs weniger übersichtlich. Die „fehlenden“ Definitionen der Eingänge und Ausgänge der Aktivität (vgl. Abbildung 6.3) sind damit begründet, dass es sich um redundante Information aus der übergeordneten Aktivität handelt und deshalb die Definition ausgelassen werden kann. Die Richtlinie **RL9** soll wie auch **RL7** und **RL8** die visuelle Komplexität der AD-Sprache reduzieren. Diese Reduzierung wird in **RL9** mit der Verringerung von parallelen Kontroll- und Objektflüssen mit identischer Quelle und Ziel bewerkstelligt. Der Kontrollfluss ist demnach implizit im gleichen Objektfluss enthalten. Neben der visuellen Komplexität sollen die Modellierungsrichtlinien **RL7**, **RL8** und **RL9** auch den Modellierungsaufwand für die Modellierer*innen minimieren.

Kapitel 7

Realisierung des Testfallgenerators

Die Implementierung, Erweiterung und Evaluation des Testfallgenerators wurde in Zusammenarbeit mit Imke Drave, Dr. Timo Greifenberg, Steffen Hillemacher, Dr. Christoph Schulze und drei aufeinanderfolgenden Masterarbeiten [Mil17, Koh19, Wes19] im Rahmen eines Industrieprojektes mit SMArDT [KMS+18, DGH+18, DGH+19] realisiert. In diesem Zuge wurde für die effiziente Umsetzung der in Kapitel 5 beschriebenen Anforderungen eine modellbasierte (voll)automatisierte Testfallerstellung eingesetzt. Abbildung 7.1 gibt eine vereinfachte Übersicht über den Prozess der Testfallerstellung und der Rolle des SysML-Modellierungswerkzeugs, des Testfallgenerators und der Testautomatisierungssoftware. Die relevanten Modellinformationen werden aus dem SysML-Modellierungswerkzeug in XML exportiert. Der Testfallgenerator importiert dieses Modell im XML-Format und erstellt die Teställe mit den vorhandenen Modellinformationen. Diese Testfälle können für beliebige Formate wie XML oder java angepasst werden, gegebenenfalls mittels einer Testautomatisierungssoftware verwendet und an einer Testumgebung ausgeführt werden.

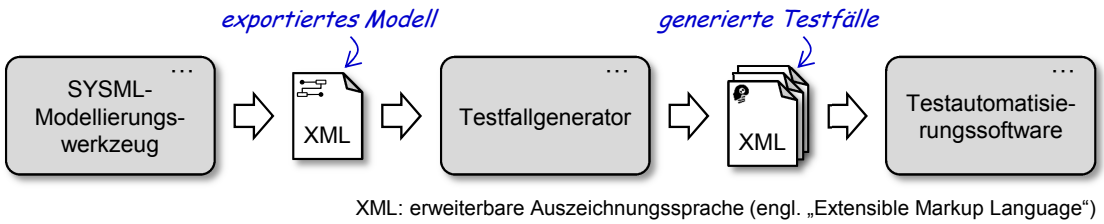


Abbildung 7.1: Vereinfachte Übersicht über den Prozess der automatisierten Testfallerstellung

Der Testfallgenerator erstellt Testfälle auf der Basis von Modellinformationen aus den SysML-Diagrammtypen AD und SC. Die modellbasierte Testfallerstellung wird in diesem Kapitel erläutert. Zu diesem Zweck wird in Abschnitt 7.1 die Testfallerstellung aus SysML-Aktivitätsdiagrammen erläutert. Die darauf basierende Testfallerstellung aus SysML-Zustandsdiagrammen wird in Abschnitt 7.2 vorgestellt. Anschließend wird in Abschnitt 7.3 auf die generierten Artefakte des Testfallgenerators eingegangen.

7.1 Testfallerstellung aus SysML-Aktivitätsdiagrammen

Ein SysML-Diagrammtyp, aus dem Modellinformationen zur Testfallerstellung verwendet werden können, ist das AD. Abbildung 7.2 illustriert die Aktivitäten von der Modellausleitung bis hin zum fertigen Testfall. Die grundlegenden Aktivitäten sind

1.) das Analysieren des SysML-Modells im jeweiligen SysML Modellierungswerkzeug,

2.) das Transformieren des intermediären ADs,

3.) die Aufbereitung der Pfade des ADs,

4.) die Validierung der Pfade und

5.) die Erstellung der Testfälle im gewünschten Format.

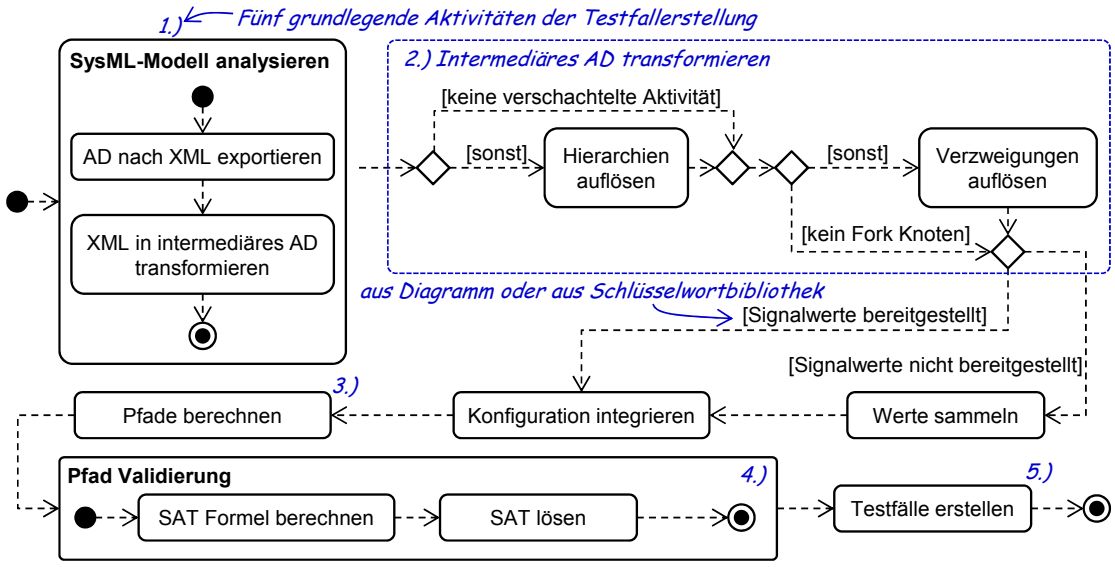


Abbildung 7.2: Ablauf der Methodik bei der automatisierten Testfallerstellung, ohne die Berücksichtigung von verwendeten Artefakten [DGH+18]

Im weiteren Verlauf wird in Unterabschnitt 7.1.1 auf die Analyse der Modellinformationen, die Übersetzung in eine DSL und der Transformation in einen Aktivitätsgraphen eingegangen. Die Methodik für das Auflösen von Hierarchien wird in Unterabschnitt 7.1.2 und das Auflösen von Verzweigungen in Unterabschnitt 7.1.3 beschreiben. Unterabschnitt 7.1.4 erläutert den Umgang mit unvollständigen Wertebereichen. Ergänzend wird in Unterabschnitt 7.1.5 die Integration einer Schlüsselwortbibliothek beschrieben. Anschließend werden in Unterabschnitt 7.1.6 die Konfigurationsmöglichkeiten, in Unterabschnitt 7.1.7 die Pfadkalkulation und in Unterabschnitt 7.1.8 die Pfad-Validierung

erläutert. Unterabschnitt 7.1.9 geht auf die Erstellung der Testfälle ein. Für weitere technische Details der Umsetzung über den Testfallgenerator ist auf [Mil17, Wes19] zu verweisen.

7.1.1 SysML-Modell analysieren und transformieren

Die in Abbildung 7.2 beschriebene Methode erfordert eine adaptive Implementierung eines Testfallgenerators. Das Ziel ist es, mit SMArDT die Testfallerstellung sicherzustellen. Es wird kein spezifisches SysML-Modellierungswerkzeug vorausgesetzt. Um die nötigen Voraussetzungen zu schaffen, wird eine Modellierungssprache benötigt, die grundsätzlich universell einsetzbar und zugleich spezifisch genug für die Testfallerstellung ist. Zu diesem Zweck wurde mit der MontiCore Language Workbench die DSL AD4S entwickelt (vgl. Abschnitt 6.2). AD4S-grammatikkonforme ADs können daher aus jedem Modellierungswerkzeug einfach in AD4S übersetzt werden. Abbildung 7.3 illustriert den Transformationsprozess der Modellinformationen im Modellierungswerkzeug in AD4S.

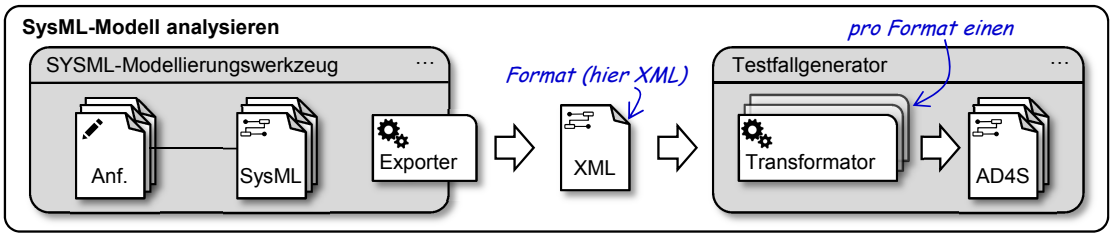


Abbildung 7.3: Prozess der Analyse und Export relevanter Informationen und die Transformation in AD4S

Nachdem die relevanten Informationen wie Aktionen, Kontrollflüsse, Wächterbedingungen und Anforderungen aus dem SysML-Modellierungswerkzeug exportiert wurden, werden diese in AD4S transformiert. Diese Methode gewährleistet eine Unabhängigkeit von Modellierungswerkzeugen. Im Falle eines Wechsels oder einer Vielzahl an Modellwerkzeugen müssen nur die jeweiligen Exporter angepasst werden. Außerdem kann mittels einer definierten Grammatik für AD4S die MontiCore Language Workbench genutzt werden. Diese stellt den Parser und Hilfskomponenten bereit, wodurch es komfortabel möglich ist, Analysen und Transformationen auf der abstrakten Syntax von AD4S zu realisieren [GKR+08, KRV10, HR17, HKR21].

Es ist zu beachten, dass der Testfallgenerator nicht zwischen Modellen verschiedener Abstraktionsebenen unterscheidet. Alle AD4S-konformen Modelle und deren Informationen werden gleich behandelt. Folglich verfährt der Testfallgenerator mit Unterabschnitt 4.2.2 unterspezifizierten Informationsflüsse auf der SMArDT-Ebene B genauso wie für (vollständig detaillierte) Signale der Ebene C. Im Verlauf dieses Kapitels umfasst somit der Begriff Signal (des Testfallgenerators) Informationsflüsse und Signale in SMArDT, weil beide im Testfallgenerator als Signale behandelt werden.

Aktivitätsgraph

Nachdem der Testfallgenerator das Modell in AD4S übersetzt hat, wird ein Aktivitätsgraph erstellt, sodass alle relevanten Informationen der Knoten vorliegen. Per Definition ist der Aktivitätsgraph ein gerichteter Graph [GD18]. Dieser Graph umfasst verschiedene Arten von Knoten (vgl. Zeilen 15 bis 21 in Abbildung 6.2) und repräsentiert die testfallgenerator-interne Darstellung des Diagramms. Der Aktivitätsgraph ist eine Schnittstelle zwischen MontiCore und der Komponenten der Testfallerstellung. Mit dieser Schnittstelle kann die Struktur des ADs den Anforderungen des Testfallgenerators angepasst werden. Diese Struktur nutzt der Testfallgenerator für das Erstellen von Testfällen, da sie alle erforderlichen Informationen enthält und Testfallgenerator-Algorithmen effizient, einheitlich und verständlich angewendet werden können. Der Aktivitätsgraph unterscheidet zwischen Signalen, Knoten und Kanten.

Signale stellen Variablen übergeordneter Programmiersprachen in einem Aktivitätsgraphen dar. Wie Variablen werden Signale zunächst deklariert und dann verwendet. Signale können daher einem konkreten Wert zugeordnet werden. Sie können somit zu unterschiedlichen Zeitpunkten unterschiedliche Werte annehmen.

Ein Signal ist mittels seines Namens, seines Signaltyps und seines Deklarationstyps identifizierbar. Der Signaltyp entspricht einem Satz von Konstanten und beschreibt, welche Zustände das Signal annehmen darf. Nur Signale desselben Typs können einander zugeordnet sein. Der Deklarationstyp des Signals bestimmt, wo und wie das Signal verwendet wird. Für den Deklarationstypen stehen die Alternativen *Eingang*(-ssignale), *Ausgang*(-ssignale) oder *temporär* zur Verfügung. Als temporär werden Signale deklariert, wenn sie nur innerhalb eines Aktivitätsgraphen, ohne Verbindung nach außen, verwendet werden. Eingangssignale dürfen per Definition nicht überschrieben werden. Ausgangssignalen und temporäre Signalen können beliebige Werte zugeordnet werden.

Knoten speichern die Graphenstruktur mithilfe von Informationen über eingehende und ausgehende Kanten. Die Knoten in dem Aktivitätsgraphen entsprechen den Knoten der definierten AD4S-Grammatik (vgl. Abschnitt 6.2). Im Aktivitätsgraphen werden Knoten durch ihre Namen unterschieden, weshalb Knotennamen innerhalb eines Aktivitätsgraphen eindeutig sein müssen. Zusätzlich kann jeder Knoten, wie in AD4S, einen Satz von natürlichsprachlichen textuellen Anforderungen enthalten.

Kanten des Aktivitätsgraphen entsprechen ebenfalls den Kanten der definierten AD4S-Grammatik (vgl. Abschnitt 6.2). Auch sie werden über einen Quellknoten und einem Zielknoten definiert und sind mit einer ID versehen.

Die auf der Basis dieses Aktivitätsgraphen ausgeführten Transformationen werden anhand des Beispiels in Abbildung 7.4 Schritt für Schritt erklärt. Die Funktionsweise des Testfallgenerators und das Beispiel wurden teilweise in [DGH+18] erstmals veröffentlicht.

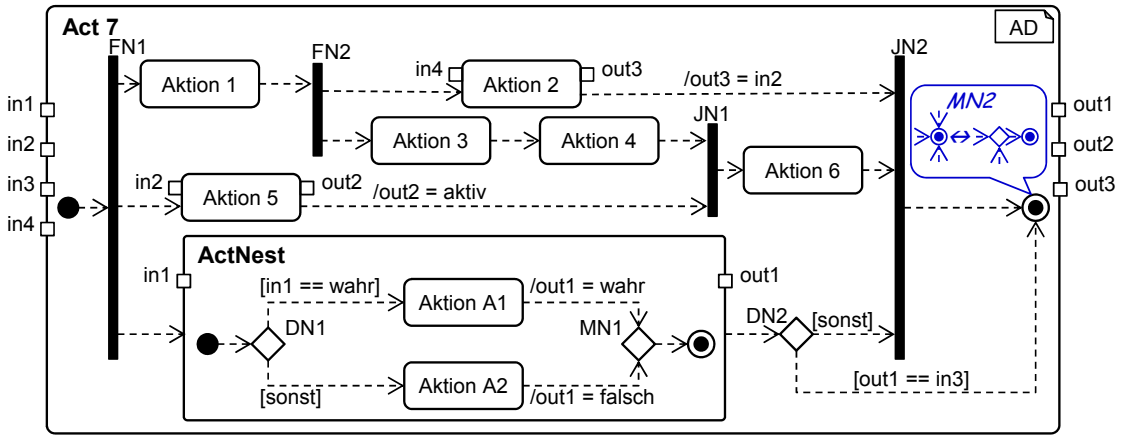


Abbildung 7.4: Beispiel einer SMArDT-Ebene B (aus [DGH+18] erweitert)

Aktivität `Act 7` aus Abbildung 7.4 mit den verschachtelten Verzweigungen und Vereinigungen ist ein vereinfachtes aber typisches Beispiel für eine SMArDT-Ebene B. Die Verzweigungen dienen nicht nur dazu, Gleichzeitigkeit auszudrücken, sondern die (noch) nicht festgelegte Reihenfolge der Aktionen beziehungsweise Aktivitäten zu modellieren (vgl. Abschnitt 6.2). Noch einmal ist zu betonen, dass für dieses Beispiel vorausgesetzt wird, dass der AD4S-Aktivitätsgraph wohlgeformt ist. Das heißt, es liegen weder dynamische noch statische Deadlocks vor und alle Modellierungsrichtlinien für AD4S sind eingehalten (vgl. Abschnitt 6.2). Auch wenn Abbildung 7.4 vielleicht auf den ersten Blick nicht wohlgeformt und wohldefiniert ist dies der Fall. Zwar sind nicht alle Verzweigungen (`FN1, FN2`), Vereinigungen (`JN1, JN2`), Alternativen (`DN1, DN2`) und Zusammenführungen (`MN1`) eins zu eins abgestimmt, aber es liegen weder dynamische noch statische Deadlocks vor und jeder Fluss endet am Endknoten.

7.1.2 Hierarchieauflösung

Hierarchien in Diagrammen erlauben den Modellierenden, zusammenhängende Unterfunktionalitäten zu strukturieren, auf mehrere Ebenen zu verteilen und gegebenenfalls Komplexität zu „verstecken“. Für die Benutzer*innen wird eine bessere Übersicht gewährleistet und folglich das Verständnis verbessert. Für die Pfadberechnung des Testfallgenerators wird allerdings eine flache Diagrammhierarchie benötigt. Daher werden verschachtelte Aktivitäten aufgelöst. Abbildung 7.5 stellt beispielhaft die Hierarchieauflösung anhand des Beispiels in Abbildung 7.4 dar.

Die Hierarchieauflösung löst verschachtelte Aktivitäten auf. Zu diesem Zweck wird der Startknoten einer verschachtelten Aktivität entfernt. Der nun frei liegende Knoten wird mit dem letzten, der verschachtelten Aktivität vorhergehenden Knoten, des überordneten Diagramms, mit einer Kante verbunden. Analog wird mit dem Endknoten des verschachtelten ADs verfahren. Der Endknoten der verschachtelten Aktivität wird entfernt und deren letzter Knoten wird mit dem ersten ehemals verbunden übergeordneten Knoten

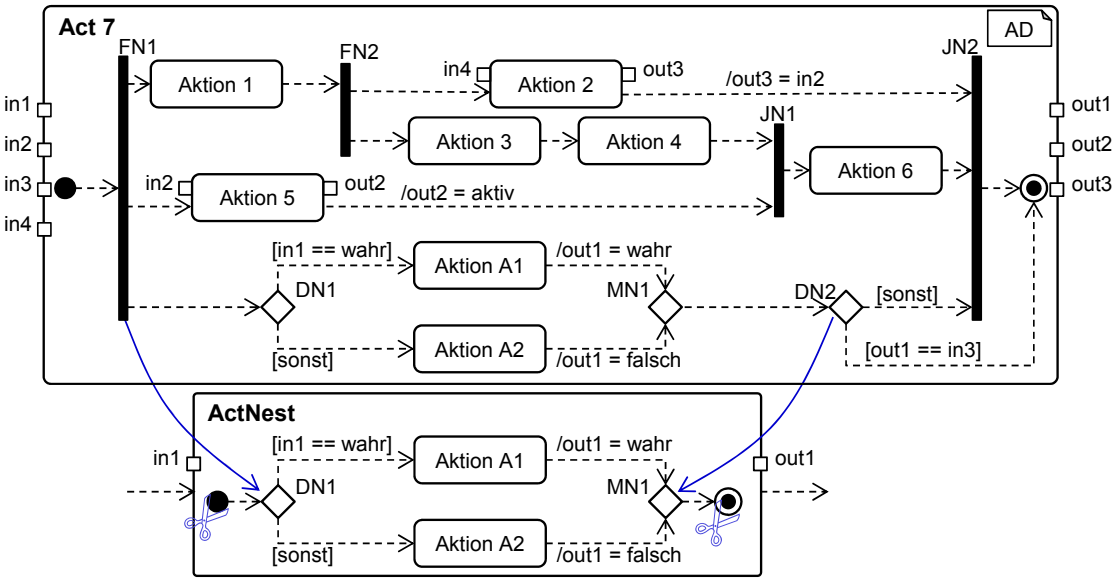


Abbildung 7.5: Darstellung der Hierarchieauflösung von Abbildung 7.4

verbunden. In Abbildung 7.5 wird der Start- und Endknoten der verschachtelten Aktivität, `ActNest` gelöscht und `FN1` mit `DN1` sowie `MN1` mit `DN2` verbunden. Falls das AD mehrere Hierarchieebenen beinhaltet werden die Diagrammhierarchien rekursiv aufgelöst.

7.1.3 Verzweigungsreduzierung

Nach der Hierarchieauflösung folgt eine Auflösung der Verzweigungen. Verzweigungen in ADs werden in SMArDT modelliert, um Unabhängigkeit in der Ausführungsreihenfolge bis hin zur überlappenden Ausführung (Parallelität) der Aktionen beziehungsweise Aktivitäten zu modellieren (vgl. Abschnitt 6.2). Beispielsweise werden Domänenkonzepte aus bestehenden kontinuierlichen Regelsystemen unverändert in die SysML überführt (vgl. Abschnitt 6.1). Die in dieser Arbeit betrachteten Testumgebungen und Testfälle benötigen allerdings einen sequenziellen Charakter (vgl. Unterabschnitt 5.3.2). Deshalb werden die Verzweigungen und deren Vereinigungen aufgelöst.

Wenn Abhängigkeiten nicht explizit modelliert sind, wie Materie, Energie oder Information (vgl. Unterabschnitt 4.2.2), kann davon ausgegangen werden, dass die Reihenfolge austauschbar ist, solange die Reihenfolge jedes einzelnen Flusses erhalten bleibt. Diese Annahme kann getroffen werden, weil die Diagramme in der SMArDT-Ebene B einen hohen Abstraktionsgrad aufweisen. Zusätzlich kann die Annahme getroffen werden, dass parallele Pfade linear sind. Um eine Unabhängigkeit der Signale zu gewährleisten, wird vorher geprüft, ob die Elemente der Pfade unterschiedliche Signale verwenden. Trifft dies zu, sind die Annahmen gültig. Die Annahme der austauschbaren Reihenfolge und die der Unabhängigkeit sind eine Grundvoraussetzung für eine valide Verzweigungsreduzierung. Abbildung 7.6 stellt den ersten Teil der Verzweigungsreduzierung von Abbildung 7.5 dar.

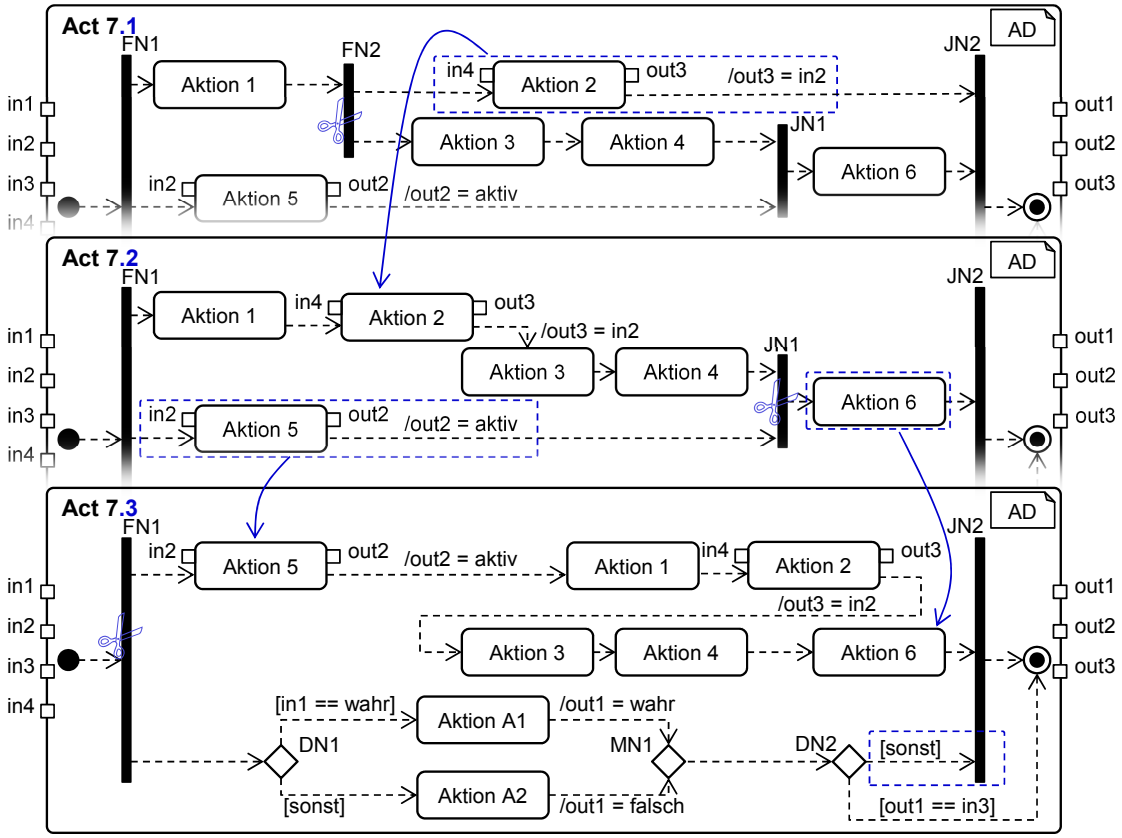


Abbildung 7.6: Ausschnitt aus der Abfolge der Verzweigungsreduzierung von Abbildung 7.4. `Act 7.1`, `Act 7.2` und `Act 7.3` sind Versionen des gleichen ADs.

Bei einer Vielzahl von geschachtelten Verzweigungsknoten beginnt die Reduzierung an einem der innersten Verzweigungsknoten. Ausgehend vom ersten angetroffenen Verzweigungsknoten durchläuft die Transformation das AD rekursiv, bis sie den innersten Verzweigungsknoten findet. Wenn jedoch kein solcher Knoten gefunden werden kann, muss entweder einer der parallelen Pfade an einem Ablaufendknoten enden oder beide Pfade müssen am Endknoten enden. Das heißt, der Startknoten und Endknoten, als die äußersten Knoten der Aktivität, werden als Synchronisationspunkt verwendet. Im Fall von Abbildung 7.6 ist der innerste Verzweigungsknoten `FN2` und der innerste Vereinigungsknoten `JN1`. Ausgehend von `FN2` werden alle parallelen Pfade berechnet und die Anfänge und Enden der Pfade markiert. Die Verzweigungsreduzierung „sequenziert“ immer zwei parallele Pfade und verbindet sie an ihrem ersten gemeinsamen Verbindungsknoten, bis nur noch ein Pfad übrig ist. Für Abbildung 7.6 bedeutet dies, dass `Aktion 2` an die ausgehende Kante von `Aktion 1` und die eingehende Kante von `Aktion 3` eingebettet wird (vgl. `Act 7.2`). Die Reihenfolge mit einer möglichen Abhängigkeit von `Aktion 3` und `4` bleibt somit erhalten. Anschließend ist `JN1` der innerste Verzweigungs-

beziehungsweise Vereinigungsknoten. Da der gemeinsame Knoten von `Aktion 5` mit dem parallelen (oberen) Pfad `FN1` ist, wird `Aktion 5` vor `Aktion 1` eingebettet (vgl. `Act 7.3`). `Aktion 6` wird nach `Aktion 4` eingebettet. Danach wird das letzte Verzweigungs-/Vereinigungsknoten-Paar reduziert (vgl. `Act 7.3`). Die Gruppe aus `DN1`, `Aktion A1`, `Aktion A2` und `MN1` wird nach `Aktion 6` im parallelen (oberen) Pfad eingebettet. Die `else`-Kante der Alternative `DN2` teilt sich mit der ausgehenden Kante von `Aktion 6` den Vereinigungsknoten `JN2`, während die andere Kante `out1 == falsch` zum Endknoten führt die alle laufenden Abläufe der Aktivität sofort beendet. Um die beiden Kanten zusammenzuschließen, werden diese mit einer neuen Zusammenführung `MN2` verbunden (vgl. Abbildung 7.7). Bezüglich dieses Schrittes wird noch einmal darauf hingewiesen, dass das Zusammenführen der parallelen Pfade in Abbildung 7.7 aus `Act 7.3` in Abbildung 7.6 nur möglich ist, da davon ausgegangen wird, dass es sich nicht um eine gleichzeitige Ausführung handelt und die Reihenfolge nicht relevant ist.

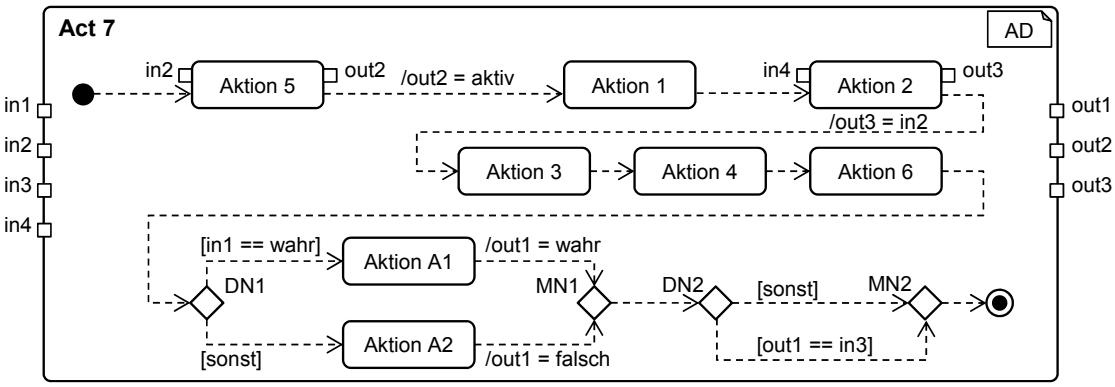


Abbildung 7.7: Das AD aus Abbildung 7.4 nach der Transformation

Nachdem die Transformation des ADs durchgeführt wurde, sind alle verschachtelte Aktivitäten, Verzweigungs- und Vereinigungsknoten eliminiert und die Operationen auf der Basis eines „flachen“ Aktivitätsgraphen sind möglich.

7.1.4 Wertebereich-Vervollständigung ohne Schlüsselwortbibliothek

Im nächsten optionalen Schritt wird der Wertebereich der Signale gesammelt und vervollständigt beziehungsweise berechnet. Wie in Unterabschnitt 7.1.5 beschrieben, wird im Idealfall extern ein Satz von Werten, die Signale annehmen können als Eingang für den Generator bereitgestellt. Allerdings gibt es für Signale in den Diagrammen der abstrakten SMArDT-Ebenen, wie der Ebene B in frühen Entwicklungsphasen häufig keine voll umfängliche Spezifikation der Werte in einer globalen Signaldatenbank oder Schlüsselwortbibliotheken. Um diese Unterspezifikation der Diagramme zu kompensieren, werden die Wertebereiche der Signale aus dem Diagramm gesammelt und komplementiert. Grundsätzlich muss bei dem in diesem Abschnitt vorgestellten Ansatz berücksichtigt werden, dass die Werte nicht richtig erfasst werden können, wenn es Fehler im modellierten

Diagramm gab oder falsche Antonyme wie `nicht_aktiv` statt `inaktiv` erstellt werden.

In der Wertebereich-Vervollständigung werden vorerst alle möglichen Werte für jedes der Signale des Diagramms in einem Durchlauf gesammelt. Während dieses Durchlaufs werden die Eingänge, Ausgänge, Wächterausdrücke und Signalzuweisungen anfangs auf die benötigten Informationen überprüft. Bei Signalen wird die Menge der möglichen Werte durch den entsprechenden Wert ergänzt, sobald die Signale in einem Wächterausdruck oder einer Zuweisung erscheinen. Bei Wächterausdrücken wird die eigentliche Semantik vernachlässigt und jede einzelne Bedingung wird separat betrachtet. Jeder Vergleich eines Signals mit einem konkreten Wert veranlasst den Testfallgenerator, den Wert in einem Typ für das Signal zu speichern. Werden allerdings zwei Signale miteinander verglichen, zum Beispiel werden mit `out1 == in3` die beiden in dieser Bedingung auftretenden Signale in der Menge der „vergleichbaren Signale“ für die nächsten Schritte gespeichert. Das heißt, alle Werte, die `out1` annehmen kann, sind auch für `in3` gültig (vgl. Abbildung 7.7). Falls keines der Signale im Diagramm einem konkreten Wert zugewiesen oder mit einem konkreten Wert verglichen wird, bleibt dieses als Paar `out1, in3` für vergleichbare Signale gespeichert. Die Signalpaare vom gleichen Typ müssen deshalb gebildet werden, weil es sonst bei einem Vergleich oder einer Signalzuweisungen zu Kompatibilitätsproblemen der Objektmengen mit den darauf definierten Operationen kommen kann. Daher müssen die Typen dieser Signale, die verglichen werden, erweitert und angepasst werden, um eine Vergleichbarkeit zu gewährleisten. Wenn die Typen der beiden Signale bereits genau denselben Satz von Werten haben, wird dieser Schritt übersprungen. Besitzen zwei Signale nicht genau denselben Satz, sind aber desselben Typs, wird die Vereinigung der beiden Sätze berechnet. Für diese Vereinigung wird ein neuer gegenseitiger Typ definiert und auf beide Signale übertragen. Diese Angleichung von Typen wird auf beliebig viele Signalen angewendet, wenn eine direkte und indirekte Typenkorrelation besteht. In Abbildung 7.7 korreliert `in3` mit `out1`. Wenn nun ein Signal `Sig1` mit `out1` korreliert ist, bedeutet dies, dass `in3` und `Sig1` ebenfalls korreliert sind und dass sie ebenfalls vom gleichen Typ sein sollten. Die Korrelationsbeziehung ist demzufolge transitiv. Sind alle Typen initialisiert und ordnungsgemäß abgeleitet, wird eine Verifizierung durchgeführt, um die minimal zulässige Anzahl von möglichen Werten in einem Typ zu prüfen. Wenn ein Datentyp nur ein Element enthält, fügt der Algorithmus automatisch einen negierten Wert hinzu, der für die korrekte Behandlung von `sonst`-Bedingung erforderlich ist. Tabelle 7.8 gibt eine Übersicht über das Ergebnis nach der Wertebereich-Vervollständigung aus dem Beispiel Abbildung 7.7.

Zunächst wird das Signal `in1` betrachtet. `in1` wird in einem Wächterausdruck auf den Wert `wahr` überprüft, was das Hinzufügen des Wertes zu dem Wertebereich von `in1` nach sich zieht. Da `in1` keine weiteren Vorkommen im Diagramm hat, wird am Ende das Antonym `falsch` dem Wertebereich hinzugefügt (vgl. Tabelle 7.8). Anschließend wird das Signal `in2` betrachtet. Da `in2` und `in4` nicht innerhalb einer Wächterbedingung oder einer Signalzuweisung verwendet werden, bleiben ihre Wertebereiche leer und der Generator warnt vor einem nicht verwendeten Eingang. Diese unvollständigen Wertebereiche können im nächsten Schritt des Testfallgenerators, `Konfiguration integrieren`, manuell ergänzt werden (Abbildung 7.2). Für `out1` sind zwei Werte, `wahr` und `falsch`,

Tabelle 7.8: Ergebnis der Wertsammlung aus Abbildung 7.7

Signal	Tatsächlicher Wert	Antonym
in1	wahr	falsch
in2		
in3	wahr, falsch	
in4		
out1	wahr, falsch	
out2	aktiv	nicht_aktiv
out3		

aus den Signalzuweisungen im Diagramm vorhanden, daher ist keine Bildung von Antonymen nötig. Da der Wert von `in3` dem Signal `out1` zugewiesen wird, kann ebenfalls der gleiche Wertebereich von `out1` für `in3` übernommen werden. Das Signal `out2` wird im Diagramm, mit einer Signalzuweisung auf `aktiv` gesetzt wird. Allerdings ist im ganzen Diagramm kein weiterer Wert beziehungsweise Antonym, für `out2` für die Wertsammlung vorhanden. Daher generiert der Algorithmus den ergänzenden Wertebereich für `aktiv` und fügt als Antonym den Wert `nicht_aktiv` hinzu (vgl. Tabelle 7.8). Da das Signal `out3` auf das nicht definierte Signal `in2` verweist, bleibt der Wertebereich ebenfalls leer und der Generator gibt eine Warnung aus. Für ein besseres Verständnis des Testfallgenerators wird kurz gedanklich die Situation konstruiert, dass die Signalzuweisung nicht `out3 = in2`, sondern `out3 = in1` ist. Wäre `out3 = in1`, so hätte die Verzweigungsreduzierung aus Unterabschnitt 7.1.3 keinen Einfluss auf die Signalzuweisung, und die Wertebereich-Vervollständigung hätte `out3` den Wertebereich von `in1` mit `wahr` und `falsch` zugewiesen.

Die Wertebereich-Vervollständigung ist ein nützliches Hilfsmittel um die unterspezifizierten Diagramme der frühen Entwicklungsstände der SMArDT-Ebene B zu verifizieren. Allerdings können die ermittelten Wertebereiche von realen Wertebereichen abweichen. Zum Beispiel könnte das Antonym von `aktiv` nicht `nicht_aktiv` sondern `passiv` sein. Derartige Abweichungen führen zum manuellen Nacharbeiten. Zusätzlich sind häufig für eine automatisierte HIL- oder VIL-Testumgebungen weitere Schritte, wie die Zuweisung der Signale aus dem Diagramm zu automatisierten Testschritten, nötig. Aus diesem Grund wird empfohlen, den im nächsten Abschnitt vorgestellten Ansatz mittels Schlüsselwörtern und mindestens einer Signaldatenbank zu verwenden.

7.1.5 Wertebereich-Vervollständigung mit Schlüsselwortbibliothek

Eine vorhandene Schlüsselwortbibliothek kann dazu beitragen den Wertebereich der Signale für generierte Testfälle zu vervollständigen. Die Schlüsselwortbibliothek erweitert folglich die Semantik von AD4S für Testzwecke. Abbildung 7.9 stellt den Zusammenhang zwischen dem Modell, der Schlüsselwortbibliothek und dem Testfallgenerator dar.

Jedes Modell, das vom Testfallgenerator verwendet werden soll, muss mit der DSL

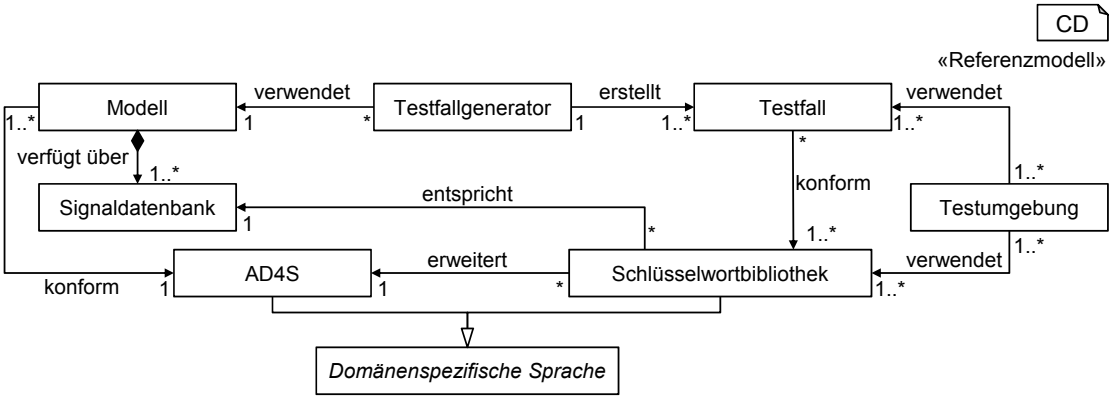


Abbildung 7.9: Ausschnitt des Zusammenhangs des Modells und der Testumgebung mit der Schlüsselwortbibliothek

AD4S-konform oder konvertierbar sein. Für eine automatisierte Testausführung an einer schlüsselwortunterstützenden Testumgebung, müssen die erstellten Testfälle konform mit der Schlüsselwortbibliothek sein. Die Konformität der Testfälle zu der Schlüsselwortbibliothek wird insofern gewährleistet, dass die Testschritte des Testfalls Schlüsselwörter der Schlüsselwortbibliothek sind (vgl. Abschnitt 3.2). Die Schlüsselwortbibliothek entspricht der Signaldatenbank, die ein Teil des Modells ist. *Entsprechen* bedeutet, dass die Semantik der (abstrakten) Signale des Modells mit denen der Schlüsselwörter übereinstimmt. Falls nötig kann der Wertebereich für Schlüsselwörter für Testzwecke erweitert werden. Ist ein Wertebereich eines Signals beispielsweise von 0 (Minimaler Wert) bis 100 (Maximaler Wert) ausgelegt, kann es für Testzecke sinnvoll sein diesen Wertebereich von -10 bis 110 zu erweitern, um Grenzbereiche zu Testen.

Theorie und Praxis 7.1 (Existierende Schlüsselwortbibliotheken im Modell)**.** *Um Inkonsistenzen zu vermeiden, ist es sinnvoll eine zentrale, eindeutige und verlässliche Datenbasis zu verwenden (vgl. Abschnitt 5.1). Infolgedessen werden die Informationen für die Schlüsselwortbibliothek aus der Signaldatenbank des Modells generiert (vgl. Abbildung 7.10). Allerdings ist dies aufgrund der vorhandenen etablierten Artefakte und Strukturen nicht immer möglich (vgl. Theorie und Praxis 4.3). Voraussichtlich gibt es in bestehenden Unternehmen schon eine etablierte zentrale Datenbank [SZ16] und einen oder mehrere Schlüsselwortkataloge.*

Abbildung 7.10 stellt ein Beispiel aus der Theorie einem Beispiel aus der Praxis gegenüber. Ein möglicher Ansatz mit einem solchen Praxisszenario umzugehen ist initial die Signaldatenbank des Modells aus der externen Signaldatenbank zu generieren. Danach ist es möglich, die externe Signaldatenbank aus der Signaldatenbank des Modells zu manipulieren. Infolgedessen bleibt die externe Signaldatenbank für etablierte Prozesse erhalten und gleichzeitig können Inkonsistenzen vermieden werden. Diese Konsistenz kann mit der „Master“-Rollenübergabe der externen Signaldatenbank auf die Signaldatenbank des Modells gewährleistet werden. Analog verhält es sich mit dem Modell, beziehungsweise der Signaldatenbank und der Schlüsselwortbibliothek. Über einen initialen Import werden die

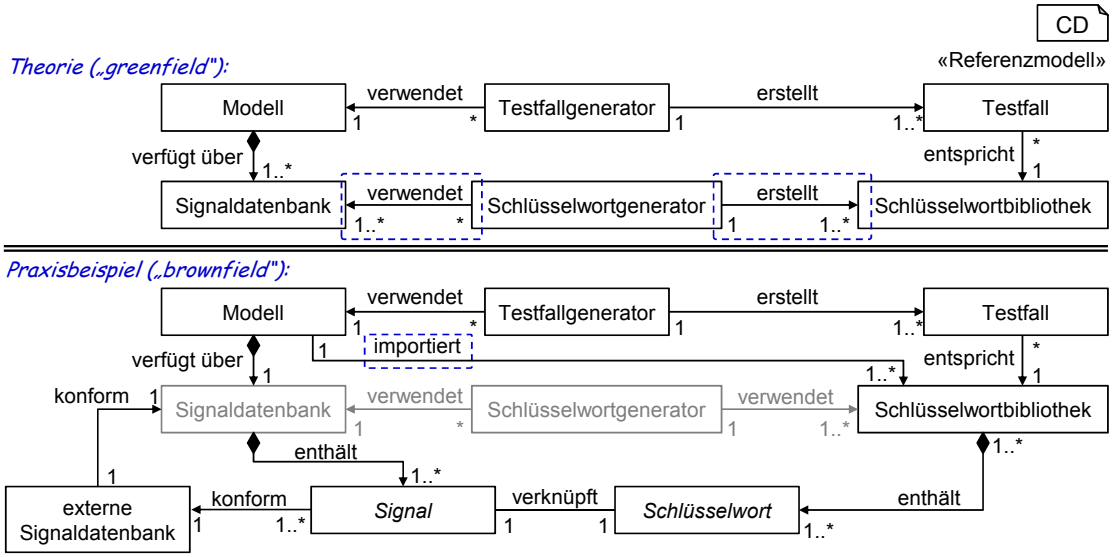


Abbildung 7.10: CD der Verknüpfung des Modells und der Testumgebung mit dem Fokus der Schlüsselwortbibliothek

Informationen des Modells und die relevanten Signale für den Test mit Schlüsselwörtern verknüpft beziehungsweise gekennzeichnet. Mit diesem Verfahren können Inkonsistenzen von Signalen und Schlüsselwörtern, wie unterschiedliche Wertebereiche oder Benennungen Bezeichnungen, Schritt für Schritt abgebaut werden. Gleichzeitig können Modifikationen über die bidirektionale Beziehung auf der jeweils anderen Seite erkannt werden. Folglich können Modifikationen an Signalen oder Schlüsselworten auch automatisiert erkannt und vorgenommen werden. Dies erleichtert iterativ eine effizientere Lösung, mit nur einer zentralen Datenbasis, zu etablieren.

Im Rahmen dieser Arbeit wurde ein Brownfield-Ansatz umgesetzt, der für das Testen, relevante Modellsignale mit Schlüsselwörtern verknüpft (vgl. Theorie und Praxis 7.1). Dieser Ansatz wurde gewählt, da eine direkte Abhängigkeit der Schlüsselwörter und der Signale aufgrund der bestehenden Tools nicht erwünscht war. Abbildung 7.11 illustriert exemplarisch die umgesetzte Methode und deren Schritte bis zu den ausführbaren Testfällen, diese sind im Wesentlichen folgende:

1.) Modellsignale werden mit den Schlüsselwörtern verknüpft.

2.) Diese Schlüsselwortinformationen und -verknüpfungen werden bei der Testfallgenerierung berücksichtigt.

3.) Mit Hilfe der spezifischen Schlüsselwortimplementierung können die Testfälle auf der Testfallumgebung ausgeführt werden.

In Abbildung 7.11 werden in einem ersten Schritt die Signale aus der Schlüsselwortbibliothek in das Modell importiert (vgl. Abbildung 7.10). Anschließend können die

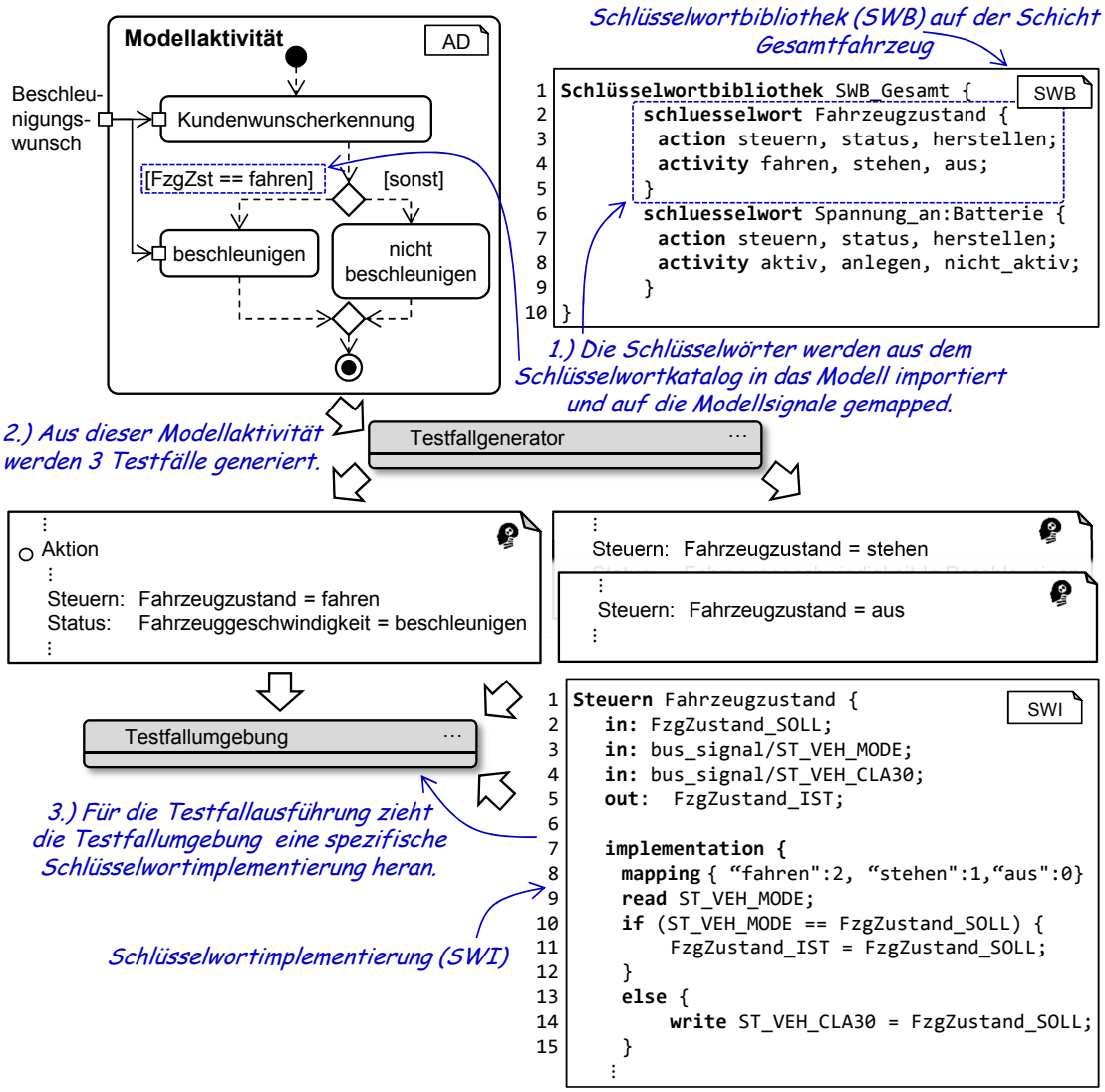


Abbildung 7.11: Schritte von einem Schlüsselwort-Modellsignal-Mapping bis zu einem ausführbaren Testfall

jeweiligen Schlüsselwörter auf die Modellsignale gemapped werden. Im Fall von Abbildung 7.11 wird das Schlüsselwort `Fahrzeugzustand` mit dem Modellsignal `FzgZst` verknüpft. In einem zweiten Schritt kann der Testfallgenerator diese Verknüpfung bei der Testfallerstellung nutzen, um schlüsselwortkonforme Testfälle zu generieren. In Abbildung 7.11 werden für jede Möglichkeit der Signalbelegung im Schlüsselwortkatalog `Steuern: Fahrzeugzustand = fahren`, `stehen` und `aus` (Siehe Zeile 4) eine Testfall erzeugt. Mit Hilfe von weiteren Konfigurationsmöglichkeiten können die Anwender*innen des Testfallgenerators die Anzahl der Testfälle auch begrenzen (siehe Unterabschnitt 7.1.6). Die generierten Testfälle können anschließend auf einer Testfallumgebung ausgeführt

werden, wenn eine Schlüsselwortimplementierung vorhanden ist. In der Schlüsselwortimplementierung werden die abstrakten Informationen des Schlüsselwortes mit konkreten Signalen wie beispielsweise Bussignalen und, wenn es erforderlich ist, einer Logik verknüpft. Hinsichtlich der Schlüsselwortimplementierung von `Steuern: Fahrzeugzustand` wird in Zeile 10 erst geprüft, in welchem Fahrzeugzustand sich das Fahrzeug befindet und dann bei Bedarf in Zeile 14 das Signal für den Schlüssel im Zündschloss manipuliert. Die Schlüsselwortimplementierung kann für jede Testfallumgebung und deren prüfstandsspezifische Charakteristika ausgelegt werden. Infolgedessen können Testfälle flexibel auf verschiedenen Testumgebungen ausgetauscht werden (vgl. Abschnitt 5.3).

Ein weiterer Vorteil von Schlüsselwörtern in dem Testfall-Generierungsprozess ist, dass die Menge der möglichen Werte für ein Signal, die mit einem Schlüsselwort verknüpft ist, endlich und eindeutig durch die Menge der Schlüsselwortwerte des zugehörigen Schlüsselworts bestimmt ist. Infolgedessen ist es möglich, den Wert eines Signals in einer „sonst“-Wächterausdruck oder eines Ungleichheitsoperators (!=) eindeutig zu bewerten. Außerdem können mittels dieser Verknüpfung selektiert Signale für die Ausleitung einer Schlüsselwortbibliothek bereitgestellt werden. Signale des Modells, die mit einem Schlüsselwort verknüpft sind, ergänzen ihren Wertebereich um den Wertebereich des Schlüsselwortes. Sind die Wertebereiche nicht konsistent, wie beispielsweise `an, aus` des Signals und `aktiv, nicht_aktiv` des Schlüsselwortes wird ein Fehler über die Konsole gemeldet. Um Inkonsistenzen von Modell und Schlüsselwortbibliothek zu vermeiden, wird eine separate Verknüpfung einzelner Signal- und Schlüsselwortwerte nicht unterstützt. Tabelle 7.12 veranschaulicht den Unterschied der Wertebereich-Vervollständigung mit und ohne der Verknüpfung von Signalen und Schlüsselwörter.

Tabelle 7.12: Ergebnis der Wertebereich-Vervollständigung aus Abbildung 7.7 mit und ohne Schlüsselwörtern. Mit Schlüsselwort verknüpfte Signale sind *kursiv*.

Signal	**Wertebereich ohne Schlüsselwortbibliothek**	**Wertebereich mit Schlüsselwortbibliothek**
in1	wahr, falsch	wahr, falsch
in2		an, aus
in3	wahr, falsch	wahr, falsch, nicht_valide
in4		
out1	wahr, falsch	wahr, falsch
out2	aktiv, nicht_aktiv	aktiv, nicht_aktiv
out3		an, aus

Aus dem Vergleich in Tabelle 7.12 geht hervor, dass die Wertebereich-Vervollständigung mit Schlüsselwörtern die Wertebereiche für Testzwecke sinnvoll ergänzen kann. Der in Tabelle 7.8 nicht vorhandene Wertebereich von Signal `in2` kann aufgrund des verknüpften Wertebereichs des Schlüsselwortes kompensiert werden. Da das Signal `out3` dem Signal `in2` zugewiesen wird, wirkt sich der kompensierte Wertebereich von `in2` ebenfalls für das Signal `out3`. Im Fall von `in3` kann der Wertebereich des verknüpften Schlüsselwortes

den Wertebereich um den Wert `nicht_valide` erweitern. Eventuell ist der Wert `nicht_valide` noch nicht in der Spezifikation vorgesehen aber für Testzecke relevant. Am Beispiel von `in4` wird deutlich, dass ohne einen definierten Wertebereich keine Aussage über eben diesen Wertebereich getroffen werden kann. Eine Verknüpfung mit Schlüsselwörtern für einen vollständigen Wertebereich ist nicht unbedingt nötig (vgl. `out1` und `out2`). Für eine Gewährleistung einer automatischen Ausführung in HIL- oder VIL-Testumgebungen und um mögliche Unterspezifikationen zu vermeiden, wird diese Verknüpfung allerdings empfohlen. Sind alle Wertebereiche vollständig, kann der Testfallgenerator die weiteren Aktivitäten aktivieren. Zusätzlich zur Wertebereich-Vervollständigung mit und ohne Schlüsselwortbibliothek ist es mit einer Konfiguration möglich, direkt Einfluss die zu erstellenden Artefakte zu nehmen.

7.1.6 Konfigurationsmöglichkeiten

Der Testfallgenerator lässt mittels einer Konfiguration bestimmte Einstellungen der testfallerstellenden Personen zu (vgl. Abbildung 7.2). Zu diesen Konfigurationen gehören neben der Integration von Vor- und Nachbedingungen für den Testfall und die Angabe von Wertebereichen der Signale auch die Begrenzung für Schleifen.

Schleifen beziehungsweise deren Ende sind für die maschinelle Verarbeitung von Diagrammen nicht ohne Weiteres entscheidbar [Tur37]. Für die maschinelle Lesbarkeit müssen Schleifen endlich sein, das heißt, dass eine Unterbrechungsbedingung nach endlich vielen Iterationen vorliegen muss. Abbildung 7.13 zeigt ein Beispiel einer endlichen Schleife für die Signalzuweisungen `/in1 = wahr`.

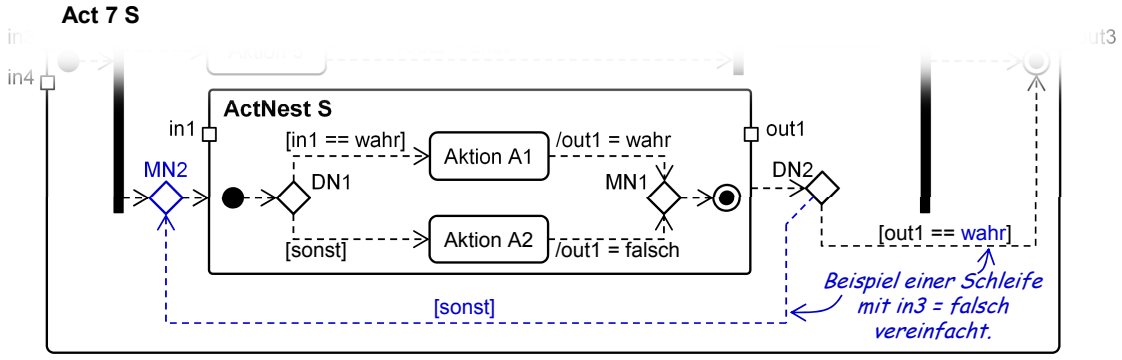


Abbildung 7.13: Diagramm ausAbbildung 7.4 mit einer eingefügten Schleifenbedingung

In Abbildung 7.13 führt die Signalzuweisungen `/in1 != wahr` dazu, dass `ActNest S` anschließend `out1 = falsch` setzt, was die `[sonst]`-Verzweigung an `DN2` in `Act 7 S` auslöst und prüft. In der folgenden Iteration wird beispielsweise die verschachtelte Aktivität `ActNest S` erneut aktiviert. Wenn in dieser Iteration `in1 = wahr` ist, wird in `ActNest S` das Signal `[out1 = wahr]` gesetzt, nachfolgend der `[out1 == wahr]`-Zweig in `DN2` genommen und die `Act 7 S` beendet. Die Schleife wäre unendlich, wenn die erste Signalzuweisung immer `/in1 != wahr` nach `MN2` wäre.

Da bei einer möglichen Pfadüberdeckung Schleifen zu einer unendlichen und damit nicht mehr durchführbaren Aufgabe werden [Tur37], wird eine beschränkte Anzahl von Schleifeniterationen wie in [Rum17] festgelegt. Zu diesem Zweck wird in der Konfiguration ein endlicher Maximalwert (1,2,3,...) festgelegt, wie oft ein Knoten in einem Pfad enthalten sein darf. Wird ein Knoten zu oft besucht, wird abgebrochen und es werden nur Testfälle für die Pfade bis zu diesem Maximalwert erstellt (vgl. Unterabschnitt 7.1.7). Die Entscheidung liegt infolgedessen bei der testfallerstellenden Person und bietet mehrere Vorteile:

- Das algorithmisch nicht entscheidbare Problem kann von Anwender*innen kontext- und situationsbedingt entschieden werden.
- Für die Anwender*innen versteckte Schleifen in Hierarchien werden ebenfalls berücksichtigt.
- Pfade aufgrund großer Wertebereiche von Signalen können ebenfalls begrenzt werden (zum Beispiel `out1` ist eine Zahl zwischen 0 und 255).

Das Problem mit der Anzahl der Schleifeniterationen kann sich nicht nur auf die nachfolgenden Rechenschritte auswirken, sondern auch auf die Effizienz der erzeugten Artefakte [LTBT17]. Aus diesem Grund wird bei dem vorgestellten Testfallgenerator die Entscheidung eines begrenzenden Maximalwertes den erfahrenen Domänenexpert*innen überlassen. Mit Hilfe von statischen Methoden für die Begrenzung der Anzahl der Schleifeniterationen wie in [HSRW98, TJ07, LTBT17] könnte der Testfallgenerator weiter verbessert werden, allerdings muss dies für den Anwendungsfall dieser Arbeit noch evaluiert werden. Der Autor dieser Arbeit vermutet aufgrund von Erfahrungen aus der Praxis und [TJ07], dass eine kleine geeignete Anzahl an Schleifeniterationen wahrscheinlich einen Großteil der Fehler finden wird. Demzufolge liefert die Begrenzung der durchlaufenen Schleifen auf einen von Domänenexpert*innen festgelegten Wert von mindestens einer Iteration ein praktikables Kriterium für die kontext- und situationsabhängigen Testfälle. Zusätzlich kann die Qualität der Testfälle mit Hilfe von definierten Vor- und Nachbedingungen gesteigert werden.

Die Integration von Vor- und Nachbedingungen ermöglicht den Tester*innen eine Anpassung der Testfälle auf die vorgesehenen Testumgebungen. Der Grund, warum die Vor- und Nachbedingungen nicht im Modell vorgehalten werden, ist, dass die Bedingungen von den volatilen Gegebenheiten der jeweiligen Testumgebung abhängen. Das Nachreichen von Wertebereichen für Signale beziehungsweise Informationsflüsse ergänzt noch unvollständige Wertebereiche für die Testfallerstellung. Dies ist vor allem für Modellsignale relevant, die nicht im Modell oder in Schlüsselwörtern definiert sind. Zusätzlich gibt es den Tester*innen die Möglichkeit, auch jenseits der definierten Wertebereiche des Modells zu testen und somit gezielt Wertebereiche zu ergänzen und zu manipulieren. Diese Wertebereiche werden in einem *Umweltmodell* beschrieben. Tabelle 7.14 stellt ein solches Umweltmodell exemplarisch dar.

Neben der Integration des Umweltmodells und der Vor- und Nachbedingungen der Testfälle ist es möglich die Testabdeckungsmethoden zu wählen und spezifische Prioritäten für die Modellelemente zu vergeben. Diese Prioritäten sind für individuelle Testabdeckungsmethoden der Pfad-Kalkulation relevant (vgl. Theorie und Praxis 7.2).

Tabelle 7.14: Umgebungsmodell mit zwei Umgebungssignalen und deren Wertebereichen

Umgebungssignal	**Wertebereich**
T	MAX; MED; MIN
in4	0%; 5%; 15%; 50%; 90%; 98%; 100%; 105%

7.1.7 Pfad-Kalkulation

Anschließend werden die Pfade gemäß der Konfiguration der Benutzer*in für die Testabdeckungsmethoden berechnet. Neben der standardmäßigen Pfadüberdeckung (engl. „path coverage“) [Lig09] ist es ebenfalls möglich, andere Abdeckungen wie beispielsweise „Modified Condition / Decision Coverage“ (MC/DC) [GC09] oder individuell gestaltete Testabdeckungskriterien der Benutzer*innen zu verwenden.

Theorie und Praxis 7.2 (Testabdeckungsmethoden)**.** *Gängige Testabdeckungsmethoden aus dem Software Engineering haben sich in der Praxis mit SMArDT und dem Testfallgenerator als zu schwach (Pfadüberdeckung), als nicht praktikabel genug (Multiple Condition Coverage (MCC) [GC09]) oder nicht geeignet für die domänenspezifische Anforderungsabdeckung (MC/DC) herausgestellt. Ein sehr einschränkender Umstand für die Qualität der Testfälle und ihrer Abdeckung ist die Verfügbarkeit und Beschaffenheit der Testumgebungen [JTH21]. Da Hardware und Software von Prüfständen und Fahrzeugen zu den Freigaben der Produktversionen hochgerüstet werden müssen, falsche Fehlermeldungen ausgeschlossen werden müssen und größtenteils an die Realzeit gebunden sind, muss die begrenzte Testzeit effektiv und effizient genutzt werden (vgl. Abschnitt 3.1). Außerdem verfügen nicht alle Prüfstände und Fahrzeuge über alle Ressourcen, wie Sonderausstattungen, Bauteile oder Software. Die Testumgebungen und die Testzeit für die Testfälle sind folglich nur mit hohen Aufwand skalierbar und müssen bei den Testabdeckungsmethoden der Testfallerstellung berücksichtigt werden.*

*Aufgrund der verschiedenen Anforderungen der unterschiedlichen Domänen wurde die individuell gestaltete Testabdeckungsstrategie für einzelne Domänen und Funktionalitäten von den Tester*innen gut angenommen. Diese individuellen Testabdeckungsmethoden ähneln MC/DC und ermöglichen einzelne Gewichtungen von Signalen, Signalausschlüsse und deren Kombination vorzunehmen. Diese individuellen Testabdeckungsmethoden werden in dieser Arbeit nicht weiter vertieft*[1]*.*

Anschließend werden die Pfade anhand eines gewählten Testabdeckungskriteriums berechnet. Im weiteren Verlauf dieses Abschnitts wird mit der standardmäßig eingestellten Pfadüberdeckung fortgefahren. Abbildung 7.15 stellt die Strukturen für die Pfadüberdeckung dar.

Grundsätzlich nutzt der `Testfallgenerator` die Informationen aus dem `Aktivitätsgraph`, um `Testfälle` zu erstellen. Für die Pfadüberdeckung wird der `Testfallgenerator` um die Klasse `PfadabdeckungTestfallgenerator` erweitert, die benötigte

[1] Aufgrund eines laufenden Patentverfahrens wird diese Option in dieser Arbeit nicht näher detailliert.

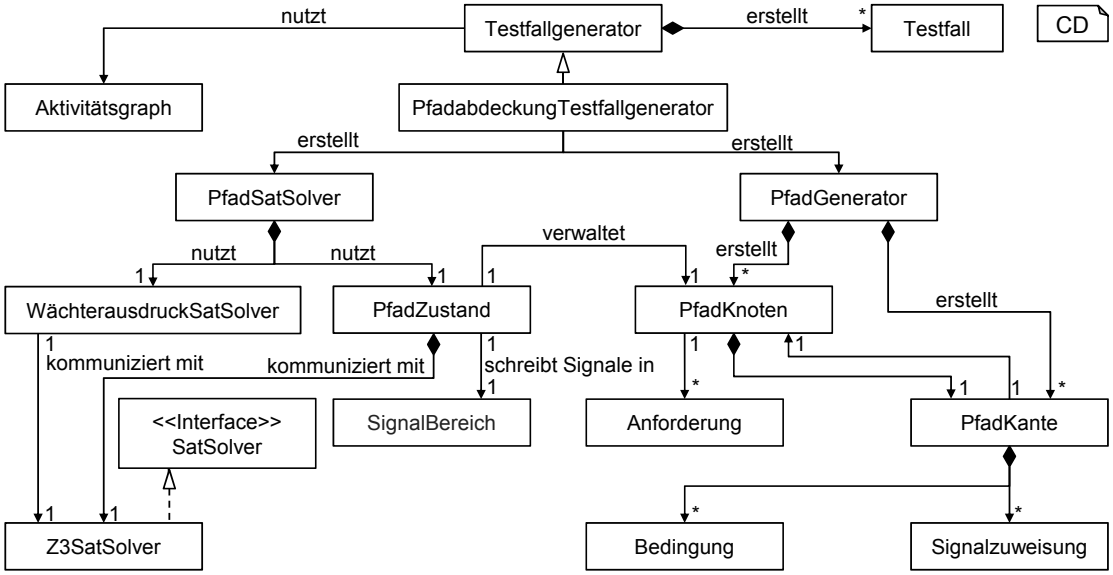


Abbildung 7.15: CD der Pfadüberdeckung des Testfallgenerators (erweitert aus [Mil17])

Hilfsmittel zur Verfügung stellt. Eines der Hilfsmittel ist der `PfadGenerator`, für die Pfaderzeugung, auf der rechten Seite des CDs in Abbildung 7.15. Das andere Hilfsmittel der `PfadSatSolver`, für die Bewertung der Erfüllbarkeit der zuvor erzeugten Pfade, auf der linken Seite des CDs.

Bei einer Pfadüberdeckung werden zunächst alle möglichen Pfade nur auf der Grundlage des `Aktivitätsgraphen` bestimmt. Zu diesem Zweck erstellt der `Pfadgenerator` eine Liste der `PfadKnoten`, die eine Knotensequenz mit eindeutigem Anfang und Ende repräsentiert. Im Gegensatz zu den Knoten in ADs und dem resultierenden Aktivitätsgraph sind `PfadKnoten` auf maximal eine ausgehende Kante, (`PfadKante`) beschränkt. Als Konsequenz wird jede Knotensequenz separat betrachtet, die mit einem `PfadKnoten` endet, der keine ausgehende `PfadKante` hat. Die vom `PfadKnoten` ausgehende `PfadKante` ist wiederum auf einen `PfadKnoten` gerichtet. `PfadKanten` besitzen die Eigenschaft, dass immer eine `Bedingung` und eine `Signalzuweisung` vorhanden sind, diese können allerdings leer sein. Zum Beispiel ist die Bedingung der vom Startknoten ausgehenden Kante eines ADs immer leer, beziehungsweise immer wahr.

Nachdem alle Permutationen von Listen der Knotensequenzen erzeugt wurden, werden die erfassten Pfade mit dem `PfadSatSolver` auf ihre Erfüllbarkeit überprüft. Zu diesem Zweck nutzt `PfadSatSolver` die Klassen `WächterausdruckSatSolver` und `PfadZustand`. Für die Berechnung der Erfüllbarkeit der einzelnen Wächterausdrücke konvertiert der `WächterausdruckSatSolver` die atomaren Ausdrücke in das für den `Z3SatSolver` geeignete Variablenformat. Anschließend können die Variablen der Wächterausdrücke gegen andere Variablen, gegen einen konstanten Wert, die Werte gegeneinander, aber auch mit Ausdrücken (engl. „Expressions“) verglichen werden. Ist der Vergleich beziehungsweise die Berechnung aller geladenen Wächterausdrücke des Modells möglich, wird das vom

`Z3SatSolver` berechnete Modell ausgegeben.

Der `PfadZustand` kapselt die `PfadKnoten` und verwaltet die Zustände der Signale entlang des Pfades. Für die Verwaltung der Signale verwendet `PfadZustand` den `Signal-Bereich`. Der `SignalBereich` liefert die aktuelle Version eines Signals und speichert die Informationen über alle verwendeten, neu spezifizierten Signale und deren verwendete Reihenfolge. Im Fall von spezifizierten Signalen wird geprüft, ob dieses Signal, schon als spezifiziertes Signal vorhanden ist. Falls es noch nicht vorhanden ist, wird eine neue Version dieses Signals mit den gleichen Eigenschaften, aber einem anderen Namen erstellt.

7.1.8 Pfad-Validierung

Für die Validierung der erfassten Pfade wird die Java-API des *Z3 SAT Solver* [dMB08] von Microsoft Research [Mic20] genutzt. Die akzeptierten Formeln müssen in diesem Tool nicht in der konjunktiven Normalform vorliegen. Infolgedessen können die Wächterbedingungen im Aktivitätsgraph direkt in bearbeitete Ausdrücke von Z3 konvertiert werden.

Die Klasse `Z3SatSolver` in Abbildung 7.15 dient als SatSolver-Schnittstelle für die Z3-Implementierung. Auf diese Weise kann die Information zurückgeben werden, ob ein gesamter Pfad zufriedenstellend ist. Zusätzlich gewährleistet die Implementierung über die Schnittstelle eine Flexibilität gegenüber anderen SAT-Solver-Implementierung.

Für den `SatSolver` wird zunächst der, dem Signal entsprechende Z3-Variablentyp und die zugehörige Variable des Typs mit vorgegebenen Namen erstellt. Auch mehrere Variablen desselben Typs können erstellt werden. Nach der Initialisierung akzeptiert der Z3 SAT Solver die Ausdrücke. Ausdrücke im Z3 SAT Solver können erzeugte Variablen, boolesche Ausdrücke oder definierte konstante Werte sein. Alle Ausdrücke sind vergleichbar. Infolgedessen können zwei Variablen miteinander, eine Variable gegen einen konstanten Wert, zwei konstante Werte und beide mit einem Ausdruck verglichen werden. Anhand des Vergleichs der Ausdrücke kann der Z3 SAT Solver entscheiden, ob es eine Interpretation der in diesen Ausdrücken verwendeten Variablen gibt, sodass die durch ihre Verknüpfung erzeugte Formel zufriedenstellend ist. Ist die Formel erfüllt, liefert der Z3 SAT Solver das berechnete Modell, das für jede Variable einen Eintrag enthält. Dieser Eintrag ist die Interpretation des Z3 SAT Solvers, die eine Konstante vom gleichen Typ wie die Variable ist. Ist die Formel nicht erfüllt, kann kein Modell aus dem Solver abgerufen werden, weil keine geeigneten Werte gefunden wurden, um die Eingabeformel zu erfüllen. Die mit dem Z3 SAT Solver ermittelten validen Modellartefakte werden anschließend in einer Liste dem Testfallersteller*innen des Testfallgenerators zur Verfügung gestellt.

7.1.9 Testfallerstellung

Die tatsächliche Erstellung der Testfalldateien im gewünschten Ausgabeformat wird über eine abstrakte Klasse, den Testfallersteller des Testfallgenerators realisiert. Der Testfallgenerator ermittelt alle Pfade im Diagramm und schreibt alle Testfälle, für alle gültigen Pfade, in Ausgabedateien. Tabelle 7.16 stellt zwei tabellarische Testfälle dar, die aus dem Diagramm in Abbildung 7.7 generiert wurden.

Tabelle 7.16: Zwei exemplarische Testfälle nach der Testfallgenerierung auf Basis des Diagrammablaufs aus Abbildung 7.7.

Aktion	**Name**	**Wert**	**Erwartung**
Steuern:	in2	an	
Status:	out2		aktiv
Steuern:	in4	90%	
Status:	out3		an
Steuern:	in1	wahr	
Status:	out1		wahr
Steuern:	in3	wahr	
Status:	out1		wahr

Aktion	**Name**	**Wert**	**Erwartung**
Steuern:	in2	aus	
Status:	out2		aktiv
Steuern:	in4	90%	
Status:	out3		aus
Steuern:	in1	falsch	
Status:	out1		falsch
Steuern:	in3	falsch	
Status:	out1		falsch

In Tabelle 7.16 ist der Unterschied zwischen Signalen, die mit einem Schlüsselwort verknüpft sind und Signalen ohne eine solche Verknüpfung vernachlässigt. Ist ein Signal mit einem Schlüsselwort verknüpft, verwendet der modifizierte Testfallgenerator die Informationen über das referenzierte Schlüsselwort, um den entsprechenden Testschritt zu konstruieren. Im Fall ohne Verknüpfung wird ein dem Ausgabeformat angepasster Testschritt mit dem Signalnamen und allen benötigten Signalwerten, Parametern und Elementen erstellt. Je nach Testumgebungsframework sind diese Testschritte automatisiert ausführbar oder müssen anschließend manuell interpretiert werden und die entsprechenden ausführbaren Testschritte konstruiert werden.

Die in Tabelle 7.16 verwendeten Werte und Signale der Testfälle wurden größtenteils mit der Wertebereichvervollständigung aus Unterabschnitt 7.1.4 und der verknüpften Schlüsselwortbibliothek aus Unterabschnitt 7.1.5 generiert. Im Gegensatz dazu wird der Wert 90% von Signal `in4` durch ein Umweltmodell von der Benutzer*in vorgegeben (Weitere Details zur Verwendung des Umweltmodells in Abschnitt 7.3). Die in Tabelle 7.16 dargestellten Testschritte können in beliebigen Ausgabeformaten ausgegeben werden. Die gewünschten Ausgabeformate umfassen beispielsweise .xml-, .xlsx- und .java-Dateien, und können über eine Implementierung des spezifischen Generators erweitert werden.

Im Rahmen von SIL-Frameworks, wie JUnit [JUn20] können Variablen der generierten Testfälle automatisch zugeordnet und automatisiert ausgeführt werden. Für HIL- und VIL-Testumgebungen wie [Tra20, dSP20], sind für eine automatisierte Ausführung der erstellten Testfälle häufig weitere Schritte der Zuordnung des Modellsignals zu dem Testumgebungssignal oder eine Verknüpfung von Modellsignal zu Schlüsselwort nötig. Zusätzlich besteht die Möglichkeit, die für die Testumgebung benötigten Eingabewerte der Vorbedingung und Nachbedingung dem Testfall hinzuzufügen. Die Testfallartefakte und deren Zusammenhänge werden in Abschnitt 7.3 näher beschrieben.

In der Funktionsentwicklung mit SMArDT hat sich für zustandsorientierte Funktionen herausgestellt, dass ADs nicht uneingeschränkt geeignet sind. Der Grund ist, dass eine explizite Modellierung der Zustände mittels Elementen des ADs zu exponentieller Vergrößerung und Unübersichtlichkeit des Modells führt. Aus diesem Grund wurde

der Testfallgenerator ebenfalls um weitere SysML-Verhaltensdiagramme erweitert. Eine Erweiterung um SC wird im nächsten Abschnitt beschrieben.

7.2 Testfallerstellung aus SysML-Zustandsdiagrammen

Neben SysML-ADs erlaubt SMArDT weitere Verhaltensdiagramme wie SC, SD, UCD und Kombinationen der Diagramme. Für zustandsorientierte Funktionen werden SCs verwendet. Abbildung 7.17 zeigt ein Beispiel eines SCs.

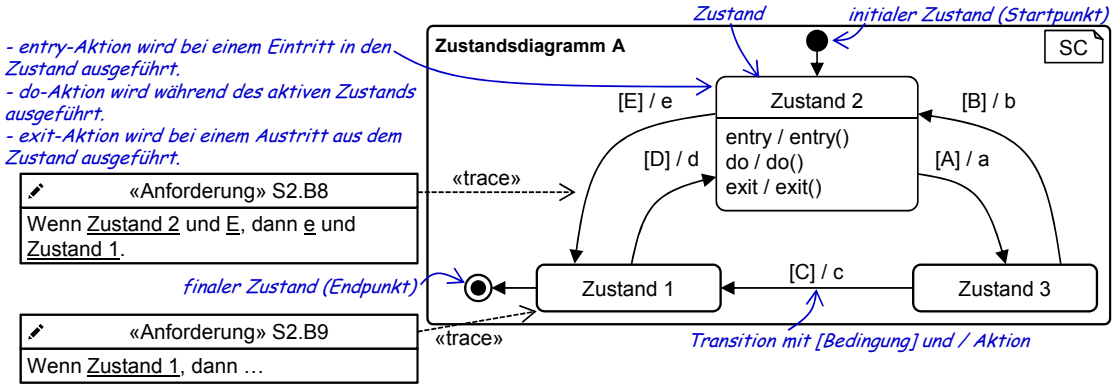


Abbildung 7.17: Beispiel eines SCs mit Anforderungen

In dem SC mit dem Namen `Zustandsdiagramm A` werden drei interne Zustände `Zustand 1`, `Zustand 2`, und `Zustand 3` dargestellt. Diese Zustände können weitere Zustände, komplette SCs oder Eingangs- (`entry`), Ausführungs- (`do`) und Ausgangsaktionen (`exit`) beinhalten. Diese Eingangs-, Ausführungs- und Ausgangsaktionen können Signalzuweisungen, wie in Abschnitt 6.2, oder jeweils ein (hierarchisches) AD enthalten. Die Eingangsaktion wird bei jedem Eintritt des SCs in den Zustand einmalig ausgeführt. Analog wird beim Austritt des Zustands die Ausgangsaktion einmalig ausgeführt. Die Ausführungsaktion in `do` wird so lange (immer) ausgeführt, wie der Zustand aktiv ist, also bis der Zustand (über die Ausgangsaktion) verlassen wird. Weitere Zustände des SCs sind der Startpunkt und der Endpunkt. In diesen „Pseudo“zuständen kann nicht dauerhaft verweilt werden und sie erlauben keine internen Aktionen. Der Endpunkt markiert das Ende des zugehörigen SCs und kann maximal einmal pro SC vorkommen. Der Startpunkt muss exakt einmal pro Diagramm vorhanden sein und markiert mittels einer Transition den Zustand, in dem das SC startet. Transitionen (Zustandsübergänge) von SCs können, ähnlich wie bei ADs (vgl. Abschnitt 6.2), die in Klammern stehenden Bedingungen und weitere Zuweisungen enthalten. Allerdings können Transitionen im SC den Anforderungen und Zuständen zugewiesen werden, was in ADs nicht möglich ist und von AD4S nicht unterstützt wird. Ein Beispiel in Abbildung 7.17 ist die Anforderung ≪`Anforderung`≫ `S2.B8` die der Transition von `Zustand 2` an `Zustand 1` über eine `trace`-Beziehung zugeordnet ist. Die Zuweisung von Anforderungen an Zustände im SC ist ähnlich wie die Zuweisung von Anforderungen an Aktionen im AD möglich.

Der Testfallgenerator ist für AD4S-konforme ADs ausgelegt. Für ein (AD4S-konformes) SC bedeutet dies, dass die SCs in ADs transformiert werden müssen, bevor sie in Abschnitt 6.2 übersetzt werden können. Im Allgemeinen ist es möglich, wie in [Rum16] diskutiert, SCs in eine reduzierte Form zu transformieren, die nur die Grundelemente enthält. Diese Transformation kann ohne Informationsverlust erfolgen. Daher ist es möglich, dass über eine Transformation ebenfalls SCs und gegebenenfalls deren Kombination mit ADs von dem AD4S-Framework unterstützt werden können. Für diese Transformationen müssen die ausgehenden Transitionen eines Zustands deterministisch sein. Deterministisch bedeutet in diesem Fall, dass nicht mehrere Transitionen eines Zustands gleichzeitig aktiviert werden können, sondern immer nur eine Transition und dass maximal ein Zustand mit der Transition erreicht werden kann. Es muss beispielsweise sichergestellt sein, dass in Abbildung 7.17 die beiden Bedingungen von `Zustand 3 B` und `C` komplementär zueinander sind.

Transformation

Das Konzept der Transformation besteht darin, die Informationen über den aktuellen internen Zustand des SCs im AD zu speichern. Daher enthält das AD bestimmte Ports, um den aktuellen Zustand des SCs zu bestimmen. Die wesentlichen Schritte der Transformation sind:

1.) Den gegenwärtigen internen Zustand des SCs ermitteln.

2.) Die Ausführungsaktion (`do`) dieses Zustands erfassen.

3.) Die Transition auf Basis des Ergebnisses der Ausführungsaktion bestimmen.

4.) Die Ausgangsaktion (`exit`) des gegenwärtigen internen Zustands identifizieren.

6.) Die zugehörige Transition bestimmen.

7.) Die Eingangsaktion des Zielzustandes der Transition erfassen.

Zunächst werden alle gegenwärtigen internen Zustände des SCs in einer Liste gesammelt. Am Beispiel von Abbildung 7.17 umfasst die Liste `Zustand 1`, `Zustand 2` und `Zustand 3`. Interne Zustände die nicht erreicht werden können, das heißt, die nicht direkt oder indirekt über Transition mit dem Startpunkt verbunden sind, werden nicht mit in die Liste aufgenommen. Diese gegenwärtigen internen Zustände der Liste werden im AD als temporäre Variablen definiert. Um den aktuellen internen Zustand des SCs mit den temporären Variablen zu bestimmen, wird im transformierten AD eine Alternative direkt nach dem Startknoten verwendet. Abbildung 7.18 stellt das aus Abbildung 7.17 transformierte AD mit der Alternative `DN1` direkt nach Startknoten dem dar.

Die aus `DN1` ausgehenden Kanten in Abbildung 7.18 ist jeweils einem Wächterausdruck zugewiesen. Folglich dient die Alternative `DN1` über die aktuelle Zustandsvariable (`aktZustand`) den gegenwärtigen internen Zustand des SCs zu bestimmen. Der Zustandswertebereich von `aktZustand` entspricht dementsprechend der Liste der gegenwärtigen

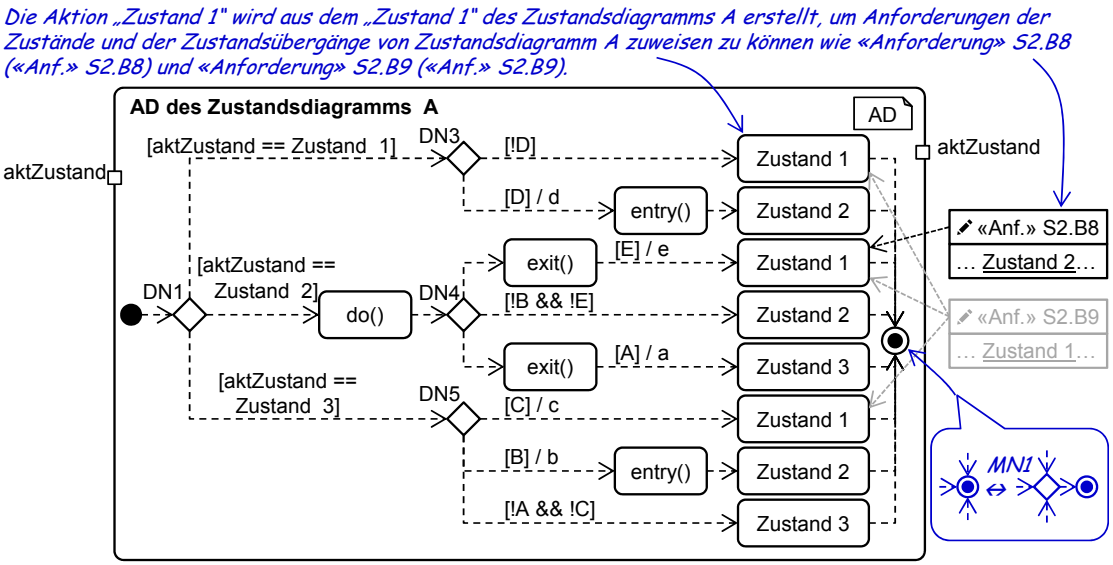


Abbildung 7.18: Vereinfachte Darstellung des transformierten ADs aus dem SC von Abbildung 7.17

internen Zustände des SCs. Zusätzlich werden die internen Zustände in einfache Aktionen umgewandelt. Zum Beispiel wird aus dem SC-Zustand `Zustand 1` aus Abbildung 7.17 die AD-Aktion `Zustand 1` aus Abbildung 7.18 erstellt. Bei der Transformation von Zustandsdiagrammen in AD4S sind die Namen aller erstellten „Zustands-"Aktion eindeutig und über die Transition mit in den Namen der Zustandsaktivität codiert. Dieser Umstand wird in Abbildung 7.18 vereinfacht dargestellt. Zu beachten ist ebenfalls, dass für jede Transition des SCs, die einen Zustand als Ziel hat, eine weitere einfache Aktion für diesen Zielzustand erstellt wird. Das bedeutet, dass für `Zustand 1` aus Abbildung 7.17 drei Aktionen in Abbildung 7.18 erstellt werden, aufgrund der Transitionen `[E]/e`, `[C]/c` und `[!D]` (`Zustand 1` wird nicht verlassen). Der Grund für die mehrfache Erstellung dieser „Zustands-"Aktionen ist die Anforderungsabdeckung. Anforderungen, die den Zuständen im SC zugeordnet sind, wie ≪`Anforderung`≫ `S2.B9` an `Zustand 1` werden jeder zugehörigen Aktion mit gleichen Namen im AD zugeordnet (vgl. Abbildung 7.17). Anforderungen, die den Transitionen im SC zugeordnet sind, wie ≪`Anforderung`≫ `S2.B8` an die Kante mit der Bedingung `E` in Abbildung 7.17, werden der erstellten Aktion aus dem Zielzustand mit der entsprechenden Kante zugeordnet. Infolgedessen hat jeder Zustandsübergang des SCs genau einen äquivalenten Pfad im AD mit den gleichen Anforderungen. Das Anhängen der Anforderungen der Transition an die entsprechende Aktion verursacht folglich keine falsche Anforderungsabdeckung. Die Namen der erstellten Aktionen des ADs und die der ursprünglichen (Ziel-)Zustände des SCs sind gleich. Diese Aktionen werden danach über eine Zusammenführung mit dem Endknoten der Aktivität verbunden (vgl. `DN2` in Abbildung 7.18b).

Außerdem werden die Aktionen der Zustände, wie Eingangs-, Ausführungs- und Aus-

gangsaktionen den zugehörigen Aktionen zugeordnet. Zunächst werden die Ausführungsaktionen (`do()`) nach der Alternative für aktuelle Zustandsvariablen (`DN1`) der zugehörigen Kante hinzugefügt. Anschließend werden die ausgehenden Übergänge jedes Zustands unter Verwendung von Entscheidungsknoten modelliert. Wenn das SC in einen neuen Zustand wechselt, werden die Ausgangsaktion (`exit()`) des aktuellen Zustands, die Zuordnung der jeweiligen Transition und schließlich die Eingangsaktion (`entry()`) des neuen Zustands hinzugefügt.

Testfallerstellung

Die Testfallerstellung auf der Basis des transformierten ADs ist analog zu Abschnitt 7.1. Allerdings muss beachtet werden, dass für SCs jede Transition durch genau einen Testfall abgedeckt wird. Folglich werden Sequenzen von Zustandsübergängen, wie zum Beispiel `Zustand 1` → `Zustand 2` → `Zustand 3` → `Zustand1` nicht berücksichtigt. Für diesen Fall müssten die Testfälle nacheinander durchgeführt werden. Außerdem werden Ausführungsaktionen mit `do()` nur einmal pro Durchlauf einer Aktivität ausgeführt. Für eine Wiederholung der Aktion mit `do()` muss die Aktivität mehrmals mit den spezifischen Parametern durchlaufen werden. Im Beispiel von Abbildung 7.18 müssten `aktZustand = Zustand 2`, `!B` und `!E` erfüllt sein, damit `do()` vor `DN4` bei wiederholen der Aktivität (`AD des Zustandsdiagramms A`) ebenfalls wiederholt ausgeführt wird.

Mit dem vorgestellten Ansatz können abhängig vom aktuellen Zustand des SCs verschiedene Wege durch das transformierte AD modelliert werden. Das heißt, die Pfade durch das AD stellen die Übergänge jedes Zustands des SC dar. Darüber hinaus sind alle Informationen und möglichen Werte der Informationsflüsse, die im ursprünglichen SC verwendet werden, auch in dem jeweiligen AD verfügbar. Folglich enthalten die ADs die gleiche Information wie die SCs, inklusive der gegenwärtigen internen Zustände und Zustandsübergange. Das ist vor allem für die Konsistenz des Modells wichtig, wenn im Modell SCs und ADs kombiniert werden. Die aus dem Modell erzeugten Testfälle können individuell angepasst und mit zusätzlichen Informationen angereichert werden. Die Struktur der erzeugten Testfälle wird im nächsten Abschnitt erläutert.

7.3 Erzeugte Artefakte des Testfallgenerators

In diesem Abschnitt werden die erzeugten Artefakte des Testfallgenerators detailliert beschrieben. In den vorangegangen Abschnitten 7.1 und 7.2 wurden die AD4S konformen Eingangselemente des Testfallgenerators betrachtet. Diese Elemente nutzt der Testfallgenerator, um aus den Informationen Testfälle zu generieren. Zusätzlich kann der Testfallgenerator weitere externe Elemente verwenden, die ermöglichen, dass die Testfälle vollständig ausführbar sind. Abbildung 7.19 veranschaulicht in einem Ausschnitt die verwendeten und erstellten Elemente des Testfallgenerators auf einer SMArDT-Ebene B.

Wie in den Abschnitten 7.1 und 7.2 beschrieben, verwendet der `Testfallgenerator` die Informationen aus den `AD4S`-konformen `Diagrammen` des `Modells`. Die im Diagramm modellierten `Knoten` und `Kanten` geben die Reihenfolge der Testschritte im `Testfall`

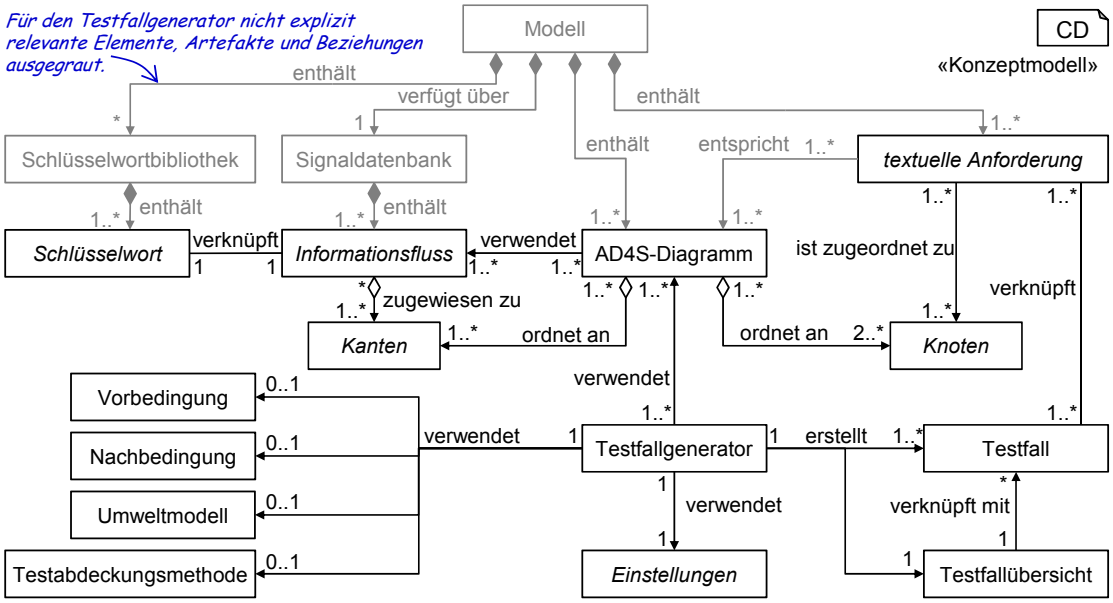


Abbildung 7.19: Benötigte und erzeugte Elemente des Testfallgenerators auf SMArDT-Ebene *B*

vor. Diese Testschritte entsprechen entweder den in Unterabschnitt 7.1.5 angesprochenen `Schlüsselwörtern` oder direkten Signalzuweisungen. Die Schlüsselwörter und die Signale kann der Testfallgenerator über die (verknüpften) `Informationsflüsse` verwenden, da die Informationsflüsse den Kanten zugewiesen werden. Ist einer Kante kein Informationsfluss zugeschrieben, wird nur die Information über direkt darauffolgenden Knoten (Reihenfolge) verwendet. Den Knoten können `textuelle Anforderungen` zugeordnet sein. Diese textuellen Anforderungen sind bidirektional mit den Testfällen verknüpft und ermöglichen eine transparente Anforderungsabdeckung. Zusätzlich zu den Testfällen wird bei jeder Testfallerstellung eine `Testfallübersicht` erzeugt. Die Testfallübersicht listet alle kompletten Testfälle inklusive ihrer `Vor-` und `Nachbedingungen` und den jeweiligen `Anforderungen` tabellarisch auf. Tabelle 7.20 gibt einen Auszug aus einer Testfallübersichtstabelle.

Tabelle 7.20: Auszug einer Testfallübersicht von generierten Testfällen

Name	**Vorbedingung**	**Aktion**		**Nachbedingung**	**Anforderung**
TF_01	T=MAX...	in2 = aus	ou2 == aktiv ...	T=MED ...	≪Anf.≫ R5D4 = ...
TF_02	T=MAX...	in2 = aus	ou2 == nicht_aktiv ...	T=MED ...	≪Anf.≫ R5D4 = ...
TF_03	T=MAX...	in2 = an	ou2 == aktiv ...	T=MED ...	≪Anf.≫ C3P0 = ...
⋮					

Die Testfallübersichtstabelle unterstützt die Tester*innen dabei, eine Übersicht über die Testfälle zu gewinnen. Dies erleichtert die manuelle Prüfung der erstellten Testfälle

und gegebenenfalls die Einstellungen des Testfallgenerators anzupassen. Zusätzlich kann eine manuelle Prüfung der Anforderungsüberdeckung erfolgen. Der Aufwand für eine Priorisierung und/oder eine manuelle Zuteilung der Testfälle auf die Testumgebungen wird ebenfalls verringert, falls dies nicht automatisiert wie in [EJK+19, BLLS14] erfolgt.

Die in der Tabelle dargestellten Vor- und Nachbedingungen werden von den Anwender*innen des Testfallgenerators hinzugefügt. Die Vor- und Nachbedingungen werden in dem Format integriert, das dem gewünschten Ausgabeformat entspricht. Zusätzlich zu den Vor- und Nachbedingungen ist es ebenfalls möglich, das `Umweltmodell` dem Testfallgenerator zur Verfügung zu stellen. Das Umweltmodell ist vor allem für Modellsignale relevant, die nicht mit Schlüsselwörtern verknüpft sind. Die Elemente des Umweltmodells können auch in einer individuellen `Testabdeckungsmethode` verwendet werden. In dieser Option ist es möglich, Bedingungen und Signalzuweisungen des Modells auszuwählen, die bei der Testfallgenerierung zu behandeln sind[2]. Mit Hilfe der Testabdeckungsmethode lassen sich individuelle Abdeckungskriterien verwirklichen, die nicht den `Einstellungen` des Testfallgenerators vorhanden sind. Neben Einstellung wie einer Diagrammauswahl, dem Ausgabeformat und einem Entwurf-Modus, kann die Testfallausführungsart eingestellt werden. Der Entwurf-Modus gibt die Zwischenergebnisse aus. Tabelle 7.21 stellt eine Variante für Testfälle dar, die für statische Signale geeignet ist.

Tabelle 7.21: Variante eines Testfalls von Tabelle 7.16 für statische Signale in einer kritischen Echtzeitumgebung.

Typ	**Aktion**	**Name**	**Wert**	**Erwartung**
Vorbedingung	Herstellen:	in2		an
	Herstellen:	in4		90%
	Herstellen:	in1		wahr
	Herstellen:	in3		wahr
Aktion	Status:	out2		aktiv
	Status:	out3		an
	Status:	out1		wahr
	Status:	out1		wahr

Die in Tabelle 7.21 dargestellte Variante stellt alle benötigten Signaleingangsbelegungen in der Vorbedingung her. Im Gegensatz zum Modellpfad (vgl. Tabelle 7.16), werden nach der Vorbedingung alle Signale anschließend in der Testfallaktion geprüft. Diese Einstellung ist insbesondere in frühen Prototypenphasen der Entwicklung oder nach einer Hochrüstung beziehungsweise einem Update der Testumgebung hilfreich. Hierüber kann geprüft werden, ob der Status korrekt vom System ausgegeben wird. Außerdem ist es möglich, erst alle Signale in der Testfallaktion per Herstellen zu setzen, um anschließend festzustellen, ob die erwarteten Systemantworten eintreten. Die letztgenannte Option ist bei kritischen Echtzeitumgebungen hilfreich, in der beispielsweise die Testfallumgebung die statischen Signale nicht schnell genug ansteuern kann.

[2]Aufgrund eines laufenden Patentverfahrens wird diese Option in dieser Arbeit nicht näher detailliert.

Kapitel 8

Evaluation und gewonnene Erkenntnisse

In diesem Kapitel werden die vom Generator erstellten Testfälle sowie manuell erstellte Testfälle aus der Praxis untersucht und miteinander verglichen. Hierbei wird zudem die entwickelte Sprache in einer Umfrage evaluiert und gewonnene Erkenntnisse erläutert. Neben dem Vergleich von manuell und automatisiert erstellten Testfällen wird der aktuelle Nutzen und das erwartete Potenzial kurz erläutert. Während der zweijährigen Implementierungs- und Einführungsphase des Testfallgenerators stellte sich schnell heraus, dass nicht der Testfallgenerator und die generierten Testfälle sondern wohlgeformte und wohldefinierte Spezifikationsmodelle die größte Herausforderung sind. Die Modelle sind hier ausschlaggebend für die Qualität der Testfälle. Aufgrund dessen wird in diesem Abschnitt ein starker Fokus auf die Richtlinien für die Modellierungssprache gelegt.

Zu diesem Zweck wird in Abschnitt 8.1 auf die generierten Testfälle eingegangen. In Abschnitt 8.2 wird eine Studie über die eingeführten Richtlinien für AD4S vorgestellt. Danach werden in Abschnitt 8.3 die Schlussfolgerung geschildert. Zusätzlich werden wichtige Erkenntnisse gesondert unter **Zusammenfassende Erkenntnis** zusammengefasst.

8.1 Ungebundenes vs. modellbasiertes Testen

In diesem Abschnitt werden die mit dem Testfallgenerator generierten Testfälle mit manuellen Testfällen verglichen. Die manuellen Testfälle sind ebenfalls modellbasiert, allerdings stützen sich diese allein/vornehmlich auf natürlichsprachliche textuelle Anforderungen. Während der Analyse der Testfälle konnten Einflussfaktoren ermittelt werden, die bei einer direkten Gegenüberstellung zu Fehlschlüssen führen können. Die in diesem Abschnitt vorgestellten Beobachtungen sind nicht rein objektiv, da die Einflussfaktoren verwoben sind und eine objektive Analyse erschweren. Einflussfaktoren sind:

- Testumgebung
- Komplexität der Funktion beziehungsweise Vernetzungsgrad der Funktion
- Funktionsgruppierungen
- Erfahrungen der Modellierer*innen in der Funktionsdomäne
- Erfahrung der Tester*innen
- Arbeitsweise der Tester*innen

Das Testobjekt in dieser Arbeit ist der elektrische Antrieb (vgl. Abschnitt 3.1). In der Funktionsdomäne des elektrischen Antriebs liegt der Fokus tradionell auf dem E/E-Anteil. Dieser Fokus wird sich zukünftig auf softwaregestützte Funktionalitäten verschieben. Der starke Anteil der Software und einer E/E Architektur erlaubt eine funktionale Betrachtungsweise, um Spezifikationsmodelle zu erstellen. Wie eine solche funktionale Betrachtungsweise in anderen Funktionsdomänen mit Fokus auf Maschinenbau etabliert werden kann und wie deren Modelle für die Testfallerstellung genutzt werden, wird in dieser Arbeit nicht behandelt. Der elektrische Antrieb wurde beispielsweise in drei Teilbereiche der Funktionsprinzipien gegliedert: *Moment_stellen*, *Laden* und *Energiemanagement*. Die Spezifikationsmodelle der Teilbereiche variieren stark. Die Modelle des Teilbereichs Moment_stellen erinnern stark an zeitkontinuierliche Regelungsmodelle[1]. Die erstellten Spezifikationsmodelle des Teilbereichs Laden eignen sich sehr gut für eine zeitdiskrete Darstellung. In diesem Teilbereich konnte der Testfallgenerator und andere softwaretechnische Verfahren besonders gut angewendet werden, da es sich um Funktionen mit einem stark diskreten Charakter handelt (vgl. Abbildung 2.4). Im Teilbereich Energiemanagement sind die einzelnen Spezifikationsmodelle größer und stellen eine größere intellektuelle Herausforderung dar. Hier ist der Vernetzungsgrad der Funktionen und Komponenten hoch. Aufgrund dieser Tatsache erwarteten die Teilnehmer*innen des Projekts hier einen besonders großen Vorteil durch den Einsatz des Testfallgenerators. Der hohe Vernetzungsgrad der Energiemanagementmodelle erschwert Gruppierungen für eine einfachere Darstellung. Die Gruppierungen in den „zeitkontinuierlichen“ Regelungsmodellen des Teilbereichs Moment_stellen wurden während des Projektverlaufs ebenfalls angepasst, reduzierten jedoch wenig Komplexität. Auch im Teilbereich Laden wurden Funktionen mehrmals umgruppiert. Die beständige Modellierung der Laden-Funktionen erlaubte es, Schnittstellen transparent darzustellen und Optimierungen der Funktionsschnittstellen vorzunehmen. Abbildung 8.1 gibt einen Einblick auf die Optimierung mittels transparenter Funktionsschnittstellen. Erkenntnis 8.1 fasst noch einmal zusammen und deckt sich mit Beobachtungen anderer Studien [BCL+21, LMT+18]:

Erkenntnis 8.1 (Reflexion mit Hilfe einer transparenten Darstellung)**.** *Eine transparente Modellierung erlaubt es früh in der Entwicklung festzustellen, ob die vorhergehenden Vorstellungen und Gedanken optimal sind.*

Aus der Spezifikation von Abbildung 8.1 ist zu erkennen, dass `Funktionsgruppe1` und `Funktionsgruppe2` verstärkt kommunizieren. Diese Anfangsvorstellung der Funktionsaufteilung kann mittels einer transparenten Darstellung geprüft werden und falls nötig optimiert werden. In Abbildung 8.1 hat dies zu Folge, dass die anfänglichen Überlegungen oder existierenden Schnittstellen bezüglich der Funktionsaufteilung nicht optimal sind und deshalb die Funktionsgruppen in die `Funktionsgruppen 1'`, `3` und `4` aufgeteilt wurden (vgl. Theorie und Praxis 4.1, 4.3 und 4.4). Insbesondere die in Unterabschnitt 4.2.2 angesprochenen (automatisiert) erstellten IBDs eignen sich für eine transparente und

[1] Dies ist eine subjektive Wahrnehmung. Ein objektiver Vergleich der Regelungsmodelle und Spezifikationsmodelle würde den Rahmen dieser Arbeit sprengen.

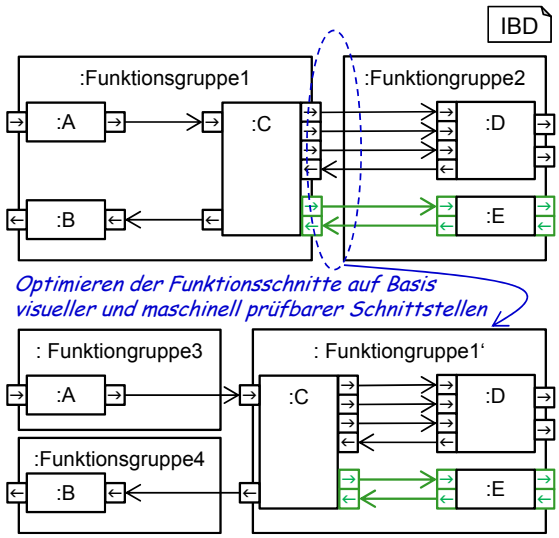


Abbildung 8.1: Mit Hilfe der transparenten Schnittstellen ist es früh möglich Funktionsschnittstellen zu optimieren.

visuelle Darstellung der Schnittstellen. Solche Gruppierungen haben auch einen Einfluss auf die Testfälle, da sie den Testfokus verändern.

Die Gruppierungen sind ebenfalls ein Indikator für die Erfahrung in der Funktionsdomäne. So werden etablierte Funktionen und Systeme weniger oft restrukturiert und angepasst, was Auswirkungen auf den Zeitpunkt der Testfallerstellung haben kann. Bei der BMW Group kann im Bereich des elektrischen Antriebs auf einen Erfahrungshorizont von mehr als einer Dekade zurückgegriffen werden. Dies ist nicht nur für die Entwicklung neuer Funktionen und der Spezifikationsmodelle relevant, sondern auch für die Tester*innen. Hier kann beobachtet werden, je größer der Erfahrungsschatz der testfallerstellenden Person, desto weniger werden Anforderungen, Spezifikationen und Modelle zu Rate gezogen. Es ist jedoch anzumerken, dass sich diese Tendenz mit umfangreicheren, aktuellen und zuverlässigeren Modellen und der zunehmenden Modellierungserfahrung der Tester*innen verschieben kann.

Eine weitere Beobachtung ist, erfahrene Tester*innen verknüpfen (erstellte) Testfälle auf Anforderungen, während weniger erfahrene Tester*innen aus Anforderungen Testfälle erstellen. Dieser Aspekt, zusammen mit der Funktionsdomäne, hatte einen besonderen Einfluss auf den Aufbau der Testfälle. Beispielsweise validierten und verifizierten besonders erfahrene Tester*innen (Erfahrung in Jahren) den Teilbereich von Moment_stellen. Diese meist erfahrungsbasierten Testfälle konnten nicht aussagekräftig mit den aus Modellen generierten Testfällen verglichen werden. Besonders, da viel Erfahrung in den Testfällen steckte, die aber noch nicht im Modell hinterlegt war.

Die Teilbereiche Laden und Energiemanagement wurden von weniger erfahrenen Tester*innen abgesichert. Die modellbasiert-erstellten Testfälle dieser Teilbereiche hatten eine hohe Ähnlichkeit zu den generierten Testfällen. Allerdings kann nicht ausgeschlos-

sen werden, dass dies ebenfalls durch die individuelle (strukturierte) Arbeitsweise der Tester*innen, der Modelle oder dem Teilbereich der Funktionsdomäne beeinflusst wird. Insbesondere die Funktionen des Teilbereichs Laden wurden mehrmals über den Projektzeitraum in Abstimmung mit den Tester*innen neu geschnitten[2].

Im weiteren Verlauf dieses Abschnitts werden Beobachtungen und Erfahrungen evaluiert. Zu diesem Zweck werden in Unterabschnitt 8.1.1 Beobachtungen und Erfahrungen über die Testbarkeit von Diagrammen geteilt. Anschließend werden in Unterabschnitt 8.1.2 die Methodik und die Testfälle der Testfallerstellung aus natürlicher Sprache, mit manuellen und generierten modellbasierten Testfällen verglichen.

8.1.1 Erfahrungen aus dem Feld von testbaren Diagrammen

Die manuelle und automatisierte Testbarkeit der Modelle konnte über den Zeitraum der initialen Projektphase verbessert werden. Anfangs lag der Entwicklungsfokus auf der Strukturierung und Systematisierung von textuellen Anforderungen. Später verlagerte sich der Fokus auf die Strukturierung und Systematisierung der grafischen Diagramme. Mit der Zeit wurden iterativ weitere Kriterien wie die Testbarkeit aus Abschnitt 5.1 herangezogen. Die in Abschnitt 4.4 angesprochenen Reviews der Tester*innen resultierten in domänen-übergreifend verständliche Modelle. Allerdings nahmen die Reviews zu Projektbeginn viel Zeit in Anspruch. Abbildung 8.2, Abbildung 8.3, Abbildung 8.4 und Abbildung 8.5 der SMArDT Ebene *B* zeigen vereinfachte Darstellungen von Funktionen aus den Teilbereichen Moment_stellen, Laden und Energiemanagement. Die Diagramme repräsentieren den Modellierungsstatus am Anfang und am Ende eines Zeitraums von 24.06.2017 bis 01.10.2020. Folglich sind die Diagramme mit und ohne die Modellierungsrichtlinien aus Abschnitt 6.3 abgebildet. Für diese Veröffentlichung wurden die Elemente anonymisiert, da diese Details nicht für die Aussage der Modellstruktur relevant sind. Zusätzlich wurden einzelne Aspekte aus Darstellungsgründen vereinfacht und gegebenenfalls auf den Stand 01.10.2020 der Modellierungsrichtlinien gebracht, ohne den Charakter der Diagramme zu verändern.

Sequenzialisierung am Beispiel von Moment_stellen

Der Einfluss der in Abschnitt 6.3 vorgestellten Richtlinien ist in der Entwicklung von `Fkt1M1` nach `Fkt1M2` deutlich erkennbar. Insbesondere die Richtlinie **RL3**, dass „jede Aktivität oder Aktion transitiv mit dem Startknoten verbunden sein muss“ hatte Auswirkungen auf die Modellierung der Diagramme. In `Fkt1M1` aus Abbildung 8.2 sind „Eingänge“ als Aktionen dargestellt. Diese Aktionen wurden mithilfe von (externen) Partitionen an andere Systeme per Funktionsgruppen zugeordnet. Die Zuordnung über Partitionen erleichtert visuell eine intellektuelle Zuordnung der Elemente, aber erschwert eine Unterscheidung der Reihenfolge und des Signalflusses. Externe Funktionen wie `Fkt_EX1` werden farblich markiert und keiner spezifischen Partition zugeordnet. Werden die Eingänge

[2] Die Änderungen beziehen sich unter anderem auf neue Entwicklungen der Ladesäulen, der Ladestecker und der Ladefunktionalitäten.

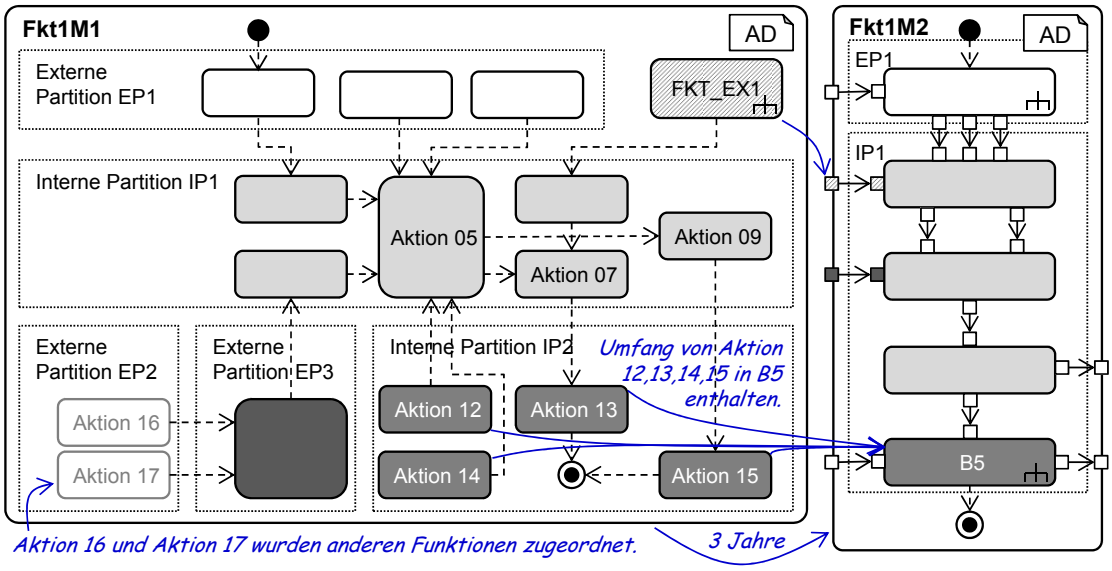


Abbildung 8.2: Vereinfacht dargestellte Funktion am Anfang der Projektlaufzeit (`Fkt1M1`) und drei Jahre später (`Fkt1M2`) aus dem Teilbereich Moment_stellen. Die Beziehungen zwischen `Fkt1M1` und `Fkt1M2` sind über die Graustufen hervorgehoben.

als Aktionen deklariert, führt das dazu, dass der Charakter `Fkt1M1` eher einem SysML-Strukturdiagramm wie einem IBD als einem Verhaltensdiagramm entspricht, da nicht die Kontrollflussinformation sondern die Information der Signalflüsse dominieren. Die intellektuell nicht eindeutig erkennbare Reihenfolge der Aktionen in `Fkt1M1` erschwert ohne Hintergrundwissen wie Domänenwissen oder textuelle Anforderungen das Verständnis des Funktionsablaufs. Ein Beispiel ist, dass sich auf `Aktion 05` keine eindeutige Reihenfolge der Aktionen `Aktion 07` oder `Aktion 09` identifizieren lässt. Zusätzlich können die nicht im Diagramm enthaltenen Informationen nicht für die maschinelle Verarbeitung verwendet werden. Eine Erklärung für diese Modellierungsart ist, dass in Modellen von bekannten Tools der Domänenexptert*innen wie Matlab/Simulink oder Modelica, die Elemente einer festgelegten Semantik entsprechen. Die Elemente in diesen Tools erfüllen eindeutig vorgegebene Aufgaben, sind visuell über Symbole unterscheidbar und sind deshalb intellektuell verständlich. Im Gegensatz zu Matlab/Simulink oder Modelica fehlen allerdings derartige Information in den SysML Knoten (hier Aktionen), die für ein intellektuelles Verständnis oder maschinelle Verarbeitung nötig sind. Denn obwohl die SysML-Elemente über ihre ID eindeutig referenziert und über Farben und Partitionen visuell gruppiert werden können, ist die Bedeutung der einzelnen Elemente stark abhängig von ihrer Benennung. Des Weiteren werden in Matlab/Simulink oder Modelica insbesondere kontinuierliche Systemfunktionen modelliert, wohingegen die Verhaltensmodelle der SysML für diskrete Systemfunktionen ausgelegt sind. In den Regelungsmodellen der kontinuierlichen Systemfunktionen wird die Reihenfolge allein über die Signalflüsse bestimmt. Über eine definierte

Abtastrate werden die diskreten Werte mit der kontinuierlichen Zeit verknüpft. In den Werten ist demzufolge die Abtastrate inbegriffen. Zu den Zeitpunkten der Abtastrate errechnet jedes Element anhand der anliegenden Eingangswerte Ergebnis(se) und gibt dieses über die Ausgangssignalflüsse weiter. Alle Modellelemente sind demnach zur gleichen Zeit aktiv und werden nicht explizit nacheinander betrachtet. Bei größeren Modellen erschwert diese Gleichzeitigkeit die intellektuelle Identifizierung der Reihenfolge von Aktionen. In Verhaltensmodellen der diskreten Systemfunktionen wird zwischen Kanten der Reihenfolge (Kontrollflüsse) und Informationsflüssen (Objektflüsse) unterschieden, dies erleichtert zwischen Informationsflüssen und dem Funktionsablauf zu unterscheiden. Folglich ist immer nur ein Element aktiv, außer es wird explizit modelliert. Im Gegensatz zu `Fkt1M1` ist `Fkt1M2` sequenziell aufgebaut, was ein Nachvollziehen der Reihenfolge von Aktionen und Aktivitäten vereinfacht. Die klare Unterscheidung von Aktionen, Ein- und Ausgängen, einem Kontrollfluss und Informationsflüssen trägt ebenfalls zum Verständnis und zur erleichterten maschinellen Verarbeitung bei. Erkenntnis 8.2 bringt es noch einmal auf den Punkt:

Erkenntnis 8.2 (Verbessertes Funktionsverständnis durch „Sequenzierung“)**.** *Die sequenzielle Reihenfolge von Diagrammen unterstützt im Gegensatz zu kontinuierlichen Regelungsdiagrammen das intellektuelle Verständnis des Aufbaus und des Verhaltens einer Funktion.*

Neben dem Einfluss der Modellierungsrichtlinien ist die Auswirkung von Umgruppierungen der Funktionen und klaren Funktionsschnittstellen ebenfalls erkennbar. So wurden die Elemente der externen Partition `EP2` komplett aus dem Diagramm entfernt, Funktionalitäten wie die der `IP2` zusammengefasst und nur notwendige Aktionen und Informationen modelliert. Die abgebildeten Aktionen wie `Fkt_EX1` oder `Aktion 12`, sind in diesem Zug auf die relevanten Informationsflüsse beziehungsweise Pins reduziert worden.

Die Entwicklung von `Fkt1M1` zu `Fkt1M2` ist mit der Einführung der Modellierungsrichtlinien aus Abschnitt 6.3 zu erklären. Zusätzlich zu den Richtlinien wurden und werden bei Bedarf Kennzahlen wie in [CMRP18] eingesetzt, um die SysML-Modelle zu optimieren. Dies ist nötig, da die SysML keine Bewertungsmethoden zur Verfügung stellt, die eine Messbarkeit der Modellkomplexität ermöglichen [CMRP18].

Informationstransformation am Beispiel von Teilbereich Laden

Auch im Teilbereich Laden war es möglich, die automatisierte Testfallerstellung und die Modellierungsrichtlinien mit Hilfe von SMArDT zu verbessern und damit die intellektuelle Beherrschbarkeit der Komplexität. Abbildung 8.3 illustriert zwei Modelle des Teilbereichs Laden. Partitionen und textuelle Anforderungen sind aus Übersichtsgründen nicht abgebildet. Aufgrund von Umgruppierungen der Funktionen ist die Funktionalität der Modelle nicht gleichwertig. Auch wenn die Funktionalitäten in Aktivität `Fkt2L2` im Vergleich zu `Fkt2L1` zugenommen haben, ist in Abbildung 8.2 die Funktionalität von `Fkt2L1` vor allem in Domänenwissen und textuellen Anforderungen hinterlegt.

Auf der rechten Seite von Abbildung 8.3 sind in `Fkt2L2` die Funktionalitäten explizit modelliert. Ehemals textuelle Informationen wurden hier in grafische Informationen

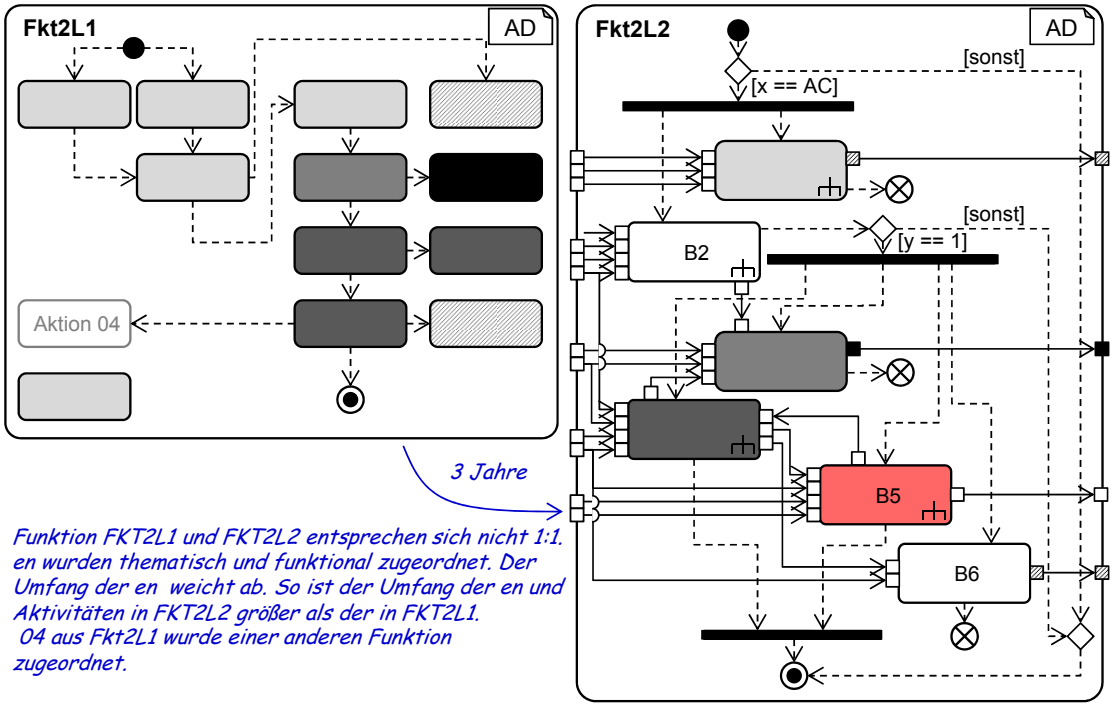


Abbildung 8.3: Vereinfachte Darstellung eines ADs am Anfang von SMArDT (`Fkt2L1`) und ein funktional äquivalentes AD drei Jahre später (`Fkt2L2`) aus dem Teilbereich Laden. Die Beziehungen zwischen `Fkt1M1` und `Fkt1M2` sind über die graue Färbung der Elemente hervorgehoben und stehen nicht im Zusammenhang mit Partitionen.

transformiert. Das Ziel der Funktionen `Fkt2L1` und `Fkt2L2` ist gleich, einen Ladestart zu ermöglichen. Allerdings sind aufgrund von Umgruppierungen der Funktionen weitere Funktionen wie `Aktion B2`, `Aktion B5` und `Aktion B6` hinzugekommen. Funktionalitäten wurden hingegen nicht aus der Funktion entfernt. Anhand der Alternativen, Verzweigungen, Ablaufendknoten und Aktionen mit eingebetteten Aktivitäten von `Fkt2L2` ist zu erahnen, wie viel Informationen zuvor nur textuell oder implizit in `Fkt2L1` vorhanden waren. Es ist zu erkennen, wie viel Komplexität in `Fkt2L1` implizit und in `Fkt2L2` explizit dargestellt ist. Außerdem ist anzumerken, dass `Aktion B5` (rot markiert), das Verhalten der Funktion im Fehlerfall modelliert. Eine solche explizite Darstellung von Verhalten im Fehlerfall erleichtert die manuelle (und automatisierte) Testfallerstellung.

Theorie und Praxis 8.1 (Design for Testability)**.** *Mit der aktiven Integration der Tester*innen in den Entwicklungs- beziehungsweise Spezifikationsprozess fließen Testbarkeitsmerkmale in das Modell und das System mit ein. Ein auf Testbarkeit ausgelegtes Design (engl. „Design for Testability“) erhöht die Testbarkeit eines Systems und dessen Subsysteme, was zu einer verbesserten Qualität führt [HMR96]. Gleichzeitig kann die Zeit bis zur Markteinführung und damit die Testkosten reduziert werden [HMR96].*

Zusätzlich wird die Fehlerbehebung bei fertigen Produkten erleichtert. Die Entwicklung konzentriert sich traditionell eher auf die Verbesserung der Spezifikationen, Prozesse und Techniken als auf das Hinzufügen expliziter Testbarkeitsmerkmalen. Zusätzlich zu den in Abschnitt 5.1 erläuterten Testbarkeitskriterien reichen diese Merkmale von einem modularen System über Zugänge zu Hardware- und Softwarekomponenten bis hin zu neuen Implementierungen von Testfunktionen. Die hinzugefügten Merkmale erleichtern die Entwicklung, Integration und Absicherung der Funktionen und Systeme. Zusätzlich können äußere Fehler im fertigen Produkt schneller analysiert und innere Fehler anhand von spezifizierten Diagnosen eindeutiger identifiziert werden. Diese integrierten und effektiven Testtechnologien sind der Schlüssel zum Erfolg auf dem heutigen wettbewerbsorientierten Markt und sollten deshalb ein Teil der Spezifikation werden.

Komplexitätsreduktion am Beispiel von Teilbereich Energiemanagement

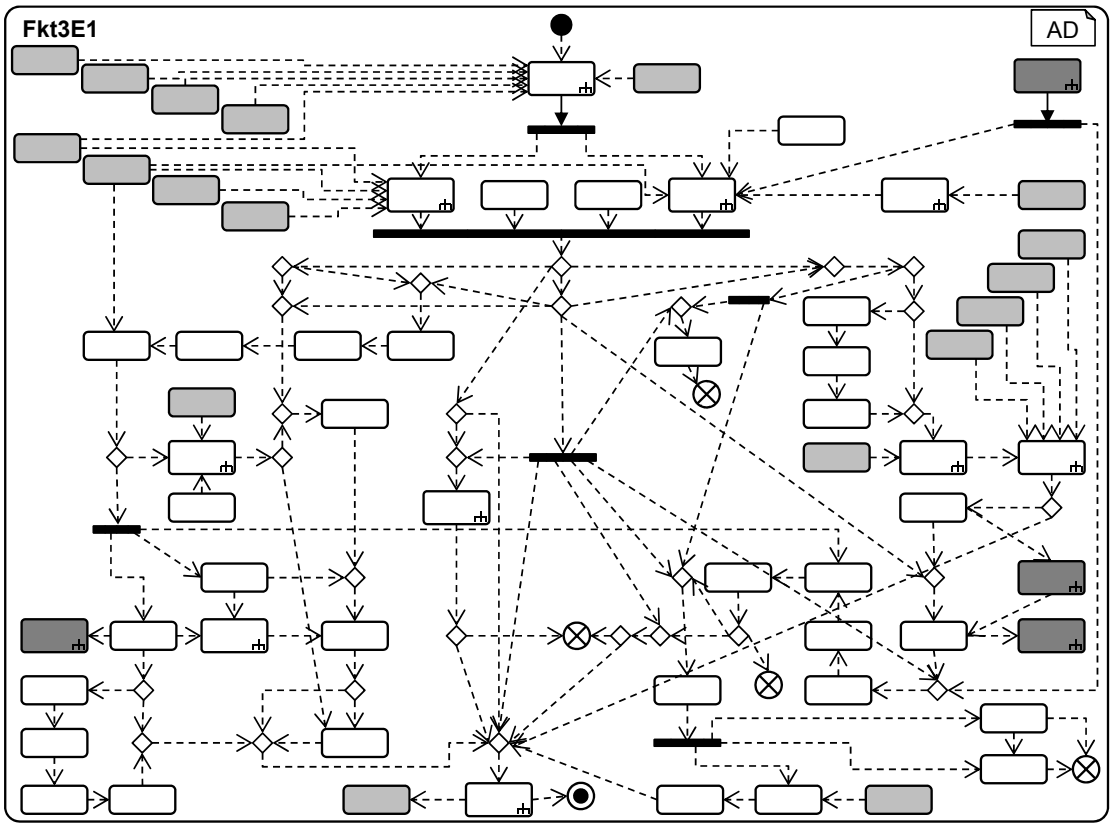


Abbildung 8.4: Vereinfachte Darstellung eines ADs des Teilbereichs Energiemanagement am Anfang von SMArDT. Die Elemente aus funktionsexternen Partitionen wurden über eine graue Färbung hervorgehoben.

Ein weiteres Szenario ist die Entwicklung des Modells aus dem Teilbereich Energie-

management (vgl. Abbildung 8.4). Aus Gründen der Übersicht wurden Partitionen, Wächterbedingungen, textuelle Anforderungen und unfertige Abschnitte des Diagramms nicht in Abbildung 8.4 abgebildet. Beide Funktionen `Fkt3E1` und `Fkt3E2` aus Abbildung 8.5 wurden/werden in dieser Form genutzt, um Testfälle mit dem entwickelten Testfallgenerator aus Kapitel 7 automatisiert zu erstellen. Es fällt auf, dass im Gegensatz zu den Diagrammen aus `Fkt1M1` und `Fkt2L1` viele Verzweigungen im `Fkt3E1` modelliert sind. Die Vielzahl an Verzweigungen kann damit erklärt werden, dass es sich bei `Fkt3E1` um eine zustandsbasierte Funktion handelt, die mit einem AD modelliert wurde.

Aus dem Grund der Zustandsbasierung wurde für das AD `Fkt3E1` in einem zweiten Anlauf eine andere Modellierungssprache der SysML, das SC gewählt. Abbildung 8.5 stellt die oberste Hierarchieebene des zustandsbasierten Modells von `Fkt3E2` vereinfacht dar. Zusätzlich wurde eine Ausführungsaktion `Fkt3E2.2` und deren eingebetteten Aktionen `Aktion 2.1` und `Aktion 2.3` dargestellt. Zwischen den Funktionen `Fkt3E1` und `Fkt3E2` liegt ebenfalls ein Zeitintervall von drei Jahren.

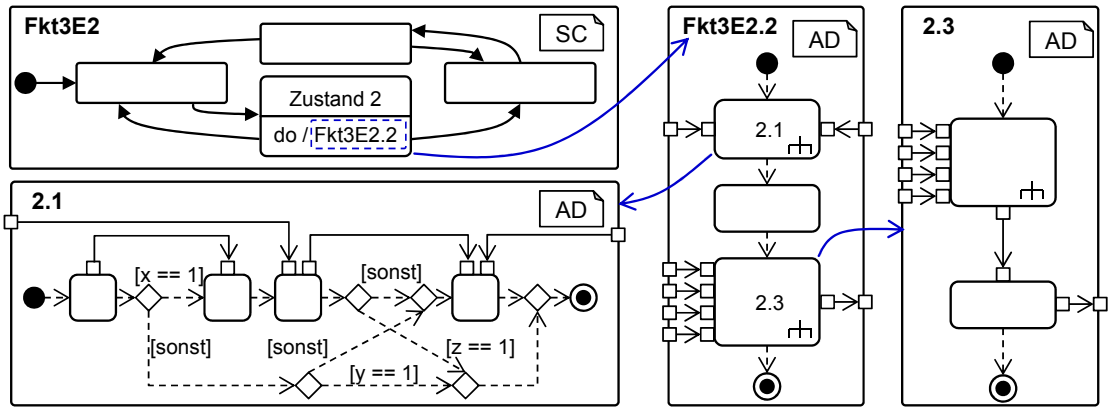


Abbildung 8.5: Vereinfachte dargestellte Funktion von Abbildung 8.4 drei Jahre später aus dem Teilbereich Energiemanagement

In den Zuständen von `Fkt3E2` sind Ausführungsaktionen (`do`) hinterlegt, die aktiv sind, während der jeweilige Zustand aktiv ist (vgl. Abschnitt 7.2). Die Möglichkeiten der Eingangsaktionen (`entry`) oder der Ausgangsaktionen (`exit`) wurden nicht genutzt. Das Ziel der Funktion und die beschriebene Funktionsweise von `Fkt3E1` ist identisch mit der von `Fkt3E2` und die Komplexität der Funktion dementsprechend gleich. Es fällt auf das mittels der versteckten Komplexität `Fkt3E2` intellektuell leichter verständlich ist, da nicht alles in einem Diagramm abgebildet ist und nur zustandsrelevante Aktionen betrachtet werden. Alle vier Ausführungsaktionen `Fkt3E2.1`, `Fkt3E2.2`, `Fkt3E2.3` und `Fkt3E2.4` der jeweiligen Zustände sind ähnlich aufgebaut. So wird die Ausführungsaktion wie in `Fkt3E2.2` vorerst in zwei bis vier eingebetteten Aktionen gegliedert und nachfolgend in diesen Aktionen weitere Aktionen modelliert. Im Beispiel von `Fkt3E2.2` werden bestimmte Aktionen in der eingebetteten `Aktion 2.1` entschieden, bevor `Aktion 2.2` ausgeführt und in `Aktion 2.3` die Ausgabe von `Fkt3E2.2` festgelegt wird. Erkenntnis 8.3 fasst die angesprochenen Punkte noch einmal zusammen:

Erkenntnis 8.3 (Das richtige Diagramm wählen)**.** *Die Wahl der Diagrammart für die Beschreibung einer Funktion wie Aktivitätsdiagramm oder Zustandsdiagramm hat bei komplexen Funktionen einen erheblichen Einfluss auf die intellektuell wahrgenommene Komplexität.*

Quantitativer Vergleich der drei Funktionen

Für ein besseres intellektuelles Verständnis werden die quantitativen Informationen aus den Modellen von Abbildung 8.2, Abbildung 8.3, Abbildung 8.4 und Abbildung 8.5 in Tabelle 8.6 zusammengefasst.

Tabelle 8.6: Quantitative Daten der Funktionen aus Moment_stellen, Laden und Energiemanagement, mit der Anzahl der Aktionen ($n_{Aktionen}$), Alternativen ($n_{Alternativen}$), Verzweigungen ($n_{Verzweigungen}$), Kanten (n_{Kanten}), Anzahl der Startknoten und eingebetteten Aktivitäten ($n_{Startknoten+Aktivitaeten}$), textuellen Anforderungen ($n_{text.Anforderungen}$), dem Mittelwert der Wörter pro Anforderung ($\overline{x}_{Woerter/Anf.}$) und der Standardabweichung der Anzahl an Wörtern pro Anforderung ($s_{Woerter/Anf.}$). Textuelle Unterdiagramme und deren Elemente und sind nicht in den Daten enthalten.

	Fkt1M1	**Fkt1M2**	**Fkt2L1**	**Fkt2L2**	**Fkt3E1**	**Fkt3E2**
n_{Aktion}	34	5	11	38	62	59
$n_{Alternativen}$	0	0	0	5	18	8
$n_{Verzweigungen}$	0	0	0	3	8	2
n_{Kanten}	118	22	41	297	392	298
$n_{Startknoten+Aktivitaeten}$	2	1	1	9	1	5
$n_{text.Anforderungen}$[3]	22	9	7	43	75	54
$\overline{x}_{Woerter/Anf.}$	40,31	53,89	15,43	33,28	26,63	31,07
$s_{Woerter/Anf.}$	67,82	68,7	2,97	26,87	17,31	21,1

Für den Teilbereich Moment_stellen ist aus Tabelle 8.6 zu entnehmen, dass die Anzahl der Aktionen, Kanten und textuellen Anforderungen abgenommen hat. Eine Erklärung ist, dass die Modellierung der Funktionen unter anderem eindeutige Schnittstellen und infolgedessen klare Funktionsgruppen des Teilbereichs Moment_stellen erwirkt hat. Für die Funktion nicht relevante Aktionen beziehungsweise Aufgaben und Anforderungen wurden in anderen Funktionen spezifiziert. Auf dieser modellbasierten Basis konnte mittels SMArDT eine funktionsübergreifende Systemarchitektur spezifiziert und im gleichen Zug die Komplexität der Funktionen reduziert werden.

Für den Teilbereich Laden ist aus Tabelle 8.6 zu entnehmen, dass die Anzahl aller Modellelemente um ein Vielfaches zugenommen hat. Die Zunahme kann damit begründet

[3] Die ausgeleiteten Rohdaten der textuellen Anforderungen wurden für die Analyse bearbeitet. Es wurden doppelte beziehungsweise identische textuelle Anforderungen einer Funktion auf eine Anforderung reduziert, da diese die Daten von $\overline{x}_{Woerter/Anf.}$ und $s_{Woerter/Anf.}$ verfälschen.

werden, dass der Funktionsumfang von `Fkt2L1` auf `Fkt2L2` zugenommen hat, auch wenn das grundlegende Funktionsziel, einen Ladestart zu ermöglichen, gleich geblieben ist. Die Anzahl der Startknoten und Aktivitäten gibt an, wie viele untergeordnete grafische Aktivitäten für die Modellierung von `Fkt2L2` verwendet wurden. Diese hierarchischen Aktivitäten ermöglichen zusammenhängende Abläufe der Funktion zu gruppieren und insofern die Komplexität leichter zu beherrschen. Zusätzlich zu den grafischen Aktivitäten werden in `Fkt2L2` textuell modellierte Aktivitäten für die Spezifikation von Unterfunktionen genutzt (vgl. Abschnitt 6.3). Diese textuellen Unterfunktionen und deren Elemente sind allerdings nicht in den Daten aus Tabelle 8.6 enthalten. Da der Umfang der textuell modellierten Unterfunktionen nur maximal eine Größe von weniger als vier Zweigen umfasst (vgl. **RL8** in Abschnitt 6.3) und in weniger als 25% der Fälle genutzt wird, führt die Vernachlässigung in Tabelle 8.6 nur zu geringen inhaltlichen Fehlern. Die Größenordnung der Fehler ist dabei so gering, dass die Aussage von Tabelle 8.6 unverändert bleibt.

Für den Teilbereich Energiemanagement ist aus Tabelle 8.6 zu entnehmen, dass die Anzahl aller Modellelemente von `Fkt3E1` nach `Fkt3E2` abgenommen hat. Die Abnahme ist damit zu erklären, dass die passende Modellierungssprache für das vorliegende zustandsbasierte Funktionsverhalten gewählt wurde. Die Wahl von SC anstatt AD auf der obersten Aktivität hatte erheblichen Einfluss auf die Reduzierung der Alternativen aufgrund der klar zugeordneten Zustände. Diese zugeordneten Zustände und deren Konsequenzen mussten infolgedessen nicht mehr über Wächterausdrücke modelliert werden. Mit der Reduzierung der Verzweigungen konnten Mehrdeutigkeiten in Funktionsabläufen gemindert werden (vgl. Erkenntnis 8.3). Über die visuelle Komplexitätsreduzierung hinaus konnte folglich auch inhaltlich die Komplexität reduziert werden. Wie auch in `Fkt2L2`, werden in `Fkt3E2` textuell modellierte Unterfunktionen genutzt. Da der Umfang der textuellen Unterfunktionen auch in `Fkt3E2` begrenzt ist und diese in weniger als 25% der Fälle genutzt werden, führt die Vernachlässigung in Tabelle 8.6 nur zu geringen inhaltlichen Fehlern. Deren Größenordnung so gering ist, dass die Aussage von Tabelle 8.6 unverändert bleibt. Erkenntnis 8.4 zieht eine Schlussfolgerung:

Erkenntnis 8.4 (Intellektuelles Verständnis führt zu inhaltlicher Komplexitätsreduzierung)**.** *Eine intellektuelle Komplexitätsreduzierung wie in Erkenntnis 8.3 hilft den Modellierenden die inhaltliche Komplexität zu reduzieren.*

Die nicht dargestellten textuellen Anforderungen der Diagramme Abbildung 8.2, Abbildung 8.3, Abbildung 8.4 und Abbildung 8.5 werden im folgenden Abschnitt exemplarisch betrachtet. Eine umfassende Analyse aller textuellen Anforderungen würde den Rahmen dieser Arbeit sprengen. Tabelle 8.6 gibt eine Übersicht der quantitativen Eigenschaften der textuellen Anforderungen. Zum einen lässt sich zwischen `Fkt1M1` und `Fkt1M2` eine Abnahme der Anzahl der Anforderungen erkennen. Zum anderen ist eine Zunahme der Wörter pro Anforderung (Mittelwert) und eine Zunahme ihrer Streuung (Standardabweichung) zu sehen. Die Reduzierung der Anforderungen ist mit der modellbasierten, erleichterten Architekturarbeit zu erklären. Die eindeutigen Schnittstellen und klaren Funktionsgruppen ermöglichen eine klare Zuordnung des Funktionsumfangs, deren Modellelemente und deren textuellen Anforderungen und dies, über die Entwicklungsdomänen

hinaus. Aus diesem Grund wird die intellektuelle Identifikation erleichtert und folglich werden Redundanzen reduziert. Zusätzlich werden Aspekte nicht mehr expliziert textuell beschrieben, da diese mit den wohldefinierten sprachlichen Vorgaben eindeutig bestimmt sind.

In Tabelle 8.6 sticht der Teilbereich Moment_stellen mit einem Mittelwert der Wörter pro Anforderung von $\overline{x}^{Fkt1M1}_{Woerter/Anf.} = 40{,}31$ und $\overline{x}^{Fkt1M2}_{Woerter/Anf.} = 53{,}89$ und einer Standardabweichung von $s^{Fkt1M1}_{Woerter/Anf.} = 67{,}82$ und $s^{Fkt1M2}_{Woerter/Anf.} = 68{,}7$ hervor. Der Grund ist eine textuelle Anforderung, die zur textuellen Beschreibung der Funktion dient. Mit $n^{Fkt1M1}_{Woerter} = 340$ und $n^{Fkt1M2}_{Woerter} = 246$ Wörtern pro Anforderung trägt diese textuelle Beschreibung zum Verständnis der Funktion bei, die nicht grafisch modelliert ist. Die grafischen Modelle entsprechen den Modellierungsrichtlinien und können folglich maschinell gelesen und verarbeitet werden. Allerdings eignen sich die Diagramme nur bedingt für eine maschinelle Verwertung, da wichtige Informationen nicht explizit modelliert sind. Tabelle 8.7 stellt die Daten der textuellen Anforderungen von `Fkt1M1` und `Fkt1M2` mit und ohne die jeweilige beschreibende Anforderung in einen Zusammenhang.

Tabelle 8.7: Quantitative Zahlen der Anforderungen der Funktionen `Fkt1M1` und `Fkt1M2` mit ($_{\text{mit}}$) und ohne ($_{\text{ohne}}$) die beschreibende Anforderung; mit textuellen Anforderungen ($n_{text.Anforderungen}$), dem Mittelwert der Wörter pro Anforderung ($\overline{x}_{Woerter/Anf.}$) und der Standardabweichung der Anzahl an Wörtern pro Anforderung ($s_{Woerter/Anf.}$).

	Fkt1M1$_{\text{mit}}$	**Fkt1M2**$_{\text{mit}}$	**Fkt1M1**$_{\text{ohne}}$	**Fkt1M2**$_{\text{ohne}}$
$n_{text.Anforderungen}$	22	9	21	8
$\overline{x}_{Woerter/Anf.}$	40,31	53,89	26,05	29,86
$s_{Woerter/Anf.}$	67,82	68,7	18,41	10,93

Die beschreibende textuelle Anforderung beschreibt in natürlicher Sprache den kompletten Funktionsablauf. Diese Beschreibung ist nur im Fall von `Fkt1M1` und `Fkt1M2` vorhanden, da in den anderen Modellen die grafischen Informationen für ein intellektuelles Verständnis ausreichen.

Die Anzahl der textuellen Anforderungen `Fkt2L1` zu `Fkt2L2` ist genau gegensätzlich zu `Fkt1M1` und `Fkt1M2`. Der Mittelwert der Wörter pro Anforderung und die Standardabweichung hat zugenommen und ähnelt dem Trend der anderen Funktionen (vgl. Tabelle 8.7). Die Zunahme der Anforderungen ist, wie auch die anderen grafischen Elemente des Modells, mit der Zunahme des Umfangs bei unverändertem Funktionsziel zu begründen.

Neben den verwendeten Daten aus Tabelle 8.6 konnte in dieser Arbeit beobachten, dass von den Anwender*innen Notizen an den SysML-Elementen angeheftet wurden. Über diese Notizen wurde über die Entwicklungsdomänen hinweg diskutiert, mit Bezug auf die Spezifikationen. Diese Informationen dokumentieren nachhaltig den Entscheidungsprozess und erleichtern folglich zukünftige Entscheidungsfindungen, auch unabhängig von der Projekterfahrung der Beteiligten.

Qualitativer Vergleich der textuellen Anforderungen

Eine weitere Beobachtung in dieser Arbeit ist, dass der Schreibstil und die Syntax der textuellen Anforderungen zunehmend unabhängiger von den verfassenden Personen werden. Dies trifft insbesondere auf maschinell verarbeitete Diagramme zu. Bei genauer Betrachtung äquivalenter textueller Anforderungen der Funktionen fällt auf, dass die grafisch „formalere“ Modellierung einen Einfluss auf die textuellen Anforderungen genommen hat. So wurde beispielsweise die textuelle Anforderung aus `Fkt1M1`,

> „Die Komponente_A[4] muss basierend auf dem Verteilungswunsch von Komponente_B das Soll-Leistungsverhältnis von Komponente_C und Komponente_D bestimmen.“ [`Fkt1M1` Ebene B]
>
> in
>
> „Die Aktion_W muss Energiefluss_X auf die Funktionsgruppe_Y und Funktionsgruppe_Z aufteilen. Es muss dabei sichergestellt werden, dass (Information_Y + Information_Z) == Energiefluss_X.“ [`Fkt1M2` Ebene B]

von `Fkt1M2` umgewandelt. Im Vergleich der beiden Anforderungen wird deutlich, dass sich die Betrachtung der Ebene B von Komponenten auf Funktionen verlagert hat. Zusätzlich ist zu erkennen, dass neben verknüpfte Modellelementen (unterstrichen dargestellt) auch zusätzlich eindeutige Informationen hinzugefügt wurden. In dem Anforderungsbeispiel wurde der Term *...Soll-Leistungsverhältnis von Komponente_C und Komponente_D bestimmen* auf *...(Information_Y + Information_Z) == Energiefluss_X* geändert. Neben eindeutigen verfolgbaren Informationen ist ebenfalls das eindeutige Verhältnis angegeben.

Die Anforderungen aus `Fkt2L2` und `Fkt3E2` greifen ebenfalls auf eindeutig verknüpfte Modellelemente zurück und sind zum Großteil mit zusätzlichen Informationen versehen. Diese vermeiden eine Mehrdeutigkeit. Ein Beispiel aus `Fkt2L2` ist, dass die Anforderung

> Die Komponente_A muss bei erkanntem Ladestecker die Rücknahme von Zustand_B bei Komponente_C anfordern.

in

> Die Komponente_A muss bei erkanntem Ladestecker die Rücknahme von Zustand_B bei Komponente_C anfordern. Die Funktion_U muss den Zustand_V von Funktion_W anfordern, wenn:
>
> - kein Zustand_LA (Information_LA == nicht_aktiv) UND
> - Zustand_LB (Information_LB == aktiv) UND
> - Zustand_LC (Information_LC == aktiv) UND
> - Zustand_LD (Information_LD == aktiv) vorliegt.

von `Fkt1M2` umgewandelt. Auch wenn die Anforderung nicht absolut klärt, wie ein *Ladestecker erkannt* wird, hat die Anforderung inhaltlich haben die textuellen Anforderungen an Nachvollziehbarkeit gewonnen. Diese gewonnene Nachvollziehbarkeit basiert auf der

[4] Für ein besseres Verständnis und aus Datenschutzgründen wurden Begriffe mit anonymisierten äquivalenten Begriffen ersetzt.

Verknüpfung der Elemente und der Eindeutigkeit, die sich in einer formaleren Sprache mit grafischer Unterstützung widerspiegelt. Dies erleichtert eine manuelle Testfallerstellung auch bei komplexen Funktionen. Eine weitere subjektive Beobachtung dieser Arbeit ist, dass auch wenn die textuellen Anforderungen aufgrund ihrer „formaleren“ Form weniger schnell gelesen werden können, sie andererseits die Chance von Fehlinterpretationen verringern. Dies kann an den klaren Beziehungen der Elemente wie $\underline{Information\_LD} == \underline{aktiv}$ und der affirmativen, nicht verneinenden Syntax liegen [WFV20]. Erkenntnis 8.5 fasst den Absatz noch einmal zusammen:

Erkenntnis 8.5 (Modellbasierung beeinflusst auch nicht formale Anforderungen)**.** *Werden Spezifikationen (grafisch) formal modelliert, hat dies ebenfalls einen Einfluss auf die Formulierung und den Umfang der in der Modellbeziehung stehenden textuellen (nicht formalen) Anforderungen.*

Die qualitative Steigerung der grafischen und textuellen Anforderungen hilft, die Komplexität des Modells beziehungsweise des Systems intellektuell beherrschbar und maschinell verwertbar zu modellieren. Dies ist wichtig, um Tester*innen und anderen Entwicklungsdomänen ein eindeutiges Verständnis für die Funktion und das System zu ermöglichen und um hochwertige Tests zu gewinnen (vgl. Theorie und Praxis 4.10). Außerdem können mit einer maschinenlesbaren Spezifikation weitere Optimierungen des Modells [CMRP18], der vorgestellten Testfallgenerierung, der Einsatz von domänenübergreifenden Schlüsselwörtern oder automatisierte Prozessketten für deren Einsatz [EJK+19] etabliert werden. Die notwendige Wohldefiniertheit und Wohlgeformtheit von Modellen für die maschinelle Verwertung und die daran gebundene Eindeutigkeit erleichtert auch die intellektuelle Verständlichkeit von Modellen und ermöglicht somit ebenfalls eine bessere manuelle Testfallerstellung.

8.1.2 Vergleich der Testfallerstellung aus natürlicher Sprache, mit manuellen und generierten modellbasierten Testfällen

In diesem Abschnitt werden Beobachtungen über die Erstellung von Testfällen auf der Basis von natürlichsprachlichen Anforderungen und auf der Basis von Modellen verglichen. Für diesen Abschnitt ist es wichtig, noch einmal zu erwähnen, dass es sich bei Modellen um wohlgeformte und wohldefinierte Modelle handelt, die mit textuellen Anforderungen angereichert werden können. Die modellbasierten Testfälle können noch in manuell und automatisiert erstellte Testfälle unterteilt werden. Da es sich bei natürlichsprachlichen Anforderungen um nicht formale beziehungsweise semiformale Elemente handelt, wird eine automatisierte Testfallerstellungsmethode für textuelle Anforderungen wie in [GBF20] in dieser Arbeit nicht betrachtet.

Der Vergleich der Testfallerstellung basiert auf einer Stichprobe von fünf Tester*innen. Angesichts der kleinen Stichprobe werden weder quantitative Aussagen getroffen noch statistische Korrelationen erhoben. Auch Aufgrund von Einflussfaktoren wie der Testumgebung, des Vernetzungsgrades der Funktionen, der Funktionsgruppe, der Erfahrung in der Funktionsdomäne, der Erfahrung der Tester*innen und des persönlichen Vorgehens können nur qualitative Aussagen getroffen werden. Testfalleigenschaften, wie die Länge

der Testfälle werden nicht betrachtet, da diese von der Arbeitsweise der Tester*innen, der Funktionsdomäne, dem Anwendungsfall und gegebenenfalls von dem Modell abhängig sind. Abbildung 8.8 stellt die Kernpunkte der Beobachtungen zusammenfassend dar.

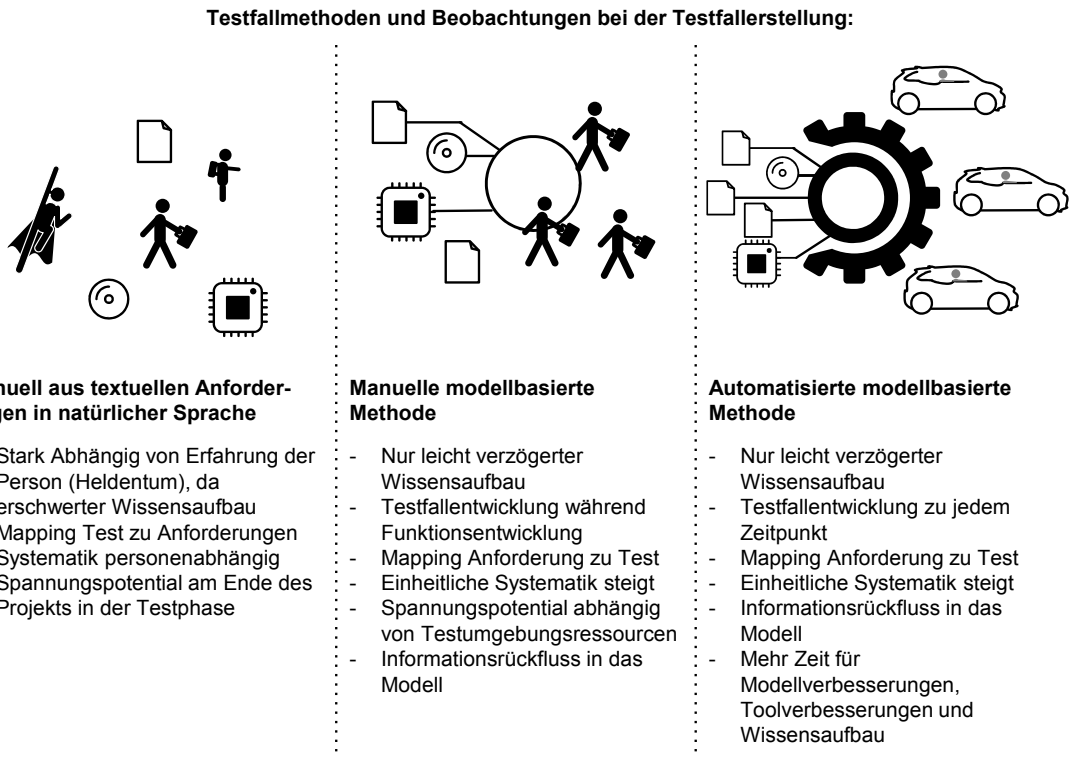


Abbildung 8.8: Kernpunkte der Beobachtungen der Testfallerstellung auf Basis von natürlichsprachlichen Anforderungen und auf der Basis von Modellen

Manuell erstellte Testfälle können von Modellen und textuellen Anforderungen abweichen, aber dies bedeutet nicht, dass diese Testfälle weniger effektiv und effizient sind. Allerdings ist diese Art der Testfallerstellung abhängig von Einzelpersonen und deren Wissen. Erfahrungsbasierte Testfälle werden zumeist nur auf textuelle Anforderungen verwiesen und nicht auf deren Basis erstellt. Infolgedessen lassen sich die Testfallerstellung und die Testfälle kaum vereinheitlichen, sind eventuell nur für die erstellende Person aussagekräftig.

Im Gegensatz zur Testfallerstellung auf Basis von textuellen Anforderungen in natürlicher Sprache können die Testfallerstellung und die Testfälle mit wohlgeformtern und wohldefinierten (grafischen) Modellen einfacher vereinheitlicht werden. Ein wohlgeformtes und wohldefiniertes (grafisches) Modell gilt auch als Basis für die Automatisierung der Testfallerstellung. In den folgenden Unterabschnitten werden die Unterschiede und Beobachtungen der verschiedenen Testfallerstellungsverfahren näher erläutert. Die Erstellung der textuellen Anforderungen beziehungsweise der wohldefinierten und wohlgeformten Modelle wird nicht betrachtet.

Testfallerstellung auf der Basis von textuellen Anforderungen

Die Testfallerstellung auf Grundalge von textuellen Anforderungen in natürlicher Sprache ist vor allem von der jeweiligen Erfahrung der testfallerstellenden Person abhängig. Das liegt daran, dass die Anforderungen oft Interpretationsspielraum offen lassen, implizit Vorwissen voraussetzen und es deshalb zu Lücken in der Absicherung kommen kann. Ein Grund für diese Lücken ist häufig das bereits fortgeschrittene Wissen der anforderungsstellenden Person. Zudem wird hier Wissen implizit vorausgesetzt, dass den Testfallersteller*innen möglicherweise nicht vorliegt. Diese Lücken werden häufig mit der Erfahrung und dem bereits erlangten Wissen der Tester*innen gefüllt.

Ist die testfallerstellende Person bereits mehrere Jahre in einem bestimmten Bereich tätig, kann häufig auf bereits bestehende etablierte Testfälle zurückgegriffen werden (vgl. Abbildung 5.8). Diese Testfälle werden in einem ersten Schritt an neue Anforderungen und Funktionalitäten angepasst und erweitert. In einem zweiten Schritt werden die Testfälle mit den Anforderungen verknüpft. Ein Vorteil dieser Herangehensweise ist die Zeitersparnis bei der Testfallerstellung. Ein Nachteil ist, dass die Verknüpfungen zwischen den bestehenden Testfällen und den „neuen“ Anforderungen für Außenstehende schwer nachvollziehbar sind. Außerdem wird auf diese Weise häufig das Review der Anforderungen von Seiten der Tester*innen weniger Beachtung gescheckt. Auf diese Weise kann womöglich auf den ersten Blick „schneller“ spezifiziert und entwickelt werden, aber einige Schwachstellen werden erst im Nachhinein in der Testphase aufgedeckt. Dieser Umstand erhöht das Spannungspotenzial am Ende des Projektes (vgl. Abbildung 3.3). In diesem erhöhten Spannungspotenzial sind vor allem wieder „Held*innen“ gefragt. Die Held*innen sind erfahrenen Personen, die kurz vor dem SOP noch schwierige Probleme lösen. Vor allem Testfallersteller*innen mit weniger Erfahrung und Vorwissen wird die Einarbeitung mit textuellen Anforderungen erschwert, auch da die Systematik für die Testfallerstellung aus nicht eindeutigen Anforderungen aufgrund der anforderungsstellenden und der testfallerstellenden Person stark variieren kann. Erkenntnis 8.6 fasst noch einmal zusammen:

Erkenntnis 8.6 (Testfallerstellung mit informalen Anforderungen ist weniger nachhaltig)**.** *Die Testfallerstellung auf Basis natürlichsprachlicher, textueller Anforderungen ist auf den ersten Blick einfach, schnell und funktioniert. Allerdings benötigt sie viel Vorwissen der testfallerstellenden Personen und trägt weniger zur Verbesserung der Anforderungen sowie der zukünftigen Entwicklungen bei.*

Manuell modellbasierte Testfallerstellung mit SMArDT

Die manuelle Testfallerstellung auf der Basis von wohlgeformten und wohldefinierten Modellen ist vor allem von der Qualität der Modelle abhängig. Umso höher die Qualität und die Quantität der Modellinformationen, desto weniger Aufwand muss in den Wissensaufbau investiert werden. Informationen liegen dann nämlich leicht zugänglich, einheitlich und verständlich vor. Im Fall einer fest vorgegebenen Spezifikationsmethode wie SMArDT können Tester*innen schon während der Funktionsmodellierung ihr Wissen aufbauen. Zusätzlich ist es möglich, über statische Tests, wie Reviews, in die Funktionsentwicklung

vorhandene Erfahrungen, Erwartungen mit einfließen zu lassen. Die statischen Tests können ebenfalls Schwachstellen der Spezifikation aufdecken und helfen Fehlerquellen früh zu vermeiden. Die modellbasierte Methode profitiert infolgedessen von dem Wissen der zuvor angesprochenen Held*innen und stellt diese allen zur Verfügung. Zusätzlich zu einer nachhaltigen Dokumentation kann mit diesem Vorgehen die Testphase entspannt werden. Der Testaufwand für neue Systeme und Funktionalitäten kann schon vor der Entwicklung abgeschätzt, Testressourcen geplant und Testfälle erstellt werden. Da die Testfälle auf Basis des wohlgeformten und wohldefinierten Modells erstellt werden, steigt die Systematik und vereinfacht die Zuordnung zu den Anforderungen. Zusätzlich können Artefakte automatisiert erzeugt werden, wie eine Testfallübersichtstabelle. Anhand dieser Tabelle können Tester*innen sich Ideen für weitere Testfälle holen. Erkenntnis 8.7 führt die Inhalte des Abschnitts noch einmal zusammen:

Erkenntnis 8.7 (Modellbasierte Entwicklung strukturiert Prozesse)**.** *Die modellbasierte Testfallerstellung benötigt initial einen hohen Aufwand der modellierenden und testfallerstellenden Personen. Der anfängliche Mehraufwand trägt zur Verbesserung der Anforderungen bei, lässt mit transparenten Ressourcen wenig Mehrdeutigkeiten zu und erleichtert somit die Testfallplanung und -erstellung.*

Generierte modellbasierte Testfälle mit SMArDT

Die generierten modellbasierten Testfälle sind von der Qualität der Modelle abhängig. Neben der Wohlgeformtheit und der Wohldefiniertheit, ohne die eine automatisierte Testfallerstellung kaum effektiv und effizient möglich wäre, ist auch der Inhalt der Diagramme essenziell. Um eine automatisierte Testfallerstellung zu ermöglichen, werden in SMArDT statische Reviews durch Tester*innen und auch automatisierte Prüfungen durchgeführt. Das gewährleistet die maschinelle Lesbarkeit. Wie auch in der „manuell modellbasierten Testfallerstellung“ können auf Basis der statischen Reviews Schwachstellen von Anforderungen früh aufgedeckt werden. Dies gilt vor allem für die Entwicklungsphase. Im Gegensatz zu der „manuell modellbasierten Testfallerstellung“ ist das Vorgehen noch systematischer und es gibt ein eindeutiges Mapping der Modellelemente (Aktionen, Blöcke, ...) und der Testfälle. Zusätzlich zu den Testfällen können ebenfalls Übersichtstabellen für manuelle Prüfungen und weitere Testfallideen genutzt werden. Ein weiterer Vorteil ist, dass auf der Basis von maschinenlesbaren Modelle weitere automatisierte Analysen, Simulationen und Methoden wie eine FMEA durchgeführt werden können. Ein Nachteil der automatisierten Testfallerstellung auf Basis der zum Teil sehr großen Modelle kann die Anzahl an erstellten Testfällen sein (vgl. Theorie und Praxis 7.2). Auch wenn ein modellbasiert automatisiert erstelltes Set an Testfällen vollständig ist, ist die Testfalldurchführung an die Prüfzeit und die Kosten für Integrationstests in Hardwaretestumgebungen begrenzt (vgl. Abschnitt 3.1). Aus diesem Grund ist häufig eine manuelle Priorisierung nötig, da derzeit noch keine Informationen über die Testumgebungen im Modell hinterlegt sind. Eine Abhilfe können weitere automatisierte Methoden wie [EJK+19, PES20, AHLL+17] schaffen, die Testfälle auf Testumgebungen anhand von Attributen und Modellen verteilen. Erkenntnis 8.8 fasst noch einmal zusammen.

Erkenntnis 8.8 (Modellgetriebene Entwicklung muss fachgerecht umgesetzt werden)**.** *Formale maschinenlesbare Modelle ermöglichen und beflügeln viele nachfolgende automatisierte Prozesse wie die modellbasierte Testfallgenerierung. Diese automatisierten Prozesse müssen allerdings zuverlässige Artefakte und einen Mehrwert liefern.*

Theorie und Praxis 8.2 (Nutzen von generierten modellbasierten Testfällen in der Zeit der Projektinitialisierung in SMArDT)**.** *Die Entwicklung der Spezifikationsmodelle in SMArDT war die der Entwicklung des Testfallgenerators voraus. Aus diesem Grund wurden die meisten Testfälle schon vor der Prototypenserie des Testfallgenerators mit der manuell modellbasierten Testfallerstellungsmethode erstellt. Diese Testfälle wurden während der Entwicklungsphase erstellt und ermöglichten schon vor dem Eintritt in die Testphase (vgl. Abbildung 3.3) eine erste Aussage über die benötigten Ressourcen zu treffen. Die automatisierte Testfallerstellung mittels Testfallgenerator wurde infolgedessen als ergänzende Maßnahme genutzt, um Absicherungslücken zu schließen. Die Evaluation der Testfälle bestätigt, dass die Aktionen der Testfälle von generierten und manuell erstellten modellbasierten Testfällen sich bei schlüsselwortbasierten Testfällen gleichen. Nicht im Modell hinterlegte Informationen, wie Prüfstandsinformationen für Vor- und Nachbedingungen des Testfalls für Testschritte müssen im Nachhinein in die Testfälle integriert werden, um eine automatisierte Testfallausführung zu ermöglichen. Dies gilt ebenso für nicht hinterlegte Schlüsselwörter der Testschritte. Diese Informationen im Modell zu hinterlegen, ist möglich und wurde geprüft. Hierfür wurde ein Testumgebungsmodell mit den nötigen Informationen erstellt. In diesem Testumgebungsmodell wurde die zu testende Funktion beziehungsweise das zu testende Diagramm eingebettet und die Testfälle generiert. Das Ergebnis waren die gleichen Testfälle, wie die der manuellen Erstellung. Da es sich hierbei nicht mehr um reine Spezifikations- oder Anforderungsmodelle handelt, wird dieses Verfahren in dieser Arbeit nicht weiter untersucht.*

Die Qualität der Testfallabdeckung für manuelle und insbesondere für die automatisierte modellbasierte Testfallerstellung steht und fällt mit der Qualität der Modelle. Werden Teile des Modells personenabhängig unterschiedlich interpretiert und/oder anders als maschinell, kann dies zu Fehlern und Qualitätseinbußen in der Absicherung führen. Um derartige Missverständnisse aufgrund von Mehrdeutigkeiten auszuschließen und eine Wohlgeformtheit und Wohldefiniertheit zu gewährleisten, wurden Modellierungsrichtlinien im Zusammenhang mit SMArDT eingeführt (vgl. Abschnitt 6.3). Diese Richtlinien dürfen die Modellierer*innen und Entwickler*innen nicht einschränken, um eine Modellierung und Entwicklung nicht zu behindern. Im nächsten Abschnitt wird eine Studie zu diesem Thema vorgestellt.

8.2 Auswirkungen der Modellierungsrichtlinien auf die Modellierung

Die Richtlinien für AD4S implizieren eindeutige Modelle, die darauf abzielen, eine gemeinsame Basis für die Interpretation der SysML-Modelle zu schaffen und gleichzeitig eine maschinelle Verarbeitung zu ermöglichen. Die positiven Auswirkungen der Richtlinien

auf die Modellierungspraxis müssen noch bewertet werden. Zu diesem Zweck wurde im Rahmen dieser Arbeit eine Studie in der Automobilindustrie durchgeführt.

8.2.1 Studiendesign und Forschungsfrage

Die Studie dieser Arbeit umfasste ein halbstrukturiertes Interview, welches die Intuitivität und den Nutzen der AD4S-Richtlinien bei der BMW Group untersucht. Dieser Abschnitt beschreibt die Forschungsfragen und die Interviewstrategie.

Forschungsfragen zum Thema Modellierung

Die Befragung soll die Intuitivität der Richtlinien prüfen, um mögliche Vorteile für die Modellierer*innen zu untersuchen. Darüber hinaus wird in einigen Fragen untersucht, ob eine der Richtlinien restriktiver sein muss. Im Rahmen des Interviews wurden die folgenden konkreten Forschungsfragen gestellt:

W1 Welchen Hintergrund und welche Erfahrungen haben die Modellierer*innen mit der SysML-Modellierung?

W2 Unterstützen die Richtlinien die Modellierer*innen bei ihrer Arbeit?

W3 Verbessern die Richtlinien das Verständnis von Modellen?

Das Hauptziel des Interviews ist es, die Sicht der Modellierer*innen auf die Richtlinien zu erfragen. Zu diesem Zweck fragt **W1** nach der Hauptfunktion der Teilnehmer*innen, ihrer beruflichen Ausbildung, ihren Erfahrungen mit einer grafischen Modellierungssprache (SysML, Matlab/Simulink) und nach der Abstraktionsschicht in SMArDT, an der sie arbeiten (vgl. Abschnitt 4.2). Zur Verdeutlichung von **W2** und **W3** wurden an verschiedenen Beispielen die Frage nach der Suffizienz, dem Nutzen, der Verständlichkeit und dem Aufwand für die Erstellung der Modelle gestellt.

Design und Durchführung der Studie

Die eine Liste von Fragen und Beispielen wurde gemäß [Ber06] entwickelt und wurde während des Gesprächs abgearbeitet. Nach [Ber06] gibt es vier Arten ein Interview zu führen:

- *informell*: Die befragte Person ist sich nicht bewusst, dass ein Interview geführt wird. Die Notizen werden im Nachhinein aus dem Gedächtnis der interviewenden Person festgehalten.
- *unstrukturiert*: Beide Personen führen bewusst ein Interview, das auf einem klaren Plan, aber auch durch ein Minimum an Kontrolle über die Antworten der Personen gekennzeichnet ist.
- *halbstrukturiert*: Das Interview erfolgt anhand einer schriftlichen Liste von Fragen und Themen, die in einer bestimmten Reihenfolge behandelt werden.
- *struktuiert*: Die befragten Personen werden gebeten, auf möglichst identische Stimuli zu antworten wie zum Beispiel einem Fragebogen.

Ein halbstrukturiertes Interviews wurde gewählt, um aus den gewonnenen Daten kontextbezogene, zuverlässigere, vergleichbarere und qualitativ hochwertigere Informationen als aus einem unstrukturierten Interview zu gewinnen. Darüber hinaus werden im Gegensatz zu einem strukturierten Interview der interviewenden Person die Freiheit gelassen, weitere Details und Hintergründe zu erfragen.

Die Interviews wurden zwanglos gehalten, mit dem Ziel, neue Perspektiven auf die Modellierungspraxis in SMArDT aufzuzeigen und zu einem besseren Verständnis des/der Befragten beizutragen. Die Liste besteht aus zehn Fragen, die in zwei Abschnitte unterteilt sind. Der erste Abschnitt untersucht den Hintergrund der Teilnehmer*innen, wie beispielsweise die berufliche Ausbildung. Tabelle 8.9 zeigt die Fragen F1 bis F6 bezüglich des Hintergrunds der Teilnehmer*innen (vgl. **W1**).

Tabelle 8.9: Interviewfragen zum Hintergrund der Teilnehmer*innen, Fragetyp und Bezug zu den Forschungsfragen

ID	Frage	Fragetyp	Zuordnung
F1	Was ist Ihre Haupttätigkeit?	Multiple Choice	W1
F2	Was haben Sie studiert? oder welchen Beruf haben Sie erlernt?	Multiple Choice	W1
F3	Wie schätzen Sie ihre Erfahrungen mit der grafischen Modellierung ein?	Likert-Skala	W1
F4	Arbeiten Sie mit SMArDT?	Multiple Choice	W1
F5	Auf welcher Abstraktionsebene arbeiten Sie?	Multiple Choice	W1
F6	Auf welcher Dekompositionsschicht arbeiten Sie?	Multiple Choice	W1

Der zweite Abschnitt enthält Beispiele mit verknüpften Fragen zur Bewertung der Richtlinien. Die Strategie bestand darin, den Teilnehmer*innen zwei Arten von Beispielen vorzustellen. Jedes Beispiel ist entweder AD4S-richtlinienkonform oder nicht.

Jede Gruppe von Beispielen bezieht sich auf eine oder mehrere Modellierungsrichtlinien, die in Abschnitt 6.3 vorgestellt wurden. Auf der Grundlage dieser Beispiele wurde geprüft, ob die Befragten die jeweiligen Richtlinien als verständlich oder unintuitiv wahrnehmen. Außerdem wird durch die Präsentation verschiedener Varianten der Richtlinien untersucht, ob eine Leitlinie restriktiver oder auf andere Weise angepasst werden sollte. Deshalb wurden die Befragten gebeten zu bewerten, ob

- das Beispiel für ihre Arbeit **ausreichend** ist,
- das Beispiel **nützlich** ist für weitere Aufgaben, die sie ausführen,
- das Beispiel und die modellierte Funktionalität **klar** und **verständlich** ist und
- das Verhältnis des **Modellierungsaufwand**s im Vergleich zu den anderen Beispielen angemessen ist.

Insgesamt wurden 14 professionelle Entwickler*innen und Modellierer*innen aus dem Bereich elektrische Antriebe der BMW Group und einen Automobilzulieferer der FEV Europe GmbH eingeladen. Die Interviews fanden vom 10. August bis zum 12. September 2018 statt. Von den 14 eingeladenen professionellen Softwareentwickler*innen, die an

SMArDT beteiligt waren, wurden schließlich 13 interviewt, da eine angeschriebene Person auf die Interview-Einladung nicht geantwortet hat.

Von den 13 Interviews wurden sieben persönlich und sechs per Videokonferenz durchgeführt. Jedes Interview wurde von demselben Interviewer und mit jeweils einem*einer Modellierer*in durchgeführt. Der angefertigte Fragebogen lag der befragten Person und dem Interviewer vor. Die Teilnehmer*innen wurden ermutigt, frei zu sprechen. Die Interviews wurden nicht aufgezeichnet, der Interviewer machte handschriftliche Notizen, die den Teilnehmer*innen nach dem Interview vorgelegt wurden. Die Notizen der Befragungsergebnisse können aus Gründen der Vertraulichkeit und der Wahrung der Anonymität nicht veröffentlicht werden. Alle Gespräche wurden auf Deutsch geführt. Im Durchschnitt dauerte ein Interview eine halbe Stunde.

8.2.2 Interview

In diesem Abschnitt werden die Ergebnisse des Interviews vorgestellt. Im ersten Teil wird der Hintergrund der Befragten vorgestellt, im zweiten Teil werden die Interviewbeispiele zusammen mit den Angaben der Befragten detailliert präsentiert.

Hintergrund der Modellierenden

Tabelle 8.10 zeigt die Fragen F1, F2, F4, F5 und F6 bezüglich des Hintergrunds der Teilnehmer*innen. Die Fragen F1, F2, F4 und F5 untersuchen den individuellen Hintergrund der Teilnehmer*innen, die in Bezug auf **W1** relevant sind. Die Frage **F1** behandelt die Haupttätigkeit der Befragten. Die Haupttätigkeit sollte den Zweck der Modellierung und der Modelle hervorheben. Vor allem die Definition von funktionalen Anforderungen (92,31%) wurde als Haupttätigkeit angegeben. Nur einige wenige Befragte gaben an, dass ihre Hauptaufgabe in der Erstellung von Diagnosen oder Sicherheitsanforderungen besteht. Obwohl sich die grafische Modellierung in der Abteilung für elektrische Antriebe etabliert hat, gaben alle Befragten (100%) an, dass textuelle Anforderungen nach wie vor ein unverzichtbarer Bestandteil des Anforderungsmanagements sind. Die Personen, die nicht angaben, dass ihre Haupttätigkeit das Modellieren von grafischen und textuellen Anforderungen sei, gaben an, direkt an textuellen Anforderungen und in der Softwareentwicklung tätig zu sein.

Um die bisherige Erfahrung der Befragten zu berücksichtigen, wurde nach dem beruflichen Hintergrund (F2) gefragt. Ähnlich wie bei der Auswertungen in Abschnitt 5.2 haben mehr als 90% der Expert*innen ihren letzten Abschluss in Elektrotechnik (53,85%), Maschinenbau (30,77%) oder Mechatronik (7,69%) gemacht. Die Informatik ist mit 7,69% weniger vertreten.

Ein wichtiger Aspekt, um den Nutzen der Richtlinien angemessen zu bewerten, ist die Erfahrung der Modellierer*innen in SysML und grafischen Modellierungssprachen im Allgemeinen (vgl. **W1**). Es wurde angenommen, dass Modellierer*innen mit weniger Erfahrung in der SysML präzisere Richtlinien und auch Erklärungen benötigen. Als Beispiel für grafische Modellierungstechniken/-sprachen erwähnte der Interviewer Matlab/Simulink,

Tabelle 8.10: Antworten des Interviews zum Hintergrund der Teilnehmer*innen mit der Stichprobengröße (n).

ID	**Frage** / Antwortoptionen	**Ergebnis**
F1	**Was ist Ihre Haupttätigkeit?**	($n = 13$)
	Erstellung von funktionalen Anforderungen:	92,31%
	Erstellung von funktionale Sicherheitsanforderungen:	7,69%
	Erstellung von Diagnose-Anforderungen:	15,38%
	Erstellung von textuellen Anforderungen:	100%
	Erstellung von grafischen Anforderungen:	76,92%
	Softwareentwicklung:	23,08%
F2	**Beruflicher Hintergrund:**	($n = 13$)
	Elektrotechnik:	53,85%
	Informatik:	7,69%
	Maschinenbau:	30,77%
	Mechatronik:	7,69%
	Sonst:	0,00%
F4	**Arbeiten Sie mit SMArDT?**	($n = 13$)
	Ja:	100,00%
	Nein:	0,00%
F5	**Abstraktionsebene:**	($n = 13$)
	Ebene A:	61,54%
	Ebene B:	76,92%
	Ebene C:	38,46%
	Ebene D:	38,46%
F6	**Dekompositionsschicht:**	($n = 13$)
	Schicht Gesamtsystem:	0,00%
	Schicht Antrieb:	23,08%
	Schicht E-Antrieb:	100%
	Schicht Komponente:	7,69%
	Sonst: „Übergreifend für Laden“	7,69%

UML, SysML und Entity-Relationship[5]. Abbildung 8.11 veranschaulicht, dass Matlab/Simulink unter den Teilnehmern*innen die bekannteste grafische Modellierungssprache ist.

Zum Zweck der statistischen Datenanalyse wurde eine gewöhnliche numerische Skala mit einer Likert- Skala von eins (keine Erfahrung) bis fünf (sehr erfahren) beschriftet. Der Mittelwert von $\overline{x} = 3{,}54$ und die Standardabweichung $s = 1{,}39$ deuten an, dass es eine große Bandbreite an Erfahrung mit SysML gibt. Bei Matlab/Simulink gaben mehr als 70% der Befragten an, dass sie „sehr erfahren“ sind. Diese Beobachtung deckt sich mit dem Einsatz von Tools wie Matlab/Simulink in [LMT+18]. Andere Modellierungssprachen wurden von den Befragten nicht erwähnt.

[5]In der Studie gingen die Befragten nur auf Matlab/Simulink und SysML ein.

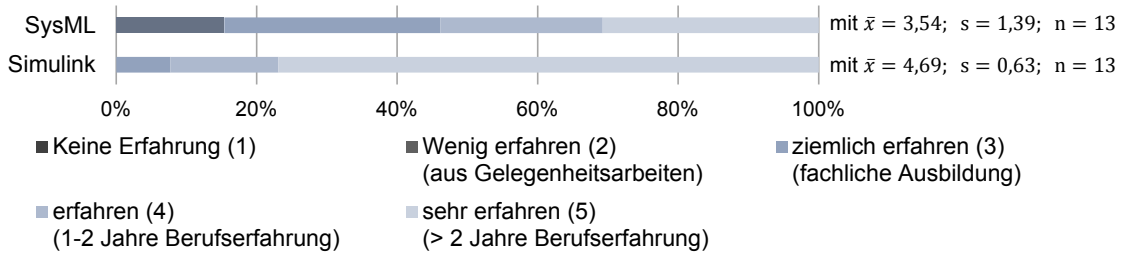


Abbildung 8.11: Die Erfahrung in der grafischen Modellierung in SysML und Simulink mit dem Mittelwert ($\overline{x}$), der Standardabweichung (s) und der Stichprobengröße (n)

Frage F4 stellt im Kontext der ersten Befragung sicher, dass die Teilnehmenden mit SMArDT arbeiten. Damit wurden die Vorkenntnisse bestätigt, die für die Beantwortung der Fragen zur Modellierung erforderlich sind.

Des Weiteren wurde nach der Abstraktionsebene in F5 und der Dekompositionsschicht in F6 gefragt. Dies sollte helfen die Antworten zu den kommenden Beispielen besser einordnen zu können. Die Mehrheit der Teilnehmer*innen arbeitet auf der SMArDT-Ebene B gefolgt von der Ebene A. Auch die Ebenen C und D waren mit 38,46% vertreten. Alle Befragten zählten sich zu der Dekompositionsschicht des elektrischen Antriebs (hier n) und vereinzelt zu der übergeordneten Schicht Antrieb ($n-1$) und der untergeordneten Schicht der Komponenten ($n+1$). Ein Teilnehmer bemerkte, dass er für das Thema Laden zuständig sei, das sich auf allen Schichten abspielt.

Nach Vorstellung der Ergebnisse zum Hintergrund der Modellierenden werden im nächsten Abschnitt die Ergebnisse der Richtlinienbewertung vorgestellt.

Fragen zur Modellierung

Als Einstieg zu den Richtlinien aus Abschnitt 6.3 wurde gefragt (Frage F7), ob es sinnvoll sei, bestimmte Elemente derselben Klasse mit Stereotypen zu versehen, um sie später zum Beispiel farblich zu kennzeichnen oder einzuordnen. Diese Frage bejahten alle Teilnehmer*innen (100%).

Anschließend wurde zu den Fragen mit den entsprechenden Beispielen übergeleitet. Abbildung 8.12 veranschaulicht die Beispiele zur Untersuchung der Richtlinien **RL6** und **RL7**. Der Beispielsatz untersucht zum einen den Nutzen von verschachtelten Aktivitäten im Vergleich zu booleschen Ausdrücken **RL7**. Die Beispiele E1.2 - E1.6 könnten eine verschachtelte Aktivität von E1.1 sein. Zusätzlich illustrieren E1.3 bis E1.6 verschiedene Wege, um die Wächterausdrücke einer Verzweigung darzustellen.

Das Beispiel E1.1 umfasst eine Aktivität, `Licht schalten` und die damit verbundenen Anforderungen `X1`, `X2` und `X3`. Die Anforderungen sind in einem semiformalen Stil verfasst, um keine Beispiele herauszuheben. Alle Anforderungen gelten für alle Beispiele, aber die grafische Darstellung ist in E1.2 bis E1.6 detaillierter. E1.2 enthält keine Wächterausdrücke, während E1.3 informelle, in natürlicher Sprache verfasste Wächterausdrücke

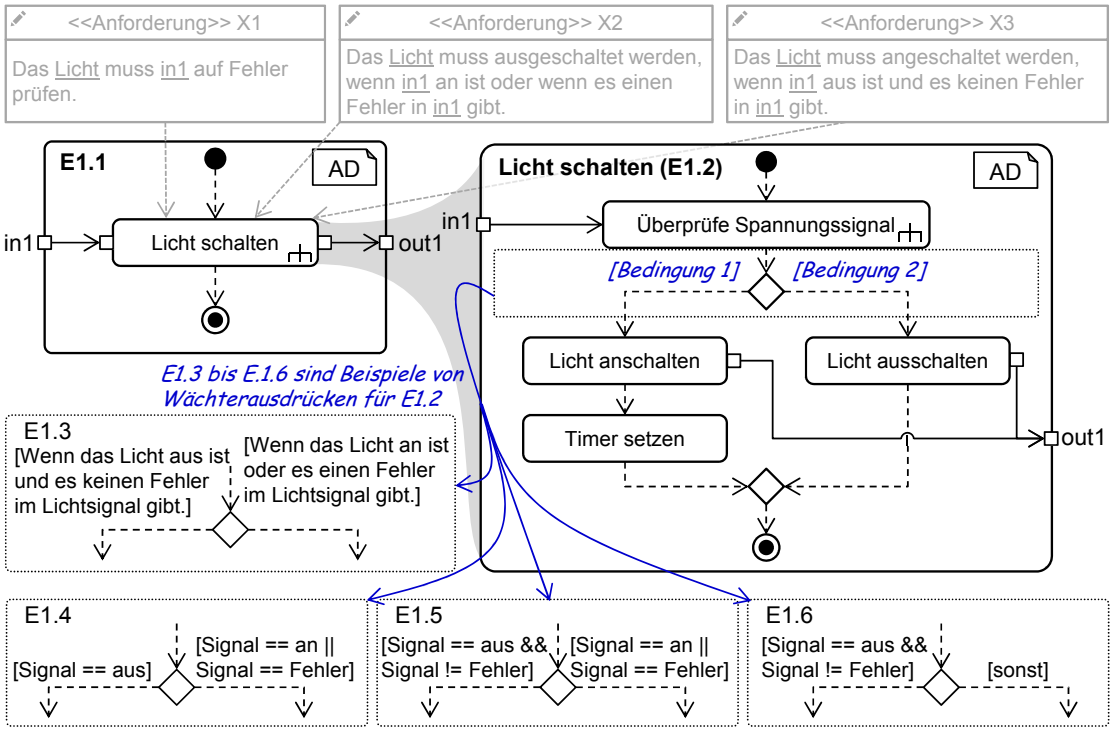


Abbildung 8.12: Beispiel von Aktivitäten, die alle die selbe Funktionalität mit den gleichen Anforderungen repräsentieren. Die Beispiele E1.3 bis E1.6 stellen verschiedene Möglichkeiten der Wächterausdrücke von E1.2 dar.

aufgreift. Alle anderen Wächterausdrücke (E1.4 - E1.6) sind formal und entsprechen der Richtlinie **GL6**, da boolesche Ausdrücke verwendet werden. Sie unterscheiden sich jedoch durch einen Grad der Mehrdeutigkeit und durch das Ausmaß, in dem sie für die automatisierte Testfallgenerierung verwendet werden können (vgl. Kapitel 7). E1.4 und E1.5 zeigen eindeutig formale Bedingungen. E1.5 vermittelt alle Informationen explizit, das heißt `Signal != Fehler`, während E1.4 eine Unterspezifikation für diesen Fall enthält. Die `sonst`-Bedingung von Beispiel E1.6 ist formal, bietet aber im Vergleich zu E1.4 und E1.5 weniger explizite Informationen für die Benutzer*innen. Alle Beispiele sind in der Praxis von SMArDT aufgetreten.

Für eine Bewertung wird zwischen der Menge der positiven und negativen Kommentare unterschieden. Darüber hinaus wird die Gesamtzahl der zu untersuchenden Kommentare, welches Beispiel am meisten diskutiert wurde, gezählt. Tabelle 8.13 veranschaulicht die Ergebnisse der Untersuchung zu den Beispielen aus Abbildung 8.12. Hierbei liegt das Hauptaugenmerk auf der Angemessenheit (ausreichen), den Nutzen und der Verständlichkeit/Klarheit.

Trotz eines Befragten, der *keinen zusätzlichen Wert in der Modellierung sieht*, bewerteten die Befragten E1.1 als positiv für hohe Abstraktionsschichten, wie die SMArDT-Ebene *A* (Tabelle 8.13). Einige der Befragten erwähnten auch, dass Systemaspekte, die

Tabelle 8.13: Das Ergebnis der Kommentare zu den Beispielen, E1.1 bis E1.6, von Abbildung 8.12 als positive (p) und negative (n) Kommentare. Mehrere Kommentare sind möglich.

	E1.1		**E1.2**		**E1.3**		**E1.4**		**E1.5**		**E1.6**	
	p	n	p	n	p	n	p	n	p	n	p	n
Angemessenheit:	6	0	2	0	0	1	5	0	5	0	3	0
Nutzen:	4	1	2	0	0	1	5	0	1	0	3	0
Verständlichkeit:	1	0	1	0	0	2	4	0	6	0	2	0
Kommentare gesamt:	12		5		4		14		12		8	

nicht im Fokus der betrachteten Funktionalität stehen, mit weniger Details modelliert werden sollten. Wenn beispielsweise die Aktivität `Licht schalten` für die modellierte Funktionalität nicht von Interesse ist, sehen die Befragten keinen Nutzen darin, alle Details zu modellieren, wie es in E1.2 aufgeführt ist. Die meisten Befragten äußerten eine Abneigung gegen die informellen Wächterausdrücke aus E1.3. Das Beispiel E1.2 wurde weniger diskutiert als die anderen „positiven“ Beispiele. Für weniger abstrakte SMArDT-Schichten, wie Ebene B, C und D, empfehlen die Modellierer*innen formale Wächterausdrücke. Die meisten Befragten empfanden E1.4 als positiv, gefolgt von E1.5 und E1.6. Ein Teilnehmer äußerte sich zu E1.6: „So würde ich es bei der Implementierung machen“. Andere merkten an, dass in den frühen Entwicklungsschichten der Signalwertebereiche noch nicht definiert ist. Ein weiterer Kommentar war, dass „die `sonst`-Option Freiheit bei der Modellierung lässt, während weitere Signalwerte vorgesehen werden“.

Die Modelle in Abbildung 8.14 bilden die Grundlage für Diskussionen über Richtlinien **RL2**, **RL3**, **RL4** und **RL9**. Mit dieser Gruppe von Beispielen wurde untersucht, ob die Richtlinie **RL9** restriktiver sein sollte. Das heißt, ein Objektfluss muss vorhanden sein, selbst wenn ein paralleler Kontrollfluss mit derselben Quelle und demselben Ziel existiert. Darüber hinaus wird in den Beispielen untersucht, ob die Leitlinie **RL2** (Jedes AD muss einen Startknoten und einen Endknoten enthalten) angepasst werden muss. Die Konsequenz wäre, dass jedes AD mindestens einen Kontrollfluss enthalten müsste.

Die Beispiele E2.1 - E2.3 realisieren alle die gleichen unterspezifizierten textlichen Anforderungen. Die Anforderung ist, dass die Lichter rot, gelb oder grün schalten können, aber niemals gleichzeitig rot und grün leuchten. Das Beispiel E2.1 zeigt eine Aktivität ohne Kontrollfluss. Sie kann als gleichzeitige oder noch nicht vorgegebene Ausführung interpretiert werden. Trotz des parallelen Kontrollflusses ist die Ausführungsreihenfolge von E2.1 und E2.2 offen für diese Interpretation. E2.3 definiert eine vorgegebene Ausführung durch einen Kontrollfluss.

Die Gesamtzahl der positiven und negativen Kommentare zu dieser Gruppe von Beispielen ist gering, da fast alle Teilnehmer*innen keine konkreten Bewertungen abgeben wollten. Die meisten der Befragten gaben an, dass es keinen Nutzen von E2.2 im Vergleich zu E2.1 gibt. Darüber hinaus erwähnte ein Befragter, dass *die zusätzlichen Linien*

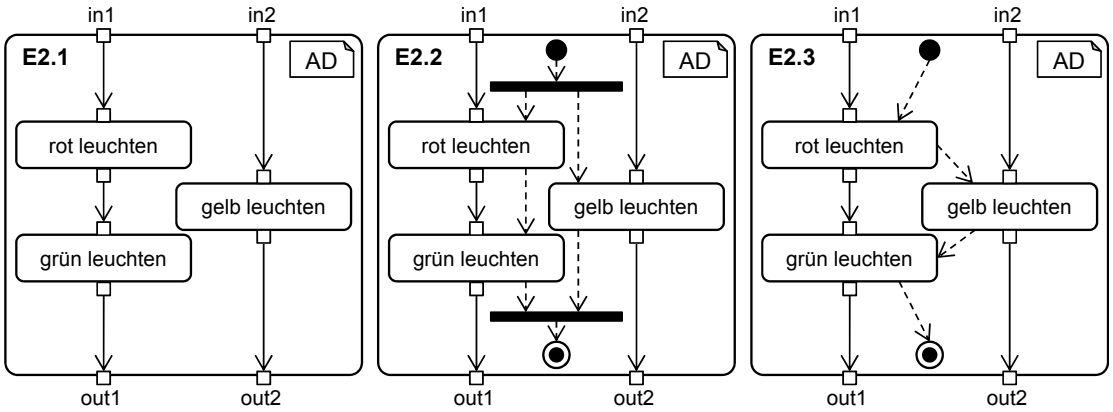


Abbildung 8.14: Drei Aktivitäten, die eine ähnliche Funktionalität repräsentieren. Die Anforderungen sind bei allen Beispielen gleich und nicht ausreichend spezifiziert.

der Kontrollflüsse das Verständnis erschweren und komplexe Diagramme durcheinander bringen. Die Ansicht der Befragten zu E2.3 war anders. Hier ist der Kontrollfluss unverzichtbar, da der Kontrollfluss eine Ausführungsreihenfolge vorgibt. Im Nachhinein überarbeiteten einige Teilnehmer ihre Antwort für E2.2. Beispielsweise erwähnte ein Befragter, dass *die Richtlinien für jedes Diagramm gleich sein sollten. Daher ist auch ein paralleler Kontrollfluss neben einem Objektfluss wichtig. Das zeigt, dass diese Information für das Verständnis der modellierten Funktionalität wichtig ist. Es gibt demnach keinen Spielraum für Interpretationen.*

Zusätzlich zu den expliziten Fragen zu E1.x und E2.x lassen die Beispiele implizit auch Rückschlüsse auf die Richtlinie **RL1** zu, da sie auf die Verwendung bestimmter AD-Elemente abzielen.

Abbildung 8.15 bietet die Grundlage für die Diskussion der Richtlinien **RL7** und **RL8**. Um die Präferenzen bei der Wahl der formalen Tätigkeitsbeschreibung zu bestimmen, wurden vier Beispiele vorgestellt. Ein Beispiel wird ohne formale Beschreibung dargestellt (E3.1). Die Beispiele E3.2 und E3.3 unterscheiden sich durch die Art der textlichen Beschreibung. Das Beispiel E3.2 hat einen vollständig definierten Wertebereich, während die Beschreibung von E3.3 die `sonst`-Option anzeigt. E3.4 modelliert eine Lösung unter Verwendung eines verschachtelten ADs.

Am meisten Aufmerksamkeit schenkten die Befragten dem Beispiel E3.4, das mehr als doppelt so viele Kommentare erhielt als jedes andere Beispiel (vgl. Tabelle 8.16). Diese Art der formalen Beschreibung wird als nützlich $(+4/-1)$ und als die verständlichste/klarste $(+8/-1)$ Darstellung angesehen. Dennoch assoziieren viele Befragte mit dem Beispiel E3.4 einen hohen Aufwand. Auch in der Studie [LMT+18] wird die Modellierung von Anforderungen mit einem erhöhten Aufwand in Verbindung gebracht. Insbesondere die Modellierung der ADs bedeutet zunächst einen hohen Aufwand, während bei der Pflege von solchen Modellen weniger Aufwand erwartet wird, verglichen mit der textuellen Beschreibung in E3.2 und E3.3. Die textuellen formalen Beschreibungen werden als

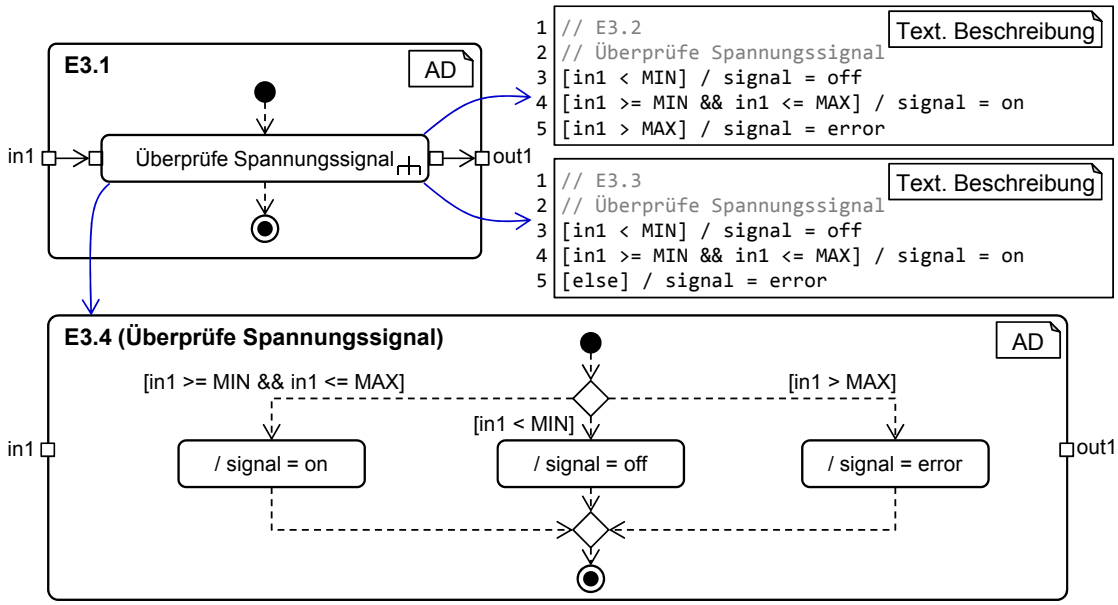


Abbildung 8.15: Eine Aktivität (E3.1) mit drei verschiedenen Beispielen für die Beschreibung des funktionalen Verhaltens

Tabelle 8.16: Das Ergebnis der Kommentare zu den Beispielen, E3.1 - E3.4, von Abbildung 8.15 als positive (p) und negative (n) Kommentare. Mehrere Angaben sind möglich.

	E3.1		**E3.2**		**E3.3**		**E3.4**	
	p	n	p	n	p	n	p	n
Angemessenheit:	2	0	2	1	4	1	4	0
Nutzen:	4	0	3	1	1	1	4	1
Verständlichkeit:	1	1	1	0	0	0	8	1
Aufwand:	0	0	0	2	0	2	4	5
Kommentare gesamt:	8		10		9		27	

weniger nützlich und weniger umfassend empfunden. Obwohl solche Beschreibungen mit weniger Aufwand erstellt werden können, ist die Pflege mit erheblicher Arbeit verbunden. Dies begründen die Befragten damit, dass die Darstellungen schwieriger zu lesen sei und sich somit mehr Fehler einschleichen würden. Hinsichtlich des Wertebereichs bevorzugen zwei Befragte einen vollständig spezifizierten Wertebereich gegenüber der `sonst`-Option, da dies eindeutiger ist. Bezüglich der Aktivität in E3.1 wurde bemerkt, dass eine formale Beschreibung, wie in Abbildung 8.15 abgebildet, unabdingbar für ein Verständnis dieser Aktivität/Aktion ist. Der Arbeitsaufwand von E3.1 wurde von den Befragten nicht erwähnt. Zwei der Befragten merkten an, dass das Beispiel E3.1 für

die Modellierung ausreichend und nützlich wäre, wenn diese eine zusätzliche informelle textuelle Beschreibung enthalten würde.

8.2.3 Validität der Ergebnisse des Interviews

Die Interviewergebnisse sind verschiedenen, für die empirische Forschung typischen Gefahren ausgesetzt. Dieser Abschnitt berichtet über diese Gefahren und damit verbundene Studien.

Im Vergleich zu einer Umfrage zeichnet sich das Interview durch eine detaillierte und qualitative Kontextbeschreibung aus, um eine komplexe reale Situation und damit verbundene Gefühle zu analysieren. Dazu gehören auch Informationen darüber, wie geeignet die richtlinienkonforme Modellierung ist (vgl. Abschnitt 6.3). Die größte Gefahr für die Validität ist daher die Befangenheit.

An dem Interview waren 13 Entwickler*innen der BMW Group und FEV Europe GmbH in der Abteilung für elektrische Antriebe beteiligt. Obwohl im Rahmen des Interviews „nur“ 13 Teilnehmer*innen befragt wurden, hält der Autor die Ergebnisse dennoch für aussagekräftig, da es sich bei den Teilnehmer*innen um Expert*innen in der Entwicklung des elektrischen Antriebs in der Automobilindustrie handelt. Da das Interview insbesondere Teilnehmer*innen der BMW Group einschloss, ist eine Verallgemeinerung der Ergebnisse und Schlussfolgerungen für eine modellgetriebene Entwicklung schwierig.

Um vergleichbare neutrale Daten von hoher Qualität zu erhalten, ging der Interviewer nach einer vordefinierten Liste von Fragen vor [Ber06]. Der Interviewer ist ein männlicher Doktorand der BMW Group und der RWTH Aachen. Er kannte acht Teilnehmer*innen im Vorfeld und ist mit dem Testfallgenerator vertraut. Der Interviewer blieb objektiv und gab den Befragten an, ohne Bedenken oder spontan zu antworten, da die Ergebnisse und Notizen nur anonymisiert herausgegeben werden. Dennoch kann nicht ausgeschlossen werden, dass die Teilnehmer*innen durch die Zugehörigkeit des Interviewers beeinflusst wurden. Ein Modellierer verwendet den Testfallgenerator. Daher wurde er gebeten, seine objektive Meinung aus der Sicht des Modellierers zu äußern. Zwei Befragte hatten wenig Modellierungserfahrung, da sie erst seit wenigen Wochen mit SMArDT arbeiten und noch keine SMArDT-Schulung besucht hatten. Das Ziel war, eine Vielzahl von Modellierer*innen für die Auswertung einzubeziehen. Folglich ist die Meinung der Personen, die den Testfallgenerator verwenden, genauso wertvoll wie die Meinung von Unerfahrenen, die den Sachverhalt von einem objektiveren Standpunkt aus beurteilen können. Da der Einfluss des Interviewers minimiert werden sollte, ließ der Interviewer die Befragten frei sprechen und lenkte nur ab und zu bei abschweifenden Themen auf die Fragen zurück. Daher wurden einige Beispiele mehr diskutiert als andere.

Die meisten Teilnehmer*innen haben eine Meinung über Abstraktionsschichten, für die sie nicht direkt entwickeln oder spezifizieren. Daher wurden sie gebeten, von ihrem Abstraktionspunkt aus zu antworten und die Daten zu anderen SMArDT-Ebenen für die Auswertung verworfen. Außerdem konzentrierte sich das Interview auf eine Modellierungsmethodik, nicht auf die Anwendbarkeit von grafischen Modellierungswerkzeugen. Daher wurden auch Daten mit positiven und negativen Kommentaren zu Werkzeugen und Verbesserungsvorschlägen für diese Bewertung verworfen.

Um Fachdiskussionen zu vermeiden, sind die vorgestellten Beispiele einfach und ohne Bezug zum Fachgebiet der Expert*innen. Die Beispiele umfassen fokussierte Anwendungsfälle aus den realen Produktionsmodellen. Die Daten von realen Szenarien mit einer höheren Komplexität können sich von den hier gewonnenen Daten unterschieden.

8.2.4 Fazit der Umfrage

Auf der Grundlage der Interviews kann der Schluss gezogen werden, dass die Modellierer*innen eine positive Einstellung zu den Modellierungsrichtlinien aufweisen. Nach dem Eindruck des Interviewers lassen sich die Teilnehmer*innen in zwei Gruppen teilen. Während die eine Gruppe gerne ungezwungen modelliert, um die Kreativität zu steigern, scheint die andere Gruppe strenge Richtlinien zu bevorzugen. Beide Parteien waren sich einig, dass Richtlinien wesentlich sind, um ein gemeinsames Verständnis der Modelle zu erhalten (**W2**, **W3**). Diese Haltung ist weitgehend unabhängig von der Hauptfunktion, der beruflichen Ausbildung, der Erfahrung mit grafischen Modellierungssprachen und der benannten Abstraktionsebene in SMArDT (**W1**). Darüber hinaus müssen die Richtlinien für alle Erfahrungslevel anwendbar sein. Es kann ebenfalls der Schluss getroffen werden, dass verschachtelte Aktivitäten die Verständlichkeit der Modelle erhöhen (vgl. Tabelle 8.16). Die Schaffung einer gemeinsamen Basis für das Modellverständnis erfordert eine verstärkte Koordination, führt aber letztendlich zu qualitativ hochwertigeren Modellen.

Während des Interviews wurde klar, dass die Experten*innen des elektrischen Antriebsstrangs an SysML weniger gewohnt sind als an Matlab/Simulink. Matlab/Simulink wird häufig zur Modellierung kontinuierlicher Systeme verwendet (vgl. Unterabschnitt 4.2.4). In Anbetracht des Hintergrunds der Befragten ist die umfangreiche Erfahrung mit Matlab/Simulink nicht überraschend und das Ergebnis unterscheidet sich infolgedessen von [Stö17].

Die Teilnehmer*innen empfanden Stereotypen als nützlich (**RL6**). Darüber hinaus werden textuelle Wächterausdrücke als notwendig empfunden (vgl. Tabelle 8.13). Diese Meinung ist weitgehend unabhängig von der Abstraktionsschicht. Daraus kann abgeleitet werden, dass die Wächterausdrücke für alle Modelle formal sein sollten (**RL6**). Hinsichtlich der Richtlinien für verschachtelte Aktivitäten, **RL7** und **RL8**, kann zu dem Schluss gekommen werden, dass die verschachtelte Aktivität in Form einer grafischen Darstellung intellektuell besser zu verstehen ist. Allerdings wurde die Modellierung verschachtelter Aktivitäten auch als aufwendiger und dementsprechend unbeliebter angesehen, verglichen mit der textuellen AD4S Form. Die verschachtelte Aktivität wird als diejenige mit der höchsten anfänglichen Arbeitsbelastung wahrgenommen. Soweit der Interviewer beurteilen kann, nehmen die Modellierer*innen mit mehr Erfahrung in der Modellierung die verschachtelte Aktivität als weniger aufwendig wahr als die weniger erfahrenen Modellierer*innen. Diese Betrachtung der Verschachtelung berücksichtigt nicht, dass die Modelle von weiteren verschiedenen Interessengruppen mehrmals angesehen werden. Deshalb sollte die detaillierte Funktionalität der verschachtelten Aktivitäten auf die am besten verständliche Weise modelliert werden, wie beispielsweise mit ADs oder SCs. Darüber hinaus muss untersucht werden, wie die Modellierer*innen bei der Erstellung verschachtelter Aktivitäten unterstützt werden können, um den anfänglichen Arbeitsaufwand zu

reduzieren. Nach Ansicht der Befragten erhöht die Richtlinie **RL9** die Klarheit der Modelle durch die Reduzierung redundanter Abläufe (vgl. Abbildung 8.14).

8.3 Gewonnene Erkenntnisse aus der Einführung von MBT in der Industrie

In diesem Abschnitt werden die gewonnenen Erkenntnisse und Herausforderungen im Zeitraum der initialen Projektphase für (automatisiertes) MBT mit SMArDT geschildert. Eine Herausforderung ist das schon bestehende, historisch gewachsene System Fahrzeug (vgl. Theorie und Praxis 4.3). Dieses System mit seinen Komponenten und deren Software wurde historisch weitestgehend lokal und isoliert entwickelt, um in weiteren Schritten von Bottom-up integriert zu werden [Bro06, MPF09]. Diese Entwicklung begünstigte eine flache Mikroarchitektur, um zeitkritische Funktionalitäten sicherzustellen (vgl. Theorie und Praxis 4.7). Die Architektur und die nötigen Prozesse wurden über die Zeit optimiert, aber erschweren die Entwicklung von heutigen und zukünftigen Komponenten-übergreifenden Funktionen. Damit sich Funktionen nicht gegenseitig ungewollt beeinflussen, sind klare Hierarchien innerhalb des Systems erforderlich. Hierfür sind allerdings eine Top-down Entwicklung und eine Schichtenarchitektur nötig (vgl. Theorie und Praxis 4.7). SMArDT verfolgt ebenfalls eine Top-down-Entwicklung, die initial im Projekt eingeführt wurde. Im konkreten Fall im Bereich des Antriebs die Anforderungen der Interessenvertreter*innen an das System in der Ebene A definiert. Anschließend wurden auf Ebene B die funktionalen Konzepte spezifiziert. Die beiden Ebenen A und B wurden anfänglich auf der „grünen Wiese“ ohne Einbeziehung von vorhandenen Lösungen modelliert, um einen Soll-Zustand zu spezifizieren. Allerdings kollidierte weitergehend Ebene B mit Ebene C, da Ebene C die aktuelle Umsetzung abbildete (vgl. Theorie und Praxis 4.3), Ebene B zu detailliert modelliert wurde und technische Lösungen vorwegnahm (vgl. Theorie und Praxis 4.2). Die Auflösung kostete viele Ressourcen des Projekts. Eine Möglichkeit, diese Kollision für bestehende Systeme zu vermeiden ist, zuerst die Schichten und die Ebenen Bottom-up von Ebene D nach Ebene A zu füllen, um in einem zweiten Schritt das System Top-down Iterativ zu verbessern und einer Schichtenarchitektur zu nähern.

Für ein solches Vorgehen hilft es, die vorhandenen Funktionen zu identifizieren und in Funktionsbibliotheken zu gruppieren (vgl. Theorie und Praxis 4.8). Während des Projektes kristallisierte sich heraus, dass die Abstraktion und die Dekomposition intellektuell einfacher nachvollzogen werden können, wenn Funktionen anstatt von Hardware oder Organisationseinheiten als Partitionierung verwendet werden. Sind die Verantwortlichkeiten (Dekomposition) funktional aufgeteilt, kommt es infolgedessen bei der Verzielung der Funktionen nicht zu Missverständnissen. Zusätzlich kann die Abstraktion den Entwicklungsstand einer Funktionalität beziehungsweise den Reifegrad eines Projektes besser abbilden und Verantwortlichkeiten zuordnen. Dies ist insbesondere für die Abgrenzung von Ebene B und Ebene C hilfreich.

Auf Basis der identifizierten Funktionen und Funktionsgruppen können Funktionsnetze wie beispielsweise in [MPF09], gebildet werden. Auf dieses Weise kann mit einem Soft-

ware/ Systems Engineering-Ansatz eine einheitliche (formale) Architektur des Systems erstellt werden. Mit dieser formalen Architektur können Zustände des Systems und Funktionsprioritäten klar dargestellt werden. Infolgedessen wird die Arbeit für weitere Funktionsentwicklungen, die Integration, die Absicherung und die funktionale Sicherheit erheblich vereinfacht (vgl. [WRF+15]). Auch die Tester*innen können anhand der Funktionsarchitektur Abhängigkeiten ihrer Funktionen leichter intellektuell erfassen und folglich effizienter arbeiten. Allerdings wird für den initialen Mehraufwand der Modellierung erfahrungsgemäß viel Überzeugungsarbeit nötig sein, um bestehende Prozesse umzustellen (vgl. Abschnitt 8.1 und [LAK+17]).

Erfahrungen haben gezeigt, dass über die zusammenhängenden Funktionen die Anforderungen von Normen wie der ISO 26262 effizienter erfüllt werden können. Dies ist zum einen aufgrund der Rückverfolgbarkeit, Redundanzfreiheit und der konsistenten Struktur der Anforderungen über Disziplinen und Teilbereiche hinweg möglich. Zum anderen kann über eine automatisierte FMEA effizienter eine Risikobewertung und Auswahl der Testfälle erfolgen.

Die einheitliche Architektur und einheitlichen wohlgeformten und wohldefinierten Modellfunktionen erlauben ebenfalls die Verknüpfung von Elementen wie Schlüsselwörtern. Die Verknüpfung der Spezifikations-Modellinformationen mit den Schlüsselwörtern erlaubte eine unabhängige Entwicklung und gleichzeitig die Integration von Testinformationen, die für den Testbetrieb notwendig sind. Die automatisiert erstellten Testfälle des Testfallgenerators waren mit diesem Schritt „auf einen Klick“ einsatzbereit, auch wenn noch nicht feststand, ob der Testfall manuell oder automatisiert umgesetzt werden sollte.

Erste Erfahrungen mit automatisiert erstellten Testfällen in der Integration zeigen, dass die Testfälle sich insbesondere für eine breite Absicherung eines Systems eignen. Die Breite und die Tiefe beziehen sich auf die Testabdeckung. Die Testabdeckungsbreite umfasst die funktionalen Bereiche wie den Start, die Regelung oder das Ende einer Funktion. Orthogonal zur Testabdeckungsbreite steht die Testabdeckungstiefe, die mit der Anzahl der Szenarien für einen funktionalen Bereich korreliert. Beispiele für die Testabdeckungstiefe sind verschiedene Szenarien, wie ein Funktionsstart getriggert werden kann. Abbildung 8.17 stellt die Tests mit manuellen und automatisierten erstellten Testfällen gegenüber.

Angenommen das Spezifikationsmodell beinhaltet die für die Absicherung benötigten Informationen für eine Funktion, dann lässt sich mit den generierten Testfällen eine besonders breite Absicherung, Testabdeckung beziehungsweise Anforderungsabdeckung realisieren. Eine einfache Testabdeckung der Funktion wäre, die Testfallbereiche Funktionsstart, -regelung und -ende mit jeweils einem Testfall zu testen und somit die Korrektheit einer Funktionsimplementierung zu bestätigen. Allerdings werden andere Testfallbereiche wie Messungen, die für die Zukunft der Funktion oder andere Funktionen von Bedeutung sind, nicht betrachtet.

Manuell erstellte Testfälle profitieren vor allem von der Erfahrung der Tester*innen. Die oft in den (natürlichsprachlichen) Anforderungen knappen Informationen reichen den Tester*innen aus, vereinzelt Testfallbereiche mit vielen denkbaren Szenarien aus der Praxis bis zu einer tiefen Testabdeckungstiefe zu testen, wie zum Beispiel der Funktionsregelung

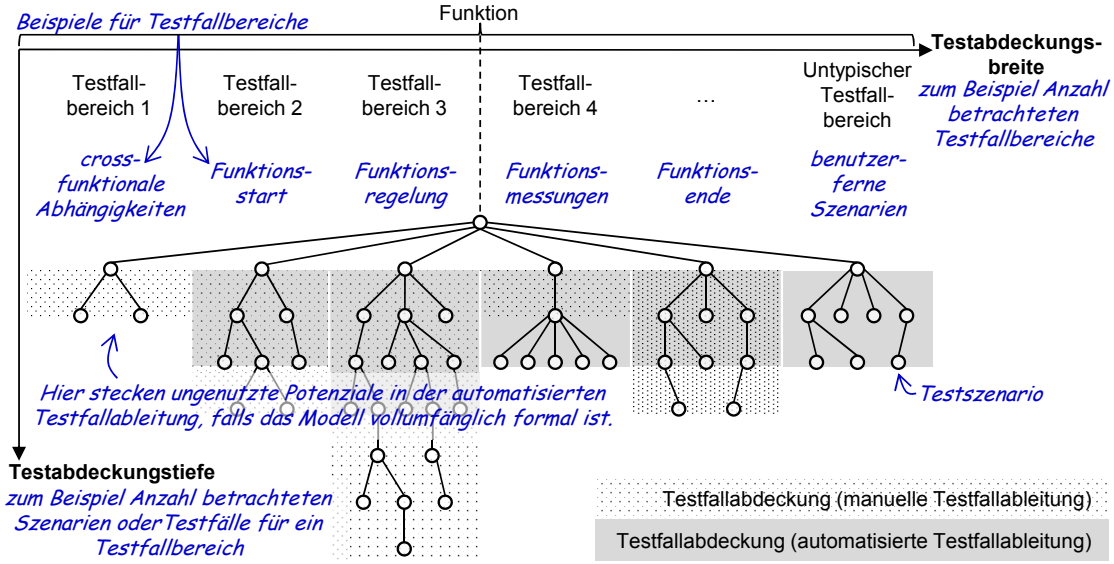


Abbildung 8.17: Manuell und automatisierte Testabdeckungsbreite und -tiefe der Integration eines Systems im Vergleich

in Abbildung 8.17. Auch crossfunktionale Abhängigkeiten zwischen den Funktionen sind häufig aus der Vergangenheit bekannt und werden getestet. Zum Beispiel kann es bei zwei nebeneinander verbauten Komponenten dazu kommen, dass eine Komponente bei Volllast warm wird und eine Funktion auf einer anderen Komponente einen Hitzefehler meldet, obwohl dieser lokale Fehler unbegründet ist. Ein Modell, das sich nur auf eine Funktion beschränkt, kann diese Abhängigkeiten nur bedingt testen. Sind jedoch alle Funktionen formal modelliert und alle relevanten Informationen enthalten, stecken in der Identifikation von crossfunktionalen Abhängigkeiten ungenutzte Potenziale in der automatisierten Testfallableitung, da auch bisher nicht bekannte Abhängigkeiten automatisiert identifiziert werden können.

Für ein Modell mit ausreichenden Informationen kann die automatisierte Testfallableitung sicherstellen, dass kein Testfallbereich vergessen wird. Bisher profilierte sich die automatisierte Testfallableitung sich im Testfallbereich von intellektuell banalen, untypischen, ungewöhnlichen, unvorstellbaren oder benutzerfernen Testfällen, die aber dennoch mögliche Szenarien abbilden. Ein Beispiel ist eine banale Spannungsmessung, die bei Tester*innen keine Beachtung fand. Diese Spannungsmessung deckte einen Fehler in der Spannungsberechnung einer Funktion auf. Zusätzlich können Tester*innen Zeit sparen, um mit ihrer Erfahrung und explorativen Tests einzelne Systembereiche tiefer zu testen (vgl. Theorie und Praxis 5.4). Werden Fehler gefunden oder funktional notwendige Änderungen nötig, können diese Informationen über statische Reviews in das Modell integriert werden und stehen in der nächsten Iteration automatisch zur Verfügung (vgl. Theorie und Praxis 5.4). Zukünftige Entwicklungen profitieren folglich davon. Die Tester*innen können demnach mehr Zeit in die Fehleranalyse, das Validieren des

Systems und dessen Verbesserung investieren, als diese Zeit für die Verifizierung und die Anforderungsabdeckung zu investieren (vgl. Theorie und Praxis 5.5). Da über das gesamte Projekt ein großer Teil der Testfälle bereits manuell modellbasiert erstellt wurden, muss sich die automatisierte modellgetriebene Testfallerstellung allerdings noch etablieren.

Kapitel 9

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine modellbasierte Methode zur Testfallerstellung von CPS in der Industrie vorgestellt. Diese Methode basiert auf SMArDT, einem Ansatz aus dem Systems und Software Engineering, und wurde mit Bezug zum MBT weiterentwickelt. Neben der Modellierung wurde zudem die technische Umsetzung in Form eines Testfallgenerators vorgestellt. Dieser Testfallgenerator erstellt auf der Basis von SysML-Verhaltensdiagrammen Testfälle, wenn diese Diagramme einer vorgegebenen DSL-Grammatik entsprechen. Die DSL und deren Kontextbedingungen sind an die abstrakte Funktionsmodellierung der vorgestellten SMArDT-Methode angepasst und ermöglichen eine Testfallerstellung bei gleichzeitigem größtmöglichem Freiraum der Entwickler*innen. Für die Kompatibilität der manuell erstellten Modelle mit der maschinenlesbaren DSL dienen Richtlinien, die ebenfalls in dieser Arbeit vorgestellt und evaluiert werden. Aus diesen maschinenlesbaren Modellen werden automatisch Testfälle erstellt. Damit diese Testfälle automatisch in der Testumgebung ausgeführt werden können, verwenden die Testfälle Schlüsselwörter, die im Modell hinterlegt werden.

Die methodischen und technologischen Inhalte dieser Arbeit wurden in einem Automobilkonzern, der BMW Group, angewendet. Erfahrungen, auftretende Herausforderungen und gewonnene Erkenntnisse aus der angewendeten Forschung wurden wissenschaftlich aufgearbeitet und an geeigneten Stellen erwähnt.

Kapitel 1 reißt das Feld, auf dem sich die Arbeit bewegt, an. Zu diesem Zweck wird die Bedeutung des Themas für komplexe CPS hervorgehoben.

Kapitel 2 und Kapitel 3 führen die Grundlagen ein, um ein einheitliches Verständnis des in dieser Arbeit betrachteten Umfangs herbeizuführen. In Kapitel 2 werden die Grundlagen für die modellbasierte Entwicklung in der Automobilindustrie, der SysML, dem MontiCore-Framework und von SMArDT vorgestellt. Darauf aufbauend werden in Kapitel 3 die Grundlagen des Testfallerstellungsprozesses, des manuellen und des modellbasierten Testens in der Automobilindustrie erläutert.

Kapitel 4 stellt einen Ansatz vor, wie etablierte Erkenntnisse aus dem Software Engineering in die Praxis von traditionell maschinenbau-geprägten Industrien fließen können, um effektiv, effizient und nachhaltig die Qualität der Produkte zu gewährleisten. Zusätzlich werden die Herausforderungen und mögliche Strategien für eine durchgängige Verwendung von Modellen in einem laufenden Tagesgeschäft beschrieben.

Der methodische Ansatz, eine Weiterentwicklung von SMArDT wendet etablierte Prinzipien des modellgetriebenen System und Software Engineering an. Zu diesen Prinzipien gehören Abstraktionsebenen, hierarchische Dekomposition, Teile-und-herrsche-Ansatz, Verbergen nicht relevanter Informationen, modulare Entwicklung und schrittweise Verfeinerung des Designs. Diese Arbeit fokusiert sich auf die automatisierte Absicherung. Infolgedessen ist die Aufmerksamkeit auf das Funktions- und Systemverständnis, die maschinenlesbare Spezifikation und die automatisierte Testfallerstellung gerichtet.

Bei der Integration einer modellbasierten Methode in die Praxis konnten einige Herausforderungen identifiziert werden, die vor allem mit den bereits etablierten Strukturen und Systemen zusammenhängen. Ein einfacher Neuanfang auf der Basis von system- und softwaretechnischen Prinzipien oder eine einfache Adaption ist nicht ohne Komplikationen und hohe Kosten möglich, das heißt, ohne das Tagesgeschäft zu unterbrechen und laufende Umsätze zu gefährden. Aus diesem Grund müssen erst der Status quo identifiziert und anschließend die Funktionen zugewiesen werden. Erst danach können über ein iterativen Vorgehen die Systeme und Strukturen softwaretechnisch optimiert werden. Die Funktionszuweisung dient der Transformation einer hardwareorientierten Entwicklung hin zu einer funktionsorientierten Entwicklung. Mit der Ablösung der Hardwareorientierung lassen sich technische Verbesserungen umsetzen, ohne die Grundfunktionalitäten des Systems zu verändern. Zusätzlich können die wachsenden (Software-) Funktionen des Systems unabhängig von der Hardware stetig verbessert werden. Die Auswirkungen dieser Transformation werden neben dem System auch die organisatorische Struktur des Unternehmens beeinflussen. In dieser Arbeit wird empfohlen, die organisatorische Transformation ebenfalls iterativ anzupassen. Neben dieser Änderung der Strukturen stellen die Nutzung der etablierten Tools, die geeignete Wahl der Abstraktionsgrade und die Entwicklungsgewohnheiten der verschiedenen Ingenieursdisziplinen eine Herausforderung dar.

Kapitel 5 stellt eine Statusanalyse der aktuellen Testfallerstellung in der Automobilindustrie und Kriterien für eine manuelle und automatisierte Testfallerstellung vor, die Spezifikationsmodelle von SMArDT und die Testfälle erfüllen müssen.

Zentrale Erkenntnisse der Statusanalyse mit dem Fokus auf Integrationstests sind, dass

- die Testspezialist*innen häufig eine Vorbildung aus der Elektrotechnik, Maschinenbau oder Mechatronik besitzen,
- die Grundlage für Testfälle größtenteils auf der Basis von natürlichsprachlichen textuellen Anforderungen oder existierenden Testfällen und Erfahrungen erstellt werden,
- der Testfokus vor allem auf dem Fahrzeug oder bereits zum Teil integrierten Subsystemen liegt,
- die Testspezialist*innen Verbesserungsmöglichkeiten bei der Testabdeckung sehen,
- das Thema modellbasiertes Testen mit positiven Erwartungen verknüpft wird und
- es erwartet wird, dass sich der hohe initiale Aufwand für die modellbasierte Testfallerstellung im Nachhinein lohnen wird.

Als Kriterien für die Testbarkeit von abstrakten Anforderungsmodellen mit SMArDT in der Praxis wurden die Spezifikationseigenschaften aktuell, einheitlich, testbar, technisch realisierbar, (für die Testzecke) vollständig und teststufenorientiert identifiziert. Zusätzlich sollten die Abstraktion und Dekomposition intellektuell und maschinenlesbar sein. Die Maschinenlesbarkeit und Einheitlichkeit setzt eine wohlgeformte, wohldefinierte und gemeinsame Sprache der einzelnen Spezifikationsmodelle des Gesamtmodells voraus. Auf der Basis dieser sprachkonformen Spezifikationsmodelle sind die erstellten Testfälle für die generische automatisierte Ausführung auf mehreren Testumgebungen ausführbar.

Als Herausforderung wurde insbesondere der Übergang von mehreren multiplen textuellen Anforderungen hin zu einem einheitlichen modellbasierten grafischen Modell identifiziert. Darüber hinaus mangelt es den Testfällen teilweise an Aussagekraft und die Integration von erfahrungsorientierten und manuellen Testfällen gestaltet sich als schwierig.

Kapitel 6 stellt die Monticore-Grammatik AD4S vor, die den Herausforderungen aus SMArDT begegnet und die Kriterien aus Kapitel 5 für eine automatisierte Testfallerstellung erfüllt. Als Problemstellung wurden Mehrzweck Modelle für ein interdisziplinäres Umfeld, die daraus resultierende interdisziplinäre Modellierungsunterstützung und die in den Ingenieurdisziplinen nicht vertrauten hohen Abstraktionsebenen identifiziert.

Aus diesem Grund wurde AD4S in einer Weise entwickelt, die den modellierenden Domänenexpert*innen größtmögliche Ausdrucksfreiheit gewährt und gleichzeitig sichergestellt, dass die Kriterien für die (automatisierte) Testfallerstellung erfüllt werden. Zu diesem Zweck wurden Modellierungsrichtlinien aufgesetzt, die eine grafische Modellierung der Domänenexpert*innen in eine einheitliche Struktur bringt.

Kapitel 7 stellt die technische Umsetzung, den Testfallgenerator vor. Dieser Testfallgenerator erstellt aus AD4S-konformen Modellen automatisch Testfälle, für das automatisierte testfallumgebungsunabhängige Testen geeignet sind. Für die AD4S-konformen Modelle dienen wohlgeformte und wohldefinierte SysML-Aktivitätsdiagramme, Zustandsdiagramme und Kombinationen dieser Diagrammtypen. Diese Modelle werden analysiert, transformiert und sukzessive für die Testfallerstellung aufbereitet. Die einzelnen Schritte werden vorgestellt und Herausforderungen in der Praxis erörtert. Zu diesen Herausforderungen gehören unter anderem die Berücksichtigung etablierter Ressourcen und die Integration von vorhandenen Erfahrungen und Wissen der Testfallersteller*innen.

Kapitel 8 stellt die gewonnenen Erkenntnisse aus der Praxis vor und evaluiert die vorgestellte Methode und den Testfallgenerator.

Eine gewonnene Erkenntnis ist, dass sich der Charakter der Diagramme über die Projektlaufzeit verändert hat. Während am Anfang vorwiegend Regelungsdiagramme mit parallelen Aktionen entworfen wurde, wurden am Ende Ablaufdiagramme mit sequentiellem Ablauf entwickelt. Neben dieser Erkenntnis wurde außerdem festgestellt, dass die Qualität der Produkte nachhaltig gesteigert werden kann, wenn die Tester*innen in der Spezifikationsphase miteinbezogen werden. Dadurch fließen ihre Erfahrungen in

die Entwicklung mit ein. Diese Entwicklungsprozesse wurden über die Projektlaufzeit fest in den Prozessen verankert. Setzen sich Tester*innen mit der Spezifikation auseinander, können also schon früh Mehrdeutigkeiten und Missverständnisse ausgeräumt werden. Zusätzlich wird schon frühzeitig ein nachhaltig dokumentiertes Funktions- und Systemverständnis der Tester*innen geschaffen. Dieses Verständnis kann in zukünftige Absicherungsstrategien fließen und infolgedessen die Effektivität steigern.

Aus der Evaluation der modellbasierten Testfälle geht hervor, dass die Qualität der Testfallabdeckung für die manuelle und insbesondere für die automatisierte Testfallerstellung stark mit der Qualität der Modelle korreliert. Für die Gewährleistung der kontextuellen Qualität der Modelle dienen Reviews seitens verschiedener Parteien, wie beispielsweise der Absicherung oder der funktionalen Sicherheit. Für die Sicherstellung der Wohlgeformtheit und Wohldefiniertheit dienen zusätzlich noch Modellierungsrichtlinien. Diese Modellierungsrichtlinien wurden in einer Studie evaluiert. Die Studie wurde mit modellierenden Domänenexpert*innen durchgeführt und ergab, dass diese die Modellierungsrichtlinien grundsätzlich befürworten. Allerdings sollten der zusätzliche Arbeitsaufwand aufgrund dieser Richtlinien über Toolverbesserungen kompensiert werden. Ungeachtet der Modellierungsrichtlinien, der Wohlgeformtheit und der Wohldefiniertheit, muss der Inhalt des Modells korrekt und sinnvoll sein. Reviews gewährleisten dabei den korrekten intellektuellen Inhalt der Modelle.

Übergreifend konnten Erkenntnisse aus der Einführung von MBT in der Automobilindustrie für ein erfolgversprechendes Vorgehen gewonnen werden. Für eine Transformation einer traditionellen Bottom-up Entwicklung hin zu einer modellbasierten Top-down Entwicklung ist es sinnvoll, zuerst vorhandene Funktionsgruppen zu identifizieren, diese iterativ zu abstrahieren und anschließend zu optimieren. Daraufhin kann ein einheitliches, wohlgeformtes und wohldefiniertes Modell für die Architekturspezifikation entwickelt werden. Auf der Basis des einheitlichen maschinenlesbaren Modells können automatisierte Prozesse etabliert werden, wie beispielsweise eine Testfallerstellung oder eine FMEA. Erfahrungen haben gezeigt, dass eine modellbasierte und automatisierte Testfallerstellung die Tester*innen sinnvoll unterstützen kann, wenn diese mit der Absicherung von CPS beschäftigt sind. Allerdings wird die manuelle explorative Tätigkeit nicht ersetzt werden können, da ein fehlerarmes CPS nur gemeinsam mit anforderungserfüllenden und explorativen Tests erreicht werden kann.

In dieser Arbeit wurde gezeigt, dass etablierte Erkenntnisse aus der Informatik wie die modellgetriebene Entwicklung in die Praxis von traditionellen maschinenbau-geprägten Industrien fließen können. Zusätzlich wurde verdeutlicht, dass die Qualität der Produkte nachhaltig, effektiv und effizient gewährleistet werden kann. Herausforderungen aus der Einführung in das Tagesgeschäft eines Automobilkonzerns wurden identifiziert und für die Forschung reflektiert.

Die vorgestellte modellgetriebene Methode SMArDT und der AD4S-konforme Testfallgenerator bilden die Grundlage zur Untersuchung weiterer Fragestellungen im Bereich des modellbasierten Systems und Software Engineering. Im Folgenden werde einige mögliche Anknüpfungspunkte vorgestellt.

Agile Methoden und SMArDT

Wie in Kapitel 1 erwähnt, verbesserten in den letzten Jahren neben der durchgängigen Verwendung von Modellen auch die agilen Methoden die Qualität und Effizienz in der Softwareentwicklung. SMArDT ermöglicht mit seinen Dekompositionsschichten, Funktionsgruppen und klaren Schnittstellen agile Entwicklungen der einzelnen Teams und Projekte. Die erstellten Entwicklungsartefakte müssen in einem übergreifenden Prozess abgestimmt und verknüpft werden. Mögliche Methoden für die Verknüpfung sind LeSS Huge [LV10] und SAFe [KL19]. In LeSS Huge laufen mehrere Scrum Teams [SB02] parallel und werden über einen übergreifenden *Product Owner* verknüpft. SAFe bildet einen Rahmen, der auf dem jeweiligen Unternehmen und Gegebenheiten angepasst wird. Erste Praxiserfahrungen mit SAFe in Verbindung mit SMArDT zeigen ein passendes Ineinandergreifen des SAFe-Rahmens und der SMArDT-Methode. Insbesondere die modellbasierte automatisierte Erstellung von Artefakten ist mit dem iterativen und schnellen Charakter der agilen Entwicklung vereinbar [HR17]. So verkürzen die agilen Arbeitsweisen mit SMArDT die Zeit, auf Änderungen präzise zu reagieren. Allerdings sollte der Mehrwert von agilen Methoden in Verbindung mit modellbasierten Methoden wie SMArDT in der Praxis noch analysiert werden [KRR18].

Testabdeckungsmethoden

Wie in Theorie und Praxis 7.2 erwähnt, sind gängige Testabdeckungsmethoden aus der Informatik für die Testabdeckung von Integrationstest in der Automobilindustrie nicht effizient genug. Insbesondere der Umstand, dass Hardware-Testumgebungen (HIL und VIL-Testumgebungen) knapp, kostspielig und nicht immer vollständig sind, erfordert eine domänenspezifische Testabdeckungsmethode, die diesen Umstand berücksichtigt. Diese domänenspezifische Testabdeckungsmethode sollte Gewichtungen von Signalen, Signalausschlüsse und deren Kombinationen ermöglichen.

Effizientes Testen

Um die Testressourcen und Hardware der verschiedenen Testumgebungen effizient zu nutzen, ist eine effiziente Zuteilung der erstellten Testfälle unumgänglich. Der in dieser Arbeit vertretene Ansatz einer zentralen, eindeutigen und verlässlichen Datenbasis erfordert, dass Informationen für eine Produktlinien-Testmethoden wie in [EJK+19] im Modell hinterlegt werden. Zu untersuchen ist, wie diese Informationen in SMArDT integriert werden und wie sich Produktlinien-Testmethoden mit domänenspezifische Testabdeckungsmethoden ergänzen können.

Das effiziente Testen ist essenziell, um die Produkte der Automobilindustrie gründlich und kostengünstig für den Einsatz beim Endkunden zu überprüfen. Die zunehmende Komplexität der Produkte macht sich besonders bei diesem letzten elementaren Schritt vor der Markteinführung bemerkbar und stellt die beteiligten Ingenieur*innen aller Disziplinen vor große Herausforderungen. Um diese Komplexität nicht nur beim Testen, sondern auch

schon in der Entwicklung beherrschbar zumachen, werden modellbasierte Methoden auf Basis von Systems Engineering in Zukunft zwingend erforderlich. Vor allem bietet eine modellgetriebene Entwicklung große Potenziale, um die einzelnen Entwicklungsschritte zu unterstützten und effizient durchzuführen. Nur wenn es gelingt, diese Methoden in der Entwicklung zu etablieren, dann können auch in Zukunft hochwertige, multifunktionale und digitale Systeme entwickelt werden, die auch die immer neuen Kundenanforderungen an Innovation, Zuverlässigkeit und Nachhaltigkeit erfüllen.

Literaturverzeichnis

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[ADS18] Deniz Akdur, Onur Demirörs, and Bilge Say. Towards Modeling Patterns for Embedded Software Industry: Feedback from the Field. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 132–136, 2018.

[AFST14] Domenico Amalfitano, Anna Rita Fasolino, Stefano Scala, and Porfirio Tramontana. Towards Automatic Model-in-the-loop Testing of Electronic Vehicle Information Centers. In *Proceedings of the 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering*, WISE '14, pages 9–12, New York, NY, USA, 2014. ACM.

[AGGS14] Allon Adir, Alex Goryachev, Lev Greenberg, and Tamer Salman. Using a High-Level Test Generation Expert System for Testing In-Car Networks. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 134:1–134:6, New York, NY, USA, 2014. ACM.

[AHL18] Syed S. Arefin, Hadi Hemmati, and Howard W. Loewen. Evaluating Specification-level MC/DC Criterion in Model-based Testing of Safety Critical Systems. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 256–265, New York, NY, USA, 2018. ACM.

[AHLL+17] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *2nd IEEE/ACM International Workshop on Variability and Complexity in Software Design,*

VACE@ICSE 2017, Buenos Aires, Argentina, May 27, 2017, pages 34–40. IEEE, 2017.

[AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AKKR21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM SIGPLAN, October 2021.

[AMN+20] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[ANT14] Bader Albdaiwi, René Noack, and Bernhard Thalheim. Pattern-Based Conceptual Data Modelling. *Frontiers in Artificial Intelligence and Applications*, 272:1–20, 2014.

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018.

[AUT19] AUTOSAR development cooperation. www.autosar.org, Online; abgerufen am 03.05.2019.

[AWS14] Harald Altinger, Franz Wotawa, and Markus Schurius. Testing methods used in the automotive industry: results from a survey. In Christof Budnik, Gabriella Carrozza, David Faragó, Baris Güldali, Barath Kumar, Vittorio Manetti, Roberto Pietrantuono, and Stephan Weißleder, editors, *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pages 1–6, 2014.

[AY10] Roy Awedikian and Bernard Yannou. Design of a validation test process of an automotive software. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 4(4):259–268, 2010.

[Bal09] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg, 3. edition, 2009.

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics*, 12, 2018.

[BBD+21a] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021.

[BBD+21b] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021.

[BBH+14] Peter Braun, Manfred Broy, Frank Houdek, Matthias Kirchmayr, Mark Müller, Birgit Penzenstadler, Klaus Pohl, and Thorsten Weyer. Guiding requirements engineering for software-intensive embedded systems in the automotive industry: The REMsES approach. *Computer Science - Research and Development*, 29(1):21–43, 2014.

[BBK+17] Jennifer Brings, Julian Bellendorf, Kevin Keller, Markus Kempe, Noyan Kurt, Alexander Palm, and Marian Daun. Applying the SPES Modeling Framework: A Case Study from the Automotive Domain. In *Proceedings of REFSQ-2017 Workshops, Doctoral Symposium, Research Method Track, and Poster Track co-located with the 22nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2017) (2017)*, 2017.

[BBK+21] Wolfgang Böhm, Manfred Broy, Cornel Klein, Klaus Pohl, Bernhard Rumpe, and Sebastian Schröck, editors. *Model-Based Engineering of Collaborative Embedded Systems*. Springer, January 2021.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autono-

mous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020.

[BBRS03] Michael von der Beeck, Peter Braun, Martin Rappl, and Christian Schröder. Automotive UML. In Luciano Lavagno, Grant Martin, and Bran Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*, pages 271–299. Springer US, Boston, MA, 2003.

[BCG+11] Manfred Broy, Samarjit Chakraborty, Dip Goswami, S. Ramesh, M. Satpathy, Stefan Resmerita, and Wolfgang Pree. Cross-layer Analysis, Testing and Verification of Automotive Control Software. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 263–272, New York, NY, USA, 2011. ACM.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, Maria Victoria Cengarle, Hans Grönninger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*. John Wiley & Sons, Ltd, Hoboken, N.J, 2009.

[BCGR09c] Manfred Broy, Maria Victoria Cengarle, Hans Grönninger, and Bernhard Rumpe. Definition of the System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*. John Wiley & Sons, Ltd, Hoboken, N.J, 2009.

[BCL+21] Antonio Bucchiarone, Federico Ciccozzi, Leen Lambers, Alfonso Pierantonio, Matthias Tichy, Massimo Tisi, Andreas Wortmann, and Vadim Zaytsev. What Is the Future of Modeling? *IEEE Softw*, 38(2):119–127, 2021.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BD95] Adolf-Peter Bröhl and Wolfgang Dröschel, editors. *Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden.* Software - Anwendungsentwicklung - Informationssysteme. Oldenbourg, München, 2. edition, 1995.

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wort-

mann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *ACM Trans. Internet Things*, 3:1–32, February 2022.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021.

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology*, 19(3):3:1–16, October 2020. Special Issue dedicated to Martin Gogolla on his 65th Birthday.

[BEK17] Ghada Bahig and Amr El-Kadi. Formal Verification of Automotive Design in Compliance With ISO 26262 Design Verification Guidelines. *IEEE Access*, 5:4505–4516, 2017.

[BEK+18] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019.

[Ben10] Sebastian Benz. *Generating Tests for Feature Interaction*. Dissertation, Technische Universität München, München, 2010.

[Ber06] Harvey Russell Bernard. *Research methods in anthropology: Qualitative and quantitative approaches*. Rowman & Littlefield, Lanham, 4. edition, 2006.

[Ber07] Antonia Bertolino. Software Testing Research: Achievements, Challenges,

Dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103, 2007.

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.

[BF14] Pierre Bourque and Richard E. Fairley. *Guide to the software engineering body of knowledge: SWEBOK(R)*. IEEE Computer Society, Los Alamitos, Calif., 3. edition, 2014.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Computer Standards & Interfaces*, 19(7):335–345, November 1998.

[BGH+98b] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst,

and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018.

[BHRW21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BK08a] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, Cambridge, Mass, 2008.

[BK08b] Eckart Bringmann and Andreas Krämer, editors. *Model-Based Testing of Automotive Systems: 2008 1st International Conference on Software Testing, Verification, and Validation*, 2008.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case*. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKRW17b] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software*, 149:437–461, March 2019.

[BLLS14] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. Multi-objective Test Suite Optimization for Incremental Product Family Testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 303–312. IEEE Computer Society, 2014.

[BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022.

[BMW19] BMW Group. Höchste Exklusivität in einzigartiger Vielfalt: Die BMW Modelloffensive im Luxussegment: www.press.bmwgroup.com/deutschland/article/detail/T0299948DE/hoechste-exklusivitaet-in-einzigartiger-vielfalt:-die-bmw-modelloffensive-im-luxussegment?language=de, Online; abgerufen am 16.08.2019.

[BNHZ18] Uwe Beyer, Kilian Nickel, Felix Hasenbeck, and Alexander Zimmermann. *Mensch und System: Ideen zu humanzentrischen Systemmodellen*. Springer Gabler, Wiesbaden, 2018.

[Boe79] Barry W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. In P. A. Samet, editor, *Euro IFIP 79*, pages 711–719. North Holland, 1979.

[BPRW20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[Bro06] Manfred Broy. Challenges in Automotive Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015.

[BSH07] Frank Buschmann, Douglas C. Schmidt, and Kevlin Henney. *On patterns and pattern languages*, Pattern-oriented software architecture v. 5. Wiley, Chichester England, 2007.

[BWC12] Marco Brambilla, Manuel Wimmer, and Jordi Cabot. *Model-driven software engineering in practice*, Synthesis lectures on software engineering 1. Morgan & Claypool, San Rafael, Calif., 2012.

[CB16] Lyra J. Colfer and Carliss Y. Baldwin. The mirroring hypothesis: theory, evidence, and exceptions. *Industrial and Corporate Change*, 25(5):709–738, 2016.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[CH05] James O. Coplien and Neil B. Harrison. *Organizational patterns of agile software development*. Pearson Prentice Hall, Upper Saddle River, NJ, 2005.

[CKS11] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum. *CMMI for development: Guidelines for process integration and product improvement*. The SEI series in software engineering. Addison-Wesley, Upper Saddle River, NJ, 3. edition, 2011.

[CMK08] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. Coverage-driven Automatic Test Generation for Uml Activity Diagrams. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 139–142, New York, NY, USA, 2008. ACM.

[CMM19] CMMI Institute. Capability Maturity Model Integration: Introducing CMMI V2.0: www.cmmiinstitute.com/cmmi, Online; abgerufen am 24.07.2019.

[CMRP18] Mohamad Chamas, Matthias Markthaler, Bernhard Rumpe, and Kristin Paetzold. Bewertungsmethodik zur Verbesserung der Testfallerstellung auf Basis von SysML. In *DFX 2018: 29th Symposium on Design for X*, pages 143–154, München, 2018.

[Con68] Melvin Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[DBP15] Markus Dahlweid, Jörg Brauer, and Jan Peleska. Model-Based Testing: Automatic Generation of Test Cases, Test Data and Test Procedures from SysML Models. In SAE International, editor, *SAE AeroTech Congress & Exhibition; 2015; Seattle, Wash.* Warrendale, Pa., 2015.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 0(49):301–328, 2019.

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative

Aspects in Existing Applications. *Journal of Object Technology*, 20:1–24, November 2021.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022.

[Dij72] Edsger Wybe Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, 1972.

[DIN08] DIN Deutsches Institut für Normung. DIN 55350-11:2008-05 Begriffe zum Qualitätsmanagement: Teil 11: Ergänzung zu DIN EN ISO 9000:2005, 2008.

[DIN11] DIN Deutsches Institut für Normung. DIN 61508: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme, 2011.

[DIN15] DIN Deutsches Institut für Normung. DIN EN ISO 9000:2015-11 Qualitätsmanagementsysteme – Grundlagen und Begriffe: Deutsche und Englische Fassung EN ISO 9000:2015, 2015.

[DIN17] DIN Deutsches Institut für Normung. DIN EN ISO/IEC 25063:2017-09 System- und Software-Engineering – Qualitätskriterien und Bewertung von Systemen und Softwareprodukten (SQuaRE): Allgemeines Industrieformat (CIF) zur Gebrauchstauglichkeit, 2017.

[DIN18a] DIN Deutsches Institut für Normung. DIN EN 61360-1:2018-07 Genormte Datenelementtypen mit Klassifikationsschema für elektrische Betriebsmittel: Teil 1: Definitionen – Regeln und Methoden, 2018.

[DIN18b] DIN Deutsches Institut für Normung. DIN EN ISO 9004:2018-08 Qualitätsmanagement - Qualität einer Organisation - Anleitung zum Erreichen nachhaltigen Erfolgs: Deutsche und Englische Fassung EN ISO 9004:2018, 2018.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[dMB08] Leonardo de Moura and Nikolaj Bjrner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International*

Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, pages 79–89, New York, NY, USA, 2020. Association for Computing Machinery.

[dSP20] dSPACE GmbH. dSpace: www.dspace.com/de/gmb/home/products/sw/datenmanagement/synect/synect_tm.cfm, Online; abgerufen am 07.06.2020.

[DSVT07] Arilo Claudio Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM.

[Dud20] Duden, Online; abgerufen am 28.02.2020.

[EAG+05] Ulrik Eklund, Örjan Askerdal, Johan Granholm, Anders Alminger, and Jakob Axelsson. Experience of introducing reference architectures in the development of automotive electronic systems. In *Proceedings of the Second International Workshop on Software Engineering for Automotive Systems*, SEAS '05, pages 1–6, New York, NY, USA, 2005. ACM.

[EAS13] EAST-ADL Association. EAST-ADL Domain Model Specification version V2.1.12, 2013.

[EF17] Christof Ebert and John Favaro. Automotive Software. *IEEE Software*, 34(3):33–39, 2017.

[EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EFLR99b] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[Est20] Jeff A. Estefan. Survey of Model-Based Systems Engineering (MBSE) Methodologies: www.omgsysml.org/MBSE_Methodology_Survey_RevB.pdf, Online; abgerufen am 30.01.2020.

[Exi20] Exida. Functional Safety Services, IACS Cybersecurity, Alarm Management, IEC 61508 Certification: www.exida.com, Online; abgerufen am 12.09.2020.

[FBBR03] Ulrich Freund, Michael von der Beeck, Peter Braun, and Martin Rappl. Architecture Centric Modeling of Automotive Control Software. In *SAE Technical Paper Series*, SAE Technical Paper Series. SAE International400 Commonwealth Drive, Warrendale, PA, United States, 2003.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[Fev20] Fev. Europe GmbH | Deutschland: www.fev.com, Online; abgerufen am 12.09.2020.

[Fey10] Dietmar Fey. *Grid-Computing: Eine Basistechnologie für Computational Science*. eXamen.press. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2010.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[FLP+11a] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für

Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FNDR98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998.

[For13] Charles Forsythe. *Instant FreeMarker starter: Structure your enterprise-class projects with FreeMarker!* Packt Pub, Birmingham, UK, 2013.

[FP08] Sebastian Fixson and Jin-Kyu Park. *The Power of Integrality: Linkages Between Product Architecture, Innovation, and Industry Structure.* Working Paper, MIT Sloan School of Management, Cambridge, 2008.

[FP11] Martin Fowler and Rebecca Parsons. *Domain-specific languages.* A Martin Fowler signature book. Addison-Wesley, Upper Saddle River, NJ and Boston and Indianapolis and San Francisco and New York and Toronto and Montreal and London and Munich and Paris and Madrid and Sydney and Tokyo and Singapore and Mexico City, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.

[Fre19] Freemarker. www.freemarker.org, Online; abgerufen am 05.06.2019.

[FS08] Hannsjörg Freund and Kai Sundmacher. Towards a methodology for the systematic analysis and design of efficient chemical processes: Part 1. From unit operations to elementary process functions. *Chemical Engineering and Processing: Process Intensification*, 47(12):2051–2060, 2008.

[Gar20] Gartner. Definition of Digitalization - Gartner Information Technology Glossary: https://www.gartner.com/en/information-technology/glossary/digitalization, 2020.

[GBF20] Vahid Garousi, Sara Bauer, and Michael Felderer. NLP-assisted software testing: A systematic mapping of the literature. *Information and Software Technology*, 126:106321, 2020.

[GC09] Kamran Ghani and John A. Clark. Automatic Test Data Generation for Multiple Condition and MCDC Coverage. In *2009 Fourth International Conference on Software Engineering Advances*, pages 152–157, 2009.

[GD18] Ralf Hartmut Güting and Stefan Dieker. *Datenstrukturen und Algorithmen.* Springer Vieweg, Wiesbaden, 4. edition, 2018.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. *4th European Congress ERTS - Embedded Real Time Software, Toulouse*, 2008.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GJM+10] Baris Güldali, Stefan Jungmayr, Michael Mlynarski, Stefan Neumann, and Mario Winter. Starthilfe für modellbasiertes Testen: Entscheidungsunterstützung für Projekt- und Testmanager. *Objektspektrum*, 2010(03):63–69, 2010.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, chapter 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GR15] Jeff Gray and Bernhard Rumpe. Models for digitalization. *Software & Systems Modeling*, 14(4):1319–1320, 2015.

[Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte.* Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[Grö10] Hans Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten: Zugl.: Aachen, Techn. Hochsch., Diss., 2010*, Aachener Informatik-Berichte, Software-Engineering 4. Shaker, Aachen, 2010.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009, LNCS 5522*, Lisbon, Portugal, 2009.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[GRS16] Paolo Giusto, S. Ramesh, and M. Sudhakaran. Modeling and Analysis of Automotive Systems: Current Approaches and Future Trends. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, pages 704–710. SCITEPRESS - Science and and Technology Publications, 2016.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software factories: Assembling applications with patterns, models, frameworks, and tools.* Wiley, Indianapolis, IN, 2004.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.* Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[HBP+17] Alireza Haghighatkhah, Ahmad Banijamali, Olli-Pekka Pakanen, Markku Oivo, and Pasi Kuvaja. Automotive Software Engineering: A Systematic Mapping Study. *Journal of Systems and Software*, 128, 2017.

[HC90] Rebecca Henderson and Kim Clark. Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms. *Administrative science quarterly*, 35(3), 1990.

[Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software*

Security & Privacy. Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015.

[Hit07] Derek K. Hitchins. *Systems engineering: A 21st century systems methodology*. Wiley series in systems engineering and management. John Wiley, Chichester, West Sussex, England and Hoboken, NJ, 2007.

[Hit21] Derek K. Hitchins. Getting to Grips with Complexity or... A Theory of Everything Else... https://systems.hitchins.net/profs-stuff/profs-books/getting-to-grip-with-comple/, Online; abgerufen am 28.02.2021.

[HJ08] Richard Hopkins and Kevin Jenkins. *Eating the IT elephant: Moving from greenfield development to brownfield*. Safari Books Online. IBM Press/Pearson plc, Upper Saddle River, N.J, 2008.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.

[HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020.

[HKK+18] Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Mike Lorang, Bernhard Rumpe, Albi Sema, Georg Strobl, and Michael von Wenckstern. Model-Based Development of Self-Adaptive Autonomous Vehicles using the SMARDT Methodology. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 163 – 178. SciTePress, January 2018.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, pages 175–182. Springer, December 2016.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann.

Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMR96] H. P. E. Vranken, M. F. Witteman, and R. C. Van Wuijtswinkel. Design for testability in hardware software systems. *IEEE Design Test of Computers*, 13(3):79–86, 1996.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, Andreas Wortmann, Alfonso Pierantonio, Mark van den Brand, and Benoit Combemale. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, 2019.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[Höl18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 language workbench: Technical Report - Aachen, December 12, 2017*, Aachener Informatik-Berichte, Software-Engineering Band 32. Shaker Verlag, Aachen, 2017.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter-Systeme VI*, Informatik-Bericht 2010-01, pages 13 – 22. fortiss GmbH, Germany, 2010.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386–405, 2018.

[HSRW98] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*, pages 12–21, 1998.

[HW17] Gerhard Hab and Reinhard Wagner. *Projektmanagement in der Automobilindustrie: Effizientes Management von Fahrzeugprojekten entlang der Wertschöpfungskette*. Springer Gabler, Wiesbaden, 5. edition, 2017.

[IBM19] IBM Engineering Requirements Management DOORS Family - Überblick - Deutschland. www.ibm.com/de-de/marketplace/requirements-management, Online; abgerufen am 07.08.2019.

[Ipe11] Mesut Ipek. *Eine Testfallspezifikationssprache für das funktionsorientierte Testen von reaktiven eingebetteten Systemen im Automobilen Bereich*. Dissertation, Technische Universität Kaiserslautern, Kaiserslautern, 2011.

[ISO96] ISO Internationale Organisation für Normung. ISO/IEC 14977:1996 Information technology - Syntactic metalanguage - Extended BNF, 1996.

[ISO11] ISO Internationale Organisation für Normung. ISO/IEC 25010:2011-03 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models: System and software quality, 2011.

[ISO12] ISO Internationale Organisation für Normung. ISO/IEC 15504-5:2012-02 Informationstechnik - Prozessbewertung - Teil 5: Beispiel eines Software-Lebenszyklus-Prozessassessmentmodells, 2012.

[ISO13a] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 29119-1:2013-09 Software and systems engineering — Software testing: Part 1: Concepts and definitions, 2013.

[ISO13b] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 29119-2:2013-09 Software and systems engineering — Software testing: Part 2: Test processes, 2013.

[ISO13c] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 29119-3:2013-09 Software and systems engineering — Software testing: Part 3: Test documentation, 2013.

[ISO15a] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 15288-1:2015-05 Systems and software engineering: System life cycle processes, 2015.

[ISO15b] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 29119-4:2015-12 Software and systems engineering — Software testing: Part 4: Test techniques, 2015.

[ISO16] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 29119-5:2016-11 Software and systems engineering — Software testing: Part 5: Keyword-Driven Testing, 2016.

[ISO17a] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 12207:2017-11 System und Software-Engineering - Software life cycle processes, 2017.

[ISO17b] ISO Internationale Organisation für Normung. ISO/IEC/IEEE 24765:2017-09: Systems and software engineering — Vocabulary, 2017.

[ISO18a] ISO Internationale Organisation für Normung. ISO 26262-10:2018: Straßenfahrzeuge - Funktionale Sicherheit - Teil 10: Leitfaden für ISO 26262, 2018.

[ISO18b] ISO Internationale Organisation für Normung. ISO 26262-1:2018: Straßenfahrzeuge - Funktionale Sicherheit - Teil 1: Vokabular, 2018.

[IST20] ISTQB. International Software Testing Qualifications Board Glossary: https://glossary.istqb.org/de/, Online; abgerufen am 04.10.2020.

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology*, 21, July 2022.

[JR04] Mark Jennings and Ravi Rangan. Managing Complex Vehicle System Simulation Models for Automotive System Development. *Journal of Computing and Information Science in Engineering*, 4(4):372, 2004.

[JTH18a] Katharina Juhnke, Matthias Tichy, and Frank Houdek. Challenges Concerning Test Case Specifications in Automotive Software Testing. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 33–40, 2018.

[JTH18b] Katharina Juhnke, Matthias Tichy, and Frank Houdek. Challenges with Automotive Test Case Specifications. In *Proceedings of the 40th Inter-*

national Conference on Software Engineering: Companion Proceeedings, ICSE '18, pages 131–132, New York, NY, USA, 2018. ACM.

[JTH18c] Katharina Juhnke, Matthias Tichy, and Frank Houdek. Quality Indicators for Automotive Test Case Specifications Conference 2018 (SE 2018), Ulm, Germany, March 06, 2018. In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Michael Striewe, Sven Strickroth, Ulrike Lucke, Horst Lichter, Dirk Riehle, Andreas Steffens, Robert Höttger, Jörg Teßmer, and Jan-Philipp Steghöfer, editors, *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018 (SE 2018), Ulm, Germany, March 06, 2018*, CEUR Workshop Proceedings, pages 96–100. CEUR-WS.org, 2018.

[JTH21] Katharina Juhnke, Matthias Tichy, and Frank Houdek. Challenges concerning test case specifications in automotive software testing: assessment of frequency and criticality. *Softw. Qual. J.*, 29(1):39–100, 2021.

[Juh21] Katharina Juhnke. *Improving the quality of automotive test case specifications*. PhD thesis, Universität Ulm, 2021.

[JUn20] JUnit. : https://junit.org/junit5/, Online; abgerufen am 25.02.2020.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018.

[Kau21] Oliver Kautz. *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Aachener Informatik-Berichte, Software Engineering, Band 46. Shaker Verlag, April 2021.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications.

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *ACM Transactions on Internet of Things*, November 2022.

[KKRvW18] Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Michael von Wenckstern. Finding Inconsistencies in Design Models and Require-

ments by Applying the SMARDT Process. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES'18)*, Univ. Hamburg, April 2018.

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.

[KL19] Richard Knaster and Dean Leffingwell. *SAFe 4.5 distilled: Applying the Scaled Agile Framework for Lean enterprises.* Addison-Wesley, Boston, 2019.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, LNCS 9679, pages 122–137. Springer, June 2016.

[KMP+07] Matthias Kirchmayr, Mark Müller, Birgit Penzenstadler, Ernst Sikora, and Thorsten Weyer. Essenzieller REMsES-Leitfaden: Projekt REMsES; Förderkennzeichen: 01 IS F06 D, 2007.

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM SIGPLAN, October 2021.

[KMS+17] Stefan Kriebel, Vincent Moyses, Georg Strobl, Johannes Richenhagen, Phillip Orth, Stefan Pischinger, Christoph Schulze, Timo Greifenberg, and Bernhard Rumpe. The Next Generation of BMW's Electrified Powertrains: Providing Software Features Quickly by Model-Based System Design. In *26th Aachen Colloquium Automobile and Engine Technology*, October 2017.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KN17] L. Krejčí and J. Novák, editors. *Model-based testing of automotive distributed systems with automated prioritization: 2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, 2017.

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[Koh19] Jonas Kohlschmidt. *Feasibility Study of Software in the Loop Testing for SMArDT*. Master thesis, Technical University of Denmark, Kongens Lyngby, Denmark, 2019.

[Koo19] Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung. V-Modell-XT-Gesamt: Das deutsche Referenzmodell für Systementwicklungsprojekte: ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/2.3/V-Modell-XT-Gesamt.pdf, Online; abgerufen am 08.04.2019.

[KPB02] Cem Kaner, Bret Pettichord, and James Bach. *Lessons learned in software testing: A context-driven approach*. Wiley, New York, 2002.

[KPM13] Abhinaya Kasoju, Kai Petersen, and Mika V. Mäntylä. Analyzing an automotive testing process with evidence-based software engineering. *Information and Software Technology*, 55(7):1237–1259, 2013.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*,

Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[KR18] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRGK18] Stefan Kriebel, Johannes Richenhagen, Christian Granrath, and Christopher Kugler. Systems Engineering mit SysML: Der Weg in die Zukunft? In *MTZ Motortechnische Zeitschrift*, pages 48–53. Springer Fachmedien Wiesbaden, Wiesbaden, 2018.

[Kri18] Stefan Kriebel. Pains in Modeling: SysML-based Deployment in an Engineering Domain: Invited Talk at 1st Workshop on Pains in Model-Driven Engineering Practice, Conference on Model Driven Engineering Languages and Systems (MODELS'18): https://sites.google.com/view/pains-2018/home, 2018.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.

[KRR18] Oliver Kautz, Alexander Roth, and Bernhard Rumpe. Achievements, Failures, and the Future of Model-Based Software Engineering. In Volker Gruhn and Rüdiger Striemer, editors, *The Essence of Software Engineering*, pages 221–236. Springer Open, Cham, Switzerland, 2018.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner

Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.

[KRSvW18] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[KRSW22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software*, 183:1–21, January 2022.

[Kru03] Philippe Kruchten. *The rational unified process: An introduction.* The Addison-Wesley object technology series. Addison-Wesley, Upper Saddler River, NJ, 3. edition, 2003.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020.

[KSS21] Alexander Knüppel, Leon Schaer, and Ina Schaefer. How much Specification is Enough? Mutation Analysis for Software Contracts. In *9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021*, pages 42–53. IEEE, 2021.

[Küh04] Thomas Kühne. What is a Model? In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*, Dagstuhl Seminar Proceedings 04101. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Deutschland, 2004.

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.

[LAK+17] Grischa Liebel, Anthony Anjorin, Eric Knauss, Florian Lorber, and Matthias Tichy. Modelling Behavioural Requirements and Alignment with Verification in the Embedded Industry. In Lu\'ıs Ferreira Pires, Slimane Hammoudi, and Bran Selic, editors, *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017*, pages 427–434. SciTePress, 2017.

[Lig09] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2. edition, 2009.

[LL10] Steve Lehto and Jay Leno. *Chrysler's Turbine Car: The Rise and Fall of Detroit's Coolest Creation*. Chicago Review Press, 2010.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[LMT+18] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an

industrial survey on the state-of-practice. *Softw. Syst. Model.*, 17(1):91–113, 2018.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.* Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LPG11] Jonathan Lasalle, Fabien Peureux, and Jérôme Guillet. Automatic Test Concretization to Supply End-to-end MBT for Automotive Mechatronic Systems. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, pages 16–23, New York, NY, USA, 2011. ACM.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[LTBT17] Tianhai Liu, Shmuel Tyszberowicz, Bernhard Beckert, and Mana Taghdiri. Computing Exact Loop Bounds for Bounded Program Verification. In Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang, editors, *Dependable software engineering*, Lecture Notes in Computer Science 10606, pages 147–163. Springer, Cham, 2017.

[LTK+18] Grischa Liebel, Matthias Tichy, Eric Knauss, Oscar Ljungkrantz, and Gerald Stieglbauer. Organisation and communication problems in automotive requirements engineering. *Requir. Eng.*, 23(1):145–167, 2018.

[LTK19] Grischa Liebel, Matthias Tichy, and Eric Knauss. Use, potential, and showstoppers of models in automotive requirements engineering. *Software & Systems Modeling*, 18(4):2587–2607, 2019.

[LV10] Craig Larman and Bas Vodde. *Practices for scaling lean & agile development: Large, multisite, and offshore product development with large-scale Scrum.* Safari Tech Books Online. Addison-Wesley, Upper Saddle River, NJ, 2010.

[LvO15] Hartmut Lehbrink and Jochen von Osterroth. *BMW: Jubilee Edition.* h.f.ullmann Publishing, Potsdam, 2015.

[MAG20] MAGNA Telemotive GmbH. Telemotive Test Automation: www.telemotive.de/de/produkte/telemotive-software-solutions/telemotive-test-automation, Online; abgerufen am 13.02.2020.

[Man10] Manfred Broy. Multifunctional software systems: Structured modeling and specification of functional requirements. *Science of Computer Programming*, 75(12):1193–1214, 2010.

[Mar02] Martin Rohdin, Lars Ljungberg and Ulrik Eklund. A Method for Model Based Automotive Software Development. *12th Euromicro Conference on Real-Time Systems*, 2002.

[Mat19] MathWorks. Simulink - Simulation und Model-Based Design:

www.de.mathworks.com/products/simulink.html, Online; abgerufen am 02.07.2019.

[Mic20] Microsoft. Microsoft Research: www.microsoft.com/en-us/research, Online; abgerufen am 20.02.2020.

[Mil17] Ivana Miljkovic. *Generating Test Cases Generating Test Cases from Activity Diagrams in an industrial Context: Master Thesis*. PhD thesis, RWTH Aachen University, Aachen, 2017.

[MJJ12] Shahbakhti Mahdi, Li Jimmy, and Hedrick J. Karl. Early model-based verification of automotive control system implementation. In *2012 American Control Conference (ACC)*, pages 3587–3592, 2012.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[Mod19] Modelica Association. Modelica: www.modelica.org, Online; abgerufen am 21.08.2019.

[Moo56] Edward F. Moore. Gedanken-Experiments on Sequential Machines. In C. E. Shannon, editor, *Automata Studies*, Annals of Mathematics Studies,

pages 129–154. Princeton University Press, Princeton, UNITED STATES, 1956.

[MPF09] Cem Mengi, Antonio Navarro Pérez, and Christian Fuß. Modellierung variantenreicher Funktionsnetze im Automotive Software Engineering. In *Proceedings Informatik 2009. GI Edition - Lecture Notes in Informatics (LNI)*. 2009.

[MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 9–12. IEEE, May 2022.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of*

the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), pages 444–454. ACM New York, 2013.

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016.

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021.

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials.* Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[MSNRR15] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[MSWD15] Hajo Meinert, Thomas Senger, Norman Wiebking, and Christian Diegel-

mann. Die Plug-in-Hybridtechnologie im neuen BMW X5 eDrive. *MTZ - Motortechnische Zeitschrift*, 76(5):16–21, 2015.

[NoM19] NoMagic. MagicDraw: www.nomagic.com/products/magicdraw, Online; abgerufen am 05.02.2019.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[OMG14] OMG Object Management Group. OMG Object Constraint Language: Version 2.4, 2014.

[OMG17a] OMG Object Management Group. OMG Systems Modeling Language (OMG SysML): Version 1.5, 2017.

[OMG17b] OMG Object Management Group. OMG Unified Modeling Language: Version 2.5.1, 2017.

[OMG19] OMG Object Management Group. www.omg.org, Online; abgerufen am 05.06.2019.

[Pat82] Gerold Patzak. *Systemtechnik - Planung komplexer innovativer Systeme: Grundlagen, Methoden, Techniken.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[PBDH16] Klaus Pohl, Manfred Broy, Heinrich Daembkes, and Harald Hönninger, editors. *Advanced model-based engineering of embedded systems: Extensions of the SPES 2020 methodology.* Springer, Cham, Switzerland, 2016.

[PBFG07] Gerhard Pahl, Wolfgang Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. *Konstruktionslehre: Grundlagen erfolgreicher Produktentwicklung ; Methoden und Anwendung.* Springer, Berlin and Heidelberg, 7. edition, 2007.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016.

[PBKS07] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, editors. *Software Engineering for Automotive Systems: A Roadmap: Future of Software Engineering, 2007. FOSE '07*, 2007.

[Pel18] Jan Peleska. Model-based avionic systems testing for the airbus family. In *2018 IEEE 23rd European Test Symposium (ETS)*, pages 1–10, 2018.

[PES20] Tobias Pett, Domenik Eichhorn, and Ina Schaefer. Risk-based compatibility analysis in automotive systems engineering. In Esther Guerra and Ludovico Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 34:1–34:10. ACM, 2020.

[Pfa10] Christian Pfaller. *Anforderungsorientierter modellbasierter Softwaretest reaktiver Systeme*. Dissertation, Technische Universität München, München, 2010.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PHAB12] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, Berlin and Heidelberg, 2012.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen*. Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.

[Piq14] Jean-Denis Piques. SysML for embedded automotive for embedded automotive systems: SysCARS methodology. In *Proceedings of Embedded Real Time Software and Systems (ERTS2)*, ERTS'2014. Touluse, France, 2014.

[Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project*. Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Pre03] Alexander Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme*. Dissertation, Technische Universität München, München, 2003.

[PS16] Stefan Pischinger and Ulrich Seiffert, editors. *Vieweg Handbuch Kraftfahrzeugtechnik*. ATZ / MTZ-Fachbuch. Springer Vieweg, Wiesbaden, 8. edition, 2016.

[PSAK04] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real. *International Journal on Software Tools for Technology Transfer*, 5(2):140–157, 2004.

[PSP09] Birgit Penzenstadler, Ernst Sikora, and Klaus Pohl. A Requirements Reference Model for Model-Based Requirements Engineering in the Automotive Domain. In Martin Glinz and Patrick Heymans, editors, *Requirements engineering: foundation for software quality*, Lecture Notes in Computer Science 5512, pages 212–217. Springer, Berlin, 2009.

[PTC19] PTC. Integrity Modeler: https://www.ptc.com/de/products/plm/plm-products/integrity-modeler-what-is-new: Online: abgerufen am, 05.02.2019.

[PTR15] Alexandre Petrenko, Omer Nguena Timo, and S. Ramesh. Model-based testing of automotive software. In Unknown, editor, *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, pages 1–6, New York, New York, USA, 2015. ACM Press.

[RBBS02] Martin Rappl, Peter Braun, Michael von der Beeck, and Christian Schröder. Automotive Software Development: A Model Based Approach. In *SAE Technical Paper*. SAE International, 2002.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RNB12] Konrad Reif, Karl-Ernst Noreikat, and Kai Borgeest. *Kraftfahrzeug-Hybridantriebe: Grundlagen, Komponenten, Systeme, Anwendungen*. ATZ / MTZ-Fachbuch. Vieweg+Teubner Verlag, Wiesbaden, 2012.

[Ros16] Hans-Leo Ross. *Functional safety for road vehicles: New challenges and solutions for E-mobility and automated driving*. Springer International Publishing, 2016.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven

Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton.* Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 380–402, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML: Sprache, Konzepte Und Methodik*. Xpert.Press. Springer, Dordrecht, 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[SB02] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*. Series in agile software development. Prentice Hall, Upper Saddle River, NJ, 2002.

[Sch08] Markus Schneider. *Logistikplanung in der Automobilindustrie: Konzeption eines Instruments zur Unterstützung der taktischen Logistikplanung vor "Start-of-Production" im Rahmen der Digitalen Fabrik: Zugl.: Regensburg, Univ., Diss., 2008*. Gabler Edition Wissenschaft. Gabler Verlag / GWV Fachverlage GmbH Wiesbaden, Wiesbaden, 2008.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sch19] Christoph Schulze. *Agile Software-Produktlinienentwicklung im Kontext heterogener Projektlandschaften*. Aachener Informatik-Berichte, Software Engineering, Band 40. Shaker Verlag, May 2019.

[SE-20] SE-Lehrstuhl für Softwaretechnik. der Rheinisch-Westfälische Technische Hochschule Aachen, 12.9.2020.

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela

Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SL12] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester Foundation Level nach ISTQB-Standard.* dpunkt.verlag, 5. edition, 2012.

[SLH+17] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. A Chrestomathy of DSL Implementations. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 103–114, New York, NY, USA, 2017. Association for Computing Machinery.

[SM91] John W. Senders and Neville P. Moray. *Human error: Cause, prediction, and reduction.* CRC Press, Hillsdale, 1991.

[SM18] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020.

[SP10] Christian Schwarzl and Bernhard Peischl, editors. *Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems: 2010 10th International Conference on Quality Software*, 2010.

[Spa20] SparxSystems. Enterprise Architect: https://www.sparxsystems.de/: Online; abgerifen am, 05.02.2020.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[SSS18] Alexander Schlie, Sandro Schulze, and Ina Schaefer. Comparing Multiple MATLAB/Simulink Models Using Static Connectivity Matrix Analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 160–171, 2018.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie.* Springer, Wien, 1973.

[Sta20] Statistisches Bundesamt. Umgang mit Veröffentlichungsfehlern: www.destatis.de/DE/Methoden/Qualitaet/richtlinie-fehlerbehandlung.pdf?__blob=publicationFile: VöFe-Richtlinie 2013, Online; abgerufen am 30.01.2020.

[Stö17] Harald Störrle. How are Conceptual Models used in Industrial Software Development? In Emilia Mendes, Steve Counsell, and Kai Petersen,

editors, *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 160–169, New York, NY, USA, 06152017. ACM.

[SV06] Thomas Stahl and Markus Völter. *Model-driven software development: Technology, engineering, management*. John Wiley, Chichester England and Hoboken NJ, 2006.

[SVWH11] Andreas Spillner, Karin Vossberg, Mario Winter, and Peter Haberl. Wie wird in der Praxis getestet? Wie wird in der Praxis getestet? Online-Umfrage in Deutschland, Schweiz und Österreich: Online-Themenspecial Testing 2011. *Objektspektrum*, 2011, 2011.

[SZ16] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. Springer Fachmedien, Wiesbaden, 2016.

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021.

[TF17] Viviane Trachsel and Marcel Fallegger. Silodenken überwinden. *Controlling & Management Review*, 61(7):42–49, 2017.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[TJ07] Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Automated Software Engineering*, 14(1):87–121, 2007.

[TMS+16] Ghizlane Tibba, Christoph Malz, Christoph Stoermer, Natarajan Nagarajan, Licong Zhang, and Samarjit Chakraborty. Testing Automotive Embedded Systems Under X-in-the-loop Setups. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 35:1–35:8, New York, NY, USA, 2016. ACM.

[Tra20] TraceTronic GmbH. ECU-Test: www.tracetronic.de/produkte/ecu-test, Online; abgerufen am 13.02.2020.

[TTS+19] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. Towards efficient analysis of variation in time and space. In Carlos Cetina, Oscar Dërnava, Leopoldo Teixeira, Thomas Thüm, and Tewfik Ziadi, editors, *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, pages 69:1–69:8. ACM, 2019.

[Tur37] Alan M. Turing. On Computable Numbers, with an Application to the

Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[UL07] Mark Utting and Bruno Legeard. *Practical model-based testing: A tools approach*. Safari Tech Books Online. Morgan Kaufmann/Elsevier, Amsterdam, 2007.

[VDI04] VDI Verein Deutscher Ingenieure e.V. VDI 2206:2004-06: Entwicklungsmethodik für mechatronische Systeme, 2004.

[Vec20] Vector Informatik GmbH. vTESTstudio - Komfortables Erstellen automatisierter Testabläufe für eingebettete Systeme: www.vector.com/de/de/produkte/produkte-a-z/software/vteststudio, Online; abgerufen am 13.02.2020.

[VF13] Andreas Vogelsang and Steffen Fuhrmann. Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 267–272, 2013.

[VFW16] Andreas Vogelsang, Henning Femmer, and Christian Winkler. Take Care of Your Modes! An Investigation of Defects in Automotive Requirements. In Maya Daneva and Oscar Pastor, editors, *Requirements Engineering: Foundation for Software Quality - 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings*, Lecture Notes in Computer Science 9619, pages 161–167. Springer, 2016.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[Wei08] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. The MK / OMG Press. Elsevier professional, 1. edition, 2008.

[Wei12] Stephan Weissleder. Modellbasiertes Testen für Varianten. *Objektspektrum*, 2012(Testing/2012), 2012.

[Wes19] Patricia Wessel. *Evaluation of a SysML Test Case Generator*. Master thesis, RWTH Aachen University, Aachen, 2019.

[WFV20] Katharina Winter, Henning Femmer, and Andreas Vogelsang. How Do Quantifiers Affect the Quality of Requirements? In Nazim H. Madhavji, Liliana Pasquale, Alessio Ferrari, and Stefania Gnesi, editors, *Requirements Engineering: Foundation for Software Quality - 26th International Working Conference, REFSQ 2020, Pisa, Italy, March 24-27, 2020, Proceedings [REFSQ 2020 was postponed]*, Lecture Notes in Computer Science 12045, pages 3–18. Springer, 2020.

[WGM+11] Stephan Weissleder, Baris Güldali, Michael Mlynarski, Arne-Michael Törsel, David Faragó, David Prester, and Mario Winter. Modellbasiertes Testen: Hype oder Realität? *Objektspektrum*, 2011(06):59–65, 2011.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014.

[Wit16] Frank Witte. *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung*. Springer Vieweg, Wiesbaden, 1. edition, 2016.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[Wor21a] World Wide Web Consortium. XML: https://www.w3.org/XML/, Online; abgerufen am 28.02.2021.

[Wor21b] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, Oktober 2021.

[WRF+15] David D. Walden, Garry J. Roedler, Kevin Forsberg, R. Douglas Hamelin, and Thomas M. Shortell, editors. *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. Wiley, Hoboken, NJ, 4. edition, 2015.

[WW03] Matthias Weber and Joachim Weisbrod. Requirements engineering in automotive development: experiences and challenges. *IEEE Software*, 20(1):16–24, 2003.

[ZBF+05] D. Ziegenbein, P. Braun, U. Freund, A. Bauer, J. Romberg, and B. Schatz, editors. *AutoMoDe - model-based development of automotive software: Design, Automation and Test in Europe*, 2005.

[Zim80] Hubert Zimmermann. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.

Anhänge

Anhang A

Abkürzungen

AD	Aktivitätsdiagramm (engl. „Activity Diagram“)
AD4S	Aktivitätsdiagramme für SMArDT (engl. „Activity Diagrams for SMArDT“)
API	Anwendungsschnittstelle (engl. „Application Programming Interface“)
ASIL	Automotive Sicherheitsanforderungsstufe (engl. „Automotive Safety Integrity Level“)
AST	abstrakter Syntaxbaum (engl. „Abstract Syntax Tree “) → Anhang B: abstrakter Syntaxbaum
BDD	Blockdefinitionsdiagramm (engl. „Block Definition Diagram“)
BEV	konventionelles Elektroauto (engl. „Battery Electric Vehicle“)
C&C	Komponenten- und Konnektor (engl. „Component and Connector“)
CAD	computergestützte Konstruktion (engl. „Computer-Aided Design“)
CD	Klassendiagramm (engl. „Class Diagram“)
CPS	Cyber-Physisches System (engl. „Cyber-Physical System “)
DSL	domänenspezifische Sprache (engl. „Domain Specific Language“)
DSML	domänenspezifische Modellierungssprache (engl. „Domain Specific Modeling Language“)
E/E	Elektrik und Elektronik
EBNF	Erweiterte Backus-Naur-Form
FMEA	Fehlermöglichkeits- und Einflussanalyse (engl. „Failure Mode and Effects Analysis“)
GPML	Universelle Modellierungssprache (engl. „General Purpose Modeling Language“)
HIL	Hardware in the Loop
IBD	Internes Blockdiagramm (engl. „Internal Block Diagram“)

ID	Identifikator
MBT	modellbasiertes Testen
MC/DC	Modified Condition / Decision Coverage
MIL	Model in the Loop
OCL	Object Constraint Language
OMG	Object Management Group
PHEV	Plug-In-Hybrid Electric Vehicle
SC	Zustandsdiagramm (engl. „State Chart“)
SD	Sequenzdiagramm (engl. „Sequence Diagram“)
SIL	Software in the Loop
SMArDT	Spezifikationsmethode für Anforderung, Design und Test
SOP	Start of Production → Anhang B: SOP
SysML	Systems Modeling Language
UCD	Anwendungsfalldiagramm (engl. „Use Case Diagram“)
UML	Unified Modeling Language
UML/P	UML/Programmiergeeignet → Anhang B: UML/P
V&V	Verifizierung und Validierung
VIL	Vehicle in the Loop
XML	erweiterbare Auszeichnungssprache (engl. „Extensible Markup Language“)

Anhang B

Glossar

Die folgenden Begriffe werden im Kontext dieser Arbeit, wie angegeben, verwendet.

Absicherung
Eine Absicherung (engl. „safeguarding“) umfasst die Fehleridentifikation und Mängelbeseitigung.

abstrakter Syntaxbaum
Ein abstrakter Syntaxbaum (engl. „Abstract Syntax Tree “) ist ein attribuierter und gelegentlich getypter Baum, der ein geparstes Eingabeprogramm oder -modell repräsentiert.

Abstraktionsebene
Eine (Abstraktions-) Ebene (engl. „level“) ist in SMArDT ein Niveau der funktionalen Abstraktion. Es gibt vier Ebenen: Ebene A (Betrachtungseinheit), Ebene B (Funktionales Konzept), Ebene C (Technische Lösung) und Ebene D (Realisierung).

Anforderung
Eine Anforderung ist eine qualitative und/oder quantitative Bedingung oder Fähigkeit, die ein System, Subsystem oder Produkt erfüllen oder besitzen muss oder soll [ISO17b].

Applikationssoftwarebaustein
Ein Applikationssoftwarebaustein ist ein Funktionsbaustein, der ausschließlich aus Anwendungssoftware besteht beziehungsweise bestehen wird.

Artefakt
Ein Artefakt ist ein konkretes in sich geschlossenes Entwicklungsergebnis, mit eindeutigen Namen [Gre19]. Artefakte können in der Software individuell speicherbare Einheiten, wie ein UML-Modell, ein SysML-Modell, eine .java-Datei oder eine .txt-Datei sein.

äußerer Fehler
Ein äußerer Fehler (engl. „failure“) ist die äußerlich sichtbare Abweichung von der Spezifikation oder der Anforderungen des Systems [ISO17b].

Basissoftwarebaustein
Ein Basissoftwarebaustein ist ein Funktionsbaustein und stellt notwendige Schnittstellen und Dienste von Hardwarebausteinen für Applikationssoftwarebausteine zur Verfügung.

Black-Box-Testfall
Ein Black-Box-Testfall aus einer Spezifikation abgeleitet, die keine ausreichenden Informationen über das innere Verhalten der Funktion beinhaltet [Lig09, ISO17b, UL07].

Dekompositionsschicht
Eine (Dekompositions-) Schicht (engl. „layer“) ist in SMArDT eine Stufe der hierarchischen Dekomposition. Der Zweck der hierarchischen Dekomposition ist Aufteilung der Funktionalität eines Systems auf weitere Subsysteme, um die sich aus der Funktionalität resultierenden Aufgaben intellektuell beherrschbar zu machen.

Disziplin
Eine Disziplin ist ein Wissenschaftszweig beziehungsweise ein akademisches Fachgebiet [Dud20]. Beispiele sind die Disziplinen Elektrotechnik, Informatik, Maschinenbau, Psychologie, Rechtswissenschaft, Soziologie etc.

Domäne
Eine Domäne, auch Produktdomäne, ist ein Spezialgebiet, in dem Expert*innen mit Fachwissen arbeiten und sich besonders gut auskennen [Dud20]. Beispiele sind die Domänen Automobil, IT, Luftfahrt, Medizin etc.

dynamischer Test
Ein dynamischer Test dient der Ermittlung von Fehlern und Verstößen gegen Standards und Spezifikationen [SL12, ISO13a, Lig09]. Er führt Systeme und Komponenten mit konkreten Eingabewerten aus [ISO13a, Lig09].

Ebene
siehe Abstraktionsebene

Fehler
Ein Fehler (engl. „failure“) ist die Nichterfüllung einer Spezifikation oder Anforderung [ISO17b].

Fehlhandlung
Eine Fehlhandlung (engl. „error“) ist ein von menschlichen Handeln hervorgerufener Fehler [ISO17b].

formales Modell
Ein formales Modell ist wohldefiniert und wohlgeformt.

Funktionsbaustein
Ein Funktionsbaustein ist ein funktionales wiederverwendbares Artefakt, der bei der Ausführung ein oder mehrere Werte liefert. Die Wiederverwendbarkeit ist mit den Instanzen gegeben, die von einem Funktionsbaustein erzeugt werden können. Ein Funktionsbaustein besteht entweder aus einem Applikationssoftwarebaustein, aus einem Basissoftwarebaustein und/oder einem Hardwarebaustein.

Funktionsbibliothek
Eine Funktionsbibliothek oder auch vollständige Funktionsgruppe beinhaltet alle von dem System bereitgestellten Funktionen.

Funktionsgruppe
Eine Funktionsgruppe gruppiert in SMArDT thematisch zusammenhängende Subfunktionen. Es gibt zwei Arten von Funktionsgruppen, M-Funktionsgruppen und S-Funktionsgruppen.

Funktionsprinzip
Ein Funktionsprinzip ist eine abstrakte Beschreibung einer Funktion. Wenn möglich soll das Funktionsprinzip keine Lösung vorwegnehmen.

Grey-Box-Testfall
Ein Grey-Box-Testfall wird ohne einen vorliegenden implementierenden Programmcode aber mit ausreichenden Informationen über das Verhalten einer Funktion erstellt [Wit16, ISO15b]. In dieser Arbeit basieren Grey-Box-Tests auf detailierten Verhaltens- und Struktursmodellen.

Hardware
Hardware sind physische Geräte, die einen Teil oder ein komplettes Informationssystems umfassen [ISO17b].

Hardwarebaustein
Ein Hardwarebaustein ist ein Funktionsbaustein, der ausschließlich aus Hardware besteht beziehungsweise bestehen wird.

Hardwarekomponente
Eine Hardwarekomponente ist ein Element, das ausschließlich aus Hardware besteht.

Informationsfluss
Ein Informationsfluss (engl. „InformationFlow“) dient der Darstellung eines Austauschs von Informationen zwischen zwei oder mehreren Elementen auf hohen Abstraktionsebenen, wie beispielsweise der SMArDT-Ebene *B* [OMG17b, OMG17a]. Informationsflüsse erlauben noch nicht vollständig spezifizierte oder weniger detaillierte Materie-, Energie- oder Informationsflüsse abstrakt zu modellieren. Auf diese Weise können abstrakte Modelle das intellektuelle Verständnis erleichtern und bewusst Entscheidungen über Details in tiefere Detailebenen des Modells verschoben werden, wie beispielsweise der Ebene *C* und *D*.

innerer Fehler
Ein Fehler (engl. „defect“) ist die Nichterfüllung einer Spezifikation oder Anforderungen [ISO17b].

Integration
Eine Integration ist eine Aktion in der Softwarekomponenten und/oder Hardwarekomponenten zu einem System kombiniert werden [ISO17b].

Komponente
Eine Komponente ist ein realer mechatronischer Teil eines Ganzen und erfüllt eine oder mehrere bestimmte Funktionen, die für das Element höherer Ordnung von Bedeutung ist [ISO17b, DIN18a, PS16].

Kontextbedingungen
Die Kontextbedingungen sind (prüfbare) Einschränkungen einer Sprache und schränken die Menge der erlaubten Modelle ein [Grö10, Rum16]. Kontextbedingungen ergänzen demnach die Definition der Syntax und ermöglichen kontextsensitive Einschränkungen [HR17].

Kundenfunktion
Die Kundenfunktion ist in SMArDT eine Beschreibung der gewünschten Funktionalität eines Systems. Zusätzlich werden die gegebenen Schnittstellen des Systems zur Umwelt und den anderen Systemen der gleichen Dekompositionsschicht (n mit $n \in \mathbb{N}$) oder übergeordneten Schichten ($n-1$ mit $n \in \mathbb{N}$) festgehalten.

Mechatronik
„Mechatronik bezeichnet das synergetische Zusammenwirken der Fachdisziplinen Maschinenbau, Elektrotechnik und Informationstechnik beim Entwurf und der Herstellung industrieller Erzeugnisse sowie bei der Prozessgestaltung“ [VDI04].

mechatronisches System
Ein mechatronisches System, besteht aus mechatronischen Komponenten und Systemen.

M-Funktionsgruppe
Eine M-Funktionsgruppe, auch mechatronische Funktionsgruppe genannt, beschreibt in SMArDT ein mechatronisches System, dessen Umsetzung noch nicht (vollständig) spezifiziert ist (siehe Funktionsgruppe). Einer M-Funktionsgruppe sind eindeutig eine Arbeitsgruppe, Softwareschnittstellen, Hardwareschnittstellen und Eigenschaften zugewiesen. Die M-Funktionsgruppe ist eine Generalisierung der Subsysteme und Komponenten.

Modell
Ein Modell ist nach [Sta73] eine abstrahierende Darstellung eines Originals, welches es (das Original) im Hinblick auf einen pragmatischen Zweck beschreibt.

modellbasierte Entwicklung
In der modellbasierten Entwicklung spielen Modelle eine wichtige Rolle für die Entwicklung eines Produktes, sind aber nicht notwendigerweise die primären Artefakte für die Entwicklung [BWC12].

modellgetriebene Entwicklung
In der modellgetriebenen Entwicklung sind Modelle die primären Artefakte und werden nicht allein zu Dokumentationszwecken erstellt werden [BWC12]. Die Modelle werden genutzt, um die Absichten von Entwickler*innen präzise zu erfassen, anstatt diese nur informal auszudrücken [GSCK04].

Modul
Ein Modul ist im V-Modell [Koo19] ein Teil einer Komponente und ein atomares Elemente der Hardware und Software [Koo19]. Das Modul entspricht in SMArDT einem Funktionsbaustein.

Referenzmodell
Ein Referenzmodell ist ein spezifisches Modell innerhalb einer Anwendungsdomäne [ANT14]. Es dient als Dokumentation für bestehende spezifische Anwendungen oder als Ausgangspunkt für die Entwicklung eines Schemas für bestimmte Anwendungen. In dieser Arbeit werden Referenzmodelle mithilfe der Modellierungstechnik des CD [OMG17b] modelliert.

S-Funktionsgruppe
Eine S-Funktionsgruppe (Software-Funktionsgruppe) gruppiert in SMArDT Funktionen, die ausschließlich aus Applikationssoftware bestehen (siehe Funktionsgruppe). Die S-Funktionsgruppe sammelt alle Applikationssoftwarefunktionen und deren Anforderungen. Die Funktionen der Softwarefunktionsgruppe werden anhand einer Softwarearchitektur auf Komponenten verteilt („deployed“).

Schicht
siehe Dekompositionsschicht

Schlüsselwort
Ein Schlüsselwort besteht aus einem oder mehreren Wörtern, die als Verweis auf eine bestimmte Aktion oder mehrere definierte Aktionen verwendet werden [ISO16].

Schlüsselwortbibliothek
Eine Schlüsselwortbibliothek ist ein Repository, das eine Reihe von Schlüsselwörtern enthält [ISO16].

SMArDT
Die Spezifikationsmethode für Anforderung, Design und Test (SMArDT) (engl. „Specification Method for Requirements, Design, and Test Engineering“) ist eine modellbasierte Entwicklungsmethode, die hierarchische Dekomposition des Systems und funktionale Abstraktion im ersten Schritt separiert betrachtet und in einem zweiten Schritt ergänzend integriert.

Software
Software ist ein Programm oder ein Satz von Programmen, Verfahren und Regeln, die ebenfalls zugehörige Dokumentation und Daten eines Computersystems umfassen [ISO17b].

Softwarebaustein
Ein Softwarebaustein ist ein abgegrenztes programmiertechnisches Artefakt [BF14]. Softwarebausteine entsprechen Softwaremodulen aus [Koo19] und ist in dieser Arbeit eine übergeordnete Kategorie für Applikationssoftwarebausteine und Basissoftwarebausteine.

Software Engineering
Das Software Engineering ist eine Ingenieursdisziplin, die mittels systematischer, disziplinierter und quantifizierbarer Ansätze, Konzepte, Prozesse und Werkzeuge die Entwicklung, den Betrieb und die Wartung von Software umsetzt [BF14, Rum16].

Softwarekomponente
Eine Softwarekomponente ist ein Element, das ausschließlich aus Software besteht.

SOP
Der Starttermin der regulären Serienproduktion (engl. „Start of Production“) ist der Start der Produktion aus Serienteilen auf Serienwerkzeugen unter Serienbedingungen [Sch08].

Spezifikation
Die Spezifikation beschreibt detailiiert die Anforderungen und die Funktionsweise von Systemelementen und deren logistische Unterstützung [Koo19]. Die Spezifikation dient der Entwicklung oder Validierung des Systems, Subsystems und/oder das Produkts [ISO17b].

statischer Test
Ein statischer Test, dient der Ermittlung von Fehlern und Verstößen gegen Standards und Spezifikationen, ohne dass ein System oder einer Komponente ausgeführt werden [SL12, ISO13a, Lig09].

Subsystem
Ein Subsystem ist in SMArDT ein Teil eines größeren Systems. Ein Subsystem der Dekompositionsschicht n ist ein System der nächsten Dekompositionsschicht $n + 1$ mit $n \in \mathbb{N}$.

System
Ein System ist eine interagierende Kombination von Elementen, die ein definiertes Ziel verfolgen [ISO17b, BF14, Pat82].

Systemmodell
Ein Systemmodell beschreibt ausgewählte Aspekte der Struktur, des Verhaltens, des Zustands und der Interaktion von Objekten und Konzepten des Systems [BCGR09b, BCGR09c].

Systems Engineering
Das Systems Engineering ist ein ganzheitlicher, interdisziplinärer Ansatz, um die Ziele der verschiedenen Disziplinen in einer gemeinsamen Lösung umzusetzen und während des Lebenszyklus zu unterstützen [ISO17b, CKS11, Wei08].

Test
Ein Test ist eine Reihe von Aktivitäten, die unter bestimmten Bedingungen auf einem System oder einer Komponente ausgeführt werden [ISO13a, ISO17b]. Dabei werden bestimmte Aspekte und Ergebnisse beobachtet und aufgezeichnet [ISO13a, ISO17b]. Dies ermöglicht eine Bewertung dieser Aspekte des Systems oder der Komponente [ISO13a, ISO17b].

Testbarkeit
Die Testbarkeit gibt die Effizienz beziehungsweise den Aufwand an, mit dem die Funktionalität und andere Eigenschaften eines (Sub-) Systems getestet werden können [ISO17b, ISO11, SL12]. Hierbei wird identifiziert, ob gewisse Kriterien erfüllt sind [ISO11, SL12].

Testen
siehe Test

Testfall
Ein Testfall ist eine Reihe von Testeingaben und Ausführungsbedingungen, die vorhersagbare Ergebnisse hervorrufen sollen [ISO17b, ISO13a].

Testfallautomatisierung
Eine Testfallautomatisierung ist eine Aktivität mit dem Ziel, die Eingabe von Daten und die Abarbeitung der Testschritte zu automatisieren.

Testfallentwurf
Ein Testfallentwurf umfasst eine grobe Vorbedingung, Anforderungen an die gewünschte Testumgebung und eine Verknüpfung zu den Anforderungen einer Spezifikation, um diese zurückverfolgen zu können.

Testfallspezifikation
Eine Testfallspezifikation definiert gleichsam eine Aktivität, aber auch das Produkt dieser Aktivität. Die Aktivität der Testfallspezifikation hat das Ziel, die einzelnen Testfallschritte und deren Reihenfolge zu definieren. Das Ergebnis der Aktivität wird ebenfalls Testfallspezifikation genannt.

Testkonzept
Ein Testkonzept beschreibt detailliert, welche Anforderungen getestet werden können, wie diese Anforderungen getestet werden sollen, welche Testumgebungen notwendig sind und wann, welche Anforderungen getestet werden sollen [ISO11, ISO13b].

Testmodell
Das Testmodell umfasst testrelevante Aspekte und stellt einen Testauftrag dar, der für die Testfallerstellung verwendet wird [ISO15b].

Testorakel
Ein Testorakel ist die Informationsquelle zur Ermittlung der erwarteten Ergebnisse eines Testfalls und kann unter anderem aus textuelle Anforderungen oder Modellspezifikationen bestehen.

Testumgebung
Die Testumgebung ist eine Umgebung, die benötigt wird, um Tests auf einer möglichst verhaltensähnlichen Konfiguration eines Systems auszuführen. Die Testumgebung umfasst Hardware, Firmware, Instrumentierung, Simulatoren, Softwarewerkzeuge und andere unterstützende Hilfsmittel, die für die Durchführung von Test vorgesehen sind oder dazu verwendet werden[IST20, ISO13a].

UML/P
Die UML/Programmiergeeignet (UML/P) ist ein Profil der UML, das in [Rum16] beschrieben wird, und für das eine textuelle Notation sowie dazu passende Werkzeuge entwickelt wurden [Sch12].

Vieraugenprinzip
Das Vieraugenprinzip stellt sicher, dass ein Artefakt oder Produkt nicht von der gleichen Person entwickelt und geprüft beziehungsweise validiert und verifiziert wird [Lig09, Wit16].

White-Box-Testfall
Ein White-Box-Testfall wird aus dem Programmcode abgeleitet [ISO15b, UL07, Lig09, Wit16].

Wirkprinzip
Das Wirkprinzip (engl. „operating principle“) ist ein Funktionszusammenhang, der die Gemeinsamkeit von einem physikalischem Effekt sowie geometrisch und stofflichen Merkmalen beschreibt, die ein reales sichtbares Prinzip darstellen [PBFG07]. Wirkprinzipien sind eine echte Teilmenge von Funktionsprinzipien.

Wohldefiniertheit
Die Wohldefiniertheit ist eine Eigenschaft der Semantik eines Modells. Ein Modell ist wohldefiniert, falls die formale Semantik des Modells nicht leer ist, alle verwendeten Elemente eindeutig definiert sind und demzufolge keine Möglichkeit der Mehrdeutigkeit besteht.

Wohlgeformtheit
Die Wohlgefromtheit ist eine Eigenschaft eines Modells. Ein Modell ist wohlgeformt, falls es syntaktisch korrekt ist und insbesondere die Kontextbedingungen erfüllt [Grö10].

Anhang C

Curriculum Vitae

Name	Markthaler
Vorname	Matthias
Geburtstag	08.07.1987
Geburtsort	Fürstenfeldbruck
Staatsangehörigkeit	deutsch
seit 2020	Testingenieur Systemintegration und -verifikation bei der BMW Group
seit 2017	Promotionsstudent Lehrstuhl für Software Engineering RWTH Aachen
2016 - 2020	Doktorand der BMW Group
2016	Abschluss als Master of Science in in Electrical Engineering an der Hochschule München
2015	Studium Electrical Engineering an der Federal University of São João del Rei
2014	Abschluss als Bachelor of Engineering in Elektrotechnik und Informationstechnik an der Hochschule München
2010 - 2016	Studium der Elektrotechnik und Informationstechnik an der Hochschule München
2008 - 2009	Studium der Elektrotechnik und Informationstechnik an der TU München
2007 - 2008	Zivildienst, Arbeiterwohlfahrt Fürstenfeldbruck
2007	Abitur am Graf Rasso-Gymnasium Fürstenfeldbruck

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.

Modeling will increasingly be used in development projects if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring, and evolution is defined.

The language workbench MontiCore [GKR+06, GKR+08, HKR21] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09], and refactoring in various modeling and programming languages [PR03]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].

In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].

To keep track of relevant structures, artifacts, and their relations in order to be able e.g. to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.

An artifact model is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real

architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRvW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.

For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKKR21].

According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11a], Class Diagrams [MRR11d], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17b, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18].

We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].

Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11a, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P, [Hab16] for MontiArc is used in domains such as cars or robotics [HRR12], and [AMN+20] for enterprise information systems based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HKR21].

In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15a, GHK+15b] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Höl18], we demonstrate how to systematically derive a transformation language in concrete syntax and e.g. in [HHRW15, AHRW17a] we have applied this technique successfully for several UML sub-languages and DSLs.

[HMSNRW16] presents how to generate extensible and statically type-safe visitors. In [MSNRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLMSN+15b, HLMSN+15a].

To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML) & the UML-P Tool

Starting with the early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] and is implemented in [Sch12].

Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied when checking variants of class diagrams [MRR11c] and object diagrams [MRR11e] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences in activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points are given in [GRR10].

We also discuss how to ensure and identify model quality [FHR08], how models, views, and the system under development correlate to each other [BGH+98b], and how to use modeling in agile development projects [Rum04], [Rum03] and [Rum02].

The question of how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99b], [FELR98] and [SRVK10].

The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also transferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11, HLMSN+15b, HLMSN+15a, HRW18, BEK+18, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].

[Wei12, HRW15, Höl18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution

have been discussed in [LRSS10] and [SRVK10]. [BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15], and [TAB+21] discusses the compositionality of analysis techniques for models.

The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented e.g. in [AHRW17a, BEH+20, BHRW21, BPRW20, HHRW15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPRW20, BEK+19].

In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, MSN17, KRV08, HLMSN+15b, HLMSN+15a, HMSNRW16, HKR21, BEK+18, BEK+19] and the backend [RRRW15, MSNRR16, GMR+16, HKR21, BEK+18, BBC+18]. In [GHK+15a, GHK+15b], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.

Language derivation is to our belief a promising technique to develop new languages for a specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].

How to automatically derive such a transformation language using a concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs.

We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.

The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services.

We use streams, statemachines, and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR10, HRR12] for architecture design and extensions for states [RRW13b, BKRW17a, RRW14, Wor16]. In [RRW13a], we introduce a code generation framework for MontiArc. [RRRW15] describes how the language is composed of individual sublanguages.

MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [NPR13], security in [HHRW15], and the robotics domain [AHRW17a, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].

[GHK+07] and [GHK+08a] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.

[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.

Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17b], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLMSN+15b, HLMSN+15a, HMSNRW16, MSNRR16, HRW18, BEK+18, BEK+19, BPRW20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].

[Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to robotics control.

[CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information, we have developed the concept of tagging languages in [GLRR15, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision, and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].

To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. And [BKRW17b] compares component and connector architectures similar to SysML' block definition diagrams.

In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99a]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] discusses the relationships between a system, a view, and a complete model in the context of the UML.

[GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these to class and object diagrams in

[MRR11f] as well as activity diagrams in [GRR10].

[Rum12] defines the semantics in a variety of code and test case generation, refactoring, and evolution techniques. [LRSS10] discusses the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.

A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code, they are not initially correct and need to be changed, evolved, and maintained over time. Model transformation is therefore essential to effectively deal with models [CFJ+16].

Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04, MRR10], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12], systematic model transformation language development [Wei12, HRW15, Höl18, HHRW15], repair of failed model evolution [KR18].

[Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97] and refining pipe-and-filter architectures is explained in [PR99]. This has e.g. been applied for robotics in [AHRW17a, AHRW17b].

Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].

Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11a] and for Simulink models in [RSW+15].

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example, cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08a, GKPR08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12].

Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15a] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: Ä digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21b] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21a] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21].

Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRvW17], autonomous driving [BR12a, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07].

In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest to European avionics [ZPK+11]. Optimized [KRSvW18] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.

A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14, Wor16, RRSW17, Wor21b]. In [RRW12], we use this language for describing requirements and in [RRW13a], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FNDR98] and [GHK+08b], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.

To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product

architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CrEST methodology for a systems engineering approach [BBR20].

Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.

In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07].

We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14, RRW13a, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20] is used as the underlying generator technology.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.

The MontiArcAutomaton language [RRW12, RRW14] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRRW15, HKR21] that perfectly fit robotic architectural modeling.

The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.

The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17a, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12a].

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment, and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed, and tested. A consistent requirement management connecting requirements with features in all development phases for the automotive domain is described in [GRJA12].

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08a]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMaRDT method [DGH+19, KMS+18].

[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.

Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a].

[MMR10] gives an overview of the state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.

MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRSW22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRSW22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [NPR13], distributed systems security [HHRW15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15].

In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales.

During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the technical specification of building services already.

We adapted the well-known concept of statemachines to be able to describe different states of a facility and validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].

Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].

We tackle these challenges by perusing a model-based, generative approach [NPR13]. At the core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale.

We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our development platforms. New services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g., in production or Internet-of-Things environments.

Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.

MontiGem [AMN+20] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15b, MSNRR15], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].

We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

Abbildungsverzeichnis

Tabellenverzeichnis