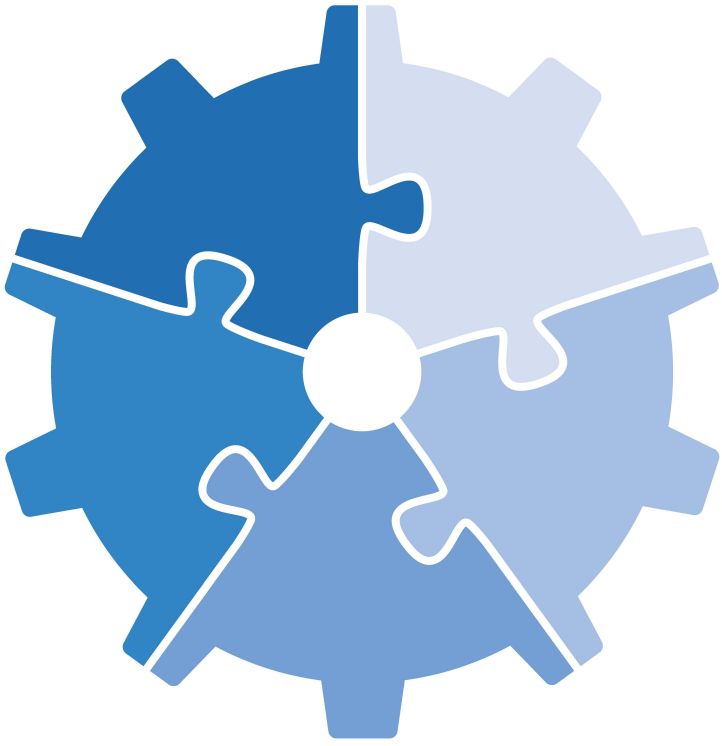
SE | RWTH AACHEN UNIVERSITY

Christoph Schulze

Agile Software-Produktlinien-entwicklung im Kontext heterogener Projektlandschaften



Aachener Informatik-Berichte,
Software Engineering

Band 40

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

Agile Software-Produktlinienentwicklung im Kontext heterogener Projektlandschaften

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

Dipl.-Inform.
Christoph Schulze

aus Düsseldorf

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessorin Dr.-Ing. Ina Schaefer

Tag der mündlichen Prüfung: 8. Februar 2019

[Sch19] C. Schulze:
Agile Software-Produktlinienentwicklung im Kontext heterogener Projektlandschaften.
Shaker Verlag, ISBN 978-3-8440-6683-8. Aachener Informatik-Berichte, Software Engineering, Band 40. Mai 2019.
www.se-rwth.de/publications/

Kurzfassung

Die Automobilindustrie sieht sich großen Herausforderungen gegenüber: Während der Markt immer kürzere Entwicklungszyklen, höhere Qualität und günstigere Produkte fordert, steigt gleichzeitig durch aktuelle Trends, wie autonomes Fahren oder die Digitalisierung des Fahrzeuges, die Komplexität und der Testaufwand des Produktes [EF17, Cas17, DTK07, Bro06].

Eine Möglichkeit erhöhte Qualität bei reduzierten Kosten zu realisieren ist durch die systematische Wiederverwendung modularer Einheiten gegeben. Die Software-Produktlinienentwicklung prognostiziert durch die Etablierung einer Software-Produktlinie eine deutliche Kosten- und Zeitreduktion durch intensive Wiederverwendung [PBL05]. Dabei birgt dieser Ansatz aber auch deutliche Gefahren: Die Entwicklung einer wiederverwendbaren Einheit erhöht durch die Einführung variabler Aspekte auch dessen Komplexität. Wird eine langfristige Strategie falsch ausgerichtet, kann es schnell dazu kommen, dass sich der erhöhte Aufwand nicht rentiert, da eine Wiederverwendung zwar technisch möglich ist, gleichzeitig aber die Nachfrage fehlt. Im Kontext eines Zulieferers ist es außerdem deutlich schwieriger, eine eigene Strategie unabhängig vom Kunden (OEMs) zu entwickeln.

Vergleicht man die durch die Forschung vorgeschlagenen Mechanismen zur Darstellung von Variabilität im Lösungsraum mit der aktuell in der Industrie gelebten Praxis, wird schnell deutlich, dass oft ein deutlicher pragmatischer Ansatz gewählt wird. Anstatt explizit Variabilität in Entwicklungsartefakten anzugeben, werden ähnliche Artefakte von Vorgängerprojekten als Basis kopiert und weiterentwickelt, das sogenannte *Clone&Own* [DRB+13]. Dieses meist für das aktuelle Software-Projekt im ersten Schritt sehr kostengünstige Verfahren wird allerdings oft sehr unsystematisch durchgeführt und birgt trotz seines kurzfristigen Mehrwertes langfristige Gefahren. Das Wissen um eine potentielle Wiederverwendung gegebener Varianten ist üblicherweise nur in den Köpfen der Experten aus Vorgängerprojekten gegeben und droht durch Personalwechsel sowie über Zeit verloren zu gehen. Des Weiteren ist die Wartung mehrerer Klone, die sich während ihrer Evolution weiterentwickeln, aber trotzdem eine gemeinsame Basis besitzen, erschwert. Insofern entsteht im ersten Moment zwar der Eindruck, dass durch unsystematisches *Clone&Own* sehr effizient Wiederverwendung betrieben werden kann, die eigentlichen Vorteile einer Software-Produktlinienentwicklung können aber nicht genutzt werden.

Innerhalb dieser Doktorarbeit wird der beschriebene Kontext im Rahmen beteiligter Industriepartner aufgegriffen und sowohl Ansätze eines *Clone&Own*-Verfahrens systematisiert sowie Methoden zur schrittweisen Migration in eine Software-Produktlinie ausgearbeitet. Dabei stehen sowohl Verfahren zur stufenweisen (extrinsischen, schnittstellen-basierten, semantischen) und automatischen Ähnlichkeitsanalyse als auch ein datenbankbasierte Etablierung und Wartung einer Software-Produktlinie im Fokus.

Basierend auf Ansätzen der agilen Software-Produktlinienentwicklung wird somit ein pragmatischer Ansatz vorgestellt, der im Kontext heterogener Projektlandschaften eines Zulieferers eine schrittweise Überführung in eine Software-Produktlinienentwicklung ermöglicht und gleichzeitig gegebene Normen der Automobilindustrie berücksichtigt, wie zum Beispiel CMMI [Pau02] oder ISO26262 [Hil12]. Der Ansatz wird durch mehrere Werkzeuge unterstützt, die im Kontext dieser Doktorarbeit in Zusammenarbeit mit Industriepartnern entwickelt und verwendet wurden.

Abstract

The automotive industry faces great challenges: the market demands shorter development cycles, higher quality and cheaper products, while at the same time the complexity and test effort is increased by current trends like autonomous driving or the digitalization of the vehicle[EF17, Cas17, DTK07, Bro06].

One possibility to achieve higher quality at reduced costs is the systematic reuse of modular units. Software product line engineering predicts a significant cost and time to market reduction through intensive reuse [PBL05]. However, this approach also poses significant dangers: the development of a reusable unit also increases its complexity. If a long-term strategy is misaligned, the increased effort does not pay off, as a reuse is technically possible, but at the same time the demand is missing. In the context of a supplier it is also much harder to define a long-term strategy independent from the customer (OEM).

If one compares the mechanisms of variablity representation in the solution space proposed by the research with the practice in the industry, it becomes clear that often a much more pragmatic approach is chosen. Rather than explicitly define variability in development artifacts, similar artifacts from previous projects are copied and evolved, the so-called *Clone&Own* approach [DRB+13]. This approach is very cheap for the current software project in the first step, but is often executed unsystematically and, despite its short-term added value, adds long-term dangers. The knowledge of a potential reuse of given variants is usually only given in the minds of experts from predecessor projects and will get lost over time or due to personnel changes. Furthermore, the maintenance of several clones that have evolved over time, but still share a common base, is difficult. In this respect, at the first moment the impression arises that through unsystematic *Clone&Own* efficient reuse can be achieved, but the actual benefits due to a established software product line are not achieved.

Within this thesis, the described context is identified at involved industry partners and approaches for a systematic *Clone&Own* and a step-wise migration to a software product line are formulated. In this thesis procedures for a gradual (extrinsic, interface-based, semantic) and automatic similarity analysis as well as for a database-based establishment and maintenance of a software product line are in focus.

Based on approaches of agile software product line engineering a pragmatic approach is proposed, which allows a supplier in the context of heterogeneous project landscapes to establish a software product line step-wise, while considering given norms of the automotive industry, like CMMI [Pau02] or ISO26262 [Hil12]. The approach is supported by several tools, which have been developed and used in cooperation with industrial partners during this thesis.

Danksagung

An dieser Stelle möchte ich mich bei allen lieben Menschen bedanken, die mich während meiner Promotion begleitet und unterstützt und somit maßgeblich zum Erfolg dieser Dissertation beigetragen haben.

Mein besonderer Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die Betreuung meiner Promotion, für die spannende und lehrreiche Zeit am Lehrstuhl, für die konstruktiven Diskussionen und für die Möglichkeit im industrienahen Umfeld promovieren zu können.

Des Weiteren danke ich Prof. Dr. Ina Schaefer für die Übernahme des Zweitgutachtens und die lehrreiche Zeit und Unterstützung während des Spes XT Forschungsprojektes.

Ebenso möchte ich mich bei Prof. Dr. Dr. van der Aalst für die Übernahme des Vorsitzes der Doktorprüfung und bei Prof. Ph. D. Lakemeyer für die Übernahme der praktische Prüfung bedanken.

Ich bedanke mich des Weiteren bei der FEV GmbH, der Daimler AG, der BMW Group, der dSPACE GmbH und der StreetScooter GmbH für die konstruktive Zusammenarbeit in erfolgreichen und sehr interessanten Industriekooperationen.

Besonders gilt in dieser Hinsicht mein Dank Dr. Johannes Richenhagen, Dr. Axel Schloßer, Jochen Pinnekamp, Dr. Günther Kessler, Dr. Philipp Orth, Christian Granrath, Hariharan Venkitachalam, Mohit Yadav (FEV GmbH), Peter Manhart (Daimler AG), Dr. Stefan Kriebel, Dr. Björn Weigold, Dr. Rolf Ebert, Jonas Karlsson (BMW Group), Markus Deppe, Dr. Jobst Richert (dSPACE GmbH) und Kai Kreiskoether.

Auch möchte ich mich natürlich bei allen Kollegen des Lehrstuhls bedanken. Besonders möchte ich mich bei Dr. Arne Haber, Dr. Andreas Wortmann, Dr. Pedram Nazari, Professor Dr. Jan Oliver Ringert, Michael von Wenckstern, Dr. Klaus Müller, Timo Greifenberg, Steffen Hillemacher, Imke Drave, Oliver Kautz, Matthias Markthaler, Dr. Alexander Roth, Dr. Markus Look und Deni Raco für die gute und erfolgreiche Zusammenarbeit in unterschiedlichsten Projekten und während zahlreicher Publikationen bedanken. Auch für das Korrekturlesen früher Fassungen meiner Dissertation und das zahlreiche konstruktive Feedback möchte ich mich an dieser Stelle noch einmal explizit bedanken.

Des Weiteren möchte ich mich auch bei der Egosoft GmbH bedanken, die es mir ermöglicht hat vor meiner Zeit am Lehrstuhl frühzeitig erste Berufserfahrungen in der sehr spannenden und herausfordernden Videospielindustrie zu sammeln und so den Grundstein für mein weiteres Schaffen zu legen. Dabei gilt mein besonderer Dank Bernd Lehahn, aber auch bei Christian Vogel, Andreas Itze, Martin Brenner, Matthias Haan und Stefan Hett möchte ich mich bedanken.

Mein größter Dank gilt meiner Familie, insbesondere meinen Eltern und meiner Freundin Janina, die mich immer und in allen Lebenslagen unterstützt haben und mich auch weiterhin unterstützen. Ohne euch wäre all dies nicht möglich gewesen!

Jüchen, Februar 2019
Christoph Schulze

Inhaltsverzeichnis

Kapitel 1

Einführung und Motivation

Innerhalb der letzten Jahrzehnte hat Software in der Automobilindustrie einen immer höheren Stellenwert erhalten [BKPS07, Bos14, SZC16, EF17, HBP+17]. Wie in Abbildung 1.1 dargestellt, hat sich der Prozentsatz der Funktionalitäten deutlich erhöht, die primär durch Software realisiert werden [CH01]. Auf Basis dieser Prognosen kann für das Jahr 2020 ein höherer Wertanteil der Software gegenüber der Hardware im Fahrzeug geschlussfolgert werden. Des Weiteren wird in Zukunft laut der *Automotive Special Interest Group* (SIG) mehr als 85% der Funktionalität durch Software realisiert werden [HDZS+15, FFLS08].

Schon zu Beginn dieses Millenniums entstanden 90% der Innovationen im Automobil im Elektronikbereich, 80% wiederum direkt durch Software [HKK04]. Die zukünftigen Herausforderungen für die Automobilindustrie sind sogar noch deutlicher in der Informatik-Domäne angesiedelt und werden den aktuellen Trend noch weiter verstärken [EF17]: Car2Cloud, E-Mobilität [SS15] und autonomes Fahren bilden die Eckpfeiler zukünftiger Entwicklungen und werden maßgeblich über die Vorherrschaft innerhalb der Automobilindustrie im nächsten Jahrzehnt mit

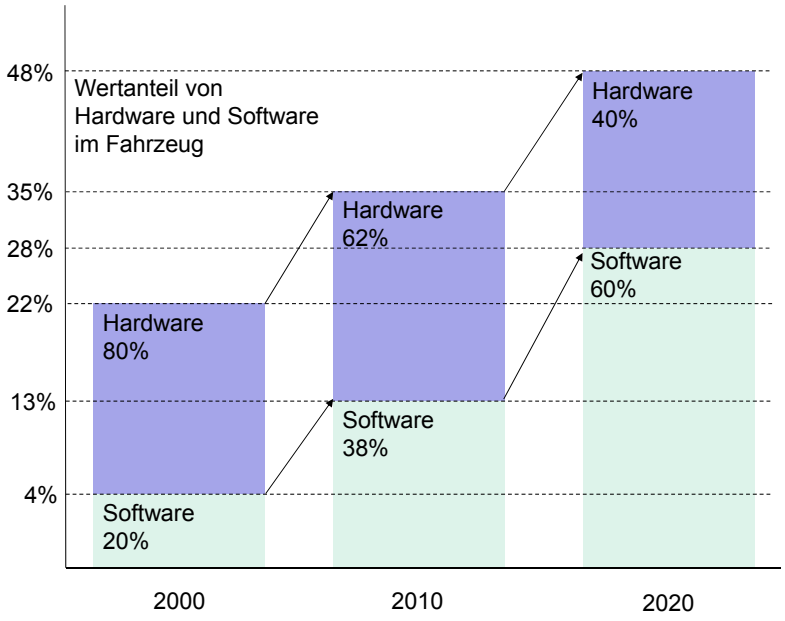


Abbildung 1.1: Steigende Bedeutung der Software im Automobilkontext [HKK04, CH01].

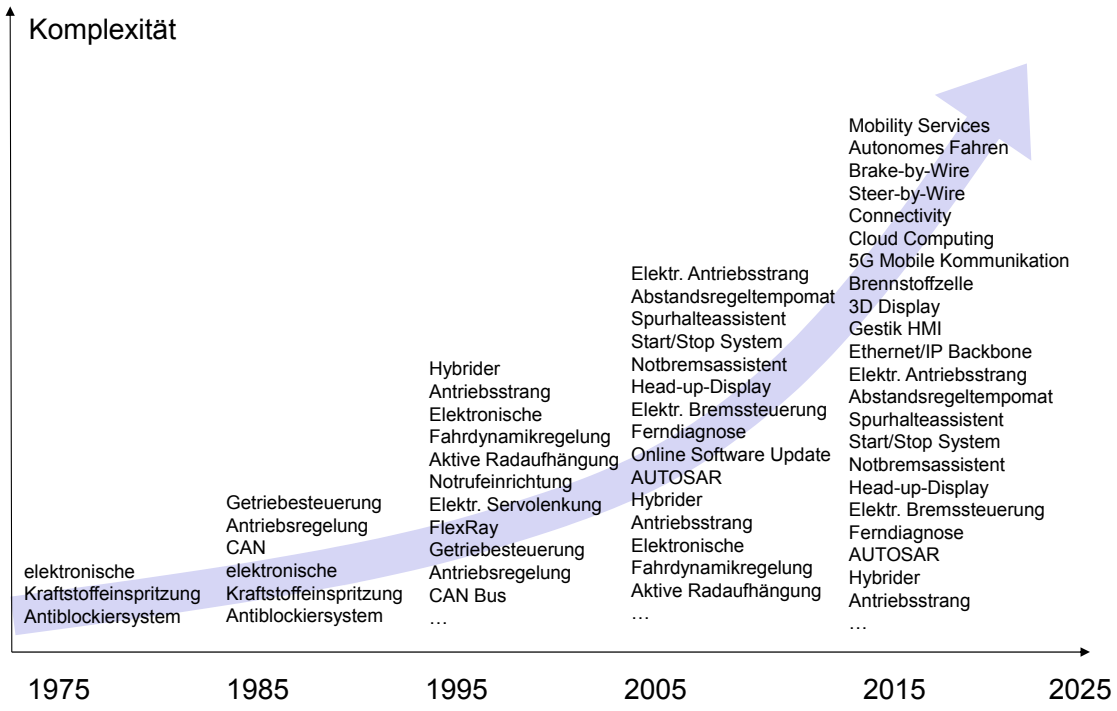


Abbildung 1.2: Komplexitätstreiber in der Automobilindustrie [EF17].

entscheiden [Cas17, EF17]. Diese Aspekte stellen deutliche Komplexitätstreiber dar, wie in Abbildung 1.2 aufgezeigt.

Dass durch diese relativ neuen Themenfelder der Fokus weiter in Richtung Software verschoben wird, macht auch die aktuelle Marktentwicklung deutlich: Firmen wie Google, Tesla oder Apple stellen im Kontext des autonomen Fahrens gegenüber den etablierten Automobilherstellern eine ernstzunehmende Konkurrenz dar: Kein Fahrzeug hat mehr autonom gefahrene Kilometer absolviert als Googles *Waymo*[1] (aktuell mehr als 1.5 Millionen Meilen). Durch *Car2Cloud* wird auch die Thematik Security im Automobilkontext immer relevanter [PBKS07]. Neben der Erweiterung des gegebenen Funktionsumfanges eines Automobils werden auch die Ansprüche an die Qualität der umgesetzten Funktionen von Jahr zu Jahr höher [DTK07]. Dies bezieht sich sowohl auf den zu realisierenden Leistungsumfang als auch die nachzuweisende Qualität der Umsetzung, gerade im Kontext sicherheitsrelevanter Funktionen [Bro06, KK09]. Entsprechende Normen innerhalb der Automobilindustrie (ISO26262 [Hil12], ISO25010 [GKFS14]) sichern eine kontinuierliche Verbesserung des Prozesses und des Ergebnisses [DTK07], fordern gleichzeitig aber kostenintensive Anpassungen und Erweiterungen [Ebe15].

Parallel dazu wird aber der Kosten- und Zeitdruck innerhalb der Automobilindustrie nicht geringer [Bro06, SS15]. Innerhalb dieses Umfeldes gilt es gerade auch für den Zulieferer systematisch neue Fachgebiete zu erschließen, die Qualität der umzusetzenden Produkte zu erhöhen

[1] `https://www.google.com/selfdrivingcar/`

und gleichzeitig die Kosten zu reduzieren. Da sich der Fokus immer mehr auf die Softwareentwicklung verschiebt, ist es gerade in diesem Kontext notwendig, die beschriebenen Ziele zu erreichen.

Innerhalb der Automobilindustrie hat sich die modellbasierte Entwicklung konsequent durchgesetzt [SEHV12, WL+11, SAL+15]. Diese wird auch durch eine größere Werkzeuglandschaft unterstützt, die sich primär im Kontext der Automobilindustrie entwickelt hat. Dabei werden Standards wie die UML (Unified Modeling Language) eher zur Modellierung von Systemsichten verwendet [WHR14], während Werkzeuge, wie Simulink oder ASCET zur Funktionsentwicklung dienen. Diese Werkzeuge und ihre zugrundeliegenden Modellierungssprachen sind über viele Jahre auf Grund neuer Anforderungen stetig gewachsen, wobei der Fokus meist auf Darstellung und Simulation von komplexen Regelungsabläufen lag. Mechanismen zur Wiederverwendbarkeit, gar objektorientierte oder aspektorientierte Ansätze finden nur zaghaft Einzug. In manchen Teildomänen wird in einer Fahrzeuggeneration nur 10% der Funktionalität verändert, gleichzeitig aber bis zu 90% der Software neu geschrieben [Bro06]. Dies kann unterschiedliche Gründe haben: Die Software wurde nicht hardwareunabhängig entwickelt und muss dementsprechend auf ein neues Steuergerät angepasst werden, entsprechende Partner haben gewechselt und ein direkter Zugriff auf den Code ist nicht gegeben oder die Funktionalität wurde in einem ersten Schritt unter solchem Zeitdruck entwickelt, dass eine saubere Neuentwicklung anstatt eines Refactorings sinnvoll erscheint.

In der Informatik wird der steigenden Software-Komplexität generell mit dem *Divide&Conquer* [Knu98] Prinzip entgegengewirkt. Nur durch die Aufteilung des gegebenen Systems in möglichst voneinander unabhängige Einheiten ist auf lange Sicht eine Testbarkeit, Wartbarkeit und Erweiterbarkeit des Systems möglich [SS15]. Des Weiteren spielt der Aspekt der Wiederverwendbarkeit zur Kostenreduktion eine primäre Rolle. Die Verwendung einer Produktlinie zur Reduktion der Kosten ist innerhalb der Automobilindustrie im Bereich der Hardwareentwicklung schon seit Jahren ein selbstverständlicher Prozess, während im Kontext der Softwareentwicklung oft pragmatisch vorgegangen wird: *Clone&Own*, die Kopie gegebener Strukturen zur Weiterverwendung (und Veränderung) ist innerhalb der Industrie ein weit verbreiteter Prozess, der aber größtenteils unsystematisch vollzogen wird [DRB+13].

Als Konsequenz findet keine systematische Wiederverwendung statt, kostenintensive Schritte der Verifikation müssen wiederholt werden, redundante Fehlererkennung und -behebung findet nacheinander oder auch parallel statt, da die gemeinsame Verbindung der Klone, beziehungsweise das Wissen um diese Verbindung, schnell verloren geht. Das Verfahren ist trotzdem innerhalb der Industrie auf Grund seiner intuitiven Zugänglichkeit sehr beliebt. Zudem ist keine langfristige Planung oder eine Synchronisation der Entwicklung innerhalb eines Projektes mit der Software-Produktlinie notwendig. Der Nutzen der kurzfristigen Wiederverwendbarkeit ist direkt erkennbar, die aufgeführten Nachteile dagegen werden erst auf lange Sicht spürbar.

Der beschriebene *Clone&Own*-Prozess steht im klaren Gegensatz zu deutlich strukturierteren Ansätze der Software-Produktlinienentwicklung aus der Forschung. Diese beschreiben meist einen Top-Down Ansatz, der sich durch eine sehr vorausschauende Arbeitsweise auszeichnet und eine Kostenreduktion ungefähr ab der dritten Wiederverwendung des Software-Produktes prognostiziert [PBL05].

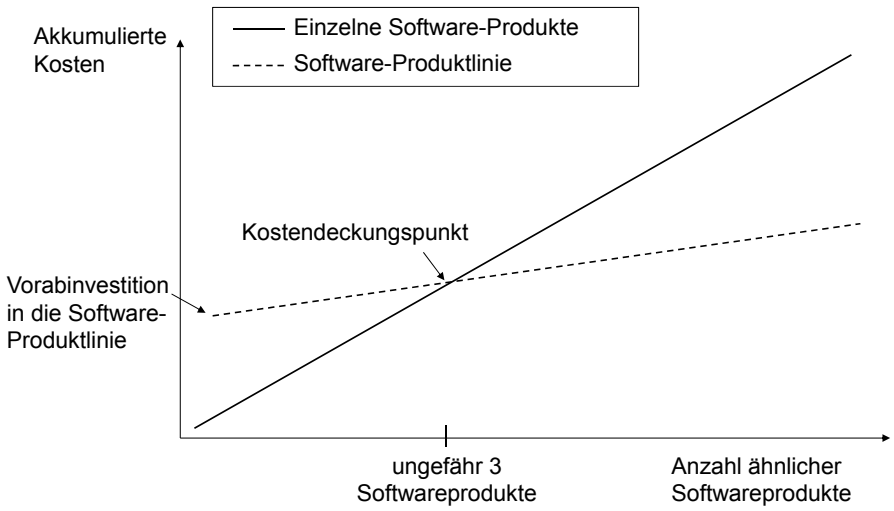


Abbildung 1.3: Kostendeckungspunkt bezüglich der Einführung einer Software-Produktlinie gegenüber einer rein produktgetriebenen Entwicklung [PBL05].

1.1 Herausforderungen

Die Entwicklung großer und komplexer Softwaresysteme lässt sich nur durch eine konsequente Zergliederung des Gesamtsystems in möglichst unabhängige und separat verifizierbare Software-Komponenten effizient realisieren. Um den Aufwand weiter zu reduzieren, ist es notwendig, diese Software-Komponenten möglichst so zu entwickeln, dass sie einen hohen Grad an Wiederverwendbarkeit erreichen. Insofern ist es ratsam, die Entwicklung einer Software-Produktlinie voranzutreiben, um den Aufwand für das jeweils einzelne Software-Produkt zu reduzieren und redundante Arbeitsschritte im Kontext der Softwareentwicklung zu vermeiden.

Diesem generellen Ansatz stehen in der alltäglichen Umsetzung folgende Probleme gegenüber: Erst einmal ist die Entwicklung einer wiederverwendbaren Software-Komponente immer aufwändiger, als die Umsetzung projektbezogener, aktueller Anforderungen. Der entsprechende Aufwand wird grob als dreifach so hoch angesehen [PBL05]. Dieser Zusammenhang ist auch in Abbildung 1.3 dargestellt. Auch wenn dieser Wert nicht präzise belegt ist und die Erfahrungen generell variieren, ist ein deutlicher Mehraufwand nicht von der Hand zu weisen.

Dieser Mehraufwand muss normalerweise in derselben Zeit umgesetzt werden, da entsprechende projektgetriebene Meilensteine die Entwicklung einer Software-Plattform nicht berücksichtigen. Auch kann nicht garantiert werden, dass in Zukunft das durch deutlichen Mehraufwand entwickelte Potential der Wiederverwendbarkeit wirklich ausgeschöpft wird. Der besten Planung zum Trotze können entweder neue Anforderungen entstehen, die vorher nicht in Betracht gezogen wurden oder politische Entscheidungen ändern die strategische Ausrichtung des Konzerns. Als Konsequenz wäre der erbrachte Mehraufwand als Verlust abzurechnen.

Als Zulieferer im Kontext der Automobilindustrie ist die strategische Ausrichtung einer Software-Produktlinie noch schwerer zu stabilisieren. Während OEMs bis zu einem gewissen

Grad in der Lage sind, eine einmal getroffene Strategie ungeachtet äußerer Umstände für einen Dreijahreszyklus aufrecht zu erhalten, ist der Zulieferer stärker auf die durch den OEM getriebene Nachfrage ausgerichtet. Diese kann er nicht kontrollieren, was eine stabile und langfristige strategische Ausrichtung und damit die Entwicklung einer rentablen Software-Produktlinie deutlich erschwert. Gleichzeitig bieten auch die innerhalb der Automobilindustrie üblichen Werkzeuge wenige ausgereifte sprachliche Mechanismen, um die Entwicklung wiederverwendbarer Software-Komponenten effizient umzusetzen.

Beide Aspekte stellen Gründe dar, weshalb in der Praxis ein pragmatischer und flexibler *Clone&Own*-Ansatz einer systematischeren, und langfristig effektiveren Software-Produktlinienentwicklung vorgezogen wird [DRB+13].

Die primäre Herausforderung, die sich durch den genannten Kontext der Automobilindustrie und dem speziellen Kontext eines Zulieferers innerhalb dieser Industrie im Bezug auf Entwicklung und Wartung einer Software-Produktlinie ergibt, ist grundsätzlich durch die heterogene und sich stetig weiterentwickelnde Projektlandschaft gegeben. Dabei spielen auch die zur Verfügung stehenden Werkzeuge und organisatorischen Strukturen mit Fokus auf das Tagesgeschäft als auch aktuell etablierte Mechanismen zur Wiederverwendung eine maßgebliche Rolle.

Im Folgenden werden einzelne Forschungsfragen definiert, die die generelle Aufgabenstellung der Entwicklung und Wartung einer Software-Produktlinie im Kontext heterogener Projektlandschaften auf vier Kernpunkte aufteilt.

- **RQ1**: Wie kann effizient und parallel zum stark kosten- und zeitgetriebenen Projektalltag auf Basis eines implizit gegebenen *Clone&Own*-Ansatzes eine Software-Plattform erstellt und weitergeführt werden, ohne dass dabei signifikanter Mehraufwand generiert wird und aktuelle Standards eingehalten werden?

- **RQ2**: Welche sprachlichen Mechanismen der gegebenen Werkzeuge innerhalb der Automobilindustrie können unter möglichst minimalen Modifikationen genutzt werden, um eine effiziente Entwicklung wiederverwendbarer Software-Komponenten zu ermöglichen?

- **RQ3**: Mit welchen automatisierten Mechanismen kann man einen solchen Prozess bezüglich Effizienz unterstützen?

- **RQ4**: Welche weiteren Werkzeuge sind notwendig, um eine Etablierung einer Software-Plattform zu unterstützen und deren Erhalt zu gewährleisten?

1.2 Struktur der Arbeit

Im Folgenden wird kurz auf die Struktur der Arbeit eingegangen.

Kapitel 2 Grundlagen Auf Grund der definierten Herausforderungen und Forschungsfragen wird in diesem Kapitel eine Übersicht über aktuelle Normen und Standards bezüglich der Softwareentwicklung in der Automobilindustrie gegeben. Dies ermöglicht ein grundsätzliches Verständnis der in dieser Domäne aktuell vorherrschenden Vorstellungen des Entwicklungsprozesses und deren Evolution. Auch wird herausgear-

beitet, welche Einhaltung von Normen für die Entwicklung juristisch bindend oder für die Projektakquise notwendig ist.

Im zweiten Teil dieses Kapitels wird auf die historische Entwicklung der Softwareproduktlinienentwicklung und deren mögliche Verflechtungen mit einer agilen Entwicklung eingegangen. Des Weiteren wird evaluiert, wie stark aktuell gelebte Ansätze in der Industrie von den Vorstellungen der Forschung abweichen und abschließend werden mögliche Beweggründe identifiziert. Dieses Kapitel eruiert Grundlagen, um maßgeblich die Forschungsfragen **RQ1** und **RQ2** beantworten zu können.

Kapitel 3 **Reaktive Software-Produktlinienentwicklung** Auf Basis des im vorherigen Kapitel skizzierten Standes wird in diesem Kapitel ein Prozess zur reaktiven Software-Produktlinienentwicklung definiert, der die genannten Probleme aufgreift und im Sinne einer agilen Entwicklung durch kurze Iterationen schrittweise eine Software-Produktlinie etabliert ohne dabei zu stark in das gegebene Tagesgeschäft einzugreifen. Dabei wird identifiziert, dass ein entsprechender Prozess möglichst durch automatisierte Ähnlichkeitsanalysen von extrinsischen, schnittstellen-basierten und semantischen Aspekten unterstützt werden muss, um den durch die Etablierung einer Software-Plattform initial gegebenen Mehraufwand möglichst gering zu halten. In diesem Kapitel werden erste Lösungen zu den Forschungsfragen **RQ1**, **RQ2** und **RQ3** entwickelt und dann in folgenden Kapiteln, wo notwendig, detaillierter ausgearbeitet.

Kapitel 4 **Ähnlichkeitsanalyse** Die Klonerkennung ist innerhalb der Informatik ein etabliertes Fachgebiet, das für die Identifikation effizienter und effektiver Ähnlichkeitsanalyseverfahren eine wichtige Grundlage bildet. Aus diesem Grunde wird in diesem Kapitel eine Übersicht über verschiedenste Ansätze der Klonerkennungsverfahren und deren Genauigkeit sowie Performanz gegeben. Abschließend wird der Stand der Forschung mit praktisch einsetzbaren Werkzeugen zur Klonerkennung im Kontext Simulink, ein Werkzeug, das in der Automobilindustrie zur Funktions- und Softwareentwicklung starke Verwendung findet, verglichen. Dieses Kapitel eruiert weitere Grundlagen auf Basis des im vorherigen Kapitel definierten Prozesses, um maßgeblich die Forschungsfrage **RQ3** zu beantworten.

Kapitel 5 **Strukturelle Ähnlichkeitsanalyse** Das Ähnlichkeitsanalyseverfahren wird in drei Teilschritte (extrinsisch, schnittstellen-basiert und semantisch) unterteilt, um auf diesem Wege den Suchraum schrittweise zu reduzieren und die Anwendung komplexerer Verfahren zu ermöglichen. In diesem Kapitel wird zuerst die extrinsische Analyse, die Klone auf Basis einer gemeinsamen Referenzarchitektur identifiziert, eingegangen. Die extrinische Analyse wurde im Kontext beteiligter Industriepartner angewendet und die Evaluation wird anschließend in diesem Kapitel diskutiert. In einem zweiten Schritt werden extrinsische Paare auf Basis ihrer Schnittstellendefinitionen analysiert, um Ähnlichkeiten zu identifizieren. Die schnittstellen-basierte Ähnlichkeitsanalyse fokussiert sich dabei auf die gerade in der Automobilindustrie durch zusätzliche Attribute hochwertigen Signaldefinitionen, die auch Aspekte wie zum

Beispiel die physikalische Einheit mit betrachten. Das Verfahren wurde wiederum im Kontext beteiligter Industriepartner evaluiert und die Ergebnisse werden im Abschluss dieses Kapitel besprochen. Dieses Kapitel konkretisiert erste Mechanismen zur Lösung der Forschungsfrage **RQ3**.

Kapitel 6 **Semantische Ähnlichkeit** Basierend auf einer extrinsischen und einer schnittstellenbasierten Ähnlichkeitsanalyse kann abschließend durch eine Analyse zugrundeliegender Verhaltensspezifikation, durch Modell, Code oder auch Testfälle, eine semantische Ähnlichkeit einzelner Ausgangssignale berechnet werden. Dazu wird die Verhaltensspezifikation in eine allgemeine Struktur überführt, um anschließend semantische Ähnlichkeiten zu bestimmen. Im Abschluss des Kapitels wird wiederum die Evaluierung der beschriebenen Verfahren diskutiert. Dieses Kapitel konkretisiert einen weiteren Mechanismus zur Lösung der Forschungsfrage **RQ3**.

Kapitel 7 **SimA Framework** Neben einer möglichst genauen Ähnlichkeitsanalyse ist die Aufbereitung der Ergebnisse zur effizienten Identifikation potentiell wiederverwendbarer Software-Komponentenvarianten unabdingbar. In diesem Kapitel wird das SimA (Similarity Analysis) Framework vorgestellt, das es ermöglicht beliebige Analyseverfahren zu kombinieren und deren Resultate entsprechend einer schrittweisen Eingrenzung des Suchraumes adäquat widerzuspiegeln. Das Kapitel geht genauer auf die Erweiterungsmöglichkeiten der Analyse und die Darstellung der Ergebnisse ein. Anschließend werden die durch die Anwendung im Kontext der beteiligter Industriepartner gesammelten Ergebnisse präsentiert und zusammengefasst. Dieses Kapitel stellt eine technische Lösung zur Unterstützung des in Kapitel 3 definierten Prozesses vor und verbindet die Teillösungen aus Kapitel 5-7 zu einer möglichen Lösung der Forschungsfragen **RQ1** und **RQ3**.

Kapitel 8 **Datenbankgestütztes Varianten-, Modell- und Signalmanagement** Um eine möglichst frühzeitige Wiederverwendung auch kleinerer Designelemente, wie Datentypen oder Signale, sowie die Weiterentwicklung einer Software-Plattform zu unterstützen, wurde im Zusammenarbeit mit der Firma dSPACE das Werkzeug SYNECT verwendet und für den genannten Einsatz modifiziert. Dabei sind aktuelle, durch die Automobilindustrie gegebene Ansprüche für einen Softwareentwicklungsprozess genauso berücksichtigt worden, wie Anforderungen, die sich durch das Langzeitziel der Etablierung und Fortführung einer Software-Produktlinie ergeben. Das Kapitel benennt diese generellen Anforderungen und bildet diese auf das Werkzeug SYNECT ab, um dort, wo nötig, Erweiterungen zu definieren. Der resultierende Entwicklungsprozess wird schrittweise an Beispielen illustriert und kann sowohl im Kontext der Software-Produkt- als auch der Software-Produktlinienentwicklung genutzt werden. Dieses Kapitel fokussiert sich auf die Lösung der Forschungsfrage **RQ4**.

Kapitel 9 **Zusammenfassung und Ausblick** Abschließend werden die gesammelten Erkenntnisse zusammengefasst und mögliche Erweiterungen des gegebenen Verfahrens aufgezeigt und diskutiert.

Kapitel 2

Grundlagen

Im Folgenden wird zuerst auf die in der Automobilbranche aktuell üblichen Prozessstandards und Normen zur Softwareentwicklung und deren historischen Ursprung eingegangen, um anschließend den Kontext eines Zulieferers in der Automobilindustrie besser verstehen zu können. Daraus werden Anforderungen für die Software-Produktlinienentwicklung abgeleitet. Dabei wird auch betrachtet, inwieweit Einflüsse der agilen Entwicklung in der Automobilindustrie gegeben oder gewünscht sind, um den heutigen Herausforderungen der Branche begegnen zu können.

In einem weiteren Schritt werden Grundlagen der Software-Produktlinienentwicklung und auch deren Evolution betrachtet. Wiederum wird dabei auch ein Augenmerk auf mögliche agile Einflüsse und gegebene Kritikpunkte der aktuell definierten Herangehensweisen gelegt. Des Weiteren wird der Stand der Forschung mit dem Stand der Industrie abgeglichen, um mögliche Unterschiede aufzudecken und gleichzeitig zu identifizieren, warum entsprechende Ansätze noch nicht oder nur teilweise den Weg in die Praxis gefunden haben.

2.1 Softwarentwicklungsprozesse in der Automobilindustrie

Im Folgenden wird eine grobe Übersicht auf gegebene Entwicklungsstandards und -normen innerhalb der Automobilindustrie gegeben. Dabei wird auch kurz auf die historische Entwicklung eingegangen, wie in Abbildung 2.1 dargestellt.

Beginnend im Jahre 1986 entwickelte das *Software Engineering Institute* (SEI) ein Reifegradmodell, das Unternehmen bei der Weiterentwicklung ihrer Prozesse und der Bewertung der Prozessstrukturen potentieller Unterauftragnehmer unterstützen soll [Pau02, REF+07]. Diese Arbeiten resultierten bereits 1987 in ersten Ergebnissen [HS89, Hum88].

Nach mehreren Jahren der Anwendung und Verbesserung wurde das initiale Reifegradmodell in das *Capability Maturity Model for Software* (CMM) [Pau02, PCCW93, PWC+94] überführt. Dieses wurde vom amerikanischen Verteidigungsministerium in Auftrag gegeben. Das Modell ermöglicht durch Evaluierung der einzelnen Prozessaspekte eines Unternehmens deren Softwareentwicklungsprozess in eine von fünf Stufen einzuordnen. Eine Verbesserung der Produktivität und Qualität sowie eine verlässlichere Zeit- und Budgetplanung konnten durch Verwendung des CMM in vielen Unternehmen nachgewiesen werden [GGK06].

Zeitgleich fanden ähnliche Entwicklungen von Prozessreife-Frameworks durch unterschiedliche Unternehmen, wie z.B. *British Telecom* oder *Bellcore* statt [REF+07].

Auf Grund der schnellen Verbreitung solcher Ansätze wurde im Jahre 1993 beschlossen, die

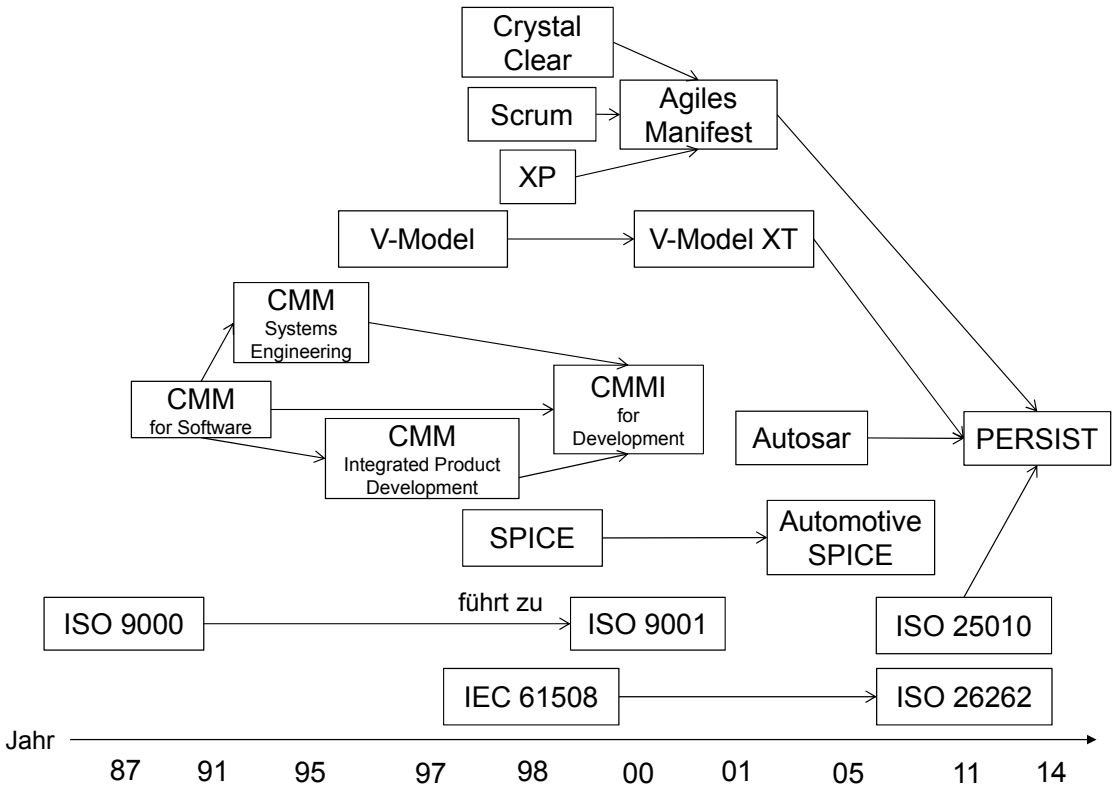


Abbildung 2.1: Entwicklungsstandards und -normen innerhalb der Automobilindustrie mit Bezug zur Software.

Entwicklung eines internationalen Standards in die Wege zu leiten. Der Standard ISO/IEC 15504 ist das Ergebnis des Projektes (*Software Process Improvement and Capability dEtermination*) SPICE und dient zur Bewertung von Softwareentwicklungsprozessen [REF+07]. Die initialen Arbeiten des SPICE Projektes waren bereits 1995 abgeschlossen, während die erste offizielle Version des Standards 1998 zur Verfügung stand.

Des Weiteren wurden 1995 das Systems Engineering CMM und 1997 das Integrated Product Development CMM entworfen, die 2000 zum heute verwendeten Reifegradmodell CMMI (Capibility Maturity Model Integrated) zusammengeführt wurden. Diese Zusammenführung adressiert mehrere Fachgebiete und ermöglicht eine domänenspezifische Anpassung auf Basis der zugrundeliegenden Quellenmodelle [ATC03].

Der Qualitätsmanagementstandard ISO 9001 [Jen95] definiert Anforderungen an ein Qualitätsmanagementsystem. Dieser ist losgelöst von der Softwareentwicklung definiert und beschreibt ein Qualitätsmanagementsystem unabhängig von der Domäne. Der Standard fordert eine Qualitätspolitik mit zugehöriger Leitung der Qualitätssicherung beim Zulieferer, ein dokumentiertes Qualitätsmanagementsystem sowie die Unabhängigkeit von Entwickler und Prüfer [DW99]. Im Jahre 2006 haben schon 900.000 Organisationen in 170 Ländern die ISO 9001

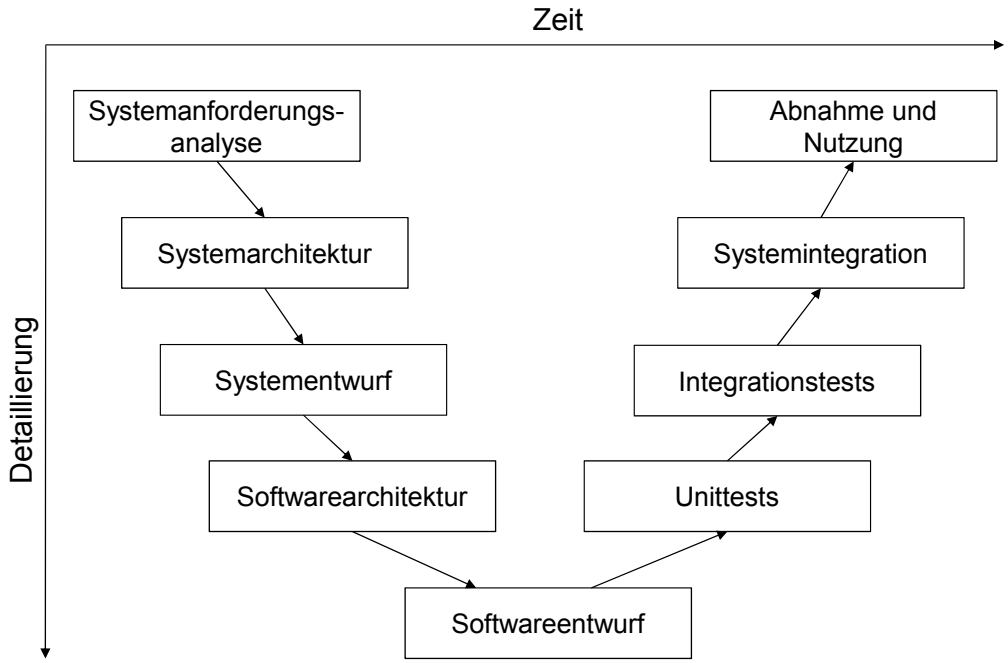


Abbildung 2.2: V-Modell 97.

umgesetzt [LT10, SSR09], im Jahre 2015 ist diese Zahl allerdings nur noch auf 1.033.936 angestiegen und grundsätzlich ist aktuell eine rückläufige Tendenz wahrzunehmen [ISO06]. Initial war die ISO 9001 ein europäisches Phänomen und wurde in den folgenden Jahren in die anderen Kontinente exportiert [SSR09]. Mehrere Studien belegen, dass dabei zertifizierte Firmen unter anderem ein schnelleres Umsatz- als auch Mitarbeiterwachstum als auch höhere Gehälter vorweisen [WB07, TK06, LT10]. Dabei ist allerdings auch die externe Motivation zur Zertifizierung durch die Anforderung des Kunden gegeben, die eine zugehörige Umsatzsteigerung bedingen kann [SSR09].

Ein in Deutschland stark verbreitetes Prozessmodell zur Softwareentwicklung ist das V-Modell [BR05]. Es wurde initial vom Bundesministerium der Verteidigung und des Bundesministerium für Inneres beauftragt und bildet die Grundlage für öffentliche Aufträge. Im Jahre 1997 wurde das V-Modell 97 veröffentlicht und im Jahre 2005 durch den Nachfolger V-Modell XT ersetzt [BR05, DW99]. Der Name wurde dabei durch die Form des Prozessmodells, das in Abbildung 2.2 dargestellt ist, geprägt.

Dort werden die einzelnen Abstraktionsebenen des Entwurfs und deren Validierung jeweils auf einer Ebene dargestellt. Der Detallierungsgrad wird von oben nach unten immer größer, während der zeitliche Ablauf von links nach rechts voranschreitet. Dies führt insgesamt zur bekannten V-Form.

Das V-Modell XT definiert eine Menge an *Vorgehensbausteinen* und *Projektdurchführungsstrategien*, um während eines projektspezifischen Tailorings (XT = eXtreme Tailoring) das Modell an den Kontext anzupassen [BR05]. Während *Vorgehensbausteine* die Produkte, Aktivitä-

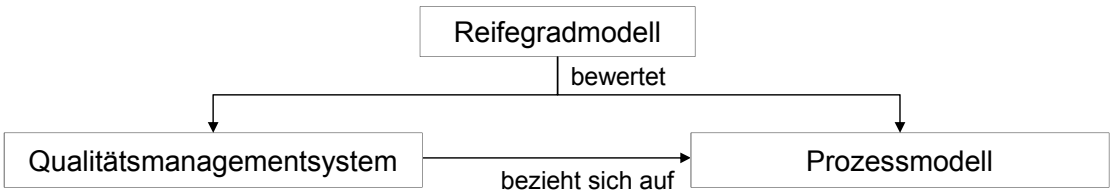


Abbildung 2.3: Zusammenhang zwischen Reifegradmodell, Qualitätsmanagementsystem und Prozessmodell [DW99].

ten und Rollen beschreiben, definieren *Projektdurchführungsstrategien* den zeitlichen Ablauf. Das Ursprungsmodell ist dabei basierend auf dem Wasserfallmodell entworfen worden [DW99], während das V-Modell XT inkrementelle, komponentenbasierte und agile Systementwicklung unterstützt [BR05].

Der Zusammenhang zwischen Reifegradmodellen, wie CMM, einem Qualitätsmanagementsystem, wie durch die ISO 9001 definiert und einem Prozessmodell, wie dem V-Modell, ist in Abbildung 2.3 dargestellt. Das Reifegradmodell bewertet unter anderem den Reifegrad des Qualitätsmanagementsystems sowie des Prozessmodells. Das Qualitätsmanagementsystem bezieht sich wiederum auf den Prozess und sorgt für eine kontinuierliche Verbesserung des Prozesses.

Automotive SPICE ist ein von dem Standard ISO/IEC 15504 [WDWD08] abgeleitetes Prozessmodell, um den aktuellen Herausforderungen der Automobilindustrie durch Reduktion der Entwicklungszeiten und -kosten zu begegnen [HDZS+15]. Dieses wurde initial 2005 publiziert. Der Standard wurde schon 2006 von den meisten OEMs eingesetzt [SK06].

Automotive SPICE stellt dabei eine domänenspezifische Ausprägung der durch die ISO/IEC 15504 vorgegebenen Praktiken dar. Weitere domänenspezifische Ausprägungen sind zum Beispiel durch Medi SPICE, Enterprise SPICE oder Banking SPICE gegeben [HDZS+15].

Dabei werden in Automotive SPICE nur 26 der 48 Prozessebereiche aus der ISO/IEC 15504 abgedeckt und fünf weitere hinzugefügt. Diese Auswahl fokussiert sich dabei auf den Kontext der Embedded-Systems und vernachlässigt bewusst andere Aspekte, wie zum Beispiel die Umsetzung typischer Client/Server-Systeme [HDZS+15].

Ein Vergleich zwischen der ISO 9001 und ISO/IEC 15504 ergab, dass Unternehmen, die durch die ISO 9001 zertifiziert sind, durch SPICE höher eingestuft werden als Unternehmen welche nicht zertifiziert sind [JH01]. Ein Abgleich zwischen ISO 9001 und CMM ergab ähnliche Resultate: zertifizierte Unternehmen erfüllen fast alle Ziele für eine Stufe 2 Zertifizierung und viele Ziele für eine Stufe 3 Zertifizierung [Pau93].

SPICE und CMMI decken größtenteils dieselben Themengebiete ab und eine 1:1-Abbildung der Prozesskategorien sowie der Reifegrade ist möglich [EPA+10]. Während aber CMMI Richtlinien und bewährte Methoden auch im Kontext der Implementierung aufgreift, ist dieser Themenbereich von SPICE nicht abgedeckt. Auf der anderen Seite werden sieben Themenbereiche, die in SPICE detailliert beschrieben sind, durch CMMI nur partiell abgedeckt [EPA+10]. Umfragen in der IT-Branche haben ergeben, dass dort hauptsächlich CMMI verwendet wird, da dieser Standard frei zur Verfügung steht und Entwickler entsprechende Schulungen vorweisen können [EPA+10].

Für die Entwicklung sicherheitsrelevanter elektronischer Systeme in Kraftfahrzeugen wurde 2011 die ISO 26262 auf Basis der ISO 61508 definiert [Hil12]. Diese erfordert im Zuge einer Gefährdungs- und Risikoabschätzung die Klassifizierung aller Anforderungen in die Stufen *QM*, *ASIL A*, *ASIL B*, *ASIL C* oder *ASIL D*. *QM* stellen dabei nicht sicherheitsrelevante Anforderungen dar, während *ASIL D* die höchste Stufe beschreibt. Während für *QM* durch die ISO 26262 keinerlei Maßnahmen vorgeschrieben sind, werden für die folgenden Stufen schrittweise weitere notwendige Prozessschritte oder Nachweise für u.a. Spezifikation, Validierung oder Tooling hinzugefügt. Die Teile 4,5 und 6 der ISO 26262 definieren dabei Entwicklungsprozesse auf System-, Hardware- und Softwareebene, die als geschachtelte V-Modelle dargestellt sind.

Eine weitere zu beachtende Norm ist die ISO 25010 [GKFS14], die Qualitätsmerkmale der Software auf Basis der Charakteristika *Funktionstauglichkeit*, *Performanz*, *Kompatibilität*, *Portabilität*, *Zuverlässigkeit*, *Sicherheit*, *Wartbarkeit* und *Anwendbarkeit* definiert. Diese übergeordneten Charakteristika werden genutzt, um weitere feingranularere Kriterien abzuleiten, die für eine konkrete Bewertung der Software verwendet werden können [Ric14].

Die bisher vorgestellten Reifegradmodelle und Normen stellen Rahmenwerke dar, um die Qualität eines Entwicklungsprozesses in unterschiedlichen Kontexten bewerten zu können. Dabei sind zwischen den einzelnen Normen und Modellen sowohl historische Verknüpfungen als auch inhaltliche Überschneidungen gegeben.

Die Tendenz zur Bewertung von Entwicklungsprozessen führte dabei zu einem stetig wachsenden Bedarf an detaillierten Nachweisen der einzelnen Aspekte und zu einem initialen Lernaufwand. Dieser Bedarf droht dabei die durch den verbesserten Prozess erlangten Effizienzvorteile durch einen organisatorischen Mehraufwand wieder zu reduzieren oder auch in Ineffizienz umzukehren [Han10]. Des Weiteren bilden die Prozessmodelle eine starre Durchführungsabfolge ab, die dem ursprünglichen Wasserfallmodell immer noch stark ähnelt [Han10].

Auf Grund der aufgeführten Defizite stellt die agile Entwicklung[1] seit 2001 einen entsprechenden Gegenpol dar, die Individuen und Interaktionen, lauffähige Software, Zusammenarbeit mit dem Kunden und die Reaktion auf Veränderung stärker gewichtet als Prozesse und Werkzeuge, umfassende Dokumentation, Vertragsverhandlungen und das Befolgen eines Planes. Am Ende des 20. Jahrhunderts gab es mehrere agile Prozessmodelle, wie zum Beispiel Extreme Programming (XP) [Bec00], Crystal Clear [Coc04] oder Scrum [Sch04]. Nach der Definition des agilen Manifestes wurden dieser der agilen Prozessfamilie zugeordnet und finden heute in der IT-Branche rege Anwendung. Testgetriebene Entwicklung [Bec02] und kontinuierliche Integration sind dabei zwei der am stärksten verbreiteten Teilmethoden der agilen Entwicklung [GPM08]. Die Automobilindustrie hat sich dagegen erst langsam mit agilen Konzepten angefreundet und innerhalb der Beauftragungsstruktur zwischen OEMs und Zulieferern sowie der Abnahme durch Zertifizierungsstellen (zum Beispiel im Kontext der ISO 26262), sind die aufgeführten Normen und Reifegradmodelle bis zum heutigen Tage aktuell und notwendig. Agile Entwicklung spielt im Kontext aktueller Herausforderungen auch in der Systementwicklung eine immer größere Rolle, um deren Effizienz zu steigern [WFS13].

Die schon in Kapitel 1 aufgeführten aktuellen Herausforderungen der Automobilindustrie und die deutlich anwachsenden Softwareanteilen stellen die Domäne aber vor die notwendige Verkürzung von Entwicklungszeiten und den Umgang mit wandelbaren Anforderungen. Insofern ist

[1] `www.agilemanifesto.org`

es ratsam, agile Aspekte in die Verbesserung der Prozesse einfließen zu lassen, ohne die Zertifizierung bestehender Normen (solange notwendig) zu gefährden. In Konsequenz ist es wichtig, zu identifizieren, bis zu welchem Grad agile Prozessmodelle aktuellen Anforderungen der Automobilindustrie, wie *CMMI*, *SPICE* oder *V-Model XT* genügen.

Automotive SPICE bietet zum Beispiel genügend Freiraum, um die Einbindung agiler Methoden zu ermöglichen [MZDH16]. Dabei ermöglichen agile Methoden den Aufwand für Projektplanung, das Änderungsmanagement und der Teamkommunikation zu reduzieren, ohne dass darunter die Kundenzufriedenheit oder die Produktqualität leiden [WFS13].

Eine Integration agiler Methoden in den Automobilkontext bringt allerdings auch einige Herausforderungen mit sich. Während zum Beispiel gängige IT-Projekte durch wenige Personen aus ähnlichen Fachbereichen realisiert werden, sind bei der Fahrzeugentwicklung oft mehr als 50 Personen beteiligt, die sowohl unterschiedlichen Fachbereichen als auch Firmen oder Abteilungen zugeordnet sind. Insofern muss immer abgewogen werden, in welchem Kontext welche agile Methoden Verwendung finden können. Welge [WFS13] definiert auf Basis von Smith [Smi07] Tendenzen, die eine agile oder eine traditionelle Entwicklung befürworten, wie in Abbildung 2.4

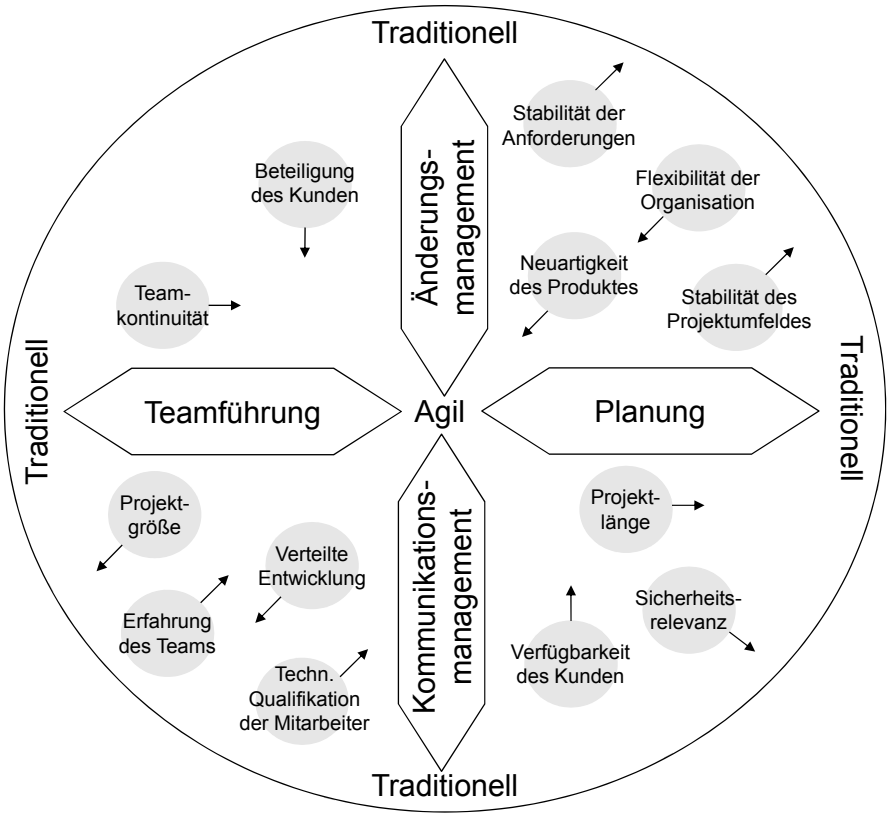


Abbildung 2.4: Übersicht einzelner Tendenzen, die den Einsatz agiler Methoden befürworten oder verhindern [WFS13, Smi07].

dargestellt. Dabei werden *Teamkontinuität*, *Beteiligung des Kunden*, *Neuartigkeit des Produktes*, *Flexibilität der Organisation*, *Verfügbarkeit des Kunden*, *technische Qualifikation der Mitarbeiter* und *Erfahrung des Teams* als Aspekte gewertet, welche eine agile Entwicklung begünstigen, während *Stabilität der Anforderungen*, *Stabilität des Projektumfeldes*, *Sicherheitsrelevanz*, *verteilte Entwicklung* und *Projektgröße* eine traditionelle Entwicklung begünstigen.

Auch in [Boe02, BT03] wird deutlich auf die notwendige Qualifikation und Erfahrung der Mitarbeiter zur erfolgreichen Anwendung agiler Methoden verwiesen.

Weitere Ansätze, die versuchen agile Methoden in traditionelle Prozesse zu integrieren sind in [PHV17, NMM05] gegeben. Dabei wird auch verstärkt auf die Notwendigkeit auf sich stetig ändernde Anforderungen eingegangen [PHV17]. Diese entstehen hauptsächlich auf Basis von kürzeren Lebenszyklen, unklaren oder unvollständigen Anforderungen [HPVM16], Verbesserungen des Designs [VPH16] sowie Termindruck [HMDZ08]. Um diesen häufigen Änderungen der Anforderungen Herr zu werden, wurde im Sinne einer agilen Entwicklung der Kunde stärker in das Projekt integriert, um die Qualität der Anforderungen zu erhöhen. Des Weiteren wurde die Anzahl der umsetzbaren Anforderungsänderungen pro Entwicklungszyklus eindeutig festgelegt, und die Umsetzung in Absprache mit dem Kunden priorisiert [PHV17]. Diese und weitere Anpassungen führten zu einem Effizienzgewinn von 20%.

Der Einsatz agiler Entwicklungsmethoden und einer szenario-basierten Regressionssimulation erlaubten eine schnelle Reaktion auf Kundenwünsche und führten im CarOLO-Projekt zu einer erfolgreichen Teilnahme an der *2007 DARPA Urban Challenge* [BR10]. Auch in der Lehre konnten Ansätze der agilen und der modell-getriebenen Entwicklung zur Implementierung von Robotikanwendungen erfolgreich eingesetzt werden [RRSW17]. Dieser Ansatz basiert auf der Verschmelzung von agilen Ansätzen und der UML, wie in [Rum17] und [Sch12] beschrieben.

Zusammenfassend befinden sich die Systementwicklung und auch die Automobilindustrie im Speziellen aktuell im Wandel und versuchen agile Aspekte, wo möglich, in ihre Prozesslandschaft zu integrieren. Dabei ist es allerdings notwendig, auf die speziellen Begebenheiten einzugehen, wie zum Beispiel die Entwicklung sicherheitsrelevanter Systeme oder die starke organisatorische Trennung einzelner Arbeitsschritte. Aktuelle Normen wie CMMI, SPICE oder ISO 26262 fordern dabei unter anderem eine saubere Architektur im Sinne der Übersichtlichkeit, geringer Abhängigkeit und Testbarkeit sowie die Nachverfolgbarkeit von Anforderungen über den gesamten Entwicklungszyklus hinweg und ein etabliertes Konfigurations-, Release- und Changemanagement.

Eine effiziente Methode zur Software-Produktlinienentwicklung muss folglich auch den gegebenen Standards genügen können und gleichzeitig möglichst agile Aspekte, wo sinnvoll, integrieren.

PERSIST (Powertrain control architecture Enabling Reusable Software development for Intelligent System Tailoring) [RPS14], ein Software-Architektur-Entwicklungsrahmenwerk verbindet beide Welten auf der Ebene der Softwareentwicklung.

Definition 2.1 *PERSIST. PERSIST (Powertrain control architecture Enabling Reusable Software development for Intelligent System Tailoring) [RPS14] ist ein Softwarearchitekturentwicklungsrahmenwerk, welches auf Basis des Standards AUTOSAR [Bun11] und der komponentenbasierten Softwareentwicklung [CLWK00] Richtlinien zur Definition einer Softwarearchitektur beinhaltet. Dabei besteht eine Architektur aus hierarchisch gegliederten Kompositionen, welche*

als Blätter Software-Komponenten enthalten. Diese Software-Komponenten bestehen wiederum aus Funktionen, welche genau einer Software-Komponente zugeordnet werden können. Dieser Zusammenhang ist in Abbildung 2.5 dargestellt. Neben Richtlinien zur Architekturdefinition beinhaltet PERSIST Ansätze aus der agilen Entwicklung, um eine kontinuierliche Integration zu

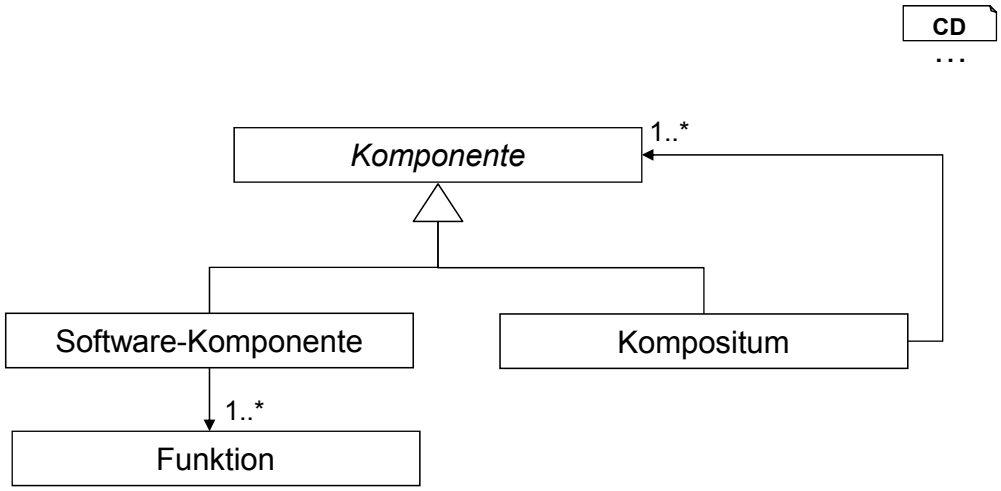


Abbildung 2.5: Aufbau einer PERSIST-Architektur.

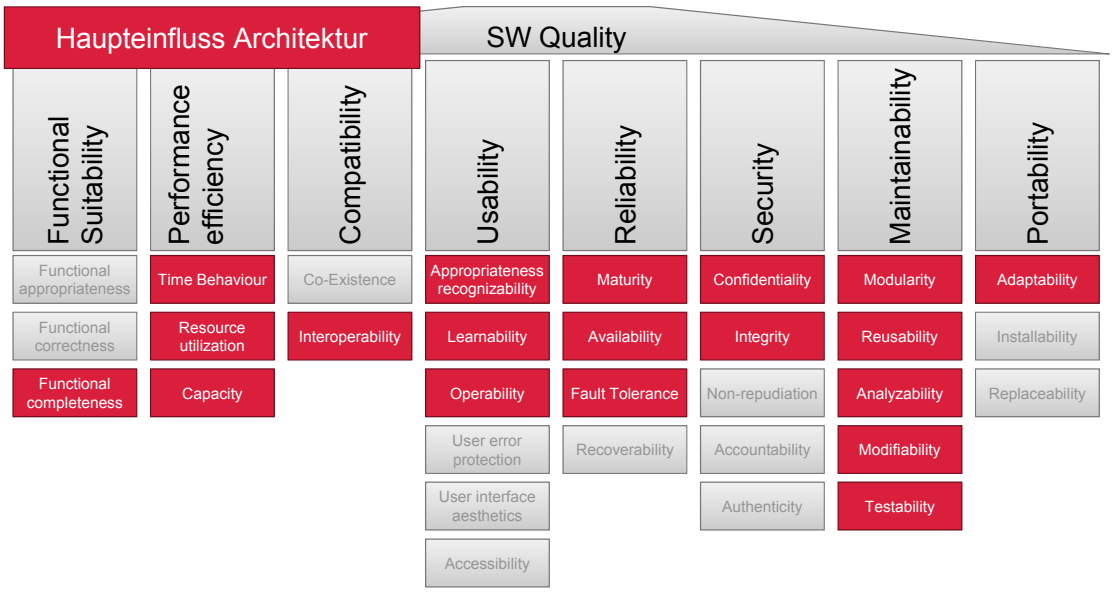


Abbildung 2.6: Qualitätsmerkmale der ISO25010, die durch PERSIST adressiert werden können [Ric14].

ermöglichen. In einem weiteren Schritt identifiziert PERSIST auf Basis der ISO 25010 Qualitätskriterien, die durch die Softwarearchitektur beeinflusst werden und leitet daraus eine Menge an Metriken ab, die durch ein zugehöriges CI-Rahmenwerk kontinuierlich überprüft werden können (siehe Abbildung 2.6) [Ric14] .

PERSIST legt fest, dass *Software-Komponenten* möglichst entsprechend dem physikalischen Layout des Systems zugeordnet werden können und somit eine *Software-Komponente* alle *Funktionen* enthält, die einer physikalischen Komponente zugeordnet werden kann. Auch die Definition der Schnittstelle von *Software-Komponenten* und *Funktionen* ist durch Richtlinien geregelt.

Definition 2.2 *PERSIST-Schnittstelle. PERSIST definiert SenderReceiver- und ClientServer-Schnittstellen [Bun11], wie in Abbildung 2.7 dargestellt. Die Signale einer Schnittstelle sind durch die Attribute Name, logischer Wertebreich (Range), Länge (Width), physikalische Einheit (Unit), sowie Datentyp (Datatype) beschrieben. Des Weiteren beschreibt das Attribut Label die genauere Art des Signals. Eingehende Signale sind entweder Eingänge, Konstanten oder Para-*

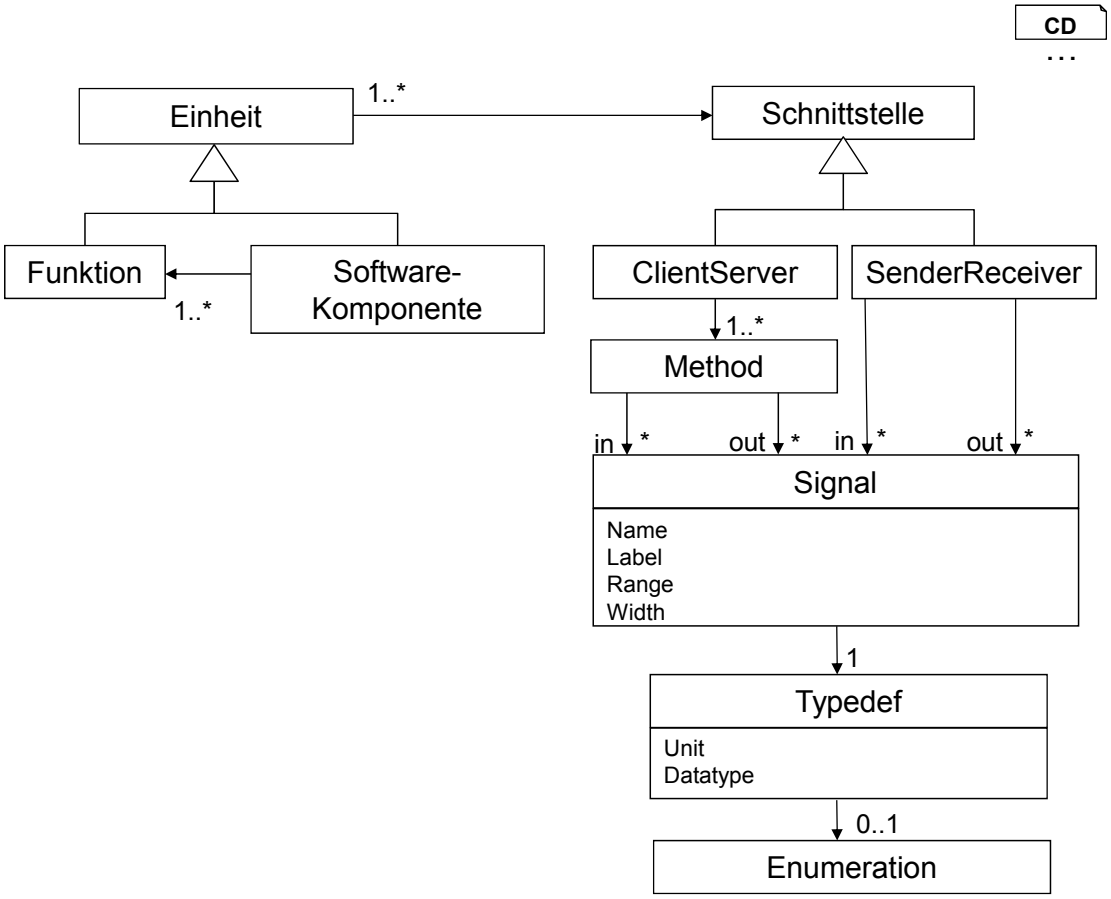


Abbildung 2.7: Schnittstellendefinition in PERSIST.

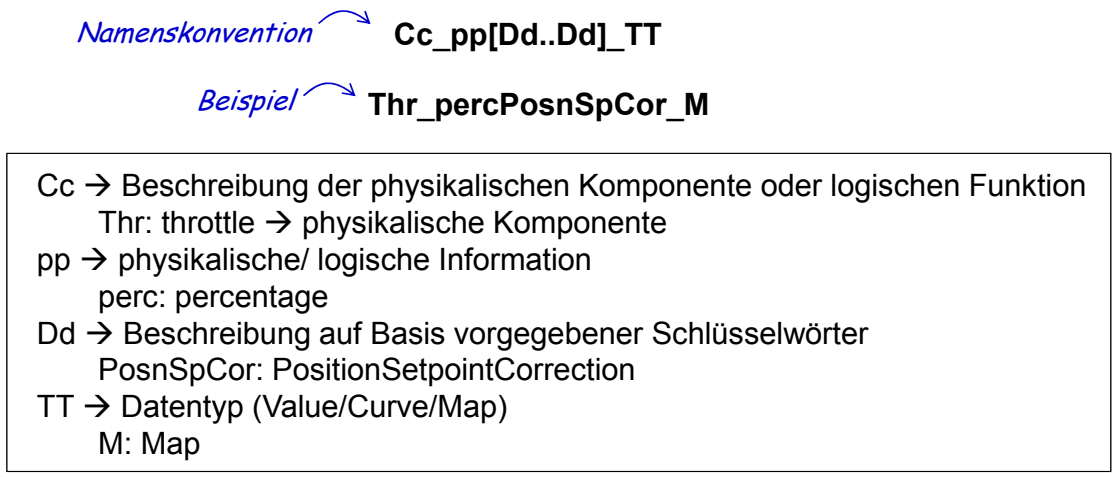


Abbildung 2.8: PERSIST-Signalnamenskonvention und zugehöriges Beispiel.

meter, während ausgehende Signale Ausgänge oder Messpunkte darstellen können.

Die Namensgebung für Signale ist dabei durch eine Signalnamenskonvention definiert, welche sowohl physikalische, funktionale als auch technische Aspekte des Signals im Namen abbildet. Diese ist in Abbildung 2.8 dargestellt. Die Namenskonvention verwendet dabei Abkürzungen vorgegebener Schlüsselwörter.

Die PERSIST-Signalnamenskonvention verlangt als Präfix ein Kürzel, welches die Quelle des Signals eindeutig identifiziert. Unter der Annahme, dass jede Quelle in einer Architektur nur einmalig existiert und somit eindeutig ist, ist jeder PERSIST-Signalname eindeutig.

Definition 2.3 *PERSIST-Autoconnect. Unter der Annahme dass jede physikalische Komponente oder logische Funktion innerhalb der PERSIST-Architektur eindeutig als Quelle identifizierbar ist, sind auf Grund der PERSIST-Namenskonvention alle Signalnamen eindeutig. Dies erlaubt eine automatische Verbindung aller Signale unterschiedlicher PERSIST-Schnittstellen derselben Hierarchiestufe auf Basis ihres Namens, wobei ausgehende Signale mit allen eingehenden Signalen desselben Namens verbunden werden.*

Ein ähnlicher Mechanismus ist auch in der Sprache MontiArc [HKRR11] gegeben.

Im Sinne einer agilen Entwicklung verwendet PERSIST kurze Entwicklungszyklen, in denen eine kleine Menge an Software-Komponenten und Funktionen entsprechend des V-Modells geplant, realisiert und verifiziert werden. Die Verifikation wird dabei deutlich durch das CI-Rahmenwerk unterstützt. PERSIST wurde seit seiner Einführung im Jahre 2014 in vielen unterschiedlichen Projekten erfolgreich eingesetzt [RPS14, VRP15, RVSP15]. PERSIST stellt ein Beispiel dar, gegebene Standards der Automobilindustrie mit einzelnen Aspekten der agilen Entwicklung zu verbinden und somit in kürzeren Entwicklungszeiten qualitativ hochwertige Ergebnisse zu produzieren. Die durch PERSIST auf Basis von AUTOSAR und der ISO 25010 eingeführten Standards auf der Ebene der Architektur- und Funktionsentwicklung (Simulink)

stellen dabei einen ersten wichtigen Schritt zur Wiederverwendbarkeit und damit zur Entwicklung einer Software-Produktlinie dar.

Im Folgenden wird nun auf die Grundlagen der Software-Produktlinienentwicklung und auch entsprechende Einflüsse aus der agilen Entwicklung eingegangen, die als weitere Basis für den in Kapitel 3 definierten Prozess zur reaktiven Software-Produktlinienentwicklung wichtig sind.

2.2 Software-Produktlinienentwicklung

Im Folgenden werden grundlegende Begrifflichkeiten zur Software-Produktlinienentwicklung eingeführt, die im weiteren Verlauf Verwendung finden werden.

Definition 2.4 *Software-Produktlinienentwicklung. Software-Produktlinienentwicklung (SPLE) ist ein Paradigma zur Entwicklung von Software-Produkten für eine spezifische Domäne unter Verwendung einer wiederverwendbaren Software-Plattform[PBL05].*

Dieses Paradigma ermöglicht verschiedene angepasste Produkte auf effiziente Weise auszuleiten und somit deutlich an Entwicklungskosten einzusparen [PBL05].

Das SPLE Paradigma teilt die Entwicklung in zwei separate Phasen ein (Abbildung 2.9):

Definition 2.5 *Domänenentwicklungsphase. Während der Domänenentwicklungsphase (DE) werden die Gemeinsamkeiten der Domäne identifiziert und basierend auf dieser Analyse eine zugehörige Software-Plattform umgesetzt. Während dieser Umsetzung werden die gemeinsamen und verschiedenen Aspekte der Software-Produktlinie definiert [PBL05].*

Artefakte der DE beinhalten variable Aspekte, die in einem weiteren Schritt aufgelöst werden müssen, um ein Software-Produkt zu erhalten.

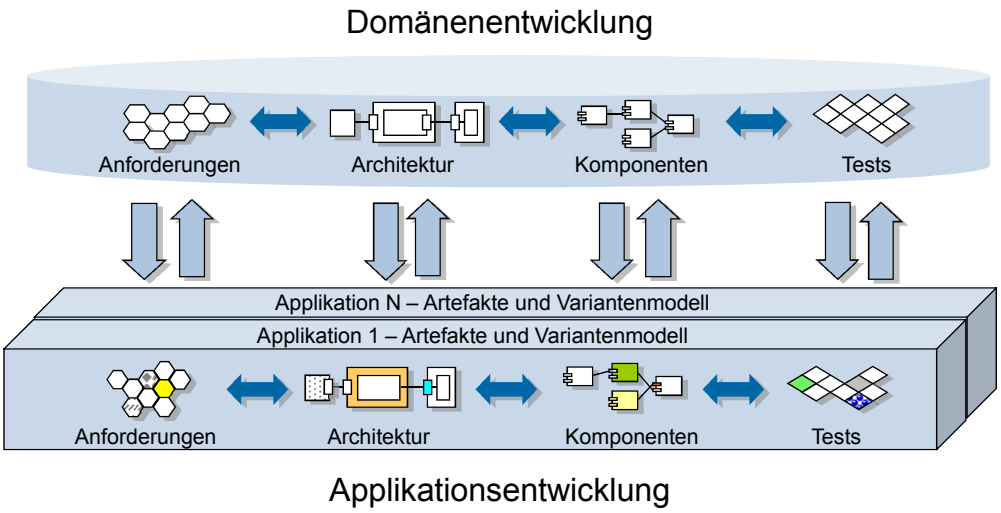


Abbildung 2.9: Domänen- und Applikationsentwicklung (basierend auf [PBL05]).

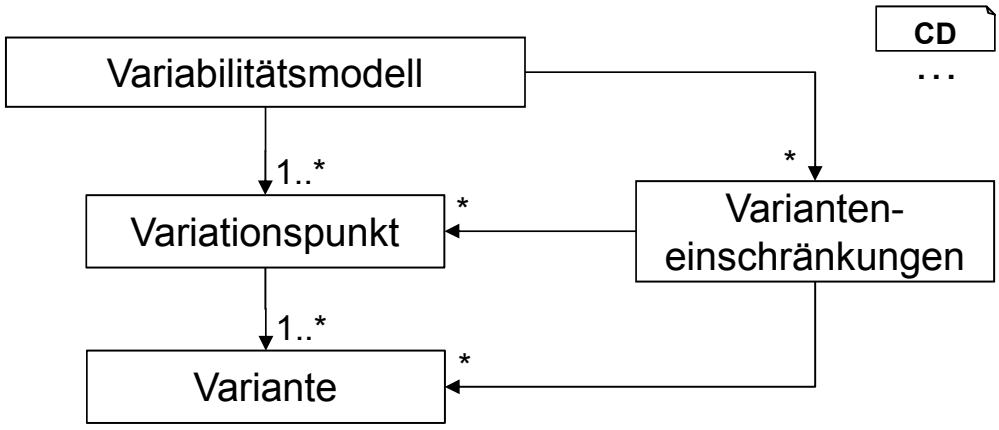


Abbildung 2.10: Grundlegender Aufbau eines Variabilitätsmodells einer Software-Produktlinie.

Definition 2.6 *Variationspunkt. Variationspunkte eines Artefakt der DE beschreiben die variablen Aspekte dieses Artefaktes. Dies beinhaltet sowohl den Variationspunkt an sich, als auch dessen mögliche Varianten. Ein Variationspunkt ist offen, insofern noch keine Variante ausgewählt wurde. Ein Variationspunkt ist gebunden, falls eine Variante gewählt wurde.*

Innerhalb der DE wird ein Variabilitätsmodell definiert, um auf einer möglichst abstrakten Ebene die Variationspunkte und Varianten der Software-Produktlinie darzustellen.

Definition 2.7 *Variabilitätsmodell. Das Variabilitätsmodell definiert die Variationspunkte und Varianten der Software-Produktlinie. Des Weiteren definiert das Variabilitätsmodell Einschränkungen und Abhängigkeiten zwischen einzelnen Varianten und Variationspunkten [PBL05].*

Bekannte Beispiele von Variabilitätsmodellen sind Merkmalmodelle [KCH$^+$90b] und das orthogonale Variablitätsmodell [PBL05]. Während Merkmalmodelle sowohl die gemeinsamen (insofern notwendige Varianten eines Variationspunktes) als auch die variablen Aspekte einer Software-Produktlinie beschreiben, beschreibt das orthogonale Variabilitätsmodell nur die variablen Aspekte der Software-Produktlinie. In Konsequenz lassen sich durch die Merkmale eines Merkmalsmodells alle durch die Software-Produktlinie realisierten Anforderungen repräsentieren, während das orthogonale Variabilitätsmodell nur optionale Anforderungen der Software-Produktlinie repräsentiert.

Der beschriebene Zusammenhang zwischen Variationspunkten, Varianten und Varianteneinschränkungen in einem Variabilitätsmodell sind in Abbildung 2.10 dargestellt.

Innerhalb der Domänenentwicklung wird im Weiteren der Definition der Referenzarchitektur eine wichtige Rolle zugeordnet, da diese die gemeinsame Struktur innerhalb der Software-Produktlinie definiert.

Definition 2.8 *Referenzarchitektur. Die Referenzarchitektur bestimmt die Struktur und die Beschaffenheit (engl. Texture) aller Anwendungen der Software-Produktlinie [PBL05]. Die Struktur bestimmt dabei die statische und dynamische Dekomposition, welche für alle Software-*

Produkte der Software-Produktlinie gültig sind [PBL05, GHK+08]. Die Beschaffenheit beschreibt dagegen Richtlinien zur Komposition und Auflösung von Variabilität.

Basierend auf den Artefakten der DE können Software-Produkte entwickelt werden.

Definition 2.9 *Applikationsentwicklungsphase. Basierend auf der Referenzarchitektur und der durch die DE gegebenen Software-Plattform werden in der Applikationsentwicklungsphase (AE) Artefakte aus der DE wiederverwendet und offene Variationspunkte gebunden, um ein spezifisches Software-Produkt auszuleiten [PBL05].*

Grundsätzlich können Artefakte, wie zum Beispiel die Referenzarchitektur entsprechend ihrer Entwicklungsphase unterschieden werden.

Definition 2.10 *Entwicklungsartefakte. Entwicklungsartefakte sind Resultate beliebiger Teilprozesse der DE oder AE [PBL05]. Domänen- beziehungsweise Anwendungs-Artefakte sind wiederum entweder der DE oder der AE direkt zuzuordnen.*

Diese Unterscheidung kann auf einzelne spezifische Entwicklungsartefakte auch direkt angewendet werden. Im Bezug auf PERSIST (siehe Abbildung 2.5) kann zwischen der Software-Komponente als Domänenartefakt und der Software-Komponentenvariante als Anwendungsartefakt unterschieden werden.

Definition 2.11 *Software-Komponente. Software-Komponente aus der DE, welche auf Basis der durch die Referenzarchitektur vorgegebenen Richtlinien in unterschiedlichen Software-Produkten Verwendung finden kann.*

Definition 2.12 *Software-Komponentenvariante. Software-Komponente aus der AE, welche auf Basis einer Software-Komponente durch Auflösung aller Variationspunkte entsteht.*

Dieser Zusammenhang zwischen Software-Komponente und Software-Komponentenvariante kann auch allgemeiner als Verhältnis zwischen Domänen- und Anwendungsartefakt beschrieben werden.

Definition 2.13 *Artefaktvariante. Anwendungsartefakt, welches durch Auflösung aller Variationspunkte eines Domänenartefakts entsteht.*

Der beschriebene Zusammenhang zwischen Software-Produktlinie, Software-Plattform, Software-Produkt, Referenzarchitektur, Software-Komponente und Software-Komponentenvariante ist in Abbildung 2.11 dargestellt.

Definition 2.14 *Variabilitätsmechanismus. Mechanismus zur Beschreibung von Variationspunkten und Varianten in Domänenartefakten (außer dem Variabilitätsmodell) sowie deren Konfiguration. Dieser Mechanismus kann als Erweiterung einer gegebenen Modellierungssprache oder als eigenständige Sprache realisiert werden.*

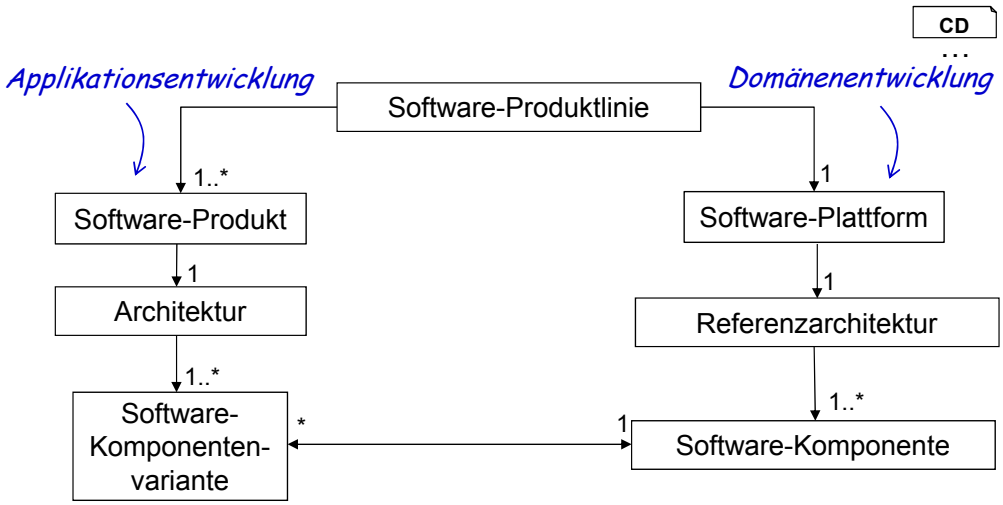


Abbildung 2.11: Grundlegende Begriffe der Software-Produktlinienentwicklung.

Unterschiedliche Variabilitätsmechanismen können grundsätzlich in annotativen, kompositionalen oder transformationalen Modellierungsansätzen unterteilt werden [SRC+12].

Annotative Ansätze beschreiben alle Variationspunkte in einem Modell [PBL05, KA09], während kompositionale Ansätze verschiedene Modellfragmente zu einem Produkt verbinden [HW07, NK08, VG07, Bos10, HRR+11]. In diesem Falle stellen die Varianten mögliche Kompositionen dar.

Im Kontext von [HW07] wird zwischen kreativen Beschreibungssprachen, zum Beispiel graphenähnlichen Sprachen und Konfigurationssprachen, wie der Verwendung von Konfigurationsparametern oder einer featuremodellbasierten Konfiguration unterschieden.

Letzteres stellt eine reduzierte Menge an möglichen Konfigurationen zur Verfügung, während Beschreibungssprachen eine deutlich höhere Flexibilität erlauben, aber auch gleichzeitig die Komplexität der Software-Produktlinie erhöhen [HW07].

Definition 2.15 *Variantenkonfiguration. Anwendungsartefakt, welches die Auflösung aller Variationspunkte eines oder mehrerer Domänenartefakte beschreibt. Die Variantenkonfiguration eines Software-Produktes löst alle Variationspunkte der zugehörigen Domänenartefakte auf. Die Variantenkonfiguration muss dabei alle Einschränkungen und Abhängigkeiten, welche durch das Variabilitätsmodell definiert sind, berücksichtigen.*

Transformationale Ansätze, wie zum Beispiel die Deltamodellierung [SBB+10, HKM+13, HHK+13, HRRS12] oder *Common Variability Language* (CVL)) [HMPO+08], beschreiben ein spezifisches Software-Produkt durch die Ausführung einer Menge von Transformationsschritten auf einem Kernmodell. In diesem Falle sind Kernmodell und Transformationsschritte Domänenmodelle. Varianten sind durch die möglichen Kombinationen aus Kernmodell und Transformationsschritten gegeben. Eine Variantenkonfiguration legt eine solche Kombination fest. Auch für Codegeneratoren kann ein transformationaler Ansatz Verwendung finden, um spezifische

Variationspunkte schrittweise erweitern zu können [GMR+16].

Definition 2.16 *Proaktive, reaktive oder extraktive Software-Produktlinienentwicklung. Eine Software-Produktlinie kann entweder durch einen proaktiven, reaktiven oder extraktiven Prozess erstellt werden [Kru02]. Der proaktive Ansatz definiert die gesamte Software-Produktlinie in einem Top Down-Ansatz, während der reaktive Ansatz eine Software-Produktlinie inkrementell realisiert. In vielen Fällen kann die initiale Software-Plattform auf Basis gegebener Software-Produkte extrahiert werden, was der extraktiven Herangehensweise entspricht.*

Neben der Aufteilung zwischen Domänen- und Applikationsentwicklung wird im Kontext der Software-Produktlinienentwicklung auch zwischen Problem- und Lösungsraum unterschieden. Während der Problemraum die Anforderungsseite der Entwicklung und auch das Variabilitätsmodell beinhaltet, repräsentiert der Lösungsraum die Umsetzung der geforderten Anforderungen.

Wird ein expliziter Variabilitätsmechanismus für den Lösungsraum in der Industrie verwendet, so basiert dieser meist auf einem annotativen Ansatz, der oft auch 150%-Modellierung genannt wird [HKM+13], da das entsprechende Modell in einem solchen Fall alle Varianten aller Variationspunkte enthält. Bei größeren Komponenten und mit einer Vielzahl an Variationspunkten ist dabei ein Verlust der Übersichtlichkeit schwer zu verhindern.

Befragungen in der Industrie haben allerdings ergeben, dass in vielen Fällen nicht nach einem systematischen Software-Produktlinienansatz mit expliziten Variabilitätsmechanismus vorgegangen wird. Stattdessen wird eine Wiederverwendung einzelner Artefakte durch Kopieren erreicht, der sogenannte *Clone&Own*-Ansatz [DRB+13]. Dieser Ansatz wird aber unsystematisch durchgeführt und führt zu einem über die Zeit deutlich steigenden Wartungsaufwand, da jeder Klon einzeln gewartet werden muss [DRB+13].

Definition 2.17 *Clone&Own. Beim Clone&Own-Verfahren werden Entwicklungsartefakte zum Zwecke der Wiederverwendung kopiert und losgelöst vom Original für ein Software-Produkt in der AE weiterentwickelt.*

Definition 2.18 *Historischer Klon. Entwicklungsartefakte die auf Grund des Clone&Own-Verfahrens einen gemeinsamen historischen Ursprung haben.*

Innerhalb der letzten Jahren wurden verschiedene Ansätze [RCC13, RC12, FLLHE14, FV03, RPK10, BRR10a] vorgestellt, um den *Clone&Own*-Ansatz zu unterstützen und ihm eine Systematik zu verleihen. Dabei werden unterschiedliche Klonerkennungsverfahren genutzt, um eine Identifikation historischer Klone und anschließende Extraktion zu ermöglichen. In Abschnitt 4.1 wird auf solche Verfahren noch im Detail eingegangen.

Grundsätzlich kann bei einer extraktiven Software-Produktlinienentwicklung zwischen drei Hauptaufgaben unterschieden werden [MZB+15]: Mit Hilfe der *Merkmalidentifikation* werden die Merkmale einer Software-Produkltinie auf Basis einer Menge von Softwarevarianten identifiziert. Des Weiteren können auch durch die Merkmalidentifikation Bedingungen zwischen den Merkmalen extrahiert werden. Das Resultat der Merkmalidentifikation ist ein Merkmalmodell. Unterschiedliche Ansätze identifizieren Gemeinsamkeiten und Unterschiede von Anforderungen [MBBA16], Modellen [ARS+14, MZKLT14, Bur17], Quellcode [ZHP+14, DRGP13] oder

Bytecode [ZH18] um auf dieser Basis einzelne Merkmale zu definieren. Dieser Schritt kann auch durch einen Domänenexperten unterstützt werden [KDO14]. Eine weiterer Ansatz besteht darin auch die Abhängigkeiten innerhalb eines Modelles oder eines Quellcodes genauer zu analysieren und auf dieser Basis für jede geschlossene Gruppe ein Merkmal zu definieren [MZKLT14]. Bei der Merkmalidentifikation wird grundsätzlich zwischen der textuellen, statischen und dynamischen Merkmalidentifikation unterschieden [DRGP13]. Diese Unterscheidung bezieht sich dabei auf die Art des Analyseverfahrens. Eine ähnliche Unterteilung ist auch für Klonerkennungsverfahren gegeben. In Abschnitt 4.1 wird auf diese Unterscheidung noch im Detail eingegangen.

Die *Merkmallokalisierung* ordnet auf Basis eines Merkmalmodells und einer Menge von Softwarevarianten jedes einzelne Merkmal ihrer konkreten Umsetzung zu. Da oft eine Merkmalidentifikation auf Basis der Analyse des Lösungsraumes erfolgt, sind Merkmalidentifikation und Merkmallokalisierung eng mit einander verknüpft.

Beim *Reengineering* werden die Artefakte der einzelnen Software-Produkte so angepasst, dass diese im Sinne der Software-Produktlinie wiederverwendbar sind. Dies bedeutet, dass abschließend alle Artefakte, die einem bestimmten Merkmal zugeordnet sind, in den Software-Varianten, deren Merkmalkonfiguration dieses Merkmal beinhaltet, verwendet werden können (Überführung Anwendungsartefakt zu Domänenartefakt).

2.2.1 Agile Software-Produktlinienentwicklung

Vergleicht man die Software-Produktlinienentwicklung mit der agilen Entwicklung, so werden größtenteils die gleichen Ziele verfolgt: erhöhte Kundenzufriedenheit sowie die Reduktion der Entwicklungskosten und -Zeiten [DPAG11]. Es wurde schon in mehreren Projekten bewiesen, dass diese Ziele durch die Software-Produktlinienentwicklung und auch durch die agile Methodik erreicht werden können [CN02b, LMD+04]. Auch suggerieren beide Ansätze, dass sie mit einer hohen Frequenz an Anforderungsänderungen effizient umgehen können [TC06]. Gleichzeitig identifizieren aber unterschiedliche wissenschaftliche Arbeiten Konflikte zwischen einer agilen Herangehensweise und der Software-Produktlinienentwicklung [MRS10, HF08]: Die agile Entwicklung fordert kurze Entwicklungszyklen mit kurzen Planungsperioden, während die proaktive Software-Produktlinienentwicklung die korrekte Hervorsage zukünftiger Anforderungen durch eine intensive Analysephase fordert.

Um mögliche Synergien und Konflikte im Detail zu evaluieren, wurde im Jahre 2002 ein erster Workshop durchgeführt [Yod02] und die agile Software-Produktlinienentwicklung explizit aufgeführt. Ungefähr 10 Jahre später wurde eine erste intensive Literaturrecherche [DPAG11] zu diesem Thema durchgeführt, die keinen der betrachteten Ansätze als klaren Favoriten identifizieren konnte. Abhängig von der jeweiligen Firmenpolitik müssen die Ansätze angepasst beziehungsweise das passende Level an Flexibilität identifiziert werden.

Die Reduktion der Langzeitinvestition während der DE-Phase ist einer der Hauptgründe agile Einflüsse in die Software-Produktlinienentwicklung mit aufzunehmen, da sich ein zu intensiver Fokus auf diese Phase als kostenintensiv und riskant herausgestellt hat [DPAG11]. Ändert sich die Marktsituation, müssen die Resultate intensiver Analysen oft verworfen werden. Der Aufbau und die Wartung einer Software-Produktlinie führt zu einem nicht zu unterschätzenden Koordinierungsaufwand und resultiert in langsameren Entwicklungszyklen [Bos10]. Im Kontext der

heutigen Herausforderungen einer sich schnell ändernden Technologielandschaft wird die Gefahr einer falsch platzierten Langzeitinvestition immer realistischer. Gerade für einen Zulieferer spielt dieser Aspekt eine wichtige Rolle. Er muss üblicherweise mit den wandelbaren Anforderungen seiner Kunden umgehen können und kann die Entwicklung einer Software-Produktlinie nur schwer im Tagesgeschäft unterbringen, da er durch die strategischen Entscheidungen seiner Kunden getrieben ist. Auch in [CKMM08] wird auf die Wichtigkeit eines intensiven Austausches zwischen der Applikations- und der Domänenentwicklung eingegangen.

Mit Hilfe eines reaktiven Ansatzes entwickelte Salion eine initiale Menge an Systemen, um dort Gemeinsamkeiten und Unterschiede in einem zweiten Schritt abzuleiten [CN02a]. Dies ist kein Einzelfall. Oft werden variable Anforderungen zuerst durch kundenspezifische Lösungen realisiert [MRS10]. Anschließend erst wird durch eine selektive Refaktorisierung schrittweise die Software-Plattform erweitert. Außerdem haben sich Klonerkennungsverfahren als nützliches Hilfsmittel erwiesen, um potentielle Kandidaten für eine Software-Plattform zu identifizieren [MRS10].

Zusammenfassend lässt sich festhalten, dass die agilen Einflüsse die Software-Produktlinienentwicklung von einer proaktiven Herangehensweise wegführten und deutlich stärker auf extraktive und reaktive Herangehensweisen setzen. Dabei steht initial deutlich stärker die Anwendungsentwicklung im Vordergrund. Des Weiteren finden in der Forschung übliche Verfahren zur Software-Produktlinienentwicklung leider in der Industrie oftmals noch keine Verwendung, so dass es stattdessen üblich ist, durch Anlegen von historischen Klonen eine Wiederverwendung zu ermöglichen. Da diese Praxis auch in der Automobilindustire in der Vergangenheit öfters durchgeführt wurde, ist es notwendig, eine Migrationsstrategie zu entwerfen, die es ermöglicht auf Basis des schon etablierten *Clone&Own*-Ansatzes größtmögliche Wiederverwendung mit möglichst geringem Aufwand zu erreichen.

2.3 Anforderungsanalyse

Auf Basis der vorangegangenen Erörterungen werden im Folgenden grundlegende Anforderungen an einen Software-Produktlinienentwicklungsprozess aufgeführt.

- **P1** *Unterstützung notwendiger Normen*
  Der Software-Produktlinienentwicklungsprozess muss, wo möglich, mit aktuell gängigen Standards und Normen konform sein und somit eine notwendige Zertifizierung unterstützen. Dabei sind die Normen *CMMI* und *ISO 26262* sowie die Aspekte Architekturdesign, Nachverfolgbarkeit der Anforderungen, Konfigurations-, Release- und Changemanagement besonders zu berücksichtigen (Bezug zu **RQ1** und **RQ4**).

- **P2** *Kurze Entwicklungszyklen*
  Der Prozess muss im Sinne einer agilen Entwicklung möglichst kurze Entwicklungszyklen erlauben (Bezug zu **RQ1**).

- **P3** *Umgang mit Anforderungsänderungen*
  Der Prozess muss mit sich stetig ändernden Anforderungen umgehen können und diesen Umstand möglichst effizient berücksichtigen (Bezug zu **RQ1** und **RQ4**).

- **P4** *Basis PERSIST*
  Aufgrund der Entwicklung im Rahmen beteiligter Industriepartner muss der Prozess, wo sinnvoll, auf PERSIST aufbauen (Bezug zu **RQ2**).

- **P5** *Einfacher Variabilitätsmechanismus*
  Der verwendete Variabilitätsmechanismus muss sich möglichst an in der Industrie etablierten Verfahren orientieren und eine geringe Einarbeitungszeit sowie einen geringen Änderungsaufwand für gegebene Funktionen bedeuten (Bezug zu **RQ2**).

- **P6** *Verschiedene Umsetzungsstrategien*
  Der Prozess und die zugehörige Werkzeuglandschaft muss generell sowohl eine extraktive, reaktive als auch proaktive Entwicklung ermöglichen. Dabei muss unter Berücksichtigung der agilen Software-Produktlinienentwicklung und Anforderung **P2** der Fokus deutlich auf den reaktiven und extraktiven Umsetzungsstrategien liegen (Bezug zu **RQ1**).

- **P7** *Beeinträchtigung des Projektgeschäftes*
  Eine Etablierung und Wartung einer Software-Plattform muss möglichst ohne Beeinträchtigung des Produktgeschäftes möglich sein und darf keinen größeren zusätzlichen Kostenfaktor darstellen (Bezug zu **RQ1**).

Kapitel 3

Reaktive Software-Produktlinienentwicklung

Die schon durch die agile Software-Produktlinienentwicklung vorgegebene Tendenz beschreibt eine Aufweichung der proaktiven Software-Produktlinienentwicklung hin zu einer deutlich reaktiveren Herangehensweise, um Fehlentwicklungen zu vermeiden und damit die Effizienz weiter zu steigern. Diese Tendenz ist unter anderem auch der Tatsache geschuldet, dass die Entwicklung einer Software-Komponente in vielen Fällen einen deutlich erhöhten Aufwand bedeutet [PBL05]. Des Weiteren wird Variabilität auf Grund des zugehörigen Komplexitätsanstiegs im Tagesgeschäft gemieden [DRB+13]. Diese Gründe führen dazu, dass in der Industrie bis zum heutigen Tage oft das sogenannte *Clone&Own*-Verfahren Verwendung findet, da dieses intuitiv und einfach zu verwenden ist und ohne eine langfristige Planung durchgeführt werden kann [DRB+13].

Während dieses Verfahren kurzfristig ohne Aufwände eine Wiederverwendung ermöglicht, führt es aber auf Dauer zu größeren Problemen, da erhöhter Wartungsaufwand notwendig ist, um mehrere historische Klone gleichzeitig zu pflegen [DRB+13]. Auch führt dieser unsystematische Ansatz dazu, dass sich die historischen Klone auf längere Sicht voneinander fortentwickeln und die Übersicht über gemeinsame Teile verloren geht.

Trotz der aufgeführten negativen Aspekte des *Clone&Own*-Verfahrens kann nicht vernachlässigt werden, dass dieser Ansatz in der Industrie häufig Verwendung findet, obwohl etablierte Grundlagen einer Software-Produktlinienentwicklung zur Verfügung stehen. Stattdessen gilt es zu verstehen, welche Vorteile dieser Ansatz bietet und welche Hürden eine systematischere Herangehensweise verhindern. Des Weiteren decken sich die in [DRB+13] aufgeführten Beobachtungen und Kommentare mit dem teilweise gegebenen Ist-Zustand innerhalb der Automobilindustrie: Varianten werden aktuell meist als historischer Klon eines Entwicklungsartefaktes erstellt und dann separiert von diesem weiterentwickelt. Die Ursache liegt dabei meist im durch den Kunden vorgegebenen knappen Zeitrahmen innerhalb des aktuellen Projektes und in dem durch die Organisationsstrukturen vorgegebenen Fokus auf das Projektergebnis.

Die Entwicklung von Domänenartefakten ohne direkten Bezug zur Anwendungsentwicklung wurde als riskant und kostenintensiv identifiziert [DPAG11].

Unter der Betrachtung der Argumentation der agilen Software-Produktlinienentwicklung und der aktuell etablierten Mechanismen innerhalb der Industrie ist eine deutlich projektbezogenere Entwicklung der Software-Plattform vonnöten, um einen Übergang in eine produktliniengetriebene Entwicklung zu ermöglichen. Dies bedeutet, dass Ergebnisse aus Projekten schrittweise in eine Software-Plattform überführt werden (reaktiv), anstatt initial eine Software-Plattform

aufzubauen (proaktiv).

3.1 Produktgetriebene Software-Produktlinienentwicklung

Es ist notwendig die gegebenen *Clone&Own*-Ansätze durch einen systematischeren Prozess zu ersetzen, der möglichst die Vorteile beider Seiten vereint. Dabei sollten die agilen Einflüsse einer reaktiven Entwicklungsweise verwendet und erweitert werden, um eine entsprechende Synergie zu ermöglichen. Dies führt generell zu einer deutlich produktgetriebeneren Entwicklung der Software-Plattform, wodurch essentielles Feedback der Anwendungsentwicklung an die Domänenentwicklung weitergegeben wird.

Definition 3.1 *Produktgetriebene Software-Produktlinienentwicklung. Reaktive Software-Produktlinienentwicklung, die eine schrittweise Weiterentwicklung der Software-Plattform auf Basis etablierter Entwicklungsartefakte oder neuer Anforderungen aus der AE fossiert. Dabei kann eine extraktive Herangehensweise fossiert werden, um auf Basis mehrerer Entwicklungsartefakte ein Domänenartefakt zu extrahieren.*

Gleichzeitig gilt aber zu beachten, dass der zusätzliche Aufwand für die AE dabei möglichst gering ausfallen soll. Der primäre Fokus liegt also darin, durch das Wiederverwendungspotential der Software-Plattform der AE die Arbeit zu erleichtern und gleichzeitig durch die Arbeit in der AE Informationen zu sammeln, die eine möglichst effiziente Weiterentwicklung der Software-Plattform ermöglichen. Dabei soll der Prozess nicht durch langfristige Entscheidungen, sondern stattdessen auf Basis aktueller Entwicklungen gesteuert werden. Dies zu erreichen und dabei den zusätzlichen Aufwand für die Software-Plattform möglichst gering zu halten ist das Ziel des im Folgenden vorgestellten Prozesses.

In Abbildung 3.1 ist ein entsprechender Prozess skizziert, welcher die aufgeführten Aspekte adressiert. Dabei legt der Prozess den Fokus auf eine frühzeitige und möglichst automatisierte Erkennung von Ähnlichkeiten einzelner Software-Komponentenvarianten zwischen verschiedenen Software-Produkten untereinander. Dabei werden in Abbildung 3.1 die Aktivitäten während der DE gräulich dargestellt. Alle anderen Aktivitäten sind Teil der AE.

Der Prozess beginnt mit der Spezifikation/Modifikation einer neuen Software-Komponentenvariante für ein Software-Produkt (Schritt 1). Dies stellt immer den Ausgangspunkt dar. Dabei ist es im ersten Schritt nur notwendig, die zugehörige Software-Komponente der Software-Komponentenvariante zu identifizieren. Dies erfolgt meist durch eine kurze funktionale Beschreibung und einen ersten Grobentwurf der Schnittstelle.

Auf Basis dieser ersten Skizze (kurze funktionale Beschreibung, Grobentwurf Schnittstelle) der Software-Komponentenvariante wird in einem zweiten Schritt (2) durch den Applikationsentwickler ein Vergleich mit der Referenzarchitektur durchgeführt. Dieser Schritt ist manuell durchzuführen und dient dazu, eine entsprechende Übereinstimmung zu identifizieren. Existiert keine zugehörige Software-Komponente, so wird initial nur die Software-Komponentenvariante entwickelt und ist somit nur Teil der Entwicklungsarchitektur (3).

Da gerade bei einer etablierten Software-Plattform grundsätzlich davon ausgegangen werden kann, dass die Entwicklung einer Software-Komponentenvariante ohne Bezug zu einer Software-Komponente eher ungewöhnlich ist und die Referenzarchitektur unter der Kontrolle

der DE steht, wird die entsprechende Zuweisung innerhalb des Projektes nochmals überprüft (4).

Dabei gilt es sowohl zu überprüfen, inwiefern die initiale Skizze nicht doch der Referenzarchitektur zugeordnet werden kann, als auch die generelle Architekturentscheidung zu überdenken und - falls notwendig - anzupassen. Diese beidseitige Überprüfung soll grundsätzlich dass Verständnis für die Referenzarchitektur auch bei den Applikationsentwicklern schärfen, als auch durch das Vier-Augen-Prinzip mögliche Fehlentscheidungen verhindern.

Ist die neue Software-Komponentenvariante auch durch die DE freigegeben, so muss diese mit üblichen Methoden gänzlich neu entwickelt werden (5). Anschließend wird eine zugehörige Software-Komponente durch Anpassung der Referenzarchitektur hinzugefügt (10).

Konnte stattdessen eine falsche Zuweisung identifiziert werden oder wurde die Software-Komponentenvariante von vorneherein einer Software-Komponente zugeordnet, so soll es durch einen direkten Vergleich aller Software-Komponentenvarianten der Software-Komponente sowie der Software-Komponente mögliche Ähnlichkeiten festgestellt werden (6). Welche Analysen auf welchen Entwicklungsartefakten dabei ausgeführt werden können, ist abhängig vom aktuellen Entwicklungsstand.

Ist ein gewisses Maß an Ähnlichkeit identifiziert, muss entschieden werden, inwiefern eine

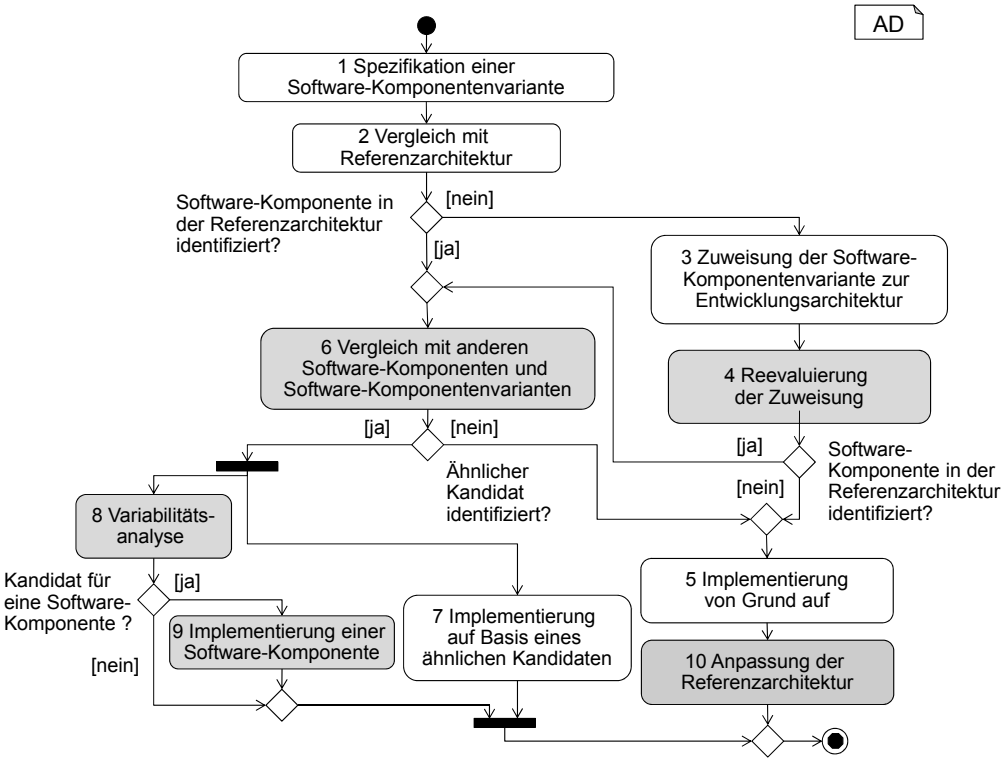


Abbildung 3.1: Prozess zur projektgetriebenen Software-Produktlinienentwicklung.
(Schraffur / weiß = Domänen- / Applikationsentwicklungsaktivitäten)

Software-Komponente zu entwickeln oder zu erweitern ist (8). Dabei spielt sowohl das Maß an Ähnlichkeit als auch der aktuelle Projektkontext (Zeit, Budget) eine Rolle. Zeitgleich kann auch mit einer Softwareprodukt-spezifischen Implementierung begonnen werden. Diese kann dann mittels *Clone&Own* den größtmöglichen Vorteil aus den erkannten Ähnlichkeiten ziehen (7), ohne sich gleichzeitig den Mehraufwand einer Domänenentwicklung aufzubürden.

Wurde entschieden, eine Software-Komponente zu entwickeln oder zu erweitern, so kann dies entweder direkt für das Software-Produkt geschehen oder parallel dazu umgesetzt werden (9).

3.2 Teilautomatisierte Software-Produktlinienextraktion

Der beschriebene Prozess beinhaltet reaktive (schrittweise Erweiterung der Softwareplattform) und extraktive (meist durch Extraktion schon gegebener Software-Komponentenvarianten) Aspekte. Dabei spielt der Vergleich ähnlicher Software-Komponentenvarianten zur Identifikation gemeinsamer Teile (Schritt 6) eine wichtige Rolle. Nur falls dieser Schritt auf eine effiziente Weise durchgeführt werden kann, führt der gesamte Prozess zur erhofften Effizienzsteigerung bei gleichzeitiger geringer Last für die AE.

Zur Analyse unterschiedlicher Software-Komponentenvarianten können verschiedene Anwendungsartefakte betrachet werden. Diese können sich sowohl bezüglich ihres Artefakttyps als auch ihrer Artefaktrolle unterscheiden.

Definition 3.2 *Artefakttyp. Dateityp des Artefakts, wie zum Beispiel Simulinkmodell (.mdl), C Code (.c) oder Exceldatei (.xls / .xlsx).*

Definition 3.3 *Artefaktrolle. Artefakte desselben Artefakttypen können in einem Entwicklungsprozess unterschiedliche Rollen, wie zum Beispiel Verhaltensspezifikation, Testspezifikation oder Schnittstellendefinition, einnehmen. Simulink-Modelle können zum Beispiel als Verhaltensspezifikation oder Umgebungsmodell verwendet werden.*

Trotz des durch die initiale Zuweisung zur Referenzarchitektur gegebenen stark reduzierten Suchraumes ist ein detaillierter manueller Abgleich mehrerer Software-Komponenten zeitintensiv, da unterschiedliche Schritte durchgeführt werden müssen:

1. Zur Verfügung stehende Entwicklungsartefakte müssen identifiziert werden.
2. Entwicklungsartefakte mit gleicher Artefaktrolle müssen überprüft werden, inwiefern sie für eine Auswertung geeignet sind.
3. Geeignete Artefakte mit gleicher Artefaktrolle müssen einer detaillierten Analyse unterzogen werden, um Gemeinsamkeiten identifizieren zu können. Dabei ist die Analyse für jede Artefaktrolle unterschiedlich und muss eventuell von unterschiedlichen Experten durchgeführt werden.

Insofern ist es notwendig, die genannten Schritte möglichst zu automatisieren, um eine effiziente Identifikation von Gemeinsamkeiten zu ermöglichen. Die Identifikation zur Verfügung

stehender Entwicklungsartefakte lässt sich leicht realisieren, falls das zugrundeliegende Unternehmen entsprechenden Standards, wie zum Beispiel durch *PERSIST* gegeben, folgt. Auch die Identifikation geeigneter Entwicklungsartefakte wird durch Standards deutlich erleichtert beziehungsweise bedingt ein effektiver automatisierter Abgleich eine Standardisierung der genutzten Artefakttypen und deren Inhalte. Wäre ein entsprechender Standard bzw. ein vereinheitlichter Entwicklungsprozess nicht gegeben, so wäre es auch schwer möglich, manuell nützliche Informationen zu extrahieren. Für eine Automatisierung ist es aber auch wichtig Artefakttypen, welche für dieselbe Artefaktrolle genutzt werden, in ein gemeinsames Zwischenformat zu transformieren. Dadurch lässt sich eine Automatisierung leichter durchführen und weitere Artefakttypen mit derselben Artefaktrolle können schneller bezüglich einer Analyse integriert werden.

Als Konsequenz lässt sich der beschriebene Prozess nur automatisieren, falls ein vereinheitlichter Prozess mit klarer Definition der genutzten Artefaktrollen und Artefakttypen und der zugehörigen Richtlinien für ihre Verwendung gegeben sind. Da wiederum, wie zuvor aufgeführt, der Prozess seine Ziele nur erfüllt, wenn eine Analyse der Ähnlichkeiten zwischen unterschiedlichen Software-Komponentenvarianten automatisiert werden kann, ist eine Standardisierung zwingend notwendig.

Weiteres Potential zur Automatisierung bieten die Analyse des Wiederverwendungspotentials bzw. die zugehörige Entwicklung einer Software-Komponente (Schritte 8 und 9). Auch die Zuweisung zur Referenzarchitektur kann automatisiert werden. Da der Architekturentwurf im Allgemeinen aber ein sehr kreativer Prozess ist und gleichzeitig eine Ähnlichkeitsanalyse auf Basis von Prosatexten ohne Verwendung einschränkender Muster (die funktionale Beschreibung) oft zu falschen Ergebnissen führen kann [Hol10], ist eine Automatisierung von Schritt 2 nicht zielführend. Des Weiteren würde eine Automatisierung in diesem Schritt den angedachten Wissensaustausch bezüglich der Referenzarchitektur zwischen DE und AE verhindern.

In Abbildung 3.2 sind die entsprechenden Aktivitäten noch einmal hervorgehoben, die ein Potential zur Automatisierung bieten.

Um nun Ähnlichkeiten zwischen Software-Komponentenvarianten automatisiert herleiten zu können, ist es notwendig zu identifizieren, mit welchen Mitteln man auf Basis welcher Artefaktrollen möglichst genaue Ähnlichkeitsaussagen erhalten kann. Dabei spielen unterschiedliche Aspekte eine Rolle. Zuerst ist es notwendig, den Zeitpunkt innerhalb der Entwicklung zu identifizieren, zu dem eine entsprechende Analyse durchgeführt wird. Auf Basis einer Automatisierung und im Geiste einer agilen Entwicklung ist es natürlich möglich und auch empfehlenswert, die entsprechenden Analysen fortwährend durch einen *Nightly Build* durchzuführen. Gleichzeitig ist trotzdem wichtig, ab welchem Zeitpunkt frühestens eine Analyse durchgeführt werden kann. Entsprechend der vorgestellten Methode können erste Analysen direkt zu Beginn der Entwicklung durchgeführt werden. Dabei stehen zu einem solchen Zeitpunkt keine formaleren Entwicklungsartefakte zur Verfügung und ein Abgleich der funktionalen Beschreibung wurde schon manuell durchgeführt. Typische weitere Entwicklungsartefakte innerhalb des Entwicklungsprozesses sind (siehe auch Abschnitt 2.1) Anforderungsspezifikation, Schnittstellendefinition, Modell, Quellcode und Testspezifikationen.

Im Folgenden wird nun im Detail auf diese Artefaktrollen eingegangen und für jede Artefaktrolle einzeln evaluiert, inwiefern dieser für eine automatisierbare Ähnlichkeitsanalyse geeignet ist.

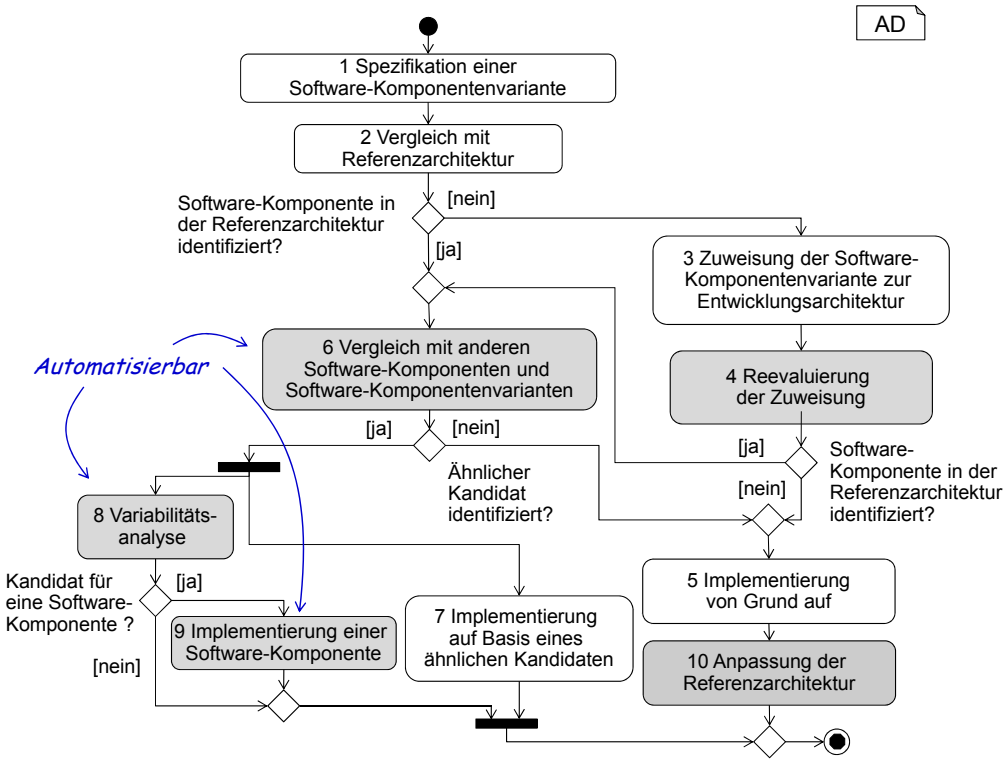


Abbildung 3.2: Potential zur Automatisierung in der produktgetriebenen Software-Produktlinienentwicklung.
(Schraffur = Domänenentwicklungsaktivitäten, weiß = Applikationsentwicklungsaktivitäten)

Anforderungsspezifikation: Auf Basis des Prozesses werden Anforderungen als erstes definiert und sollten für die Definition der funktionalen Beschreibung der Software-Komponentenvariante schon genutzt worden sein. Insofern ist es möglich, die entsprechenden Anforderungen, die durch die Software-Komponentenvariante umzusetzen sind, dieser zuzuordnen. Da Anforderungen aber in der Industrie meist nicht formalisiert sind, sondern in Prosa vorliegen, ist es schwer, daraus fundierte Aussagen bezüglich deren Ähnlichkeiten zu treffen [Hol10]. Entsprechende Ansätze führen nur durch deutliche Restriktionen der Satzstruktur und des zu verwendenden Vokabulars zu Ergebnissen mit wenigen falsch positiven oder falsch negativen Ergebnissen [Hol10]. Gleichzeitig ist es innerhalb der Softwareentwicklung durchaus üblich, dass zwar ein grobes Verständnis einer Software-Komponentenvariante gegeben ist, weitere Details und viele Sonderfälle aber erst direkt im Modell hinzugefügt werden. In diesem Falle werden auch zugehörige Anforderungen erst später formuliert oder direkt modellbasiert spezifiziert. In beiden Fällen stehen textuelle Anforderungen nicht zur Verfügung. Da die erforderlichen Informationen nicht zur Verfügung stehen oder nicht formal genug für eine automatisierbare Auswertung beschrieben sind, stellen textuelle Anforderungsdokumente nicht die beste Grundlage für eine

automatisierte Ähnlichkeitsanalyse dar.

Schnittstellendefinition und Modelle: In einem zweiten Schritt werden innerhalb der Entwicklung die Funktionen der Software-Komponente und deren Schnittstellen definiert. Grundsätzlich ist in der Automobilindustrie davon auszugehen, dass nach einer initialen Konzeptphase die Definition der Funktionalität modellbasiert umgesetzt wird (dies geschieht meist mit Simulink, siehe auch Abschnitt 2.1). Die Verwendung beider Artefaktrollen ist normalerweise durch entsprechende Standards definiert und beide bieten ein formales Fundament auf dessen Basis eine automatisierbare Ähnlichkeitsanalyse möglich ist.

Quellcode: Des Weiteren bietet auch der Quellcode ein ausreichend formales Fundament zur Analyse. Da die Modelle aber oft zur Code-Generierung verwendet werden und dies meist erst im späteren Verlauf der Entwicklung möglich ist, bieten sich Modelle im Kontext der Automobilindustrie deutlich stärker an: Sie stehen frühzeitiger zur Verfügung und beschreiben semantisch dieselben Informationen wie der generierte Quellcode.

Testspezifikation: Während der Entwicklung ist es auch notwendig, die umgesetzte Funktionalität zu testen. Dabei ist es notwendig, parallel zur Modellentwicklung zugehörige Testspezifikationen zu definieren. Entsprechend eines Vier-Augen-Prinzips werden diese Tests oft von einer weiteren Person auf Basis der Anforderungen definiert. Diese sind also zeitlich meist unabhängig von den Modellen und basieren auf derselben Schnittstellendefinition. Im Kontext einer testgetriebenen Entwicklung [Bec02] werden Testspezifikationen sogar vor der eigentlichen Modellspezifikation durchgeführt. Testspezifikationen approximieren dabei das eigentliche Verhalten, um den Testaufwand möglichst gering zu halten, ohne dass dabei einzelne Aspekte der Anforderungen nicht adressiert werden. Auf Basis entsprechender Richtlinien (CMMI, ISO26262, siehe auch Abschnitt 2.1) ist es sogar zwingend notwendig, jegliche Anforderungen durch zugehörige Testfälle zu adressieren. Insofern können Testspezifikationen als eine formalere Darstellung der Anforderungen gesehen werden. Der Grad der Formalität ist dabei ausreichend, um eine automatisierte Ähnlichkeitsanalyse zu ermöglichen.

Zusammenfassend wurden unterschiedliche Artefaktrollen identifiziert, die für eine automatisierte Ähnlichkeitsanalyse zur Verfügung stehen: Schnittstellen, Modelle, Quellcode und Testspezifikationen. Während Schnittstellen rein strukturelle Informationen bezüglich der Software-Komponentenvariante beschreiben, definieren Modelle, Quellcode und Testspezifikationen auch semantische (funktionale) Aspekte. Betrachtet man gleichzeitig noch die initiale manuelle Zuweisung zur Referenzarchitektur beziehungsweise zur Software-Komponente und beschreibt dies als extrinsische ID (ähnlich zu [BRR$^+$10b]) ergibt sich das in Abbildung 3.3 dargestellte Bild.

Gleichzeitig ergibt sich durch den vorgestellten Prozess und die Abhängigkeiten zwischen den einzelnen Ebenen eine grundlegende Reihenfolge der Analyse, die es erlaubt, schrittweise den Suchraum zu reduzieren: extrinsisch, schnittstellen-basiert, semantisch. Dies ist auch im Kontext einer Automatisierung hilfreich, da spätere Analysen deutlich aufwendiger sind, gerade im Kontext der Semantik. Des Weiteren ist es hilfreich, die ähnlichen Elemente der Schnittstellen zweier Software-Komponentenvarianten einander zuzuweisen, bevor auf Basis dieser Zuweisung eine funktionale Ähnlichkeit nachgewiesen werden kann. Wird diese Reihenfolge nun wieder den zugrundeliegenden Artefaktrollen zugeordnet, findet zuerst ein manueller Abgleich der funktionalen Beschreibung und dessen Zuordnung zur Referenzarchitektur und Software-

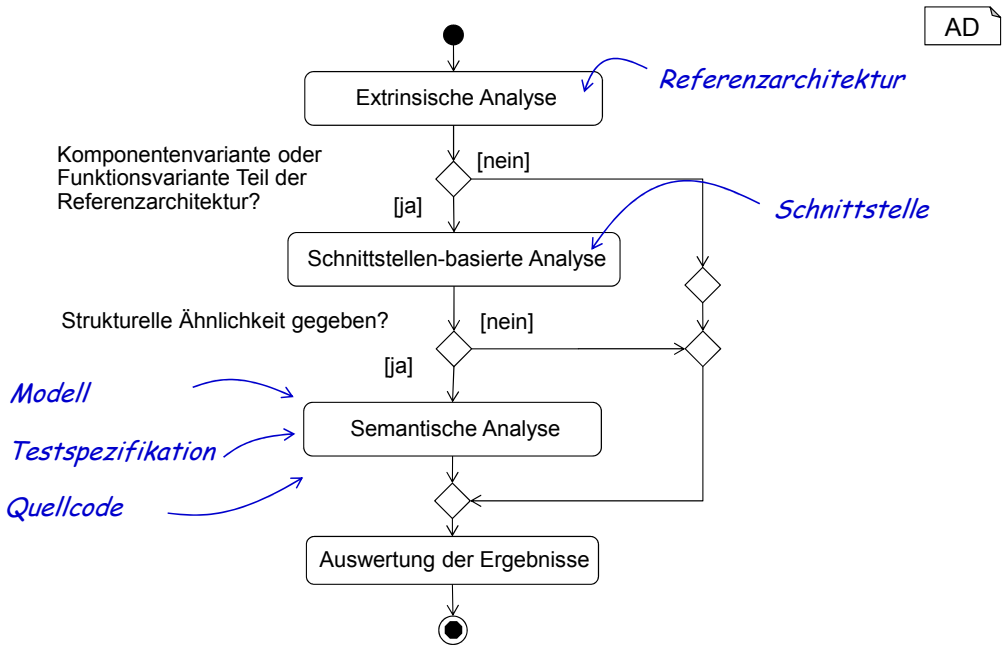


Abbildung 3.3: Schrittweise Reduktion des Suchraumes durch Ähnlichkeitsanalysen auf extrinsischer, schnittstellen-basierter und semantischer Ebene.

Komponente statt. Anschließend werden auf Basis der Schnittstellendefinition strukturelle Ähnlichkeiten identifiziert. Auf Basis dieser Ähnlichkeiten können wiederum in einem abschließenden Schritt mit Hilfe von Modellen, Quellcode oder Testspezifikationen semantische Analysen durchgeführt werden.

Anschließend kann auf Basis der identifizierten Ähnlichkeiten manuell eine Entscheidung bzgl. des Wiederverwendungspotentials getroffen werden.

Die Identifikation von gleichen Codeteilen in der Software wird im Forschungsgebiet der Klonerkennung [RCK09] behandelt. Des Weiteren gibt es auch Anpassungen dieser Verfahren zur Analyse von Modellen [RCK09].

Die Erkennung von Ähnlichkeiten auf Basis von Schnittstellen, Modellen, Code und Testspezifikationen spielt in diesem Prozess eine wichtige Rolle. Nur mit deren effizienten und korrekten Umsetzung kann der Prozess sein Potential zur Unterstützung der Softwareproduktlinienentwicklung entfalten. In Kapitel 4 werden zugehörige Grundlagen diskutiert und anschließend in Abschnitt 5.1, Abschnitt 5.2 und Kapitel 6 die extrinsischen, schnittstellenbasiert und semantischen Ähnlichkeitsanalysen definiert, die dies möglich machen.

3.3 Kompositionaler Variabilitätsmechanismus

Ist ein Potential zur Wiederverwendung identifiziert und soll mittels Software-Komponenten realisiert werden, so ist in einem weiteren Schritt zu klären welcher Variabilitätsmechanismus

verwendet wird. Wie schon in Abschnitt 2.2 erläutert, kann dies generell durch annotative, transformationale und kompositionale Ansätze geschehen.

Innerhalb der Industrie wird meist zur Modellierung einer Software-Produktlinie ein 150%-Ansatz [GKPR08] verwendet, ein annotativer Ansatz. Dieser beschreibt in einem Modell alle Variationspunkte und Varianten. Dies kann bei einem größeren Modell mit mehreren Variationspunkten und Varianten schnell dazu führen, dass man die Übersicht über das Modell verliert. Gleichzeitig ist es notwendig, auf Basis dieses Modells Entwicklungsartefakte auszuleiten, da sowohl bezüglich des Speicheraufwandes, bezüglich einzelner Sicherheitsaspekte als auch zum Schutze der Software-Produktlinie ein Software-Produkt nicht die gesamte Software-Plattform enthalten darf (selbst wenn durch eine entsprechende Vairantenkonfiguration auszuschließen ist, dass andere Teile aktiv genutzt werden).

Bei einem kompositionalen Ansatzes sind die genannten Probleme nicht gegeben, da eine Komposition immer nur die relevanten Elemente enthält. Modernere Modellierungssprachen als auch Simulink unterstützen normalerweise ein Referenz-Mechanismus, der die Komposition referenzierter Modelle ermöglicht.

Definition 3.4 *Entwicklungsartefaktreferenz. Eine Entwicklungsartefaktreferenz erlaubt ein Entwicklungsartefakt im Kontext anderer Entwicklungsartefakte mehrfach zu referenzieren und dieses somit wiederzuverwenden. Zur Laufzeit ist es möglich referenzierte Entwicklungsartefakte für jeden Kontext einmal separat zu instanziieren (insofern instanziierbar) und somit mehrere Instanzen zu erhalten.*

Des Weiteren sind, wie in Abschnitt 2.1 beschrieben, innerhalb etablierter Software-Entwicklungsprozesse in der Automobilindustrie eine größere Menge von unterschiedlichen Werkzeugen involviert, die aktuell nicht direkt einen annotativen oder transformationalen Ansatz unterstützen. Kompositionen sind dagegen über Referenzmechanismen leicht realisierbar. Insofern lässt sich ein kompositionaler Ansatz leichter in den Köpfen der Mitarbeiter verankern und in eine gegebene Werkzeugkette integrieren.

Transformationale Ansätze, wie zum Beispiel die Delta-Modellierung [SBB+10, HKM+13, HHK+13], beschreiben zwar einen generell intuitiven Ansatz, benötigen aber zur Umsetzung größere Anpassungen an bisherigen Werkzeugen. Auch stellt sich die Wartung der einzelnen Transformationsschritte und das Verständnis für die resultierende Gesamttransformation als komplexer dar und muss durch entsprechende Sichten unterstützt werden. Als Resultat benötigen transformationale Ansätze zur Einführung in ein industrielles Umfeld den meisten Schulungs- und Implementierungsbedarf.

Im Vergleich dazu kann ein kompositionaler Ansatz mit geringem Schulungs- und Implementierungsbedarf in einen gegebenen Entwicklungsprozess integriert werden. Dabei bietet PERSIST mit seiner an AUTOSAR angelehnten Richtlinien zur Software-Architekturentwicklung und der Komposition aus Software-Komponenten und Funktionen den notwendigen Ausgangspunkt, um einen kompositionalen Ansatz ohne größere Anpassungen an gegebene Richtlinien zu integrieren.

Innerhalb dieses Softwarearchitekturentwicklungsrahmenwerkes - und auch analog zur produktgetriebenen Softwareproduktlinienentwicklung - bildet die Software-Komponente im Lösungsraum den variablen Hauptaspekt.

Die bisherigen Überlegungen spiegeln den schon in Abbildung 2.11 dargestellten Zusammenhang wieder. Ausgehend von einer entsprechenden Variantenkonfiguration beinhaltet ein Software-Produkt eine Menge an Software-Komponentenvarianten, welche eine Teilmenge aller Software-Komponenten der Referenzarchitektur repräsentieren. Im Falle einer Neuentwicklung während der AE können auch Software-Komponentenvarianten ohne Bezug zu einer Software-Komponente und damit zur Referenzarchitektur entstehen (Schritt 5 in Abbildung 3.1). Dieser Bezug sollte aber zeitnah hergestellt werden (Schritt 10 in Abbildung 3.1).

Jede Software-Komponentenvarianten wird wiederum durch Komposition einer Teilmenge der durch die zugehörige Software-Komponente definierten Funktionen gebildet (Vergleich dazu siehe auch Aufbau einer PERSIST-Architektur in Abbildung 2.5). Auch in diesem Falle können Funktionen im Zuge einer Neuentwicklung ein Anwendungsartefakt darstellen.

Die Schnittstelle der Software-Komponentenvariante ist indirekt durch die Schnittstellen ihrer Funktionen gegeben. Durch PERSIST-Autoconnect ergeben sich alle im Kontext der Software-Komponentenvariante offenen Signale, welche die Schnittstelle der Software-Komponentenvariante beschreiben.

Für Funktionen können bewusst keine Varianten gebildet werden, um auf dieser Ebene einen *Clone&Own*-Ansatz zu unterbinden. Erfahrungen im Einsatz der produktgetriebenen Softwareproduktlinienentwicklung haben deutlich gemacht, dass eine starke Tendenz zur Bildung von Funktionsvarianten, welche sich nur gering unterschieden, gegeben war. Diese wurden über das Anlegen einer direkten Kopie realisiert - in Konsequenz mussten entsprechende Wartungsschritte wiederholt durchgeführt werden.

Geringe Unterschiede zwischen einzelnen Funktionen stellen den kompositionalen Ansatz aber auch vor entsprechende Probleme: Konsequenterweise müssten die Unterschiede in einzelne Funktionen ausgegliedert werden, was den Anteil an möglichen Funktionen deutlich erhöht und gleichzeitig ein Gesamtverständnis der Funktionalität erschwert. Ein entsprechendes Beispiel ist in Abbildung 3.4 dargestellt.

In diesem Beispiel sind die unterschiedlichen Varianten farblich voneinander getrennt. In der oberen Hälfte sind beide Varianten dargestellt. Dabei ist in Variante *B* ein weitere Berechnung, auf Basis einer weiteren Eingabe hinzugefügt worden, die die Berechnung derselben Ausgabe beeinflusst.

In einem kompositionalen Ansatz, wie unten links in Abbildung 3.4 dargestellt, würde man die Software-Komponente nun in drei Funktionen aufteilen müssen, um die möglichen Kompositionen zu realisieren, obwohl diese zur Berechnung derselben Ausgabe genutzt werden. Wäre dies für jede Funktionalität der Software-Komponente notwendig, so würde sich die Anzahl der möglichen Software-Komponentenvarianten deutlich erhöhen. Eine Darstellung in einem 150% Modell ist dagegen weiterhin überschaubar, da die durchschnittliche Größe einer Funktion eher gering ist.

Die Variation ist dabei nur durch einen entsprechenden Zusatz in der Berechnung gegeben. Die Funktion und auch der berechnete Ausgangswert bleiben inhaltlich gleich. Um dort eine Auftrennung zu vermeiden, wird auf dieser Ebene der kompositionale Ansatz mit einem annotativen Ansatz verbunden, um die Vorteile beider Welten zu vereinen. Auf der Ebene einer Funktion sind generell keine größeren Modelle mehr gegeben. Ein Verlust der Übersichtlichkeit durch Hinzufügen der Variationspunkte und Varianten direkt im Modell ist eher ausgeschlossen.

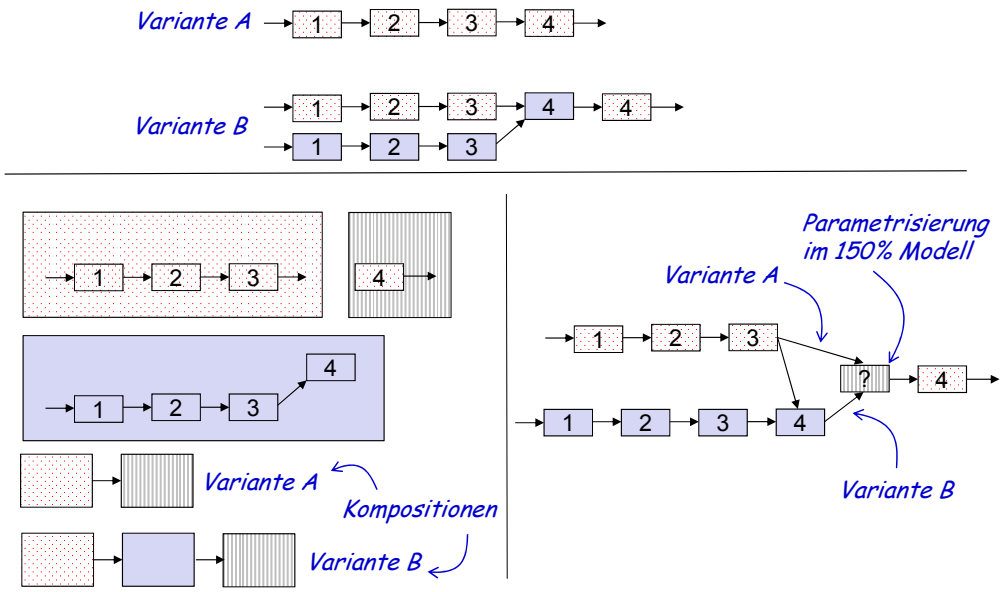


Abbildung 3.4: Darstellung unterschiedlicher Software-Komponenten als Komposition oder als 150% Modell.

Gleichzeitig kann dadurch aber ein Verlust der Übersicht durch zu viele, sehr kleine Einheiten vermieden werden.

Definition 3.5 *Funktionsvariante. Die Variationspunkte einer Funktion sind durch ihre Parameter gegeben. Eine Funktionsvariante wird durch eine unterschiedliche Parametrisierung gebildet.*

Zusammenfassend wird die Variation des Systems durch die Komposition von Software-Komponentenvarianten definiert, die wiederum auf der Komposition von parametrisierten Funktionen basiert. Abhängig von der entsprechenden Parametrisierung ist eine andere Funktionsvariante gegeben. Dieser Zusammenhang ist nochmals in Abbildung 3.5 dargestellt.

Die Software-Plattform definiert ihren Aufbau über die Referenzarchitektur, die eine beliebige Menge an Software-Komponenten enthält. Jede Software-Komponente beinhaltet eine Menge an Funktionen, die wiederum eine beliebige Menge an Parametern zur Verfügung stellen. Für jede Software-Komponente existieren eine beliebige Menge an Varianten, die eine Teilmenge der Funktionen der Software-Komponente beinhaltet. Für jede Funktionsvariante ist eine Parametrisierung definiert, falls die Funktion Parameter beinhaltet.

Ein Software-Produkt besteht aus einer Teilmenge der zur Verfügung stehenden Software-Komponentenvarianten. Sollen mehrere Instanzen einer Software-Komponente gebildet werden können (zum Beispiel für die Software-Komponente *ReifendruckKontrolle* die Software-Komponentenvarianten *ReifendruckKontrolleLinks* und *ReifendruckKontrolleRechts*), so muss weiterhin die Eindeutigkeit aller Signalnamen der Software-Komponentenvariantenschnittstelle gegeben sein. Dies ist über zusätzliche Funktionen möglich, welche nur als Namensadapter fun-

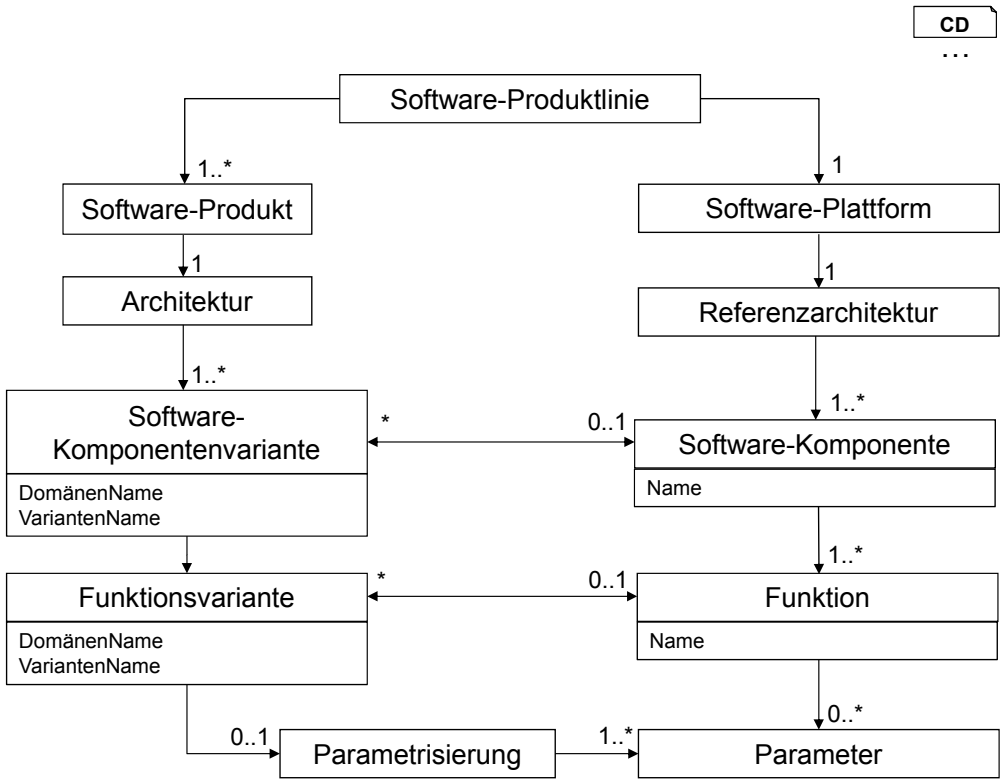


Abbildung 3.5: Kompromiss aus kompositionalem und annotativem Ansatz als Variabilitätsmechanismus im Lösungsraum.

gieren (Adaption des Signals *Reifendruck* zu *ReifendruckLinks*). Des Weiteren wird zwischen dem Domänennamen und dem Variantennamen einer Variante unterschieden. Existiert eine Assoziation zu einer Software-Komponente oder Funktion, so ist der Domänenname gleich dem Namen dieses Artefakts und erlaubt über diesen eine direkt sichtbare Zuordnung. Ist keine zugehöriges Domänenartefakt gegeben, so ist der Domänenname gleich dem Variantennamen. In dem beschriebenen Fall ist *ReifendruckKontrolle* der Domänenname und *ReifendruckKontrolle-Links* der Variantenname.

Abschließend ist zu klären, unter welchen Vorraussetzungen eine Parametrisierung der Funktion erfolgen soll und unter welchen Voraussetzungen eine neue Funktion zwecks Komposition zu definieren ist. Eine Funktion stellt dabei auf einer Menge an Eingabewerten eine Berechnung zur Verfügung, die oft nur einen Ausgabewert produziert. Bezieht sich nun die Variation nur auf die Anpassung der Berechnung des gleichen Ausgabewertes, so sollte diese über eine Parametrisierung realisiert werden. Wird stattdessen ein neuer Wert berechnet, der nur den gegebenen Ausgabewert oder einen Zwischenwert als Eingabe nutzt, so sollte dies durch eine Komposition realisiert werden, da auf diesem Wege unterschiedliche Funktionen extrahiert werden können. Diese stellen dann im eigentlich Verständnis auch keine Varianten der Funktion dar.

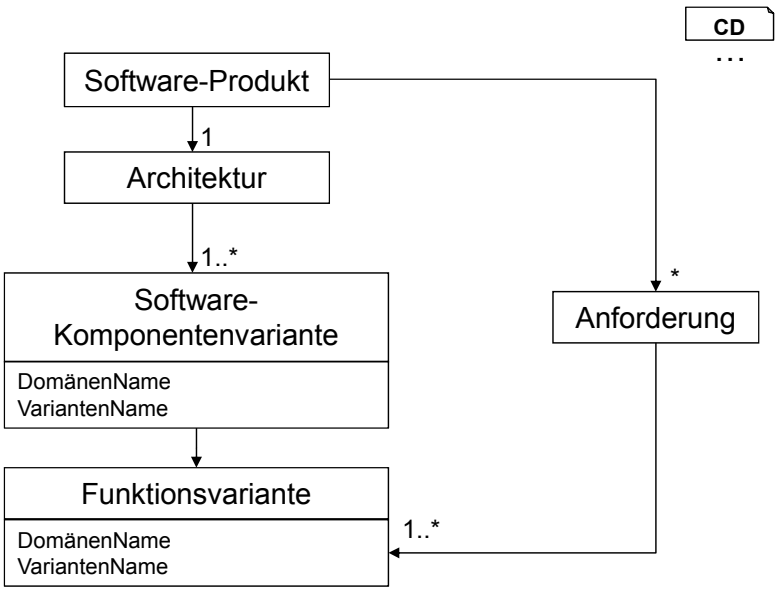


Abbildung 3.6: Nachverfolgbarkeit der Anforderungen in der Software-Produktentwicklung.

3.4 Schrittweiser Übergang von Software-Produkten zur Software-Produktlinie

Ein weiterer Vorteil des beschriebenen Variabilitätsmechanismus ist dadurch gegeben, dass die gegebenen Richtlinien zur Komposition auch schon während der Entwicklung von Software-Komponentenvarianten in der AE ohne direkten Bezug zur DE Verwendung finden können (während Schritt 5 oder 7 aus Abbildung 3.1). Dadurch ist eine spätere Überführung in die Software-Plattform deutlich leichter durchzuführen (Schritt 9 aus Abbildung 3.1). Betrachtet man ausschließlich die Entwicklung einer Software-Komponentenvariante und berücksichtigt die notwendige Nachverfolgbarkeit der Anforderungen (siehe Anforderung **P1** auf Seite 25), so ergibt sich das in Abbildung 3.6 dargestellte Bild.

Die Anforderungen, die durch ein Software-Produkt realisiert werden, können direkt den einzelnen Funktionsvarianten zugeordent werden. Des Weiteren sind diese dann auch indirekt der zugehörigen Software-Komponentenvariante zuzuordnen. Eine Funktionsvariante enthält keine Variationspunkte und insofern ist keine Parametrisierung zum Binden von Variationspunkten notwendig. Wird der Entschluss gefasst, eine Software-Komponente auf Basis einer oder mehrerer Software-Komponentenvarianten zu etablieren, so kann zwischen drei verschiedenen Qualitätsstufen der Software-Komponente bezüglich ihres Detailgrades und ihrer Art der Wiederverwendung unterschieden werden.

Definition 3.6 *Software-Komponentenhülle. Die Software-Komponentenhülle definiert nur ihren Namen und eine grundsätzlichen funktionale Beschreibung und legt somit den Domänennamen der zugehörigen Software-Komponentenvarianten fest. Des Weiteren definiert die Software-*

Komponentenhülle aber keine Menge an zur Verfügung stehenden Funktionen. Entsprechende Software-Komponentenvarianten basieren auf Funktionsvarianten ohne Bezug zu Funktionen.

Software-Komponentenhüllen werden in Schritt 10 aus Abbildung 3.1 definiert.

Definition 3.7 *Software-Komponentenmuster. Das Software-Komponentenmuster definiert neben ihrem Namen eine Menge an zur Verfügung stehenden Funktionen. Die Funktionen sind über Anforderungen mit dem Variantenmodell verknüpft. Des Weiteren sind Parametrisierungen der Funktionen über Anforderungen mit dem Variantenmodell verknüpft. Die Verknüpfung mit dem Variantenmodell legt unter Beachtung der Varianteneinschränkungen die möglichen Kompositionen der Funktionen fest. Die Menge der zur Verfügung stehenden Funktionen kann dabei zur Bildung einer Software-Komponentenvariante durch weitere Funktionsvarianten ohne Bezug zu einer Funktion erweitert werden.*

Software-Komponentenmuster erlauben durch Hinzufügen beliebiger Funktionsvarianten noch größere Freiheiten in der AE.

Definition 3.8 *Software-Komponentenvorlage. Im Unterschied zum Software-Komponentenmuster können bei Bildung einer Software-Komponentenvariante nur die zur Verfügung stehenden Funktionen verwendet werden.*

Software-Komponentenvorlagen stellen den letzten Entwicklungsschritt innerhalb der DE dar. Weitere Anpassungen, die durch neue Anforderung aus der AE notwendig werden, müssen direkt an der Software-Komponente in der DE vorgenommen werden.

Der beschriebene Zusammenhang zwischen Problem- und Lösungsraum, welcher bei der Etablierung eines Software-Komponentenmusters entsteht ist in Abbildung 3.7 dargestellt. Vergleicht man Abbildung 3.7 mit Abbildung 3.6, so wird deutlich, dass bei der Überführung einer Software-Komponentenhülle zu einem Software-Komponentenmuster folgende Schritte notwendig sind (Schritte 2, 6 und 9 in Abbildung 3.1):

1. Zuerst müssen alle Software-Komponentenvarianten einer Software-Komponentenhülle identifiziert werden. Dies ist über den Domänennamen leicht möglich (Schritt 2).

2. Durch Abgleich aller Funktionsvarianten der einzelnen Software-Komponentenvarianten müssen Funktionen, Anforderungen und zugehörige Parameter extrahiert werden, so dass anschließend alle Software-Komponentenvarianten auf Basis der extrahierten Funktionen gebildet werden können (Schritte 6 und 9).

3. Anschließend müssen die Anforderungen der Funktionsvarianten den Funktionen zugeordnet werden. Anforderungen, die eine Parametrisierung bedingen müssen separat aufgeführt und den Parametrisierungen zugeordnet werden (Schritt 9).

4. Variationspunkte, Varianten und Varianteneinschränkungen müssen dem Variantenmodell hinzugefügt und die Varianten mit den Anforderungen verknüpft werden (Schritt 9).

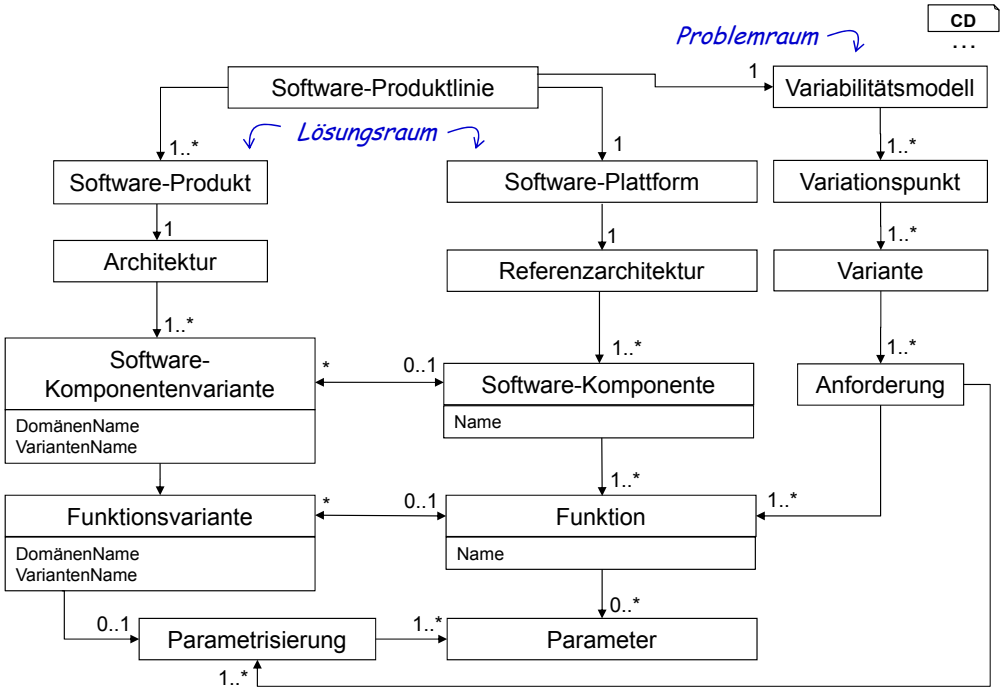


Abbildung 3.7: Zusammenhang zwischen Problem- und Lösungsraum.

Bezogen auf die in Abschnitt 2.2 beschriebene Aktivitäten *Merkmalidentifikation*, *Merkmallokalisierung* und *Reengineering* legt der beschriebene Prozess einen Fokus auf eine automatische Unterstützung des Reengineerings. Eine Merkmallokalisierung ist durch die etablierte Referenzarchitektur leicht möglich. Dabei schreibt der beschriebene Prozess aber in keinster Weise vor, wie eine Verbindung zwischen der Referenzarchitektur und dem Merkmalmodell festzulegen ist. Er sichert nur ab, dass möglichst alle Software-Produkte einer Software-Produktlinie zur Referenzarchitkeur konform sind, obwohl keine proaktive Software-Produktlinienentwicklung durchgeführt wird. Die Merkmal-Identifikation und -Lokalisierung findet insofern während der Anpassungen der Referenzarchitektur (Schritt 10) und der Variabilitätsanalyse (Schritt 8) statt. Durch den vorangegangenen Schritt 6 unterstützen auch die Ähnlichkeitsanalysen die nachfolgende Merkmal-Identifikation und -Lokalisierung.

In den folgenden Kapiteln wird nun zuerst auf automatisierte Ähnlichkeitsanalysen eingegangen, welche einen Abgleich von Software-Komponentenvarianten und Funktionsvarianten unterstützen sollen. In Kapitel 8 wird die Etablierung von Software-Komponentenmustern und -vorlagen durch ein datenbankgestütztes Varianten-, Modell- und Signalmanagement näher erläutert.

Kapitel 4
Ähnlichkeitsanalysen

In Kapitel 3 wurde die Notwendigkeit einer möglichst automatischen Erkennung von ähnlichen Artefakten deutlich, um eine effiziente schrittweise Überführung einzelner Software-Komponentenvarianten in eine Software-Plattform zu ermöglichen. Im Folgenden werden nun als weitere Grundlagen Verfahren aus der Klonerkennung und der semantischen Differenzierung näher beleuchtet, da diese als Fundament für eine Ähnlichkeitserkennung innerhalb dieser Arbeit dienen.

4.1 Klonerkennung

Bei der Klonerkennung wird grundsätzlich zwischen textuellen und graphischen Klonen unterschieden [GKHB10, RBS13, ACD+12, SK16b].

Definition 4.1 *Textueller Klon. Klone, welche in Entwicklungsartefakten enthalten sind, die nur durch eine rein textuelle Repräsentation beschrieben werden. Zugehörige Entwicklungssprachen sind textuelle Entwicklungssprachen.*

Der Codeklon ist ein textueller Klon. In vielen Fällen [RBS13, SK16b] wird der Begriff Codeklonerkennung genutzt, da dieser meist der Hauptanwendungsfall ist, obwohl sich die Erkennung auf textuelle Klone bezieht.

Definition 4.2 *Graphischer Klon. Klone, welche in Entwicklungsartefakten enthalten sind, die auch graphische Informationen beinhalten. Zugehörige Entwicklungssprachen sind graphische Entwicklungssprachen.*

Modellierungssprachen gelten oft als graphische Entwicklungssprachen. Ein Modell stellt im Allgemeinen aber nur eine Abstraktion gegenüber dem modellierten System dar [Rum17], ohne dass dabei eine Aussage über die Art der Darstellung getroffen wird. Graphische Entwicklungsartefakte enthalten gegenüber textuellen Entwicklungsartefakten nur weitere layoutspezifische Informationen [ACD+12], wie zum Beispiel die Position innerhalb des Diagramms, die Farbe oder Orientierung. Diese Attribute können zwar auch zur Identifizierung von Klonen genutzt werden, führen aber auch schnell zu falsch negativen Ergebnissen. Anpassungen an der Positionierung oder farblichen Darstellung erschweren in einem solchen Fall die Identifikation sonst identischer Klone. Werden diese Attribute dagegen in einem Vorverarbeitungsschritt in der textuellen Darstellung des Modells entfernt ist es möglich, direkt Verfahren zur textuellen Kloneerkennung zu nutzen [ACD+12].

Während auf Basis der in graphischen Entwicklungsartefakten gegebenen zusätzlichen Attribute auch separate Klassifizierungen für textuelle [RCK09, Kos07] und graphische Klone [GKHB10, Stö13] existieren, können diese auf Grund der genannten Gemeinsamkeiten auch unter folgender Definition zusammengeführt werden [RBS13, SK16b]:

1. **Typ-1 (exakte Klone)**: Identische Fragmente, die sich nur durch Abweichungen in Kommentaren und Leerzeichen oder Absätzen unterscheiden. Im Kontext von graphischen Entwicklungsartefakten darf eine Anpassung des Layouts keinen semantischen Einfluss haben. In manchen graphischen Sprachen nimmt zum Beispiel die Anordnung einzelner Elemente Einfluss auf die Ausführungsreihenfolge [GKHB10].

2. **Typ-2 (Umbenannte / parametrisierte Klone)**: Fragmente, die sich zuzüglich zu den durch Typ-1 definierten Änderungen nur durch Änderungen der Identifikatoren, Literalen und Typen unterscheiden.

3. **Typ-3 (Sehr ähnliche Klone)** Zusätzlich durch die in Typ-2 definierten Änderungen, können sich die Klone auch durch Hinzufügen oder Hinwegnehmen einzelner Anweisungen bzw. Elemente unterscheiden.

4. **Typ-4 (Semantische Klone)** Funktional identische Fragmente, die sich aber syntaktisch und strukturell beliebig unterscheiden können.

Die Klassifikationen von Typ-1 bis Typ-3 führen schrittweise neue Arten möglicher Unterschiede ein, so dass Verfahren, die in der Lage sind, Typ-3 Klone zu erkennen, auch Typ-2 Klone erkennen. Der semantische Klon stellt dagegen eine sehr allgemeine Form eines Klons ohne notwendige syntaktische Ähnlichkeit dar. Insofern ist nur ein bedingter Zusammenhang zu den vorherigen Klassifizierungen gegeben. In [GKHB10] wird im Kontext von graphischen Klonen ein Typ-0 zur Klassifizierung exakter Klone ohne Anpassung des Layouts eingeführt, da dieser Aspekt bei graphischen Entwicklungsartefakten eine größere Rolle spielen kann. Auf Basis der Argumentation von [ACD+12] stellt die Verarbeitung von layoutspezifischen Informationen für die Identifikation höherstufiger Klone eher ein Problem dar. Des Weiteren sind Layoutanpassungen oft üblich und stellen für die Wiederverwendung im Kontext einer Softwareproduktlinie kein Hindernis dar, so dass diese Unterscheidung im Folgenden nicht weiter betrachtet wird.

Des Weiteren erwähnt [RBS13] weiterhin explizit „strukturelle Klone", die sich auf Design- und Analyseartefakte auf Ebene der Architektur beziehen sowie Funktionsklone, die explizit nur Klone auf einer spezifischen Granularitätsstufe, wie der Funktion oder der Prozedur, beinhalten. Diese Art der Klontypen finden in [RBS13] und [SK16b] aber wenig Beachtung.

Für die in Kapitel 3 beschriebene produktgetriebene Softwareproduktlinienentwicklung kann während einer schrittweisen Extraktion die Identifikation aller vier Klontypen relevant sein. Zur Unterstützung eines *Clone&Own*-Ansatzes wäre meist nur die Identifikation von historischen Klonen notwendig. Diese würden auf Grund ihrer Historie syntaktisch ähnliche Elemente enthalten, was in den meisten Fällen Klone bis zum Typ-3 entspricht. Nur durch intensivere Refactoringmaßnahmen ohne zugehörige funktionale Änderungen kann nach Anlegen einer Kopie ein Typ-4 Klon entstehen. Bei einer parallelen Entwicklung ähnlicher Funktionalität, welche ohne

größere Absprachen erfolgt, sind dagegen grundsätzlich Typ-4 Klone als Ergebnis zu erwarten, da syntaktische Gemeinsamkeiten eher zufällig entstehen würden.

Die Identifikation von Ähnlichkeiten stark modifizierter Kopien als auch paralleler Entwicklung ist für eine produktgetriebene Softwareproduktlinienentwicklung auch relevant. Gleichzeitig ist die Identifikation von semantischen Ähnlichkeiten historischer Klone für eine Extraktion hilfreich ist. Insofern sind alle vier beschriebenen Klontypen und Verfahren, welche diese erkennen können, für eine Ähnlichkeitsanalyse relevant. Wie schon in Kapitel 3 aufgeführt sind für eine frühzeitige Ähnlichkeitsanalyse gerade auch Artefakte, die nur die Struktur der Software beschreiben, von Bedeutung. Leider sind zur Erkennung von strukturellen Klonen wenige Ansätze gegeben. Gerade Verfahren mit einem reinen Fokus auf Schnittstellendefinitionen werden nicht aufgeführt [RBS13, RCK09].

Neben der Klassifizierung der Klontypen können auch die angewandten Verfahren auf Basis ihres Anwendungsbereichs sowie der verwendeten Technik klassifiziert werden.

Es wird zwischen folgenden Ansätzen unterschieden [RBS13, RCK09]:

1. **Textbasierte Ansätze**: Diese Ansätze versuchen auf Basis einer textuellen Repräsentation des Fragments (normalerweise direkt der Code) ohne intensivere Vorverarbeitungsschritte Klone zu identifizieren. Vorteil dieser Herangehensweise ist, dass diese Ansätze meist sprachenunabhängig angewendet werden können. Dabei werden unterschiedliche Mechanismen, wie Fingerprinting [Joh94], zeilenbasierte Analyse [DRD99, WM05] oder positionsbasierte Analysen [LJ05] verwendet, um Klone zu erkennen.

2. **Lexikalische Ansätze**: Basis dieser Analysen ist eine Überführung des Textes in lexikalische *Token*. Anstatt einer zeilenbasierten Analyse werden Token zuerst klassifiziert (zum Beispiel Unterscheidung von Identifizierern und Literalen und sonstigen Tokens) und anschließend analysiert. Durch die initiale Überführung sind nachfolgende Analysen robuster gegenüber Umbenennungen, da sie losgelöst von konkreten Benennungen durchgeführt werden können (Typ-2). Üblicherweise wird ein Suffixbaum [McC76] oder Suffixarray verwendet, um im nächsten Schritt potentielle Klone zu identifizieren [Bak95, KKI02, BJ07]. Des Weiteren finden Data-Mining Ansätze [LLMZ06] sowie Hybride aus token- und zeilenbasierten Ansätzen Verwendung [SCD03].

3. **Baumbasierte Ansätze**: Die Überführung der Quellen in einen abstrakten Syntaxbaum (AST) ermöglicht eine leichte Identifikation von Typ-3 Klonen, bei gleichzeitig höherem Aufwand durch notwendige Vorverarbeitungsschritte. Entsprechende Verfahren basieren größtenteils auf dynamischer Programmierung [Yan91, BYM+98]. Neuere Verfahren extrahieren charakteristische Vektoren und vergleichen diese, um Ähnlichkeiten zu bestimmen [NNP+09, SWP+09]. Um die Genauigkeit eines baumbasierten Ansatzes mit der Geschwindigkeit einer lexikalischen Analyse zu verbinden, nutzen [TG06] und [FFK08] wiederum einen Suffixbaum auf Basis des ASTs.

4. **Graphenbasierte Ansätze**: Der *Program Dependency Graph* (PDG) [FOW87] wird traditionell im Kontext der Kompileroptimierung verwendet, findet aber auch im Kontext der Klonerkennung Verwendung. Dabei können entsprechende Verfahren auch semantische Klone (Typ-4) identifizieren. Dazu müssen isomorphe Teilgraphen identifiziert werden;

ein Problem, das allgemein als NP-vollständig eingestuft wird [BB76, BSJL92, KH01, Kri01, Hor90]. Um das Problem zu umgehen, verwenden [KH01, Hor90] Slicing [Tip95] (und identifizieren somit isomorphe Slices anstatt generelle Teilgraphen). In [GJS08] wird eine Beziehung zwischen dem PDG und der zugehörigen syntaktischen Struktur aufgebaut, um anschließend auf Basis eines vektorbasierten Ansatzes Klone zu identifizieren. [HK11] kombiniert *Forward Slicing* und *Backward Slicing* und verwendet Heuristiken bezogen auf minimale Klongröße sowie maximale Größe von Äquivalenzklassen, um sowohl die Qualität des Ergebnisses als auch die Performanz zu erhöhen. Eine weitere Möglichkeit praktikable Ergebnisse zu erlangen, ist durch eine Begrenzung der maximalen Länge der zu identifizierenden Teilgraphen gegeben [Kri01]. Modellbasierte Ansätze überführen das gegebene Modell meistens erst in eine graphenbasierte und normalisierte Form [DHJ+08, ABSH11, PNN+09], um anschließend Klone zu identifizieren. Dabei unterscheidet sich die Normalisierung im Detail. Während [DHJ+08] nur irrelevante Details entfernt, werden in [ABSH11] auf Basis mathematischer Regeln (z.B. Normalisierung trigonometrischer Funktionen oder Anwendung des Distributionsgesetzes) Transformationen durchgeführt, um auch teilweise semantische Klone (Typ-4) identifizieren zu können. In [HJS11] wird ein inkrementeller, indexbasierter Ansatz verfolgt, um sich wiederholende Klonanalysen auf sich weiterentwickelnden Modellen zu beschleunigen. [MRR14, MRR11a, MRR11c] überführen Aktivitätsdiagramme (ADs) in binäre Entscheidungsdiagramme („Binary Decision Diagrams", BDDs), um semantische Unterschiede zu erkennen. Dabei wird der Zustandsraum beider Aktivitätsdiagramme abgebildet und überprüft, inwiefern es endliche Pfade gibt, deren letzte Zustände bezüglich Ein- und Ausgabeverhalten nicht übereinstimmen. Ein weiterer Ansatz semantische Unterschiede zu erkennen, ist in [MRR11b] auf Basis von Klassendiagrammen (CDs) gegeben. Die CDs, welche Teil der Sprachfamilie UML/P [Rum16, Rum17] sind und unter Verwendung von MontiCore [KRV10] entwickelt wurden, werden dabei in Alloy [Jac12] überführt, um gegebene semantische Beziehungen durch einen SAT-Solver auszuwerten. Ziel ist es dabei mögliche endliche Instanzen der CDs (Objektdiagramme) zu identifizieren, die nur durch eines der verglichenen Klassendiagramme instanziierbar sind. In [HWL+14a] wird ein datenfluss-orientierter Ansatz zur Klonanalyse von Funktionsblockdiagrammen verfolgt. Dieser vergleicht entsprechend des Datenflusses schrittweise einzelne Blöcke auf Basis ihrer Ähnlichkeit bezogen auf Name, Schnittstelle und der Ähnlichkeit benachbarter Blöcke.

5. **Metrikbasierte Ansätze**: Metrikbasierte Ansätze bewerten auf Basis unterschiedlicher Attribute in einem ersten Schritt die einzelnen Fragmente und stellen diese Bewertung als Vektor dar. In einem zweiten Schritt werden dann diese Vektoren verglichen, um Klone zu identifizieren. Dabei findet die Bewertung auf unterschiedlichen Strukturen statt, wobei meist eine Überführung in einen AST oder einen KFG (*Kontrollflussgraph*) verwendet wird. Metriken werden dabei zum Beispiel auf Basis von Namen, Layout und Kontrollflüssen erhoben [MLM96]. [PMDL99] verwendet die Messgrößen Anzahl Anweisungen, zyklomatische Komplexität, Anzahl Parameter, Verwendung nicht-lokaler Variablen und die Anzahl lokaler Variablen innerhalb einer Methode, um Klone zu identifizieren. [HKI08] verwendet Metriken, um identifizierte Klone bezüglich ihres Refactoringaufwandes zu be-

werten. [BT14] kombinieren sieben verschiedene Metriken, wobei die beste Kombination durch schrittweises Hinzufügen einzelner Metriken und zugehöriger Bewertung durch gegebenes Benchmark identifiziert wurde. [SK16a] extrahieren 70 Merkmale auf Basis von AST und PDG und verwenden maschinelles Lernen zur Identifikation der Klone. In [WTS+16] werden Klone von Klassen objektorientierter Systeme auf Basis einzelner Metriken bezogen auf deren Methoden und Attributen identifiziert. Dabei werden einzelne Aspekte , wie Datentyp, Modifizierer oder Name, unterschiedlich gewichtet. Eine hohe Gewichtung der Namen der Methoden und Attribute gegenüber anderen Aspekten führte dabei zu besseren Ergebnissen.

6. **Hybride Ansätze**: Hybride kombinieren unterschiedliche Methoden aus den einzelnen genannten Ansätzen, um das Ergebnis zu verbessern. Die Kombination von PDGs und syntaktischer Struktur [GJS08] sowie die Verwendung eines Suffixbaumes auf Basis eines ASTs [TG06, FFK08] stellen entsprechende Hybride dar.

Zwischen 2009 und 2016 wurde in drei verschiedenen Arbeiten eine größere Übersicht über den aktuellen Stand der Forschung im Kontext der Klonerkennung erarbeitet [RCK09, RBS13, SK16b]. Während in [SK16b] und [RBS13] durch Literaturrecherche bzw. einem systematischen Literaturreview nur die Ergebnisse anderer Arbeiten zusammentragen wurden, haben

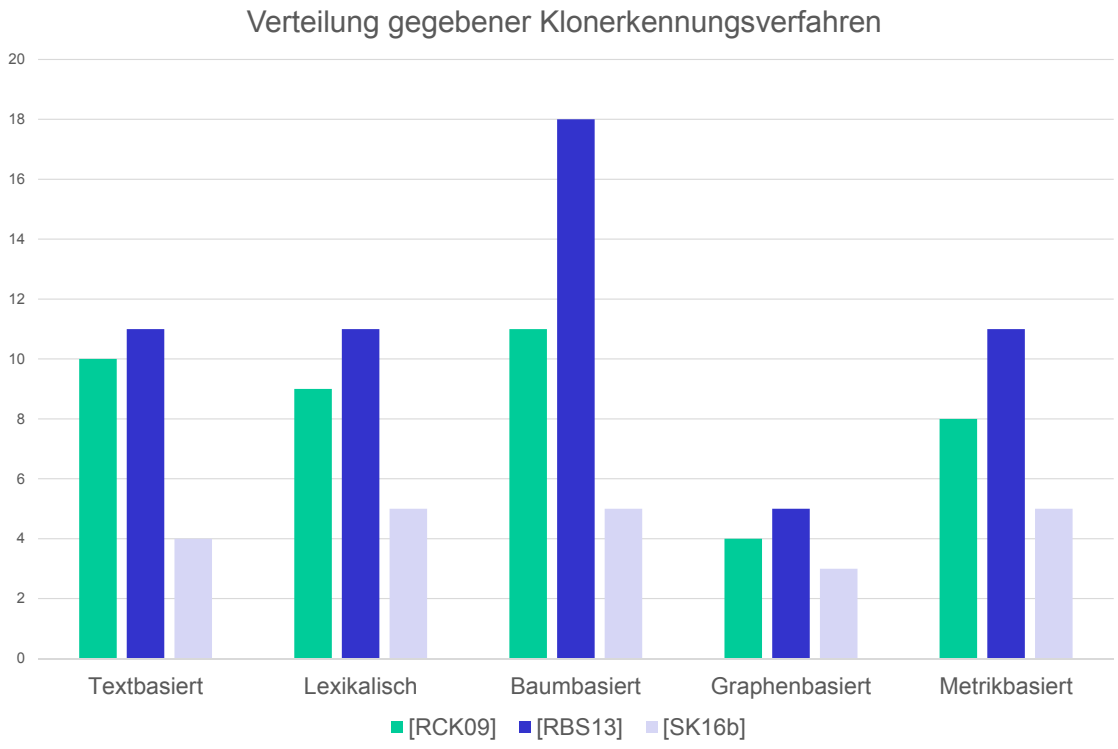


Abbildung 4.1: Verteilung gegebener Klonerkennungsverfahren (basierend auf [RCK09, RBS13, SK16b]).

[RCK09] die identifizierten Verfahren auch anhand vier verschiedener Szenarien (entsprechend der vier Klontypen) evaluiert. Alle Arbeiten teilen die vorgestellten Ansätze in die aufgeführten Kategorien ein, so dass ein direkter Vergleich bezüglich der aktuellen Verteilung von Klonerkennungsverfahren direkt möglich ist. Dieser Vergleich ist in Abbildung 4.1 dargestellt. Grundsätzlich basiert [SK16b] auf einer geringeren Menge an Arbeiten und es wurde dort auch versucht, möglichst eine Gleichverteilung der betrachteten Kategorien umzusetzen. Vergleicht man die einzelnen Evaluierungen, so stellen insgesamt baumbasierte Verfahren den stärksten Vertreter dar, während graphenbasierte Verfahren deutlich schwächer vertreten sind. Während die Anzahl baumbasierter Verfahren aber nur in [RBS13] überproportional vertreten sind, stellen graphenbasierte Verfahren grundsätzlich eine Minderheit dar. Auch modellbasierte Verfahren stellen im Gegensatz zu textbasierten Verfahren eine deutliche Minderheit dar [RCK09, RBS13, SK16b, HJS11, ABSH11].

Betrachtet man in einem zweiten Schritt die Fähigkeit der einzelnen Kategorien, die unterschiedlichen Klontypen zu identifizieren, so ergibt sich auf Basis von [RCK09, RBS13, SK16b] die durch Abbildung 4.2 und Abbildung 4.3 dargestellte Verteilung. Während [RCK09] eine detaillierte Evaluierung vorweist, die eine feingranulare Abstufung der Leistungsfähigkeit ermöglicht, führt [RBS13] die generelle Befähigung auf, eine Klonkategorie zu erkennen. In [SK16b] wird eine entsprechende Relation zur Klonkategorie ohne Verfahrenskategorie zusammengefasst.

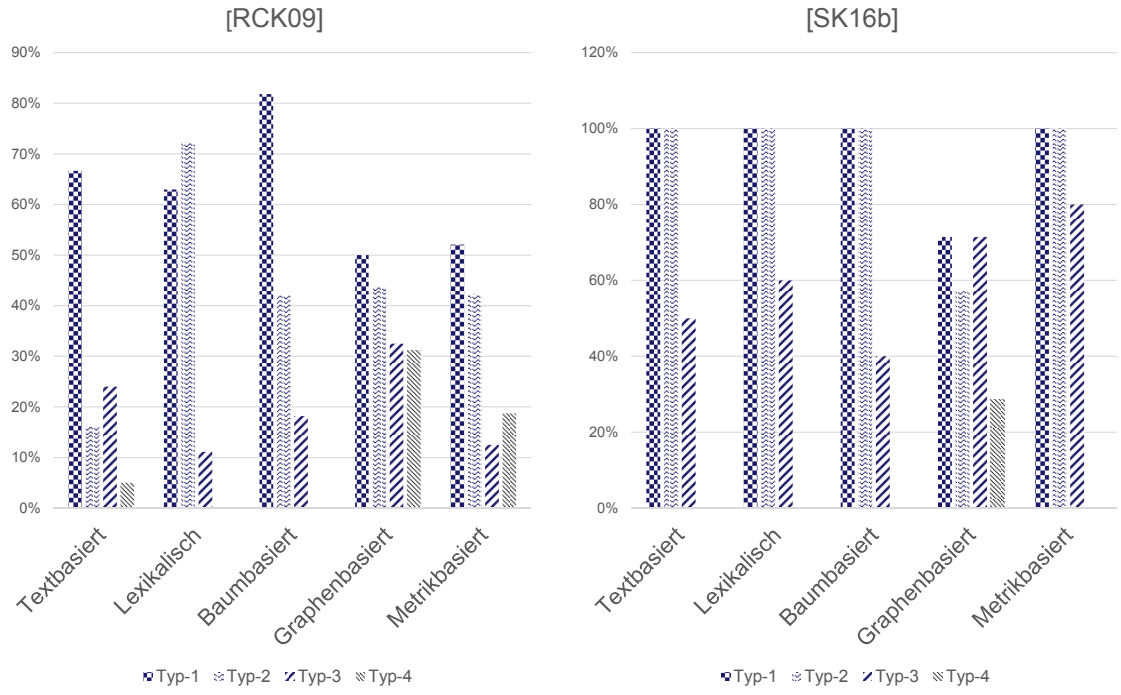


Abbildung 4.2: Adressierte Klontypen je verwendetem Erkennungsverfahren (basierend auf [RCK09, RBS13]).

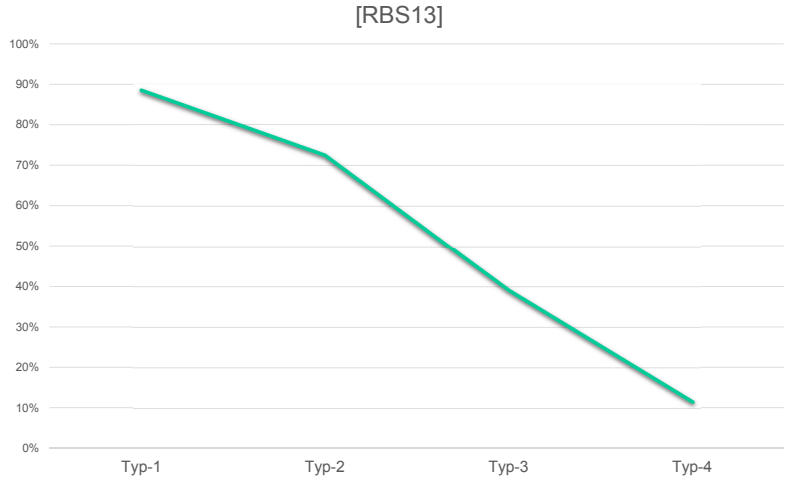


Abbildung 4.3: Adressierte Klontypen (basierend auf [SK16b]).

Vergleicht man die Ergebnisse aus [RCK09] und [RBS13] ergibt sich ein sehr ähnliches Bild, auch wenn in [RCK09] die generelle Leistungsfähigkeit bezogen auf einen Klontyp der einzelnen Verfahren mit abgebildet werden konnte (bezogen auf den durch den Klontypen gegenüber dem Vorgänger neu eingeführten Änderung). Nur graphenbasierte Verfahren ermöglichen in allen Kategorien ein durchschnittliches Ergebnis zu erreichen, während baumbasierte und lexikalische Verfahren sich am besten für Typ-1 und Typ-2 Klone eignen. Textbasierte Verfahren scheitern meist bei größeren Umbenennungen und Typ-3 Klone stellen alle Kategorien im Durchschnitt vor größere Probleme. Semantische Klone (Typ-4) werden dagegen von den meisten Verfahren gar nicht adressiert. Nur graphenbasierte Verfahren sind in der Lage diese zu erkennen.

Die Verteilung in Abbildung 4.3 stellt ein ähnliches Bild dar. Hierbei wird aber nur die explizite Erwähnung des zu identifizierenden Klontypen innerhalb der recherchierten Arbeiten aufgeführt, was den geringen Anteil an Typ-1 Klonen erklärt. Während die Identifikation exakter Klontypen in der Forschung kein Thema mehr darstellt, ist die Identifikation semantischer Klone weiterhin nur durch wenige Verfahren addressiert.

Betrachtet man den Ursprung von Klonen, so wird dieser in der Verwendung des *Clone&Own*-Ansatzes gesehen: Historische Klone werden bewusst auf Grund einer gemeinsamen Basis erstellt und dann separat voneinander fortentwickelt. Insofern kann man in den meisten Fällen von einer syntaktischen Basis ausgehen (Typ-1,Typ-2, Typ-3). Die Betrachtung parallel voneinander unabhängig entwickelter Elemente mit gleicher oder ähnlicher Funktionalität ist erst Bestandteil neuerer Arbeiten [WAB+16, JDH10, JS09, EL12, PR14]. Dabei wurde festgestellt, dass semantische Klone oft nur eine sehr geringe syntaktische Ähnlichkeit aufweisen [WAB+16, JDH10]. Alle aufgeführten Verfahren analysieren entweder Programmcode oder Modelle (meist Simulink), während sich nur wenige Arbeiten auf strukturelle Aspekte stützen [BJ09, BJ05, BAJ11].

Zusammenfassend sind die vorgestellten Verfahren in der Lage mit Klonen bis zu Typ-3 umzugehen, wobei sich die *Precision* (Anzahl korrekt positiver / Anzahl identifizierter Klone) und

der *Recall* (Anzahl korrekt positiver / Anzahl existierender Klone) der Verfahren unterscheidet. Da einzelne Kategorien für unterschiedliche Klontypen besonders gut geeignet sind, ist die Verwendung eines Hybriden oder die parallele Anwendung mehrerer Verfahren empfehlenswert.

Gleichzeitig stellen die vorgestellten Arbeiten den Stand der Forschung dar. Im Kontext beteiligter Industriepartner und einer möglichst industrietauglichen Verwendung gegebener Klonerkennungsverfahren ist es auch notwendig, den beschriebenen Stand mit der aktuell innerhalb von Simulink gegebenen praktisch einsetzbaren Werkzeuglandschaft abzugleichen. Dabei ist es wiederum notwendig die verwendeten Verfahren den aufgeführten Klontypen zuzuordnen.

4.2 Praktisch einsetzbare Klonerkennungsverfahren für Simulink

Im Folgenden werden die Leistungsfähigkeit der Klonerkennungsverfahren der Werkzeuge *SimDiff v4.5.1.0*[1], *dSpace ModelCompare v2.6*[2] und *Simulink Report Generator v5.0* [3] bezogen auf die unterschiedlichen Klonkategorien im Detail vorgestellt. Diese Werkzeuge stellen die aktuell für Simulink zur Verfügung stehenden und breiter im Einsatz befindlichen Produkte zur Klonerkennung dar.

Der Vergleichsalgorithmus des *Simulink Report Generators* basiert auf [CRGW96] und analysiert die zugrundeliegende XML-Struktur der Simulinkmodelle, während die Dokumentationen von *SimDiff4* und *dSpace ModelCompare* nicht näher auf das zugrundeliegende Verfahren eingehen. Grundsätzlich ließ sich somit leider nicht erkennen, welche Verfahrenskategorien zum Einsatz kommen. Um eine Kategorisierung der Werkzeuge entsprechend Abschnitt 4.1 zu ermitteln, wurde, ähnlich wie in [RCK09], für jede Klonkategorie zugehörige Szenarien aufgebaut. Die Simulink-Modelle der Szenarien wurden so gewählt, dass die Unterschiede immer nur einen bewussten Aspekt betrafen, um zu identifizieren, inwiefern diese Änderungen beziehungsweise ihr Ursprung noch erkannt werden. Dies ermöglicht, abhängig von den Änderungen, die eine Erkennung des Klons verhindern, einen Rückschluss auf den Klontypen zu schließen, welcher durch das evaluierte Verfahren erkannt werden kann. Abschließend ist es dann möglich eine Zuordnung, wie in [RBS13, RCK09] gegeben, auch für die genannten Werkzeuge durchzuführen.

Da man der Dokumentation bzw. den Konfigurationsmöglichkeiten der Werkzeuge außerdem entnehmen konnte, dass sie auch layoutspezifische Informationen verarbeiten können, wurde dieser Aspekt und dessen Einfluss mit betrachtet. Insofern sind alle Werkzeuge darauf ausgerichtet graphische Klone zu erkennen.

Zur Evaluierung wurden folgende Simlunk-Modelle mit dem Basismodell *Simple* (siehe Abbildung 4.4), das nur eine einzelne Rechenoperation enthält, verglichen:

1. **Layout**: Das gesamte Layout des Simulink-Modells *Simple* wurde geändert, ohne dass dadurch eine semantische Änderung entsteht (was in Simulink generell nicht möglich ist - Typ-1 Klon).

[1] `http://www.ensoftcorp.com/DE/simdiff/`
[2] `https://www.dspace.com/de/gmb/home/products/sw/pcgs/model_compare.cfm`
[3] `https://de.mathworks.com/products/SL_reportgenerator/features.html`

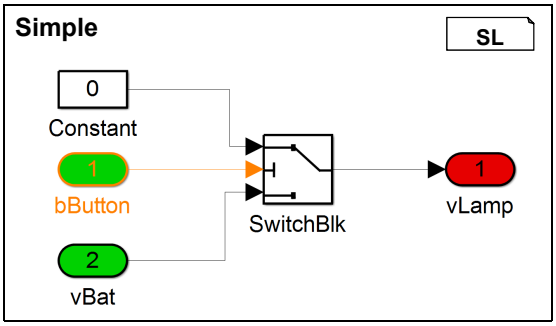


Abbildung 4.4: Basismodell *Simple* zur Evaluierung unterschiedlicher graphischer Klonerkennungsverfahren.

2. **SmallRename**: Teile des Namens der vorhandenen Blöcke wurden umbenannt (Leichter Typ-2 Klon).
3. **CompleteRename**: Der gesamte Name der vorhandenen Blöcke wurde umbenannt (Schwerer Typ-2 Klon).
4. **CompleteRenameLayout**: Neben der Umbenennung der Namen der vorhandenen Blöcke wurde außerdem das gesamte Layout geändert (notwendig zur Identifikation, inwiefern schwere Typ-2 Klone nur auf Basis von Layout-Informationen identifiziert werden).
5. **Added**: Ein neuer Block wird hinzugefügt (Typ-3 Klon).
6. **Removed**: Ein neuer Block wird entfernt (Typ-3 Klon).
7. **Hierarchy**: Mittels Refactoring wurde eine Hierarchieebene hinzugefügt (dient zur Identifikation, in welchem Kontext Klone erkannt werden können).
8. **Semantic**: Dieselbe Funktionalität wird syntaktisch anders beschrieben (leichtes Beispiel (Verwendung eines anderen Basismodells): A && B oder B && A).

Auf Basis dieses Vergleichs ist das Ergebnis für alle Werkzeuge sehr ähnlich, wie in Abbildung 4.5 dargestellt.

Alle Werkzeuge sind ungeachtet von Anpassungen des Layouts, kleineren Namensanpassungen und dem Hinzufügen oder Entfernen einzelner Blöcke, in der Lage die gemeinsamen Aspekte als Klone zu identifizieren.

Eine komplette Anpassung der Namen ist nicht durch *SimDiff 4* unterstützt, wobei eine zusätzliche Anpassung des Layouts deutlich macht, dass *dSpace ModelCompare* und *Simulink Report Generator* nur unter Berücksichtigung von layoutspezifischen Informationen in der Lage sind, den Klon zu erkennen.

Ein weiteres Problem stellt das Hinzufügen oder Entfernen von Hierarchieebenen dar. Keines der Werkzeuge ist in der Lage, anschließend den Klon zu erkennen. Dies deutet deutlich darauf hin, dass Klone nur innerhalb derselben Hierarchieebene gesucht werden können. Da alle drei

Abbildung 4.5: Identifizierte Klone durch die Werkzeuge *dSpace ModelCompare*, *Simulink Report Generators* und *SimDiff4*.

Hersteller mit der hohen Performanz der Klonanalyse werben und diese auch durch probeweise Anwendung an größeren Modellen bestätigt werden konnte, ist dieser Umstand nicht sehr verwunderlich. Leider führt dies aber dazu, dass bei größeren Refactoringmaßnahmen (durch Umbenennung bei gleichzeitiger Modifikation des Layouts oder durch Anpassung der Architektur) eine Klonerkennung nicht mehr möglich scheint.

Weniger überraschend ist, dass auch kein semantischer Klon erkannt wird. Die in Abbildung 4.5 dargestellte Bewertung stellt nur bezüglich der aufgeführten Kriterien eine Aussage dar und kommt zu dem Schluss, dass die Werkzeuge nur unter milden Rahmenbedingungen (keine größeren Refactoringmaßnahmen) in der Lage sind, Klone bis zu Typ-3 zu erkennen. Welches der Werkzeuge in der praktischen Anwendung in unterschiedlichen Kontexten am besten zu verwenden wäre, wurde nicht evaluiert. Auch eine detaillierte Analyse bezüglich Precision und Recall wurde nicht durchgeführt.

Zusammenfassend verfügen die Werkzeuge bezüglich der Ergebnisdarstellung, Unterstützung verschiedenster Simulinkversionen, Unterstützung beim Zusammenführen einzelner Varianten und Analyse großer Modelle über die von einem kommerziellen Produkt zu erwartende Qualität. Bezüglich ihrer Mächtigkeit decken aber sowohl die industriellen Werkzeuge als auch die Ansätze aus der Forschung nicht alle möglichen Klontypen ab oder betrachten alle zur Verfügung stehenden Artefaktrollen.

4.3 Family Model Mining

Ein *Familienmodell* beschreiben im Lösungsraum alle möglichen Konfigurationen einer Software-Plattform und geben somit einen Überblick über ihre implementierungs-spezifische Variabilität [WHSS13]. Beim *Family Model Mining* wird versucht auf Basis gegebener Produkte ein Familienmodel zu extrahieren [WHSS13, HWL+14b]. Im Familienmodell wird zwischen notwendigen, optionalen, und alternativen Teilen unterschieden. Dabei ist es - im Gegensatz zur Klonerkennung - nicht ausreichend nur Ähnlichkeiten zu identifizieren. Stattdessen müssen auch die Unterschiede zwischen den einzelnen Produkten genauer analysiert werden, um zu identifizieren, inwiefern ein optionaler, notwendiger oder alternativer Teil gegeben ist.

Der in Kapitel 3 beschriebene Prozess unterscheidet nicht explizit zwischen notwendigen, alternativen oder optionalen Funktionen oder Parametrisierungen. Eine Software-Komponente definiert nur alle möglichen Funktionen und Parametrisierungen. Inwiefern diese notwendig, alternativ oder optional sind, wird nur indirekt über das Variantenmodell und zugehörige Einschränkungen festgelegt. Die Wartung einer weiteren Beschreibung der Variabilität im Lösungsraum stellt gerade im Hintergrund häufigerer Änderungen durch eine reaktive Herangehensweise auch einen nicht zu unterschätzenden Aufwand dar. Gleichzeitig erlaubt eine entsprechende Übersicht bei größeren Software-Komponenten den Überblick zu bewahren.

Insofern stellt das Familienmodell eine mögliche Erweiterung der produkt-getriebenen Software-Produktlinienentwicklung dar. Diese wird im Kontext dieser Arbeit aber nicht weiter verfolgt.

In den folgenden Kapiteln werden nun für den in Kapitel 3 definierten Prozess und auf Basis der in diesem Kapitel gesammelten Erkenntnisse unterschiedliche Verfahren zur Ähnlichkeitsanalyse vorgestellt, die zuzüglich zu den bisher vorgestellten Verfahren Verwendung finden können. Dabei werden die Artefakte Schnittstellendefinition, Simulink-Modell und Testspezifikation und die Erkennung struktureller und semantischer Klone fokussiert, um die aufgezeigten Lücken möglichst zu schließen.

Kapitel 5

Strukturelle Ähnlichkeitsanalyse

Wie in Kapitel 3 erläutert, bedarf es für eine effiziente teilautomatisierte Softwareproduktlinienextraktion einer schrittweisen Reduktion des Suchraumes. Diese soll in drei Schritten erfolgen: extrinsische Analyse, Analyse der Schnittstellen und abschließend eine semantische Analyse, um entsprechende Ähnlichkeiten zwischen Softwarekomponentenvarianten zu identifizieren.

Im Folgenden wird nun zuerst auf die strukturellen Ähnlichkeitsanalyse bezogen auf die Referenzarchitektur und die Schnittstellendefinitionen eingegangen. Im anschließenden Kapitel werden Vefahren zur semantischen Ähnlichkeitsanalyse genauer beschrieben.

5.1 Extrinsische Analyse

Die im Folgenden beschriebene extrinsische Analyse bezieht sich auf die durch die Software-Plattform definierte Referenzarchitektur. Grundsätzlich lässt sich der Begriff aber auch allgemeiner formulieren.

Definition 5.1 *Das **extrinsische Merkmal eines Entwicklungartefaktes** ist die Beschreibung der grundsätzlichen Funktionalität des Artefaktes losgelöst von variantenspezifischen Details.*

Das extrinische Merkmal wird in der Domänenentwicklungsphase für jedes Domänenartefakt separat definiert und muss innerhalb der Domänenentwicklung eindeutig sein. Das extrinsische Merkmal eines Anwendungsartefaktes ist auf eine für den jeweiligen Artefakttypen festgelegte Art und Weise aus den extrinsischen Merkmalen der verwendeten Domänenartefakte zusammengesetzt. Die extrinsichen Merkmale der Anwendungsartefakte sind in der Anwendungsentwicklung nicht eindeutig. Sie ermöglichen aber eine eindeutigen Rückschluss auf die ursprünglichen Domänenartefakte.

Das extrinsische Merkmal ermöglicht über die Software-Plattform und deren Domänenartefakte eine eindeutige Identifikation einzelner Anwendungsartefakte unterschiedlicher Projekte mit demselben Ursprung (denselben Domänenartefakten), insofern die Vorgaben der Software-Plattform eingehalten wurden.

Das extrinische Merkmal von Software-Komponentenvarianten und Funktionsvarianten ist durch ihren Domänennamen (siehe Abbildung 3.5 auf Seite 38) gegeben. Der Domänenname ist dabei mit dem Namen des ursprünglichen Domänenartefaktes identisch. Sind die Namen zweier Anwendungsartefakte identisch, so sind diese extrinsisch äquivalent.

Dazu muss entweder ein flacher Namensraum gegeben sein oder der Domänenname ist über den durch die Referenzarchitektur gegebenen vollqualifizierten Namen des Domänenartefaktes definiert.

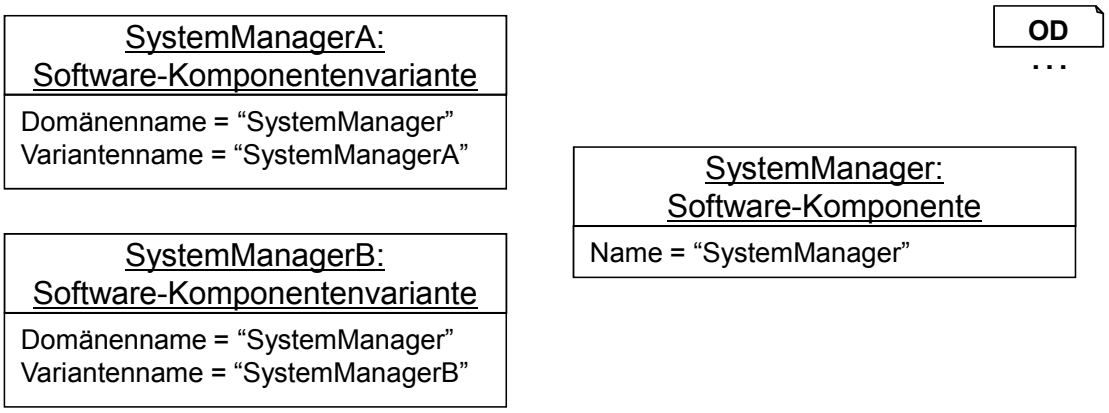


Abbildung 5.1: Gemeinsames extrinisches Merkmal zweier Software-Komponentvarianten und ihrer Software-Komponente.

Die extrinsische Analyse überprüft die extrinsische Äquivalenz (Namensgleichheit bezogen auf den Domänennamen) von Software-Komponentenvarianten und Software-Komponenten, sowie von Funktionsvarianten und Funktionen und identifiziert extrinische Paare.

Definition 5.2 *Extrinsisches Paar. Sind zwei Entwicklungsartefakte extrinsisch äquivalent so bilden sie ein extrinsiches Paar.*

Ein entsprechendes Beispiel ist in Abbildung 5.1 aufgeführt. Die beiden Software-Komponentenvarianten *SystemManagerA* und *SystemManagerB* basieren auf der Software-Komponente *SystemManager*. Der Domänenname beider Software-Komponentenvarianten ist dabei identisch zum Namen der Software-Komponente. Das extrinsche Merkmal aller drei Entwicklungsartefakte ist somit identisch und *(SystemManagerA, SystemManagerB)*, *(SystemManagerA, SystemManager)* und *(SystemManagerB, SystemManager)* bilden extrinsche Paare.

Basiert die Entwicklung einer Funktions- oder Software-Komponentenvariante nicht auf einer Software-Komponente oder Funktion (wie in Abbildung 3.1 beschrieben), so ist auch keine extrinsische Analyse möglich. Es ist kein Domänenname und somit kein extrinsisches Merkmal gegeben. Dies führt zu den folgenden Konsequenzen:

1. Der manuellen Wartung und Evolution der Referenzarchitektur wird ein großer Stellenwert zugeordnet, obwohl die weiteren Entwicklungsschritte innerhalb eines Projektes nicht durch weitere Vorgaben durch die Software-Plattform beschränkt sind. Die entsprechenden eher geringen Aufwände ermöglichen eine frühzeitige Identifikation ähnlicher paralleler Arbeiten und festigen das Verständnis einer gemeinsamen Architektur.

2. Der zugrundeliegende Suchraum wird deutlich eingeschränkt. Da die nachfolgenden Ähnlichkeitsanalysen aufwendig sind, ist es sehr hilfreich durch einen ersten sehr starken Filter die Anzahl der falsch positiven Ergebnisse deutlich zu reduzieren. Gleichzeitig wird auf Grund der manuellen Wartung der Referenzarchitektur davon ausgegangen, dass falsch negative Ergebnisse auf Dauer auszuschließen sind.

In der extrinsischen Analyse werden somit Software-Komponentenvarianten einzelner Software-Produkte über das extrinsische Merkmal analysiert. Sollten ähnliche Software-Komponentenvarianten oder Funktionsvarianten parallel oder zeitlich voneinander getrennt entwickelt werden, so würde dies über eine extrinsische Analyse sichtbar. Zwei Software-Komponentenvarianten würden in einem solchen Fall den gleichen Domänennamen besitzen.

In Konsequenz erlaubt die angewandte extrinsische Analyse einen frühzeitigen Überblick auf die Softwareentwicklung innerhalb des gesamten Unternehmens. Mit Hilfe einer etablierten Referenzarchitektur kann somit nicht nur identifiziert werden, in welchen Zeiträumen, welche Funktionalitäten umgesetzt werden, sondern auch welche Tätigkeiten wiederholt stattfinden.

5.1.1 Historische Ähnlichkeit

Auf Basis einer extrinsischen Ähnlichkeit ist es neben einer folgenden strukturellen und semantischen Ähnlichkeitsanalyse auch möglich, historische Zusammenhänge extrinsischer Paare auszuwerten. Dies ist gerade dann sehr einfach möglich, wenn vollzogene Kopiervorgänge im Sinne eines *Clone&Own* durch das genutzte Versionsverwaltungssystem unterstützt wurden. Wird zum Beispiel *Subversion* (SVN) als Versionsverwaltungssystem genutzt und gegebene Artefakte durch Verwendung des Befehls *svn copy* beziehungsweise direkt durch Anlegen eines *Branches* (was indirekt zur Verwendung von *svn copy führt*) kopiert, so kann der vollzogene Kopiervorgang nachträglich zurückverfolgt werden. *SVN* stellt zum Beispiel die Quelle und Ursprungsrevision eines Kopiervorgangs in der Versionshistorie zur Verfügung. Insofern ist es so leicht möglich, den genauen Zeitpunkt sowie die ursprüngliche Quelle des Kopiervorgangs zu identifizieren. In einem zweiten Schritt kann durch ein Versionsverwaltungssystem die Anzahl der Revisionen vor und nach dem Kopiervorgang identifiziert werden, um eine historische Ähnlichkeit eines extrinsischen Paares abzuleiten. Dabei wird die Anzahl der gemeinsamen Revisionen (Revisionen vor dem Kopiervorgang) mit der Anzahl der separaten Revisionen (Revisionen nach dem Kopiervorgang) verglichen, wie in Abbildung 5.2 dargestellt.

Seien $R_{A,B}$ die Revisionen vor dem Kopiervorgang und seien R_A und R_B die Revisionen nach dem Kopiervorgang, so ergibt sich die historische Ähnlichkeit durch
$Sim_H = R_{A,B}/(R_A + R_B + R_{A,B})$.

Dieses sehr einfache Verfahren ermöglicht die direkte Identifikation der Quelle und eine erste grobe Einschätzung des Unterscheidungsgrades zwischen Variante *A* und Variante *B*. Dabei ist diese Analyse sehr grob, da jede Revision bezüglich des Maßes an vollzogenen Änderungen gegenüber der Vorgängerrevision gleich gewertet wird. Gibt es innerhalb des Entwicklungsprozesses konkrete Regeln für den Zeitpunkt eines *Commit*s (Anlegen einer neuen Revision), wie zum Beispiel die Umsetzung einer Anforderung, so würde dies die Genauigkeit der Analyse erhöhen. Die Identifikation der Quelle des Kopiervorgangs sowie eine erste grobe Abschätzung des Grades der anschließenden Abweichung sind nichtsdestotrotz neben weiterführenden Analysen hilfreiche Indikatoren.

Im Folgenden wird nun im Detail auf die Ergebnisse der extrinsischen Analyse eingegangen.

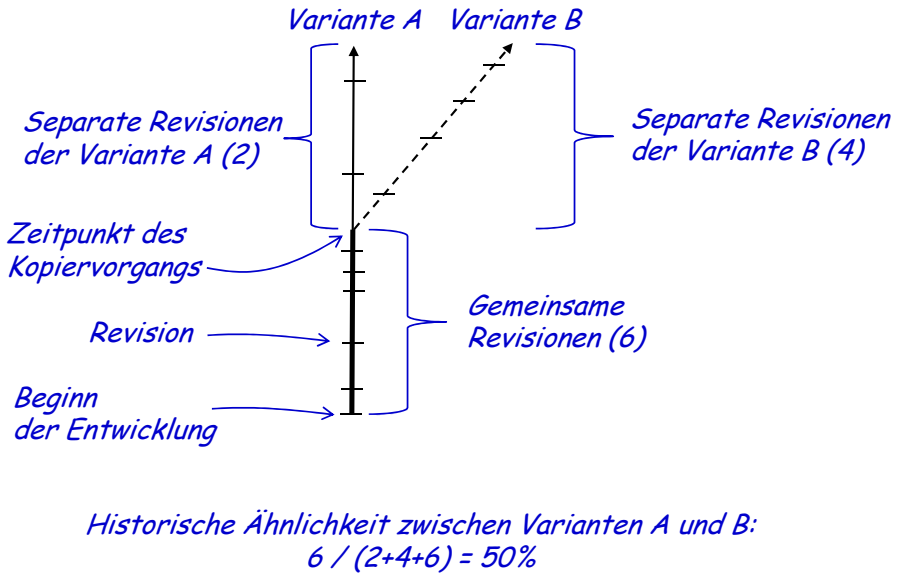


Abbildung 5.2: Historische Ähnlichkeit.

5.1.2 Evaluierung

Die Ergebnisse einer Anwendung der extrinsischen Analyse auf alle *PERSIST*-kompatiblen Projekte ist in Abbildung 5.3 dargestellt. Die Referenzarchitektur besteht dabei aktuell aus 219 Software-Komponentenhüllen, von denen der Hauptteil (125 (57%) Software-Komponentenhüllen) kein extrinisches Paar von Software-Komponentenvarianten aufweist. 94 (43%) Software-Komponentenhüllen weisen dagegen mindestens zwei extrinsisch gleiche Anwendungsartefakte auf, während 61 (28%) Software-Komponentenhüllen sogar mindestens drei extrinsisch gleiche Anwendungsartefakte aufweisen.

Diese 61 Software-Komponentenhüllen können auf Basis der Faustregel, dass sich die Entwicklung eines Software-Komponentenmusters erst bei zweifacher Wiederverwendung lohnt (siehe Kapitel 1), als potentielle Kandidaten gewertet werden, für die sich eine detaillierte Betrachtung bzgl. schnittstellen-basierter und semantischer Analyse lohnt. Des Weiteren sind in Abbildung 5.3 die Anzahl extrinsisch gleicher Anwendungsartefakte im Balkendiagramm genau aufgeschlüsselt.

Auch ist es möglich, eine direkte Zuweisung der extrinisch gleichen Software-Komponentenvarianten zu den jeweiligen Projekten zu extrahieren. Die gegebene Analyse wurde auch manuell überprüft, um fehlerhafte Zuweisungen zu Software-Komponentenhüllen zu identifizieren. Dabei konnten keine falschen Zuweisungen identifiziert werden, da diese durch die Wartung der Referenzarchitektur (Schritte 4 und 10 aus Abbildung 3.1 auf Seite 29) schon ausgeschlossen werden konnten. Auf Basis der etablierten Projektzuweisungen ließ sich deutlich die durch die Experten bekannte Entwicklungshistorie abbilden, so dass bei vielen extrinsischen Paaren ein historischer Zusammenhang über Projektgrenzen hinweg bekannt ist. Während dieses Wissen aber über die Zeit oder durch Personalwechsel verloren gehen würde, kann es durch eine extrin-

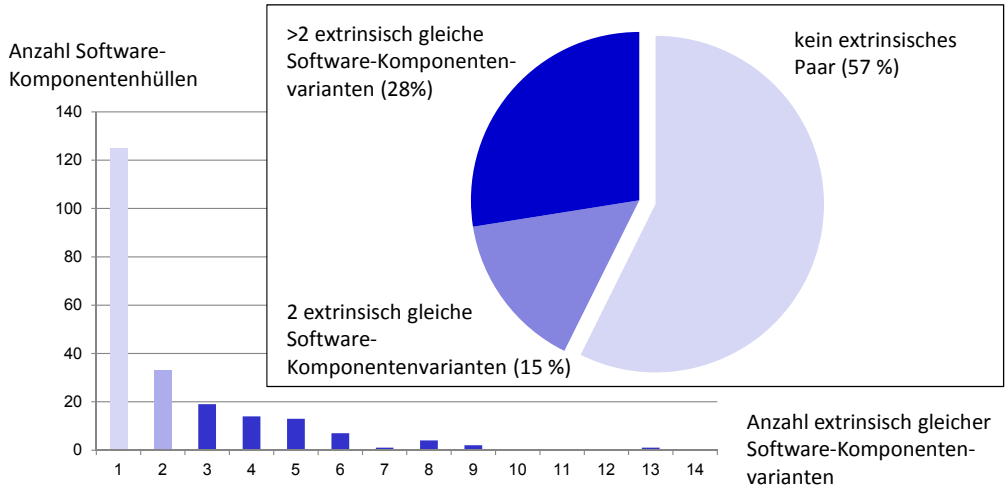


Abbildung 5.3: Ergebnisse der extrinsischen Evaluierung.

sische Analyse und durch konsequente Wartung der Referenzarchitektur über die Zeit gesichert werden.

Die gegebene Übersicht erlaubt einen schnellen ersten Gesamtüberblick über die direkt gegebenen Wiederverwendungspotentiale auf Basis der gegebenen Historie. Sie macht deutlich, dass nach aktuellem Entwicklungsstand ein Wiederverwendungspotential schon auf Basis der extrinsischen Analyse nur für ungefähr ein Drittel der Software-Komponentenhüllen gegeben ist und bestätigt damit einen eher pragmatischen Umgang mit der Entwicklung von Software-Komponentenvorlagen oder -mustern.

5.2 Schnittstellen-basierte Ähnlichkeitsanalyse

Wie schon in Kapitel 3 aufgeführt, ist die extrinsische Analyse nur ein erster Schritt zur Identifikation von potentiellen Software-Komponenten- und Funktionsvarianten, die entweder in eine Software-Plattform überführt oder in der AE direkt als Ausgangspunkt einer Variantenentwicklung genutzt werden können. Die Vorauswahl über das extrinsische Merkmal erlaubt es einen Gesamtüberblick über die gegebene Software-Projektlandschaft und deren Wiederverwendungspotentiale zu erlangen. Des Weiteren ermöglicht sie es aber auch, mit geringem Aufwand den Suchraum für komplexere Ähnlichkeitsanalysen zu reduzieren. Diese sind nötig, um eine genauere Aussage bezüglich des Wiederverwendungspotentials zu erlangen.

Auf Basis eines durch die extrinsische Analyse reduzierten Suchraumes ist es nun in einem zweiten Schritt notwendig, die strukturelle Ähnlichkeit auf Basis von Schnittstellen zwischen zwei oder mehr Software-Komponentenvarianten oder Funktionsvarianten zu identifizieren.

Im Folgenden wird zuerst der Begriff der Schnittstelle abstrakt als Metamodell definiert, bevor dieser dann für die Automotive-Domäne konkretisiert wird.

Domänenunabhängig dient eine Schnittstelle zum Austausch von Daten zwischen unterschied-

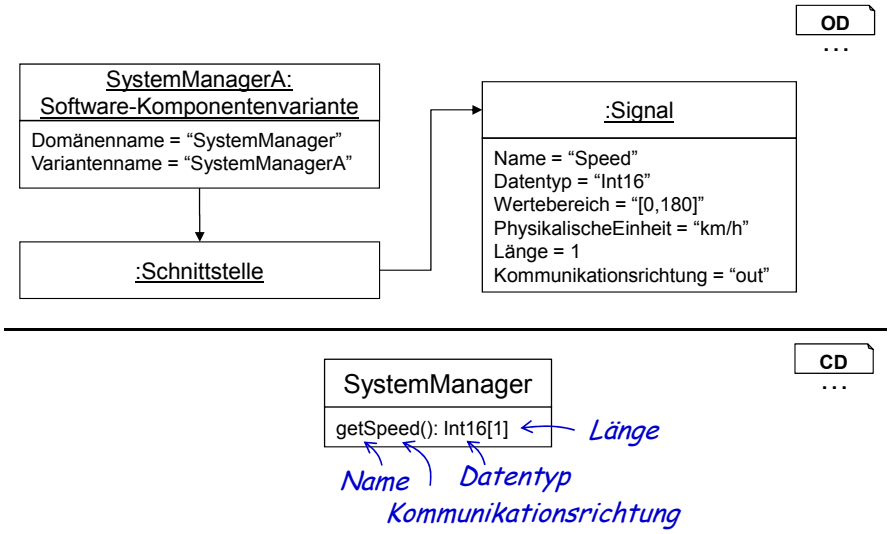


Abbildung 5.4: Beispiele eines Datenelementes einer PERSIST-konformen Schnittstelle (Signal) und einer Methodensignatur (Parameter).

lichen Komponenten. Standardisierte Schnittstellen erleichtern dabei den Austausch von Komponenten. Durch die reaktive Produktlinienentwicklung können schrittweise für jedes extrinsische Merkmal standardisierte Schnittstellen entwickelt werden. Dabei unterstützt die schnittstellen-basierte Ähnlichkeitsanalyse die Identifikation von gleichen, ähnlichen und unterschiedlichen Datenelementen von Schnittstellenvarianten.

Jedes Datenelement einer Schnittstelle ist dabei durch unterschiedliche Attribute beschrieben, die die mögliche Form der auszutauschenden Daten eingrenzen. Aufgabe der schnittstellen-basierten Ähnlichkeitsanalyse ist es Schnittstellenvarianten zu analysieren und Paare identischer oder ähnlicher Datenelemente auf Basis ihrer Attribute zu identifizeren. Mit Hilfe dieser Paare kann in einem Folgeschritt eine standardisierte Schnittstelle definiert werden, zu der die analysierten Schnittstellenvarianten mit möglichst geringem Anpassungsaufwand identisch sind.

Im Folgenden wird das in Abbildung 5.4 aufgeführte Beispiel verwendet, um die folgenden Definitionen zu motivieren. In diesem Beispiel ist sowohl ein Signal einer PERSIST-konformen Software-Komponente, als auch eine Methodendeklaration in einem Klassendiagramm dargestellt.

Ein Signal der Schnittstelle der Software-Komponentenvariante *SystemManagerA* ist durch die Attribute *Name*, *Datentyp*, *logischer Wertebereich*, *physikalische Einheit*, *Länge* und *Kommunikationsrichtung* beschrieben. Die Attributswerte für die jeweiligen Attribute sind *Speed* (für den Attibutstyp *Name*), *Int16* (für das Attribut *Datentyp*), *[0,180]* (für das Attribut *Wertebereich*), *km/h* (für das Attribut *physikalische Einheit*), *1* (für das Attribut *Länge*) und *out* (für die *Kommunikationsrichtung*).

Während für das Signal *Speed* in der Automotive-Domäne übliche Attribut bezüglich *Datentyp*, *logischer Wertebereich*, *physikalische Einheit*, *Länge* und *Kommunikationsrichtung* definiert sind, ist in gängigen Hochsprachen der Rückgabeparameter für die Methode *getSpeed()*

nur durch *Datentyp*, *Kommunikationsrichtung* und *Länge* definiert.

Es ist somit deutlich, dass abhängig von der Domäne unterschiedliche Attribute für jedes auszutauschende Datenelement einer Schnittstelle gegeben seien können. Die folgenden Definitionen beachten diesen Umstand und ermöglichen somit eine domänenunabhängige schnittstellenbasierte Ähnlichkeitsanalyse.

Definition 5.3 *Schnittstelle. Eine Schnittstelle beschreibt eine Menge von Datenelementen, die über diese ausgetauscht werden können.*

Definition 5.4 *Schnittstellenvariante. Eine Schnittstelle einer Software-Komponenten- oder Funktionsvariante.*

Datenelemente werden über eine Schnittstelle ausgetauscht. Die mögliche Form dieses Datenelementes ist dabei durch eine Menge von Attribute beschrieben.

Definition 5.5 *Attribut. Jedes Attribut definiert mittels Attributstyp und Attributswert einen Teilaspekt der möglichen Form eines Datenelementes der Schnittstelle zur Laufzeit.*

Definition 5.6 *Attributstyp. Der Attributstyp legt die möglichen Attributswerte eines Attributs fest. Nur Attributswerte desselben Attributstypen können bezüglich einer Ähnlichkeitsanalyse miteinander verglichen werden.*

Definition 5.7 *Attributswert. Konkreter Wert eines Attributs für ein Datenelement. Mögliche Attributswerte sind dabei durch den Attributstypen definiert.*

Der Zusammenhang zwischen *Schnittstelle*, *Datenelement* und dessen *Attributen* ist nochmals in Abbildung 5.5 dargestellt.

Auf der Beschreibungsebene (*M2*) ist der Zusammenhang zwischen *Schnittstelle* und *Datenelement* sowie die verfeinerten Datenelemente *SignalTyp* und *ParameterTyp* mit ihren Attributen und Attributstypen definiert. Auf der Ebene *M1* werden diese instanziiert, um konkrete PERSIST-Schnittstellen oder Methoden zu definieren. Ein Datenelement einer Schnittstelle einer PERSIST-konformen Software-Komponente ist zum Beispiel ein Signal, während ein Datenelement einer Methodensignatur zum Beispiel ein Parameter ist. Die entsprechenden Attributswerte sind aus dem Eingangsbeispiel aus Abbildung 5.4 übernommen.

Zur Laufzeit werden über instanziierte Komponenten Datenelemente mit konkreten Werten verschickt, wie in Abbildung 5.6 dargestellt. Die Instanz der Software-Komponentenvariante *SystemManagerA* gibt das Signal *Speed* mit dem konkreten Wert fünf zurück. Auf Grund des verwendeten Signaltyps *Speed* (siehe Abbildung 5.5) ist des Weiteren festgelegt, dass es sich um ein ausgehendes Signal der Länge 1 handelt, dessen Werte zwischen 0 und 180 liegen. Außerdem repräsentiert der konkrete Wert 5 Kilometer pro Stunde.

Auf Basis dieses Zusammenhanges ist es nun möglich eine domänenunabhängige Schnittstellenbasierte Ähnlichkeitsanalyse zu definieren.

Definition 5.8 *Schnittstellen-basierten Ähnlichkeitsanalyse. Die schnittstellen-basierte Ähnlichkeitsanalyse identifiziert Ähnlichkeiten zwischen Schnittstellenvarianten durch Bildung von Mengen ähnlicher Datenelemente. Ähnliche Datenelemente werden dabei durch Auswertung der einzelnen Attribute identifiziert. In einem zweiten Schritt wird durch die Wahl eines Repräsentanten*

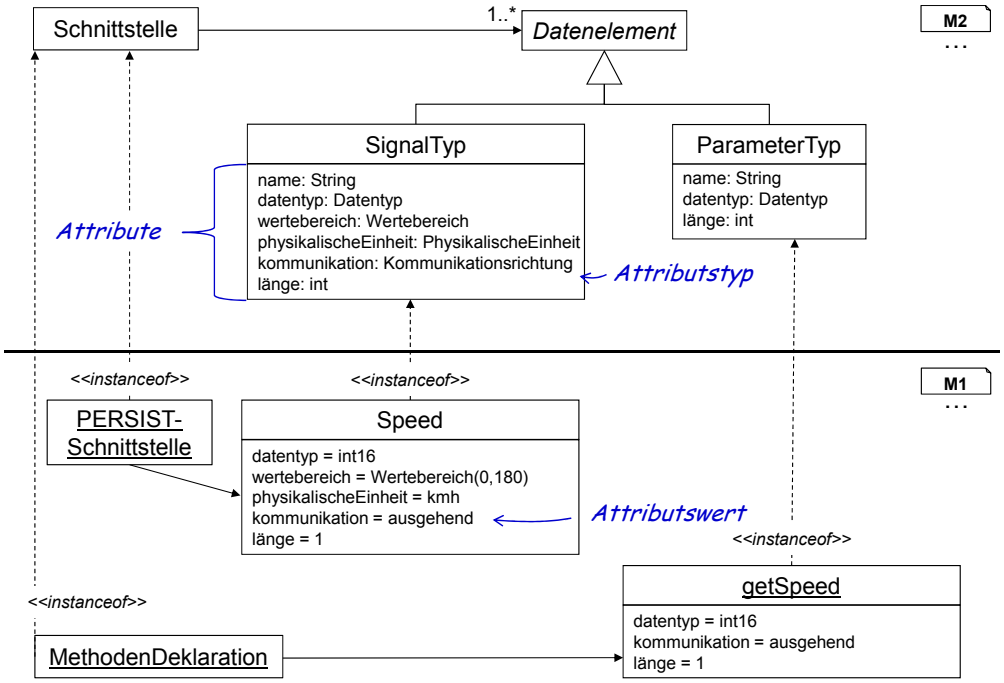


Abbildung 5.5: Abstrakte Definition einer Schnittstelle und verfeinerte Definitionen von Datenelementen zur Instanziierung von PERSIST-Schnittstellen und Methodendeklarationen.

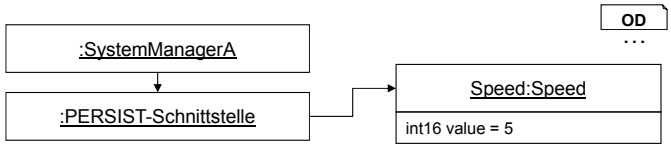


Abbildung 5.6: Instanziierte Software-Komponentevariante *SystemManagerA* mit konkretem Datenwert für das Signal *Speed*.

für jede Menge ähnlicher Datenelemente eine Schnittstelle für ein Domänenartefakt definiert. Die Wahl des Repräsentanten wird dabei durch eine Maximierung der Ähnlichkeit zu allen anderen ähnlichen Datenelementen bestimmt.

Definition 5.9 *Menge ähnlicher Datenelemente. Eine Menge ähnlicher Datenelemente enthält ähnliche Datenelemente verschiedener Schnittstellenvarianten. Dabei enthält eine Menge ähnlicher Datenelemente jeweils nur ein Datenelement jeder Schnittstellenvariante.*

Definition 5.10 *Ähnliche Datenelemente. Zwei Datenelemente unterschiedlicher Schnittstellenvarianten sind ähnlich, wenn sie Attribute desselben Typs definieren und deren Werte ähnlich sind.*

Definition 5.11 *Ersetzungskompatibilitätsrelation. Sei A die Menge der Attributsstypen und V_a die Menge der möglichen Werte des Attributstyps $a \in A$. Die Ersetzungskompatibilitätsrelation $C_a \subseteq (V_a \times V_a)$ beinhaltet alle Paare von möglichen Attributswerten, wo der erste Attributswert durch den zweiten Attributswert ersetzt werden kann.*

Definition 5.12 *Ähnliche Attributswerte. Seien x, y zwei Attributswerte des Attributstypen a, so ist x ähnlich zu y, wenn gilt: xC_ay.*

Jeder Attributstyp beschreibt dabei eine eigene Menge an möglichen Attributswerten. Die Ähnlichkeit zwischen diesen Attributswerten ist dabei für jeden Attributstyp unterschiedlich. Die genaue Definition von Ähnlichkeit einzelner Attributswerte muss somit für jeden Attributstypen separat festgelegt werden. Für den Attributstypen *physikalischeEinheit* sind zum Beispiel die Attributswerte *km/h* und *m/s* ähnlich, während *km/h* und *kg* nicht ähnlich sind. Die physikalische Einheit *m/s* kann in *km/h* überführt werden und beide Einheiten definieren eine Geschwindigkeit, während durch die physikalische Einheit *kg* ein Gewicht definiert wird. Auch definiert die Ersetzungskompatibilitätsrelation nur eine generelle Ähnlichkeit, ohne einen expliziten Ähnlichkeitswert (zwischen 0% und 100%) festzulegen.

Definition 5.13 *Ähnlichkeitsfunktion.*
Die Ähnlichkeitsfunktion $sim_c : (V_c \times V_c) \rightarrow \{x | x \geq 0, x \leq 1, x \in \mathbb{R}\}$ definiert für zwei Attributswerte eines Attributstyps einen Ähnlichkeitswert zwischen 0% und 100%.
Dabei gilt: $\forall x, y \in V_c : (x, y) \in C_p \leftrightarrow sim_p(x, y) > 0$.

Für jeden Attributstyp muss eine Ersetzungskompatibilitätsrelation und eine Ähnlichkeitsfunktion definiert sein, um eine schnittstellenbasierte Ähnlichkeitsanalyse durchführen zu können. Beide basieren dabei auf der Analyse der syntaktischen Struktur der Schnittstellen. Die physikalischen Einheiten *m/s* und km/h sind sich zum Beispiel weniger ähnlich als die physikalischen Einheiten *m/s* und *cm/s*.

Im Folgenden wird die schnittstellen-basierten Ähnlichkeitsanalyse und verwendete Attributstypen für die Automotive-Domäne sowie zugehörige Ersetzungskompatibilitätsrelationen und Ähnlichkeitsfunktionen vorgestellt. Dieses Verfahren ist grundsätzlich von der Domäne losgelöst einsetzbar. Die nachfolgende Evaluierung wird allerdings zeigen, dass das Verfahren gerade in der Automobilindustrie auf Grund der statischen Architektur und der größeren Anzahl an unterschiedlichen Attributstypen sehr gute Ergebnisse liefert. Das Verfahren wurde 2016 bei der SPLC vorgestellt [KRR+16] und wird im Folgenden nochmal detaillierter beschrieben.

5.2.1 Konzept

Die Schnittstellenähnlichkeitsanalyse wird im Kontext von PERSIST auf Basis von *SenderReceiver*-Schnittstellen (siehe 2.2 auf Seite 17) durchgeführt. Deswegen wird das Verfahren im Folgenden zuerst abstrakt auf Basis der eingeführten Definitionen und losgelöst von der Automobilindustrie definiert. Erst im zweiten Teil dieses Unterkapitels wird das Verfahren exemplarisch für den Automobilkontext verfeinert und evaluiert. Begleitend wird das Verfahren mittels eines fiktiven Beispiels aus der Automobilindustrie erläutert.

Die schnittstellen-basierte Ähnlichkeitsanalyse kann in fünf Teilschritte untergliedert werden:

1. **Import der Schnittstellen:** Zuerst werden die Schnittstellen in ein standardisiertes Format überführt. Dies erleichtert die Unterstützung weiterer, zukünftig integrierbarer Schnittstellenformate.

2. **Element-Ähnlichkeitsanalyse:** In einem zweiten Schritt werden alle Datenelemente einer Schnittstellenvariante mit allen Datenelementen der anderen Schnittstellenvarianten auf Basis ihrer Attribute verglichen, um einen Ähnlichkeitswert zu berechnen. Auf Basis der berechneten Ähnlichkeiten wird ein Graph erstellt, der die Datenelemente als Knoten und die Ähnlichkeitswerte als gewichtete Kanten enthält.

3. **Bildung der Mengen ähnlicher Datenelemente:** Mit Hilfe des konstruierten Graphen wird ein gewichtetes Matching [Bol04] berechnet, das die bestmögliche Zuordnung auf Basis der berechneten Ähnlichkeitswerte darstellt.

4. **Identifikation des Repräsentanten für jede Menge ähnlicher Datenelemente:** In einem weiteren Schritt wird der Repräsentant für jede Menge ähnlicher Datenelemente berechnet. Dafür wird das Datenelement identifiziert, das bezüglich der anderen Datenelemente der Menge ähnlicher Datelemente die größte Ähnlichkeit aufweist.

5. **Evaluierung der Resultate:** Die in den Schritten zwei bis vier gesammelten Erkenntnisse werden zur Analyse aufbereitet und in unterschiedlichen Sichten dargestellt, um die Extraktion der gewünschten Informationen zu erleichtern.

Im Folgenden werden die genannten Schritte formal beschrieben und anhand eines Beispiels dargestellt, das in Abbildung 5.7 aufgeführt ist.

In diesem Beispiel berechnen unterschiedliche Varianten den aktuellen Systemzustand basierend auf dem vorherigen Systemzustand (Signale *Init*, *StartUp* und *SystemState*), der Geschwindigkeit (Signale *Speed* und *Velocity* zusammen mit entsprechenden Qualifizierern *StatusSpeed* und *VelocityQualifier*) und weiteren Datenelementen, wie Beschleunigung, Temperatur oder Batterie-Spannung. Im Beispiel sind jeweils direkt die Attributswerte der jeweiligen Signale der Schnittstellen aufgeführt. Die Kanten zwischen den einzelnen Knoten in Abbildung 5.7 verbinden dabei die manuell identifizierten ähnlichsten Paare zwischen den Signalen der einzelnen Schnittstellen und die Knoten jedes Teilgraphen bilden eine Menge ähnlicher Datenelemente. Die im Folgenden vorgestellte Ähnlichkeitsanalyse sollte möglichst in der Lage sein, diese Mengen zu identifizieren. Dabei sind folgende Problemstellungen beschrieben:

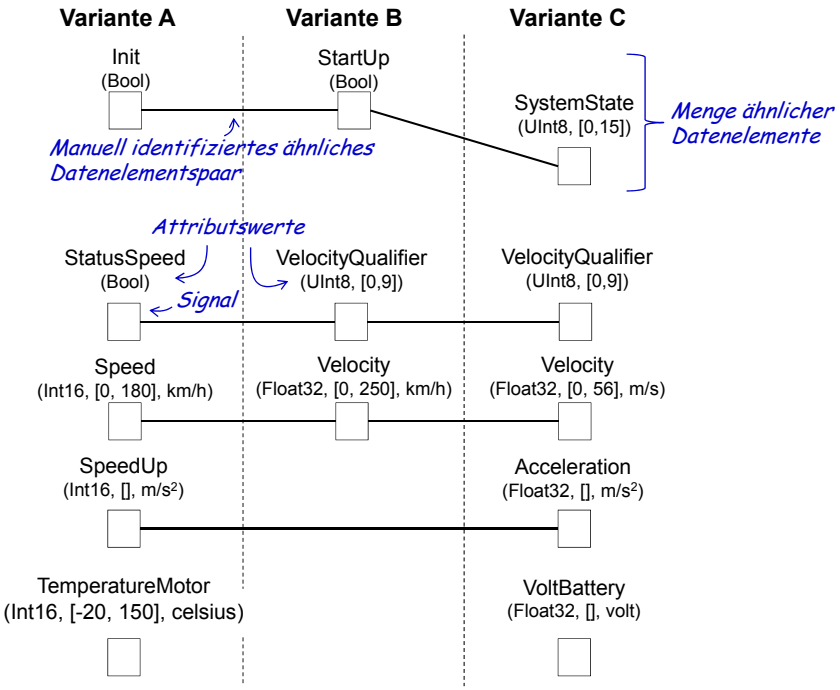


Abbildung 5.7: Beispiel vereinfachter Schnittstellenvarianten und zugehörige manuell identifizierte Mengen ähnlicher Datenelemente.

- **Unterschiedliche Namen:** Die meisten Signale, zwischen denen eine Ähnlichkeit suggeriert wird, sind unterschiedlich benannt, wobei die Benennung so ausgefallen ist, dass noch nicht einmal eine teilweise Übereinstimmung existiert (*Init* / *StartUp* / *SystemState* oder *Speed* / *Velocity*).

- **Unterschiedliche Datentypen:** In *VarianteA* und *VarianteB* ist der Systemzustand nur durch einen Booleschen Wert (*Init* / *StartUp*) dargestellt, während *VarianteC* in der Lage ist, mittels Aufzählung 16 verschiedene Zustände darzustellen. Dieselbe Problematik ist bezüglich des *Velocity*-Qualifizierers gegeben. Die Signale *Speed* / *Velocity* und *SpeedUp* / *Acceleration* sind entweder durch Festpunkt- oder Fließpunkt-Arithmetik zugehörige Datentypen gekennzeichnet, was technischen Rahmenbedingungen geschuldet ist.

- **Unterschiedliche physikalische Einheiten:** Das Signal *Velocity* der Schnittstellenvarianten *VarianteA* und *VarianteB* ist mit unterschiedlicher Genauigkeit definiert (m/s und km/h). An anderer Stelle (*TemperaturMotor* / *VoltBattery*) erlaubt die Angabe verschiedener phyiskalischer Einheiten die Vermeidung falsch positiver Ergebnisse (Zuordnung von Datenelementen, welche nicht ähnlich sind).

- **Unterschiedliche logische Wertebereiche:** Zusätzlich zu den durch den Datentyp definierten technischen Wertebereich kann ein zusätzlicher logischer Wertebereich das Signal weiter einschränken. Diese Information ist abhängig vom gewählten Quotienten der phy-

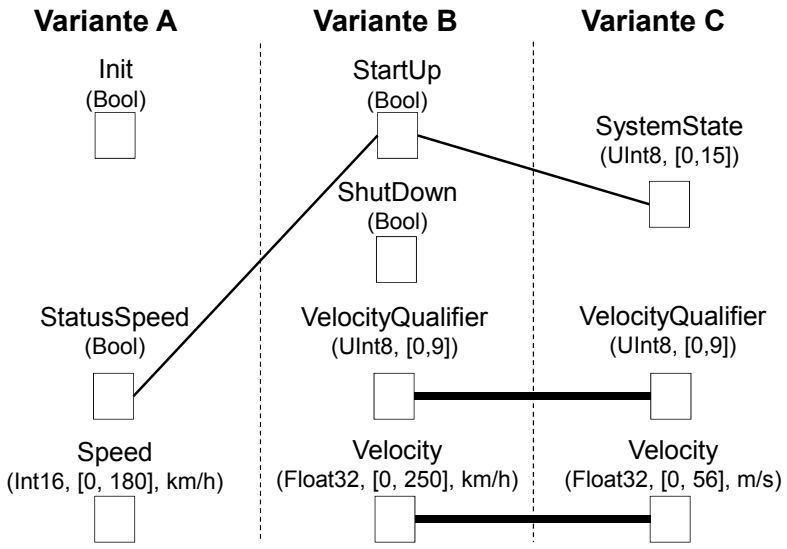


Abbildung 5.8: Namensbasierte Ähnlichkeit einzelner Signale unterschiedlicher Schnittstellenvarianten.

sikalischen Einheit, wie zum Beispiel für das Datenelement *Velocity* dargestellt.

Das Beispiel veranschaulicht schon die im Kontext der Automobilindustrie verwendeten Attributstypen: Name, Datentyp, physikalische Einheit und logischer Wertebereich. Zusätzlich dazu sind auch die Länge eines Elementes relevant, um zwischen atomaren Elementen und Arrays zu unterscheiden. Im Vergleich zu Parametern, die generell in der Softwareentwicklung Verwendung finden, stellt die Automobilindustrie somit bezüglich ihrer Signale zwei zusätzliche Attributstypen (physikalische Einheit und logischer Wertebereich) zur Verfügung, auf deren Basis eine schnittstellen-basierte Ähnlichkeitsanalyse durchgeführt werden kann.

Ein Versuch, eine korrekte Zuweisung nur auf Basis eines einzelnen Attributs herzuleiten, ist in vielen Beispielen nicht möglich, wie auch in Abbildung 5.8 und Abbildung 5.9 dargestellt. Hierbei werden jeweils Ähnlichkeiten zwischen einzelnen Signalen für das Attribut *Name* (Abbildung 5.8) und das Attribut *Datentyp* (Abbildung 5.9) identifiziert. Umso stärker die Kante ausgeprägt ist, umso deutlicher ist die Ähnlichkeit zweier Signale auf Basis des selektierten Attributstyps. Keine Kante stellt wiederum keine Ähnlichkeit dar.

In beiden Fällen ist es nicht möglich, auf Basis eines Attributs den gewünschten Zusammenhang aus Abbildung 5.7 abzuleiten. Stattdessen ist es notwendig, auf alle zur Verfügung stehenden Attribute zurückzugreifen.

Abhängig von der Domäne stellt ein Datenelement der Schnittstelle unterschiedliche Attributstypen zur Verfügung. Für jedes dieser Attributstypen kann es unterschiedliche Verfahren geben, um eine Ähnlichkeit zweier Attributswerte zu bestimmten.

Grundsätzlich ist also das generelle Verfahren zur Bestimmung von Ähnlichkeiten zwischen zwei Datenelementen, der Berechnung entsprechender Mengen ähnlicher Datenelemente und der Identifikation von Repräsentanten der Mengen ähnlicher Datenelemente von der Berechnung der Ähnlichkeit der Attributswerte domänenspezifischer Attributstypen zu trennen.

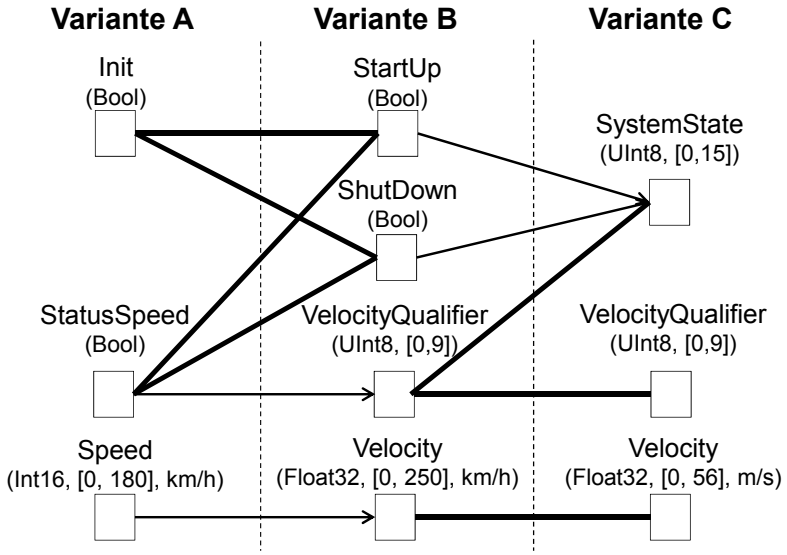


Abbildung 5.9: Typbasierte Ähnlichkeit einzelner Signale unterschiedlicher Schnittstellenvarianten.

Ähnlichkeit einzelner Attributsswerte einer Datenelementes

Im Folgenden werden Ersetzungskompatibilitätsrelationen und Ähnlichkeitsfunktionen für die aufgeführten Attributstypen definiert. Attributswerte können zueinander gleich (100 % Ähnlichkeit), ähnlich (100% > Ähnlichkeit > 0 %) oder unterschiedlich (0 % Ähnlichkeit) sein.

Bezüglich der beschriebenen Attributstypen ist eine Ersetzungskompatibilitätsrelation wie folgt definiert:

- **Name** Für zwei Namen n_1, n_2, ist eine Überführung immer möglich.
  Somit ist $C_{name} = I_{name}$.
- **Länge** Eine Ersetzungskompatibilität zweier verschiedener Längen (Anzahl Elemente) w_1, w_2 ist nie möglich, insofern gilt: $w_1 C_{width} w_2 \leftrightarrow w_1 = w_2$.
- **Wertebereich** Ein Wertebereich $r_1 := [min_1, max_1] = \{x | x >= min_1,$
  $x <= max_1\}$ ist zu $r_2 := [min_2, max_2] = \{x | x >= min_2, x <= max_2\}$ ersetzungskompatibel, wenn der erste Wertebereich eine Teilmenge des zweiten Wertebereichs ist. Insofern gilt $r_1 C_{range} r_2 \leftrightarrow r_1 \subseteq r_2$.
- **Datatyp** Datentyp d_1 ist zu Datentyp d_2 ersetzungskompatibel, wenn eine entsprechende Repräsentation möglich ist. Zum Beispiel kann ein 8 Bit Integer von einem 16 Bit Integer dargestellt werden, andersherum ist dies aber nicht möglich.
- **Physikalische Einheit** Eine Ersetzungskompatibilität zweier physikalischer Einheiten u_1, u_2 ist nach den SI-Einheitenkonvertierungen [TT01] möglich. Zum Beispiel kann Meter pro Sekunde in Kilometer pro Stunde umgerechnet werden (siehe Beispiel in Abbildung 5.7).

- **Kommunikationsrichtung** Eine Ersetzungskompatibilität verschiedener Kommunikationsrichtungen c_1, c_2 ist nie gegeben, insofern gilt: $c_1 C_{communication} c_2 \leftrightarrow c_1 = c_2$.

Auf Basis der Ersetzungskompatibilitätsrelation lässt sich die Ähnlichkeit $sim_p(p_1, p_2)$ eines Attributswertes p_1 zum Attributswert p_2 bestimmen:

$$sim_p(p_1, p_2) = \begin{cases} 0 & \text{if } \neg(p_1 C_p p_2) \\ 1 & \text{if } p_1 = p_2 \\ x & \text{if } p_1 C_p p_2 \wedge p_1 \neq p_2, \end{cases}$$

wobei $x \in [0, 1]$ durch attributstypspezifische Ähnlichkeitsfunktionen definiert werden, die im Folgenden aufgeführt sind.

- **Name** Die Ähnlichkeit zweier Namen $sim_{name}(n_1, n_2)$ wird durch die Levenstein-Distanz berechnet [Lev66].

- **Länge** Ist $w_1 C_{width} w_2$, so gilt schon, dass $w_1 = w_2$. Insofern ist $sim_{width}(w_1, w_2) := 1$.

- **Wertebereich** Seien $r_1 = [r_{1,min}, r_{1,max}]$ und $r_2 = [r_{2,min}, r_{2,max}]$ zwei Wertebereiche mit $R_1 = r_{1,max} - r_{1,min}$, $R_2 = r_{2,max} - r_{2,min}$. Durch die Ersetzungskompatibilität von r_1 zu r_2 ist schon gewährleistet, dass der Wertebereich von r_1 die Teilmenge des anderen Wertebreiches ist. Die Ähnlichkeit ist dann durch die Kardinalität der Schnittmenge in Relation zur Kardinalität des größeren Wertebereich definiert: $sim_{range}(r_1, r_2) := \frac{max(R_1,R_2)-|R_1-R_2|}{max(R_1,R_2)}$.

- **Datentyp** Sei $bit(x)$ die Anzahl Bits eines Datentyps, so ist die Ähnlichkeit zweier kompatibler Datentypen d_1, d_2 durch das Verhältnis minimaler Bits zu maximalen Bits definiert: $sim_{datatype}(d_1, d_2) := \frac{min(bit(d_1),bit(d_2))}{max(bit(d_1),bit(d_2))}$

- **Physikalische Einheit** Laut [TT01] ist die Quantität Q einer physikalischen Einheit durch die SI Basiseinheiten formulierbar: $[Q] := 10^n \times m^\alpha \times kg^\beta \times s^\gamma \times A^\delta \times K^\epsilon \times mol^\zeta \times cd^\eta$, wobei $-24 \leq n \leq 24$. Unter der Annahme, dass diesselben SI-Basiseinheiten verwendet werden (siehe Ersetzungskompatibilität), können nur noch die Exponenten n_1 und n_2 unterschiedlich sein. Da gilt $-24 \leq n_1, n_2 \leq 24$, lässt sich folgende Berechnung schlussfolgern: $sim_{unit}(n_1, n_2) := \frac{49-(|n_1-n_2|)}{49}$. Umso geringer die Distanz zwischen den Exponenten beider physikalischer Einheiten ist, umso größer ist die Ähnlichkeit.

- **Kommunikationsrichtung** Ist $c_1 C_{width} c_2$, so gilt schon, dass $c_1 = c_2$.
  Insofern ist $sim_{communication}(c_1, c_2) := 1$.

Sei $Props$ die Menge aller zu untersuchenden Attribute und $prop(i, p)$ die Funktion, die für ein Datenelement i und ein Attribut p den konkreten Attributswert zurückgibt. Um auf Basis der einzelnen Ähnlichkeitswerte für jedes Attribut des Datenelementes nun einen Gesamtwert für das Datenelementpaar zu berechnen, wird der Durchschnitt unter Berücksichtigung einer Gewichtung w_p der einzelnen Attribute gebildet:

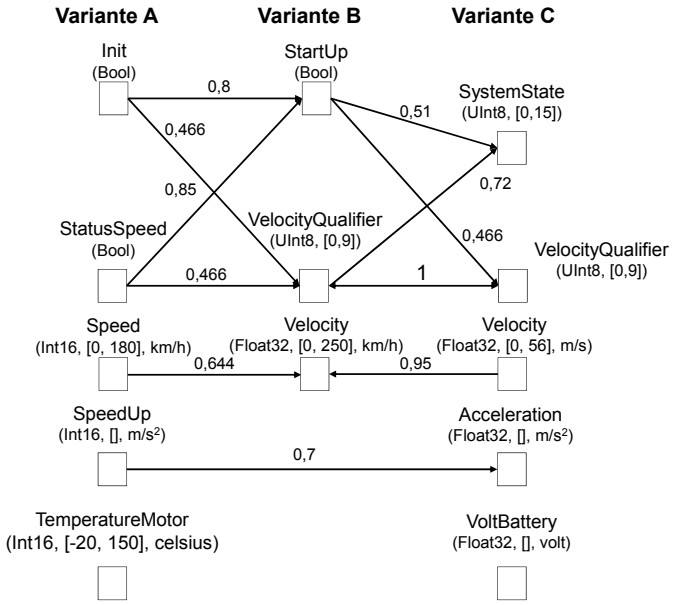


Abbildung 5.10: Berechnete Ähnlichkeitswerte zwischen einzelnen Datenelementen auf Basis der Ersetzungskompatibilitätsrelation und Ähnlichkeitsfunktion.

$$sim(s_1, s_2) = \frac{\sum\limits_{a \in Props} w_p * sim_p(prop(s_1, p), prop(s_2, p))}{\sum\limits_{a \in Props} w_p}$$

g.d.w.

$$\forall p \in Props : C_p(prop(s_1, p), prop(s_2, p)) \neq 0$$

Ansonsten $sim(s_1, s_2) = 0$.

Diese ist wiederum gerichtet. Grundsätzlich kann die Ersetzungskompatibilitätsrelation und auch die Ähnlichkeitsfunktion für jedes Attribut ausgetauscht werden. Des Weiteren kann die genannte Gewichtung der Attribute für unterschiedliche Domänen oder Anwendungsfälle angepasst werden.

Die Anwendung der eingeführten Funktionen auf das in Abbildung 5.7 beschrieben Beispiel ist in Abbildung 5.10 dargestellt.

Berechnung der Mengen ähnlicher Datenelemente

Auf Basis der extrahierten Ähnlichkeitswerte ist es nun notwendig, in einem zweiten Schritt die Mengen ähnlicher Datenelemente zu identifizieren . Dafür werden die berechneten Ähnlichkeitsbeziehungen in einem ersten Schritt in einen Graphen überführt (Datenelemente als Knoten, Ähnlichkeitswerte als gewichtete und gerichtete Kanten), um in einem weiteren Schritt ein maximales Matching abzuleiten [Bol04]. Dies garantiert eine entsprechende Paarbildung mit maxi-

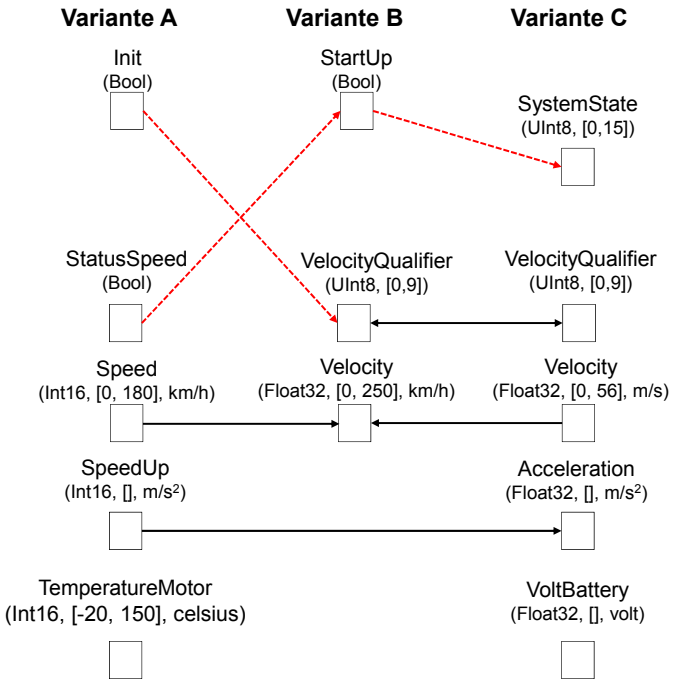


Abbildung 5.11: Berechnete Matches (gefilterte Matches sind rot und gestrichelt markiert).

malem Gewicht und somit einer maximalen Ähnlichkeit. Zur Berechnung der Matches werden jeweils nur Datenelemente zweier Schnittstellen herangezogen und somit Paare von Datenelementen für jede Schnittstellenkombination berechnet. In Abbildung 5.11 sind die entsprechende Paare weiterhin als Graph dargestellt. Alle Kanten, die nicht zum maximalen Matching zwischen *Variante A* und *Variante B* oder zwischen *Variante B* und *Variante C* gehören, wurden entfernt. Wiederum wurden die Beziehungen der Signale der Schnittstellen von *Variante A* und *Variante C* nicht dargestellt, um die Lesbarkeit zu erhöhen. Generell würden diese aber immer berechnet werden.

Vergleicht man das Beispiel in Abbildung 5.11 mit der in Abbildung 5.7 manuell festgelegten Ähnlichkeit, so wird deutlich, dass nur die Zuweisung der Datenelemente *Init*, *StartUp*, *SystemState* und die Zuweisung der Datenelemente *StatusSpeed*, *VelocityQualifier* fehlerhaft sind. Bei genauerer Betrachtung sind bei den aufgeführten Problemfällen immer Wahrheitswerte involviert. Bei Wahrheitswerten ist aber neben dem Datentyp nur das Attribut *Name* für einen weiteren Vergleich relevant. Existiert aber keine Namensähnlichkeit zwischen diesen Signalen, so kann eine Zuweisung falsch sein. Um dies zu verhindern, kann ein entsprechender Schwellenwert für Attribute eingeführt werden (in diesem Fall für das Attribut *Name*), um falsch positive Ergebnisse zu vermeiden. Durch rote, gestrichelte Kanten wird in Abbildung 5.11 anschaulich dargestellt, dass falsche Zuweisungen durch einen entsprechenden Schwellenwert vermieden werden können. Gleichzeitig wird aber auch eine korrekte Zuweisung (in diesem Fall die Kante zwischen *StartUp* und *SystemState*) entfernt. Dies veranschaulicht, dass es generell unmöglich ist, einen Schwellenwert zu finden, der in jedem Fall ein optimales Ergebnis ermöglicht (keine

falsch positiven oder falsch negativen Ergebnisse).

Nach Bildung des maximalen Matches M zweier Schnittstellen S_1 und S_2 mit den Datenelementen E_1 und E_2 wird eine durchschnittliche gerichtete Ähnlichkeit zwischen diesen Schnittstellen wie folgt berechnet:

$$sim_{dir}(S_1, S_2) = \frac{\sum_{(e_1,e_2)\in M} sim(e_1, e_2)}{|E_1|}$$

Die ungerichtete Ähnlichkeit beider Schnittstellen wird wiederum durch den Durchschnitt der gerichteten Ähnlichkeiten definiert:

$$sim(S_1, S_2) = \frac{sim_{dir}(S_1, S_2) + sim_{dir}(S_2, S_1)}{2}$$

In dem gegebenen Beispiel aus Abbildung 5.11 wären folgende Ähnlichkeiten berechnet (ohne Schwellenwert):

$$sim(VarianteA, VarianteB) = \frac{0,392 + 0}{2} = 0,196$$

$$sim(VarianteB, VarianteC) = \frac{0,5 + 0,39}{2} = 0,445$$

Berechnung der Repräsentanten der Mengen der ähnlichen Informationen

Zur Berechnung der Mengen der ähnlichen Datenelemente aller verglichenen Variantenschnittstellen werden, wie schon in Abbildung 5.11 illustriert, die einzelnen maximalen Matches verbunden. So entstehen aus den Paaren einzelne Graphen, deren Knoten jeweils eine Menge ähnlicher Datenelemente bilden.

Für jeden dieser Graphen wird nun in einem nächsten Schritt der Repräsentant der Menge ähnlicher Datenelemente berechnet. Um dies zu ermöglichen, wird mittels Edmonds Algorithmus [Edm65] ein gerichteter maximaler Spannbaum (Arborescence) berechnet.

Definition 5.14 *Arborescence. „Ein Teilgraph $T = (V, F)$ von G ist eine Arborescence bezogen auf die Wurzel r genau dann, wenn T keinen Zyklus beinhaltet, und für jeden Knoten $v \neq r$ existiert in F genau eine Kante, welche v betritt.“ [KT06]*

In einem ersten Schritt werden hierzu die Richtungen der durch die gerichtete Ähnlichkeitsfunktion berechneten Kanten umgedreht (da die Wurzel ein Datenelement repräsentieren soll, das andere Datenelemente ersetzen kann). Der maximale Spannbaum wiederum hat bezüglich der Ähnlichkeit eine maximale Gewichtung. Dieser ist immer existent, aber in Einzelfällen nicht eindeutig. Zum Beispiel können alle Datenelemente einer Menge ähnlicher Datenelemente zueinander jeweils gleich ähnlich sein. Die Wurzel dieses Spannbaumes (nach Korrektur der Kantenrichtung) ist somit der Repräsentant der Menge ähnlicher Datenelemente (auf Grund der maximalen Ähnlichkeit). Bezogen auf das Beispiel aus Abbildung 5.11 sind die identifizierten Arborescencen der Mengen ähnlicher Datenelemente in Abbildung 5.12 dargestellt. Der Repräsentant ist jeweils farblich hervorgehoben.

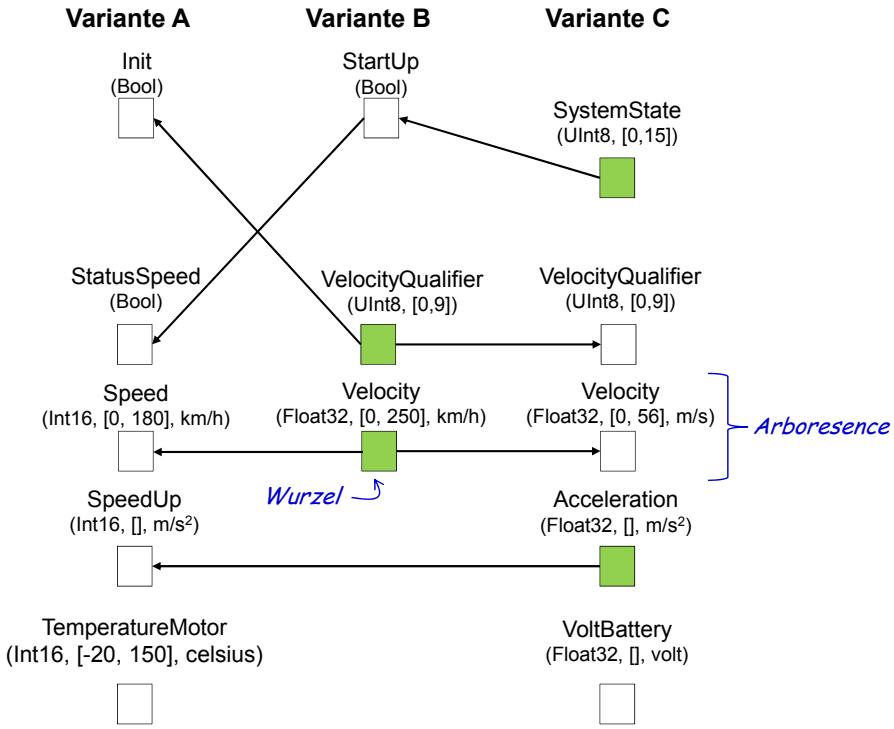


Abbildung 5.12: Identifizierte Arborescencen und ihre Wurzeln (farblich hervorgehoben, Repräsentanten der Menge ähnlicher Datenelemente).

Bei der Bildung der Menge ähnlicher Datenelemente können auch falsch positive Ergebnisse, die während des Matchings entstehen, identifziert werden. In Abbildung 5.13 ist eine entspre-

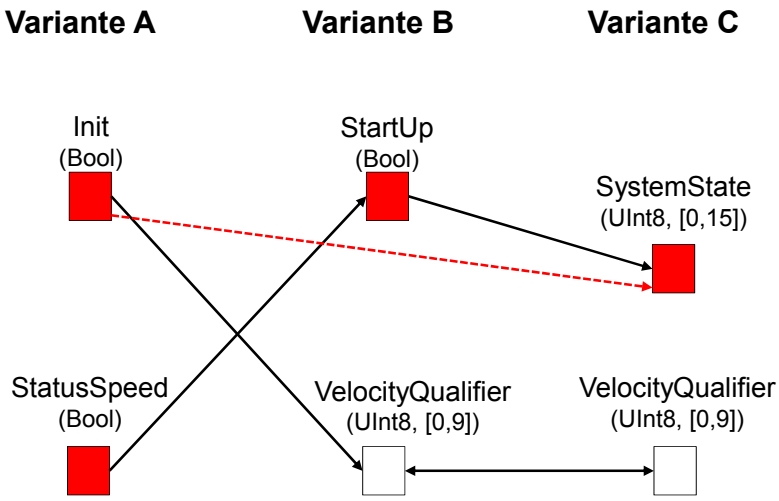


Abbildung 5.13: Ungültige Menge ähnlicher Datenelemente.

chendes Beispiel dargestellt. Unter Berücksichtigung der Ähnlichkeiten der Schnittstellenvarianten *VarianteA* und *VarianteC* (hier rot gestrichelt dargestellt) wird eine ungültige Menge ähnlicher Datenelemente berechnet, da diese zwei Datenelemente derselben Schnittstellenvariante (*Init* und *StatusSpeed*) enthält.

5.2.2 Evaluierung

Basierend auf den innerhalb der Evaluierung der extrinsischen Analyse identifizierten extrinsischen Paaren wurde eine weiterführende Evaluierung der schnittstellen-basierten Ähnlichkeitsanalyse durchgeführt. Dabei legt die Evaluierung einen primären Fokus auf die Genauigkeit des Verfahrens. Dabei war es wichtig, nicht nur die generelle Genauigkeit des Verfahrens (korrekte Paare im Verhältnis zu allen identifizierten Paaren), sondern auch die Beeinflussung unterschiedlicher Faktoren auf die Genauigkeit das Verfahren zu bestimmen. Ausgehend von den Beobachtungen, die schon durch das initiale Beispiel aus Abbildung 5.7 beschrieben worden sind, ist man von zwei Hypothesen ausgegangen.

- **H1** Die Genauigkeit ist abhängig von der Menge an Attributen, die zur Auswertung zur Verfügung stehen.
- **H2** Die Genauigkeit ist abhängig von einem gesetzten Schwellenwert für das Attribut *Name*. Dieser Schwellenwert kann nicht nur die Anzahl der falschen Ergebnisse reduzieren, sondern beeinflusst auch maßgeblich das Verhältnis zwischen falsch positiven und falsch negativen Ergebnissen.

Um die Genauigkeit des Verfahrens in Hinblick auf einen modifizierbaren Schwellenwert und die zur Verfügung stehenden Qualität der Signaldefinition (Anzahl definierte Attribute in Relation zur maximal möglichen Anzahl an definierbaren Attributen) zu bewerten, ist es notwendig, die korrekte Identifkation von ähnlichen Signalen parallel manuell durch einen Experten durchführen zu lassen. Da eine entsprechende manuelle Analyse für größere Modelle sehr zeitaufwendig ist, wurde die Anzahl der zu vergleichenden Einheiten reduziert, so dass von den vorselektierten Software-Komponenten sechs Software-Komponenten mit insgesamt 15 Varianten ausgewählt wurden. Dabei wurden insgesamt durch eine manuelle Analyse 721 Signalpaarungen identifiziert (inklusive der Paarungen von Signalen ohne Partner). Die Ergebnisse der automatischen Analyse wurden mit den Ergebnissen der manuellen Analyse verglichen, um die Genauigkeit des Verfahrens zu bewerten und den Anteil an falsch positiven und falsch negativen Ergebnissen berechnen zu können. Dabei wurden die Ergebnisse der manuellen Analyse als 100% korrekt interpretiert.

Bezüglich der Hypothese **H1** sind in Abbildung 5.14 die Genauigkeit des Verfahrens in Relation zur Qualität der Signaldefinition aufgezeigt. Deutlich ist ein Anstieg der Genauigkeit ab einer Qualität der Signaldefinition von 60 % zu erkennen. Der zugehörige Spearmen Korellationskoeffizient von 0,23 und ein p-Wert unterhalb von 0,001 zeigen einen deutlichen Zusammenhang zwischen Qualität der Signaldefinition und Genauigkeit auf. Bei genauerer Betrachtung der Daten sind die Attribute bezüglich ihrer Häufigkeit in folgender Reihenfolge aufgeführt: *Name*, *Länge*, *Datentyp*, *physikalische Einheit* und *Wertebereich*.

Bezüglich der Genauigkeit reicht in den meisten der vorliegenden Fällen das zusätzliche Arrtibut physikalische Einheit aus, um eine korrekte automatische Signalzuweisung zu erreichen (identische Zuweisung wie durch manuelle Analyse). Dies erklärt auch den deutlichen Anstieg der Genauigkeit der automatischen Analyse bei einer Qualität der Signaldefinition von 60 %. Gleichzeitig ist auch deutlich, dass, wie schon durch das Beispiel aus Abbildung 5.7 angedeutet, eine korrekte automatische Zuordnung von Signalen mit dem Datentyp *Boolean*, die maximal die Attribute *Name* und *Datatype* zur Verfügung stellen, ohne weitere Hilfsmittel nicht zuverlässig umgesetzt werden kann (Genauigkeit der automatischen Analyse liegt bei 50 %).

Um in diesen Fällen eine deutlich höhere Genauigkeit zu erreichen, kann ein entsprechender Schwellenwert für die Namensähnlichkeit gesetzt werden.

In Abbildung 5.15 wird der entsprechende Zusammenhang aufgezeigt. Erst ab dem Schwellenwert von 0,81 nimmt die Genauigkeit der automatischen Analyse wieder ab. Die Abnahme der Genauigkeit ist durch die gesteigerte Anzahl falsch negativer Ergebnisse zu begründen (durch die manuelle Analyse identfizierte ähnliche Signalpaare, die als solche nicht erkannt werden). Dieser Zusammenhang ist in Abbildung 5.16 beschrieben. Deutlich ist eine Reduktion der erkannten falschen positiven Resultate zu erkennen (falsche ähnliche Signalpaare).

Gleichzeitig ist ab dem Schwellenwert 0,65 ein erster Anstieg falsch negativer Ergebnisse zu erkennen. Bei weiterer Erhöhung des Schwellenwerts steigt die Zahl der falsch negativen Ergebnisse weiter an bei gleichzeitiger weiterer Reduktion der falsch positiven Ergebnisse. Bei einem Schwellenwert von 0,77 ist die Anzahl falsch positiver Ergebnisse geringer als die Anzahl falsch negativer Ergebnisse. Natürlich spiegelt der beschriebene Zusammenhang nur das gegebene Beispiel wieder. Bei unterschiedlichen Komponentenvarianten und zugehörigen unterschiedlichen Schnittstellen werden unterschiedliche Schwellenwerte ein Optimum an Genauigkeit liefern. Auch ist der deutliche positive Zusammenhang zwischen Schwellenwert und Genauigkeit auf den konsequenten Einsatz einer PERSIST-Signalnamensrichtlinie (siehe 2.2 auf Seite 17) zu-

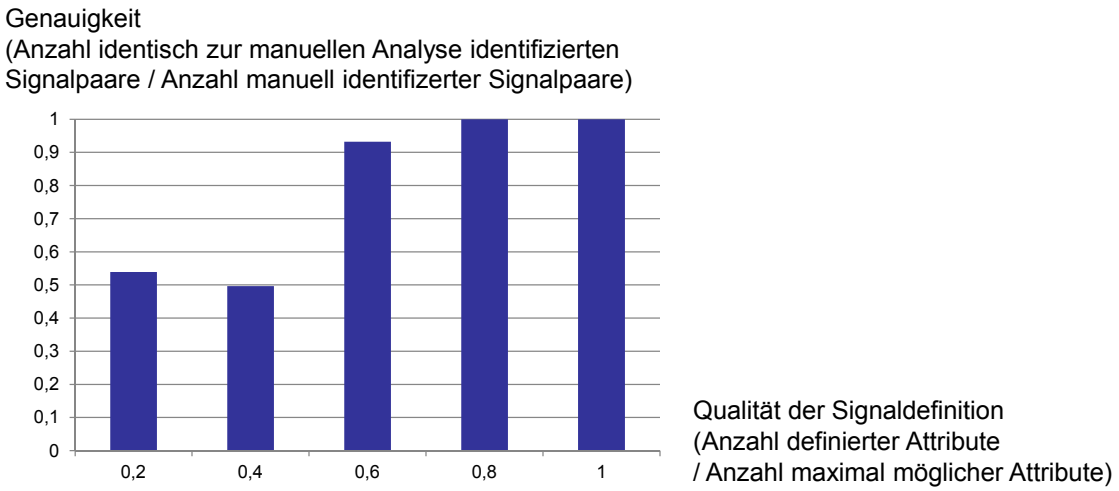


Abbildung 5.14: Genauigkeit der schnittstellen-basierten Ähnlichkeitsanalyse in Relation zur Qualität der Signaldefinition.

rückzuführen. Die problematische Zuweisung von Signalnamen, welche im Beispiel aus Abbildung 5.7 angeführt werden, kann nur durch Einführung falsch negativer Ergebnisse verhindert werden.

Dies ist in diesen Beispielen deutlicher der Fall als in der aktuellen Evaluierung, da eine nicht vorhandene Namensähnlichkeit kein Auschlussprinzip darstellt, insofern keine Signalnamenrichtlinie verwendet wird. Des Weiteren bietet die PERSIST-Signalnamensrichtlinie einen Signalnamenaufbau, der die Gruppierung semantischer Teilaspekte beinhaltet. Als Konsequenz lässt sich von einer lexikalische Ähnlichkeit zweier Signalnamen auf eine semantische Ähnlichkeit schließen. Dies erklärt den in Abbildung 5.15 deutlich dargestellten Zusammenhang zwischen Genauigkeit und dem verwendeten Schwellenwert.

Gilt es nur die Ähnlichkeit zwischen zwei Schnittstellenvarianten zu berechnen, so sollte der Schwellenwert möglichst so gewählt werden, dass die Anzahl falscher Ergebnisse (positiv wie negativ) möglichst minimiert wird. Ist es dagegen wichtig, die Repräsentanten der Menge ähnlicher Datenelemente zu berechnen, um auf deren Basis eine Schnittstelle zu definieren, so sollten falsch negative Ergebnisse möglichst vermieden werden.

Dies liegt daran, dass in einem Folgeschritt falsch positive Ergebnisse entweder durch ein manuelles Review oder durch weitere automatisierte Analyseschritte deutlich leichter identifiziert werden können als falsch negative Ergebnisse. Sind die entsprechenden falsch negativen

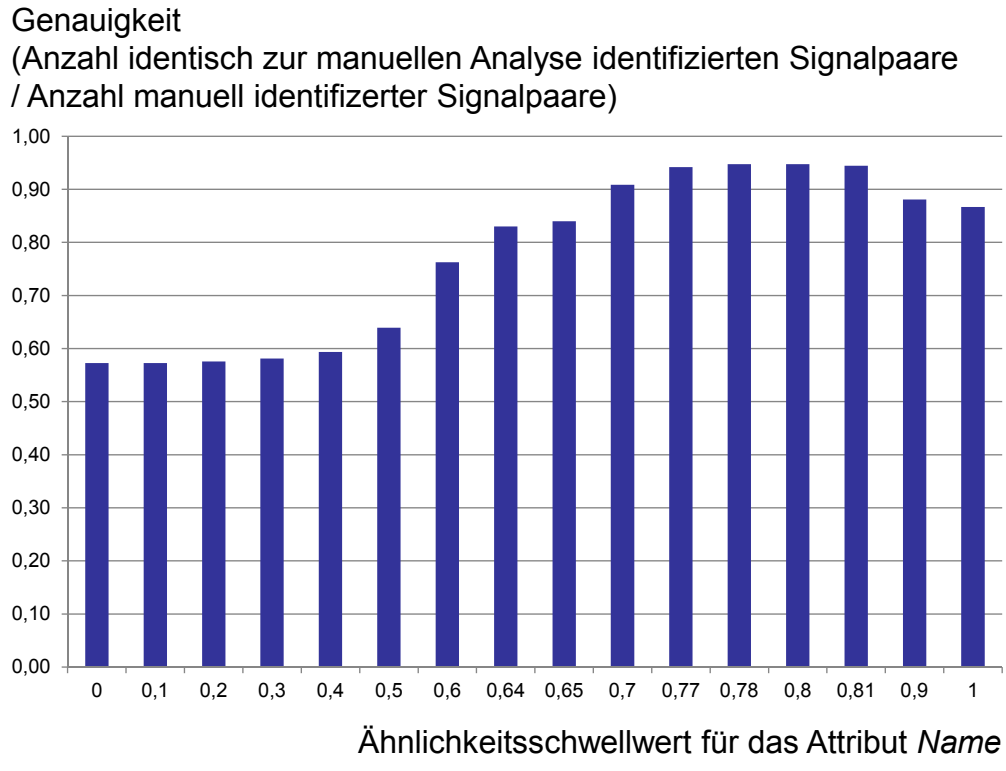


Abbildung 5.15: Genauigkeit der schnittstellen-basierten Ähnlichkeitsanalyse in Relation zum gesetzten Schwellenwert.

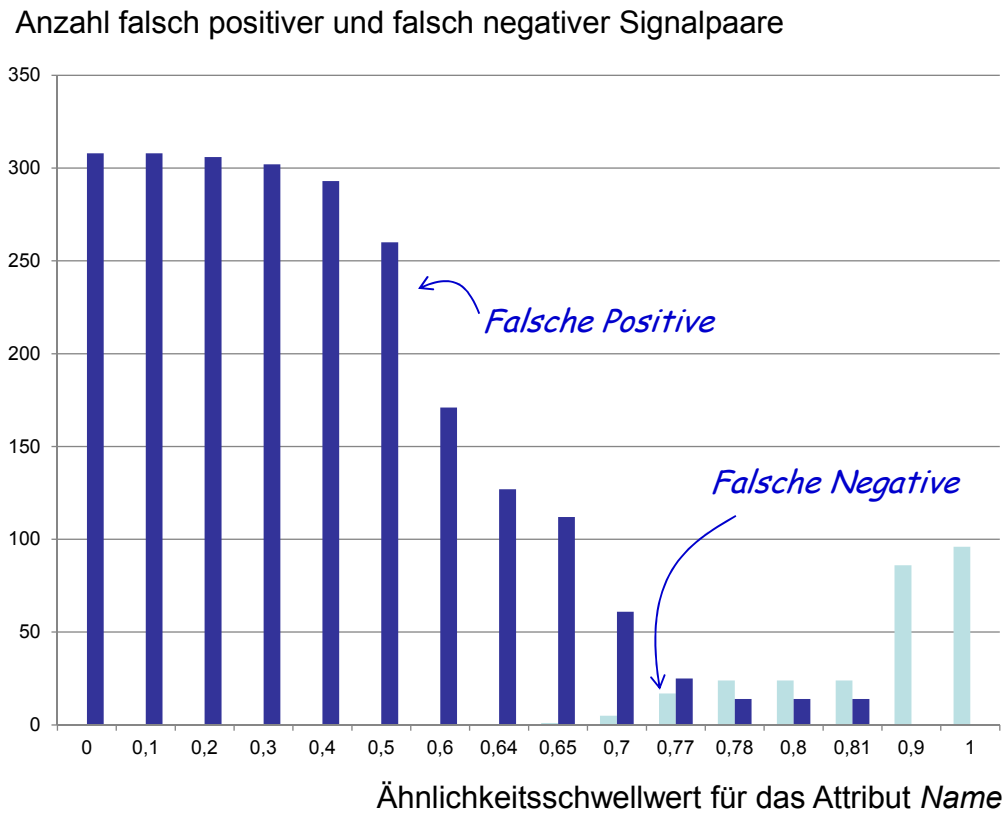


Abbildung 5.16: Falsch positive und falsch negative Ergebnisse der schnittstellen-basierten Ähnlichkeitsanalyse in Relation zum verwendeten Schwellenwerts für die Namensähnlichkeit.

Ergebnisse erst einmal aussortiert, finden sie in weiteren Analysen keine weitere Verwendung.

In einem weiteren Schritt wurden Repräsentanten der Mengen ähnlicher Datenelemente berechnet. Dabei wurde eine durchschnittliche Ähnlichkeit von 29% zwischen den verschiedenen Software-Komponentenvarianten ermittelt. Insgesamt lag die spezifische Ähnlichkeit aller Software-Komponentenvarianten zwischen 11% und 67%.

In einer weiteren Evaluierung wurde untersucht, inwiefern die schnittstellen-basierte Ähnlichkeitsanalyse auch genutzt werden kann, um die Evolution einer einzelnen Schnittstelle zu untersuchen. Das Ergebnis ist in Tabelle 5.17 dargestellt. Hierbei sind neben der Revisionsnummer und entsprechendem Kommentar jeweils die Anzahl an Signalen und Parametern als auch die gerichtete Ähnlichkeit aufgeführt. Diese hilft die Art der Änderung zwischen den einzelnen Revisionen zu unterscheiden. Ist eine Ähnlichkeit nur in eine Richtung reduziert, so wurden entsprechende Signale hinzugefügt oder entfernt, während eine beidseitige Reduktion der Ähnlichkeit auf eine Anpassung von Attributswerten schließen lässt. Liegt die Ähnlichkeit beidseitig auf 100%, so wurde keine Änderung an betrachteten Attributen vollzogen. Die berechneten Ähnlichkeitsverhältnisse innerhalb der Historie einer Schnittstelle waren in den ausgewerteten Beispielen (7 Software-Komponentenvarianten) immer korrekt, obwohl kein Schwellenwert für

Revision	↑	↓	Commit-Kommentar	IN	OUT	CAL
8	1.0	1.0	description	29	25	28
7	1.0	0.84	branch merge	29	25	28
6	1.0	0.98	new signal	21	23	25
5	1.0	1.0	description	20	23	25
4	1.0	0.911	new signals	20	23	25
3	1.0	0.85	new signals	20	17	25
2	0.99	0.99	typos	19	17	17
1	-	-	creation	19	17	17

Tabelle 5.17: Revisionshistorie mit Ähnlichkeitsrelation auf Basis der schnittstellen-basierten Ähnlichkeitsanalyse zwischen einzelnen Revisionen.

die Namensähnlichkeit verwendet wurde. Dies lässt sich auf den starken Zusammenhang der einzelnen Revisionen einer Schnittstelle zurückführen. Da innerhalb eines Commits niemals unterschiedliche Änderungsoperationen (hinzufügen, umbenennen, entfernen) gemischt wurden, ist eine hohe Genauigkeit nicht verwunderlich.

Nichtsdestotrotz können die vorgestellten Ergebnisse nicht generalisiert werden. Die verglichenen Komponenten sind alle durch eine kleine Gruppe Entwickler spezifiziert. Des Weiteren beziehen sich die ausgewertete Modelle alle auf eine Subdomäne innerhalb der Automobilentwicklung. In diesem Kontext konnte aber ein deutlicher Zusammenhang zwischen Qualität der Signaldefinition und Genauigkeit sowie Schwellenwert der Namensähnlichkeit und Genauigkeit hergestellt werden. Insofern konnten im aktuellen Kontext die Hypothesen **H1** und **H2** bekräftigt werden.

Kapitel 6

Semantische Ähnlichkeiten

Wie schon in Abschnitt 4.1 erläutert, können Klone generell in vier unterschiedliche Kategorien eingeteilt werden [RBS13]: exakte Klone (Typ 1), umbenannte/parametrisierte Klone (Typ 2), sehr ähnliche Klone (Typ 3) und semantische Klone (Typ 4). Zur Unterstützung einer produktgetriebenen Software-Produktlinienentwicklung sind alle vier Typen relevant. Um unterschiedliche Varianten zu identifizieren, sind generell Klone von Typ 2 bis Typ 4 interessant. Basierend auf der Annahme, dass nur historische Klone gegeben sind, wäre ein Fokus auf Typ 3 Klone (sehr ähnliche Klone) ausreichend, da auf Grund der zugehörigen Historie in den meisten Fällen eine gemeinsame syntaktische Basis gegeben ist. Nur bei einer längerfristigen Historie, die auch Refactoring-Maßnahmen ohne funktionale aber mit deutlichen strukturellen Änderungen beinhaltet, wären Mechanismen zur Identifikation von Typ3-Klonen nicht ausreichend. Geht man stattdessen auch von parallelen und vollständig losgelösten Entwicklungen von Klonen aus so ist eine gemeinsame strukturellen und syntaktischen Basis über die Schnittstelle hinaus meist nicht gegeben. Insofern wäre es notwendig ein Klonerkennungsverfahren zu nutzen, das in der Lage ist semantische Klone zu identifizieren.

Erweiterte endliche Eingabe / Ausgabe Automaten (I/O-EFAs) [Rum96, ZK12] stellen ein mögliches Format dar, auf deren Grundlage durch die Berechnung von Simulationsrelationen [Gla01] eine semantische Gleichheit zweier Automaten hergestellt werden kann. Problematisch bei der Berechnung einer Simulationsrelation ist die Transformation in ein Transitionssystem. Dabei werden die internen Variablen durch konkrete Zustände ersetzt, was bei mehreren internen Variablen mit größeren Wertebereichen schnell zum Zustandsraumexplosionsproblem führen kann [BK08]: Der Zustandsraum wächst exponentiell in Relation zur Anzahl der internen Variablen (fünf interne Variablen mit jeweils 10 expliziten Werten, würden einen Zustandsraum von 10^5 ergeben). Generell sind semantische Analysen entsprechender Modelle genauer, stellen zugehörige Verfahren aber vor ein NP-hartes Problem [RBS13] (siehe auch Abschnitt 4.1). Um Ergebnisse in praktikabler Zeit zu erhalten, ist es notwendig durch entsprechende Approximation oder weitere Hilfsschritte den potentiellen Zustandsraum vor einer Transformation in ein Transitionssystem zu reduzieren. Im Kontext einer semantischen Ähnlichkeitsanalyse können auch ungenauere, aber effizientere Verfahren auf Basis eines I/O-EFAs durchgeführt werden, um semantische Gemeinsamkeiten zu identifizieren. Dabei ist es nicht unbedingt notwendig, diese durch Berechnung einer Simulationsrelation zu realisieren. Ähnlich zu unterschiedlichen Verfahren der semantischen Klonerkennung [GJS08, TG06, FFK08] (siehe auch Abschnitt 4.1) ist es auch im Sinne eines Hybriden nach Transformation in einen I/O-EFA möglich, syntaktische Analysen zur Ähnlichkeitsanalyse zu nutzen. Dabei ist es hilfreich, wenn vorher durch eine schnittstellenbasierte Ähnlichkeitsanalyse syntaktisch unterschiedliche Signale vereinheitlicht

werden können.

Des Weiteren stellt sich die Frage, welche Entwicklungsartefakte für eine entsprechende Extraktion beziehungsweise Ähnlichkeitsanalyse zu welchem Zeitpunkt zur Verfügung stehen. Bezogen auf den Prozess, der in Kapitel 3 beschrieben wurde, ist es wünschenswert möglichst frühzeitig innerhalb der Entwicklung die entstehenden Artefakte bezüglich ihrer Semantik zu vergleichen, um redundante Entwicklungen zu vermeiden.

Bezogen auf einen typischen Softwareentwicklungsprozess innerhalb der Automobilindustrie wären somit textuelle Anforderungen die ersten Entwicklungsartefakte, die zur Verfügung stehen, um einen semantischen Abgleich zu ermöglichen. Wie schon in Abschnitt 4.1 aufgeführt, gestaltet sich eine semantische Analyse von Prosatexten auf Grund ihrer informellen Natur als sehr schwierig [Hol10]. Insofern ist es notwendig, auf eine formalere Darstellung der Anforderungen zurückzugreifen. Diese kann entweder in Form von Modellen geschehen, die als detailliertere Anforderungsspezifikation gesehen werden können, oder auch in Form von Testfällen, die zur Überprüfung der Anforderungen definiert sind. In einer testgetriebenen Entwicklung stehen die entsprechenden Testfälle vor der Implementierung zur Verfügung und stellen eine formale Approximation des Verhaltens dar. Wie genau diese Approximation bezüglich der Implementierung ist, lässt sich in einer späteren Phase durch Bestimmung der Testabdeckung genauer festlegen.

Zusammengefasst sind nun folgende Überlegungen gegeben:

1. Um alle möglichen Szenarien einer Produktextraktion und Produktsynchronisation abdecken zu können, ist es notwendig, Klone bis Typ 4 (semantische Klone) zu identifizieren.

2. Semantische Analysen sind meist NP-hart, insofern bedarf es einer Approximation der gegebenen Verhaltensbeschreibung oder erweiterte Transformationsschritte auf Basis der zu erwartenden Ähnlichkeit, bevor eine Transformation in ein Transitionssystem durchgeführt werden kann.

3. Eine weitere Alternative besteht darin, nach einer Transformation in eine formale Struktur semantische und syntaktische Verfahren zu kombinieren und eine Transformation in ein Transitionssystem zu vermeiden.

4. Entsprechend der produktgetriebenen Software-Produktlinienentwicklung sollten möglichst Artefakte genutzt werden, die frühzeitig in der Entwicklung zur Verfügung stehen.

5. Anforderungen in Prosatexten stehen frühzeitig zur Verfügung, sind aber für eine fundierte Analyse nicht formal genug.

6. Testfälle sind eine formalere Darstellung von Anforderungen und stellen eine Approximation des Gesamtverhaltens dar.

7. Testfälle können in einer testgetriebenen Entwicklung frühzeitig zur Verfügung stehen.

Zusammenfassend bietet die Testfallspezifikation alle notwendigen Eigenschaften, um eine effiziente und hinreichend valide (abhängig vom Grad der Testabdeckung) semantische Ähn-

lichkeitsanalyse zu ermöglichen. Modelle, die das Verhalten explizit spezifizieren, wie zum Beispiel Simulinkmodelle, stellen eine exakte Darstellung dar, müssen aber approximiert oder durch frühzeitigen Abgleich reduziert werden, um Ergebnisse in praktikabler Zeit zu ermöglichen.

Im Folgenden wird nun auf Basis der gerade durchgeführten Argumentation ein Verfahren vorgestellt, das durch Vergleich gegebener Verhaltensmodelle oder Testspezfikationen einzelner Funktionen eine semantische Ähnlichkeitsanalyse durchführt. Dabei werden unterschiedliche Ansätze vorgestellt, um die gegebenen Entwicklungsartefakte in I/O-EFAs zu überführen. Diese sind in Abbildung 6.1 skizziert: Abhängig vom zugrundeliegenden Entwicklungsprozess und dem aktuellen Stand innerhalb eines Projektes können Informationen entweder auf Basis einer Verhaltens- oder einer Testspezifikation gewonnen werden (①). Liegt ein Modell vor, das das konkrete Verhalten spezifiziert, so kann dieses in einen I/O-EFA überführt werden (②). In einem weiteren Schritt wird versucht durch Abgleich mit dem I/O-EFA, der das Verhalten der verglichenden Funktionvariante beschreibt, den potentiellen Zustandsraum zu reduzieren (③). In diesem Schritt werden auch ungenauere, aber effizientere syntaktische Verfahren zur Ähnlichkeitsanalyse verwendet werden, um weitere Indikatoren zur Ähnlichkeitsanalyse zur Verfügung zu stellen. War die Reduktion erfolgreich oder ist der potentielle Zustandsraum in einer prakti-

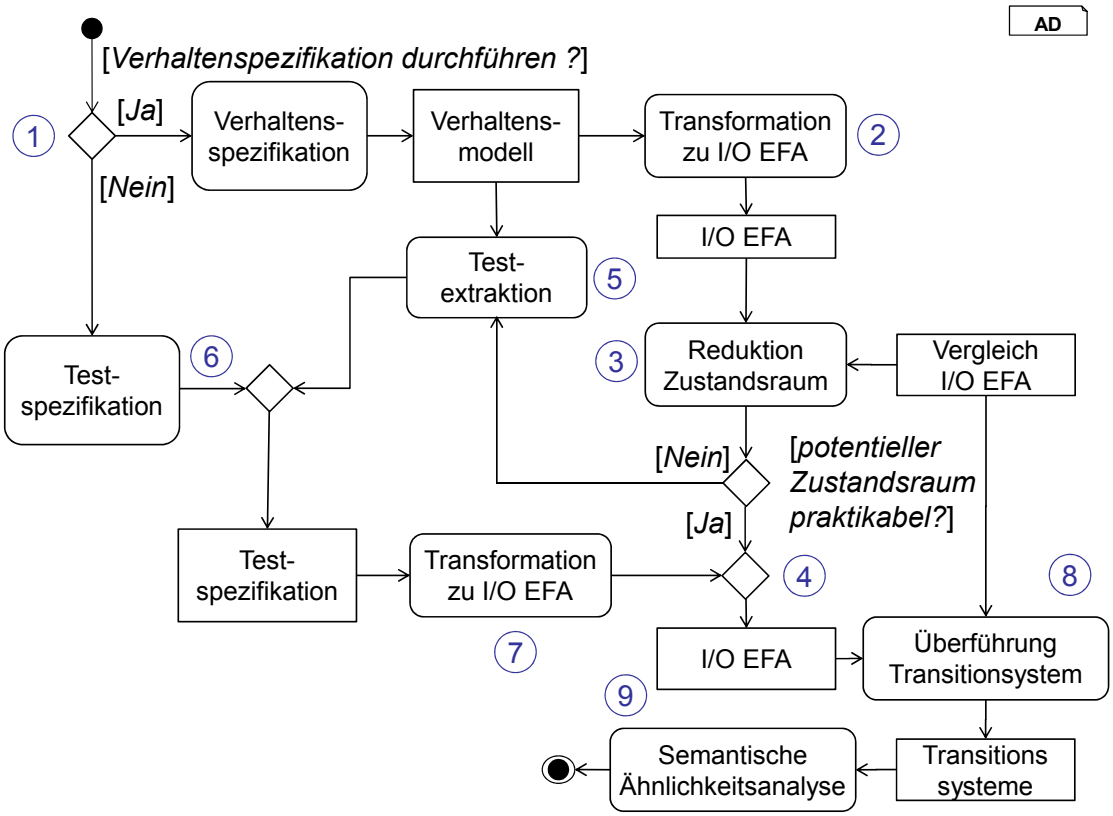


Abbildung 6.1: Mögliche Artefaktrollen zur Transformation in einen I/O-EFA und weiterführende Schritte zur semantischen Ähnlichkeitsanalyse.

kablen Größe gegeben, so wird der resultierende I/O-EFA für weitere Analyseschritte verwendet (④). Ist dagegen der potentielle Zustandsraum zu groß für weitere Analysen, kann das Verhaltensmodell genutzt werden, um mittels einer Testextraktion eine Testspezifikation als Approximation des definierten Verhaltens zu erlangen (⑤). Als Ergebnis liegt nun eine Testspezifikation vor - dies wäre auch der Fall, falls initial direkt nur eine manuelle Testspezifikation durchgeführt worden wäre (⑥).

In beiden Fällen wird die Testspezifikation in einen I/O-EFA überführt (⑦). Dieser I/O-EFA beinhaltet auf Grund der Art der Quellartefaktrolle keinerlei interne Zustände. Testspezifikationen definieren entsprechende Zustände nur explizit. Sie bieten deswegen eine approximierte, aber dafür praktikablere Verhaltensbeschreibung. Abschließend werden beide I/O-EFAs in Transitionssysteme überführt und eine semantische Ähnlichkeitsanalyse durchgeführt. Ist eine Transformation oder eine Testextraktion nicht möglich, so bietet der syntaktische Abgleich extrahierter I/O EFAs auf Basis vorliegender Verhaltensmodelle schon in Schritt drei erste Ergebnisse.

6.1 Konzept

Definition 6.1 *Semantische Ähnlichkeit. Die semantische Ähnlichkeit zweier Funktionen ist definiert als Grad des gleichen Verhaltens.*

Definition 6.2 *Semantische Ähnlichkeitsanalyse. Die semantische Ähnlichkeitsanalyse berechnet die semantische Ähnlichkeit zweier Funktionsvarianten bezogen auf einzelne ausgehende Datenelemente der Schnittstellenvarianten, die durch eine vorherige schnittstellenbasierte Ähnlichkeitsanalyse Teil der gleichen Menge ähnlicher Datenelemente sind.*

Um eine entsprechende semantische Auswertung durchführen zu können (bzw. diese deutlich zu erleichtern), ist es notwendig, gegebene Verhaltensbeschreibungen auf eine gleiche Sprachform zu überführen, auf welcher dann die semantische Analyse durchgeführt werden kann. Dieser initiale Schritt ermöglicht es, das gegebene Verfahren in weiteren Schritten durch Unterstützung weiterer Artefakttypen (unterschiedliche Verhaltensbeschreibungssprachen) zu erweitern. Zum Beispiel sind unterschiedliche Sprachen zur Testfall- (Signal Builder oder CTE) oder Verhaltensspezifikation (z.B. Simulinkmodelle, C Code oder Stateflows) möglich.

Das generelle Zielformat, auf dem die semantische Ähnlichkeitsanalyse durchgeführt wird, ist der I/O-EFA. Dieses ermöglicht eine Verhaltensbeschreibung einer Funktion auf Basis der Eingangs- und Ausgangsvariablen sowie weiterer interner Variablen:

Definition 6.3 *I/O-EFA. Ein I/O-EFA $A = (S, s_0, D, d_0, U, Y, E, f_0)$ besteht aus einer Menge von Zuständen S (sowie Startzustand $s_0 \in S$), einer Menge von internen Variablen D (sowie Initialbedatung d_0), einer Menge von Eingangs- und Ausgangsvariablen U und Y sowie einer Menge von Transitionen E ($o_e \xrightarrow[y=h_e(d,u);d=f_e(d,u)]{[g_e(d,u)]} t_e$). Des Weiteren ist ein Endzustand $f_0 \in S$ gegeben [ZK12]. Ein Wechsel zwischen Startzustand $o_e \in S$ zu einem Zielzustand $t_e \in S$ wird ausgeführt, wenn die entsprechende Bedingung $g_e(d, u)$ gilt und der Automat sich aktuell im entsprechenden Startzustand befindet. Vor dem Zustandswechsel werden entsprechend die*

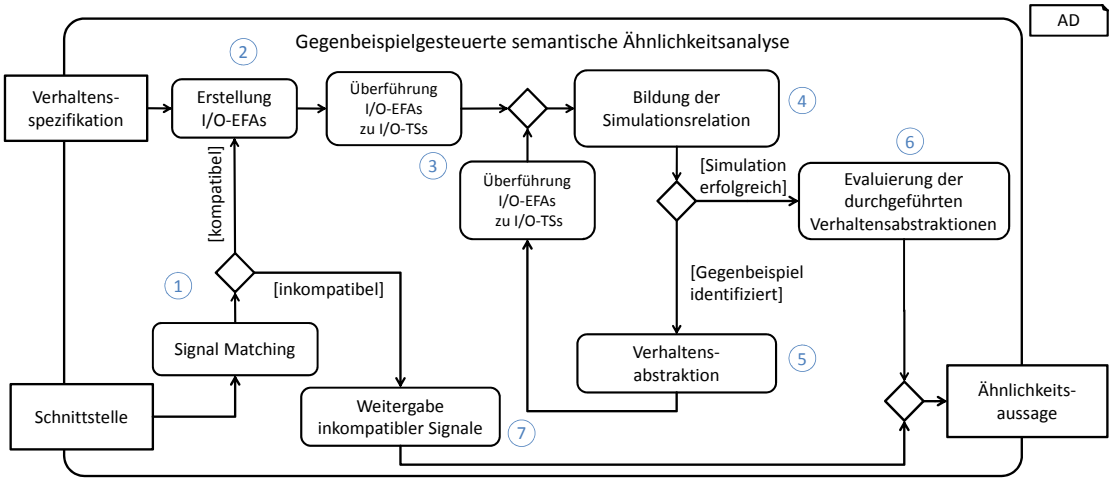


Abbildung 6.2: Grundsätzlicher Ablauf der gegenbeispielgesteuerten semantischen Ähnlichkeitsanalyse [Thi15].

Ausgabefunktion $h_e : D \times U \rightarrow Y$ und die Update-Funktion $f_e : D \times U \rightarrow D$ auf Basis der aktuellen Eingabewerte und der aktuellen internen Werte ausgeführt.

Das Grundkonzept der semantischen Ähnlichkeitsanalyse besteht darin, das Verhalten beider Funktionsvarianten solange zu abstrahieren, bis ein identisches Verhalten übrig bleibt. Dazu wird schrittweise unähnliches Verhalten entfernt. Der anschließend gemessene Grad der Abstraktion, der notwendig ist, um eine Verhaltensgleichheit zu erlangen, spiegelt dann indirekt den Grad der semantischen Ähnlichkeit wieder.

Definition 6.4 *Gegenbeispielgesteurte semantische Ähnlichkeitsanalyse (CEGAS). Die gegenbeispielgesteuerte semantische Ähnlichkeitsanalyse (Counter Guided Abstraction Similarity Metric (CEGAS)) abstrahiert schrittweise das Verhalten beider Funktionsvarianten bis eine Simulationsrelation zwischen beiden gebildet werden kann. Dazu wird schrittweise versucht eine Simulationsrelation zu berechnen. Ist dies nicht möglich, so wird ein Gegenbeispiel berechnet. Dieses Gegenbeispiel wird verwendet um den I/O-EFA der zu simulierenden Funktionsvariante durch Entfernung des zum Gegenbeispiel führenden Verhaltens soweit zu abstrahieren, dass das zugehörige Gegenbeispiel bei einer weiteren Berechnung der Simulationsrelation nicht wieder auftritt. Dieser Vorgang wird solange wiederholt, bis eine Simulationsrelation berechnet werden kann.*

In Abbildung 6.2 sind die generellen Schritte der gegenbeispielgesteuerte semantischen Ähnlichkeitsanalyse dargestellt.

Im Folgenden wird kurz auf die einzelnen Schritte eingegangen, um einen Gesamtüberblick zu ermöglichen. Anschließend wird jeder einzelne Teilschritt im Detail erläutert.

Da das Verhalten auf Basis von Ein- und Ausgangsvariablen beschrieben ist, muss in einem initialen Schritt eine Zuordnung der Elemente der entsprechenden Schnittstellen beider Funkti-

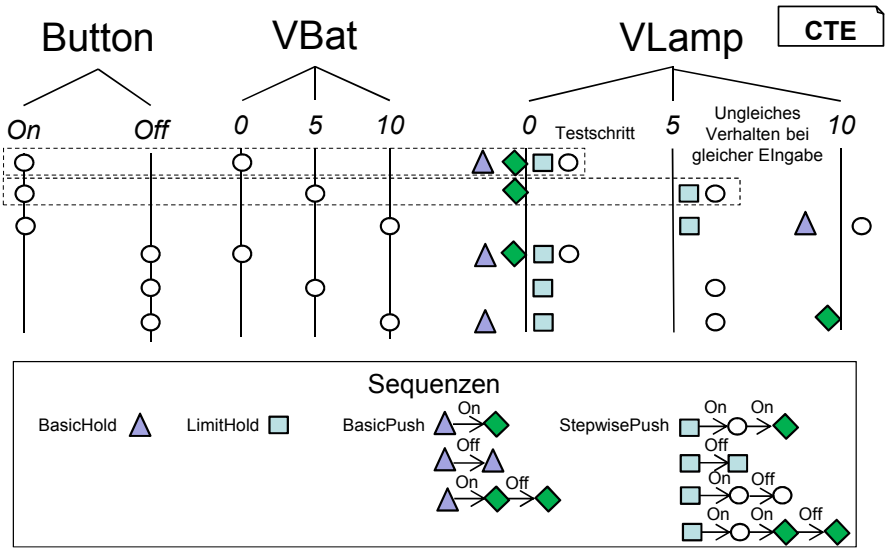


Abbildung 6.3: Testsequenzen für verschiedene Lichtsteuereinheiten.

onsvarianten durchgeführt werden (①). Dazu wird eine schnittstellenbasierte Ähnlichkeitsanalyse durchgeführt, wie in Abschnitt 5.2 beschrieben.

Das Resultat dieser Analyse beinhaltet unter anderem die Menge ähnlicher Datenelemente für die Ausgabeelemente der Schnittstellenvarianten. Auf Basis dieser Mengen ähnlicher Datenelemente kann in einem folgenden Schritt eine semantische Ähnlichkeitsanalyse durchgeführt werden. Ist keine Menge ähnlicher Datenelemente für Ausgabeelemente gegeben, so ist auch eine weitere semantische Analyse nicht möglich. Stattdessen reichen die Ergebnisse der schnittstellenbasierten Ähnlichkeitsanalyse aus (⑦).

Ist eine Zuordnung erfolgt, so werden initial die gegebenen Verhaltensmodelle in I/O-EFAs überführt (②). Welche Artefaktrollen für eine Transformation in I/O-EFAs genutzt werden, ist im Detail in Abbildung 6.1 beschrieben.

Um in einem folgenden Schritt eine mögliche Verhaltensgleichheit durch Herstellung einer Simulationsrelation [Gla01] zu identifizieren, ist es notwendig, den gegebenen I/O-EFA in ein I/O-Transitionssystem (I/O-TS) zu überführen (③). Dabei werden alle internen Variablen in explizite Zustände transformiert und die Ausgabefunktionen von der Transition zum Zustand hin verschoben.

Auf Basis der I/O-TSs wird anschließend eine Simulationsrelation berechnet (④). Kann keine Relation hergestellt werden, so wird ein entsprechendes Gegenbeispiel erzeugt, das die Eingabekette beschreibt, die ungleiches Verhalten zwischen zwei Funktionsvarianten hervorruft.

Als Reaktion auf die gescheiterte Simulation wird das Verhalten der Funktionsvarianten dahingehend abstrahiert, dass sich für das gegebene Gegenbeispiel eine Verhaltensgleichheit ergibt. Anschließend wird wiederum eine Simulationsrelation berechnet (⑤).

Ist diese gegeben, so ist keine weitere Abstraktion mehr notwendig und das Maß der vollzogenen Abstraktionen wird evaluiert, um eine semantische Ähnlichkeit zu berechnen (⑥).

Im Folgenden werden die genannten Einzelschritte im Detail erläutert.

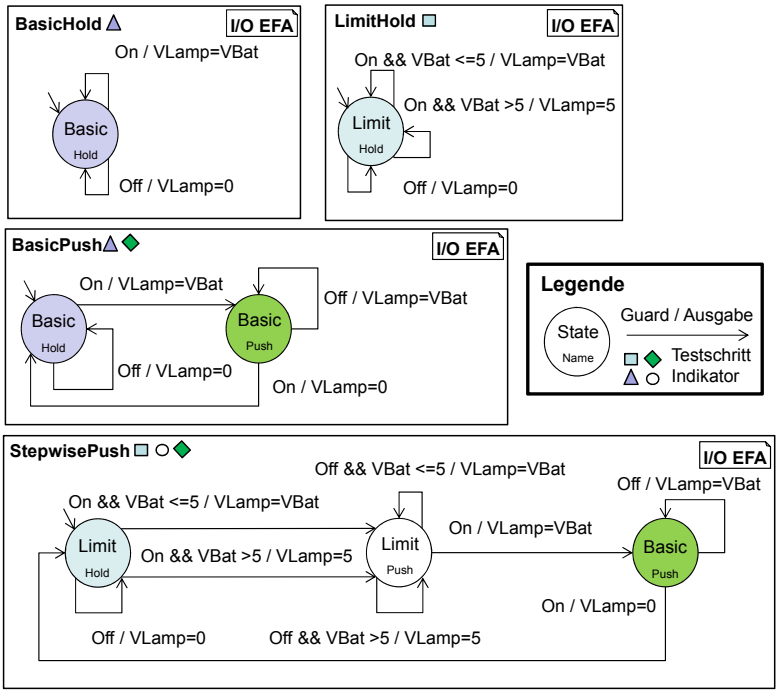


Abbildung 6.4: I/O-EFAs basierend auf den Testspezifikationen in Abbildung 6.3.

Dazu wird das in Abbildung 6.3 als Testspezifikationen und in Abbildung 6.4 als I/O-EFAs aufgeführte Beispiel genutzt. Des Weiteren sind die Beispiele in Abbildung 6.5 und Abbildung 6.6 auch als Simulink- bzw. Stateflow-Modelle gegeben. Die verschiedenartigen Umsetzungen in Simulink sind bewusst so gewählt, um die Vorteile einer semantischen Analyse gegenüber einer rein syntaktischen Analyse darzustellen. Deswegen ist auch in Abbildung 6.6 eine von Stateflow verschiedene Realisierung unterschiedlicher Zustände (über *Delay* Block) dargestellt. Dieses fiktive und vereinfachte Beispiel stellt vier Varianten einer Lichtsteuereinheit dar. Diese vier Varianten beschreiben teilweise ähnliches Verhalten auf Basis derselben Schnittstelle: Eingaben sind durch die aktuelle Volt-Spannung einer angeschlossenen Batterie (*VBat*) und einen Indikator, inwiefern ein zugehöriger Lichtschalter gedrückt ist (*Button*), gegeben.

Aufgabe aller vier Funktionsvarianten ist es, auf Basis dieser Eingaben zu bestimmen, welche Spannung an die Lichteinheit weitergegeben wird (*VLamp*), also wie stark das Licht leuchten soll.

Die einfachste Steuereinheit (*BasicHold*) übergibt die gegebene Batteriespannung direkt an die Lichteinheit, solange der Lichtschalter gedrückt ist. Ist er nicht gedrückt, so wird keinerlei Spannung weitergegeben - das Licht bleibt aus.

Die Variante *LimitHold* unterscheidet sich zur Basis nur dadurch, dass die Ausgangsspannung auf maximal fünf Volt limitiert wird.

Die Variante *BasicPush* ist die erste Variante, die mehrere Zustände beinhaltet: Wurde der Lichtschalter einmal gedrückt, wird die Batteriespannung direkt an die Lichteinheit weitergegeben (Licht ein). Wird der Schalter anschließend ein zweites Mal gedrückt, so wird wiederum

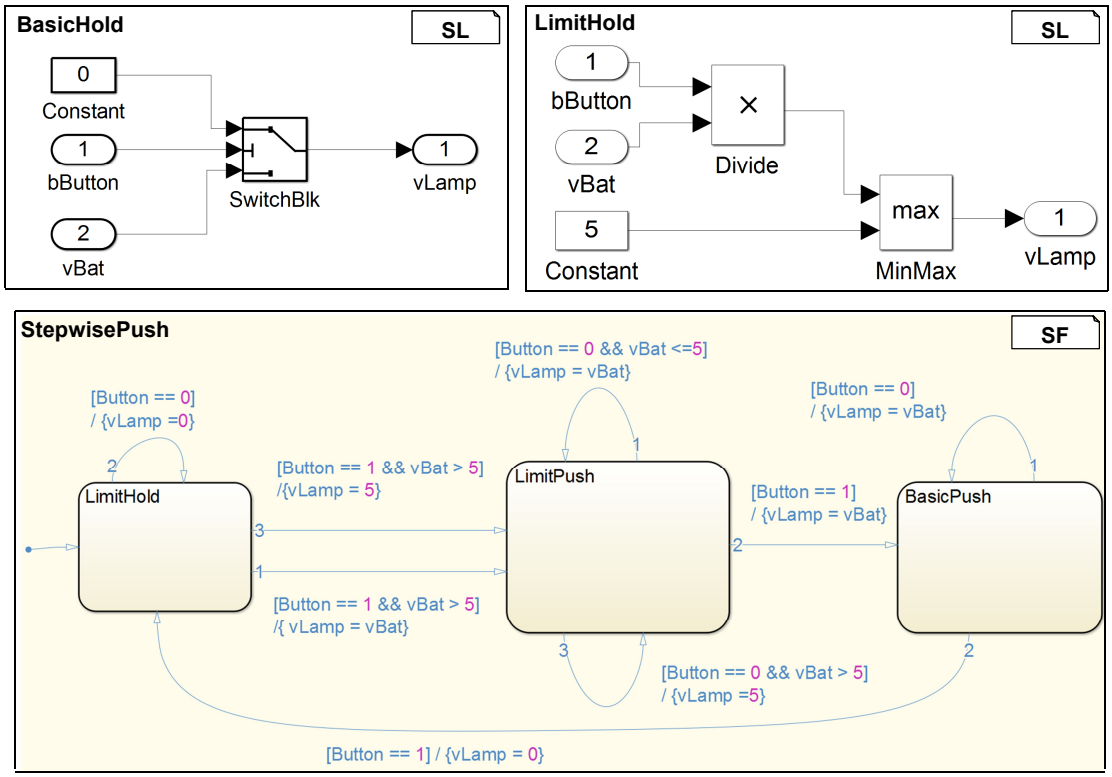


Abbildung 6.5: *BasicHold*, *LimitHold* und *StepwisePush* als Simulink- beziehungsweise Stateflow-Modell.

keine Spannung übertragen (Licht aus).

Die komplizierteste Variante *StepwisePush* verbindet die vorher aufgeführten Funktionalitäten: Wird der Lichtschalter einmalig gedrückt, so wird die Batteriespannung auf maximal fünf Volt limitiert übertragen (gedimmtes Licht), während beim zweiten Betätigen des Lichtschalters die Spannung ganz übertragen wird (volles Licht). Bei der dritten Betätigung des Lichtschalters ist die Übertragung wieder auf null reduziert (Licht aus).

Die in Abbildung 6.3 aufgeführte Testspezifikation beinhaltet alle Varianten und ist als Klassifikationsbaum [GG93] dargestellt. Die Klassifikationsbaummethode [GG93] ist ein Partitionstestmethode [OB88], die die systematische Erstellung von BlackBox-Tests auf Basis von Äquivalenzklassen ermöglicht, die über den Eingangs- und Ausgangsvariablen gebildet werden. Dabei werden in einem ersten Schritt die Äquivalenzklassen der Eingangs- und Ausgangsvariablen definiert. Diese sind in Abbildung 6.3 direkt unter den Repräsentanten der Äquivalenzklassen (Eingaben links, *On*/*Off* für *Button* und 0/5/10 für *VBat*, Ausgaben rechts, nur 0/5/10 für *VLamp*) dargestellt. Basierend auf diesen Äquivalenzklassen beziehungsweise deren Repräsentanten werden Kombinationen gebildet, um einzelne Testschritte zu definieren. Diese Testschritte können wiederum in einer festgelegten (zeitlichen) Reihenfolge verbunden sein, um eine Testsequenz zu definieren.

In Abbildung 6.3 sind unterschiedliche Testschritte, die auf Basis derselben Eingaben unterschiedliches Verhalten definieren, durch unterschiedliche Symbolik (Dreieck, Raute, Quadrat, Kreis) und Farbe (Lila, Grün, Hellblau, Weiß) in derselben Zeile dargestellt, um die unterschiedlichen beschriebenen Varianten übersichtlich darzustellen.

Separat sind wiederum weiter unten die Testsequenzen den einzelnen Varianten der Lichtsteuereinheiten zugeordnet. Der Übergang von einem Testschritt zu einem anderen ist dabei durch die Eingangsvariable *Button* definiert. Zyklen wiederum sind durch sich wiederholende Testschritte angegeben (z.B. dritte Testsequenz für *BasicPush*). Die vier unterschiedlichen Symbole stellen dabei die vier unterschiedlichen Teilzustände dar, die sich in der einen oder anderen Variante wiederfinden lässt.

Dieselben Varianten sind genauso auch als I/O-EFAs in Abbildung 6.4 dargestellt. Dort wiederum wurden dieselben Farben verwendet, um einen Abgleich zwischen Automat und Testfallspezifikation zu erleichtern. Auch sind die zugehörigen Symbole nochmals neben dem Automatennamen dargestellt. Neben der entsprechenden Transition sind Bedingung und Ausgabefunktion durch einen Querstrich getrennt. Interne Variablen sind nicht gegeben.

Abschließend sind die beschriebenen Varianten auch als Simulink- beziehungsweise Stateflow-Modelle in Abbildung 6.5 und Abbildung 6.6 dargestellt. Bewusst sind in diesen Beispielen die Varianten syntaktisch unterschiedlich realisiert, so dass ein syntaktisches Verfahren zugehörige Ähnlichkeiten nicht identifizieren kann. Zum Beispiel ist in *BasicHold* die direkte Weiterleitung der Spannung bei gedrücktem Knopf durch einen Switch-Block realisiert, während dieselbe Funktionalität in *LimitHold* durch einen Multiplikator realisiert ist. Tortz syntaktischer Unterschiede führen beide Varianten zum gleichen Ergebnis ($bButton == 0 => VLamp =$

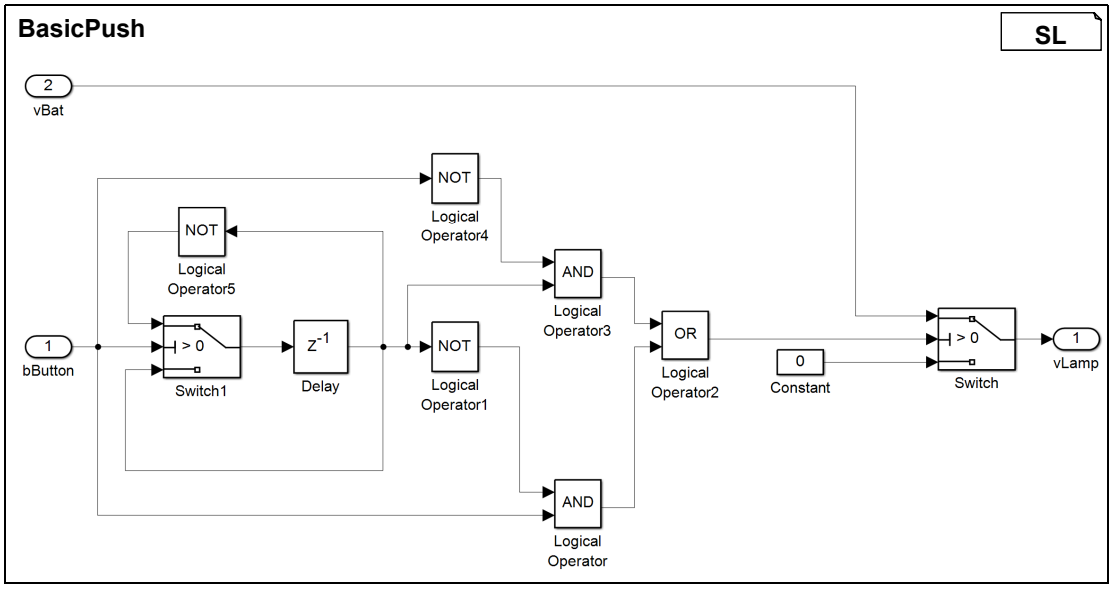


Abbildung 6.6: *BasicPush* als Simulinkmodell (*Delay*-Block zur Beschreibung einer internen Variable).

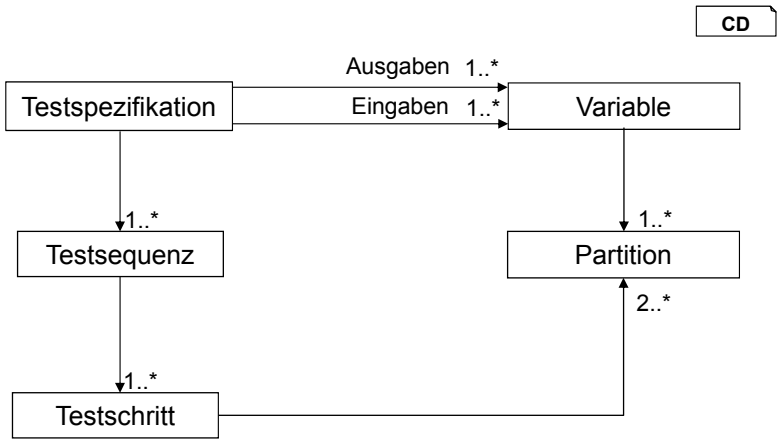


Abbildung 6.7: Notwendige Informationen einer Testspezifikation für eine Transformation zu einem I/O-EFA.

$0, bButton == 1 => vLamp = VBat$). Im dritten Beispiel aus Abbildung 6.5 wird die Funktionalität durch Stateflow-Modell realisiert, eine syntaktische Analyse mit den ersten beiden Beispielen ist nur bedingt möglich. Als Gegenbeispiel wiederum ist *BasicPush* in Abbildung 6.6 mittels *Delay*-Block realisiert, um den Zustand des gedrückten Schalters zu realisieren. Auch hier wäre eine Stateflow-Repräsentation sinnvoll, diese könnte aber syntaktisch mit dem Modell aus Abbildung 6.6 nicht verglichen werden.

6.1.1 Transformation von Testspezifikationen zu I/O-EFAs

Die Transformation der gegebenen Verhaltensspezifikationen zu I/O-EFAs sind jeweils typabhängig. Dabei können für ähnliche Artefakttypen auch Vortransformationen definiert werden, um die Unterstützung weiterer Entwicklungsartefakte zu vereinfachen. Zum Beispiel können unterschiedliche Formate von Testspezifikationen schon auf ein generalisiertes Format überführt werden, bevor eine Transformation in I/O-EFAs durchgeführt wird. Auch für Simulink-Modelle und C Code ist eine Kontrollflussgraphendarstellung ein sinnvolles Zwischenformat, bevor eine finale Transformation durchgeführt wird.

Im Folgenden wird im Detail die Transformation von Testspezifikationen in Form von Klassifikationsbäumen zu I/O-EFAs auf Basis der gegebenen Beispiele aus Abbildung 6.3 und Abbildung 6.4 erläutert (Schritt ⑦ aus Abbildung 6.1). Dieses Verfahren wurde schon in [RRS$^+$16] beschrieben. Auch eine Unterstützung von SignalBuildern, ein Testfallspezifikationstyp innerhalb von Simulink, wurde umgesetzt [Ess16], um eine Transformation von Simulink-Modellen zu I/O-EFA über den Design Verifier zu ermöglichen. Des Weiteren wurden Transformationen von Simulinkmodellen zu CFGs und in einem weiteren Schritt zu I/O-EFAs [RSW$^+$15] umgesetzt. Diese Verfahren ist inspiriert von [ZK12] und wird abschließend besprochen.

Für eine Transformation in einen I/O-EFA müssen Testspezifikationen generell die in Abbildung 6.7 dargestellten Informationen beinhalten: Eine Testspezifikation beinhaltet eine Menge an Variablen, die entweder als Eingangs- oder Ausgangsvariable definiert sind und deren Wer-

tebereich in endlich viele Teilstücke partitioniert ist (Äquivalenzklassen). Des Weiteren ist eine Menge an Testsequenzen definiert, die wiederum eine geordnete Menge an Testschritten beinhaltet. Ein Testschritt wiederum referenziert für jede Variable eine konkrete Partitionierung.

Auf Basis dieser Informationen und weiterführenden Informationen bezüglich der Variable durch die gegebene Schnittstelle kann eine Transformation zu I/O-EFAs wie folgt stattfinden:

Ein- und Ausgangsvariablen: Die Ein- und Ausgangsvariablen und entsprechende Datentypen und Wertebereiche können auf Basis der Schnittstelle übernommen werden. Für jede Ausgangsvariable wird dabei separat ein Automat erstellt, der die Berechnung der Variable beschreibt. Dies erlaubt semantische Ähnlichkeiten für jede Ausgangsvariable getrennt zu evaluieren.

Interne Variablen: Testspezifikationen beschreiben keine internen Variablen, folglich werden auch bei der Transformation keine erstellt.

Zustände: Der I/O-EFA besteht mindestens aus dem Initialzustand s_0 und dem Endzustand f_0. Für jede Transition $e_i \in E$, die von einem Testschritt i einer Testsequenz der Länge n erstellt wird, wird ein eigener Zielzustand $s_i \in S$ erstellt, falls eine disjunkte Bedingung der Transition oder eine Wiederholung der Transition vorliegt (siehe Transitionen) und der Testschritt nicht der letzte Schritt der Sequenz ist ($i \neq n$).

Transitionen: Die Transformation in Transitionen stellt den komplexesten Teil der Transformation dar, da folgende Aspekte berücksichtigt werden müssen:

1. **Darstellung von Schleifen:** Semantisches Verhalten beinhaltet des öfteren Schleifen, die in Automaten gut beschrieben werden können. In üblichen Formaten von Testspezifikationen sind diese aber nicht explizit definiert.

2. **Überschneidung von Teilsequenzen:** Teile unterschiedlicher Testsequenzen bilden in einem I/O-EFA zusammen einen Zustand. Diese Aspekte müssen zusammengeführt werden, um eine 1-zu-1 Darstellung der Testspezifikation als I/O-EFA zu vermeiden.

3. **Nichtdeterminismus von Testspezifikationen:** Fehlerhafte Spezifikationen könnten Mehrdeutigkeiten enthalten. In weiteren Schritten sind aber nur deterministische I/O-EFAs zur Berechnung einer Simulationsrelation zulässig.

4. **Interpretation der Partitionierung:** Die gegebenen Testspezifikationen beschreiben das Eingabe-/Ausgabeverhalten nur an den durch die Repräsentanten der Partitionierung definierten Grenzwerten. Für die dazwischenliegenden Bereiche muss auf Basis der Partition eine sinnvolle Interpretation des Verhaltens definiert werden.

5. **Unterspezifizierte Testspezifikationen:** Gegebene Spezifikationen werden in den meisten Fällen das Verhalten des zu testenden Systems nicht vollständig spezifizieren. Insofern kann auch das Verhalten, das durch den I/O-EFA dargestellt wird, nicht vollständig sein.

Für jeden Testschritt innerhalb einer Testsequenz wird eine Transition erstellt. Die Bedingung der Transition $g_e(d, u)$ wird als Konjunktion über alle Eingabevariablen beziehungsweise über deren durch die Partitionierung zugehörigem Repräsentanten konstruiert. Die Ausgabefunktion

wiederum gibt den entsprechenden Wert der ausgewählten Partitionierungen der Ausgangsvariablen wieder. Da keine internen Variablen gegeben sind, wird auch keine spezifische Updatefunktion $f_e(d, u)$ definiert.

Start- und Zielzustand sind abhängig vom Auftreten des Testschrittes innerhalb einer Testsequenz. Dabei ist der Startzustand des ersten Testschrittes immer der Initialzustand des Automaten s_0. Stellen die letzten beiden Testschritte einer Testsequenz denselben Testschritt dar (gleiche Auswahl der Repräsentaten für alle Variablen), so wird dieser Umstand als Schleife interpretiert und Start- und Zielzustand sind identisch. Dies ist eine notwendige Hilfsmaßnahme, um Schleifen im Automaten auf Basis von Testfällen herleiten zu können, da Testsequenzen explizit ein entsprechendes Verhalten nicht definieren (zu mindestens nicht in den typischen uns vorliegenden Testspezifikationsformaten). Die erwarteten Testfälle, um Schleifen eindeutig identifizieren zu können, entsprechen dabei dem Testabdeckungskriterium $C2_a$ [Lig09] mit genau zwei Schleifendurchläufen, wobei diese Durchläufe allerdings identisches Ausgabeverhalten vorweisen müssen, um als solche erkannt zu werden.

Die Transitionen haben jeweils den Zielzustand der Transition $e_{i-1} \in E$, die im letzten Testschritt $i-1$ der Testsequenz erstellt wurde, als Startzustand. Auf diese Weise können identische Transitionen (gleiche Bedingung, gleiche Ausgabefunktion, gleicher Startzustand) entstehen, die nur auf Grund ihrer Herkunft (unterschiedliche Testsequenz) in einem neuen Zielzustand münden würden. In Konsequenz würde ein nicht-deterministischer Automat gebildet werden, der für die gleiche Bedingung unterschiedliche Zielzustände erlaubt. Um dies zu vermeiden, muss jede neue Transition bezüglich ihres Startzustandes eine disjunkte Bedingung aufweisen, um hinzugefügt zu werden. Da generell alle Bedingungen $g_e(d, u)$ aus Konjunktionen über die Vertreter der zugehörigen Partitionen gewählt werden und die Testspezifikationen in jedem Testschritt für jede Eingangsvariable einen Vertreter auswählen müssen, sind die konstruierten Bedingungen entweder disjunkt oder identisch. Insofern können keine Transitionen ignoriert werden, die nur teilweise bezüglich einer gegebenen Transition übereinstimmen. Dieser Umstand erleichtert die Konstruktion des I/O-EFAs. Ist es gewünscht, die Belegung einzelner Variablen innerhalb der Testspezifikation offen zu lassen (da in diesem Kontext für die Beeinflussung der Variable unbedeutend), so kann in einem Transformationsschritt die Bedingung durch Hinzufügen der negierten Konjunktion der möglichen Belegungen der offenen Variablen durch andere nicht-disjunkte Bedingungen die geforderte Bedingung hergestellt werden. Wäre dies nicht möglich bzw. die resultierende Bedingung unerfüllbar, wäre wiederum die zugrundeliegende Testspezifikation nicht deterministisch.

Des Weiteren könnten Transitionen vorliegen, die zwar bezüglich ihrer Bedingung und ihres Startzustandes identisch sind, gleichzeitig aber unterschiedliche Ausgabefunktionen vorweisen. In Konsequenz würde dies aber bedeuten, dass die Testspezifikation nicht-deterministisch ist, da sie für dieselbe Eingabesequenz unterschiedliche Ausgaben definiert. Generell wird davon ausgegangen, dass die vorliegenden Testspezifikationen deterministisch sind. Ist dies nicht der Fall, so wird die Spezifikation durch das beschriebene Verfahren in einen deterministischen I/O-EFA überführt (erste Transition wird übernommen, weitere werden ignoriert). Dieser I/O-EFA ist dann aber gegenüber der Testspezifikation nicht mehr semantisch identisch.

In Abbildung 6.8 ist exemplarisch eine Transformation dargestellt. Im oberen Teil der Abbildung sind zwei Testsequenzen aus dem aufgeführten Beispiel der Variante *StepwisePush* dar-

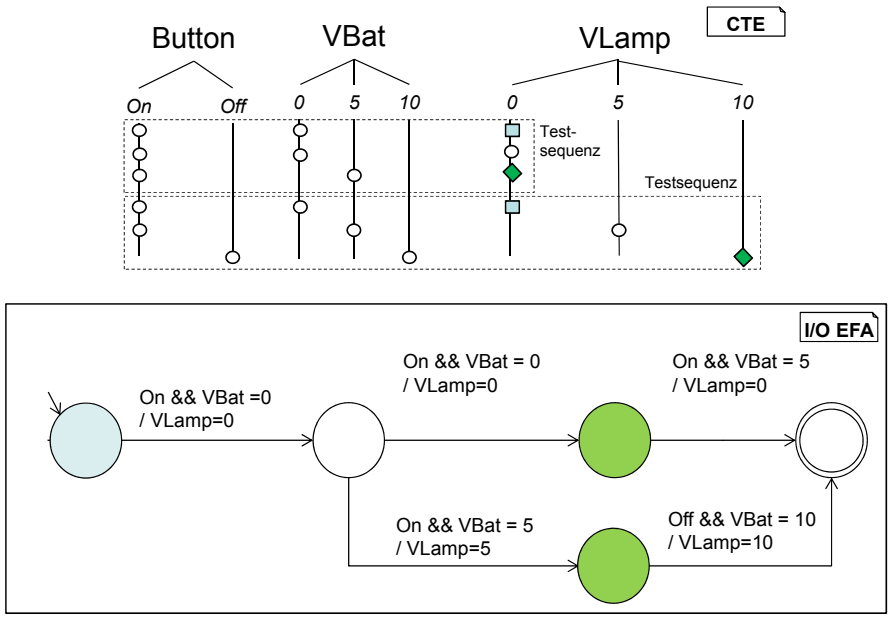


Abbildung 6.8: Beispielhafte Transformation von Testsequenzen zu I/O-EFA.

gestellt. Nacheinander überführt, würde der I/O-EFA entstehen, der im unteren Teil der Abbildung 6.8 dargestellt ist. Jegliche Sequenzen beginnen im Startzustand des I/O-EFAs. Da die Eingangsbedingungen für den ersten Testschritt der beiden Testsequenzen identisch sind, wird nur im ersten Schritt eine entsprechende Transition hinzugefügt. Danach sind weitere Bedingungen unterschiedlich und entsprechend separate Transitionen und Zustände werden hinzugefügt.

In diesem Schritt sind die Bedingungen der Transitionen nur auf die durch die Partitionierung vorliegende Repräsentanten beschränkt. Das Verhalten zwischen den durch die Repräsentanten feststehenden Grenzwerten ist allerdings nicht definiert. Während dies für unstetige Datentypen das beschriebene Verhalten exakt wiedergibt, ist es für stetige Datentypen notwendig, die gegebenen Lücken zu füllen. Durch Ersetzen der Bedingungen durch entsprechende Intervallabfragen kann stattdessen der gesamte Bereich einer Partitionierung abgedeckt werden. Dabei sind die jeweiligen Intervalle $I = i_1, .., i_n$ der Partition $P = p_1, ..p_n$ durch den Repräsentanten der zugehörigen Partition und den jeweiligen Nachfolger bestimmt: $i_x = [p_x, p_{(x+1)}], x \in 1..n-1$. Das abschließende Intervall beinhaltet weiterhin nur den Repräsentanten: $i_n = [p_n, p_n]$. Bei dieser Aufteilung ist die Interpretation der zugehörigen Ausgabefunktion offen. Die beiden naheliegendsten Optionen stellen die Beibehaltung der beschriebenen Konstante oder eine (lineare) Interpolation über die zugehörigen Repräsentanten der Ausgabefunktion dar.

In Abbildung 6.9 sind die entsprechenden Bedingungen des Automaten aus Abbildung 6.8 in Intervalle überführt und die zugehörige Ausgabefunktion durch eine Interpolation ersetzt.

Grundsätzlich kann davon ausgegangen werden, dass fehlerfreie Testspezifikationen zwar deterministisch, aber gleichzeitig meist auch unvollständig sind. Insofern ist nicht für jede mögliche Eingabesequenz ein Verhalten spezifiziert und damit der resultierende Automat unterspezifiziert. Um dies darzustellen, wird nach Abschluss der Transformation aller Testschritte in

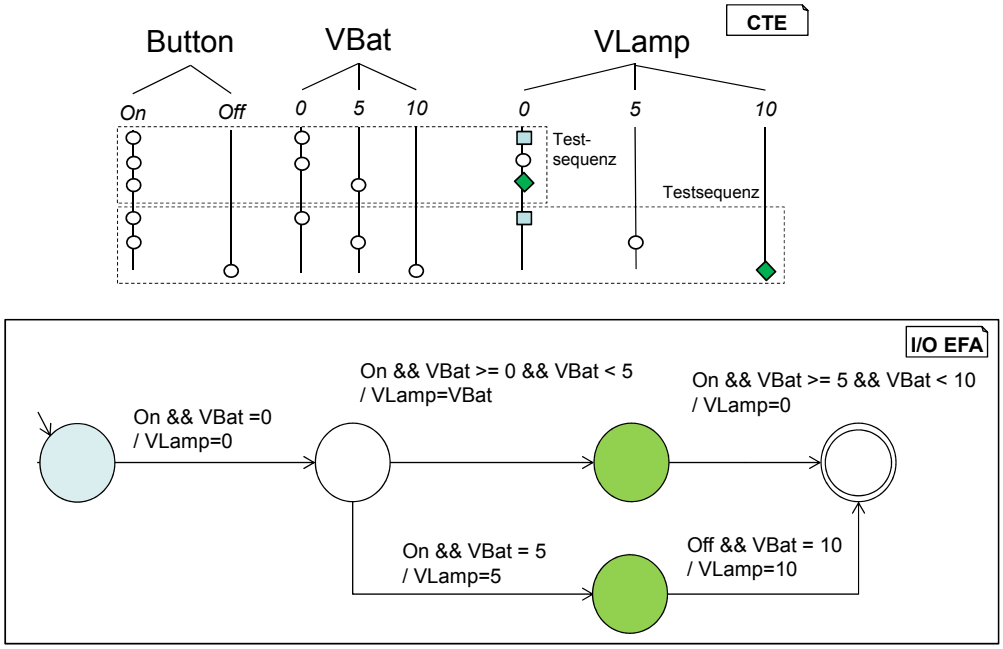


Abbildung 6.9: Beispielhafte Auflösung der Repräsentanten durch Intervalle innerhalb der Transitionsbedingungen.

Transitionen über alle Zustände iteriert und eine Negation der Disjunktion der Bedingungen der einzelnen Transitionen eines Zustandes gebildet. Diese Disjunktion bildet die Bedingung einer zusätzlichen Transition, die ausgehend vom aktuellen Zustand zum Zielzustand f_0 führt. Das zugehörige Verhalten ist undefiniert, weswegen die zugehörige Ausgabefunktion mit $f_e = \Phi$ beschrieben ist.

Definition 6.5 *Undefiniertes Verhalten. Um undefiniertes Verhalten in einem I/O-EFA zu beschreiben, wird die Ausgabefunktion $f_e = \Phi$, verwendet. Φ stellt dabei ein Sonderzeichen dar, das undefiniertes Verhalten beschreibt und für das bezüglich einer Simulationsrelation jegliches Verhalten akzeptabel ist.*

Φ wird im weiteren Verlauf bezüglich der Abstraktion des Verhaltens zur Bestimmung der Verhaltensähnlichkeit eine deutliche Rolle spielen. Bei der Transformation von Testspezifikationen zu I/O-EFAs dient es dazu, Unterspezifikationen im I/O-EFA explizit auszuweisen. Auf Basis der beschriebenen Transformation ist nun in jedem Zustand ein vollständiger Zustandsübergang beschrieben: Für jegliche Eingabekombinationen ist eine Transition definiert.

Insofern ist nach der Transformation garantiert, dass der resultierende Automat vollständig und deterministisch ist. Dies sind für die Bildung der Simulationsrelation zwei notwendige Bedingungen.

6.1.2 Testextraktion auf Basis von Simulinkmodellen

Der *Simulink Design Verifier*[1] ermöglicht für ein Simulinkmodell die Generierung von Inputstimuli, die einem konfigurierbaren Testabdeckungskriterium bis zu MC/DC (Modified Condition / Decision Coverage) genügen. Dabei ist es auch möglich, die maximale Schleifentiefe festzulegen. Werden die generierten Eingabestimuli anschließend auf das Modell angewendet und die zugehörigen Ausgaben gespeichert, so ist es möglich in einem Folgeschritt Testspezfikationen zu extrahieren, die den unter Abbildung 6.7 definierten Bedingungen genügen.

Auf diesem Wege können automatisiert formale Abstraktionen der durch die Simulink-Modelle definierten Funktionsvarianten gewonnen werden. Dabei ist allerdings zu beachten, dass sich Testabdeckungskriterien, wie zum Beispiel MC/DC, auch durch die Struktur des Modells stark beeinflussen lassen können [RWH08], so dass dieser Ansatz mit manuell definierten Tesfallspezifikationen nicht gleichzusetzen ist.

Die Integration von *Simulink Design Verifier* wurde von Herrn Esser in einer Bachelorarbeit [Ess16] umgesetzt und Details sind dort zu entnehmen.

6.1.3 Transformation von Simulinkmodellen zu I/O-EFAs

Die Transformation von Simulinkmodellen zu I/O-EFAs ist über eine Zwischentransformation zu Kontrollflussgraphen realisiert. Dies erlaubt Modellierungssprachen sowie Programmiersprachen über ein gemeinsames Zwischenformat in I/O-EFAs zu überführen und somit die Unterstützung weiterer Formate zu erleichtern.

Die Transformation eines Simulinkmodells in einen Kontrollflussgraphen („Control Flow Graph", CFG) basiert auf der Arbeit von Zhou und Kumar [ZK12]. Insofern wird dieser Teilschritt im Kontext dieser Arbeit nur kurz erläutert.

Grundsätzlich beschreibt der extrahierte CFG die schrittweise Ausführung der einzelnen atomaren Simulinkblöcke. Der Simulationszyklus von Simulink besteht dabei aus drei Schritten: Initialisierung, Berechnung der Ausgabe und Berechnung der internen Variablen. Während die Initialisierung nur einmalig durchgeführt wird, findet die Berechnung der Ausgabe und der internen Variablen in jedem Simulationsschritt auf Basis der zugeordneten Frequenz statt. Der Zeitschritt eines Simulationsschritts ist dabei durch die Simulink-Modellkonfiguration von außen vorgegeben und kann modifiziert werden. Die genaue Ausführungsreihenfolge der atomaren Blöcke in Simulink kann bei kompilierfähigen Modellen direkt von Simulink extrahiert werden (Simulink sorted order [Mat15]).

Bei einer entsprechenden Transformation der Modelle auf Basis der gegebenen Ausführungsreihenfolge in einen CFG werden zuerst alle Elemente entfernt, die keinen Einfluss auf die Ausführungsreihenfolge und das Verhalten nehmen. Anhand der in Abbildung 6.10 dargestellten Klassifizierung können in Simulink virtuelle Subsysteme (Kompositionen) und virtuelle Busse (Signalkompositionen) aufgelöst werden. Atomare Blöcke und nicht-virtuelle Blöcke müssen dagegen bei einer Transformation berücksichtigt werden. Dabei repräsentieren entsprechende konditionale Subsysteme mit einfacher oder mehrfacher Ausführung gängige Strukturen der Programmierung, wie If-Then-Else oder eine For-Schleife. Nicht-virtuelle Subsysteme werden

[1] `https://www.mathworks.com/products/sldesignverifier.html`

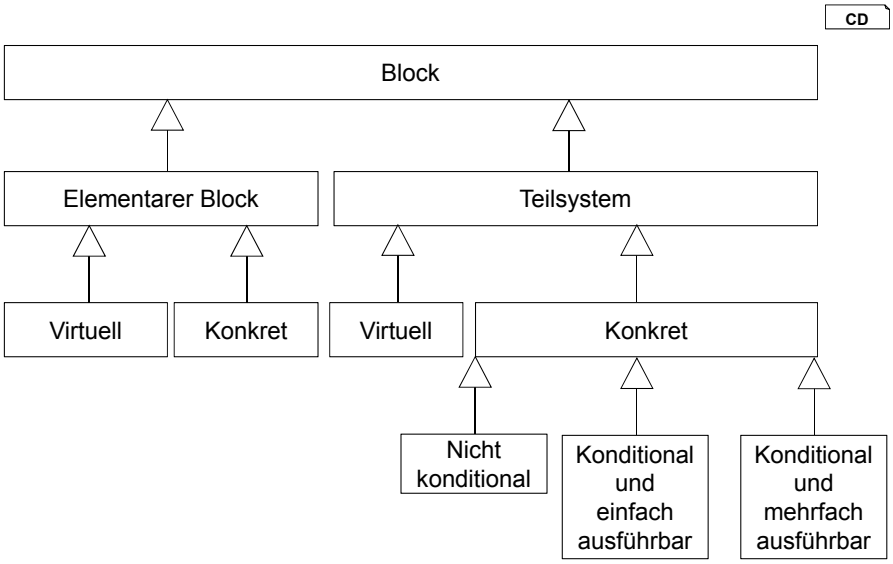


Abbildung 6.10: Klassifizierung von Simulinkblöcken [The14].

genutzt, um eine strikte Ausführungsreihenfolge zu erzwingen, virtuelle Subsysteme beeinflussen stattdessen die Ausführungsreihenfolge nicht.

Grundsätzlich ist es bei der Transformation in einen CFG für jeden atomaren Blocktyp nötig, eine eigene Transformationsregel bezüglich der Ausgabe- und Update-Funktion zu spezifizieren. Es ist nicht möglich diese Information direkt aus Simulink heraus zu gewinnen.

In Abbildung 6.11 ist der extrahierte CFG für das Simulinkmodell der Funktionsvariante *BasicPush* aus Abbildung 6.6 dargestellt.

Das beschrieben Verfahren überführt nur reine Simulink-Modelle, Stateflows werden nicht unterstützt. Eine entsprechende Transformation von Stateflows nach I/O EFAs ist zwar auch möglich [LK11], wurde aber im Kontext dieser Arbeit nicht weiter verfolgt.

Ist der CFG gegeben, so beschreibt dieser die notwendigen Berechnungsschritte innerhalb eines Simulationsschrittes - im Gegensatz zu einem I/O-EFA, der das Gesamtverhalten der Funktionsvariante und die möglichen Zustandswechsels im Lebenszyklus beschreibt. Insofern ist es in einem weiteren Schritt notwendig, den CFG in einen I/O-EFA zu überführen. Dazu wird in einem ersten Schritt eine Abhängigkeitsanalyse bezüglich der Variablen auf Basis der einzelnen atomaren Zuweisungsschritte durchgeführt, um das Zuweisungsverhalten bezüglich interner und Ausgangsvariablen zu extrahieren.

In der Tradition des Program Slicings [Tip95] werden alle Transitionen und deren zugehörige Bedingungen evaluiert, um zu identifizieren, welcher Berechnungsschritt direkt oder indirekt die entsprechende Variablen beeinflusst.

Dies wird erreicht, indem der CFG traversiert und schrittweise die offenen Variablen auf Basis vorheriger Zuweisungen substituiert werden. Dabei werden die zugehörigen Bedingungen der Zuweisung, die zur Substitution verwendet wurden, mit der gegebenen Bedingung der aktuellen Zuweisung konkateniert, wie in Abbildung 6.12 dargestellt. Die kursiv dargestellten Aus-

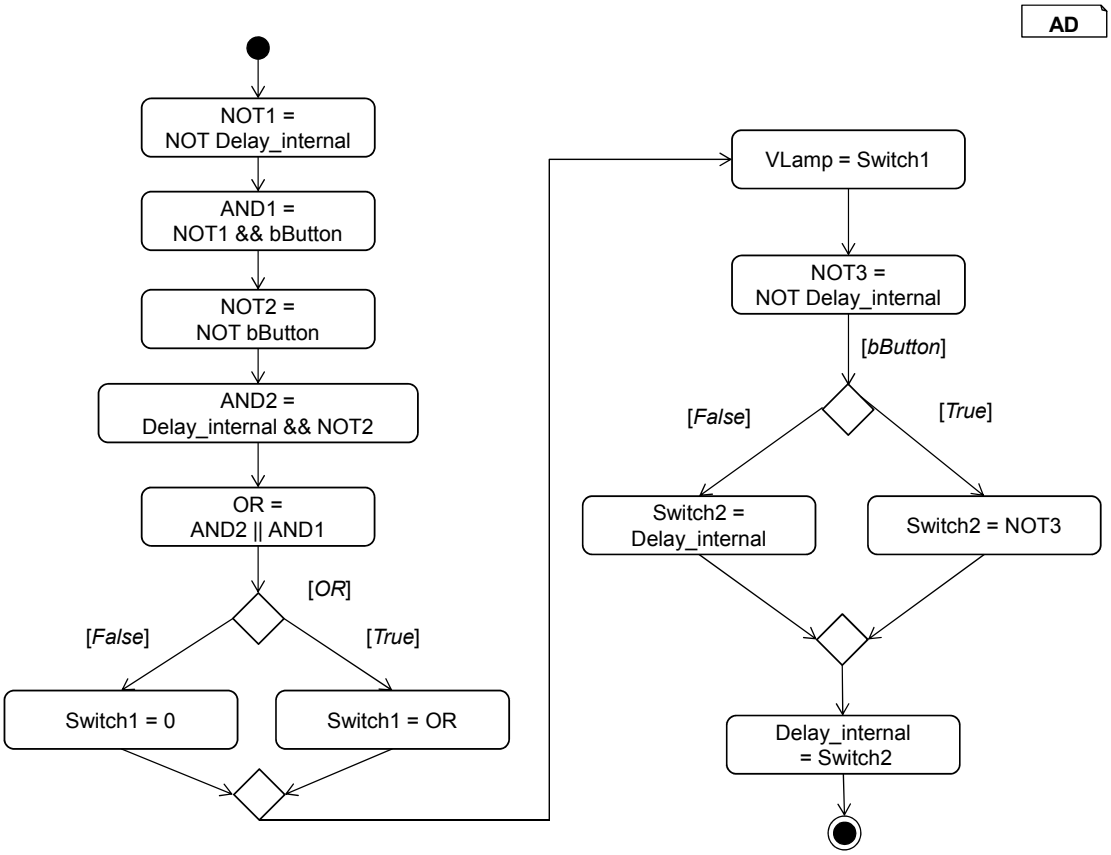


Abbildung 6.11: CFG des Simulinkmodells *BasicPush* aus Abbildung 6.6.

drücke sind dabei substituiert worden. Diese Vorgehensweise ähnelt stark der Symbolic Execution [PV09] und führt zum gleichen Problem: Auf Grund der notwendigen Konkatenation aller möglichen Paare von Bedingungen wächst die Komplexität exponentiell zur Anzahl der Verzweigungen (Pfadexplosionsproblem [CS13]). In Konsequenz ist eine Transformation nur für kleinere Verzweigungsgrade praktikabel - eine Transformation größerer Systeme ist insofern nicht möglich.

In Abbildung 6.13 ist die Transformation des CFG von Abbildung 6.6 der Funktionseinheit *BasicPush* zu einem I/O-EFA exemplarisch dargestellt. Der resultierende Automat besteht immer nur aus einem konkreten Zustand, der eigentliche Zustandsraum ist noch über interne Variablen beschrieben und wird erst bei der Transformation in ein I/O-TS in konkrete Zustände überführt. Dies ist als zweiter Schritt unterhalb des ersten Automaten in Abbildung 6.13 dargestellt. Dabei ist wiederum das Problem der Zustandsraumexplosion gegeben. Zusammenfassend ist also eine Transformation der Simulinkmodelle in I/O Transitionssysteme, die die Basis zur Bildung eine Simulationsrelation bilden, durch zwei Probleme innerhalb der exponentiellen Komplexitätsklasse (in Bezug auf Verzweigungstiefe und Anzahl Zustandsvariablen) beschrieben. In Konsequenz kann nur eine Teilmenge aller gegebenen Simulinkmodelle (mit geringem

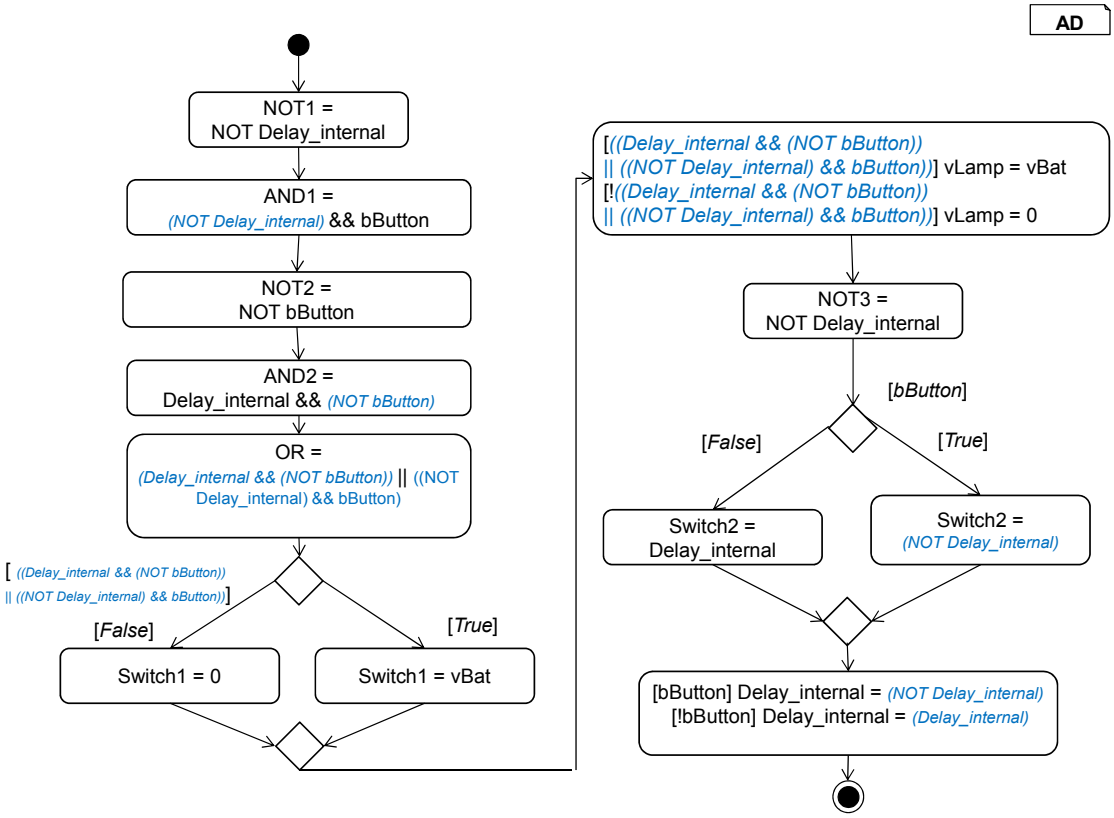


Abbildung 6.12: Substitution auf Basis des CFGs aus Abbildung 6.11 für *BasicPush*.

Verzweigungsgrad und kleinem Zustandsraum) in praktikabler Zeit überführt werden. Um die Komplexität gering zu halten, wird auf Basis der Substitution für jede Ausgangsvariable ein eigener Automat extrahiert und insofern werden nur notwendige Abhängigkeiten mit überführt.

6.1.4 Reduktion des Zustandsraumes

In Konsequenz sind weitere Hilfsmechanismen notwendig, um die Anwendbarkeit des Verfahren zu erhöhen. Wie schon in Abbildung 6.1 aufgeführt und in Unterabschnitt 6.1.2 im Detail erläutert kann unter Verwendung von Testspezifikationen (Schritt ⑥) oder durch eine automatisierte Testfallextraktion (Schritt ⑤) eine geeignete Approximation erreicht werden, um praktikable Zustandsräume zu erhalten.

Des Weiteren ist es aber auch nach der Transformation des CFGs in einen I/O-EFA und vor der Transformation in einen I/O-TS durch Abgleich der zu überprüfenden Automaten möglich, den potentiellen Zustandsraum zu reduzieren. Lässt sich eine Beziehung zwischen den internen Variablen der jeweiligen Automaten herstellen, so ist es nicht notwendig, den zugehörigen Zustandsraum aufzuspannen - das Zustandsraumexplosionsproblem wäre vermieden.

Die einfachste Möglichkeit besteht darin, entsprechende Zuweisungen der internen Variablen paarweise auf syntaktische oder in einem zweiten Schritt auf semantische Gleichheit (insofern

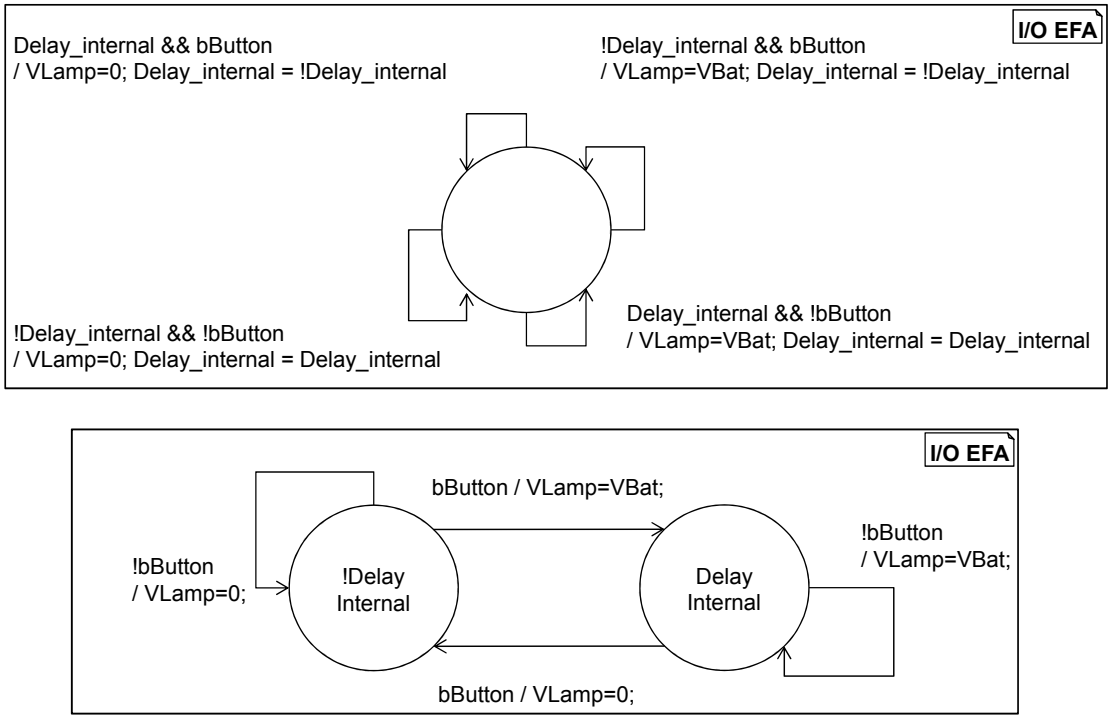


Abbildung 6.13: I/O-EFA auf Basis des substituierten CFG und anschließende Aufspannung des Zustandsraumes.

keine Abhängigkeiten zu weiteren internen Variablen gegeben sind) zu überprüfen. Ist dies der Fall und die Berechnung der internen Variable basiert nur auf sich selbst und eingehenden Variablen, so kann die entsprechende interne Variable durch eine Eingangsvariable ersetzt werden - die Berechnungen sind für jede Situation identisch. Zu beachten ist nur, dass der für die interne Variable definierte Wertebereich größer seien kann als die durch die Berechnungsfunktion wirklich berechneten Werte. In dem Fall, dass der Wertebereich eine echte Teilmenge des Wertebereiches der Variable darstellt, ist es möglich, dass durch die beschriebene Ersetzung nun unterschiedliches Verhalten identifiziert wird. Dieses kann aber in der Realität nicht auftreten, da die interne Variable auf Basis der Berechnungsfunktion diesen Wert niemals annehmen würde. Das gleiche Prinzip kann auch für Ausgangsvariablen genutzt werden. Sind diese syntaktisch identisch, so ist eine Kompatibilität bezüglich der Ausgangsvariable schon hergestellt. Hierbei besteht, im Gegensatz zu internen Variablen, nicht das Problem, dass ein Unterschied zwischen dem Wertebereich der Variablen und dem Wertebereich der Berechnungsfunktion zu Fehlinterpretationen führt, da in diesem Fall die Ausgabevariable nicht als Basis für weitere Berechnungen genutzt wird. Ist allerdings keine gemeinsame Entwicklungshistorie der zu vergleichenden Modelle gegeben, ist es unwahrscheinlich, dass die Berechnung der internen Variablen 1-zu-1 zuzuordnen wäre. Dieses Verfahren kann in angepasster Form nicht nur zur Reduktion des Zustandsraumes verwendet werden, sondern auch zur direkten Ähnlichkeitsanalyse auf Basis syntaktischer Analysen (ähnlich wie bei [GJS08, TG06, FFK08]). Da durch Analyse der Mengen

ähnlicher Datenelemente syntaktische Unterschiede bezüglich involvierter Eingabe- und Ausgabevariablen eliminiert werden können, kann bei hoher syntaktischer Ähnlichkeit einzelner Transitionen auf eine semantische Ähnlichkeit geschlossen werden. Diese Analyse kann wiederum mit deutlich geringerem Aufwand auf Basis der Levenstein-Distanz [Lev66] durchgeführt werden.

6.1.5 CEGAS (Counterexample Guided Abstraction Similarity)

In den vorherigen Abschnitten wurden die schon am Anfang des Kapitels in Abbildung 6.1 aufgeführten unterschiedlichen Wege zur Extraktion von I/O-EFAs auf Basis gegebener Entwicklungsartefakte im Detail erläutert. Dieser initiale Schritt überführt allerdings nur die gegebenen Informationen in das gewünschte Format, um semantische Ähnlichkeitsanalyse auf Basis einer iterativen Simulationsrelationsbildung durchzuführen. Wie in Abbildung 6.2 bereits kurz erläutert, wird versucht, zwischen zwei I/O-EFAs eine Simulationsrelation herzustellen. Dabei werden in einem ersten Schritt die jeweiligen I/O-EFAs in I/O-TSs überführt.

Diese beinhalten keine internen Variablen und die entsprechende Ausgabefunktion ist nicht für jede Transition, sondern für jeden Zustand definiert. Diese Transitionssysteme bilden den Ausgangspunkt für die Bildung einer Simulationsrelation.

Eine Simulationsrelation zwischen zwei I/O-TS A und B ist gegeben, falls jede Ausführungssequenz, die für A definiert ist auch in B definiert ist und für jede dieser Ausführungssequenzen dasselbe Ergebnis durch die Ausgangsvariablen zurückgegeben wird. Insofern kann B A simulieren, wenn eine Simulationsrelation zwischen A und B gegeben ist, da dann für jegliche Eingaben B dieselben Ausgaben produziert wie A. Dabei ist die Simulationsrelation nicht symmetrisch. Es kann sein, dass eine Simulationsrelation zwischen B und A nicht existiert [RSW+15].

Definition 6.6 *Zwischen zwei I/O TS $A = (S_A, s_{A0}, U, Y, E_A)$ und $B = (S_B, s_{B0}, U, Y, E_B)$ ist eine Simulationsrelation $R \subseteq S_A \times S_B$ gegeben, falls für alle $(a, b) \in R$ gilt, dass $y_a \equiv y_b$ (Ausgaben der Ausgabenfunktionen von I/O TS A und B) und für jede Transition $a \xrightarrow{[g_a(u)]} a' \in E_A$ in B für jede mögliche Eingabe eine entsprechende Transition $b \xrightarrow{[g_b(u)]} b' \in E_B$ existiert, so dass $(a', b') \in R$ [RSW+15].*

Existiert eine Eingabesequenz u, die zu unterschiedlichem Ausgabeverhalten führt, so kann keine Simulationsrelation gebildet werden. Stattdessen kann u als Zeuge für die Unterschiedlichkeit verwendet werden, da auf Basis der Eingabesequenz die zugehörige Transition $b \xrightarrow{[g_b(u)]} b' \in E_B$ identifiziert werden kann, die die Bildung einer Simulationsrelation verhindert.

Die detaillierte Ausarbeitung der Transformation der I/O-EFAs zu I/O-TSs und die Berechnung der Simulationsrelation unter Verwendung des SMT-Solvers Z3 [MB08, CGBD13] ist nicht Teil dieser Arbeit, sondern wurde im Kontext einer anderen Kompatibilitätsanalyse von Softwarekomponenten entwickelt [RSW+15]. Das Verfahren dient allerdings in angepasster Form als Grundlage für die gegenbeispielgesteuerte semantische Ähnlichkeitsanalyse. Dabei wird das Sonderzeichen Φ für Ausgabefunktionen eingeführt, um eine beliebige Ausgabe zu definieren.

Definition 6.7 $y_a(u, d) = \Phi \implies y_a \equiv y_b \forall e_b \in T_b$.

Ist nun die Bildung einer Simulationsrelation nicht möglich, so kann die Ausgabefunktion einer dafür verantwortlichen Transition des zu simulierenden I/O-EFA durch Φ ersetzt werden. Dadurch können Teile eines I/O-EFAs, die die Bildung einer Simulationsrelation verhindern, schrittweise abstrahiert werden, bis eine Relation gebildet werden kann.

Definition 6.8 *Seien A,B zwei I/O-EFAs und seien A', B' zwei durch eine Simulation abstrahierte I/O-EFAs, so ist der symmetrische Grad an Ähnlichkeit $(G_{CEGAS_n}(A,B))$ zwischen A und B wie folgt definiert:*

$$G_{CEGAS_n}(A,B) = \max_{i,j}\left(\frac{|M_{CEGAS}(A'_{n,a_i},B'_{n,b_j})|}{|M_{all}|}\right)$$

$M_{CEGAS}(A',B')$ ist die Menge aller Transitionen mit bestimmtem Verhalten (kein ϕ) von A' und B', während M_{all} alle Transitionen der unmodifizierten Automaten A und B enthält. Insofern ist die semantische Ähnlichkeit durch das Verhältnis unmodifizierter Transitionen, welche die Bildung einer Simulationsrelation ermöglichen, zu ursprünglich gegebenen Transitionen gegeben.

Die Anwendung der CEGAS ist in Abbildung 6.14 exemplarisch dargestellt. Um eine übersichtliche Darstellung zu ermöglichen, wird wiederum auf die Repräsentation von Abbildung 6.4 zurückgegriffen, die den Endzustand oder zusätzliche Transitionen zur Beschreibung von unterspezifiziertem Verhalten ausblendet. In diesem Beispiel kann *BasicHold LimitHold* simulieren, wenn die limitierende Transition (rot markiert im oberen rechten Automaten) abstrahiert wird, während *LimitHold BasicHold* simulieren kann, wenn die Ausgabefunktion bezüglich *VLamp* (rote Transition im unteren linken Automaten) abstrahiert wird. Wird die symmetrische *CEGAS* Metrik auf beide Automaten angewendet, ergibt sich eine semantische Ähnlichkeit von 60%.

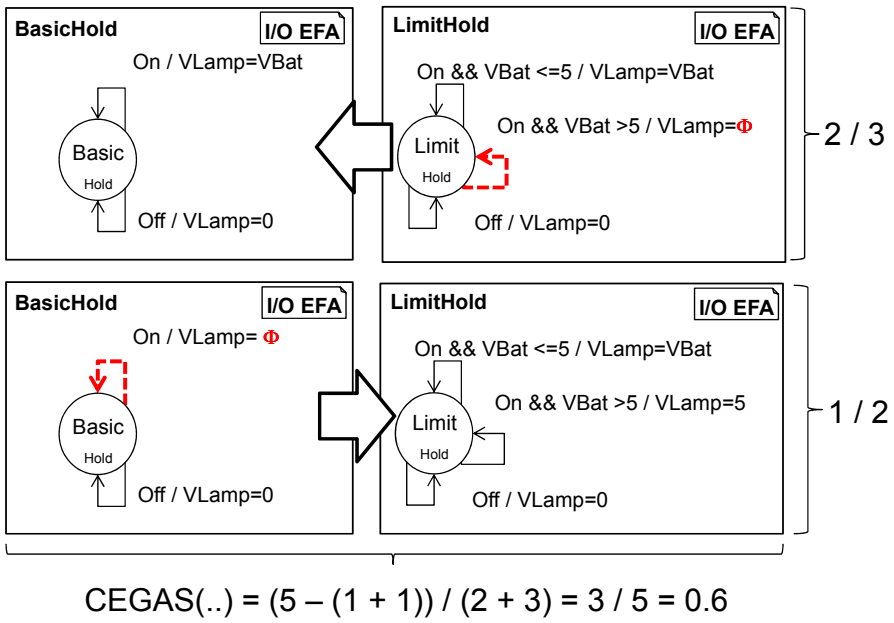


Abbildung 6.14: Anwendung der CEGAS auf *BasicHold* und *LimitHold*.

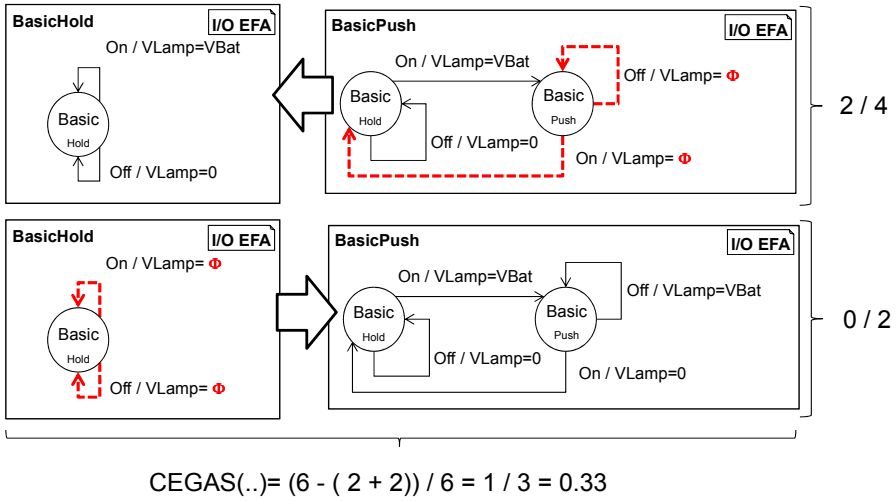


Abbildung 6.15: Anwendung der CEGAS auf *BasicHold* und *BasicPush*.

Während *CEGAS* in dieser Form gute Ergebnisse für zustandslose Funktionsvarianten liefert (insofern ein einziger Zustand), kann die Metrik für Funktionsvarianten, wie z.B. *BasicPush* oder *StepwisePush*, die mehrere Zustände beinhalten, ein fehlerhaftes Ergebnis berechnen. Dieses Problem trifft auf, wenn sich nur Teilgraphen des Zielautomaten simulieren lassen. Dieser Zusammenhang ist in Abbildung 6.15 genauer dargestellt, wo eine semantische Ähnlichkeit zwischen *BasicHold* und *BasicPush* berechnet wurde.

In diesem Beispiel kann eine Simulation von *BasicHold* durch *BasicPush* nur erreicht werden, indem der gesamte Automat abstrahiert wird. Dies führt zu einer Ähnlichkeit von 33%, während ein Ergebnis von 50% eher dem erwarteten Wert entsprechen würde.

Durch Anwendung einer n-begrenzten Simulationsrelation kann stattdessen identifiziert werden, dass eine 1-begrenzte symmetrische Simulationsrelation gegeben ist, aber eine 2-begrenzte wiederum nicht.

Das Beispiel in Abbildung 6.16, das den Vergleich zwischen *StepwisePush* und *BasicPush* beschreibt, stellt allerdings ein weiteres Problem dar, das sich auch durch Anwendung einer n-begrenzten Simulationsrelation nicht beheben lässt.

Der Zustand *BasicHold* ist sehr ähnlich zum Zustand *LimitHold* (siehe Abbildung 6.14: 60%) und die Zustände *BasicPush* sind gleich. Die CEGAS leitet aber nur einen Ähnlichkeitswert von 16% ab. Das Problem besteht darin, dass auch ein n-begrenzter Ansatz keine Ähnlichkeiten identifiziert, insofern nicht der gegebene Startzustand gewechselt werden kann (möglicher Startzustand *BasicPush* ergäbe eine 1-begrenzte Simulationsrelation).

Kombiniert man nun die Auswahl eines geeigneten Startzustandes, um anschließend eine n-begrenzte Simulationsrelation mit geringer Abstraktion zu identifizieren, so kann eine Ähnlichkeit von 44% abgeleitet werden, wie in Abbildung 6.17 dargestellt.

Beginnend am Zustand *BasicPush* kann eine 3-begrenzte Simulationsrelation mit deutlich weniger Abstraktionen hergestellt werden. Des Weiteren sind in Abbildung 6.17 alle Teilgraphen und deren Ähnlichkeiten bezogen auf unterschiedliche Begrenzungen (2 und 1) ausgewiesen.

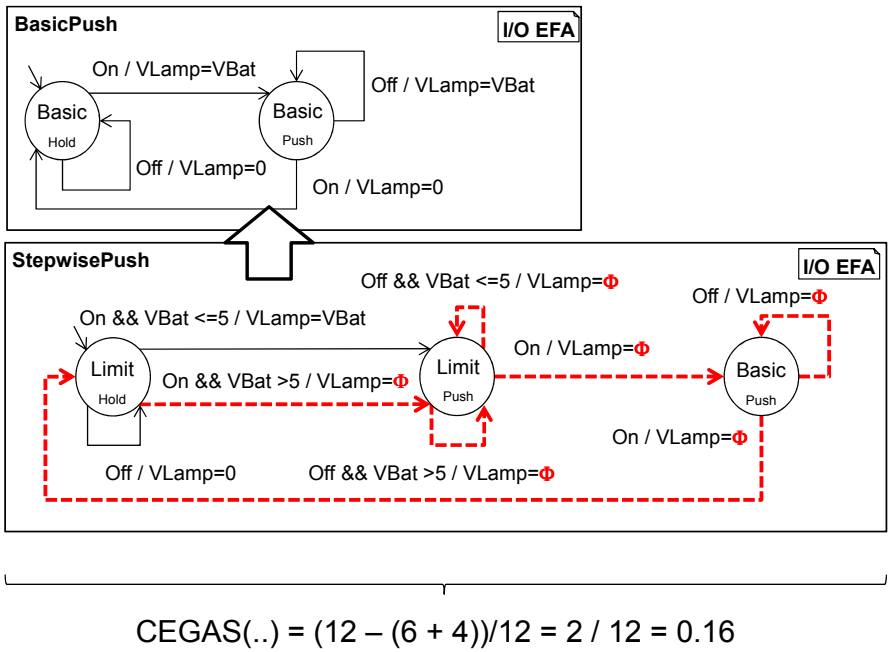


Abbildung 6.16: Anwendung der CEGAS auf *StepwisePush* und *BasicPush*.

Diese Kandidaten können mit in Betracht gezogen werden, wenn eine Funktion auf Basis einer semantischen Ähnlichkeitsanalyse extrahiert werden soll. Berücksichtigt man die beschriebenen Aspekte so ist eine n-begrenzte gegenbeispielgesteurte semantische Ähnlichkeitsanalyse wie folgt gegeben:

Definition 6.9 *Begrenzte gegenbeispielgesteurte semantische Ähnlichkeitsanalyse. Seien A, B zwei I/O-EFAs und A'_{n,a_i}, B'_{n,b_j} zwei abstrahierte I/O-EFAs basierend auf einer n-begrenzten Simulation beginnend bei den Zuständen a_i und b_j; dann ist der symmetrische Grad der Ähnlichkeit $G_{CEGAS_n}(A,B)$ zwischen A und B wie folgt gegeben:*

$$G_{CEGAS_n}(A,B) = \max_{i,j}\left(\frac{|M_{CEGAS}(A'_{n,a_i},B'_{n,b_j})|}{|M_{all}|}\right)$$

Die höchste Ähnlichkeit aller möglichen Startzustandskonstellationen zwischen A und B wird für eine Grenze n extrahiert, um die Ähnlichkeit abzuleiten. Diese Analyse unterstützt die Identifikation von semantischen Ähnlichkeiten von Teilgraphen und damit Teilen des Gesamtverhaltens zweier verschiedener Funktionsvarianten.

Anstatt die verschiedenen Grenzwerte separat voneinander zu bewerten und nur das Maximum zu identifizieren, können auch alle möglichen Teilgraphen unter Verwendung der Siebformel (Prinzip von Inklusion und Exklusion) [Ste07] in Betracht gezogen werden. Dabei werden alle überlappenden Teilgraphen betrachtet und ihre Ähnlichkeit entsprechend ihrer Größe in Relation zur Gesamtgröße der Automaten gewichtet. Die Berechnung aller möglichen Paare und deren Ähnlichkeitswerte stellt dabei aber einen deutlich zu großen Aufwand dar und erste

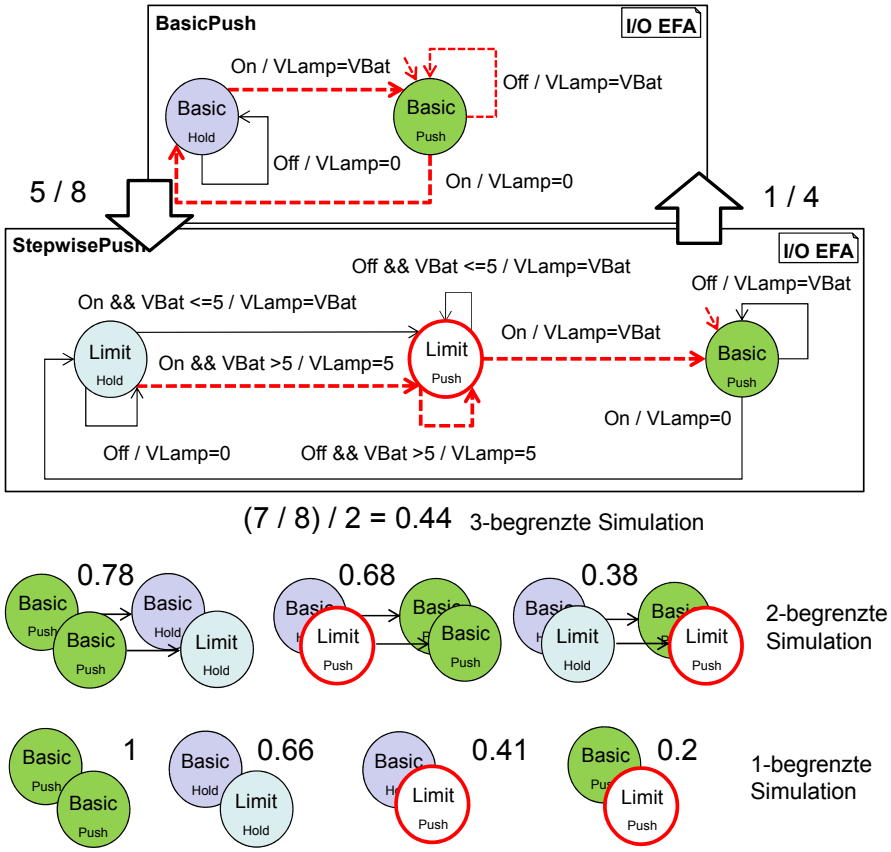


Abbildung 6.17: Anwendung der n-begrenzten CEGAS auf *StepwisePush* und *BasicPush*.

Ergebnisse, die probeweise berechnet wurden, ergaben keinen nennenswerten Mehrwert gegenüber der begrenzten gegenbeispielgesteurten semantischen Ähnlichkeitsanalyse. Insofern wurde dieser Ansatz nicht weiter verfolgt. Die Implementierung des Verfahrens wurde durch eine Masterarbeit auf Basis der MontiCore [GKR+08] Sprache MontiArcAutomaton [RRW14], einer Erweiterung der Sprache MontiArc [HKRR11] um I/O^{ω}-Automaten [Rum96], durchgeführt und ist [Thi15] zu entnehmen.

6.1.6 Unterschiedliche Schnittstellen

Beim Vergleich unterschiedlicher Funktionsvarianten kann es dazu kommen, dass durch eine schnittstellenbasierte Ähnlichkeitsanalyse Mengen ähnlicher Datenelemente bezüglich der Berechnung einer Ausgabe identifiziert werden können, eine Funktionsvariante aber mehr Eingaben zur Berechnung benötigt. Da in diesem Fall für einen Automaten für die zusätzliche Eingangsvariable keine Bedingungen definiert sind, ist der Automat im Kontext einer größeren Eingabemenge unterspezifiziert und in Konsequenz nicht-deterministisch. Um dieses Problem zu umgehen, ist es notwendig, durch einen weiteren Transformationsschritt die gegebenen

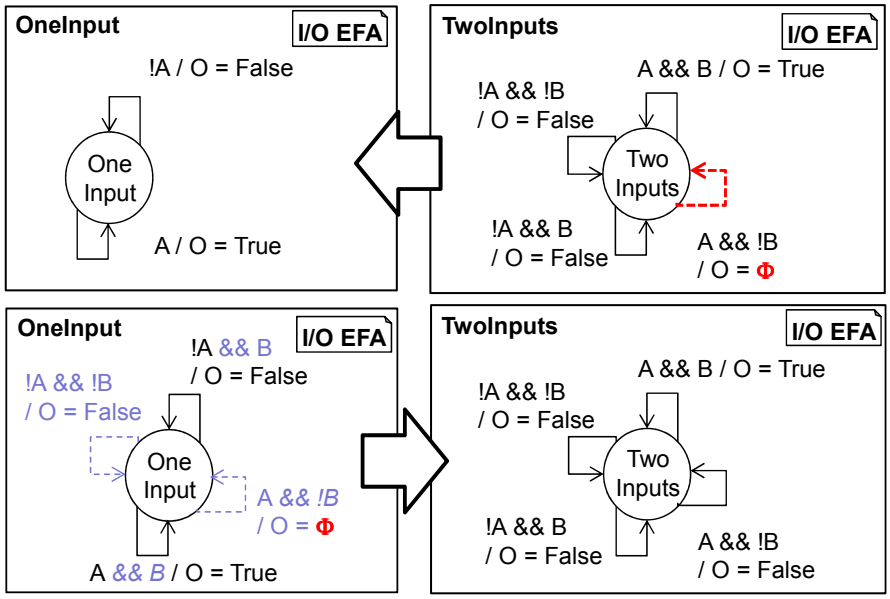


Abbildung 6.18: Umgang mit zusätzlichen Eingangsvariablen bei der CEGAS.

Transitionen durch zusätzliche Bedingungen zu erweitern und neue hinzuzufügen, wie in Abbildung 6.18 dargestellt. Fehlende Bedingungen und Transitionen bezüglich des Parameters *B* werden hinzugefügt, ohne dass das Ausgabeverhalten verändert wird.

Anschließend ist es möglich, die CEGAS anzuwenden und auch nur Teile der neu hinzugefügten Transitionen zu abstrahieren, so dass insgesamt eine genauere Ähnlichkeitsaussage möglich ist.

6.2 Evaluierung

In diesem Abschnitt werden die Ergebnisse der beschriebenen semantischen Ähnlichkeitsanalysen diskutiert. Dabei wird ein Fokus auf die Performanz als auch die Genauigkeit des Verfahrens in Relation zu einer manuellen Bewertung gelegt.

Die Evaluierung der CEGAS wurde auf drei vorselektierten Software-Komponenten durchgeführt, die sich zu diesem Zeitpunkt in der Entwicklung befanden. Die Vorauswahl wurde dabei auf Basis der folgenden Kriterien durchgeführt:

1. Mindestens ein extrinsisches Paar auf Basis der Evaluierung aus Unterabschnitt 5.1.2 ist für die entsprechende Software-Komponentenhülle gegeben.

2. Für mindestens ein extrinsisches Paar wurden Testspezifikationen durch unterschiedliche Personen im notwendigen Format spezifiziert.

3. Die Testspezifikationen sind nicht identisch.

4. Für mindestens ein extrinsisches Paar sind die Schnittstellenvarianten soweit ähnlich (siehe Abschnitt 5.2), dass für mindestens eine Ausgabevariable und die zur Berechnung notwendigen Eingabevariablen beider Software-Komponentenvarianten jeweils Repräsentanten der Mengen ähnlicher Datenelemente gebildet werden konnten.

5. Basierend auf zugehörigen Implementierungen beider Software-Komponentenvarianten kann eine semantische Ähnlichkeit in einem angemessenen Zeitrahmen manuell evaluiert werden.

6. Die identifizierte Ähnlichkeit sollte weder 0% noch 100% betragen.

7. Um die unterschiedlichen Aspekte des Verfahrens abdecken zu können, sollten alle evaluierten extrinischen Paare Repräsentanten aus folgenden Gruppen beinhalten: diskrete Ausgabe, kontinuierliche Ausgabe, zustandslos, zustandsbehaftet, gleiche Schnittstelle, zusätzlicher Port. Die Trennung zwischen diskreter und kontinuierlicher Ausgabe dient dabei zur getrennten Betrachtung interpolierter Intervalle.

Für jede der genannten Gruppen konnte ein extrinisches Paar identifiziert werden. Zwei Funktionsvarianten einer virtuellen Temperatursensor-Software-Komponente repräsentieren ein zustandsloses Verhalten unter Verwendung einer kontinuierlichen identischen Schnittstelle. Zwei Funktionen verschiedener virtueller Handbremsen beschreiben wiederum ein zustandsloses Verhalten, welches durch eine diskrete Schnittstelle realisiert wird. Des Weiteren unterscheiden sich beide Funktionen bezüglich ihrer Anzahl an Eingangssignalen, um das verglichenen Ausgangssignal zu berechnen.

Für komplexeres zustandsbehaftetes Verhalten konnte leider kein extrinsiches Paar identifiziert werden, das den genannten Ansprüchen genügt. In diesem Fall wurde stattdessen auf eine Kontroll-Software-Komponente zurückgegriffen, deren Evolution adäquate Informationen durch zwei verschiedene Versionen zur Verfügung stellt. Dieses Beispiel stellt auch einen weiteren Verwendungszweck von Ähnlichkeitsanalysen in den Vordergrund, genau wie auch schon in Unterabschnitt 5.2.2 für schnittstellenbasierte Analysen aufgeführt: Die Analyse der Evolution einer Software-Produktlinie.

Um das geistige Eigentum beteiligter Unternehmen zu schützen, kann im Folgenden leider nicht im Detail auf die analysierten Funktionsvarianten eingegangen werden.

Tabelle 6.19 listet die erwähnten extrinischen Paare (erste Spalte) und die Größe der Schnittstellenvarianten (fünfte und sechste Spalte) auf. Die vierte Spalte führt die Anzahl der durch die Testspezifikation gegebenen Testschritte für den auszuwertenden Ausgabewert auf. Der Grad der semantischen Ähnlichkeit in Spalte drei ist durch eine manuelle Analyse auf Basis gegebener Implementierungen (Simulinkmodelle) gegeben, ähnlich zu [BRR10a].

In einem ersten Schritt werden die Testspezifikationen für die einzelnen Funktionsvarianten der identifizierten Software-Komponentenvarianten in einen I/O-EFA transformiert. Die Dauer der Transformation ist in der letzten Spalte aus Tabelle 6.19 dargestellt. Die meiste Zeit wird dabei für die notwendigen SMT-Aufrufe zur korrekten Erstellung des I/O-EFAs benötigt.

In Tabelle 6.20 sind die Ergebnisse der CEGAS aufgeführt. Die Ergebnisse für die *Sensor*, die *Handbremse* und der *Controller* Software-Komponentenvarianten entsprechen ungefähr der manuellen Auswertung. Des Weiteren ist die Ausführungsdauer sowohl für die Transformation

Name	Ähnlichkeit	Testschritte	In	Out	Dauer (ms)
Sensor (Var. A)	33%	5	1	1	412
Sensor (Var. B)		3	1	1	268
Park Brake (Var. A)	75%	17	3	2	3930
Park Brake (Var. B)		5	2	2	651
Controller (Ver. 1)	90%	149	23	3	131702
Controller (Ver. 2)		165	24	3	167750

Tabelle 6.19: Software-Komponentenvarianten, die zur Evaluierung der CEGAS verwendet werden.

als auch die Analyse für eine Anwendung der unbegrenzten CEGAS Metrik im praktikablen Bereich. Die *Controller* Software-Komponentenvariante entspricht mit ihrer Schnittstellengröße (24 Eingänge, 3 Ausgänge) und der Menge an Testschritten (165 Testschritte) einer komplexeren Funktion (bei einer bezüglich Testbarkeit vernünftigen Funktionsarchitektur). Diese Evaluierung hat eher einen demonstrativen Charakter. Auf Grund des enormen Aufwands zur manuellen Evaluierung einer semantischen Analyse und der geringen Menge an potentiellen extrinischen Paaren konnte nur gezeigt werden, dass die CEGAS im Kontext dieser Beispiele korrekte Ergebnisse in praktikabler Zeit zur Verfügung stellt.

Der Nutzen der n-begrenzten CEGAS konnte in diesem Kontext nicht evaluiert werden, da die in Abbildung 6.14, Abbildung 6.15 und Abbildung 6.16 aufgeführten Problemstellungen nicht auftraten beziehungsweise keine Zwischenschritte hinzugefügt wurden.

Neben der Evaluierung der CEGAS durch direkten manuellen Abgleich der Quellen, wurden alle genannten semantischen Analysen auch im Sinne des in Kapitel 3 definierten Prozesses genutzt, um alle zur Verfügung gestellten Software-Komponentenvarianten und Funktionsvarianten bezüglich ihrer Ähnlichkeit zu bewerten. Innerhalb dieser Evaluierung stellte sich eine syntaktische Analyse extrahierter I/O-EFAs auf Basis von Simulinkmodellen, wie in Unterabschnitt 6.1.4 beschrieben, bezüglich Durchführbarkeit und Genauigkeit als am zuverlässigsten heraus. Dies hat folgende Gründe:

1. Das Problem einer Zustandsraumexplosion durch interne Variablen verhindert deutlich häufiger eine Durchführbarkeit als das Pfadexplosionsproblem während der Substitution.

2. Testfälle liegen oft nicht vor oder entsprechen nicht dem notwendigen Format.

3. Der *Simulink Design Verifier* kann bei Modellen, die einen größeren Zustandsraum beschreiben, nur Teile der für das gewünschte Abdeckungskriterium nötigen Testfälle ex-

Funktionseinheit	Dauer (ms)	CEGAS Metrik	Manuelle Bewertung
Sensor	4214	30 (0 / 60)%	33%
Park Brake	5283	75 (75 / 75) %	75%
Controller	315784	95,25 (92,5 / 98) %	90%

Tabelle 6.20: Ergebnisse der CEGAS (gerichtete Ähnlichkeit in Klammern).

trahieren. Die Dokumentation erwähnt dieses Problem, geht aber nicht im Detail auf die Ursachen ein. Des Weiteren beeinflusst die Struktur des Modells, wie schon in Unterabschnitt 6.1.2 beschrieben, maßgeblich die Form der extrahierten Testfälle, so dass nur bedingt von einer semantischen Analyse gesprochen werden kann.

4. Eine gegenbeispielgesteuerte, semantische Ähnlichkeitsanalyse abstrahiert auf Basis von Gegenbeispielen nach aktuellem Ansatz immer die gesamte Transition. Dies führt bei Varianten mit vielen Verzweigungen zu genauen Ergebnisse. Beschreiben die Varianten stattdessen Unterschiede in komplexen Rechnungen, die aber wenige oder keine Verzweigungen beinhalten, so ist das Verfahren sehr ungenau. Dies war bei vielen analysierten Varianten der Fall.

5. In den meisten Fällen sind die identifizierten Klone vom Typ 3, so dass ein Hybrid aus semantischer und syntaktischer Analyse gute Ergebnisse liefert.

Auch wenn ein syntaktischer Abgleich nach Transformation in einen I/O-EFA in der Anwendung auf das Portfolio der beteiligten Industriepartner das beste Ergebnis liefert, ist dieses Verfahren generell nicht in der Lage Typ 4 Klone zu identifizieren. Eine Anwendung auf die Beispiele aus Abbildung 6.5 und Abbildung 6.6 führt erwartungsgemäß zu keinem korrekten Ergebnis. Wird das Verfahren allerdings auf die Simulink-Modelle aus Abschnitt 4.2 angewendet und mit den Ergebnissen der Werkzeuge *SimDiff v4.5.1.0*, *dSpaceModelCompare v2.6* und *Simulink Report Generator v5.0* verglichen, so kann es - im Gegensatz zu den anderen Kandidaten - in allen Fällen außer dem semantischen Klon korrekte Aussagen treffen. Dies impliziert eine vorherige korrekte extrinsische und schnittstellenbasierte Ähnlichkeitsanalyse. Da Hierarchien vor der Auswertung aufgelöst werden und das Layout der Modelle sowie die Benennung interner Strukturen nicht betrachtet wird, ist dies auch nicht überraschend. Würde man allerdings das Verfahren um eine Normalisierung der Bedingungen und Ausgabefunktionen der Transitionen erweitern, wie in [ABSH11], wäre auch der semantische Klon aus Abschnitt 4.2 zu identifizieren.

Im Vergleich zu den Produkten *SimDiffv4.5.1.0*, *dSpaceModelCompare v2.6* und *Simulink Report Generator v5.0* stellt der beschriebene Prototyp zur semantischen Analyse bezüglich Performanz, Stabilität, Menge an unterstützten Simulinkblocktypen und Aufbereitung der Ergebnisse aufgrund seines prototypischen Charakters keine Konkurrenz dar. Eine Überführung in ein Produkt oder eine Integration in gegebene Produkte ist aufgrund der erhöhten Genauigkeit empfehlenswert.

Kapitel 7

Similarity Analysis Framework

Der in Kapitel 3 definierte Prozess zur produktgetriebenen Software-Produktlinienentwicklung hebt die Notwendigkeit zur Identifikation ähnlicher Varianten hervor. Auf Basis einer solchen Identifikation soll durch eine Extraktion schrittweise die Software-Plattform erweitert werden. Dabei wurde initial nicht näher auf die Art der Ähnlichkeitsanalysen und der zu untersuchenden Artefaktrollen eingegangen.

In Abschnitt 3.2 wurde wiederum die semi-automatisierte Analyse als Notwendigkeit für einen effizienten Prozess dargelegt und infolge die Potentiale einzelner Artefaktrollen für eine Ähnlichkeitsanalyse identifiziert. Als Konsequenz wurde abgeleitet, dass eine Ähnlichkeitsanalyse in drei aufeinanderfolgenden Schritten (extrinsisch, schnittstellen-basiert und semantisch) auf Basis der Architektur, der Schnittstellen, der Testfälle sowie der Verhaltensspezifikationen eine praktikable Analyse mit vielversprechenden Ergebnissen ermöglicht.

In Abschnitt 5.1, in Abschnitt 5.2 und in Kapitel 6 wurden wiederum einzelne Analyseverfahren und deren Ergebnisse in der Anwendung bei beteiligten Industriepartnern vorgestellt.

Unabhängig von der Qualität der Ergebnisse ist es aber weiterhin notwendig, ein Framework zu definieren, das es erlaubt, eine beliebige Menge an unterschiedlichen Ähnlichkeitsanalysen auf Basis einer vorgegebenen Struktur in vordefinierter Reihenfolge durchzuführen und die zugehörigen Ergebnisse so zu präsentieren, dass eine Identifikation potentieller Software-

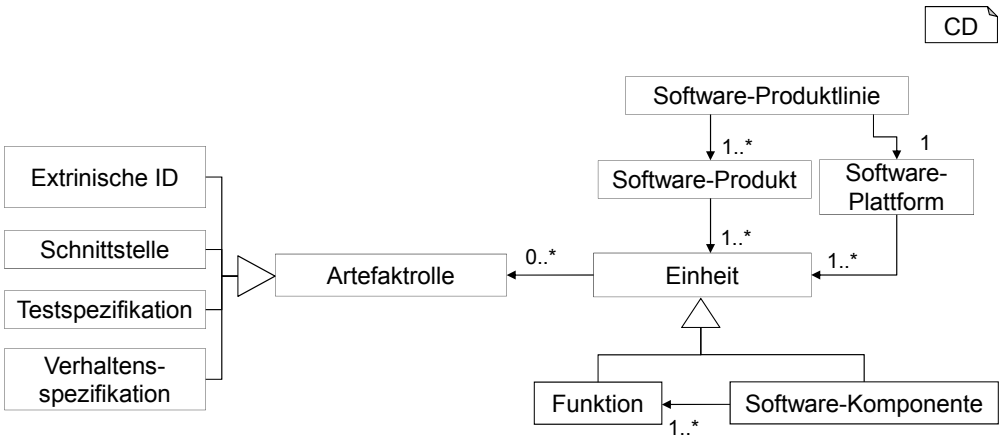


Abbildung 7.1: Durch das Framework SimA unterstützte Struktur.

Komponenten und Funktionen mit geringem Zeitaufwand möglich ist. Eine entsprechende Struktur ist in Abbildung 7.1 skizziert. Die gesamte betrachtete Entwicklung besteht dabei aus einer beliebigen Menge an Software-Produkten, die wiederum eine beliebige Menge an Software-Komponenten und Funktionen (beziehungsweise Software-Komponentenvarianten und Funktionsvarianten) beinhaltet. Jeder Software-Komponentenvariante oder Funktionvariante können Entwicklungsartefakte zugewiesen werden, auf deren Basis eine Ähnlichkeitsanalyse durchgeführt werden kann. Hierbei sind in Abbildung 7.1 als Beispiele die in Abschnitt 3.2 identifizierten potentiellen Artefaktrollen aufgeführt. In Zukunft können aber auch noch weitere Artefaktrollen und Artefakttypen unterstützt und dementsprechend hinzugefügt werden. Des Weiteren ist eine Produktlinie definiert, die wiederum aus einer Menge an Funktionen und Software-Komponenten mit zugehörigen Entwicklungsartefakten besteht. Dies erlaubt nicht nur die Analyse von Varianten einzelner Software-Produkte, sondern auch den Vergleich mit Domänenartefakten, was auch die Schritte 2 und 4 aus Abbildung 3.1 auf Seite 29 unterstützt.

Da die vorangegangene Analyse möglicher Klonerkennungsverfahren sich nur auf den aktuellen Stand der Technik beziehen kann und auch die Menge an Verfahren, die im Kontext einer Dissertation umgesetzt oder integriert werden kann, begrenzt ist, sollte das Framework zukünftige Anpassungen mit möglichst geringem Aufwand zulassen.

Auch die Verfahren zur Aufbereitung der Daten, sowie deren Darstellung müssen separat voneinander durchführbar und auswechselbar sein, um möglichen zukünftigen Ansprüchen zu genügen.

In den folgenden Abschnitten wird das Framework SimA (Similarity Analysis) vorgestellt, das die aufgeführten Punkte umsetzt und es ermöglicht, Schritt sechs aus Abbildung 3.1 effizient umzusetzen. Zuerst werden in Abschnitt 7.1 die Anforderungen für das Framework definiert, um dann in Unterabschnitt 7.2.1 auf die Grobarchitektur des Frameworks einzugehen. In Unterabschnitt 7.2.3 wird näher auf den logischen Ablauf des Frameworks und die Struktur der Resultate eingegangen, während in Abschnitt 7.4 die Implementierung und die Verwendung des Frameworks beschrieben werden.

7.1 Anforderungen an SimA

Die aufgeführten Überlegungen führen primär zu Anforderungen bezüglich der Flexibilität des zu entwickelnden Frameworks als auch bezüglich der Darstellung der Ergebnisse. Im Folgenden werden diese nun aufgeführt und als Basis für die Erläuterungen in den nächsten Abschnitten verwendet.

1. Anforderungen an das Framework
   a) **SimA1** *Beliebige Projektstrukturen*
      Das Framework muss beliebige Projektstrukturen unterstützen können, solange sich diese auf die in Abbildung 7.1 definierte Struktur abbilden lassen. Dabei müssen beliebige Artefaktrollen und zugehörige Artefakttypen unterstützt werden können.
   b) **SimA2** *Beliebige Ähnlichkeitsanalyse*
      Das Framework muss beliebige Ähnlichkeitsanalysen leicht in den gegebenen Work-

flow integrieren können. Dazu ist es notwendig unterschiedliche Technologien oder Programmiersprachen zu unterstützen, wie zum Beispiel Java oder Python.

c) **SimA3** *Kopplung von Ähnlichkeitsanalysen*
Um die Genauigkeit der Analysetypen zu verbessern, muss es möglich sein, Analyseverfahren zu koppeln und so auf Basis mehrerer Verfahren eine Ähnlichkeit berechnen zu können.

d) **SimA4** *Initialer Workflow*
Der initial festgelegte Workflow muss die angedachte Reihenfolge aus extrinsischer, schnittstellen-basierter und semantischer Analyse umsetzen.

e) **SimA5** *Modifizierbarer Workflow*
Der initial festgelegte Workflow muss leicht adaptierbar sein, um weitere Schritte hinzuzufügen oder den Ablauf modifizieren zu können.

f) **SimA6** *Vorverarbeitung*
Das Framework muss das möglichst einfache Hinzufügen beliebiger Vorverarbeitungsschritte unterstützen. Ein Beispiel für einen solchen Vorverarbeitungsschritt wäre das Generieren von Testfällen auf Basis eines Simulinkmodell, wie in Unterabschnitt 6.1.2 beschrieben.

g) **SimA7** *Leichte Modifikation der Ergebnisdarstellung*
Auf Basis eines leicht anpassbaren Workflows und der variablen Anzahl an Ähnlichkeitsanalysen, muss es möglich sein, die Darstellung der Ergebnisse auf Basis einer grundlegenden Struktur ebenso flexibel anpassen zu können.

h) **SimA8** *Generischer Bericht*
Die Generierung des Berichts auf Basis der ausgeführten Analysen muss möglichst losgelöst von den Analysen und dem Gesamtprozess realisiert werden, so dass beidseitige Anpassungen unabhängig voneinander durchgeführt werden können.

2. Anforderungen an die Darstellung der Ergebnisse

a) **SimA9** *Interaktiver Bericht*
Der resultierende Bericht muss eine interaktive Navigation durch die einzelnen Aspekte ermöglichen, um effizient Wiederverwendungspotentiale identifizieren zu können. Dabei muss es möglich sein, nach relevanten Daten zu filtern.

b) **SimA10** *Darstellung aller Einheiten und Software-Produkte*
Auf Basis der Analysen muss das Framework eine Gesamtübersicht über alle Einheiten und deren Auftreten in den analysierten Software-Produkten ermöglichen. Dies ermöglicht ein frühzeitiges Verständnis der Referenzarchitektur sowie möglicher Kollaborationen zwischen einzelnen Software-Produkten.

c) **SimA11** *Darstellung der Qualität der zugrundeliegenden Informationen*
Das Framework muss bei der Darstellung der Resultate auch auf den zugrundeliegenden Informationsgehalt eingehen und aufzeigen, in welchen Software-Produkten für welche Software-Komponenten und Funktionen welche Artefakte in welcher Qualität zur Verfügung stehen.

d) **SimA12** *Detailgrad der Ergebnisse*
Die Ergebnisse müssen sowohl in übersichtlicher abstrakter Form als auch in einem höchst möglichen Detailgrad zur Verfügung stehen. Es muss möglich sein, die Detailtiefe interaktiv nach Bedarf zu wechseln. Auf diese Weise ist es möglich Wiederverwendungspotentiale schnell zu identifizieren und anschließend im Detail zu verifizieren.

e) **SimA13** *Direkter Zugriff auf zugrundeliegende Artefakte*
Bei Betrachtung der Ergebnisse, muss es möglich sein direkt auf die zugrundeliegenden Artefakte zugreifen zu können, um die Resultate manuell verifizieren zu können.

f) **SimA14** *Darstellung der Ergebnisse*
Die Ergebnisse müssen sowohl in tabellarischer Form als auch durch Nutzung geeigneter Diagrammtypen dargestellt werden können. Dabei muss die Auswahl der verwendeten Spalten durch den Nutzer modifizierbar sein.

g) **SimA15** *Freier Zugriff auf die Ergebnisse*
Der Zugriff auf die Ergebnisse muss ohne Verwendung des SimA Frameworks auf Basis möglichst geringer technologischer Abhängigkeiten möglich sein.

7.2 Konzept

In diesem Abschnitt werden zuerst die Grobarchitektur des SimA Frameworks und anschließend deren Einbettung in die Arbeitsumgebung eines Industriepartners beschrieben. In einem zweiten Schritt wird auf den aktuell umgesetzten Arbeitsablauf und die daraus resultierenden Ergebnisse eingegangen. Abschließend wird die Struktur des generierten Berichtes vorgestellt.

7.2.1 Architektur

Die Anforderung **SimA1** bis Anforderung **SimA8** stellen einen hohen Anspruch an die Modifizierbarkeit der Architektur des Frameworks. Dabei ist es notwendig, auf unterschiedliche Projektstrukturen, unterschiedliche Ähnlichkeitsanalysen auf Basis verschiedener Technologien, angepasste Arbeitsprozesse, verschiedene Darstellungsformen und neue Vorverarbeitungsschritte kostengünstig eingehen zu können.

Die Architektur des Frameworks SimA, dargestellt in Abbildung 7.2, realisiert dies durch eine komponentenweise Trennung der genannten Aspekte.

Im Folgenden wird genauer auf die einzelnen Komponenten eingegangen [Muc16].

- **Workflow** Der *Workflow* definiert im Sinne des Entwurfsmusters *Template Method* [VHJG95] einen generellen Arbeitsprozess, der von einem *Client* gestartet werden kann. Dieser Workflow dient auch als *Fassade* [VHJG95] für die externe Ausführung. Die konkrete Ausführung des Arbeitsprozesses kann dabei durch Ableiten der abstrakten Basisklasse beliebig modifiziert werden (Anforderung **SimA5**). Als konkreter Workflow ist dabei durch SimA die schrittweise Durchführung von extrinsischer, schnittstellen-basierter und semantischer Analyse umgesetzt (Anforderung **SimA4**). Details dieses Arbeitsprozesses werden später in diesem Abschnitt noch erläutert. Die Komponente *Workflow* nutzt

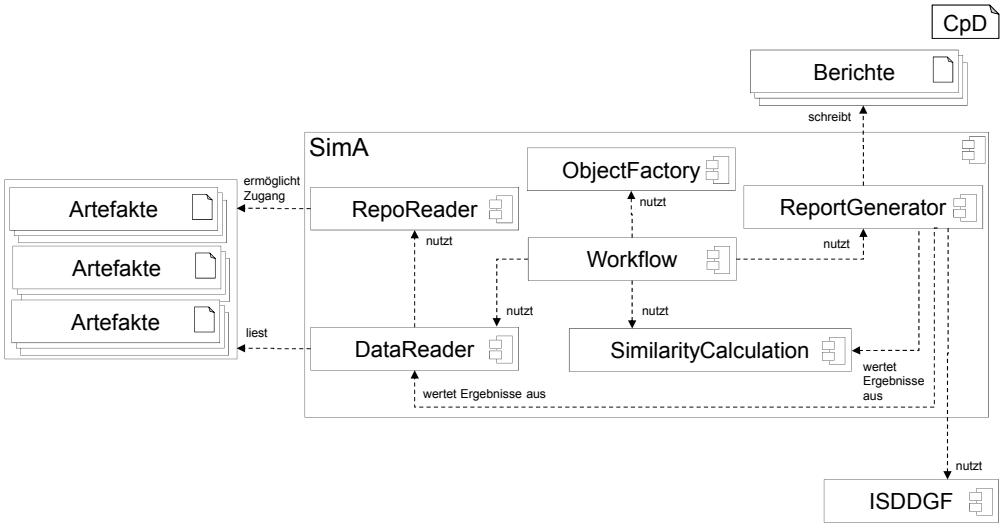


Abbildung 7.2: Komponenten des SimA Frameworks [Muc16].

bei der Durchführung die Komponenten *DataReader*, *ObjectFactory*, *SimilarityCalculation* und *ReportGenerator*, die im Folgenden näher erläutert werden.

- **DataReader** Der *DataReader* übernimmt die Aufgabe, die vorliegenden Artefakte zur Ähnlichkeitsanalyse einzulesen und notwendige Daten zu importieren und auf die in Abbildung 7.1 dargestellte interne Datenstruktur zu überführen. Der *DataReader* verwendet dafür den *RepoReader*, der den Zugang zu unterschiedlichen Datenverwaltungssystemen ermöglicht. Die Komponenten bietet dabei für unterschiedliche Artefakte abstrakte Basisklassen, die beim Wechsel der zugrundeliegenden Projektstruktur entsprechend des *Strategy* [VHJG95] Entwurfsmusters angepasst werden können (Anforderung **SimA1**). Des Weiteren unterstützt der *DataReader* für jeden dieser Klassen auch die Definition eines Vorverarbeitungsschrittes (Anforderung **SimA6**).

- **SimilarityCalculation** Diese Komponente verwaltet die einzelnen Ähnlichkeitsanalyseverfahren. Entsprechend des Entwurfsmusters *Strategy* stehen für die aktuell identifizierten Analysetypen extrinsisch, schnittstellen-basiert und semantisch jeweils abstrakte Basisklassen zur Verfügung, die auch die Struktur der Ergebnisse festlegen. Eine konkrete Implementierung muss jeweils die Einbindung eines zu unterstützenden Verfahrens zur Verfügung stellen und die Ergebnisse aufbereiten. Auf diesem Wege ist ein Austausch von Verfahren leicht möglich (Anforderung **SimA2**) und diese können auch gekoppelt werden (Anforderung **SimA3**).

- **ReportGenerator** Der *ReportGenerator* erstellt einen Bericht auf Basis der Informationen, die durch den *DataReader* und die *SimilarityCalculation* zur Verfügung gestellt werden. Die Struktur des Berichtes und die zugehörige Struktur dieser Komponente werden dabei später noch im Detail diskutiert. Nur bei Anpassungen der in Abbildung 7.1 de-

finierten Projektstruktur oder beim Hinzufügen neuer Ähnlichkeitsanalysetypen (neben extrinsisch, schnittstellen-basiert und semantisch) wäre die Anpassung dieser Komponente notwendig. Die generelle Logik zur Generierung des Berichtes ist in das Framework *ISDDGF* ausgelagert, da dieses auch im Kontext eines datenbankgestützten Varianten-, Modell- und Signalmanagements Verwendung findet (Details siehe Kapitel 8). Änderung an diesem Framework wären bei Anpassungen innerhalb von SimA nicht notwendig (Anforderung **SimA8**).

- **ObjectFactory** Die *ObjectFactory* übernimmt im Sinne des Entwurfsmusters *Factory* [VHJG95] die Instanziierung der notwendigen Objekte der einzelnen Komponenten. Dies ermöglicht eine flexible Konfiguration der einzelnen Teilaspekte, ohne dass der Arbeitsprozess angepasst werden muss. Die *ObjectFactory* bietet entsprechende Methoden zur Konstruktion für alle notwendigen Klassen der einzelnen Komponenten des SimA Frameworks, die von der Komponente *Workflow* genutzt werden. Diese können für eine spezifische Konstruktion überschrieben werden, um gewünschte abgeleitete Klassen der Basisschnittstellen der Komponenten zu verwenden (Anforderung **SimA5**, Anforderung **SimA8**, Anforderung **SimA2** und Anforderung **SimA1**).

7.2.2 Einbettung des SimA Frameworks in Unternehmenskontext

Die vorangegangene Beschreibung des SimA Frameworks erklärt nur dessen grundsätzlichen Aufbau ohne Verwendung spezifischer Ähnlichkeitsanalysen oder deren Einbettung in einen konkreten Entwicklungskontext.

Im Folgenden wird nun auf die notwendigen Schritte eingegangen, um das Framework in einen Entwicklungskontext einzubetten und spezifische Ähnlichkeitsanalysen zu verwenden. Dabei wird das Framework in den Entwicklungskontext eines beteiligten Industriepartners eingebettet und die in Abschnitt 5.1, Abschnitt 5.2 und Kapitel 6 definierten Ähnlichkeitsanalysen integriert, wie in Abbildung 7.3 dargestellt. Dabei sind folgende Änderungen notwendig:

- **Workflow** Der grundsätzliche Arbeitsprozess zur schrittweisen Durchführung von extrinsischer, schnittstellen-basierter und semantischer Analyse ist im Framework schon umgesetzt. Da die Komponente *Workflow* sich der *ObjectFactory* bedient, die separat festlegt, welche konkreten Klassen zur Verwendung des Workflows genutzt werden, sind an dieser Stelle keine Anpassungen notwendig, außer der konkreten Zuteilung einer *Factory*-Instanz.

- **RepoReader** Im Entwicklungskontext des Industriepartners werden alle Software-Produkte und die Software-Produktlinie durch die Versionsverwaltung *SVN* verwaltet und somit in separaten Repositories abgelegt. Aus Effizienz- und Speicherplatzgründen ist es dabei nicht sinnvoll, während der Ausführung des SimA Frameworks alle Dateien der jeweiligen Software-Produkte auszuchecken. Stattdessen unterstützt ein *SVNRepoReader* die Überprüfung, welche Entwicklungsartefakte jeweils zur Verfügung stehen und checkt nur diese bei Bedarf aus. Die entsprechenden Klassen wurden dem SimA Framework hinzugefügt, um in Zukunft andere Entwicklungskontexte unter Verwendung von SVN direkt unterstützen zu können.

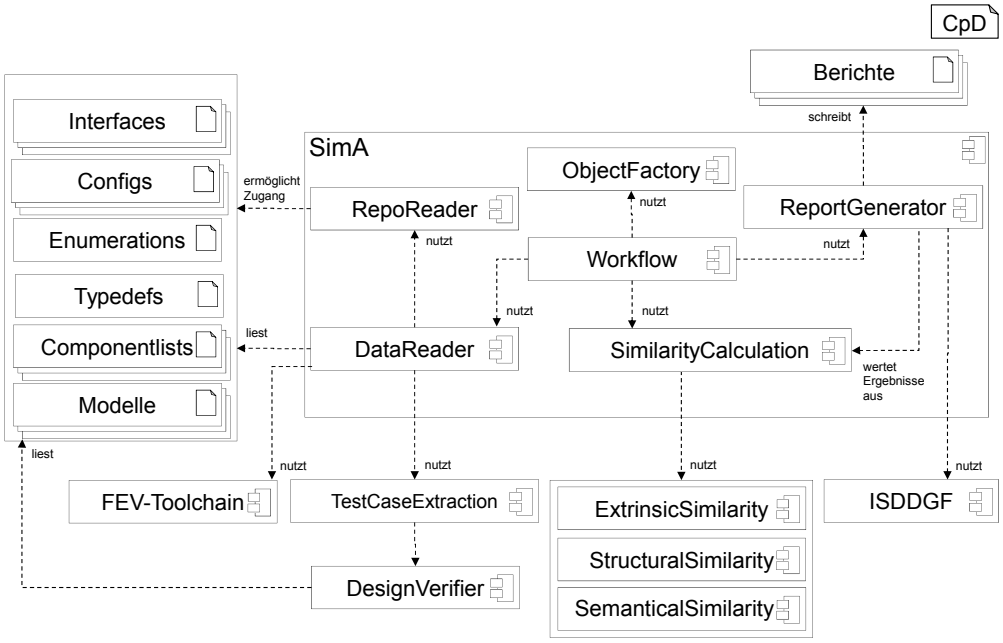


Abbildung 7.3: Komponenten des SimA Frameworks im Kontext des Industriepartners [Muc16].

- **DataReader** Über den *SVNRepoReader* erhält der *DataReader* direkten Zugriff auf die notwendigen Entwicklungsartefakte. Um dort dann aber aus den Artefakten die notwendigen Daten erheben zu können, ist wiederum eine spezifische Umsetzung notwendig. Die durch PERSIST zur Verfügung stehenden Entwicklungsartefakte (alle in Abbildung 7.3 dargestellten Artefakte außer der Modelle) sind alles Exceldokumente. Die gegebene Werkzeugkette bietet dabei eine größere Menge an Matlab-Skripten an, die es ermöglichen, die notwendigen Informationen zu erhalten. Insofern ist dort wiederum die Implementierung konkreter Klassen für die jeweiligen Artefakttypen notwendig. Diese stellen auf Grund der Verwendung der durch den Industriepartner gegebenen Werkzeugkette größtenteils reine Adapterklassen dar. Des Weiteren wird für die Verwendung von Testfallspezifikationen zur Ähnlichkeitsanalyse ein Vorverarbeitungsschritt definiert. Ist für eine identifizierte Einheit keine Testfallspezifikation, stattdessen aber ein Simulink-Modell gegeben, so wird die in Unterabschnitt 6.1.2 beschriebene Testfallextraktion auf Basis des *DesignVerifiers* genutzt, um Testfallspezifikationen zur Verfügung zu stellen.

- **SimilarityCalculation** Die Komponente bietet einzelne Strategien (extrinsisch, schnittstellen-basiert und semantisch) an, ohne dass das SimA Framework in seiner Grundausstattung eine Implementierung dieser Strategien vorweist. Um nun die in Abschnitt 5.1, Abschnitt 5.2 und Kapitel 6 beschriebenen Ähnlichkeitsanalysen verwenden zu können, müssen diese eingebunden werden. In einem ersten Schritt ist es notwendig, die Eingabedaten der Entwicklungsartefakte in das durch die Analyse vorgeschriebene Format zu transformieren. Da die Analysen direkt im Kontext der beteiligten Industriepartner ent-

wickelt wurden, existieren dabei schon entsprechende Transformatoren. Nach Abschluss der Analyse müssen auch die Ergebnisse in das durch die *SimilarityCalculation* vorgegebene Format transformiert werden. Dafür müssen weitere Transformatoren implementiert werden. Im Falle einer Sprachinkompatibilität (zum Beispiel Java und Python) ist es weiterhin notwendig, diese Übersetzung auf Basis eines zu definierenden Austauschformats (zum Beispiel JSON) durchzuführen.

- **ObjectFactory** Die neu definierten Klassen, die zur Einbettung in den gegebenen Kontext und zur Integration der Ähnlichkeitsanalysen implementiert wurden, müssen durch eine neue *Factory* instanziiert werden. Diese *Factory* wird dann dem Workflow zugeordnet.

7.2.3 Ablauf des SimA Frameworks

Der initial durch das SimA Framework beschriebene Arbeitsprozess ist in Abbildung 7.4 dargestellt. Dabei wird auch angezeigt, welche Informationen ab welcher Aktivität zur Verfügung stehen und als entsprechender Bericht ausgeleitet werden können. In der konkreten Realisierung werden aber zuerst alle Daten gesammelt und verarbeitet, bevor ein entsprechender Bericht unter Verwendung des *ISDDGF* Frameworks generiert wird. In einem ersten Schritt werden über den *DataReader* unter Verwendung des *RepoReader*s alle Software-Produkte und zugehörigen Einheiten identifiziert. Anschließend werden für alle Einheiten die zur Verfügung stehenden Entwicklungsartefakte identifiziert. Dabei werden auch, wie in Unterabschnitt 7.2.2 erläutert,

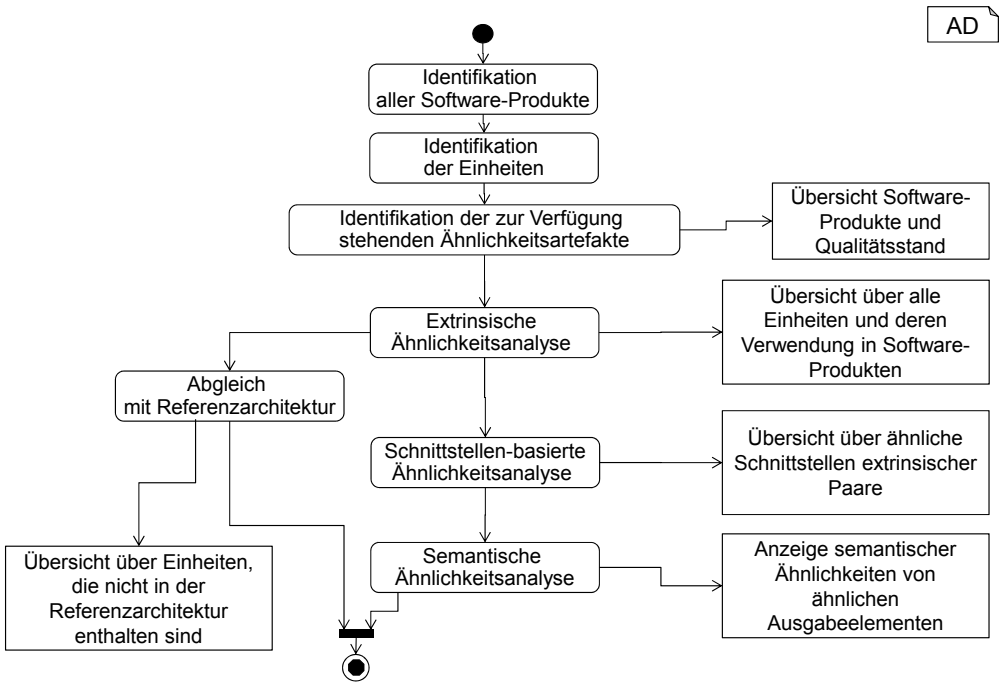


Abbildung 7.4: Grober Workflow des SimA Frameworks und resultierende Berichte.

bei Bedarf Vorverarbeitungsschritte durchgeführt, wie die Testfallgenerierung auf Basis von Simulink-Modellen.

Nach diesem Schritt ist es bereits möglich, eine erste Übersicht über alle zur Verfügung stehenden Software-Produkte und deren Qualitätsstand hinsichtlich einer Ähnlichkeitsanalyse auszuleiten. Der Qualitätsstand bezieht sich dabei auf die Menge der zur Verfügung stehenden Entwicklungsartefakte.

Anschließend wird die extrinsische Ähnlichkeitsanalyse durchgeführt. Auf Basis dieser kann direkt eine Gesamtübersicht über alle Einheiten und deren Verwendung in den verschiedenen Software-Produkten sowie eine Übersicht des Wiederverwendungspotentials auf Basis der extrinsischen Ähnlichkeitsanalyse extrahiert werden, wie in Abschnitt 5.1 aufgeführt.

Auf Basis der extrinsischen Analyse ist es dann in einem parallelen Prozessschritt möglich einen Abgleich mit der Referenzarchitektur durchzuführen, um alle Einheiten zu identifizieren, die aktuell nicht Teil der Referenzarchitektur sind und somit keine extrinische ID besitzen. Dies unterstützt den Arbeitsschritt 4 (Reevaluierung der Zuweisung) aus Abbildung 3.1 auf Seite 29.

Des Weiteren wird auf Basis der extrinsischen Ähnlichkeitsanalyse die schnittstellen-basierte Ähnlichkeitsanalyse für alle extrinsischen Paare durchgeführt. Als Resultat ist es wiederum möglich, alle Schnittstellenvarianten extrinsischer Paare separat darzustellen.

Für jedes ausgehende ähnliche Elementpaar mit entsprechenden eingehenden ähnlichen Elementpaaren kann abschließend eine semantische Ähnlichkeitsanalyse durchgeführt und auf dieser Basis eine semantische Ähnlichkeit dargestellt werden.

Falls in Zukunft weitere Ähnlichkeitsanalysetypen verwendet werden sollen, können die Komponenten *Workflow* (Integration in den Arbeitsprozess), *DataReader* (Identifikation des zugehörigen Entwicklungsartefaktes), *SimilarityCalculation* (Definition des Ähnlichkeitsanalysetyps) und *ReportGenerator* (Darstellung des Ergebnisses) entsprechend angepasst werden, um diese zu unterstützen.

7.2.4 Darstellung der Ergebnisse

Der in Abbildung 7.4 dargestellte Arbeitsablauf skizziert die nach Durchführung zur Verfügung stehenden Informationen. Die Anforderungen Anforderung **SimA9** bis Anforderung **SimA15** aus Abschnitt 7.1 stellen dabei hohe Ansprüche an die Zugänglichkeit der Ergebnisse. Dies bezieht sich sowohl auf die Struktur der Ergebnisse als auch auf die technischen Hürden zur Darstellung dieser.

Im Folgenden wird näher auf den Aufbau der Ergebnisse eingegangen, der eine effiziente Navigation durch die Ergebnisse der Ähnlichkeitsanalysen ermöglicht. Anschließend wird in Abschnitt 7.3 auf das Framework eingegangen, das die Generierung von HTML-Seiten auf Basis von Python unter Verwendung von *Bootstrap* und *NVD3* ermöglicht. In Abschnitt 7.4 wird anschließend noch detaillierter auf die Ergebnisse und den Umgang mit den generierten Berichten eingegangen.

Die Struktur des generierten Berichtes ist dabei wie in Abbildung 7.5 aufgebaut.

Als initiale Zugangspunkte existieren zwei Berichte: die Übersicht über alle Software-Produkte und deren Qualitätsstand sowie ein Link zur Referenzarchitektur (Anforderung **SimA10**, Anforderung **SimA11**). Diese Informationen werden aus den Schritten *Identifikation aller Projekte*,

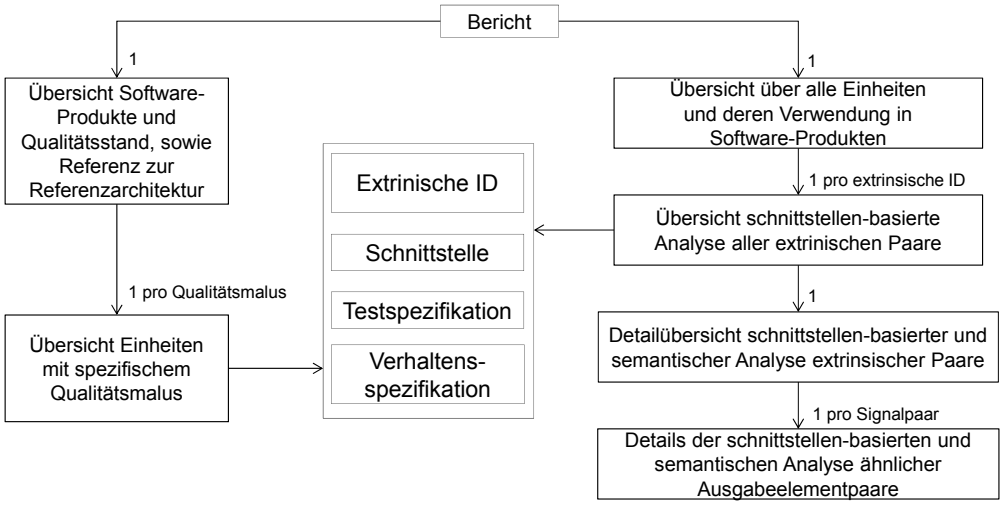


Abbildung 7.5: Navigationsbaum des generierten Berichts.

Identifikation aller Einheiten, *Identifikation der zur Verfügung stehenden Entwicklungsartefakte* und *Abgleich mit Referenzarchitektur* aus Abbildung 7.4 gewonnen.

Des Weiteren ist es möglich, bezogen auf einzelne Qualitätsmali (zum Beispiel fehlende Schnittstelle oder nicht Teil der Referenzarchitektur), sich eine jeweils separate Übersicht anzeigen zu lassen. Die Navigation findet dabei immer direkt durch Auswahl des entsprechenden Qualitätsmali des Software-Produktes in der Gesamtübersicht statt. Bei konkreter Anzeige der jeweiligen Einheiten ist es dann auch direkt möglich auf die vorhandenen Entwicklungsartefakte sowie auch die Verzeichnisse zuzugreifen, in denen Entwicklungsartefakte vermisst werden (Anforderung **SimA9**, Anforderung **SimA13**, Anforderung **SimA12**).

Der zweite Teil des Berichtes erlaubt sowohl eine Übersicht über alle identifizierten extrinischen IDs und deren Auftreten in den Software-Produkten (Anforderung **SimA10**) als auch Übersichtswerte über gegebene extrinsische, schnittstellen-basierte und semantische Ähnlichkeiten (Anforderung **SimA12**). Auch werden die Ergebnisse der extrinsischen Analyse zusammenfassend grafisch dargestellt, wie auch schon in Unterabschnitt 5.1.2 aufgeführt (Anforderung **SimA14**). Ausgehend von dieser Übersicht ist es sowohl möglich direkt auf das Basisverzeichnis der zugehörigen Einheit im jeweiligen Software-Produkt als auch auf die Übersichtsberichte der schnittstellen-basierten Analyse zuzugreifen (Anforderung **SimA9**).

Die Übersicht über die schnittstellen-basierte Analyse aller extrinsischen Paare steht jeweils für jede extrinsische ID mit mindestens einem extrinsischen Paar zur Verfügung und stellt die Resultate mittels tabellarischer und grafischer Übersichten dar (Anforderung **SimA14**). In dieser Übersicht sind wiederum auch für die jeweiligen Einheiten die zur Verfügung stehenden Entwicklungsartefakte aufgelistet (Anforderung **SimA13**).

Ausgehend von dieser Ansicht kann direkt auf eine detaillierte schnittstellen-basierte und semantische Analyse gewechselt werden (Anforderung **SimA9**, Anforderung **SimA12**). Diese stellt die ähnlichen Signale, Parameter, Messpunkte und Konstanten der Schnittstellen dar.

Von dieser Ansicht aus kann durch Auswahl der entsprechenden Elementpaare direkt auf eine Darstellung der Ähnlichkeiten dieser Elemente bezogen auf ihre Attribute gewechselt werden. Sind die notwendigen Bedingungen für eine semantische Analyse erfüllt, so sind dort auch entsprechende Details aufgezeigt (Anforderung **SimA9**, Anforderung **SimA12**).

Diese Struktur bildet alle durch den initialen Arbeitsprozess des Frameworks SimA abgeleiteten Informationen ab und bereitet diese in einer Form auf, um die Anforderungen Anforderung **SimA9** bis Anforderung **SimA15** zu erfüllen.

In Abschnitt 7.4 wird der Umgang mit diesem Bericht anschaulich an Beispielen erläutert.

Im Folgenden wird aber zuerst auf das Framework *ISDDGF* eingegangen, das die Generierung des Berichtes ermöglicht.

7.3 Interactive Software Design Document Generation Framework

Das Framework *Interactive Software Design Document Generation Framework* (*ISDDGF*) erlaubt die Generierung von HTML-Webseiten mittels Python. Basierend auf dem Ansatz eines statischen Seitengenerators verwendet das Framework eine Template Engine um die Generierung von HMTL-Seiten durchzuführen. Dabei unterstützt ein Paket an CSS-Dateien die Darstellungsqualität der generierten Seiten.

Da teilweise auch dynamische Inhalte dargestellt werden sollen (Filtern, Sortieren, angepasste Darstellung) ist des Weiteren eine Unterstützung von entsprechenden dynamischen Inhalten notwendig.

Im Kontext dieser Disseration existieren zwei unterschiedliche Aspekte, die aus unterschiedlichen Gründen den Bedarf der Generierung von interaktiven Berichten mit möglichst geringen technischen Abhängigkeiten aufweisen: Auf der einen Seite das aktuell besprochene Framework *SimA* zur Ähnlichkeitsanalyse und auf der anderen Seite Anpassungen an der Software SYNECT, die im Kontext des Industriepartners zur Definition und Wartung der Softwareproduktlinie genutzt werden soll. SYNECT erlaubt dabei ein datenbankgestütztes Varianten-, Modell und Signalmanagement. Weitere Details zu SYNECT werden in Kapitel 8 erläutert. Dort wird auch der weitere Bedarf nach einem Framework wie *ISDDGF* deutlich. Dabei sind die zugrundeliegenden Anforderungen an ein solches Framework sehr ähnlich zu den Anforderungen aus Abschnitt 7.1 und es besteht nicht die Notwendigkeit, zum Verständnis des Frameworks vorab Kapitel 8 zu lesen. Da aber ein generelles Verständnis des Frameworks notwendig ist, um die detaillierten Erläuterungen zum Bericht des *SimA* Frameworks zu verstehen, wird schon an diesem Punkt näher auf *ISDDGF* eingegangen.

Der Aufbau des Frameworks *ISDDGF* ist in Abbildung 7.6 dargestellt. Im Folgenden werden die einzelnen Komponenten näher erläutert [Kha16]:

- **UI Component** Die Komponente *UI Component* stellt die Schnittstelle zur Verfügung auf deren Basis der Entwickler einhzelne Aspekte eines HTML-Bericht definieren kann. Sie besteht hauptsächlich aus drei Unterkomponenten, die die tabellarische Darstellung (*UI Table*), die grafische Darstellung (*UI Chart*) sowie grundsätzliche Steuerungselemente

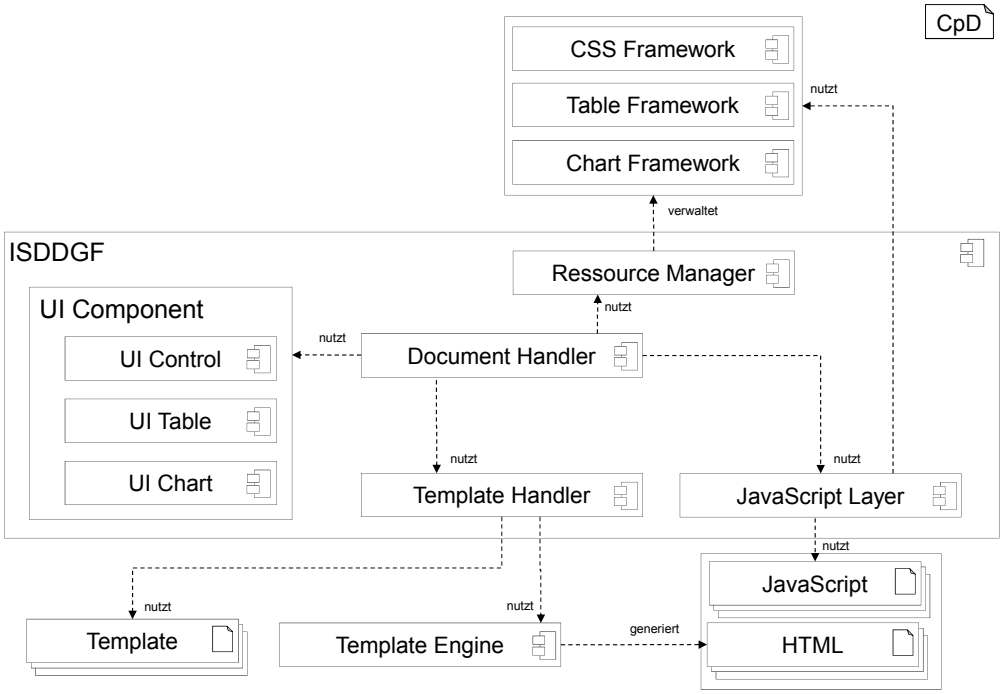


Abbildung 7.6: Architektur des Frameworks *ISDDGF* (basierend auf [Kha16]).

einer HTML-Seite (*UI Control*) repräsentieren. Dies bildet die geforderten Hauptdarstellungselemente ab und erlaubt deren Konfiguration.

- **Document Handler** Der *Document Handler* stellt wiederum das Hauptdokument zur Verfügung, auf dessen Basis die Komponente der **UIComponent** eingebettet werden können. Er stößt die Code-Generierung sowie die Ressourcenverwaltung an und bildet den Einstiegspunkt für den Entwickler.

- **Ressource Manager** Die Ressourcenverwaltung wird während der Codegenerierung über den *Document Handler* bei auftretenden Events informiert und übernimmt die Verwaltung aller technischen Abhängigkeiten des resultierenden Berichtes. Ist die Generierung abgeschlossen, so werden alle technischen Abhängigkeiten zur Ausführung dem generierten Bericht hinzugefügt, so dass dieser keinerlei offene Abhängigkeiten mehr aufweist (Anforderung **SimA15**).

- **Template Handler** Der *Template Handler* stellt die Schnittstellen zwischen einer externen Template Engine und dem *Document Handler* dar. Er fungiert als *Adapter* [VHJG95] und ermöglicht die zugrundeliegende Template Engine leicht auszutauschen.

- **JavaScript Layer** Eine Ansammlung von Java-Skripten, die es ermöglichen, den generierten Berichten dynamische Aspekte hinzuzufügen. Die entsprechenden Methoden müs-

sen über den *Document Handler* registriert werden und werden anschließend nach der Generierung durch den *Ressource Manager* dem Bericht hinzugefügt.

- **Template Engine** Verwendete Template Engine die auf Basis definierter Templates HMTL generiert. Sie bildet das Herzstück der Generierung, ist aber kein direkter Bestandteil des Frameworks, sondern kann ausgetauscht werden. Der Adapter *Template Handler* bildet dabei die Schnittstelle zwischen dem Framework *ISDDGF* und der externen Template Engine.

- **CSS Framework** Paket aus CSS-Dateien, die den Stil der HMTL-Webseiten definiert. Auch diese sind austauschbar und deren Verwendung wird über den *Ressource Manager* verwaltet.

- **Table Framework** Paket aus CSS (und eventuell JavaScript)-Dateien, die die Darstellung von Informationen in Form von Tabellen übernimmt (Anforderung **SimA14**).

- **Chart Framework** Paket aus CSS (und eventuell JavaScript)-Dateien, die die Darstellung von Informationen in Form von Diagrammen übernimmt (Anforderung **SimA14**).

Das Framework *ISDDGF* ist so aufgebaut, dass es unabhängig von der verwendeten *Template Engine*, dem *CSS Framework*, dem *Table Framework* und dem *Chart Framework* genutzt werden kann.

Soll ein entsprechendes Framework ausgetauscht werden, so sind nur der *RessourceManager* und der *TemplateHandler* anzupassen.

Im Kontext der Webentwicklung stehen dabei eine größere Auswahl an Template Engines und entsprechender CSS Frameworks zur Verfügung und es ist absehbar, dass diese in Zukunft weiterentwickelt, beziehungsweise durch neuere und bessere Frameworks ersetzt werden.

Um nun für den aktuellen Zeitpunkt eine möglichst gute Wahl zu treffen, wurde eine intensive Recherche angestoßen, um für jeden notwendigen Frameworktypen den geeigneten Kandidaten zu identifizieren [Kha16].

Im Folgenden werden schrittweise die betrachteten Kandidaten und evaluierten Attribute für jedes Framework beschrieben, um anschließend eine Auswahl zu treffen.

Dabei wurden folgende Attribute bei jedem Frameworktypen betrachtet [Kha16]:

- **Anwendbarkeit** Anwendbarkeit bezogen auf die Verwendung des Frameworks aus der Sicht des Entwicklers. Dabei spielt sowohl der Aufwand der Integration als auch die Mechanismen zur Erweiterung eine maßgebliche Rolle.

- **Erweiterbarkeit** Möglichkeiten der Erweiterung der Komponenten des Frameworks.

- **Feature** Anzahl an unterstützten Mechanismen beziehungsweise von Haus aus zur Verfügung stehenden Komponenten.

- **Community** Indikator für die Stärke der *Community*. Daraus lassen sich sowohl die Anzahl weiterer zur Verfügung stehender Komponenten als auch die Unterstützung bei auftretenden Problemen ableiten.

- **Dokumentation** Qualität und Quantität der durch den Anbieter zur Verfügung gestellten Dokumentation.

7.3.1 Template Engine

Für Template Engines zur Generierung von HTML auf Basis von Python haben sich in einer initialen Recherche folgende Kandidaten herauskristallisiert [Kha16]: *Jinja2*, *Mako*, *Cheetah*, *Pystache* und *Genshi*. In Tabelle 7.7 ist das Ergebnis der detaillierten Evaluierung auf Basis der genannten Attribute aufgezeigt. Dabei wurden Punkte von eins (sehr schlecht) bis fünf (sehr gut) vergeben.

Namen	Anwend.	Erweiter.	Feature	Comm.	Doc.	Summe
Jinja2	5	5	5	5	4	**24**
Mako	5	4	4	4	4	**21**
Cheetah	4	4	4	4	4	**20**
Pystache	4	3	4	4	4	**19**
Genshi	4	3	4	3	4	**18**

Tabelle 7.7: Evaluierung der Template Engines [Kha16].

Jinja2 stellt sich dabei als führend in allen Kriterien heraus und bietet eine sehr starke Community als auch eine große Menge an Funktionen. Details der Evaluierung können [Kha16] entnommen werden.

7.3.2 CSS Framework

Zuzüglich zu den bisher betrachteten Attributen sind bei der Evaluierung der CSS Frameworks noch folgende Attribute relevant [Kha16]:

- **UI Feature** Neben dem generellen Maß an zur Verfügung stehenden Featuren wird hierbei speziell auf die Menge an direkt verwendbaren, qualitativ hochwertigen UI-Komponenten geachtet.

Namen	Anw.	Erw.	Feature	UI Feat.	Cross	Comm.	Doc.	Summe
Twitter Bootstrap	4	3	4	5	4	5	5	**30**
Foundation	3	4	4	4	3	4	4	**26**
Dojo Toolkit	4	3	3	4	3	3	5	**25**
Brython	5	3	3	3	3	3	3	**23**
uikit	4	3	3	4	3	2	3	**22**
HTML Kickstart	4	2	2	4	3	2	3	**20**

Tabelle 7.8: CSS Framework Evaluierung [Kha16].

- **Cross Platform & Browser Support** Die Komponenten des Frameworks sollen auf möglichst vielen Endgeräten (PC, Tablet, Smartphone) verwendbar sein und auch möglichst alle gängigen Browser unterstützen.

Auf Basis unterschiedlicher Reviews [Gub13, Llc14, Iva16] wurden die Vorauswahl auf die folgenden Frameworks beschränkt: *Twitter Bootstrap*, *Foundation*, *Dojo Toolkit*, *Brython*, *uikit* und *HTML Kickstart*. In Tabelle 7.8 sind die Ergebnisse der Evaluierung aufgeführt.

Auf Grund der überragenden Menge an zur Verfügung stehenden UI-Komponenten, einer stark etablierten Community und einer Kompatibilität mit gängigen Browsern und Plattformen wurde *Twitter Bootstrap* als Favorit ausgewählt. Details der Evaluierung sind [Kha16] zu entnehmen.

7.3.3 Table Framework

Wiederum wurde für eine detaillierte Evaluierung auf Basis unterschiedlicher Reviews [Dee11, Dar13, JQu15a] eine Vorauswahl getroffen. Die Ergebnisse der Evaluierung ist in Tabelle 7.9 dargestellt.

Namen	Anw.	Erw.	Feature	UI Feat.	Cross	Comm.	Doc.	Summe
Bootstrap Table	5	4	4	5	4	5	5	**32**
DataTables	4	3	5	5	3	5	5	**30**
Jquery Bootgrid	4	3	4	5	3	3	4	**26**
Editable Grid	4	2	4	4	2	3	4	**26**
Slick Grid	4	2	4	3	2	3	4	**22**

Tabelle 7.9: Table Framework Evaluierung [Kha16].

Wiederum hat sich *Bootstrap Table* als Favorit herausgestellt, da es bezüglich UI-Komponenten, Community und Plattformunabhängigkeit überzeugte.

7.3.4 Chart Framework

Abschließend wurde ausgehend von unterschiedlichen Berichten [Sin15, Jqu15b, Rah15] auch eine Vorauswahl der zu evaluierenden *Chart Framework*s getroffen. Ein weiterer sehr vielver-

Namen	Anw.	Erw.	Feat.	UI Feat.	Cross	Comm.	Doc.	Int.	Sum.
NVD3	4	4	5	5	3	4	4	5	**34**
Chart.js	4	4	3	5	4	4	5	4	**33**
Chartist.js	4	4	4	4	4	4	4	4	**32**
C3.js	5	3	4	4	4	4	3	4	**31**
Plottable	4	3	4	5	3	3	3	4	**29**
Jqplot	4	3	5	4	3	3	3	4	**29**

Tabelle 7.10: Chart Framework Evaluierung [Kha16].

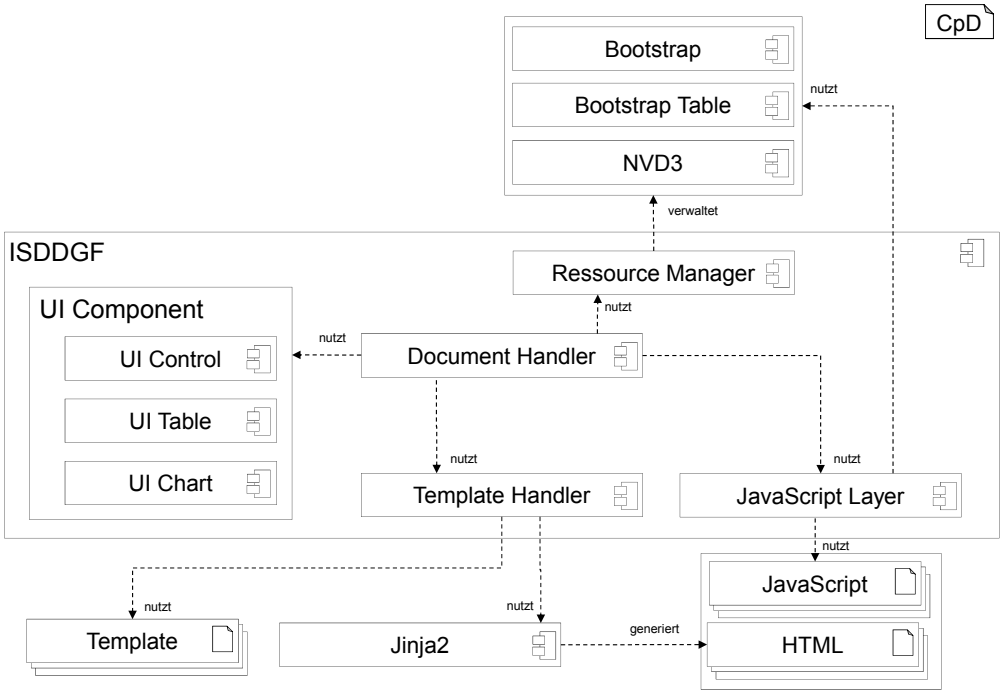


Abbildung 7.11: *ISDDGF* unter Verwendung von *Jinja2*, *Bootstrap*, *Bootstrap Table* und *NVD3*.

sprechender Kandidat war das Framework *Google Chart Library*. Da dieses keinen Zugriff auf den Quellcode erlaubt und gleichzeitig das Einverständnis einfordert, dass das Unternehmen Google auf die verwerteten Daten zugreifen darf, wurde es aber aus der weiteren Evaluierung ausgeschlossen.

Die Ergebnisse der Evaluierung sind wiederum in Tabelle 7.10 dargestellt. Dabei wurden auch die Interaktionsmöglichkeiten mit den dargestellten Diagrammen mit bewertet.

Da innerhalb dieser Evaluierung das Ergebnis erstaunlich knapp war, wurde die finale Auswahl auf Grund der hohen Interaktionsmöglichkeiten und persönlicher Wahrnehmung bezüglich der Darstellung der Diagramme getroffen.

Auf Basis der beschriebenen Auswahl ergibt sich das in Abbildung 7.11 dargestellte Gesamtbild. Nachdem nun im folgenden Abschnitt kurz auf das Vorgehen bei der Implementierung des *SimA* und des *ISDDGF* Frameworks eingegangen wird, wird anschließend der Umgang mit den durch *SimA* generierten Berichten beschrieben.

7.3.5 Implementierung

Die Implementierung der Frameworks *SimA* und *ISDDGF* wurden jeweils in zwei separaten Masterarbeiten umgesetzt [Kha16, Muc16]. Da diese Arbeiten zeitgleich durchgeführt wurden, war es möglich das Framework *ISDDGF* direkt in der Verwendung für *SimA* als auch in der Implementierung verschiedener Berichte für das Werkzeug SYNECT zu testen und im Sinne

einer agilen Entwicklung auf geänderte Anforderungen und Anmerkungen direkt reagieren zu können. Dabei half der rege Austausch zwischen beiden Entwicklern, Fehler im Design und Implementierung frühzeitig zu erkennen und dadurch die Qualität des Ergebnisses zu verbessern.

Details bezüglich der Implementierung sind [Muc16] (*SimA*) und [Kha16] (*ISDDGF*) zu entnehmen.

7.4 Tutorial

Im Folgenden werden zuerst die in Abbildung 7.5 skizzierten Teile des aus dem Framework *SimA* auf Basis des Frameworks *ISDDGF* unter Verwendung der in Abschnitt 5.1, Abschnitt 5.2 und Kapitel 6 definierten Ähnlichkeitsmetriken generierten Berichte im Detail erläutert. Anschließend wird auszugsweise auf die in Abschnitt 7.2 und Abschnitt 7.3 skizzierten Anpassungsmöglichkeiten der Frameworks *SimA* und *ISDDGF* eingegangen. Dabei sind in allen dargestellten Beispiele die Projektnamen verdeckt um das geistige Eigentum der involvierten Projektpartner und ihrer Kunden zu schützen.

7.4.1 Betrachtung der Ergebnisse der Ähnlichkeitsanalysen

Die Grundstruktur der Berichte basiert auf einer Anordnung unterschiedlicher Tabellen (repräsentiert unter Verwendung von *Bootstrap Table*) und Diagramme (repräsentiert unter Verwendung von *NVD3*). Des Weiteren ist für jede Seite ein Header definiert, der den Titel des aktuellen Teilberichtes darstellt. Werden auf einer Seite mehrere Tabellen und/oder Diagramme angezeigt, so sind diese wiederum durch separate Titel gekennzeichnet.

In Abbildung 7.12 ist die Grundstruktur der Tabellen anhand der Übersicht aller Software-Produkte und deren Qualitätsstand exemplarisch dargestellt.

Jede Tabelle bietet dabei eine Schnittstelle zur Konfiguration der angezeigten Spalten, zur Suche, zur spaltenweisen Suche, zur erweiterten Suche, zum Löschen der aktuellen Filter, zum spaltenweisen Sortieren und zum Export in ein spezifisches Dateiformat. Des Weiteren ist es möglich die Anzahl Zeilen / Elemente zu filtern, die gleichzeitig angezeigt werden.

In Abbildung 7.13 wird der Dialog angezeigt, der es ermöglicht, einen detaillierten Filter zu definieren. Dabei können für jeweils einzelne Spalten separate Suchkriterien in Form einfacher Zeichenketten angegeben werden.

Eine Suche ist des Weiteren auch spaltenweise durch direkte Eingabe in das spaltenspezifische Feld unter dem Titel der Spalte möglich, wie in für die Spalte *Projekt* Abbildung 7.14 dargestellt.

Die aktuelle Anzeige der Spalten kann, wie in Abbildung 7.15 aufgezeigt, angepasst werden, um den Fokus auf bestimmte Aspekte zu ermöglichen.

Auf Basis der aktuellen Filter und Spaltenselektionen können die dargestellten Daten zur weiteren Verarbeitung in die Dateiformate *JSON*, *XML*, *CSV*, *TXT*, *SQL* und *MS-Excel* exportiert werden, wie in Abbildung 7.16 dargestellt.

Des Weiteren ist es durch Auswahl der nach oben und unten zeigenden Dreiecke neben dem Titel der Spalte möglich, eine spaltenweise Sortierung (aufwärts oder abwärts) durchzuführen. Dies ist exemplarisch in Abbildung 7.17 durch eine absteigende Sortierung der Software-Produkte nach Anzahl der Einheiten, die keine extrinische ID aufweisen, veranschaulicht.

Diese Sortierung ermöglicht zum Beispiel eine effiziente Identifikation aller Software-Produkte und deren Einheiten, die im Kontext der produktgetriebenen Software-Produktlinienentwicklung aus Abbildung 3.1 auf Seite 29 in Schritt vier reevaluiert werden müssen.

Die gerade beschriebenen Features können im Kontext jeder dargestellten Tabelle genutzt werden, um effizient auf die gewünschte Aspekte und Daten zugreifen zu können (Anforderung **SimA14** und Anforderung **SimA9**).

Nach Vorstellung der generellen Struktur und der zugehörigen Features werden im Folgenden nun die einzelnen Teilberichte und deren Verbindungen im Detail besprochen.

Zuerst ist in Abbildung 7.18 eine Übersicht über alle Software-Produkte und deren Qualitätsstand gegeben (Vergleich zur Gesamtstruktur siehe Abbildung 7.5). Diese Übersicht ermöglicht die Qualität der Software-Produkte bezüglich ihrer Entwicklungsartefakte zu bemessen und gleichzeitig auch Fehler im Verfahren der Datenextraktion (*DataReader* in *SimA*) zu identifizieren. Fehlen entsprechende Artefakte, kann es auch sein, dass diese nicht entsprechend gegebener Richtlinien abgelegt wurden. In diesem Fall können entweder dem *DataReader* neue Sonderfälle hinzugefügt oder die Datenstruktur des entsprechenden Software-Produktes angepasst werden. Ist ein sehr hoher Prozentsatz fehlender Artefakte eines Typs gegeben, so spricht das dafür, dass entweder dieses Software-Produkt den Artefakttypen nicht zur Verfügung stellt oder die Artefakte nicht richtlinienkonform abgelegt wurden.

Durch Auswahl eines einzelnen Eintrags kann auf die Sicht, die in Abbildung 7.19 dargestellt ist, gewechselt werden. In dieser Sicht werden alle Einheiten des ausgewählten Software-Produkts angezeigt, die das Ähnlichkeitsattribut der entsprechenden Zeile nicht vorweisen kön-

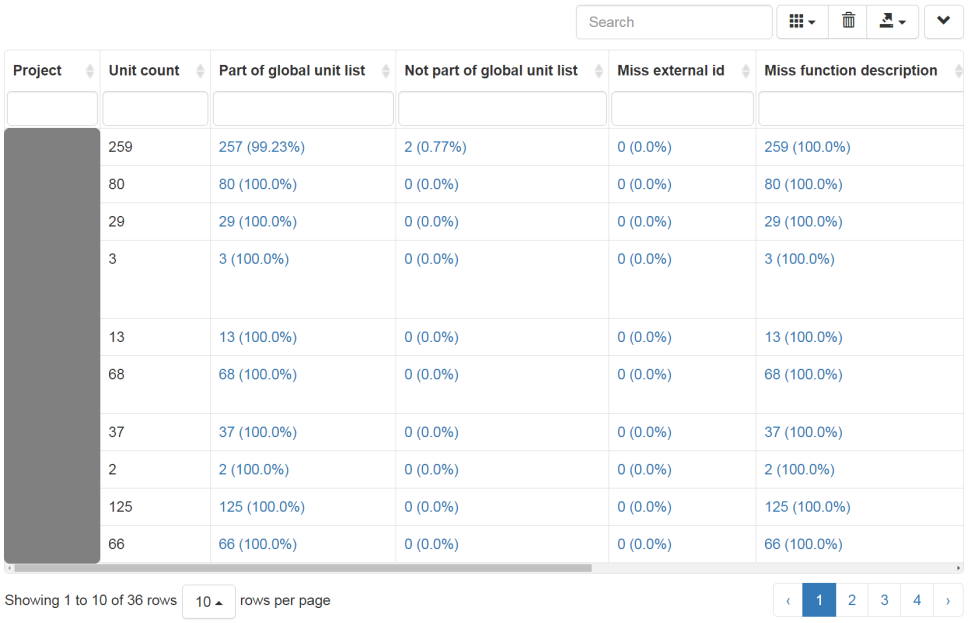


Abbildung 7.12: Grundstruktur der generierten Berichte des *SimA* Frameworks.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

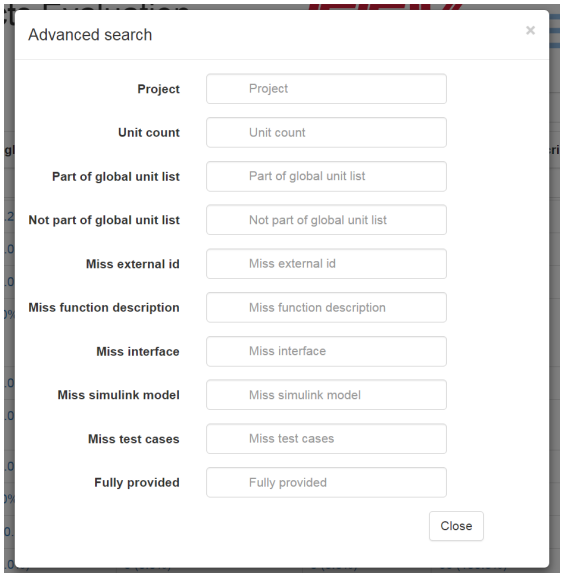


Abbildung 7.13: Erweiterte Suche in einem *SimA*-Bericht.

nen. In dem aktuellen Beispiel sind alle Einheiten des Software-Produkt *FRITS* aufgeführt, für die keine Schnittstellendefinition identifiziert werden konnte.

Zur Unterstützung der Reevaluierung aller Einheiten, die keine extrinsische ID aufweisen (Schritt 4 in Abbildung 3.1), kann über die Übersicht aus

Project	Unit count	Part of global unit list	Not part of global unit list	Miss external id	Miss function description	Miss interface	Miss sim
Dieter							
Dieter [redacted]	68	68 (100.0%)	0 (0.0%)	0 (0.0%)	68 (100.0%)	0 (0.0%)	2 (2.94%)
Dieter [redacted]	131	130 (99.24%)	1 (0.76%)	0 (0.0%)	131 (100.0%)	2 (1.53%)	5 (3.82%)
Dieter	0	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Dieter [redacted]	32	32 (100.0%)	0 (0.0%)	0 (0.0%)	32 (100.0%)	0 (0.0%)	3 (9.38%)

Abbildung 7.14: Spaltenspezifisches Filtern in einem *SimA*-Bericht.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

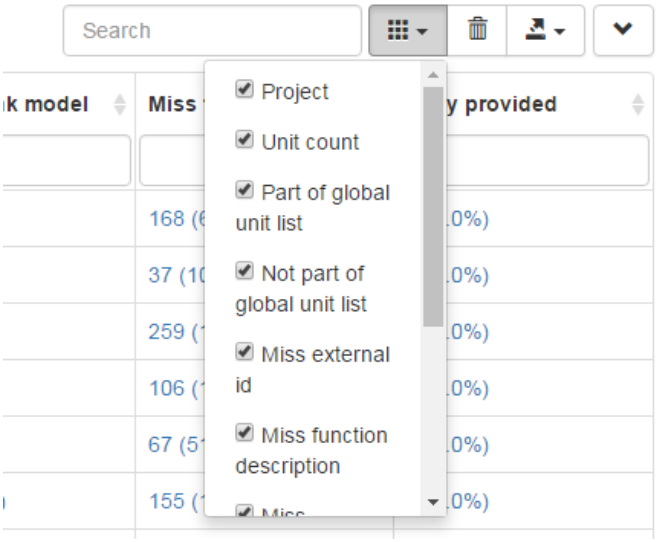


Abbildung 7.15: Anzeige spezifischer Spalten in einem *SimA*-Bericht.

Abbildung 7.18 auch ein entsprechender Teilbericht aufgerufen weren. Dieser führt zu der Übersicht, die in Abbildung 7.20 dargestellt ist. In dieser Übersicht kann direkt auf die zur Verfügung stehenden Artefakte zugegriffen werden, falls diese für eine Reevaluierung benötigt werden (Anforderung **SimA13**).

Als weiterer Einstiegspunkt ist die Übersicht der Ergebnisse der extrinsischen Analyse gegeben, wie in Abbildung 7.21 dargestellt. Diese stellt die Ergebnisse für eine direkte Übersicht in zwei Diagrammtypen dar. Das Balkendiagramm repräsentiert jeweils die Anzahl extrinsischer IDs gruppiert nach extrinsischen Paaren. In diesem Beispiel sind 732 Einheiten ohne jegliches Paar gegeben, während zum Beispiel 76 extrinsische IDs fünf Einheiten aufweisen.

Bezogen auf die Faustregel, dass sich die Entwicklung einer generischen Komponente erst nach zweifacher Wiederverwendung lohnt (Vergleich siehe Kapitel 3), stellt das Kuchendia-

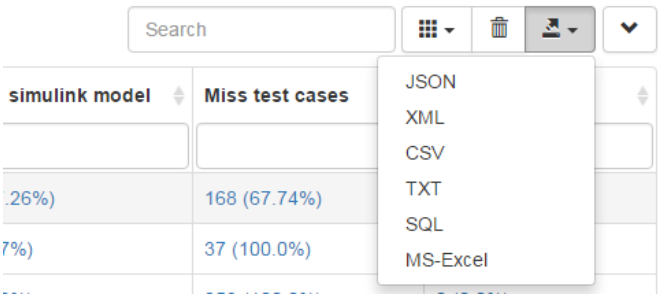


Abbildung 7.16: Export der selektierten Daten in gewünschtes Dateiformat.

Project	Unit count	Part of global unit list	Not part of global unit list
[redacted]	248	244 (98.39%)	4 (1.61%)
	37	34 (91.89%)	3 (8.11%)
	259	257 (99.23%)	2 (0.77%)
	106	105 (99.06%)	1 (0.94%)
	131	130 (99.24%)	1 (0.76%)
	155	154 (99.35%)	1 (0.65%)

Abbildung 7.17: Sortierung nach Anzahl Einheiten ohne extrinsische ID.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

gramm aus Abbildung 7.21 den Anteil an extrinsischen IDs mit einer, mit zwei und mit drei oder mehr Einheiten dar. Dabei bezieht sich dieser Wert nur auf die Auswertung der extrinsischen ID und soll eine erste grobe Übersicht über das aktuelle Wiederverwendungspotential geben.

Eine detaillierte Übersicht bietet stattdessen die in Abbildung 7.22 dargestellte Tabelle. Diese

Overview Projects Evaluation

FEV SE Software Engineering RWTH AACHEN UNIVERSITY

Search

Project	Unit count	Part of global unit list	Miss function description	Miss interface	Miss simulink model	Miss test cases
[redacted]	259	257 (99.23%)	259 (100.0%)	5 (1.93%)	0 (0.0%)	259 (100.0%)
	80	80 (100.0%)	80 (100.0%)	0 (0.0%)	0 (0.0%)	80 (100.0%)
	29	29 (100.0%)	29 (100.0%)	29 (100.0%)	1 (3.45%)	29 (100.0%)
	3	3 (100.0%)	3 (100.0%)	3 (100.0%)	0 (0.0%)	3 (100.0%)
	13	13 (100.0%)	13 (100.0%)	13 (100.0%)	13 (100.0%)	13 (100.0%)
	68	68 (100.0%)	68 (100.0%)	0 (0.0%)	2 (2.94%)	27 (39.71%)
	37	37 (100.0%)	37 (100.0%)	27 (72.97%)	11 (29.73%)	37 (100.0%)
	2	2 (100.0%)	2 (100.0%)	2 (100.0%)	2 (100.0%)	2 (100.0%)
	125	125 (100.0%)	125 (100.0%)	52 (41.6%)	57 (45.6%)	86 (68.8%)
	66	66 (100.0%)	66 (100.0%)	40 (60.61%)	40 (60.61%)	66 (100.0%)

Showing 1 to 10 of 36 rows 10 rows per page ‹ 1 2 3 4 ›

Abbildung 7.18: Übersicht über alle Software-Produkte und deren Qualitätsstand.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

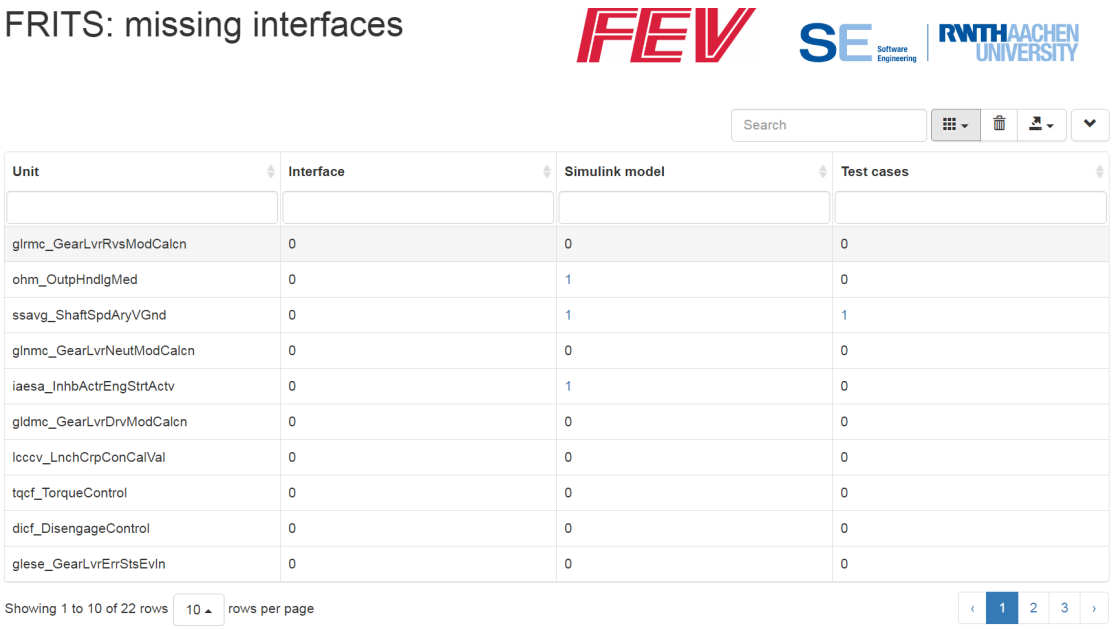


Abbildung 7.19: Teilbericht aller Einheiten eines Software-Produkts mit fehlender Schnittstellendefinition.

listet für jede extrinsische ID deren maximale, durchschnittliche und minimale Ergebnisse der schnittstellen-basierten und semantischen Analyse (der Übersicht wegen ausgeblendet) als auch deren Auftreten in den Software-Produkten auf (ebenfalls ausgeblendet).

Da aus Gründen der Lesbarkeit nicht alle Spalten angezeigt werden können, ist in Abbildung 7.23 das Auftreten einzelner extrinsischer IDs in unterschiedlichen Software-Produkten separat aufgeführt. Dabei wurde wiederum eine Vorauswahl an Software-Produkten getroffen.

Auf Basis dieser Tabelle kann auch nach Verwendungen in einzelnen Software-Produkten gefiltert werden, um ähnliche Varianten aus anderen Software-Produkten zu identifizieren. Zum Beispiel wird in Abbildung 7.24 nach allen extrinsischen IDs gefiltert, die auch im Software-Produkt *CCAP* vorkommen. Auf Basis dieser Information wäre es möglich, die Entwicklung in diesem Software-Produkt an der Entwicklung in anderen Software-Produkten zu orientieren beziehungsweise abzustimmen.

Wird in einem Software-Produkt in einem ersten Schritt auf Basis der Referenzarchitektur eine Grobarchitektur entworfen und diese Information durch *SimA* ausgewertet, ist es direkt möglich, auf Artefakte aus anderen Software-Produkten zuzugreifen und diese wiederzuverwenden. Dies ermöglicht ein deutlich systematischeres Vorgehen.

In Abbildung 7.25 sind die Details der schnittstellen-basierten Analyse aufgeführt. Durch Auswahl der Anzahl der Einheiten einer extrinsischen ID aus Abbildung 7.21 wird dieser Teilbericht aufgerufen.

Er beinhaltet sowohl die Auflistung aller Einheiten der extrinsischen ID und deren zur Verfügung stehenden Entwicklungsartefakte (was wiederum durch Auswahl einen direkten Zugriff

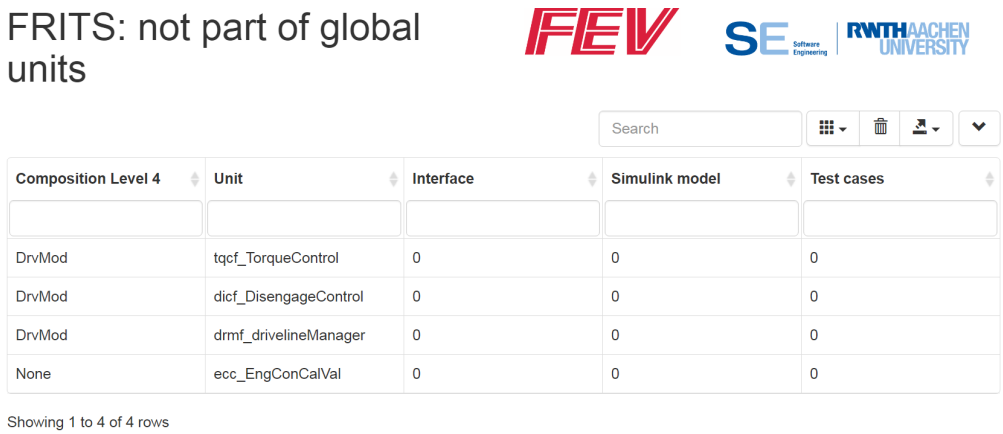


Abbildung 7.20: Teilbericht aller Einheiten mit fehlender extrinischer ID.

ermöglicht) als auch eine Übersicht der schnittstellen-basierten Ähnlichkeiten zwischen den Einheiten der jeweiligen Software-Produkte in Form eines Balkendiagrammes und einer Tabelle (Abbildung 7.26). Dabei ist der Informationsgehalt beider Darstellungsformen identisch und stellt nur eine andere Form der Darstellung dar.

Im Balkendiagramm werden für jedes Software-Produkt einzeln jeweils Balken aufgeführt, die die schnittstellen-basierte Ähnlichkeit zu den anderen Software-Produkten in der Reihenfolge ihres Auftretens darstellen. Der erste Balken des Software-Produktes *Dieter* stellt zum

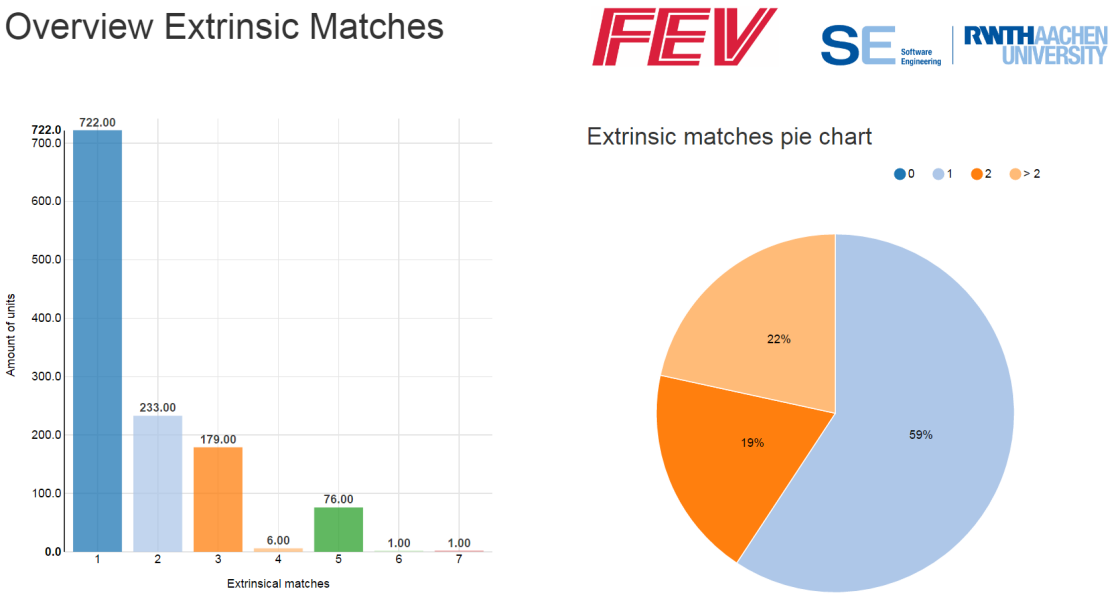


Abbildung 7.21: Teilbericht der extrinsischen Analyse (1/2).

Unit	Component	Extrinsic matches	Max. Structural similarity	Avg. Structural similarity	Min. Structural similarity
GCC_GlbConCalVal	GlbDa	7 (19.44 %)	0.0 %	0.0 %	0.0 %
GlbDa	GlbDa	6 (16.67 %)	-	-	-
CHS_ChrgnSp	ChCtl	5 (13.89 %)	100.0 %	100.0 %	100.0 %
SBA_TcSmBypActr	TcS	5 (13.89 %)	100.0 %	100.0 %	100.0 %
IV3_InjActr	Inj3	5 (13.89 %)	100.0 %	100.0 %	100.0 %
tcc_TraConCalVal	TrSDa	5 (13.89 %)	100.0 %	77.31 %	55.56 %
TQD_TqDstr	CoPOM	5 (13.89 %)	100.0 %	100.0 %	100.0 %
FLA_FullLoadActvn	TqDmd	5 (13.89 %)	100.0 %	100.0 %	100.0 %
IV1_InjActr	Inj1	5 (13.89 %)	100.0 %	100.0 %	100.0 %
WSA_WstgSmActr	TcS	5 (13.89 %)	100.0 %	100.0 %	100.0 %

Showing 1 to 10 of 1218 rows 10 rows per page ‹ 1 2 3 4 5 ... 122 ›

Abbildung 7.22: Teilbericht der extrinsischen Analyse (2/2).

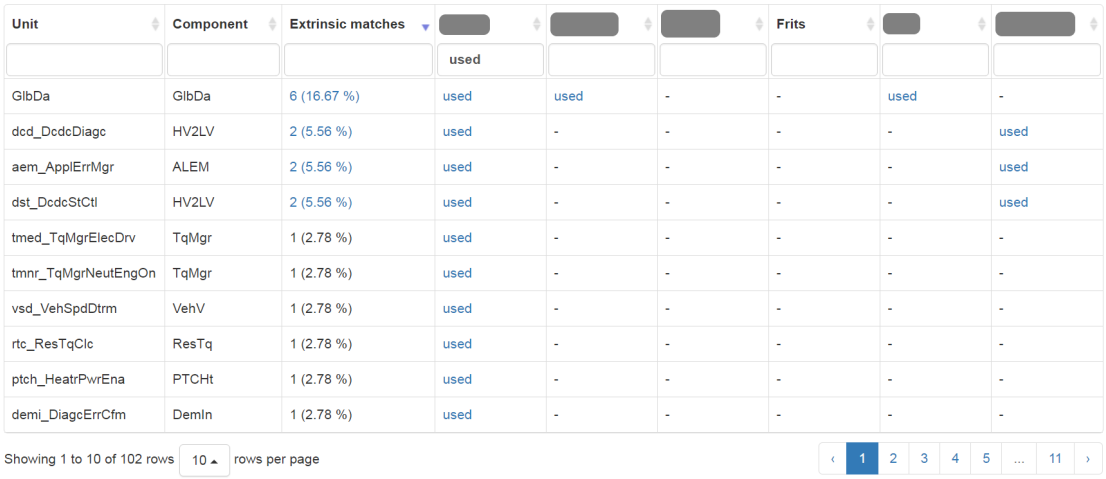
Beispiel die schnittstellen-basierte Ähnlichkeit zur Einheit des Software-Produktes *FRITS* dar, während der zweite Balken die schnittstellen-basierte Ähnlichkeit zum Software-Produkt *MIKA2* darstellt. Die tabellarische Darstellung beschreibt dieselbe Information in Matrixdarstellung.

Durch Auswahl eines Links *Detail signal similarities* erhält man Zugriff auf die Detaildarstellung der schnittstellen-basierten und semantischen Ähnlichkeitsanalyse, die aus Platzgründen in

Unit	Component	Dieter [redacted]	Dieter [redacted]	Dieter [redacted]	FRITS	Falcon
TOST_TraOilSmpT	TOST	used	used	used	used	-
LoB_LvBatt	LoB	used	-	used	used	-
TOPC_TraOilPmpCtl	TOPC	used	-	used	-	-
tom_TraOpmMngr	CoTOM	used	used	used	-	-
TOFC_TraOilFlowCalcn	TOFC	used	used	used	-	-
tge_TarGearEvln	TrSAC	used	-	-	-	-
TqDmd_TqDmd	TqDmd	used	-	used	used	-
tda_TqDmdArbn	TqDmd	used	-	used	used	-
TCIPt_TraCmdInfoPt	TCIPt	used	-	used	-	-
TrSAC_TraSysActrCtl	TrSAC	used	-	used	-	-

Showing 1 to 10 of 1218 rows 10 rows per page ‹ 1 2 3 4 5 ... 122 ›

Abbildung 7.23: Übersicht über das Auftreten einzelner extrinischer IDs in unterschiedlichen Software-Produkten. (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)



Unit	Component	Extrinsic matches	[redacted]	[redacted]	[redacted]	Frits	[redacted]	[redacted]
			used					
GlbDa	GlbDa	6 (16.67 %)	used	used	-	-	used	-
dcd_DcdcDiagc	HV2LV	2 (5.56 %)	used	-	-	-	-	used
aem_ApplErrMgr	ALEM	2 (5.56 %)	used	-	-	-	-	used
dst_DcdcStCtl	HV2LV	2 (5.56 %)	used	-	-	-	-	used
tmed_TqMgrElecDrv	TqMgr	1 (2.78 %)	used	-	-	-	-	-
tmnr_TqMgrNeutEngOn	TqMgr	1 (2.78 %)	used	-	-	-	-	-
vsd_VehSpdDtrm	VehV	1 (2.78 %)	used	-	-	-	-	-
rtc_ResTqClc	ResTq	1 (2.78 %)	used	-	-	-	-	-
ptch_HeatrPwrEna	PTCHt	1 (2.78 %)	used	-	-	-	-	-
demi_DiagcErrCfm	DemIn	1 (2.78 %)	used	-	-	-	-	-

Showing 1 to 10 of 102 rows 10 rows per page ‹ 1 2 3 4 5 ... 11 ›

Abbildung 7.24: Auftreten von extrinischen IDs aus der Sicht eines Software-Produkts. (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

Abbildung 7.27 und in Abbildung 7.28 dargestellt ist. Entsprechend der Schnittstellendefinition aus *PERSIST* (Vergleich siehe auch Abschnitt 2.1) sind die Vergleiche entsprechend der Typen *Eingangssignal*, *Ausgangssignal*, *Parameter*, *Konstante* und *Messpunkt* separiert. Grün markierte Zeilen stellen Mengen ähnlicher Elemente dar, die sich über zwei oder mehrere Schnittstellenvarianten hinweg ergeben haben. Durch Verwendung entsprechender Filter ist es hier auch möglich, nach bestimmten Signalen oder Ähnlichkeitswerten zu suchen.

Der Wert der schnittstellen-basierten Ähnlichkeit ist dabei jeweils in Klammern angegeben. Sind mehr als nur zwei Elemente Teil einer Menge ähnlicher Elemente, so sind die separierten Werte durch einen Querstrich ihres Auftretens entsprechend nacheinander angeordnet.

Des Weiteren ist für ausgehende Signalen nach Angabe der schnittstellen-basierten Ähnlichkeiten nach einer Trennung durch das Schlüsselwort *Sem* auch die jeweils semantische Ähnlich-

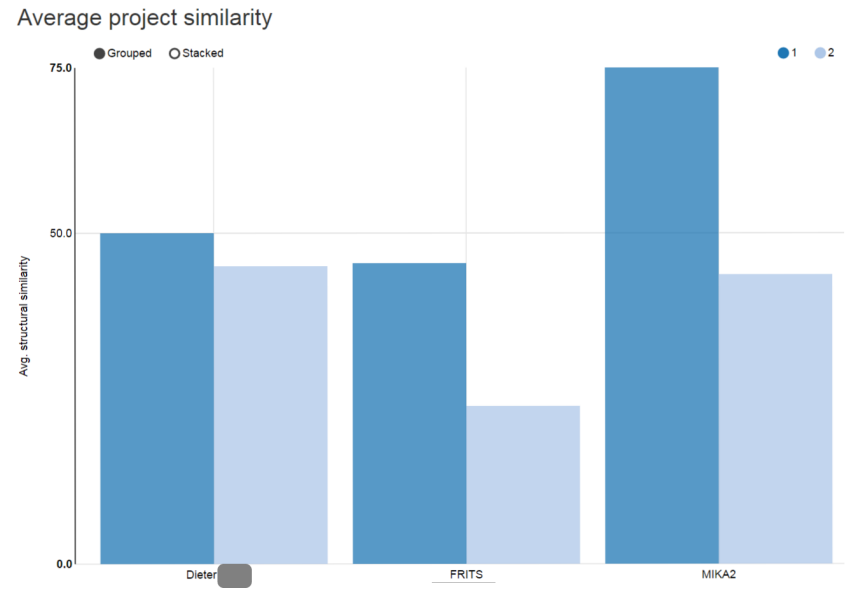
Similar units

Search

Project	Unit	Interface	Simulink model	Test cases
[redacted]	lccce_LnchCrpCluCpEvln	1	1	5
	lccce_LnchCrpCluCpEvln	1	1	1
	lccce_LnchCrpCluCpEvln	1	1	5

Showing 1 to 3 of 3 rows

Abbildung 7.25: Gesamtübersicht der schnittstellen-basierten Analyse (1/2). (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)



Structural analysis overview

Search

	Dieter	FRITS	MIKA2
Dieter	-	50.0	45.0
FRITS	45.45	-	23.86
MIKA2	75.0	43.75	-

Showing 1 to 3 of 3 rows

Abbildung 7.26: Gesamtübersicht der schnittstellen-basierten Analyse (2/2).
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

keit gegeben.

Abschließend ist es möglich, durch Auswahl eines schnittstellen-basierten oder semantischen Ähnlichkeitswertes auf die detaillierte Analyse eines Signalpaares zuzugreifen, die exemplarisch in Abbildung 7.29 dargestellt ist. Hier werden für jedes in der schnittstellen-basierten Analyse betrachtetes Attribut sowohl die Attributsinstanz als auch der zugewiesene Ähnlichkeitswert ausgewiesen. Diese Repräsentation dient primär der Analyse potentieller falscher positiver.

Dies ist der letzte Teilbericht, der durch das Framework *SimA* generiert wird, und schließt die in Abbildung 7.5 definierte Struktur entsprechend ab.

Fix

Search

lccce_LnchCrpCluCpEvIn (MIKA2)	lccce_LnchCrpCluCpEvIn (Dieter	lccce_LnchCrpCluCpEvIn (FRITS)
No matching records found		

Output signals

Search

lccce_LnchCrpCluCpEvIn (MIKA2)	lccce_LnchCrpCluCpEvIn (Dieter	lccce_LnchCrpCluCpEvIn (FRITS)
LCS_tqClu1CpTar(100.0 , Sem: -)	LCS_tqClu1CpTar(100.0 , Sem: -)	
LCS_tqClu2CpTar(100.0 , Sem: -)	LCS_tqClu2CpTar(100.0 , Sem: -)	
	LCS_stClu1TqCpAllwd(93.43 , Sem: -)	LCS_tqCluCpReq(93.43 , Sem: -)
	LCS_tqLnchCrpCluEstim(93.65 , Sem: -)	LCS_tqCluCpLnchCrpReq(93.65 , Sem: -)
	LCS_stClu2TqCpAllwd(95.16 , Sem: -)	LCS_stCluCtl(95.16 , Sem: -)
LCS_stClu2TqCpAllwd		
LCS_tqLnchCrpCluEstim		
LCS_stClu1TqCpAllwd		
		LCS_tqPtCluFastDecReq
		LCS_bPtCluTqSlowDecReqActv

Showing 1 to 10 of 12 rows 10 rows per page ‹ 1 2 ›

Abbildung 7.27: Detailübersicht schnittstellen-basierter und semantischer Analyse extrinsischer Paare (1/2). (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

Measurement Points

Search

lccce_LnchCrpCluCpEvln (MIKA2)	lccce_LnchCrpCluCpEvln (Dieter [redacted])	lccce_LnchCrpCluCpEvln (FRITS)
	LCS_noCluPrio_MP	

Showing 1 to 1 of 1 rows

Input signals

Search

lccce_LnchCrpCluCpEvln (MIKA2)	lccce_LnchCrpCluCpEvln (Dieter [redacted])	lccce_LnchCrpCluCpEvln (FRITS)
lcacc_bLnchCrpStgyEna (-/-)	lcd_bLnchCrpDeac (-/-)	lcd_bEngTqReq (-/-)
lccp_tqLnchCluCpActvTarPls (100.0 / 100.0)	lccp_tqLnchCluCpActvTarPls (100.0 / 100.0)	lccp_tqLnchCluCpActvTarPls (100.0 / 100.0)
LCS_stClu2TqCpTar (100.0)	LCS_stClu2TqCpTar (100.0)	
cccs_tqCrpCluCpActvTar (100.0 / 100.0)	cccs_tqCrpCluCpActvTar (100.0 / 100.0)	cccs_tqCrpCluCpActvTar (100.0 / 100.0)
lcacc_tqCluCpActvTarDirChg (100.0 / 100.0)	lcacc_tqCluCpActvTarDirChg (100.0 / 100.0)	lcacc_tqCluCpActvTarDirChg (100.0 / 100.0)
lcpcc_tqCluCpPasTar (100.0 / 95.05)	lcpcc_tqCluCpPasTar (100.0 / 95.05)	lcetl_tqCluClsgTar (95.05 / 95.05)
	lccsc_bGearTarEquGearActv (89.62)	lcd_bEngFlplTqSlowDecReqActv (90.25)
	lccsc_bGearTarShaft1 (88.78)	lccc_bEngFlplTqSlowDecReqActv (89.46)
	LCS_stClu1TqCpTar	
		lcd_tqEngFlplSlowDecReq

Showing 1 to 10 of 14 rows 10 rows per page ‹ 1 2 ›

Abbildung 7.28: Detailübersicht schnittstellen-basierter und semantischer Analyse extrinsischer Paare (2/2). (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

First	Second	Data type	Label type	Name	Range	Unit value	Width
Dieter [redacted]	FRITS	(Float32 / Float32) SimilarityDegree.Equal, 100.0%	(IN / IN) SimilarityDegree.Equal, 100.0%	(lcpcc_tqCluCpPasTar / lcetl_tqCluClsgTar) SimilarityDegree.Similar, 70.27%	(-400.0:1648.0 / -400.0:1648.0) SimilarityDegree.Equal, 100.0%	(Nm / Nm) SimilarityDegree.Equal, 100.0%	(1.0 / 1.0) SimilarityDegree.Equal, 100.0%

Showing 1 to 1 of 1 rows

Abbildung 7.29: Details der schnittstellen-basierten und semantischen Analyse eines Signalpaares. (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

Kapitel 8

Datenbankgestütztes Varianten-, Modell- und Signalmanagement

Die in Abschnitt 3.2 definierte produktgetriebene Software-Produktlinienentwicklung ermöglicht eine schrittweise Etablierung und Wartung einer Software-Plattform durch einen reaktiven und extraktiven Ansatz. Dieser erlaubt aus pragmatischen Gründen eine Entwicklung losgelöst von der Software-Produktlinie durchzuführen und die Ergebnisse schrittweise zu integrieren. Während dieser Bottom-up Ansatz es ermöglicht, das Software-Produkt ohne den initial notwendigen Mehraufwand der DE zu implementieren, ist es nicht im Interesse der Software-Produktlinie, dass diese Entwicklung keinerlei Schnittmenge mit der etablierten Software-Plattform aufweist. Umso größer die Gemeinsamkeiten zwischen Software-Produkt und Software-Plattform sind, umso leichter ist anschließend eine Migration. Außerdem ist es auch im Interesse der Software-Produktlinie, dass ein proaktiver Ansatz beziehungsweise eine direkte Umsetzung einer Software-Komponente (Schritt 9 in Abbildung 3.1 auf Seite 29), falls gewünscht, unterstützt wird. Neben einer frühzeitigen Verwendung und Wartung der Referenz-Architektur (Schritte 2-4 in Abbildung 3.1) ist es auch hilfreich einen globalen Zugriff auf die Schnittstellen der Einheiten der Software-Plattform zu ermöglichen, damit diese sowohl bei einer Bottom-up als auch einer Top-down Entwicklung Verwendung finden. Dabei ist es hilfreich, nicht nur die gesamte Schnittstelle, sondern auch deren einzelne Elemente getrennt zu versionieren und auf diese einen separaten Zugriff zu ermöglichen. Dies erlaubt auf der einen Seite eine kontrollierte Evolution der Software-Produktlinie und auf der anderen Seite eine feingranulare Wiederverwendung einzelner Elemente, ohne direkt die gesamte Schnittstelle übernehmen zu müssen. Unter Betrachtung der in Abschnitt 2.1 aufgeführten Standards zur Software-Entwicklung in der Automobilindustrie bildet die in Abbildung 8.1 dargestellte Schnittstellenbeschreibung eine möglichst allgemeine Darstellung, die auch mit PERSIST kompatibel ist. Dabei werden auch durch AUTOSAR definierte Client/Server-Schnittstellen unterstützt.

Eine Schnittstelle, *ClientServer* oder *SenderReceiver*, besteht aus einer Menge von Signalen, die entweder als Eingangs- oder Ausgangssignal (Parameter) im Kontext einer Schnittstelle verwendet werden. Das Signal selbst ist durch die Attribute *Name*, *Range* (logischer Wertebereich) und *Width* (Dimension) beschrieben und referenziert außerdem eine Typdefinition (Typedef). Diese beinhaltet den Datentypen (*Datatype*) und die physikalische Einheit (*Unit*). Ist der Datentyp eine Enumeration, so ist auch die exakte Definition dieser separat aufgeführt. Um eine Wiederverwendung auf unterschiedlichen Granularitätsstufen zu ermöglichen, ist es notwendig, die Schnittstelle, das Signal, die Typdefinition und die Enumeration als separate wiederverwendbare Elemente zu definieren und zu warten. Gleichzeitig ist es notwendig, einen globalen Zugriff

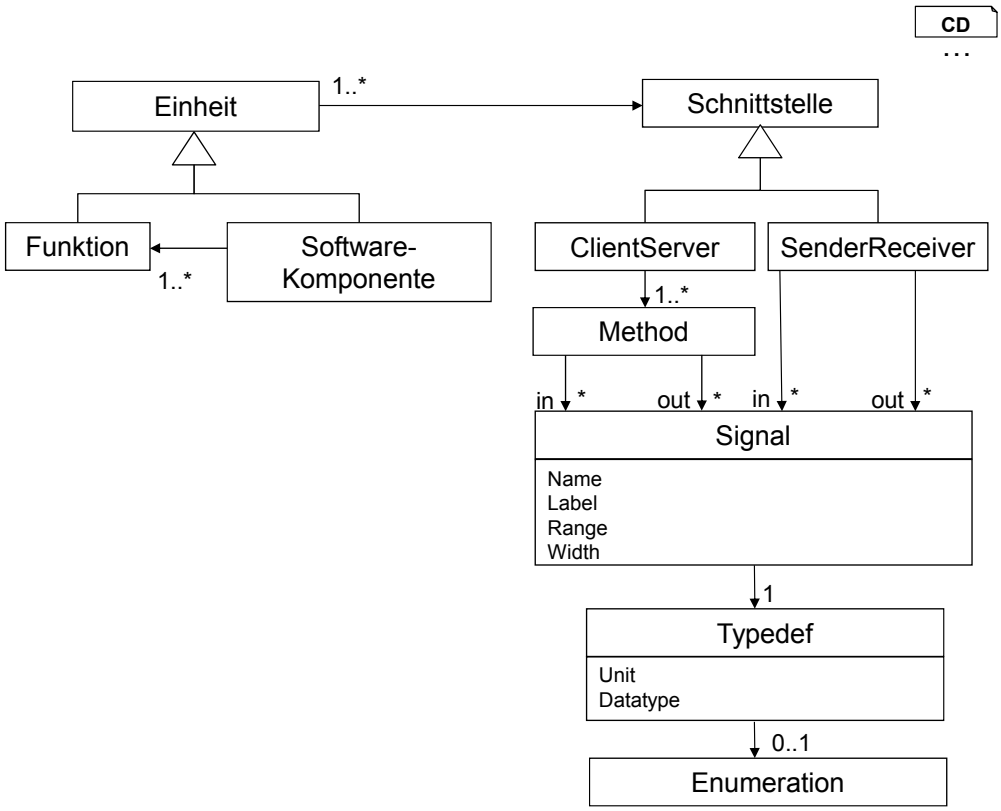


Abbildung 8.1: Allgemeine Schnittstellen-Definition im Automobilkontext.

auf diese Elemente zu ermöglichen, um die Wiederverwendung möglichst zu vereinfachen. Um gleichzeitig aber eine kontrollierte Evolution der Software-Produktlinie zu ermöglichen, müssen Zugriffsrechte feingranular definiert werden können. Auch muss es möglich sein, die Verwendung einzelner Einheiten in unterschiedlichen Software-Produkten direkt zu erkennen, um bei Änderungsanfragen entsprechende Change Impact Analysen durchführen zu können.

Ein datenbankgestütztes Varianten-, Modell- und Signalmanagement kann diese Anforderungen erfüllen, ohne bei einer gesteigerten Nutzung der Software-Plattform auf lange Sicht Engpässen im Kontext einer Multi-User-Nutzung zu unterliegen. Zusammengefasst muss ein datenbankgestützter Ansatz folgende Anforderungen bedienen:

1. **DB1** *Trennung von produkt- und produktlinien-spezifischen Elementen*
   Eine Trennung von produkt- und produktlinienspezifischen Elementen muss möglich sein.

2. **DB2** *Feingranulare Versionierung*
   Jede der in Abbildung 8.1 aufgeführten Klassen muss als separates Element mit entsprechender Versionierung definiert werden können.

3. **DB3** *Referenzierung über Projektgrenzen*

Die einzelnen Elemente der Datenbank können in unterschiedlichen Kontexten referenziert werden. Das Anlegen einer Kopie ist nicht notwendig. Dies erlaubt eine direkte Wiederverwendung, ohne dass separate Elemente angelegt werden. Insofern können Software-Produkte sich auf Elemente der Software-Plattform beziehen.

4. **DB4** *Versionierte Referenzen*
   Referenzen auf Elemente müssen sich auf spezifische Versionen des Elementes beziehen. Dies ist notwendig, da bei der Evolution der Software-Plattform nicht direkt alle Software-Produkte mit angepasst werden können.

5. **DB5** *Kompositionale Variabilität*
   Um den in Abschnitt 3.3 beschriebene kompositionalen Variabilitätsmechanismus abbilden zu können, ist es notwendig, Komponentenvarianten und eine variantenbezogene Parametrisierung von Funktionen definieren zu können.

6. **DB6** *Variabilitätsmodell*
   Es ist notwendig ein Variabilitätsmodell zu definieren, das in der Tradition von Merkmalmodellen [CE00, KCH+90a] alle gemeinsamen und variablen Aspekte beinhaltet, so dass dieses mit der durch Software-Produktlinie realisierten Anforderungen verbunden werden können (wie in Abbildung 3.7 auf Seite 41 dargestellt).

7. **DB7** *Verbindung zwischen Anforderungen und Varianten*
   Zwischen den Varianten des Variabilitätsmodells und den Anforderungen, die diese repräsentieren, muss eine Verbindung definiert werden können.

8. **DB8** *Verbindung zwischen Varianten, Komponentenvarianten und Parametrisierungen*
   Zwischen einzelnen Varianten des Variabilitätsmodells und den Software-Komponten, den Funktionen, sowie den variantenspezifischen Parametrisierungen der Funktionen muss eine direkte oder indirekte Verbindung etabliert werden können (wie in Abbildung 3.7 auf Seite 41 dargestellt).

9. **DB9** *Verbindung zwischen Anforderung und Implementierung*
   Im Sinne einer Serienentwicklung im Kontext der Automobilindustrie muss eine entsprechende durchgängige Nachverfolgbarkeit von Anforderungen zur Implementierung für die einzelnen Software-Produkte gewährleistet sein.

10. **DB10** *Definition von Variantenkonfigurationen*
    Auf Basis des Variabilitätsmodells müssen valide Variantenkonfigurationen definiert und invalide identifiziert werden können.

11. **DB11** *Ausleitung eines Produktes*
    Auf Basis einer gegebenen Verbindung zwischen Anforderungen, den Varianten des Variabilitätsmodells den Software-Komponenten und den Funktionen mit zugehöriger Parametrisierung der Funktionen muss auf Basis einer validen Variantenkonfiguration eine direkte Ausleitung des Software-Produktes möglich sein.

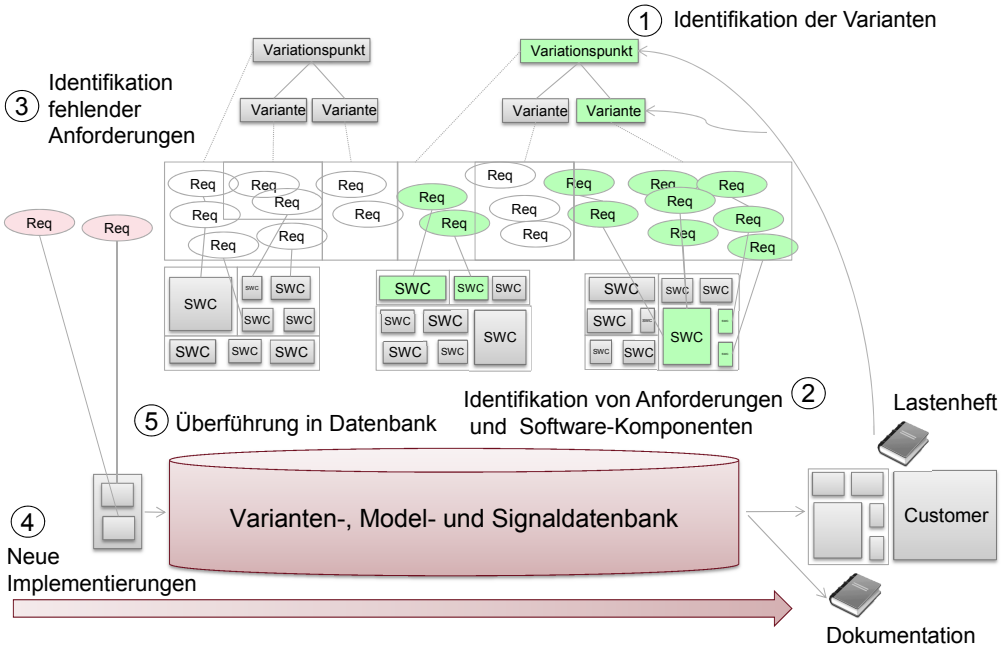


Abbildung 8.2: Datenbankgestützte reaktive Software-Produktlinienentwicklung.

12. **DB12** *Identifikation von Abhängigkeiten*
    Es muss möglich sein, die direkten und indirekten Abhängigkeiten einzelner Elemente der Datenbank zu anderen Elementen und Software-Produkten aufzulisten.

13. **DB13** *Konfiguration der Zugriffsrechte*
    Schreib- und Leserechte an produktlinien- und produktspezifischen Elementtypen müssen separat eingestellt werden können, um einen Konflikt zwischen DE und AE zu vermeiden.

14. **DB14** *Anbindung an Simulink*
    Um einen möglichst effizienten Arbeitsablauf zu ermöglichen, müssen die in der Datenbank hinterlegten Daten in der aktuellen Arbeitsumgebung der Entwickler zur Verfügung stehen.

15. **DB15** *Separater Offline-Zugriff*
    Um einen möglichst effizienten Arbeitsablauf zu ermöglichen, müssen die in der Datenbank hinterlegten Daten auch in unterschiedlichen Aufbereitungsformen offline zur Verfügung stehen.

Mit Hilfe der beschriebenen Anforderungen lässt sich ein reaktiver Prozess skizzieren, der auf Basis einer etablierten Softwareproduktlinie eine effiziente Bedienung des Tagesgeschäfts fördert und gleichzeitig eine konsequente Weiterentwicklung der Softwareproduktlinie ermöglicht. Dieser Prozess ist in Abbildung 8.2 skizziert.

Auf Basis eines durch den Kunden gegebenen Lastenhefts werden in einem ersten Schritt die vom Kunden geforderten Varianten identifiziert (1). Da sowohl eine Verbindung zwischen den Varianten und den Anforderungen als auch den Funktionen, Software-Komponentenvarianten und Funktionsparametrisierungen gegeben ist, ist es möglich, die zugehörigen Einheiten direkt zu identifizieren (2). Des Weiteren werden auch alle notwendigen doch aktuell nicht vorhandenen Varianten und Anforderungen identifiziert (3). Für diese ist es in einem zweiten Schritt notwendig, zugehörige Software-Komponenten, Funktionen und Funktionsparameter zu realisieren (4) und in einem dritten Schritt in die Datenbank zu überführen (5). Wiederum können hier zuerst Software-Komponentenvarianten und Funktionsvarianten implementiert werden (bei Verwendung von Software-Komponentenmustern (siehe Definition 3.7 auf Seite 40).

Durch eine Verbindung zwischen Varianten und Anforderungen und in einem weiteren Schritt zwischen Anforderungen und Software-Komponentenvarianten, Funktionen und Funktionsparametern ist es möglich auf Basis einer validen Variantenkonfiguration, die dem Lastenheft des Kunden entspricht, ein initiales Software-Produkt zu komponieren. Die zugehörigen Funktionsvarianten referenzieren, wo gegeben, auch die zugehörigen Implementierungen. Dieses initiale Software-Produkt kann im Folgenden verwendet werden, um ein lauffähiges Software-Produkt in einem Integrationsschritt zu etablieren. Der Fokus dieser Arbeit liegt dabei nicht auf dem eigentlichen Integrationsschritt, sondern in den beschriebenen Vorarbeiten.

Im Folgenden werden die beschriebenen Arbeitsschritte im Detail erläutert.

8.1 Datenbankgestützte reaktive Software-Produktlinienentwicklung mit SYNECT

Um ein datenbankgestütztes Varianten-, Modell- und Signalmanagement, wie durch die in Kapitel 8 definierten Anforderungen beschrieben, durch eine eigene Implementierung umzusetzen, ist grundsätzlich ein hoher Aufwand notwendig. Dies ist im Kontext einer Dissertation nicht zu realisieren, gerade wenn die Qualität eines Produktes zu erreichen ist.

Seitens des Industriepartners wurde entschieden, dass die beschriebene datenbank-gestützte entsprechender Prozess möglichst zeitnah durch ein hochwertiges Werkzeug unterstützt werden soll, damit der Prozess sowohl in Serien- als auch in Prototypenentwicklung Verwendung finden kann. Als zu verwendendes Werkzeug wurde durch den Industriepartner SYNECT in der Version 1.5 (zu diesem Zeitpunkt aktuellste Version) von der Firma dSPACE vorgegeben, das den Hauptteil der in Kapitel 8 definierten Anforderungen (Anforderung **DB1**, Anforderung **DB2**, Anforderung **DB3**, Anforderung **DB4**,Anforderung **DB5**, Anforderung **DB9**, Anforderung **DB6**, Anforderung **DB10**, Anforderung **DB13** und Anforderung **DB15**) direkt oder indirekt erfüllt. Für fehlende Aspekte (Anforderung **DB7**, Anforderung **DB8**, Anforderung **DB11**, Anforderung **DB12** und Anforderung **DB14**) und notwendige Modifikationen wurden Konzepte definiert und diese anschließend im Zuge einer kundenspezifischen Anpassung seitens dSPACE und durch Unterstützung einzelner studentischer Arbeiten realisiert. Im weiteren Verlauf hat dSPACE mit den Versionen 2.3 und 2.4 Synect erweitert, so dass die Anforderungen Anforderung **DB7**, Anforderung **DB8** und Anforderung **DB12** in diesen aktuelleren Versionen nun auch erfüllt sind.

Um alle Anforderungen umsetzen zu können, ist es notwendig, die SYNECT-Module *SYNECT Base*, *Signal & Parameter Management*, *Variant Management* und *Model Management* zu

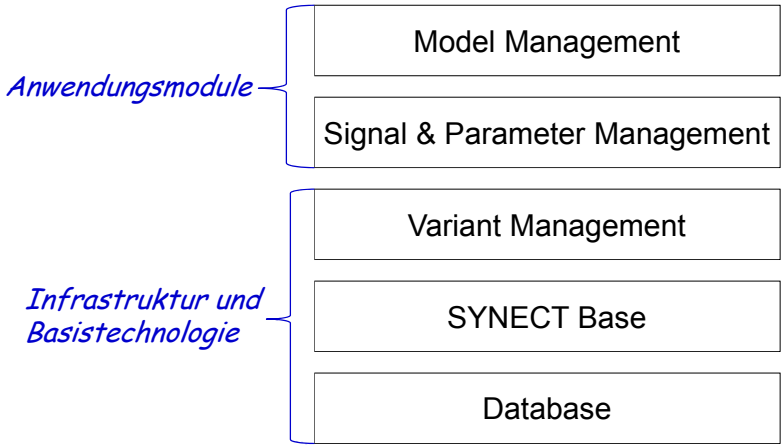


Abbildung 8.3: Übersicht verwendeter SYNECT-Module.

nutzen, wie in Abbildung 8.3 dargestellt.

Für weitere Informationen zu diesen Modulen kann die durch dSPACE zur Verfügung gestellte Dokumentation genutzt werden[1].

Im Folgenden werden die einzelnen Arbeitsschritte, die zur Durchführung des Prozesses aus Abbildung 8.2 notwendig sind, nacheinander direkt in der Werkzeugumgebung von Synect illustriert. Dabei definiert SYNECT selbst keinen expliziten Prozess, schafft aber Rahmenbedingungen, in denen eigene Vorstellungen zu realisieren sind. Wie im Umgang mit jedem Werkzeug oder Framework ist es dabei notwendig, Kompromisse einzugehen. In Abbildung 8.4 sind die Hauptaktivitäten, die durch SYNECT unterstützt werden, aufgeführt. Dabei werden die ersten beiden Schritte noch in einem Anforderungsmanagementwerkzeug durchgeführt und ihre Ergebnisse mit der Modelldatenbank synchronisiert. In Folgeschritten werden zugehörige Varianten identifiziert und fehlende Variationspunkte, Varianten und Varianteneinschränkungen im Variabilitätsmodell hinzugefügt und anschließend Verbindungen zwischen Anforderungen und Funktionen, Funktionsparametern, Bugs und Implementierungen sowie dem Variabilitätsmodell und Anforderungen und Parametrisierungen etabliert. Implementierungen referenzieren dabei die Verhaltensspezifikationen der Funktionen.

Sind diese etabliert, werden Schnittstellen und Verhaltensspezifikationen erstellt oder modifiziert. Außerdem können, falls notwendig, auf Basis neuer Variationspunkte und neuer Varianten, neue Variantenfkonfigurationen erstellt oder gegebene angepasst werden. Nach Abschluss dieser Schritte ist es dann möglich, auf Basis der etablierten Verbindungen ein Software-Produkt auszuleiten.

Nach Vorstellung des Prozesses und der einzelnen Arbeitsschritte wird auf die Details der Implementierung eingegangen.

[1] `https://www.dspace.com/de/gmb/home/products/sw/datenmanagement/synect.cfm`

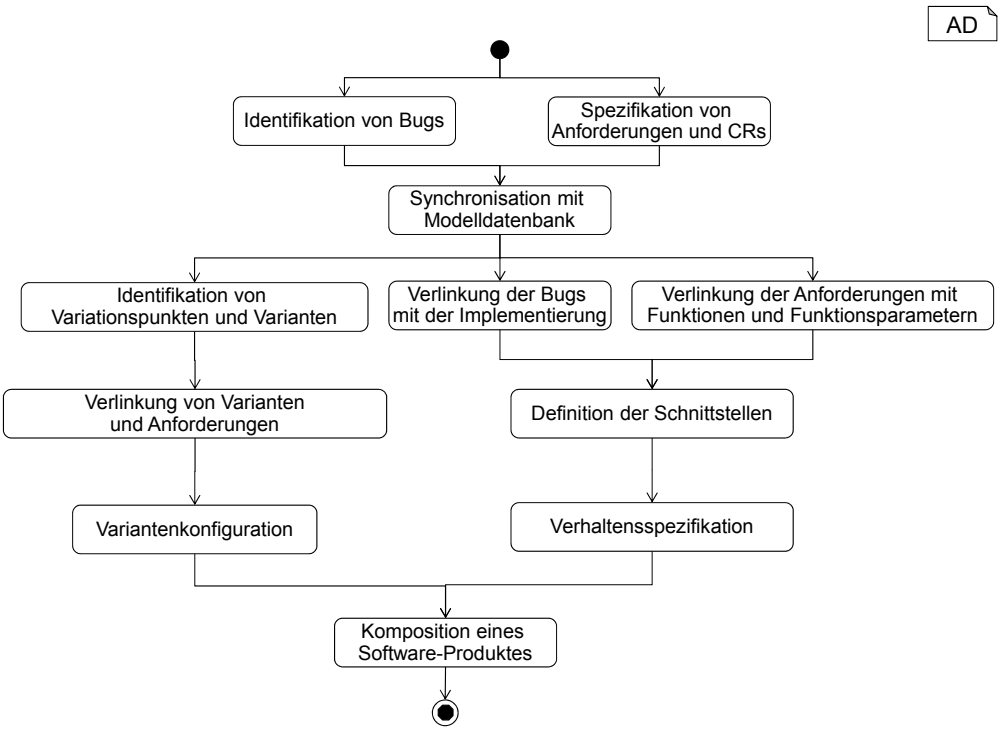


Abbildung 8.4: Datenbankgestützte reaktive Software-Produktlinienentwicklung.

8.1.1 Synchronisation mit Anforderungsmanagementwerkzeug

Zur Übersicht ist in Abbildung 8.5 das durch SYNECT vorgegebene Datenmodell abgebildet. Des Weiteren sind auf der linken Seite zugehörige Anforderungen (Requirement) und Änderungsanfragen (CR) aus einem zugehörigen Anforderungsmanagementwerkzeug abgebildet.

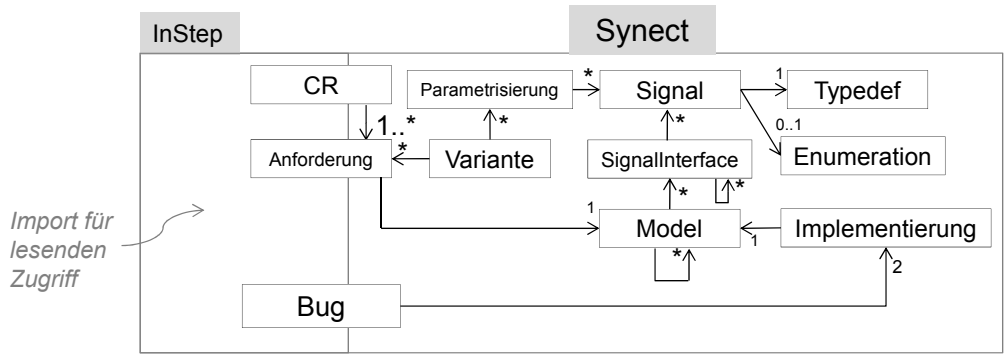


Abbildung 8.5: Ausschnitt Datenmodell SYNECT.

Beim Industriepartner wird das Werkzeug Instep[2] verwendet, wobei aber beliebige Anforderungsmanagementwerkzeuge angebunden werden können. Änderungsanfragen beziehen sich dabei immer auf schon existierende Anforderungen. Da teilweise der Arbeitsschritt, eine Anforderung entsprechend anzupassen, weitere Schritte innerhalb der Umsetzung nicht blockieren soll, werden sowohl Anforderungen als auch Änderungsanfragen mit SYNECT synchronisiert.

Des Weiteren ist es auch möglich, entsprechende Bugs mit SYNECT zu synchronisieren. Dadurch kann deren Bearbeitung angestoßen und deren Auftreten und Beheben einzelnen Implementierungsversionen und indirekt damit Modellversionen zugeordnet werden. SYNECT stellt als Datenbankelemente neben dem Repräsentanten für Anforderungen (kategorisiert als *CR*, *Anforderung* oder *Bug*) *Modell* ((repräsentieren Einheiten)), *Signal*, *SignalInterface* (repräsentiert Schnittstelle), *Signal*, *Typedef*, *Enumeration* und *Implementation* (repräsentiert Verhaltensspezifikation) ab. Dabei ist folgender Zusammenhang gegeben:

Ein *Modell* kann eine beliebige Menge an Submodellen enthalten und jedes dieser Modelle definiert wiederum eine Schnittstelle bestehend aus einer beliebigen Menge an *SignalInterface*s, die wiederum aus einer Menge aus Signalen besteht. *SignalInterface*s können wiederum hierarchisch gegliedert werden, um weitere Schachtelungen zu ermöglichen. Dies ist vor allen Dingen zur Unterstützung komplexerer Schnittstellen, wie *ClientServer*, hilfreich. Ein *Signal* bezieht sich wiederum auf einen konkreten *Typedef* und optional auch auf eine *Enumeration*.

Im Vergleich zur allgemeinen Schnittstellendefinition, die in Abbildung 8.1 definiert wurde, ist ein deutlich ähnlicher Zusammenhang dargestellt. Dies ist auch nicht weiter verwunderlich, da SYNECT auch mit Fokus auf die Automobilindustrie entworfen wurde.

[2] `https://www.microtool.de/projektmanagement-mit-in-step-blue/`

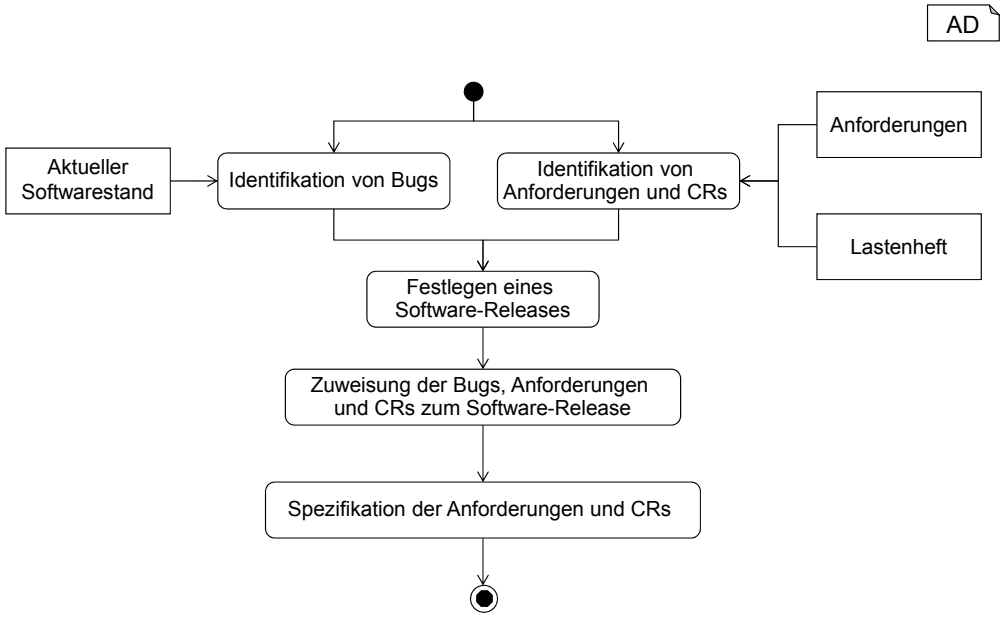


Abbildung 8.6: Spezifikation von Anforderungen und Bugtracking.

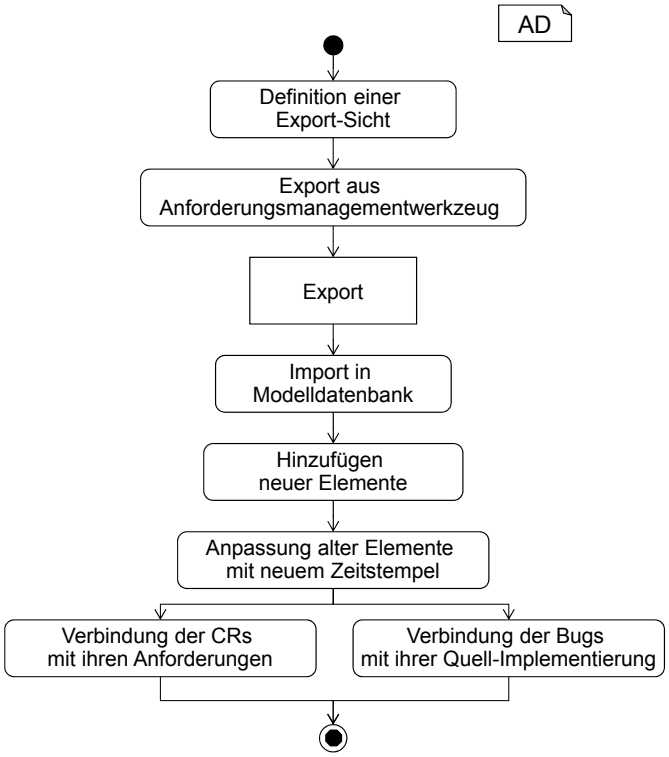


Abbildung 8.7: Übergang zwischen Anforderungsmanagementwerkzeug und SYNECT.

Die durch den kompositionalen Ansatz geforderte Auftrennung in *Funktion* und -*Software-Komponente* lässt sich durch Schachtelung von *Model*s repräsentieren. Während keine neuen Klassen seitens SYNECT hinzugefügt werden können, ist es möglich, beliebige Assoziationen (Links) und auch Attribute für einzelne Klassen hinzuzufügen. Dadurch ist eine Trennung von Funktion und Software-Komponente mittels klassifizierendem Attribut möglich. In anderen Fällen wurde ähnlich vorgegangen.

Im Folgenden wird nun schrittweise erläutert, wie die einzelnen Elemente in SYNECT der Software-Plattform so hinzugefügt werden können, dass anschließend die in Abbildung 8.2 geforderte Nachverfolgbarkeit zur automatischen Komposition von Software-Produkten auf Basis einer Variantenkonfiguration möglich ist.

Der Übergang vom Anforderungsmanagement zum Architekturdesign ist durch die Schritte beschrieben, die in Abbildung 8.6 und Abbildung 8.7 dargestellt sind. Zuerst werden zugehörige Anforderungen, CRs und Bugs definiert und anschließend, falls bekannt, einem Software-Release zugeordnet. Der Software-Release stellt dabei den Release dar, zu dem das entsprechende Element umgesetzt werden soll. Ein Bug enthält zudem die Zuweisung zu einem weiteren Software-Release, der den Release darstellt, in dem der zugehörige Bug identifiziert wurde.

Beide Informationen, die Liste aller Anforderungen, CRs, Bugs und auch die zugeordneten Software-Releases können dann jederzeit mit SYNECT synchronisiert werden. Bei der Syn-

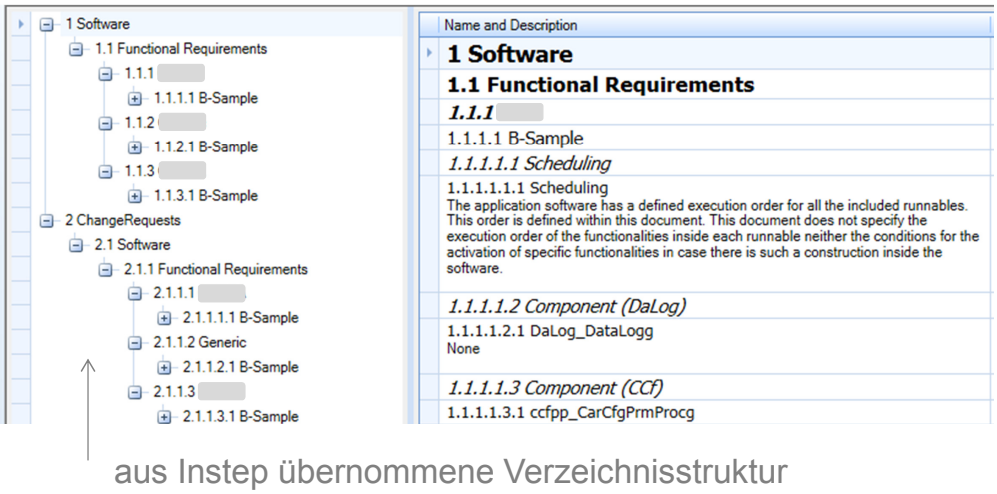


Abbildung 8.8: Darstellung von Anforderungen in SYNECT.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

chronisation werden alle synchronisierten Elemente (Anforderungen, CRs, Bugs) in der hierarchischen Struktur abgebildet, die durch das Anforderungsmanagementwerkzeug vorgegeben ist. Eine Synchronisation ist schrittweise möglich, falls jedem einzelnen Element eine eindeutige ID zuzuordnen ist.

Die Repräsentation von Anforderungen, CRs und Bugs innerhalb von SYNECT ist in Abbildung 8.8 dargestellt. Dabei ist eine Navigation über die links abgebildete Baumstruktur möglich. Auf der linken Seite sind die einzelnen Elemente durch Titel und textuelle Beschreibung dargestellt. Alle weiteren Eigenschaften können in einem separaten Fenster betrachtet werden. Der Zugriff ist hierbei nur lesend gegeben, Modifikationen müssen im Anforderungsmanagementwerkzeug durchgeführt werden. Die einzige Änderung, die auf Basis der angezeigten Sicht durchgeführt werden kann, ist die Modifikation von Verbindungen der Anforderungen, CRs und Bugs zu anderen Elementen innerhalb von SYNECT. Auch wenn in SYNECT beliebige Verbindungen gebildet werden können, verwendet der hier definierte Prozess nur Verbindungen von Anforderungen oder CRs zu Modellen (in denen diese umgesetzt werden) sowie von Bugs zu Implementierungen (wo diese gefunden wurden und wo diese behoben werden bzw. wurden), wie in Abbildung 8.8 dargestellt. Des Weiteren werden Anforderungen (und nur diese) mit dem Variabilitätsmodell innerhalb von SYNECT verknüpft. Dies wird später im Detail erläutert.

Die Synchronisation zwischen Anforderungswerkzeug und SYNECT kann dabei über verschiedene Austauschformate erfolgen. Es ist möglich in Instep beliebige Sichten direkt als Exceldatei zu exportieren, was eine gesteuerte Synchronisation erlaubt: Der Nutzer ist in der Lage, durch Angabe geeigneter Filter nur eine gewünschte Menge zu synchronisieren. Die Synchronisation findet in zwei Schritten statt. Zuerst wird in Instep eine entsprechende Sicht ausgewählt und durch Aufruf des Kontextmenüs ein Excel-Export ausgeführt. Im darauffolgenden Schritt wird innerhalb von SYNECT das jeweilige Anforderungsdokument aufgerufen, das synchroni-

siert werden soll. Nachdem das richtige Dokument ausgewählt wurde, kann über den Import-Reiter ein Instep-spezifischer Import aufgerufen werden. Nach Auswahl der exportierten Exceldatei wird die Synchronisation durchgeführt. Sind für den Prozess eindeutige Sichten zur Synchronisation definiert, so kann dieser Schritt auch dahingehend automatisiert werden, dass ein einzelner Aufruf seitens SYNECT ausreicht, um den Export und den anschließenden Import anzustoßen.

Während der Synchronisation wird auf Basis der eindeutigen ID jedes Elementes versucht, ein entsprechendes Match in der SYNECT-Datenbank zu finden. Wurde kein Match gefunden, so wird ein neues Element angelegt. Wurde ein Match gefunden, so hilft ein weiteres Attribut aus dem Anforderungsmanagementwerkzeug zu identifizieren, inwiefern ein Update notwendig ist: der Zeitpunkt der letzten Änderung. Bietet das zu synchronisierende Anforderungsmanagementwerkzeug diese Information nicht, so kann ein detaillierter Vergleich aller Informationen genutzt werden, um eine entsprechende Änderung zu identifizieren. Dies birgt allerdings auch die Gefahr, dass bei einer fehlerhaften Synchronisation (zum Beispiel mit altem Datenstand) valide Stände überschrieben werden. Dies ist gerade dann problematisch, wenn Synchronisationen nicht atomar direkt nach Änderung im Anforderungswerkzeug durchgeführt werden können, was in diesem Kontext der Fall ist. Insofern können Synchronisationen, die zeitlich verschoben zur Anpassung durchgeführt werden, neuere Stände überschreiben. Aus diesem Grund werden Änderungen nur auf Basis des Zeitstempels der letzten Änderung durchgeführt. Des Weiteren

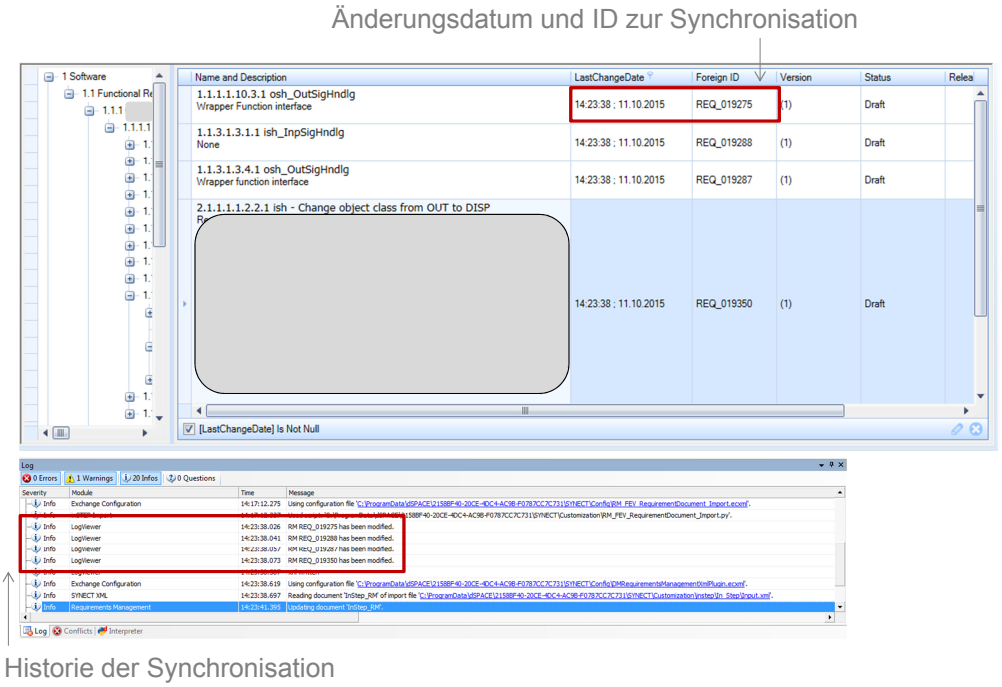


Abbildung 8.9: Anzeige der synchronisierte Anforderungen in SYNECT.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

wird der Synchronisationsmechanismus für alle anderen aktiven Anwender gesperrt, solange eine Synchronisation stattfindet, um diesen als atomare Transaktion durchführen zu können. In Abbildung 8.9 ist eine Übersicht nach erfolgter Synchronisation aufgeführt. Der Log am unteren Bildschirmrand informiert dabei über vollzogene Änderungen. Im oberen Fenster sind alle Anforderungen mit letztem Zeitpunkt der Änderung und ihrer ID aufgeführt. Es ist dabei möglich, in den Sichten von SYNECT nach beliebigen Attributen zu filtern. In diesem Beispiel sind alle Anforderungen aufgeführt, die einen Änderungszeitpunkt definiert haben.

Wurde eine Änderung identifiziert, so wird das aktuelle Element durch die Änderungen überschrieben und auch deren Anordnung in der Hierarchie gegebenenfalls angepasst.

Innerhalb von SYNECT werden alle in Abbildung 8.8 dargestellten Elemente separat versioniert. Dabei kann sich jedes Element entweder im Zustand *Draft* oder *Released* befinden. Elemente in *Draft* können modifiziert werden und es wird keine neue Version bei jeder Änderung angelegt. Erst wenn das Element bewusst in den Zustand *Released* versetzt wird, ist die aktuelle Version abgeschlossen und kann nicht weiter modifiziert werden. Um weitere Änderungen vornehmen zu können, muss eine neue Version des Elementes angelegt werden, was das Element in der neuen Version wieder in den Zustand *Draft* überführt.

Wurde nun während einer Synchronisation eine Änderung für ein Element identifiziert, das sich im Zustand *Draft* befindet, so wird die Änderung direkt durchgeführt. Ist das Element dagegen im Zustand *Released*, so wird zuerst eine neue Version angelegt und anschließend die Änderung durchgeführt.

8.1.2 Verbindung der Anforderungen

Nachdem eine Synchronisation durchgeführt wurde, kann mit der Umsetzung der synchronisierten Anforderungen, CRs oder Bugs begonnen werden. Dabei kann der in Abbildung 8.9 vorgestellte Filtermechanismus verwendet werden, um nur Änderungen seit der letzten Synchronisation zu betrachten, wie in Abbildung 8.10 dargestellt.

Auf Basis dieses Filters ist es nun möglich, schrittweise für jede Änderung Folgeschritte durchzuführen. Wie in Abbildung 8.11 dargestellt, ist es nun notwendig, eine Assoziation zwischen den synchronisierten Anforderungen und CRs zu den Modellen herzustellen.

Dabei müssen die in Abbildung 8.12 beschriebenen Schritte durchgeführt werden.

Existieren entsprechende Modelle für Funktion oder Software-Komponente nicht, so müssen diese zuerst definiert werden. Anschließend kann manuell eine Verbindung gezogen werden. Neben der Filterung nach dem letzten Änderungszeitpunkt kann es auch hilfreich sein, alle Anforderungen ohne gegebene Verbindung zu identifizieren, um offene Aufgaben abzuschließen. Dies ist in Abbildung 8.13 dargestellt.

Wurden die entsprechenden Partner für eine Verbindung identifiziert, so kann diese über Kontextmenüs manuell durchgeführt werden, wie in Abbildung 8.14 dargestellt.

Ist während der Erstellung einer Anforderung oder eines Bugs (bzw. des Bug-Eintrags) die Zuweisung zur Software-Komponente oder Funktion schon eindeutig, so kann durch Verwendung entsprechender Schlüsselwörter die Verbindung in SYNECT auch automatisiert durchgeführt werden. In den Beispielen aus Abbildung 8.13, Abbildung 8.10 oder Abbildung 8.9 sind jeweils Anforderungen aufgeführt, die im Titel jeweils zu Beginn das Kürzel oder den gesamten

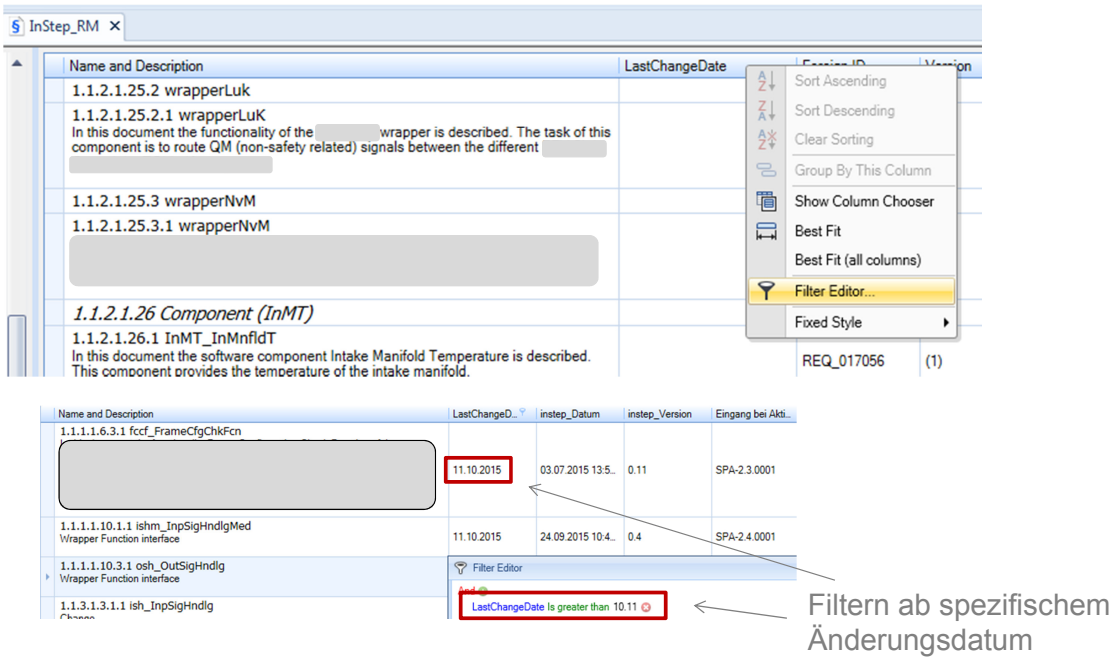


Abbildung 8.10: Anforderungen gefiltert nach Datum der letzten Änderung in SYNECT. (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

Namen der Funktion oder Software-Komponente aufführen. Auf Basis einer solchen Konvention

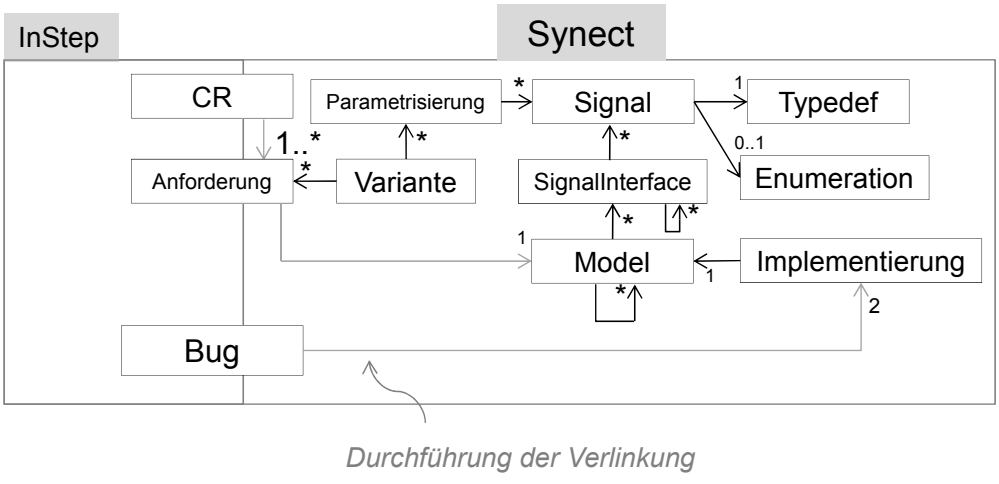


Abbildung 8.11: Durchführung der Verbindungen zwischen Anforderungen und Modell sowie Bug und Implementierung in SYNECT.

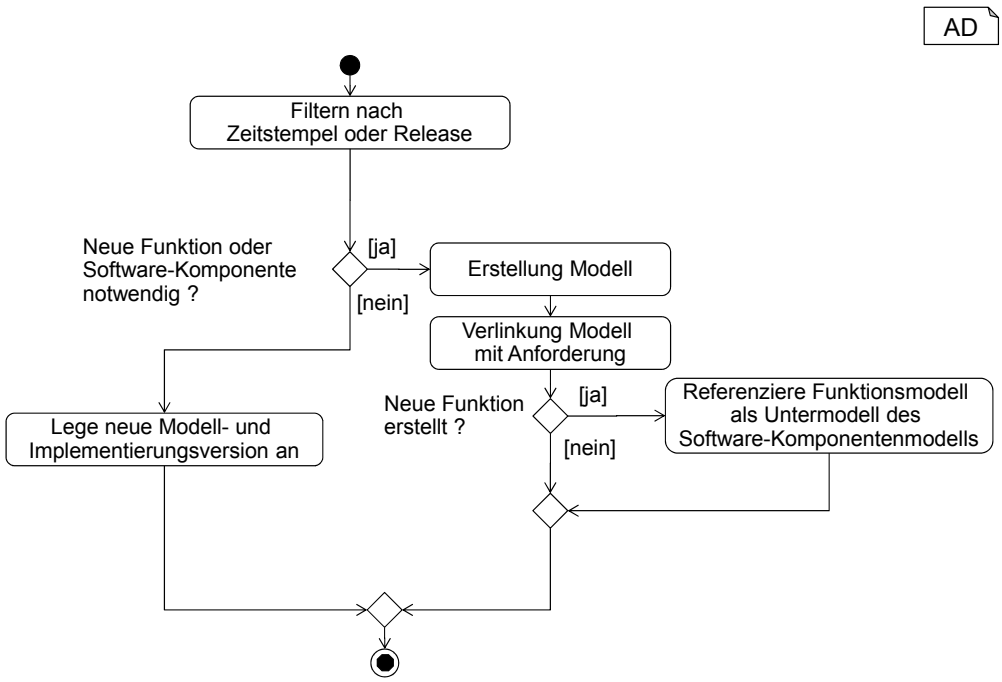


Abbildung 8.12: Anpassung der Modelle auf Basis neuer oder modifizierter Anforderungen in SYNECT.

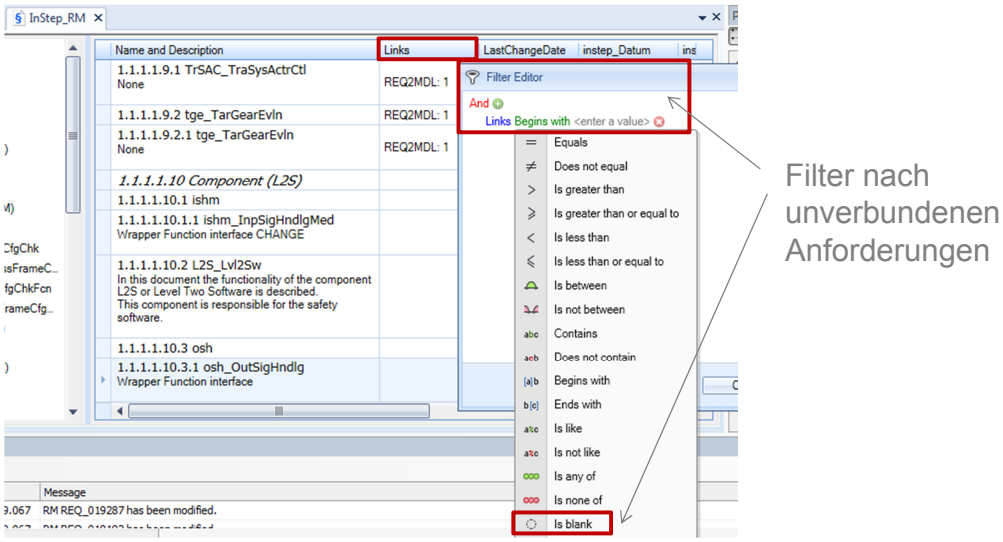


Abbildung 8.13: Darstellung aller Anforderungen ohne gegebene Verbindung in SYNECT.

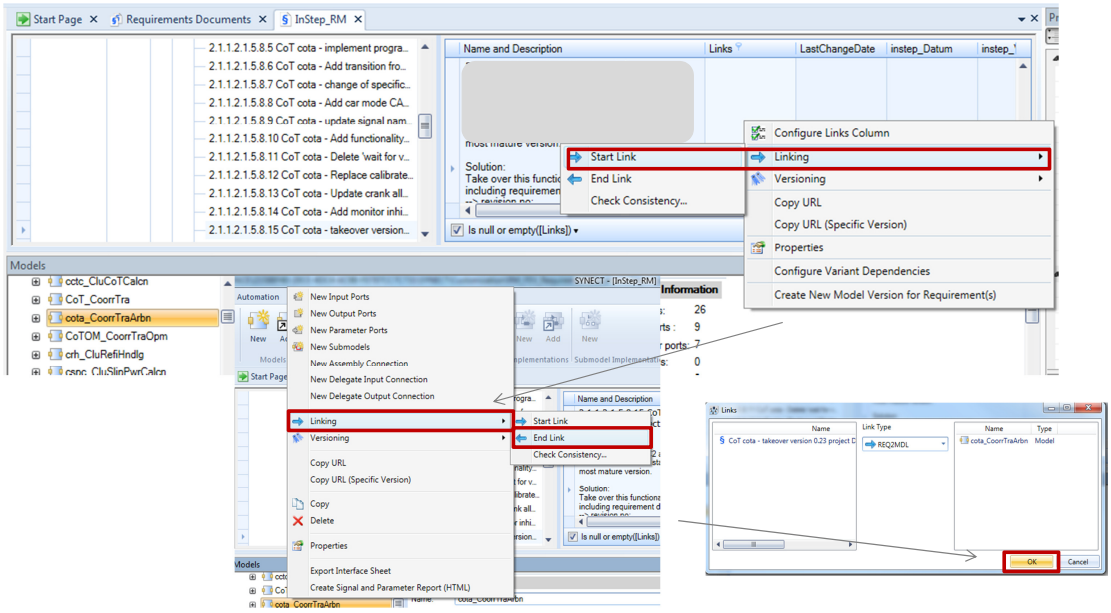


Abbildung 8.14: Manuelle Verbindung von Anforderung und Modell in SYNECT.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

ist eine automatisierte Verbindung noch nicht verlinkter Elemente leicht durchzuführen.

Verbindungen innerhalb von SYNECT beziehen sich immer auf eine bestimmte Version bzw. auf die Versionen der Quelle und des Ziels. Dies ist notwendig, um während der Evolution der Software-Produktlinie die entstandenen Zusammenhänge archivieren zu können und um auf ältere Versionen zurückgreifen zu können (Anforderung 4).

Dabei werden aber innerhalb von SYNECT bei Erstellung einer neuen Version die zugehörigen Links nicht mitgezogen. Ist zum Beispiel die Anforderung *Req* in Version 1 mit dem Modell der Funktion *Fkt* in Version 1 verlinkt, so wird bei einer Synchronisation mit dem Anforderungsmanagementwerkzeug wie folgt vorgegangen: Ist die Anforderung *Req* seit der letzten Synchronisation geändert worden und befindet sich diese im Zustand *Release*, so wird eine neue Version 2 der Anforderung *Req* erstellt. Diese ist nun nicht mehr mit dem Modell *Fkt* verbunden. Normalerweise sollte aber eine Anpassung der Anforderung Einfluss auf das gleiche Modell nehmen. Dies muss nicht immer der Fall sein. Im Sinne eines Refactorings der Architektur könnte zum Beispiel auch eine neue Zuordnung gewünscht sein. Grundsätzlich muss aber bei einer neuen Version der Anforderung eine neue Verbindung gesetzt werden. Ist dabei das Modell aktuell im Zustand *Draft*, so kann die Verbindung direkt gezogen werden.

Grundsätzlich garantiert SYNECT aber, dass in dem Moment, in dem ein Element in den Zustand *Released* gesetzt wird auch alle Elemente, die das Element referenziert, in den Zustand *Released* gesetzt werden. Bezogen auf Abbildung 8.5 führt der Übergang einer Anforderung in den Zustand *Released* zur Überführung aller Modelle, SignalInterfaces, Signals, Typedefs und Enumerations, die direkt oder indirekt verbunden sind, in den Zustand *Released*. Gleichzeitig

führt auch der Release einer Implementierung zum Release aller direkt oder indirekt verbundenen Elemente.

Dieser Mechanismus ist sehr hilfreich, da folgerichtig eine Schnittstelle zum Beispiel erst in den Zustand *Released* gesetzt werden kann, wenn die Definition ihrer Signale abgeschlossen ist. Eine Anforderung ist auch erst dann abgeschlossen, wenn die entsprechenden Modifikationen vorgenommen und modifizierte Elemente in den einen Zustand *Released* gesetzt wurden. Anstatt nun aber jedes Element einer Verbindungskette separat in den Zustand *Released* überführen zu müssen, erlaubt der beschriebene Mechanismus eine deutliche Zeitersparnis. Problematisch ist dabei allerdings, dass SYNECT vor der Überführung keine Übersicht aller direkt oder indirekt verbundenen Elemente auflistet. Um dieses Problem zu umgehen, wurde ein Bericht implementiert, der einem die Konsequenzen eines Releases vor dessen Durchführung aufzeigt. Auf diesen Punkt wird zu einem späteren Zeitpunkt noch einmal genauer eingegangen.

Für den aktuellen Kontext einer erneuten Verbindung zwischen Anforderung und Modell ist dieser Mechanismus relevant, da er dazu führt, dass ein Modell nach *release* einer Anforderung auch im Zustand *Released* sein sollte. Ist dies nicht der Fall, so müsste bereits eine andere Anforderung (oder ein zugehöriger CR) zu einer neuen Version des Modells geführt haben, die sich aktuell noch in der Umsetzung befindet. Um einzelne Änderungen in der Versionshistorie explizit nachweisen zu können, ist es sehr hilfreich, jede Änderung separat zu versionieren (Anforderung 9). Insofern sollte in diesem Falle keine Verbindung auf die aktuelle Version des Modells stattfinden, sondern stattdessen auf den Abschluss der aktuellen Version gewartet werden.

Um diesen Prozessschritt zu unterstützen, ist es möglich, wie in Abbildung 8.15 dargestellt, die Erstellung einer neuen Modellversion und die Verbindung von Anforderung zu Modell in einem Schritt durchzuführen. Dieser Schritt kann auch mit der eigentlichen Synchronisation direkt verbunden werden. Des Weiteren wird nicht nur, falls notwendig, eine neue Version des Modells, sondern auch der Implementierung erstellt und diese entsprechend verbunden (Schritt (5) in Abbildung 8.12). Die Implementierung referenziert dabei eine zugehörige URL eines zugrundeliegenden Versionsverwaltungswerkzeugs, wie zum Beispiel SVN oder CVS. Bei Erstellung einer neuen Version der Implementierung (in SYNECT) wird dabei die Referenz auf einen noch nicht existierenden *Tag* der aktuellen (noch zu implementierenden) Version gesetzt. Ein späterer Release der Implementierung ist dabei dann nur möglich, wenn die Referenz gültig ist und damit eine Implementierung existiert.

In Kombination mit einer automatischen Verbindung auf Basis eines Schlüsselwortes kann so der teilweise sehr zeitintensive Schritt einer manuellen Verbindung fast vollkommen vermieden werden. Da die entsprechenden Mechanismen im Log detaillierte Informationen über vollzogene Schritte auflisten, ist es für Einzelfälle möglich, ungewollte Schritte wieder zu revidieren.

Auch das Erstellen einer neuen Funktion oder Software-Komponente (Abbildung 8.15) und die anschließende Verbindung kann auf Basis verwendeter Schlüsselwörter automatisiert werden.

8.1.3 Schnittstellendefinition

Ein Modell kann eine beliebige Menge an Schnittstellen referenzieren, die wiederum eine beliebige Menge an Signalen referenziert. Bei der Abarbeitung einer Anforderung, eines CRs oder

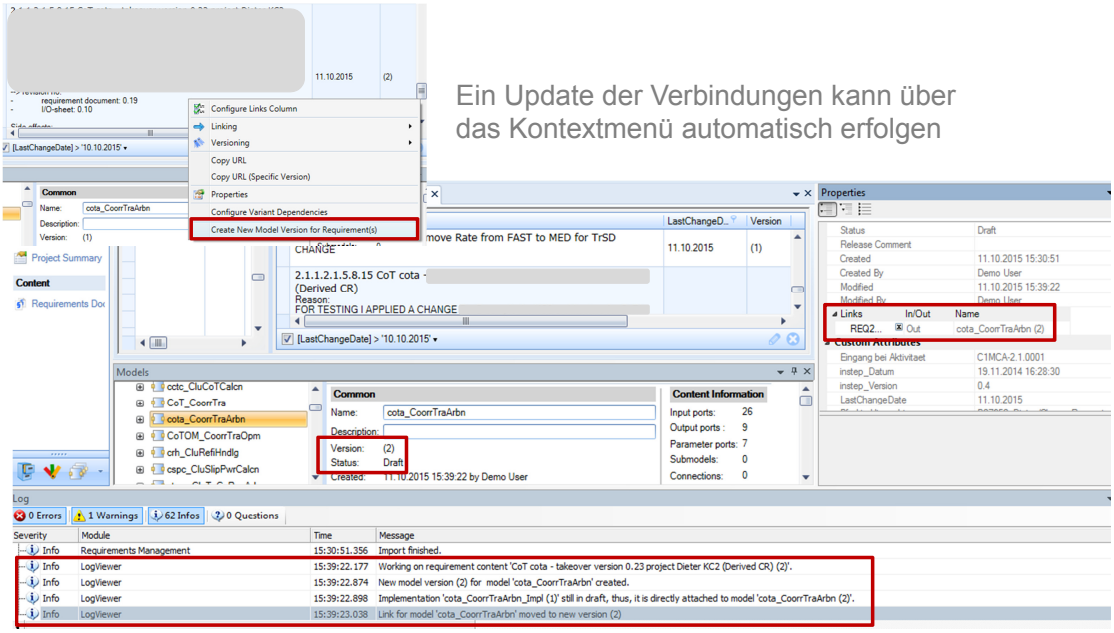


Abbildung 8.15: Automatisierte Verbindung von Elementen neuerer Versionen in SYNECT. (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

eines Bugs kann nun eine Schnittstellenänderung (bzw. Neuerstellung) nötig sein. Neue Schnittstellen und Signale müssen zuerst erstellt werden, bevor diese referenziert werden können. In Abbildung 8.16 sind dabei die notwendigen Schritte ausgehend von entweder neu erstellten Modellen oder einer neuen Modellversion dargestellt.

Beim Anlegen eines Elementes kann in SYNECT dabei immer direkt auf mögliche Referenzen zugegriffen werden. Ungültige Referenzen sind nicht möglich. In Abbildung 8.17 ist zum Beispiel dargestellt, wie für ein neu angelegtes Signal eine entsprechende Enumeration ausgewählt wird. Dabei kann nur auf schon definierte Enumerations in der aktuell neuesten Version zugegriffen werden. Dieser Mechanismus erleichtert die Anpassung und Erstellung einzelner Elemente und unterstützt deutlich den Erhalt und die Weiterentwicklung einer Software-Produktlinie. Ein schneller Zugriff auf gegebene Stände ist jederzeit kontextbezogen möglich und somit einfacher als die Erstellung neuer Elemente. Des Weiteren wird ein direkter Zugriff auf ältere Stände vermieden, aber nicht verhindert, so dass auch versionsbedingte Konflikte gelöst werden können.

Wie schon in Unterabschnitt 8.1.2 detailliert beschrieben, sind alle Referenzen auf explizite Versionen bezogen. In Konsequenz wird eine Änderung eines Signals (als neue Version) nicht direkt durch die entsprechenden Schnittstellen und Modelle übernommen. Eine direkte Übernahme jeglicher Änderungen in allen Schnittstellen wäre auch fatal, da ältere Stände und Software-Produkte nicht jede Änderung direkt übernehmen können oder jemals übernehmen

werden (Anforderung 4). Gleichzeitig kann dieser Mechanismus aber auch die tägliche Arbeit erschweren, da bei jeder Signaländerung die Referenzen der Schnittstellen und anschließend (bei Versionsänderung) auch die Referenzen des Modells manuell angepasst werden müssen (eine Verbesserung ist auch hier ab SYNECT 2.3 gegeben). Des Weiteren ist dies auch eine deutliche Fehlerquelle, falls einer der genannten Zwischenschritte vergessen wird. Insofern ist es auch hier wieder notwendig, durch eine automatische Abarbeitung der genannten Schritte den Arbeitsprozess zu verbessern. Eine Synchronisation der Versionen und ihrer Referenzen ist dabei aus zwei Richtungen möglich. Entweder können ausgehend vom Signal die Schnittstellen und folgend die Modelle auf den neuesten Stand gebracht werden oder umgekehrt ausgehend vom Modell die Schnittstellen und Signale. Beispielhaft ist dies in Abbildung 8.18 dargestellt.

Eine Synchronisation aller Modelle und Schnittstellen kann dabei vorher durch eine Abhängigkeitsanalyse abgesichert werden, um alle Konsequenzen frühzeitig zu erkennen.

8.1.4 Implementierung

Eine Implementierung bezieht sich immer auf ein Modell und beinhaltet deren Umsetzung. Dabei bildet SYNECT nicht ab, mit welchen Technologien die Implementierung umgesetzt wird,

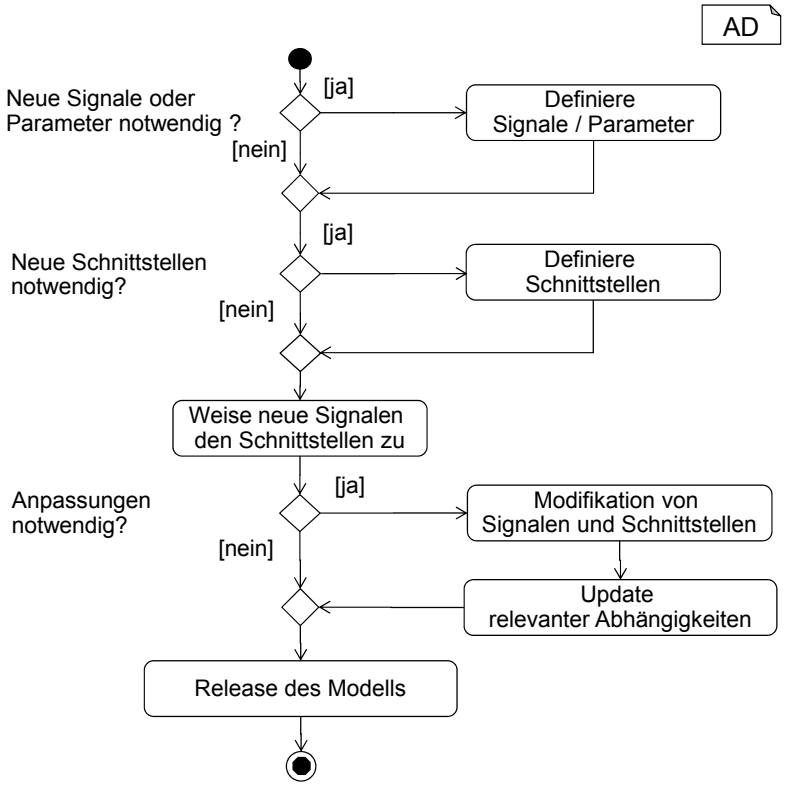


Abbildung 8.16: Arbeitsschritte zur Anpassung der Schnittstelle in SYNECT.

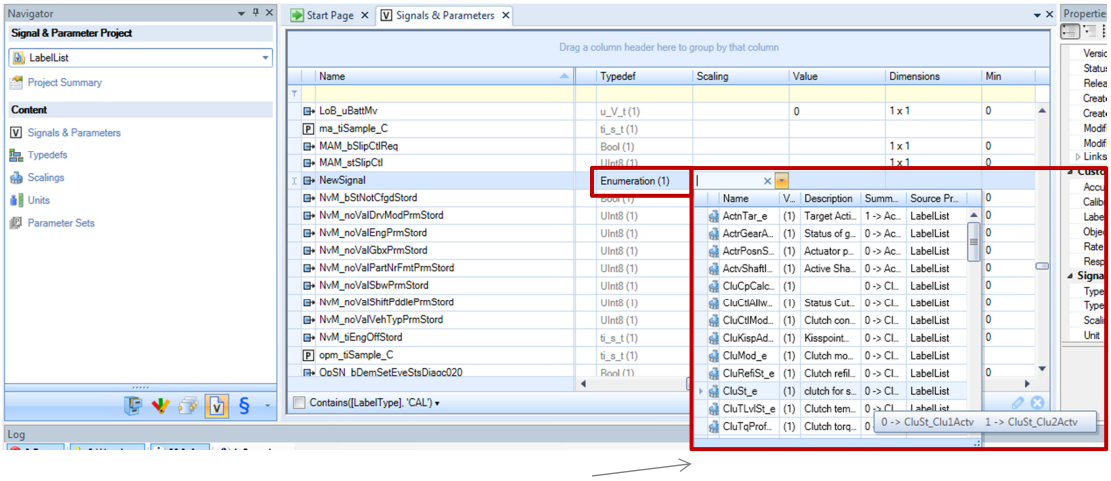


Abbildung 8.17: Definition einzelner Signale in Synect.

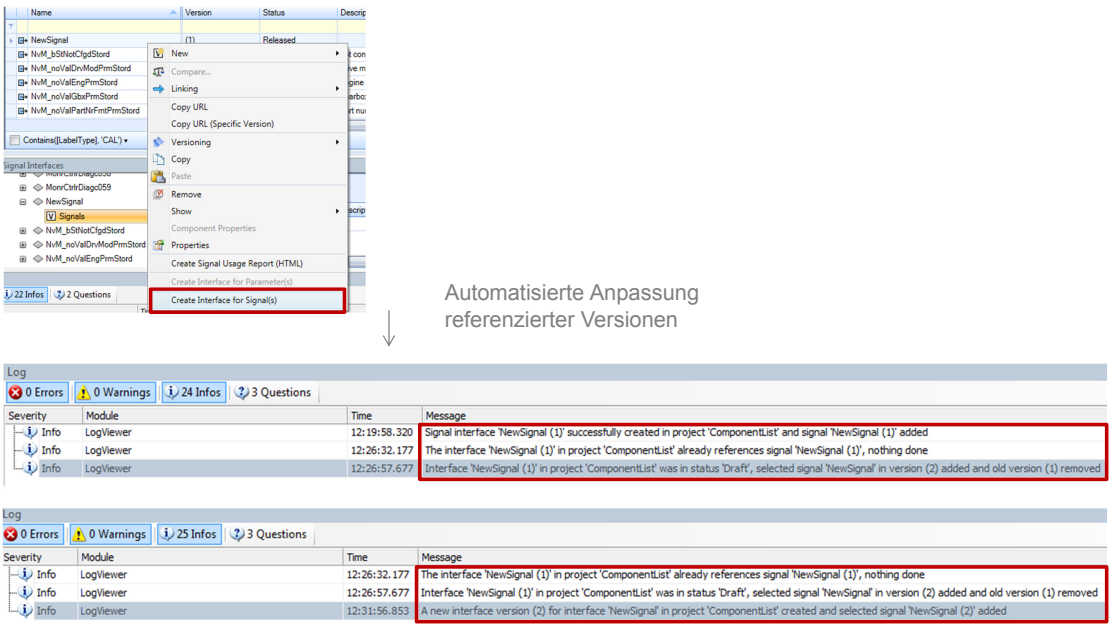


Abbildung 8.18: Versionsanpassung abhängiger Schnittstellen in Synect.

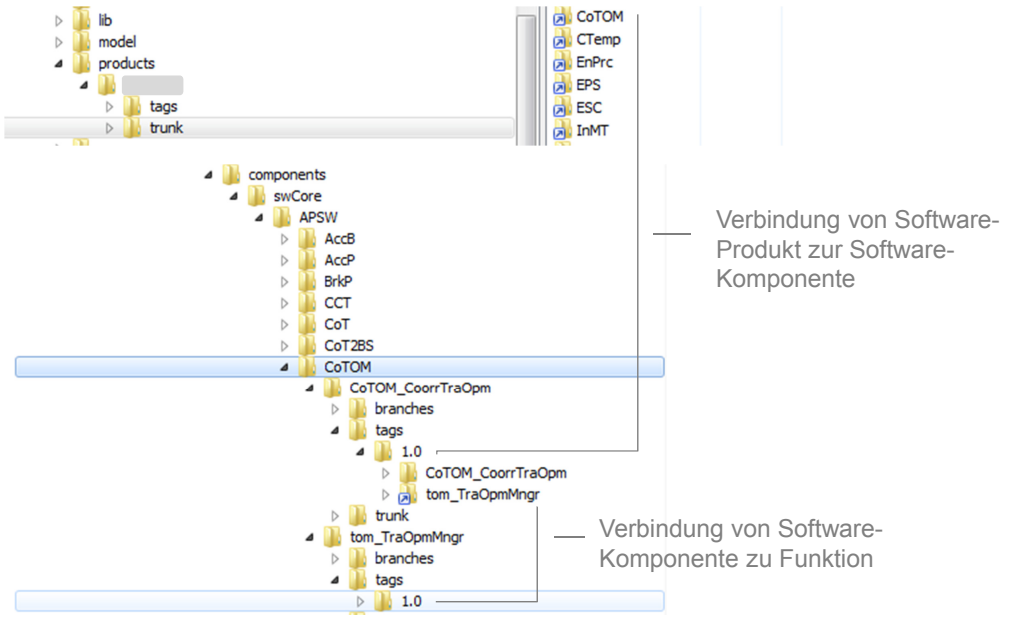


Abbildung 8.19: Kompositionale Variabilitätsmechanismus umgesetzt durch *External*s im SVN. (Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

sondern referenziert nur die zugehörige Ablageadresse. Wie schon in Unterabschnitt 8.1.2 beschrieben, wird parallel zur neuen Modellversion auch eine neue Implementierungsversion angelegt (wenn im Zustand *Draft*). Sind die Anpassungen an der Schnittstelle durchgeführt, so kann das Modell released werden, um anschließend die Umsetzung durchzuführen. Der Industriepartner verwendet SVN als Versionsverwaltungswerkzeug und für Funktionen und Komponenten werden jeweils eine separate Verzeichnisstruktur (*Trunk*, *Tags*, *Branches*) angelegt. Dies erlaubt bei der Umsetzung eines kompositionalen Variabilitätsmechanismuses eine explizite Versionierung (*Tag*) und Parallelentwicklung (*Branches*) jeder einzelnen Funktion und deren Kompositionen (Software-Komponente).

Eine Komposition von Funktionen wird SVN-seitig dabei durch *External*s umgesetzt: Innerhalb eines Verzeichnisses der Software-Komponente sind alle möglichen Funktionen der Software-Komponente und alle Software-Komponentenvarianten aufgeführt. Das Verzeichnis der Software-Komponentenvariante referenziert dabei wiederum mittels *External* die zugehörigen Funktionen, die in der aktuellen Software-Komponentenvariante genutzt werden. Bei festen Versionsständen (*Tag*) einer Software-Komponentenvariante werden dabei auch wiederum nur feste Versionsstände einer Funktion referenziert.

Beispielhaft ist die beschriebene SVN-Struktur in Abbildung 8.19 dargestellt. Dort ist eine Funktion (*tom_TraOpmMngr*) und eine Software-Komponentenvariante (*COTOM_CoorrTraOpm* (Variantenname)) der Software-Komponente *COTOM* (Domänenname) dargestellt: Die Version 1.0 der Software-Komponentenvariante referenziert dabei die Version 1.0 der Funktion. Des Weiteren ist dort auch dargestellt, wie in einem nächsten Schritt durch Komposition

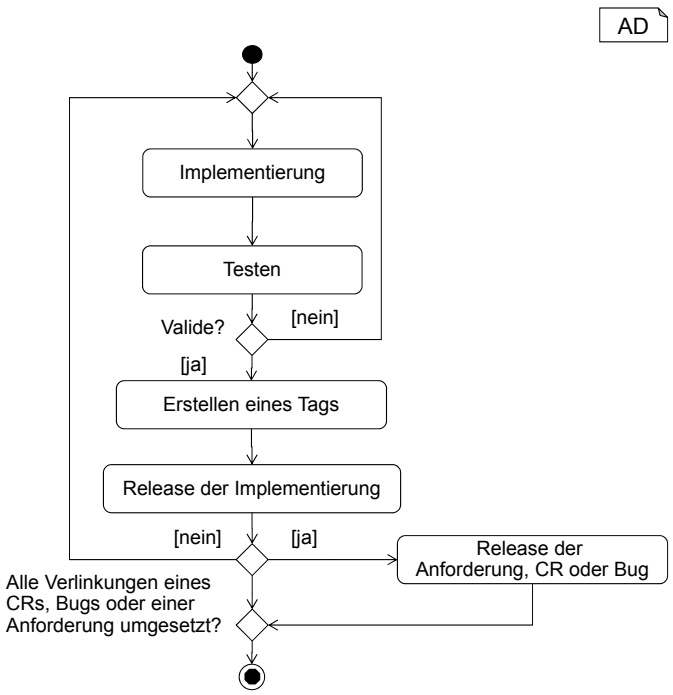


Abbildung 8.20: Arbeittschritte zur Implementierung, Test und Abschluss der Anforderung in SYNECT.

von Software-Komponentenvarianten ein Software-Produkt (*Interner Projektname*) komponiert wird. Dieses befindet sich im aktuellen Beispiel noch in der Entwicklung (*trunk*).

Derselbe Zusammenhang ist in SYNECT abgebildet: Jede Implementierung referenziert bei Erstellung einer neuen Version direkt den voraussichtlichen *Tag*, der durch die Versionsnummer in SYNECT vorgegeben ist. Nach Abschluss der Implementierung (und Testdurchführung) kann der Entwickler im SVN einen *Tag* entsprechend der Versionsnummer anlegen. Damit ist die Implementierung abgeschlossen und kann innerhalb von SYNECT released werden. Ist dagegen die *URL* nicht existent, es wurde also noch kein entsprechender *Tag* gebildet, so ist der Release der Implementierung in SYNECT auch nicht möglich.

Gleichzeitig werden in SVN dieselben Referenzen zwischen Modellen abgebildet, wie sie auch in SYNECT dargestellt werden. Ein Skript erlaubt dabei die Konfiguration der *External*s im SVN auf Basis der Hierarchiebildung in SYNECT.

Nach Abschluss der Implementierung können auch die zugehörigen Anforderungen, Bugs oder CRs in SYNECT released werden, falls alle Abhängigkeiten umgesetzt worden sind. Anschließend kann im Anforderungsmanagementwerkzeug noch der zugehörige Status angepasst werden (z.B. *implementiert*). Auch dieser Schritt lässt sich während der Synchronisation zwischen SYNECT und dem Anforderungswerkzeug automatisieren. Die zugehörigen Prozessschritte sind in Abbildung 8.20 dargestellt.

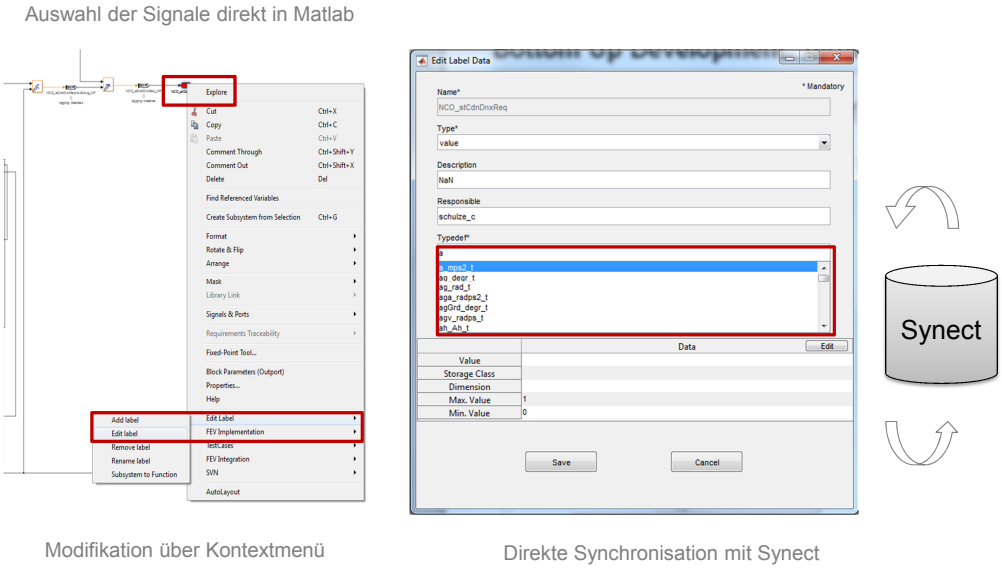


Abbildung 8.21: Direkter Zugriff auf die SYNECT-Datenbank in Simulink.

Synchronisation mit Matlab Simulink

Eines der meist genutzten Werkzeuge zur Verhaltensdefinition in der Automobilindustrie ist Matlab Simulink. Während SYNECT eine gute Übersicht über die gesamte Software-Produktlinie, ihre Funktionen, ihre Software-Komponenten, Signale und Schnittstellen sowie deren Zusammenhänge zu den Anforderungen und Varianten ermöglicht, spielen diese Aspekte während der konkreten Umsetzung eine untergeordnete Rolle.

Im Interesse der Software-Produktlinie ist es wichtig, dem Entwickler alle notwendigen Informationen direkt in seiner gewohnten Entwicklungsumgebung zur Verfügung zu stellen. Jeglicher zusätzlicher Arbeitsschritt und jegliche zusätzliche Lizenzabhängigkeit verringern die Wahrscheinlichkeit, dass Informationen, die durch die Software-Plattform gegeben sind, auch vom Entwickler aufgegriffen und wiederverwendet werden.

In Konsequenz wurde ein Plugin für Simulink entwickelt, das es ermöglicht, direkt auf Signal-, Typedef- und Enumeration-Definitionen aus SYNECT zuzugreifen und diese in Simulink direkt zu verwenden. Auch ist es möglich, neue Signale zu definieren, wie in Abbildung 8.21 dargestellt.

Offline-Zugang

Außerdem ist es möglich, eine HTML-basierte Darstellung (siehe Abbildung 8.22) des Datenstands aus SYNECT auszuleiten und somit einen Offline-Zugriff zu ermöglichen. Diese Darstellung ermöglicht sowohl eine identische Navigation durch die Sichten, die auch in SYNECT möglich wären als auch einen direkten Zugriff auf den entsprechenden Eintrag in SYNECT. Änderungen können aber auf diese Weise nicht durchgeführt werden. Auch findet in aktueller

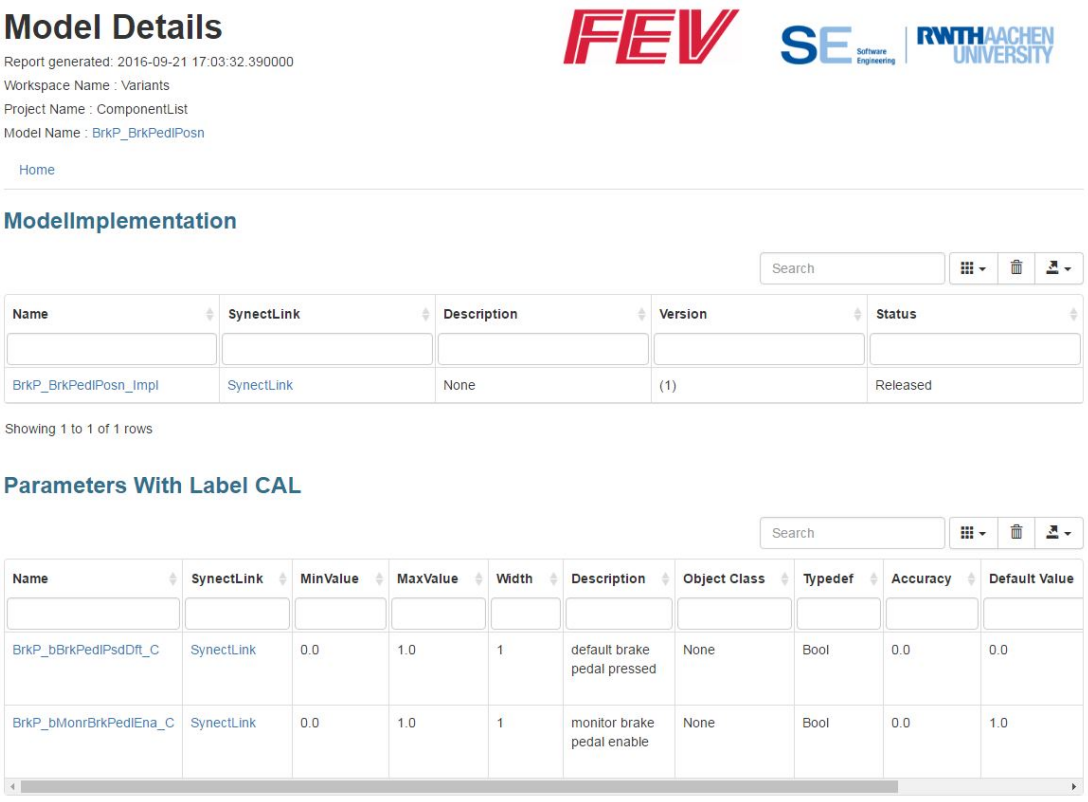
Form kein Update der Daten bei Abruf der Information statt. Stattdessen wird der entsprechende Bericht bei Nachfrage für den gesamten Datenbestand generiert.

8.1.5 Variantenkonfiguration

Der bisher beschriebene Prozess etabliert eine Verbinkung der Anforderungen mit Funktionen und Software-Komponenten und in einem weiteren Schritt mit deren Implementierungen. Bis auf die Einteilung in Software-Komponenten und Funktionen und die mögliche Bildung von Software-Komponentenvarianten wurde aber noch nicht genauer auf das Variabilitätsmodell und deren Verbindung zu den bisher beschriebenen Elementen eingegangen, wie in Abbildung 8.23 aufgezeigt.

Wie in Abbildung 8.2 aufgeführt und in Abschnitt 3.3 beschrieben, ist dabei eine Verbindung zwischen dem Variabilitätsmodell und den Funktionen sowie spezifischen Parametrisierungen von Funktionen notwendig. Diese Verbindung soll dabei über die Anforderungen erfolgen.

Insofern kann nach Synchronisation der Anforderungen mit dem Anforderungsmanagement-werkzeug in einem Folgeschritt direkt identifiziert werden, inwiefern durch neue oder geänderte Anforderungen das Variabilitätsmodell oder deren Verbindungen zu den Anforderungen ange-



Model Details

Report generated: 2016-09-21 17:03:32.390000
Workspace Name : Variants
Project Name : ComponentList
Model Name : BrkP_BrkPedlPosn

Home

ModelImplementation

Search

Name	SynectLink	Description	Version	Status
BrkP_BrkPedlPosn_Impl	SynectLink	None	(1)	Released

Showing 1 to 1 of 1 rows

Parameters With Label CAL

Search

Name	SynectLink	MinValue	MaxValue	Width	Description	Object Class	Typedef	Accuracy	Default Value
BrkP_bBrkPedlPsdDft_C	SynectLink	0.0	1.0	1	default brake pedal pressed	None	Bool	0.0	0.0
BrkP_bMonrBrkPedlEna_C	SynectLink	0.0	1.0	1	monitor brake pedal enable	None	Bool	0.0	1.0

Abbildung 8.22: HTML-basierter Zugriff auf Synectdatenbestand.

passt werden müssen. Dies kann meist direkt in Absprache mit dem Anforderungsspezifizierer abgestimmt werden.

Wie in Abbildung 8.24 dargestellt, kann in einem zweiten Schritt die etablierte oder veränderte Verbindung zwischen Variabilitätsmodell und Anforderung mit dem Anforderungswerkzeug (automatisiert) synchronisiert werden. In diesem Fall wird im Anforderungswerkzeug für jede Anforderung hinterlegt, in welchen Software-Produkten diese umgesetzt werden. Auf das entsprechende Attribut kann im Anforderungsmanagementwerkzeug nur lesend zugegriffen werden.

Das *Variant Management* Module in SYNECT besteht dabei aus primär zwei Sichten, die in Abbildung 8.25 dargestellt sind: dem Variabilitätsmodell und der Variantenkonfiguration.

Das Variabilitätsmodell beschreibt alle Variationspunkte und deren mögliche Variante, wobei für jeden Variationspunkt nur eine Variante ausgewählt werden kann. Außerdem können Varianten hierarchisch angeordnet werden, wobei die Auswahl einer höher angeordneten Varianten die unteren impliziert.

Weiterhin ist es möglich Varianteneinschränkungen durch aussagenlogische Formeln zu definieren. Auf Grund der Varianteneinschränkungen ist es möglich, Variabilitätsmodelle von SYNECT in Featuremodelle oder orthogonale Variablitätsmodelle zu transformieren. Entsprechende Variabilitätsmodelle sind aber nicht gut lesbar, da Aspekte wie beliebige Auswahlen oder optionale Auswahlen umständlich dargestellt werden müssen. Weiterführende Mechanismen, wie zum Beispiel Merkmalattribute oder Multi-Merkmale [CSHL13] werden dagegen nicht unterstützt.

Über das Kontextmenü einer Anforderung ist es dann in SYNECT möglich, eine Verbindung zwischen einer Variante und einer Anforderung herzustellen (siehe Abbildung 8.26). SYNECT definiert dabei noch eine Zwischenelement, die Variantenabhängigkeit. Diese erlaubt die Auswahl mehrerer Varianten unterschiedlicher Variationspunkte zu bündeln.

Des Weiteren ist es auch möglich, eine Parametrisierung einer Funktionsvariante mit einer Variante zu verbinden. Dies ermöglicht dann eine variantenbezogene Parametrisierung einzelner

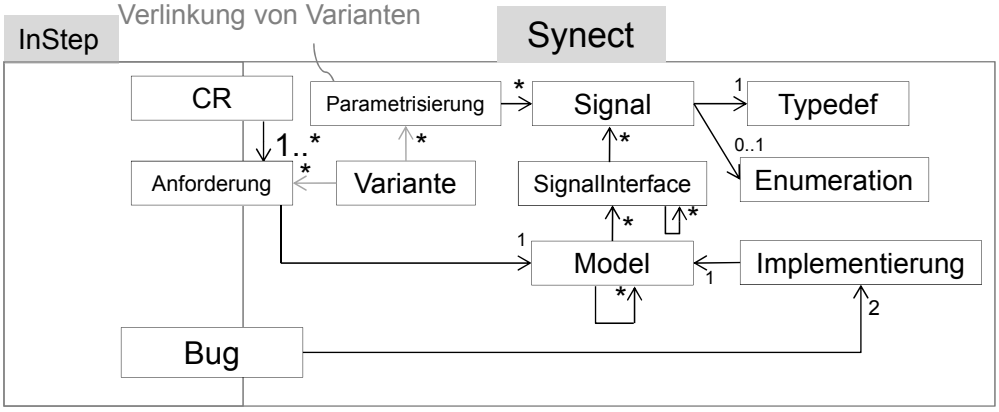


Abbildung 8.23: Verbindung des Variabilitätsmodells zu Anforderungen und Parametern in SYNECT.

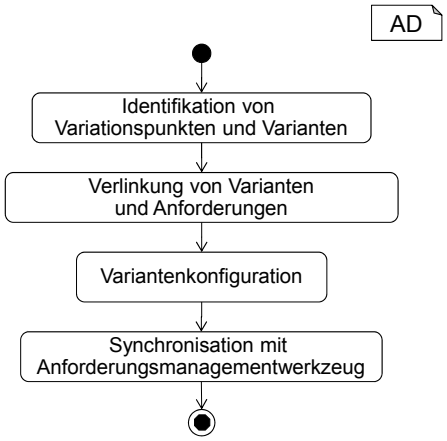


Abbildung 8.24: Anpassungen der Variantenkonfiguration in SYNECT.

Funktionen, wie in Abschnitt 3.3 gefordert.

Ist das Variabilitätsmodell definiert, so kann in einem Folgeschritt auf Basis dieses Modells eine Variantenkonfiguration definiert werden. Dazu muss für jeden Variationspunkt eine Variante ausgewählt werden. Ist die Auswahl auf Grund von Varianteneinschränkungen ungültig, so wird dies angezeigt. Im Beispiel aus Abbildung 8.25 ist der Variationspunkt *HybridActiveSync* und die Variante *On* markiert. Dieser ist im Software-Produkt *SPA* in der Variante *On* gegeben,

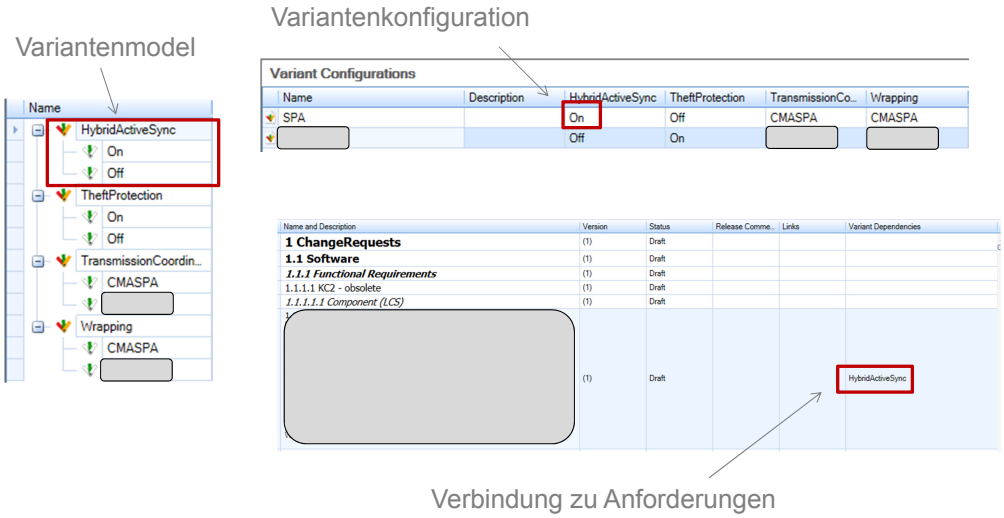


Abbildung 8.25: Variablitätsmodell und Variantenkonfiguration, sowie deren Verbindung von Varianten zu Anforderungen in SYNECT.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

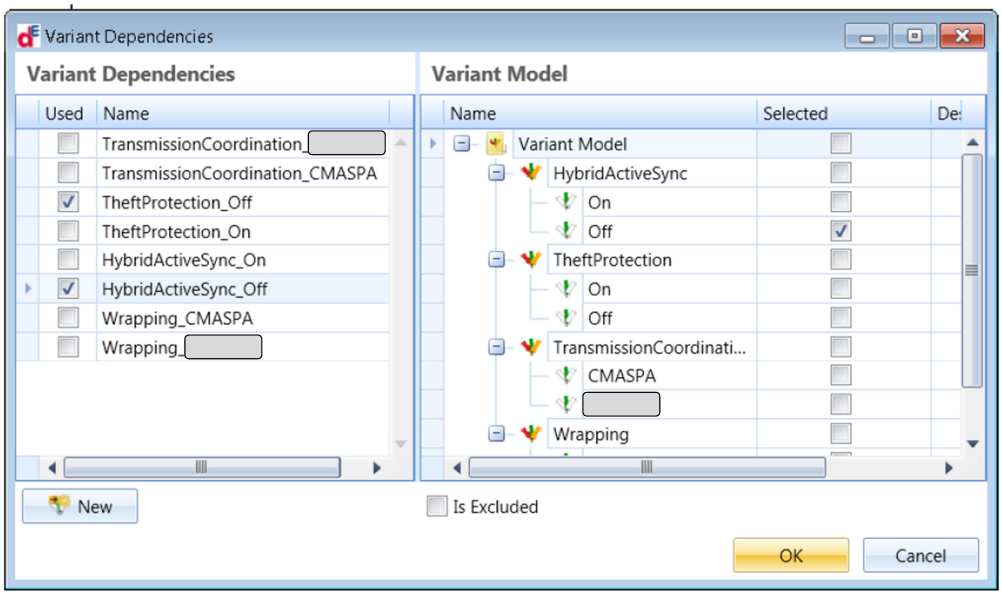


Abbildung 8.26: Auswahl einer Variantenabhängigkeit in SYNECT.
(Sensible Daten von beteiligten Projektpartnern wurden grau überdeckt.)

was dazu führt, dass die angezeigte Anforderung und alle indirekt verbundenen Elemente zur Komposition des Software-Produktes gehören.

Nach Festlegung der Varianten-Abhängigkeit und nach Definition der Varianten wird außerdem berechnet, welche Anforderung welcher Variante zugehörig ist. Dabei gehört eine Anforderung zu einer Variante, falls diese gar keine Variantenabhängigkeiten definiert hat oder die Abhängigkeit durch die Konfiguration der Variante erfüllt ist. Die Zugehörigkeiten werden als separates Attribut festgehalten und während der Synchronisation mit dem Anforderungswerkzeug übertragen.

8.1.6 Komposition eines Software-Produktes

Auf Basis einer Variantenkonfiguration und einer Verbindung der Anforderungen mit dem Variabilitätsmodell und indirekt mit den Implementierungen kann jederzeit durch Auswahl einer Variantenkonfiguration ein Software-Produkt komponiert werden.

Dieses Software-Produkt wird in einem separaten Model- und Signal-Management-Projekt ausgeleitet und beinhaltet entsprechende Modelle, Schnittstellen und Signale, die direkt oder indirekt mit den Varianten der Variantenkonfiguration verbunden sind. Alle Modelle, Schnittstellen und Signale bilden dabei nur Referenzen auf das Ursprungsprojekt (die Software-Plattform) und sind als *Shared-Items* gekennzeichnet.

Alle *Shared-Items* referenzieren dabei immer die Version zum Zeitpunkt der Komposition und eine Modifikation der Referenz ist nur in der Software-Plattform möglich. Auch können *Shared-Items* nur von Elementen im Zustand *Released* erstellt werden. Während der Ausleitung

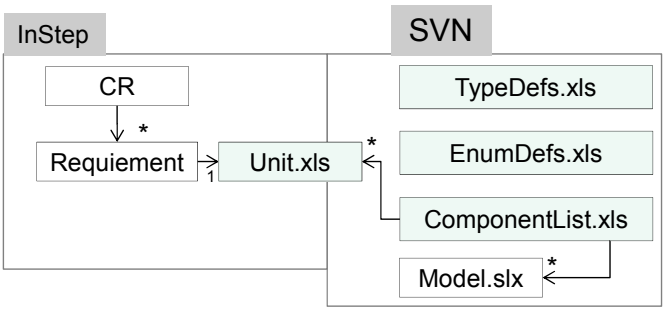


Abbildung 8.27: Entwicklungsartefakte des Industriepartners vor der Einführung von SYNECT.

wird für jedes Element, das ausgeleitet werden soll sich aber aktuell noch im Zustand *Draft* befindet, eine entsprechende Warnung im Log eingetragen. So ist es leicht möglich, noch offene Arbeitsschritte zu identifizieren und zu beenden.

Es ist außerdem jederzeit möglich, ein komponiertes Software-Produkt auf den neuesten Stand zu bringen. Dabei werden neue Elemente hinzugefügt und alte auf den neuesten Stand gebracht. Der neueste Stand spiegelt dabei die neueste Version wieder, die im Zustand *Released* zur Verfügung steht.

Neben den referenzierten Elementen der Software-Plattform ist es nun möglich, weitere Elemente im projekt-spezifischen Model- und Signal-Management-Projekt hinzuzufügen. So ist eine Trennung zwischen Domänen- und Anwendungsartefakten jederzeit möglich (Anforderung 1), ohne dass eine Gesamtsicht auf die Produktlinie oder das Software-Produkt verloren geht.

8.2 Implementierung

Die zugrundeliegende Ausgangslage beim Industriepartner vor Anpassung des Softwareproduktlinienprozesses (PERSIST) zur Verwendung einer Datenbank gab eine excelbasierte Definition von Schnittstellen, Signalen, Typedefs und Aufzählungen vor, wie in Abbildung 8.27 dargestellt.

Es existierten separate Exceldateien für die Schnittstelle jeder Funktion oder Software-Komponentenvariante, eine separate Liste aller Software-Komponentenvarianten und Funktionen in einem Software-Produkt (*ComponenteList.xls*) sowie eine Auflistung aller Typedefs und eine Auflistung aller Enumerationen.

Dabei wurden allerdings nur Typedefs und Enumerationen explizit in unterschiedlichen Software-Produkten geteilt. PERSIST-spezifische Automatisierungen, wie zum Beispiel die Generierung von Simulink- oder AUTOSAR-Rahmenmodellen und zugehörigen Initialisierungen basieren auf diesen Artefakttypen.

Bei der Einführung von SYNECT war es notwendig, alle bisherigen Software-Produkte nach SYNECT überführen zu können und auch einen nahtlosen Übergang bei laufenden Software-Produkten zu ermöglichen. Dazu war es in einem ersten Schritt notwendig, die gegebenen Artefakttypen auf Repräsentaten innerhalb von SYNECT abzubilden und dann einen Importer zu implementieren. Um eine schrittweise Migration der beim Industriepartner gegebenen Werk-

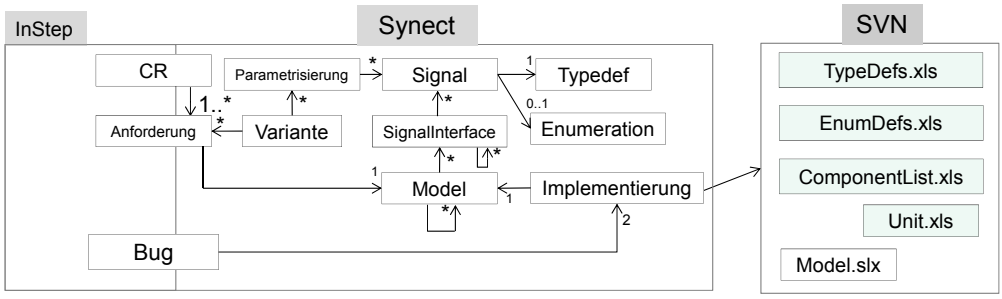


Abbildung 8.28: Entwicklungsartefakte beim Industriepartner mit SYNECT.

zeugkette zu ermöglichen, wurde auch ein Exporter umgesetzt, um weiterhin auch die alten Artefakttypen zur Verfügung zu stellen.

In Konsequenz wurde neben dem Importer auch ein Exporter umgesetzt, der den Datenbestand in SYNECT wiederum in die in Abbildung 8.27 aufgeführten Excelformate überführt. Dies hat den bedeutenden Vorteil, dass die Migration der Werkzeugkette anschließend schrittweise erfolgen kann und frühzeitig beliebige Software-Produkte nach SYNECT überführt werden können, da alle Abhängigkeiten zu alter Technologie weiterhin bedient werden. Außerdem erlaubt die automatisierte Durchführung von Import und Export mit anschließendem Vergleich der Ein- und Ausgaben eine genaue Validierung der Implementierung. Unter der Garantie, dass die aus SYNECT exportierten Daten exakt den importierten Daten entsprechen, ist es auch möglich, Projektleiter von der Migration laufender Software-Produkte zu überzeugen.

In Kombination mit dem Datenmodell aus Abbildung 8.5 ergibt sich somit das Gesamtbild aus Abbildung 8.28.

Die Implementierung innerhalb von SYNECT verweist dabei auf ein SVN-Verzeichnis, das sowohl die Implementierung (Simulinkmodell, Code etc.), als auch die Excel-Generate enthält, die zur weiteren Bearbeitung benötigt werden.

Die Firma dSPACE bietet für SYNECT 1.5 sowohl eine Python- als auch eine COM-Schnittstelle zur Implementierung eigener Funktionalitäten an. Abbildung 8.29 gibt dabei eine Übersicht über alle realisierten Komponenten.

- **Importer** Import PERSIST-bezogener Artefakte (Schnittstellen, Enumerationen, Typdefinitionen, Komponentenliste, Signalliste) eines Software-Produktes in die SYNECT-Datenbank.

- **Exporter** Export der SYNECT-Daten eines Software-Produktes in PERSIST-bezogene Artefakte (Schnittstellen, Enumerationen, Typdefinitionen, Software-Komponentenliste, Signalliste, Software-Komponentenkonfiguration). Die Komponentenkonfiguration wird dabei als Zwischenformat genutzt, um auf ihrer Basis eine Software-Komponentenvariante *Externals* im SVN zu komponieren (Details siehe Abbildung 8.19).

- **Helper** Als Basis für weitere Erweiterungen der SYNECT-Basisfunktionalität wurden Hilfsklassen und Methoden definiert. Grundsätzlich wird das Sammeln spezifischer Da-

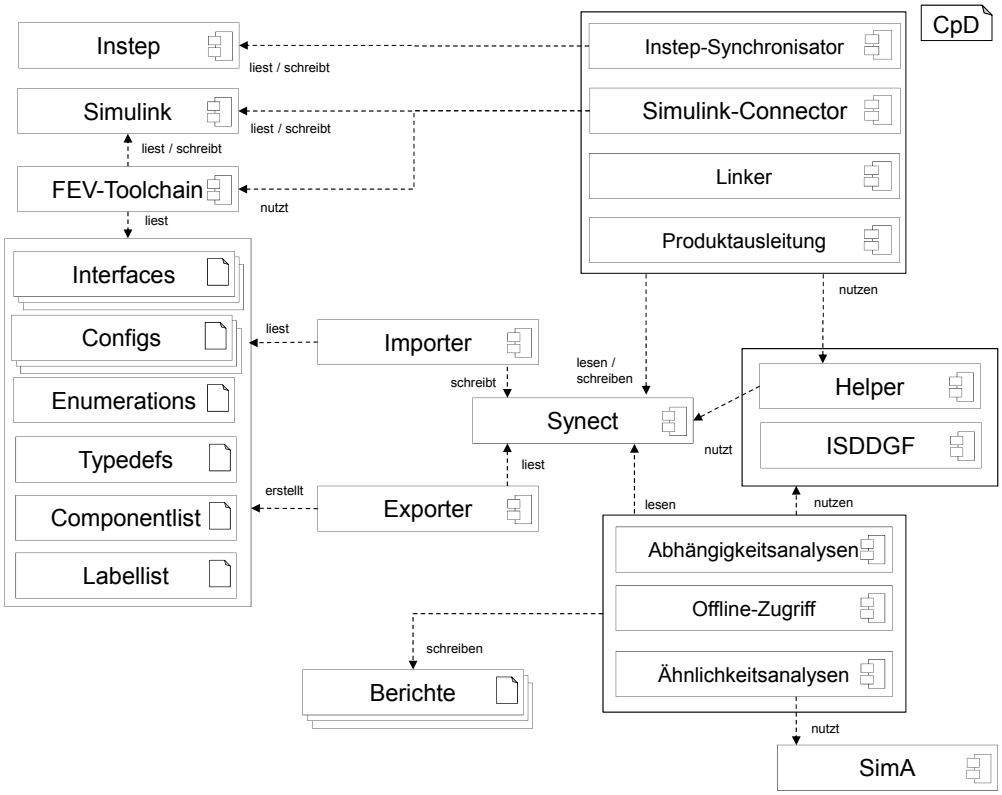


Abbildung 8.29: Übersicht über alle im Kontext der Industrieparnter-spezifischen Anpassungen umgesetzten Komponenten für SYNECT.

ten durch Kollektoren unterstützt und damit auch die Logik der Datensammlung von deren Bewertung und Modifikation getrennt. Neben speziellen Kollektoren bietet ein allgemeiner Kollektor mit der Identifikation von abhängigen Elementen bis zu einer definierbaren Tiefe und Richtung unter bestimmten Filterkriterien eine Basisfunktionalität, die durch SYNECT nicht angeboten wurde. Kollektoren geben dabei für jeden gefundenen Elementtypen eine Map zurück.

- **ISDDGF** Als weitere Basis wird zur Erstellung entsprechender Berichte das in Abschnitt 7.3 vorgestellte Framework verwendet. Dies erlaubt, die gesammelten Daten beziehungsweise die berechneten Ergebnisse in dynamisch konfigurierbaren Tabellen und Diagrammsichten darzulegen. Durch Links ist es dabei sowohl möglich einen komplexeren Bericht anzufertigen als auch direkten Zugriff auf einzelne Elemente in SYNECT zu ermöglichen. Das Framework wird von den Komponenten *Abhängigkeitsanalysen*, *Offline-Zugriff* und *Ähnlichkeitsanalysen* verwendet, um entsprechende Berichte zu generieren. Beispiele sind in Abbildung 8.22 sowie in Kapitel 7 gegeben.

- **Produktausleitung** Die Komponente wird verwendet, um auf Basis einer validen Vari-

antenkonfiguration und zugehörigen Verbindungen von Varianten zu Anforderungen (und indirekt zu Funktionen) und Parametrierungen ein Software-Produkt zu konfigurieren, wie in Unterabschnitt 8.1.6 beschrieben.

- **Linker** Die Komponente unterstützt das automatisierte Verbinden von Anforderungen zu Modellen und Implementierungen sowie die Anpassung von Versionen von beliebigen Elementen auf Basis einer veralteten versionierten Verbindung. Details zu dieser Funktionsweise sind in Unterabschnitt 8.1.2 und in Unterabschnitt 8.1.3 beschrieben.

- **Synect-Connector** Der *Synect-Connector* verbindet unter Verwendung eines durch den Industriepartner schon gegebenes Simulink-Plugin Simulink mit SYNECT. Dadurch ist es möglich, wie in Abbildung 8.21 dargestellt, direkt von Simulink aus auf SYNECT zuzugreifen.

- **Instep-Synchronisator** Der Instep-Synchronisator synchronisiert die Daten des Anforderungsmanagementwerkzeugs Instep mit SYNECT, wie in Unterabschnitt 8.1.1 beschrieben. Des Weiteren werden die bei der Variantenkonfiguration berechneten Beziehungen zwischen Software-Produkt und Anforderungen in das Anforderungsmanagementwerkzeug übertragen.

- **Abhängigkeitsanalysen** Unter Verwendung des Kollektors für referenzierten Elemente ermöglicht diese Komponente die Generierung von Berichten, wie zum Beispiel die Anzeige aller Funktionen die ein Signal benutzen oder die Anzeige aller referenzierten Elemente an, die nicht im Zustand *Released* sind.

- **Offline-Zugriff** Diese Komponente stellt die SYNECT-Sichten des Signal- und Modellmanagements unter Verwendung des *ISDDGF* Frameworks dar. Ein Beispiel ist in Abbildung 8.22 gegeben.

- **Ähnlichkeitsanalysen** Diese Komponente stellt unter Verwendung der Rahmenwerke *ISDDGF* und *SimA* Ähnlichkeitsanalysen von Signalen zur Verfügung. Diese Analyse erlaubt die Identifikation von Signalklonen innerhalb der Software-Plattform.

- **SimA** Das Rahmenwerk für Ähnlichkeitsanalysen ist in Kapitel 7 genau beschrieben.

Kapitel 9

Zusammenfassung und Ausblick

In dieser Arbeit wurde auf Basis der identifizierten aktuellen Herausforderungen in der Automobilindustrie (Kapitel 1) und einer intensiven Analyse aktuell in dieser Domäne vorherrschenden Entwicklungsnormen und -trends (Kapitel 2) die Notwendigkeit einer agilen Herangehensweise auch im Kontext einer Software-Produktlinienentwicklung herausgearbeitet. Um diese Notwendigkeit zu befriedigen wurde ein produktgetriebener Software-Produktlinienentwicklungsprozess definiert, der auf Basis automatisiert durchführbarer Ähnlichkeitsanalysen eine schrittweise Etablierung und Weiterentwicklung einer Software-Plattform ermöglicht (Kapitel 3).

Da die automatische Erkennung von Ähnlichkeiten einen notwendigen Bestandteil darstellt, um eine produktgetriebene Software-Produktlinienentwicklung effizient durchführen zu können, wurde in einem weiteren Schritt detailliert auf aktuell durch Forschung und Industrie zur Verfügung stehende Klonerkennungsverfahren eingegangen (Kapitel 4). Auf Basis dieser Grundlage wurden Verfahren zur extrinsischen (Abschnitt 5.1), schnittstellen-basierten (Abschnitt 5.2) und semantischen Ähnlichkeitsanalyse (Kapitel 6) ausgearbeitet, implementiert und evaluiert, um eine produktgetriebene Software-Produktlinienentwicklung zu unterstützen. Im Zuge der Evaluierung konnten die Vor- und Nachteile der beschriebenen Verfahren identifiziert und deren Nutzen im Kontext beteiligter Industriepartner bewertet werden. Dabei war es möglich die Grundlage zur Extraktion mehrerer Software-Komponenten zu legen und den möglichen Nutzen des Verfahrens somit zu verdeutlichen.

Im Anschluss wurde das Framework *SimA* vorgestellt (Kapitel 7), das durch eine adäquate Aufbereitung der durch Ähnlichkeitsanalysen erhobenen Daten eine zielorientierte Etablierung und Wartung der Software-Plattform ermöglicht. Auf Grund seiner modularen Struktur kann das Framework auch genutzt werden, um in Zukunft weitere Ähnlichkeitsanalysen und unterschiedliche Kontexte mit geringem Mehraufwand zu unterstützen.

Abschließend wurden Erweiterungen am Werkzeug SYNECT, das für eine datenbankgestützte reaktive Software-Produktlinienentwicklung beim Industriepartner Verwendung finden soll, und der daraus resultierende Entwicklungsprozess vorgestellt (Kapitel 8). Diese Erweiterungen ermöglichen eine direkte Integration in die gegebene Entwicklungsumgebung. Des Weiteren werden Arbeitsschritte durch Automatismen unterstützt, die für die durch Normen geforderte Nachverfolgbarkeit der Anforderungen, sowie einem etablierten Konfigurations-, Änderungs- und Releasemanagement nötig sind. Dieses Werkzeug trägt weiterhin zu einer effizienten Wartung und Erweiterung einer Software-Plattform bei und kann auch im Kontext einer Serienentwicklung eingesetzt werden.

Durch diese Schritte konnten für die in Kapitel 1 formulierten Forschungsfragen entsprechende mögliche Antworten gefunden werden.

- **RQ1**: *Wie kann effizient und parallel zum stark kosten- und zeitgetriebenen Projektalltag auf Basis eines implizit gegebenen Clone&Own-Ansatzes eine Software-Plattform erstellt und weitergeführt werden, ohne dass dabei signifikanter Mehraufwand generiert wird und aktuelle Standards eingehalten werden?*

  Der beschriebene reaktive Software-Produktlinienentwicklungsprozess ermöglicht eine automatisierte und insofern kostengünstige Analyse des zur Verfügung stehenden Portfolios. Basieren gegebene Software-Komponenten auf einem impliziten *Clone&Own*-Ansatz, so können Ähnlichkeiten zwischen ursprünglichen Klonen parallel zum Projektalltag identifiziert werden. Die durch Normen geforderte Nachverfolgbarkeit der Anforderungen über einzelne Abstraktions- und Entwicklungsebenen hinweg ist im Projektalltag in einem ersten Schritt ein notwendiges Übel, das zusätzlichen Aufwand erfordert. Wird das in der Software-Produktlinienentwicklung übliche Variantenmodell direkt mit entsprechenden Anforderungen verknüpft und erlaubt der beschriebenen Ansatz einen kompositionalen Aufbau von Varianten, so kann die notwendige Nachverfolgbarkeit direkt zur Ausleitung einzelner konkreter Varianten genutzt werden. Dies unterstützt im Weiteren auch das Konfigurationsmanagement, ohne dass neben der Verknüpfung von Variantenmodell und Anforderungen ein weiterer Arbeitsschritt notwendig wäre.

- **RQ2**: *Welche sprachlichen Mechanismen der gegebenen Werkzeuge innerhalb der Automobilindustrie können unter möglichst minimalen Modifikationen genutzt werden, um eine effiziente Entwicklung wiederverwendbarer Software-Komponenten zu ermöglichen?*

  Ein kompositionaler Ansatz zur Darstellung der Variabilität im Lösungsraum stellt nicht nur unter den für **RQ1** genannten Gründen einen Vorteil dar. Etablierte Werkzeuge innerhalb der Automobilindustrie unterstützen meist nur einen Referenzmechanismus, um einzelne Komponenten wiederzuverwenden. Die Darstellung in einem 150%-Modell führt bei größeren Modellen schnell zur Unübersichtlichkeit während transformationale Ansätze durch etablierte Werkzeuge und Prozesse keine Unterstützung finden. Da ein rein kompositionaler Ansatz bei kleineren Variationen wiederum bezüglich der Menge möglicher Komponenten und Kompositionen zur Unübersichtlichkeit führen, ist ein Kompromiss aus kompositionalem und annotativem Ansatz zielführend, da beide durch etablierte Werkzeuge und Prozesse unterstützt werden und eine Unübersichtlichkeit sowohl durch zu große Modelle als auch durch zu viele kleine Komponenten vermieden werden kann.

- **RQ3**: *Mit welchen automatisierten Mechanismen kann man einen solchen Prozess bezüglich Effizienz unterstützen?*

  Wie schon bezüglich **RQ1** erläutert, kann eine zum Tagesgeschäft parallele Analyse des gegebenen Portfolios nur durch ein automatisierte Ähnlichkeitsanalyse effizient durchgeführt werden. Dabei muss der Suchraum schrittweise reduziert werden, um auch aufwändigere Analysen in praktikabler Zeit durchführen zu können. Auf Basis der zur Verfügung stehenden Artefakte hat sich dabei eine schrittweise extrinsische, strukturelle und abschließend semantische Ähnlichkeitsanalyse als zielführend herausgestellt.

- **RQ4**: *Welche weiteren Werkzeuge sind notwendig, um eine Etablierung einer Software-Plattform zu unterstützen und deren Erhalt zu gewährleisten?*

  Um eine Etablierung, Wartung und Erweiterung einer Software-Produktlinie zu unterstützen ist es hilfreich einzelne Entwicklungsartefakte, wie zum Beispiel Signale, Funktionen oder Komponenten, möglichst frühzeitig zur Verfügung zu stellen und diese separat verwalten zu können. Des Weiteren ist es notwendig die in **RQ1** geforderte Nachverfolgbarkeit bis zum Variantenmodell durch Werkzeuge zu unterstützen. Für diesen Zweck wurde das durch das erweiterte Werkzeug SYNECT mögliche datenbankgestützte Varianten-, Modell- und Signalmanagement vorgestellt.

Der definierte produktgetriebene Software-Produktlinienentwicklungsprozess sowie die verwendeten Ähnlichkeitsanalysen ermöglichen Industriepartnern einen direkte Übersicht über ihr aktuelles Portfolio an Software-Komponentenvarianten sowie deren Potentiale zur Wiederverwendung. Durch die in dieser Arbeit durchgeführte Evaluierung wurden auch die Vor- und Nachteile der einzelnen Analyseverfahren aufgeführt. Diese stellen aber im Verhältnis zu einer längeren Erprobung in der Praxis erste Erkenntnisse da, so dass die genutzten Verfahren über Zeit auf Basis weiterer identifizierter Typen von falsch positiven oder falsch negativen Ergebnissen verbessert werden können. Auch eine Einbindung weiterer Analyseverfahren verspricht eine erhöhte Genauigkeit.

Inwiefern der definierte Kompromiss aus kompositionalem und annotativem Ansatz als Variabilitätsmechanismus ausreichend ist oder für Spezialfälle weitere Anpassungen wünschenswert sind, kann wiederum auch nur durch intensive Anwendung des Verfahrens evaluiert werden. Auch ist aktuell kein Verfahren implementiert, der eine Generierung einer Funktionsvariante auf Basis einer Parametrisierung durchführt. Dies kann aber unter geringerem Aufwand nachgezogen werden.

Die Evaluierung der CEGAS zeigte noch Schwächen bezüglich einzelner Unterschiede innerhalb einer Verzweigung auf. Da dort nicht direkt eine syntaktische Bewertung ausreichend ist, ist eine entsprechende Erweiterung nicht trivial. Die semantische Ähnlichkeitsanalyse einzelner diskreter Funktionen und deren schrittweisen Abstraktion ist somit ein offener Punkt, dessen Lösung das Verfahren deutlich verbessern würde. Auch eine möglichst effiziente Identifikation des Startzustandes einer n-begrenzten Analyse stellt eine mögliche Verbesserung dar.

Die syntaktische Analyse auf Basis des extrahierten I/O-EFAs führte schon zu guten Ergebnisse, die durch eine Normalisierung der analysierten Bedingungen und Ausgabefunktionen noch weiter verbessert werden können.

Während es die beschriebene schnittstellen-basierte Ähnlichkeitsanalyse auf Basis der Repräsentanten identifizierter Mengen ähnlicher Datenelemente direkt erlaubt Schnittstellen für Funktionen zu generieren, ist es auf Basis der semantischen Ähnlichkeitsanalyse auf Grund der unterschiedlichen möglichen Quellartefakte und auf Grund der separaten Analyse einzelner Ausgabesignale aktuell nicht möglich direkt Verhaltensspezifikationen für Funktionen auszuleiten. Erfahrungen bei der manuellen Implementierung von Software-Komponenten auf Basis der gegebenen semantischen Analysen können dabei hilfreiche Informationen liefern um diesen Transformationsschritt in Zukunft soweit wie möglich zu automatisieren.

Das durch SYNECT (Version 1.5) zur Verfügung gestellte Variabiltätsmodell stellt im Vergleich zu Merkmalmodellen eine umständlichere Möglichkeit dar um optionale, sich ausschließende oder notwendige Variationspunkte zu definieren. Auch an dieser Stelle gibt es Verbesserungspotential, da diese Arbeit das Themengebiet der Variabilitätsmodelle nur ansatzweise behandelt.

Zusammenfassend ermöglicht der beschriebene Ansatz eine schrittweise Etablierung einer Software-Produktlinie unter Beachtung der in der Automobilindustrie vorherrschenden Normen und Trends und unter Berücksichtigung aktueller Einflüsse der agilen Entwicklung. In Konsequenz kann in einem ersten Schritt sowohl das Tagesgeschäft auf Basis fundierter Analysen durch ein systematisches *Clone&Own* unterstützt als auch sukzessive eine Software-Plattform etabliert werden. Die parallele Unterstützung durch das Werkzeug SYNECT und deren Anbindung an Matlab Simulink erlaubt weiterhin einen kontrollierten Aufbau einer Software-Plattform. Diese Kombination kann sowohl pragmatische Ansätze der Wiederverwendung im Sinne einer produktgetriebenen Software-Produktlinienentwicklung als auch eine reaktive oder proaktive Software-Produktlinienentwicklung unterstützen. Dies erhöht die Wahrscheinlichkeit der Akzeptanz unterschiedlicher Interessengruppen eines Unternehmens und somit die erfolgreiche Etablierung einer langfristig stabilen Software-Produktlinie deutlich.

Literaturverzeichnis

[ABSH11] Bakr Al-Batran, Bernhard Schätz und Benjamin Hummel: *Semantic Clone Detection for Model-Based Development of Embedded Systems*, Seiten 258–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[ACD+12] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan und A. Stevenson: *Models are code too: Near-miss clone detection for Simulink models*. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Seiten 295–304, Sept 2012.

[ARS+14] Manar H. Alalfi, Eric J. Rapos, Andrew Stevenson, Matthew Stephan, Thomas R. Dean und James R. Cordy: *Semi-automatic identification and representation of subsystem variability in simulink models*. In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, Seiten 486–490. IEEE, 2014.

[ATC03] Ahern, Richard Turner und Aaron Clouse: *CMMI Distilled: A Practical Introduction to Integrated Process Improvement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2. Auflage, 2003.

[BAJ11] Hamid Abdul Basit, Usman Ali und Stan Jarzabek: *Viewing Simple Clones from Structural Clones' Perspective*. In: *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, Seiten 1–6, New York, NY, USA, 2011. ACM.

[Bak95] B. S. Baker: *On finding duplication and near-duplication in large software systems*. In: *Proceedings of 2nd Working Conference on Reverse Engineering*, Seiten 86–95, Jul 1995.

[BB76] Harry G. Barrow und Rod M. Burstall: *Subgraph isomorphism, matching relational structures and maximal cliques*. Information Processing Letters, 4(4):83–84, 1976.

[Bec00] Kent Beck: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Bec02] Kent Beck: *Test-Driven Development: By Example*. Addison-Wesley Professional, 1. Auflage, November 2002.

[BJ05] Hamid Abdul Basit und Stan Jarzabek: *Detecting Higher-level Similarity Patterns in Programs*. In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, Seiten 156–165, New York, NY, USA, 2005. ACM.

[BJ07] Hamid Abdul Basit und Stan Jarzabek: *Efficient Token Based Clone Detection with Flexible Tokenization*. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, Seiten 513–516, New York, NY, USA, 2007. ACM.

[BJ09] H. A. Basit und S. Jarzabek: *A Data Mining Approach for Detecting Higher-Level Clones in Software*. IEEE Transactions on Software Engineering, 35(4):497–514, July 2009.

[BK08] Christel Baier und Joost Pieter Katoen: *Principles of Model Checking*. MIT Press, 2008.

[BKPS07] M. Broy, I. H. Kruger, A. Pretschner und C. Salzmann: *Engineering Automotive Software*. Proceedings of the IEEE, 95(2):356–373, Feb 2007.

[Boe02] B. Boehm: *Get ready for agile methods, with care*. Computer, 35(1):64–69, Jan 2002.

[Bol04] Béla Bollobás: *Extremal Graph Theory*. Courier Corporation, 2004.

[Bos10] J. Bosch: *Toward Compositional Software Product Lines*. Software, IEEE, 27(3):29–34, May 2010.

[Bos14] R. Bosch: *Bosch Automotive Electrics and Automotive Electronics: Systems and Components, Networking and Hybrid Drive*, 2014.

[BR05] Manfred Broy und Andreas Rausch: *Das neue V-Modell XT*. Informatik-Spektrum, 28(3):220–229, 2005.

[BR10] C. Berger und B. Rumpe: *Supporting Agile Change Management by Scenario-Based Regression Simulation*. IEEE Transactions on Intelligent Transportation Systems, 11(2):504–509, June 2010.

[Bro06] Manfred Broy: *Challenges in Automotive Software Engineering*. In: *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, Seiten 33–42, New York, NY, USA, 2006. ACM.

[BRR10a] Christian Berger, Holger Rendel und Bernhard Rumpe: *Measuring the Ability to Form a Product Line from Existing Products*. In: *Variability Modelling of Software-intensive Systems (VaMos)*, 2010.

[BRR+10b] Christian Berger, Holger Rendel, Bernhard Rumpe, Carsten Busse, Thorsten Jablonski und Fabian Wolf: *Product Line Metrics for Legacy Software in Practice*. In: *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, Band 2, 2010.

[BSJL92] Denis M. Bayada, Richard W. Simpson, A. Peter Johnson und Claude Laurenco: *An algorithm for the multiple common subgraph problem*. Journal of Chemical Information and Computer Sciences, 32(6):680–685, 1992.

[BT03] Barry Boehm und Richard Turner: *Balancing Agility and Discipline: A Guide for the Perplexed, Portable Documents*. Addison-Wesley Professional, 2003.

[BT14] G. Bansal und R. Tekchandani: *Selecting a set of appropriate metrics for detecting code clones*. In: *2014 Seventh International Conference on Contemporary Computing (IC3)*, Seiten 484–488, Aug 2014.

[Bun11] Stefan Bunzel: *AUTOSAR – the Standardized Software Architecture*. Informatik-Spektrum, 34(1):79–83, 2011.

[Bur17] Jaime Font Burdeus: *Location of Features as Model Fragments and their Co-Evolution*. 2017.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna und L. Bier: *Clone detection using abstract syntax trees*. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, Seiten 368–377, Nov 1998.

[Cas17] Herman Casier: *Trends and Characteristics of Automotive Electronics*. In: *Wideband Continuous-time $\Sigma\Delta$ ADCs, Automotive Electronics, and Power Management*, Seiten 127–140. Springer, 2017.

[CE00] Krzysztof Czarnecki und Ulrich W. Eisenecker: *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CGBD13] David Cok, Alberto Griggio, Roberto Bruttomesso und Morgan Deters: *The 2012 SMT Competition*. In: *SMT 2012*, Band 20 der Reihe *EPiC Series*. EasyChair, 2013.

[CH01] Mercer Management Consulting und Hypovereinsbank: *Automobiltechnologie 2010*, 2001.

[CKMM08] R. Carbon, Jens Knodel, Dirk Muthig und G. Meier: *Providing Feedback from Application to Family Engineering - The Product Line Planning Game at the Testo AG*. In: *Software Product Line Conference, 2008. SPLC '08. 12th International*, Seiten 180–189, Sept 2008.

[CLWK00] Xia Cai, M. R. Lyu, Kam Fai Wong und Roy Ko: *Component-based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes*. In: *Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, APSEC '00, Seiten 372–, Washington, DC, USA, 2000. IEEE Computer Society.

[CN02a] Paul Clements und Linda Northrop: *Salion, Inc.: A Software Product Line Case Study*. Technischer Bericht CMU/SEI-2002-TR-038, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002.

[CN02b] Paul Clements und Linda Northrop: *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[Coc04] Alistair Cockburn: *Crystal Clear a Human-powered Methodology for Small Teams*. Addison-Wesley Professional, first Auflage, 2004.

[CRGW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina und Jennifer Widom: *Change Detection in Hierarchically Structured Information*. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, Seiten 493–504, New York, NY, USA, 1996. ACM.

[CS13] Cristian Cadar und Koushik Sen: *Symbolic Execution for Software Testing: Three Decades Later*. Communications of the ACM, 56(2):82–90, 2013.

[CSHL13] Maxime Cordy, Pierre Yves Schobbens, Patrick Heymans und Axel Legay: *Beyond Boolean Product-line Model Checking: Dealing with Feature Attributes and Multi-features*. In: *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, Seiten 472–481, Piscataway, NJ, USA, 2013. IEEE Press.

[Dar13] Priya Darshini: *5-best-responsive-table-plugins-concepts*. `http://www.programming-free.com/2013/10/5-best-responsive-table-plugins-concepts.html`, October 2013. Letzter Zugriff: 29.08.2016.

[Dee11] Sam Deering: *35 Amazing jQuery Tables*. `https://www.sitepoint.com/30-amazing-jquery-tables/`, March 2011. Letzter Zugriff: 29.08.2016.

[DHJ+08] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. F. Girard und S. Teuchert: *Clone Detection in Automotive Model-Based Development*. In: *2008 ACM/IEEE 30th International Conference on Software Engineering*, Seiten 603–612, May 2008.

[DPAG11] Jessica Díaz, Jennifer Pérez, Pedro P. Alarcón und Juan Garbajosa: *Agile Product Line Engineering - A Systematic Literature Review*. Software: Practice and Experience, 41(8):921–941, Juli 2011.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker und Krzysztof Czarnecki: *An Exploratory Study of Cloning in Industrial Software Product Lines*. In: *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, Seiten 25–34, Washington, DC, USA, 2013. IEEE Computer Society.

[DRD99] S. Ducasse, M. Rieger und S. Demeyer: *A Language Independent Approach for Detecting Duplicated Code*. In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, Seiten 109–118, 1999.

[DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers und Denys Poshyvanyk: *Feature location in source code: a taxonomy and survey*. Journal of software: Evolution and Process, 25(1):53–95, 2013.

[DTK07] A. Dold, M. Trapp und R. Koschke: *Herausforderungen und Erfahrungen eines OEM bei der Gestaltung Sicherheitsgerechter Prozesse.* In: *GI Jahrestagung (2)*, Seiten 536–540, 2007.

[DW99] Wolfgang Dröschel und Manuela Wiemers: *Das V-Modell 97: der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*. Walter de Gruyter GmbH & Co KG, 1999.

[Ebe15] C. Ebert: *Implementing Functional Safety*. IEEE Software, 32(5):84–89, Sept 2015.

[Edm65] Jack Edmonds: *Paths, Trees, and Flowers*. Canadian Journal of mathematics, 17(3):449–467, 1965.

[EF17] Christof Ebert und John Favaro: *Automotive Software*. IEEE Software, 34(3):33–39, 2017.

[EL12] Rochelle Elva und Gary T. Leavens: *Semantic Clone Detection Using Method IOE-behavior*. In: *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, Seiten 80–81, Piscataway, NJ, USA, 2012. IEEE Press.

[EPA+10] N. Ehsan, A. Perwaiz, J. Arif, E. Mirza und A. Ishaque: *CMMI / SPICE based process improvement*. In: *2010 IEEE International Conference on Management of Innovation Technology*, Seiten 859–862, June 2010.

[Ess16] Markus Esser: *Semantische Ähnlichkeitsanalysen auf Basis extrahierter Teststimuli*. Bachelorarbeit, Lehrstuhl für Software Engineering, RWTH Aachen, 2016.

[FFK08] Raimar Falke, Pierre Frenzel und Rainer Koschke: *Empirical evaluation of clone detection using syntax suffix trees*. Empirical Software Engineering, 13(6):601–643, 2008.

[FFLS08] F. Fabbrini, M. Fusani, G. Lami und E. Sivera: *Software Engineering in the European Automotive Industry: Achievements and Challenges*. In: *2008 32nd Annual IEEE International Computer Software and Applications Conference*, Seiten 1039–1044, July 2008.

[FLLHE14] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon und Alexander Egyed: *Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants*. In: *Proceedings of the 2014 IEEE International Conference on*

Software Maintenance and Evolution, ICSME '14, Seiten 391–400, Washington, DC, USA, 2014. IEEE Computer Society.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein und Joe D. Warren: *The Program Dependence Graph and Its Use in Optimization*. ACM Trans. Program. Lang. Syst., 9(3):319–349, Juli 1987.

[FV03] D. Faust und C. Verhoef: *Software product line migration and deployment*. Software: Practice and Experience, 33(10):933–955, 2003.

[GG93] Matthias Grochtmann und Klaus Grimm: *Classification trees for partition testing*. Software Testing, Verification and Reliability, 3(2):63–82, 1993.

[GGK06] Diane L Gibson, Dennis R Goldenson und Keith Kost: *Performance results of CMMI-based process improvement*. Technischer Bericht, DTIC Document, 2006.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt und Bernhard Rumpe: *Modelling Automotive Function Nets with Views for Features, Variants, and Modes*. In: *Proceedings of ERTS '08*, 2008.

[GJS08] Mark Gabel, Lingxiao Jiang und Zhendong Su: *Scalable Detection of Semantic Clones*. In: *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, Seiten 321–330, New York, NY, USA, 2008. ACM.

[GKFS14] Oleksandr Gordieiev, Vyacheslav Kharchenko, Nataliia Fominykh und Vladimir Sklyar: *Evolution of Software Quality Models in Context of the Standard ISO 25010*, Seiten 223–232. Springer International Publishing, Cham, 2014.

[GKHB10] Nicolas Gold, Jens Krinke, Mark Harman und David Binkley: *Issues in Clone Classification for Dataflow Languages*. In: *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, Seiten 83–84, New York, NY, USA, 2010. ACM.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell und Bernhard Rumpe: *Modeling Variants of Automotive Systems using Views*. In: *Proceedings of Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, Seiten 76–89, March 2008.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler und 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*, Seiten 925–926, 2008.

[Gla01] R.J. van Glabbeek: *The Linear Time-Branching Time Spectrum I - The Semantics of Concrete, Sequential Processes*. In: *Handbook of Process Algebra*. Elsevier, 2001.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze und Andreas Wortmann: *Modeling Variability in Template-based Code Generators for Product Line Engineering*. CoRR, abs/1606.02903, 2016.

[GPM08] Yaser Ghanam, Shelly Park und Frank Maurer: *A Test-Driven Approach to Establishing & Managing Agile Product Lines*. In: *SPLiT 2008–Fifth International Workshop on Software Product Line Testing*, Seite 46, 2008.

[Gub13] Jacob Gube: *10 Best Responsive HTML5 Frameworks*. `http://designinstruct.com/roundups/html5-frameworks/`, July 2013. Letzter Zugriff: 29.08.2016.

[Han10] Eckhart Hanser: *Agile Prozesse: Von XP über Scrum bis MAP*. Springer-Verlag, 2010.

[HBP+17] Alireza Haghighatkhah, Ahmad Banijamali, Olli Pekka Pakanen, Markku Oivo und Pasi Kuvaja: *Automotive software engineering: A systematic mapping study* . Journal of Systems and Software, 128(C):25 – 55, 2017.

[HDZS+15] Holger Höhn, Klaudia Dussa-Zieger, Bernhard Sechser, Bernd Hindel und Richard Messnarz: *Software Engineering nach Automotive SPICE: Entwicklungsprozesse in der Praxis-Ein Continental-Projekt auf dem Weg zu Level 3*. dpunkt. verlag, 2015.

[HF08] Geir K. Hanssen und Tor E. Fígri: *Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering*. Journal of Systems and Software, 81(6):843–854, Juni 2008.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe und Ina Schaefer: *Engineering Delta Modeling Languages*. In: *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, Seiten 22–31, Tokyo, Japan, September 2013. ACM.

[Hil12] Martin Hillenbrand: *Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen*, Band 4. KIT Scientific Publishing, 2012.

[HJS11] Benjamin Hummel, Elmar Juergens und Daniela Steidl: *Index-based Model Clone Detection*. In: *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, Seiten 21–27, New York, NY, USA, 2011. ACM.

[HK11] Yoshiki Higo und Shinji Kusumoto: *Code Clone Detection on Specialized PDGs with Heuristics*. In: *2011 15th European Conference on Software Maintenance and Reengineering*, Seiten 75–84, March 2011.

[HKI08] Yoshiki Higo, Shinji Kusumoto und Katsuro Inoue: *A Metric-based Approach to Identifying Refactoring Opportunities for Merging Code Clones in a Java Software System*. J. Softw. Maint. Evol., 20(6):435–461, November 2008.

[HKK04] Bernd Hardung, Thorsten Kölzow und Andreas Krüger: *Reuse of Software in Distributed Embedded Automotive Systems*. In: *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, Seiten 203–210, New York, NY, USA, 2004. ACM.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe und Ina Schaefer: *First-class Variability Modeling in Matlab/Simulink*. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, Seiten 4:1–4:8, New York, NY, USA, 2013. ACM.

[HKRR11] Arne Haber, Thomas Kutz, Jan Oliver Ringert und Bernhard Rumpe: *MontiArc 1.1.3 - Architectural Modeling Of Interactive Distributed Systems*. Technischer Bericht, RWTH Aachen University, 2011. (to appear).

[HMDZ08] Klaus Hoermann, Markus Mueller, Lars Dittmann und Joerg Zimmer: *Automotive SPICE in Practice: Surviving Implementation and Assessment*. Rocky Nook, 1st Auflage, 2008.

[HMPO+08] O. Haugen, B. Moller-Pedersen, J. Oldevik, G.K. Olsen und A. Svendsen: *Adding Standardized Variability to Domain Specific Languages*. In: *Software Product Line Conference, 2008. SPLC '08. 12th International*, Seiten 139 –148, sept. 2008.

[Hol10] Jörg Holtmann: *Mit Satzmustern von textuellen Anforderungen zu Modellen*. OBJEKT-spektrum, RE/2010 (Online Themenspecial Requirements Engineering), 2010.

[Hor90] Susan Horwitz: *Identifying the Semantic and Textual Differences Between Two Versions of a Program*. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, Seiten 234–245, New York, NY, USA, 1990. ACM.

[HPVM16] Andrei Hutanu, Gabriela Prostean, Stephan Volker und Dumitru Mnerie: *Opportunity Analysis of Change Requests in Automotive Projects*, Seiten 1087–1095. Springer International Publishing, Cham, 2016.

[HRR+11] A. Haber, H. Rendel, B. Rumpe, I. Schaefer und F. van der Linden: *Hierarchical Variability Modeling for Software Architectures*. In: *2011 15th International Software Product Line Conference*, Seiten 150–159, Aug 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe und Ina Schaefer: *Evolving Delta-oriented Software Product Line Architectures*. In: *Proceedings of the 17th Monterey Conference on Large-Scale Complex IT Systems: Development, Operation and Management*, Seiten 183–208, Berlin, Heidelberg, 2012.

[HS89] W. S. Humphrey und W. L. Sweet: *A Method for Assessing the Software Engineering Capability of Contractors*. In: *Software Risk Management*, Seiten 219–245. IEEE Press, Piscataway, NJ, USA, 1989.

[Hum88] W. S. Humphrey: *Characterizing the software process: a maturity framework*. IEEE Software, 5(2):73–79, March 1988.

[HW07] Florian Heidenreich und Christian Wende: *Bridging the gap between features and models*. Aspect-Oriented Product Line Engineering (AOPLE07), 2007.

[HWL+14a] Sönke Holthusen, David Wille, Christoph Legat, Simon Beddig, Ina Schaefer und Birgit Vogel-Heuser: *Family Model Mining for Function Block Diagrams in Automation Software*. In: *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, SPLC '14, Seiten 36–43, New York, NY, USA, 2014. ACM.

[HWL+14b] Sönke Holthusen, David Wille, Christoph Legat, Simon Beddig, Ina Schaefer und Birgit Vogel-Heuser: *Family model mining for function block diagrams in automation software*. In: *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*, Seiten 36–43. ACM, 2014.

[ISO06] Bλ ISO: *The ISO Survey of Certifications 2015*, 2006.

[Iva16] Alex Ivanovs: *Top 22 Best Free HTML5 Frameworks for Responsive Web Development 2016*. `https://colorlib.com/wp/html5-frameworks/`, January 2016. Letzter Zugriff: 29.08.2016.

[Jac12] Daniel Jackson: *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[JDH10] E. Juergens, F. Deissenboeck und B. Hummel: *Code Similarities Beyond Copy & Paste*. In: *2010 14th European Conference on Software Maintenance and Reengineering*, Seiten 78–87, March 2010.

[Jen95] Michael G. Jenner: *Software Quality Management and ISO 9001: How to Make Them Work for You*. John Wiley & Sons, Inc., New York, NY, USA, 1st Auflage, 1995.

[JH01] Ho Won Jung und Robin Hunter: *The relationship between ISO/IEC 15504 process capability levels, ISO 9001 certification and organization size: An empirical study* . Journal of Systems and Software, 59(1):43 – 55, 2001.

[Joh94] J. H. Johnson: *Substring matching for clone detection and change tracking*. In: *Proceedings 1994 International Conference on Software Maintenance*, Seiten 120–126, Sep 1994.

[JQu15a] Learning JQuery: *12 Best Free jQuery Table Plugins*. `http://www.learningjquery.com/2015/09/12-best-free-jquery-table-plugins`, September 2015. Letzter Zugriff: 17.08.2016.

[Jqu15b] Learning Jquery: *15 Top jQuery Chart Plugins for Developers*. `http://www.learningjquery.com/2015/03/15-top-jquery-chart-plugins-for-developers/`, 2015. Letzter Zugriff: 17.08.2016.

[JS09] Lingxiao Jiang und Zhendong Su: *Automatic Mining of Functionally Equivalent Code Fragments via Random Testing*. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, Seiten 81–92, New York, NY, USA, 2009. ACM.

[KA09] Christian Kästner und Sven Apel: *Virtual Separation of Concerns – A Second Chance for Preprocessors*, September 2009. Refereed Column.

[KCH+90a] Kyo Kang, Sholom Cohen, James Hess, William Novak und A. S. Peterson: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technischer Bericht, Software Engineering Institute, Carnegie Mellon University, 1990.

[KCH+90b] Kyo Kang, Sholom Cohen, James Hess, William Nowak und Spencer Peterson: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technischer Bericht, Software Engineering Institute - Carnegie Mellon University, 1990.

[KDO14] Christian Kästner, Alexander Dreiling und Klaus Ostermann: *Variability mining: Consistent semi-automatic detection of product-line features*. IEEE Transactions on Software Engineering, 40(1):67–82, 2014.

[KH01] Raghavan Komondoor und Susan Horwitz: *Using Slicing to Identify Duplication in Source Code*, Seiten 40–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[Kha16] Safdar Khan: *Interactive Software Design Document Generation Framework for the Automotive Industry*. Masterarbeit, Lehrstuhl für Software Engineering, RWTH Aachen, 2016.

[KK09] H. Klar und B. Klages: *Calibration methods and tools for drive train electronics*. In: *3rd Int. Symposium on Development Technlogy, Wiesbaden, Germany*, Seiten 178–189, 2009.

[KKI02] T. Kamiya, S. Kusumoto und K. Inoue: *CCFinder: a multilinguistic token-based code clone detection system for large scale source code*. IEEE Transactions on Software Engineering, 28(7):654–670, Jul 2002.

[Knu98] Donald Ervin Knuth: *The art of computer programming: sorting and searching*, Band 3. Pearson Education, 1998.

[Kos07] Rainer Koschke: *Survey of Research on Software Clones*. In: *Duplication, Redundancy, and Similarity in Software*, Nummer 06301 in *Dagstuhl Seminar Proceedings*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[Kri01] J. Krinke: *Identifying similar code with program dependence graphs*. In: *Proceedings Eighth Working Conference on Reverse Engineering*, Seiten 301–309, 2001.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer und Christoph Schulze: *Interface-based Similarity Analysis of Software Components for the Automotive Industry*. In: *International Systems and Software Product Line Conference (SPLC '16)*, Seiten 99–108, Beijing, China, September 2016. ACM.

[Kru02] C. Krueger: *Eliminating the Adoption Barrier*. Software, IEEE, 19(4):29–31, 2002.

[KRV10] Holger Krahn, Bernhard Rumpe und Steven 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.

[KT06] Jon Kleinberg und Éva Tardos: *Algorithm Design*. Pearson Education India, 2006.

[Lev66] VI Levenshtein: *Binary Codes Capable of Correcting Deletions, Insertions and Reversals*. In: *Soviet Physics Doklady*, Band 10, Seite 707, 1966.

[Lig09] Peter Liggesmeyer: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, 2009.

[LJ05] Seunghak Lee und Iryoung Jeong: *SDD: High Performance Code Clone Detection System for Large Scale Source Code*. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, Seiten 140–141, New York, NY, USA, 2005. ACM.

[LK11] M. Li und R. Kumar: *Stateflow to Extended Finite Automata Translation*. In: *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, Seiten 1–6, July 2011.

[Llc14] Ingenium Llc.: *TOP FIVE BEST RESPONSIVE WEB DESIGN FRAMEWORKS*. `https://www.ingeniumweb.com/blog/post/top-five-best-responsive-web-design-frameworks/2223/`, February 2014. Letzter Zugriff: 29.08.2016.

[LLMZ06] Z. Li, S. Lu, S. Myagmar und Y. Zhou: *CP-Miner: finding copy-paste and related bugs in large-scale software code*. IEEE Transactions on Software Engineering, 32(3):176–192, March 2006.

[LMD+04] M. Lindvall, Dirk Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May und T. Kahkonen: *Agile Software Development in Large Organizations*. Computer, 37(12):26–34, Dec 2004.

[LT10] David I. Levine und Michael W. Toffel: *Quality Management and Job Quality: How the ISO 9001 Standard for Quality Management Systems Affects Employees and Employers*. Manage. Sci., 56(6):978–996, Juni 2010.

[Mat15] Mathworks: *Simulink User's Guide*. Technical Report R2015a, MATLAB & SIMULINK, 2015.

[MB08] Leonardo Moura und Nikolaj Bjørner: *Z3: An Efficient SMT Solver*. In: *TACAS*, Band 4963 der Reihe *LNCS*. Springer, 2008.

[MBBA16] Mariem Mefteh, Nadia Bouassida und Hanêne Ben-Abdallah: *Mining Feature Models from Functional Requirements*, 2016.

[McC76] Edward M. McCreight: *A Space-Economical Suffix Tree Construction Algorithm*. J. ACM, 23(2):262–272, April 1976.

[MLM96] J. Mayrand, C. Leblanc und E. M. Merlo: *Experiment on the automatic detection of function clones in a software system using metrics*. In: *1996 Proceedings of International Conference on Software Maintenance*, Seiten 244–253, Nov 1996.

[MRR11a] Shahar Maoz, Jan Oliver Ringert und Bernhard Rumpe: *ADDiff: Semantic Differencing for Activity Diagrams*. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, Seiten 179–189, New York, NY, USA, 2011. ACM.

[MRR11b] Shahar Maoz, Jan Oliver Ringert und Bernhard Rumpe: *CDDiff: Semantic Differencing for Class Diagrams*, Seiten 230–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert und Bernhard Rumpe: *A Manifesto for Semantic Model Differencing*, Seiten 194–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[MRR14] Shahar Maoz, Jan Oliver Ringert und Bernhard Rumpe: *An Interim Summary on Semantic Model Differencing*. CoRR, abs/1409.0384, 2014.

[MRS10] Kannan Mohan, Balasubramaniam Ramesh und Vijayan Sugumaran: *Integrating Software Product Line Engineering and Agile Development*. Software, IEEE, 27(3):48–55, 2010.

[Muc16] Hossain Muhamad Muctadir: *Similarity Analysis Framework for Software Product Line Extraction*. Masterarbeit, Lehrstuhl für Software Engineering, RWTH Aachen, 2016.

[MZB+15] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein und Yves Le Traon: *Bottom-up adoption of software product lines: a generic and extensible approach*. In: *Proceedings of the 19th International Conference on Software Product Line*, Seiten 101–110. ACM, 2015.

[MZDH16] Markus Müller, Jörg Zimmer, Lars Dittmann und Klaus Hörmann: *Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren*. dpunkt. verlag, 2016.

[MZKLT14] Jabier Martinez, Tewfik Ziadi, Jacques Klein und Yves Le Traon: *Identifying and visualising commonality and variability in model variants*. In: *European Conference on Modelling Foundations and Applications*, Seiten 117–131. Springer, 2014.

[NK08] Natsuko Noda und Tomoji Kishi: *Aspect-oriented modeling for variability management*. In: *Software Product Line Conference, 2008. SPLC'08. 12th International*, Seiten 213–222. IEEE, 2008.

[NMM05] Sridhar Nerur, RadhaKanta Mahapatra und George Mangalaraj: *Challenges of migrating to agile methodologies*. Communications of the ACM, 48(5):72–78, 2005.

[NNP+09] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi und T. N. Nguyen: *ClemanX: Incremental clone detection tool for evolving software*. In: *2009 31st International Conference on Software Engineering - Companion Volume*, Seiten 437–438, May 2009.

[OB88] T. J. Ostrand und M. J. Balcer: *The Category-partition Method for Specifying and Generating Functional Tests*. Communications of the ACM, 31(6):676–686, Juni 1988.

[Pau93] Mark C. Paulk: *Comparing ISO 9001 and the Capability Maturity Model for Software*. Software Quality Journal, 2(4):245–256, 1993.

[Pau02] Mark Paulk: *Capability Maturity Model for Software*. John Wiley & Sons, Inc., 2002.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Krüger und Thomas Stauner: *Software Engineering for Automotive Systems: A Roadmap*. In: *2007 Future of Software Engineering*, FOSE '07, Seiten 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[PBL05] Klaus Pohl, Günter Böckle und Frank J. van der Linden: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PCCW93] M. C. Paulk, B. Curtis, M. B. Chrissis und C. V. Weber: *Capability maturity model, version 1.1*. IEEE Software, 10(4):18–27, July 1993.

[PHV17] G Prostean, A Hutanu und S Volker: *Impact of agile methodologies on team capacity in automotive radio-navigation projects*. IOP Conference Series: Materials Science and Engineering, 163(1):012013, 2017.

[PMDL99] J. F. Patenaude, E. Merlo, M. Dagenais und B. Lague: *Extending software quality assessment techniques to Java systems*. In: *Proceedings Seventh International Workshop on Program Comprehension*, Seiten 49–56, 1999.

[PNN+09] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi und Tien N. Nguyen: *Complete and Accurate Clone Detection in Graph-based Models*. In: *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Seiten 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[PR14] B. Priyambadha und S. Rochimah: *Case study on semantic clone detection based on code behavior*. In: *2014 International Conference on Data and Software Engineering (ICODSE)*, Seiten 1–6, Nov 2014.

[PV09] Corina S. Păsăreanu und Willem Visser: *A survey of new trends in symbolic execution for software testing and analysis*. International Journal on Software Tools for Technology Transfer, 11(4):339, Aug 2009.

[PWC+94] Mark C Paulk, Charles V Weber, Bill Curtis, Mary Beth Chrissis *et al.*: *The capability maturity model: Guidelines for improving the software process*, Band 441. Addison-wesley Reading, Massachusetts:, 1994.

[Rah15] Syed Fazle Rahman: *16 JavaScript Libraries for Creating Beautiful Charts*. `https://www.sitepoint.com/15-best-javascript-charting-libraries//`, 2015. Letzter Zugriff: 17.08.2016.

[RBS13] Dhavleesh Rattan, Rajesh Bhatia und Maninder Singh: *Software clone detection: A systematic review*. Information and Software Technology, 55(7):1165 – 1199, 2013.

[RC12] Julia Rubin und Marsha Chechik: *Combining related products into product lines*. In: *Fundamental Approaches to Software Engineering*, Seiten 285–300. Springer, 2012.

[RCC13] Julia Rubin, Krzysztof Czarnecki und Marsha Chechik: *Managing Cloned Variants: A Framework and Experience*. In: *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, Seiten 101–110, New York, NY, USA, 2013. ACM.

[RCK09] Chanchal K. Roy, James R. Cordy und Rainer Koschke: *Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach*. Sci. Comput. Program., 74(7):470–495, Mai 2009.

[REF+07] Terence P. Rout, Khaled El Emam, Mario Fusani, Dennis Goldenson und Ho Won Jung: *{SPICE} in retrospect: Developing a standard for process assessment*. Journal of Systems and Software, 80(9):1483 – 1493, 2007. Evaluation and Assessment in Software EngineeringEASE06.

[Ric14] Johannes Martin Richenhagen: *Entwicklung von Steuerungs-Software für den automobilen Antriebsstrang mit agilen Methoden*. Dissertation, RWTH Aachen University - Lehrstuhl für Verbrennungskraftmaschinen, 2014.

[RPK10] Uwe Ryssel, Joern Ploennigs und Klaus Kabitzsch: *Automatic Variation-point Identification in Function-block-based Models*. In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, Seiten 23–32, New York, NY, USA, 2010. ACM.

[RPS14] Johannes Richenhagen, Stefan Pischinger und Axel Schloßer: *PERSIST—A Flexible and Automatically Verifiable Software Architecture for the Automotive Powertrain*. Journal of Electrical Engineering, 2:108–115, 2014.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen und Michael von Wenckstern: *Test-driven Semantical Similarity Analysis for Software Product Line Extraction*. In: *International Systems and Software Product Line Conference (SPLC '16)*, Seiten 174–183, Beijing, China, September 2016. ACM.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze und Andreas Wortmann: *Teaching Agile Model-driven Engineering for Cyber-physical Systems*. In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ICSE-SEET '17, Seiten 127–136, Piscataway, NJ, USA, 2017. IEEE Press.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann: *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert und Peter Manhart: *Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution*. In: *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, Seiten 141–150, New York, NY, USA, 2015. ACM.

[Rum96] Bernhard Rumpe: *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, Technische Universität München, 1996.

[Rum16] Bernhard Rumpe: *Modeling with UML: Language, Concepts, Methods*. Springer Publishing Company, Incorporated, 1st Auflage, 2016.

[Rum17] Bernhard Rumpe: *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RVSP15] Johannes Richenhagen, Hariharan Venkitachalam, Axel Schloßer und Stefan Pischinger: *PERSIST — a scalable software architecture for the control of various automotive hybrid topologies*. Technischer Bericht, SAE Technical Paper, 2015.

[RWH08] A. Rajan, M. Whalen und M. Heimdahl: *The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage*. In: *2008 ACM/IEEE 30th International Conference on Software Engineering*, Seiten 161–170, May 2008.

[SAL+15] Mahdi Shahbakhti, Mohammad Reza Amini, Jimmy Li, Satoshi Asami und J. Karl Hedrick: *Early model-based design and verification of automotive control system software implementations*. Journal of Dynamic Systems, Measurement, and Control, 137(2):021006, 2015.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani und Nico Tanzarella: *Delta-oriented programming of software product lines*. Software Product Lines: Going Beyond, Seiten 77–91, 2010.

[SCD03] N. Synytskyy, J. R. Cordy und T. Dean: *Resolution of static clones in dynamic Web pages*. In: *Fifth IEEE International Workshop on Web Site Evolution, 2003. Theme: Architecture. Proceedings.*, Seiten 49–56, Sept 2003.

[Sch04] Ken Schwaber: *Agile Project Management with Scrum*. Microsoft Press, Redmond, WA, USA, 2004.

[Sch12] Martin Schindler: *Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P*. Dissertation, RWTH Aachen, 2012.

[SEHV12] Thomas Stahl, Sven Efftinge, Arno Haase und Markus Völter: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt. verlag, 2012.

[Sin15] Vaibhav Singhal: *20 best JavaScript charting libraries*. `http://thenextweb.com/dd/2015/06/12/20-best-javascript-chart-libraries/#gref/`, 2015. Letzter Zugriff: 17.08.2016.

[SK06] Hans Sassenburg und David Kitson: *A comparative analysis of CMMI and automotive SPICE*. European SEPG, Amsterdam/Netherlands (June 2006), 2006.

[SK16a] A. Sheneamer und J. Kalita: *Semantic Clone Detection Using Machine Learning*. In: *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Seiten 1024–1028, Dec 2016.

[SK16b] Abdullah Sheneamer und Jugal Kalita: *A Survey of Software Clone Detection Techniques*. International Journal of Computer Applications, Seiten 0975–8887, 2016.

[Smi07] Preston G. Smith: *Flexible product development: building agility for changing markets*. John Wiley & Sons, 2007.

[SRC+12] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo und Karina Villela: *Software Diversity: State of the Art and Perspectives*. International Journal on Software Tools for Technology Transfer, 14(5):477–495, 2012.

[SS15] Peter Schiele und Alexander Siller: *Qualitätssicherung in der Software-Entwicklung am Beispiel des BMW i3*. In: *GI-Jahrestagung*, Seiten 1699–1709, 2015.

[SSR09] Paulo Sampaio, Pedro Saraiva und António Guimaraes Rodrigues: *ISO 9001 certification research: questions, answers and approaches*. International Journal of Quality & Reliability Management, 26(1):38–58, 2009.

[Ste07] Angelika Steger: *Diskrete Strukturen 1. Kombinatorik, Graphentheorie, Algebra*. Springer, Berlin, 2. Auflage, 2007.

[Stö13] Harald Störrle: *Towards clone detection in UML domain models*. Software & Systems Modeling, 12(2):307–329, 2013.

[SWP+09] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan und Zhendong Su: *Detecting Code Clones in Binary Executables*. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, Seiten 117–128, New York, NY, USA, 2009. ACM.

[SZC16] Jörg Schäuffele, Thomas Zurawka und R Carey: *Automotive Software Engineering, Second Edition*. SAE International, 2016.

[TC06] Kun Tian und Kendra Cooper: *Agile and Software Product Line Methods: Are They So Different*. In: *1st International Workshop on Agile Product Line Engineering*, 2006.

[TG06] Robert Tairas und Jeff Gray: *Phoenix-based Clone Detection Using Suffix Trees*. In: *Proceedings of the 44th Annual Southeast Regional Conference*, ACM-SE 44, Seiten 679–684, New York, NY, USA, 2006. ACM.

[The14] Christian Theis: *Automatische Evaluierung von Funktionskomponenten-Kompatibilität in Bezug auf Evolution und Variation*. Masterarbeit, Lehrstuhl für Software Engineering, RWTH Aachen, 2014.

[Thi15] Kevin Thissen: *Testbasierte Kompatibilitätsanalysen von Funktionskomponenten*. Masterarbeit, Lehrstuhl für Software Engineering, RWTH Aachen, 2015.

[Tip95] Frank Tip: *A Survey of Program Slicing Techniques*. Journal of Programming Languages, 3:121–189, 1995.

[TK06] Ann Terlaak und Andrew A. King: *The effect of certification with the {ISO} 9000 Quality Management Standard: A signaling approach* . Journal of Economic Behavior & Organization, 60(4):579 – 602, 2006.

[TT01] Barry N. Taylor und Ambler Thompson: *The international system of units (SI)*. 2001.

[VG07] Markus Voelter und Iris Groher: *Product line implementation using aspect-oriented and model-driven software development*. In: *Software Product Line Conference, 2007. SPLC 2007. 11th International*, Seiten 233–242. IEEE, 2007.

[VHJG95] John Vlissides, Richard Helm, Ralph Johnson und Erich Gamma: *Design patterns: Elements of reusable object-oriented software*. Reading: Addison-Wesley, 49(120):11, 1995.

[VPH16] Stephan Volker, Gabriela Prostean und Andrei Hutanu: *Research of Automotive Change Management and Supportive Risk Management*, Seiten 1097–1108. Springer International Publishing, Cham, 2016.

[VRP15] Hariharan Venkitachalam, Johannes Richenhagen und Stefan Pischinger: *A Generic Control Software Architecture for Battery Management Systems*. Technischer Bericht, SAE Technical Paper, 2015.

[WAB+16] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan Peter Ostberg und Jasmin Ramadani: *How are functionally similar code clones syntactically different? An empirical study and a benchmark*. PeerJ Computer Science, 2:e49, März 2016.

[WB07] Victor B. Wayhan und Erica L. Balderson: *TQM and Financial Performance: What has Empirical Research Discovered?* Total Quality Management & Business Excellence, 18(4):403–412, 2007.

[WDWD08] Karl W Wagner, Walter Dürr, Karl W Wagner und Walter Dürr: *Reifegrad nach ISO/IEC 15504 (SPiCE) ermitteln*. Carl Hanser Verlag GmbH & Co. KG, 2008.

[WFS13] Matthias Welge, Christian Friedrich und Atique Shair: *Integration von agilen Methoden in der Systementwicklung*. Tag des Systems Engineering: Zusammenhänge erkennen und gestalten, Seite 341, 2013.

[WHR14] J. Whittle, J. Hutchinson und M. Rouncefield: *The State of Practice in Model-Driven Engineering*. IEEE Software, 31(3):79–85, May 2014.

[WHSS13] David Wille, Sönke Holthusen, Sandro Schulze und Ina Schaefer: *Interface variability in family model mining*. In: *Proceedings of the 17th International Software Product Line Conference co-located workshops*, Seiten 44–51. ACM, 2013.

[WL+11] D. von Wissel, P Moreno Lahore *et al.*: *Renault Model-Based Design-Powertrain Control Development Process*. In: *23rd Int. AVL Conference Engine & Environment, Graz, Austria*, 2011.

[WM05] R. Wettel und R. Marinescu: *Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments*. In: *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, Seiten 63–, Sept 2005.

[WTS+16] David Wille, Michael Tiede, Sandro Schulze, Christoph Seidl und Ina Schaefer: *Identifying Variability in Object-Oriented Code Using Model-Based Code Mining*, Seiten 547–562. Springer International Publishing, Cham, 2016.

[Yan91] Wuu Yang: *Identifying Syntactic Differences Between Two Programs*. Softw. Pract. Exper., 21(7):739–755, Juni 1991.

[Yod02] Joseph W. Yoder: *Workshop on Software Reuse and Agile Approaches*. In: *Software Reuse: Methods, Techniques, and Tools*, Seiten 336–336. Springer, 2002.

[ZH18] Tewfik Ziadi und Lom Messan Hillah: *Software Product Line Extraction from Bytecode based applications*. In: *International Conference on Engineering of Complex Computer Systems*, 2018.

[ZHP+14] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane und Yves Le Traon: *Towards a language-independent approach for reverse-engineering of software product lines*. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, Seiten 1064–1071. ACM, 2014.

[ZK12] Changyan Zhou und Ratnesh Kumar: *Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata.* Discrete Event Dynamic Systems, 22(2):223–247, 2012.

Abkürzungsverzeichnis

AE	Anwendungsentwicklung
AD	Aktivitätsdiagramm
AST	Abstrakter Syntaxbaum, Abstrakt Syntax Tree
BDD	Binäre Entscheidungsdiagramme, Binary Decision Diagrams
CD	Klassendiagramm, Class Diagram
CEGAS	Gegenbeispiel gesteuerte semantische Ähnlichkeitsanalyse, CounterExample Guided Abstraction Similarity Metric
CFG	Kontrollflussgraph, Control Flow Graph
CMM	Capability Maturity Model for Software
CMMI	Capability Maturity Model Integrated
CmpD	Komponentendiagramm, Component Diagram
CR	Änderungsanfrage, Change Request
CVL	Common Variability Language
DE	Domänenentwicklung
ID	Identität
ISDDGF	Interactive Software Design Document Generation Framework
I/O-EFA	Erweiterter endlicher Eingabe/Ausgabe-Automat, Input/Ouput Extended Finite Automata
MAA	MontiArc Automaton
OD	Objektdiagramm
PDG	Programabhängigkeitsgraph, Program Dependency Graph
PERSIST	Powertrain control architecture Enabling Reusable Software development for Intelligent System Tailoring
SF	Stateflow

SimA Similarity Analysis Framework

SL Simulink

SPICE Software Process Improvement and Capibility dEtermination

SPL Software-Produktlinie

SPLE Software-Produktlinienentwicklung

SVN Apache Subversion

SW-C Software-Komponente, Software Component

UML Unified Modeling Language

Abbildungsverzeichnis

Tabellenverzeichnis

Related Interesting Work from the SE Group, RWTH Aachen

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." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], 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] 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]. 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 reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivative of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and 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)

Starting with an 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 two books [Rum16] and [Rum12] 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 objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is 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+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question 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 [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using 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]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using 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], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

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] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, **?**] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. 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]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type 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]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. 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 and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] 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 on class and object diagrams in [MRR11e] 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 evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact 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. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [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], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & 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+08] 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 [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] 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 for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

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] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] 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 [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

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 composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modeling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

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 requirements management that connects requirements with features in all phases of the development 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+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. 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 development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current 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. As tooling infrastructure, the SSELab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. 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 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 to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

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, to lower the barrier for web-based innovations and to 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, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. 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 own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

Literaturverzeichnis

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[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, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[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.

[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+98] 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.

[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.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 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.

[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 Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[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.

[EFLR99] 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.

[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.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] 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.

[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.

[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+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[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.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[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.

[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.

[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.

[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.

[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.

[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.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[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.

[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.

[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.

[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.

[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.

[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.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[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.

[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.

[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.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[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. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] 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.

[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.

[MRR14] 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.

[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.

[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.

[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.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 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.

[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.

[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.

[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 *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[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, 2te Auflage*. Springer Berlin, September 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.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[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.

[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.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[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.