Fraunhofer
FOKUS

FRAUNHOFER-INSTITUT FÜR OFFENE KOMMUNIKATIONSSYSTEME

Tibor Farkas

Regelbasierte Konformitätsprüfung kollaborativer Artefakte

FRAUNHOFER VERLAG

Fraunhofer-Institut für
Offene Kommunikationssysteme FOKUS

Tibor Farkas

Regelbasierte Konformitätsprüfung kollaborativer Artefakte

FRAUNHOFER VERLAG

Kontaktadresse:
Fraunhofer FOKUS, MOTION
Kaiserin-Augusta-Allee 31
10589 Berlin
Tel: 030 3463-7000
E-Mail: info@fokus.fraunhofer.de
www.fokus.fraunhofer.de

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
ISBN: 978-3-8396-0266-9

D 83

Zugl.: Berlin, TU, Diss., 2010

Druck: Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Für den Druck des Buches wurde chlor- und säurefreies Papier verwendet.

© by FRAUNHOFER VERLAG, 2011
Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 80 04 69, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon 07 11 9 70-25 00
Telefax 07 11 9 70-25 08
E-Mail verlag@fraunhofer.de
URL http://verlag.fraunhofer.de

Dissertation

Regelbasierte Konformitätsprüfung kollaborativer Artefakte

vorgelegt von
Diplom-Ingenieur
Tibor Farkas
geboren in Berlin

von der Fakultät IV – Elektrotechnik und Informatik –
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften

– Dr.-Ing. –

Gutachter: **Prof. Dr. Ina Schieferdecker,**
Prof. Dr. Dr. h.c. Radu Popescu-Zeletin

Technische Universität Berlin, Fakultät Elektrotechnik und Informatik / Fraunhofer Gesellschaft

Gutachter: **Prof. Dr. Holger Giese**
Hasso-Plattner-Institut für Softwaresystemtechnik GmbH an der Universität Potsdam

Tag der wissenschaftlichen Aussprache: 6. Dezember 2010, Berlin

D 83

Technische Universität Berlin

Fakultät IV - Elektrotechnik und Informatik

Institut Entwurf und Testen von Telekommunikationssystemen

Franklinstraße 28/29

10587 Berlin, Germany

http://www.iv.tu-berlin.de

Hasso-Plattner-Institut für Softwaresystemtechnik GmbH an der Universität Potsdam

Fachgebiet Systemanalyse und Modellierung

Prof-Dr-Helmert-Str. 2-3

14482 Potsdam

http://www.hpi.uni-potsdam.de

Meiner Familie,

Rabea und Marvin

Kurzfassung

Im modellbasierten Entwicklungsprozess eingebetteter Systeme stellen industrielle Normenwerke und daraus resultierende Prozessrichtlinien konkrete Anforderungen an die Erstellung von Spezifikationen, Dokumenten und Daten einer IT-Umgebung, sogenannter *Artefakte* (elektronische Arbeitserzeugnisse). In einer prozesslogischen Sicht bestehen übergreifende, wechselseitige Abhängigkeiten zwischen den Artefakten, welche aus der Kombination von Artefakt-Eigenschaften selbst oder durch die Anwendung gekoppelter Werkzeuge resultieren, sogenannter *kollaborativer Artefakte*. Entwicklungsrichtlinien legen für solche kollaborativen Artefakte prozessübergreifend fest, wie Spezifikationen einheitlich formuliert, Architekturen konform modelliert, Dokumente konsistent bearbeitet, Simulationen fehlerfrei entworfen, eingebettete Software generiert oder Daten in spezieller Formatierung für die Testautomatisierung abgelegt werden. Heutzutage erfordert der ingenieurmäßige Entwurf eines eingebetteten Systems die nachgewiesene *Konformität* (Erfüllung der Anforderungen) zu prozessspezifischen Entwicklungsrichtlinien, gerade in kollaborativ durchgeführten Arbeitsschritten. Daher kommt der statischen Konformitätsanalyse logisch-abhängiger, nicht-funktionaler Eigenschaften zur Entwurfszeit eine besondere Bedeutung zu. Als zunehmend problematisch erweist sich jedoch die Eingeschränktheit der heutzutage verfügbaren, automatisierten Ansätze zur *regelbasierten Konformitätsprüfung*, da diese die vorherrschenden Abhängigkeiten und die notwendige Reichweite von interdisziplinär geltenden Entwicklungsrichtlinien sowie die komplexen Wechselwirkungsaspekte der involvierten Artefakte nicht auf Konformität abprüfen können. Dieser Umstand macht viele Abstimmungsiterationen der Mitarbeiter nötig, verursacht hohe Review-Aufwände im Prozess und birgt schließlich versteckte Risiken durch menschliche Nachlässigkeit bei der manuellen Artefakt-Durchsicht.

Ausgehend von regelbasierten Ansätzen zur statischen Analyse einzelner Artefakte basiert die Idee dieser Arbeit auf einer Erweiterung rein singulärer Ansätze hin zu einer *werkzeugübergreifenden* Konformitätsprüfung prozesslogisch abhängiger Artefakte durch die konzentrierte Abbildung jener Abhängigkeiten in ein *logisches Artefakt.* Durch Abstraktion und Transformation kollaborativer Artefakte in ein plattformunabhängiges Datenmodell sowie der Formalisierung natürlich sprachlicher Entwicklungsrichtlinien in digitale Regeln (computerausführbare Algorithmen), können interdisziplinäre Prozessrichtlinien und ihre semantischen Abhängigkeiten durch das *logische Artefakt* beschrieben, auf diesem ausgeführt und schließlich die Einhaltung der Konformität automatisiert abgeprüft werden. Neben dem theoretisch fundierten Lösungsansatz werden eine Methodik, werkzeugtechnische Umsetzungen sowie reale Fallbeispiele anhand geltender Richtlinien aus dem industriellen Kontext der Automobilindustrie vorgestellt.

Die Arbeit entstand in einer fünfjährigen Forschungstätigkeit am Fraunhofer Institut für offene Kommunikationssysteme (FOKUS). Dabei wurde sie entscheidend motiviert durch die Prozessanforderungen der Volkswagen AG und der engen Zusammenarbeit mit der Carmeq GmbH. Partiell veröffentlichte Forschungsergebnisse werden hierbei erstmals zusammengefasst sowie weiterführende Anforderungen anhand von Fallbeispielen und technische Aspekte anhand neuer Technologien konzipiert und evaluiert.

Abstract

In model-based development process of embedded systems, business standards and the resulting process guidelines have an impact on the way of preparing documents, modeling and computer data definition in an IT-infrastructure, known as artifacts (one of many kinds of data produced during the development). In a process point of view there are logical coherences and inter-dependencies between these artifacts that result from the combination of artifact properties themselves or through the application of coupled tool-chains, so-called collaborative artifacts. For collaborative artifacts development guidelines define the way of how specifications are uniformly expressed, how architecture design is modeled conform, how documents are consistently processed, how simulation models are designed well, in what manner software is being generated and how test case specifications are made efficiently. Today, engineering of an embedded system requires the proven conformity (fulfillment of the requirements) to process-specific development guidelines, especially in collaborative development steps. Therefore, the static compliance analysis of logically dependent, non-functional properties gains a special significance at design time. However, increasingly problematic becomes the limitation of rule-based approaches for automated compliance checking to one artifact available today, since overall dependencies, the necessary range of disciplines within development standards and the complex interactions between involved artifacts cannot be automatically checked. Because of this circumstance, many employees are doing time intensive and error prone reviews of documents, models or data - this poses hidden risks caused by human negligence in the manual artifact review.

Based on rule-based test approaches to static analysis of individual artifacts, the idea of this work is based purely on an extension of singular approaches towards an overall compliance audit of logically dependent artifacts due to the concentrated image of those dependencies in a logical artifact. Through abstraction and transformation of collaborative artifacts in a platform-independent data model, as well the formalization of textual guidelines into digital rules (executable algorithms), a logical artifact is created. The logical artifact can cover overall process guidelines, can describe semantic dependencies between collaborative artifacts and can be the source for complex conformance checks that could be executed automatically. In addition to the theoretically approach, an own methodology, implementation of two compliance checker tools and real case examples are presented in this work, based on existing guidelines from the industrial context of the automotive industry.

This work resulted in a five-year research work at the Fraunhofer Institute for Open Communication Systems (FOKUS) and was decisively motivated by the process requirements of Volkswagen AG and by the close cooperation with the Carmeq GmbH. For the first time partially published research results are summarized by this work. Furthermore new requirements and concepts are introduced and evaluated on the basis of case studies and technical aspects.

Danksagung

Ich hätte es nicht für möglich gehalten, dass nach allen Recherchen, den gesammelten Erfahrungen, unzähligen Gesprächen und den Studien zu dem Thema die Arbeit doch noch zum Abschluss kommen würde. Für die Unterstützung bei der Entstehung dieser Doktorarbeit möchte ich daher vielen Menschen einen herzlichen Dank aussprechen.

Zunächst möchte ich mich bei Frau Prof. Dr. Ina Schieferdecker und bei Herrn Prof. Dr. Dr. h. c. Radu Popescu-Zeletin bedanken, denn sie brachten mir sehr viel ihrer Geduld entgegen und sorgten mit wertvollen Ratschlägen für das Gelingen der Arbeit. Frau Prof. Dr. Ina Schieferdecker gilt mein besonderer Dank für die wissenschaftlichen Freiräume und die Unterstützung zur Vertiefung meines Themas im Arbeitsgebiet ‚Embedded Systems Engineering'. Herrn Prof. Dr. Dr. h. c. Radu Popescu-Zeletin möchte ich ganz besonders für seine Unterstützung danken, dieses Thema auch im industriellen Kontext erproben zu können. Viele Dienstreisen, Vorträge, Kritik und Ermutigungen haben mich auch persönlich voran gebracht − ich bin sehr dankbar dafür.

Des Weiteren möchte ich Herrn Prof. Dr. Holger Giese insbesondere meinen Dank aussprechen, der mir während der Entstehung die Möglichkeit zur Veröffentlichung meiner Forschungsergebnisse gegeben und diese Arbeit mit seinen kompetenten Ratschlägen wesentlich verbessert hat.

Das konzipierte Prüfsystem zur industriellen Produktreife führen zu können verdanke ich dem reichen Erfahrungsschatz von Herrn Prof. Dr. Wolfgang Merker. Vielen Dank für die Unterstützung als Mentor und die wertvollen Hinweise im Kontext der Automotive-Domäne.

Ganz besonders möchte ich mich für die Geduld und die Freiräume bei meiner Frau Rabea bedanken − durch ihre aufmunternden Worte konnte ich so manche Durststrecke überwinden – ohne sie wäre die Doktorarbeit niemals möglich gewesen.

Dr. Marc Born, Torsten Klein, Dr. Eckhard Moeller und Harald Röbig danke ich an der Stelle für ihre großartige Unterstützung bei der Entwicklung des ersten Konzeptes zu Beginn und weiteren Anregungen zu meiner Idee, ihrer Inspiration sowie der vielen kreativen Diskussionen. In den fünf Jahren meiner Forschungsarbeit habe ich ihren Rat und ihren Beistand immer sehr geschätzt.

Ein großer Dank geht auch an meine Kollegen am Fraunhofer FOKUS und Freunde, denn die Zusammenarbeit mit ihnen war ein großer Mehrwert bei der Erstellung meiner Doktorarbeit. Ich danke Dr. Klaus-Peter Eckert für das Brainstorming über Vor- und Nachteile von IT-Technologien im Kontext MDA, Klaus-Dietrich Engels für die Diskussionen über Methoden und Werkzeuge bei unseren ‚Kanalrunden', Andreas Hinnerichs und Carsten Neumann für die gemeinsamen Arbeiten in Forschungsprojekten und an Veröffentlichungen, Tom Ritter und Christian Hein für ihre Unterstützung bzgl. des OCL-Prüfsystems und der wertvollen Zusammenarbeit, Dr. Tilmann Rassy für die Einweisung in die mathematischen Grundlagen, Dr. Mang Li für die Diskussionen über Methodik des Prüfsystems, Nguyen Viet Thang und Ying Wang für die Unterstützung bei der Implementierung des Prüfsystems und der Regelwerke sowie nicht zuletzt Dr. Justyna Zander für ihre hilfreiche Unterstützung im Bereich Testspezifikation.

Berlin, im April 2010
Tibor Farkas

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Hochkomplexe technische Produkte in Schlüsselbranchen der deutschen Wirtschaft wie moderne Automobile, Großraumflugzeuge oder Hochgeschwindigkeitszüge stellen hohe Anforderungen an heutige Engineering-Prozesse dieser Domänen. Dies betrifft Hersteller und Lieferanten im Produktentstehungsprozess, welche die kollaborative Planung, Spezifikation, Simulation, Implementierung und Test solcher Produkte absichern müssen. In technischen Produkten sind dabei zunehmend miteinander vernetzte, elektronische Teilsysteme integriert, die so genannten eingebetteten Systeme, welche vermehrt aus softwareintensiven Elektrik-/Elektronik-Bauteilkomponenten (E/E-Systemen) aufgebaut sind [LIGG05]. Das Gesamtmarktvolumen für eingebettete Systeme manifestiert den Trend und wird laut Bitkom-Studie bereits im Jahr 2007 für Deutschland auf über 18,7 Mrd. € geschätzt [BIT08]. Die Anbieter tragen dazu 3,7 Mrd. € bei und die Wertschöpfung in den Domänen umfasst mehr als 15 Mrd. €. Im Automobilbau wird der Trend besonders deutlich: Keine andere Technologie hat das Auto in den vergangenen 25 Jahren so stark verändert wie die Elektronik [SZC05]. E/E-Systeme sind bisher die wesentlichen Innovationstreiber für bis zu 90 % aller Innovationen im Automobil [FKFS07]. Davon entfällt der größte Anteil von prognostizierten 80 % rein auf die Software-Entwicklung für die Systeme [LHFB02]. Der reine Softwareanteil erbringt rund 30 % der Wertschöpfung eines Mittelklassefahrzeugs [HRH02]. Dadurch getrieben werden sich die Elektronikkosten zum Hauptkostentreiber in der Fahrzeugentwicklung etablieren. Im Jahr 2015 werden diese bis zu 35 % der Gesamtentwicklungskosten betragen [ADL07]. Aufgrund des aktuellen Wandels hin zu elektrischen Antrieben [VDA09] wird der Softwareanteil durch die Elektromobilität noch viel stärker ansteigen, als heute vermutet [LIEBL00].

Durch den Innovationstrend geraten Engineering-Prozesse dieser Domänen unter enormen Druck: Die technische Entwicklungskomplexität nimmt zu, sodass immer mehr benötigtes Wissen durch verschiedene Akteure (Beraterfirmen, Systemlieferanten, Bauteilelieferanten) bereitgestellt werden muss. Die Entwicklungszeiten müssen verkürzt werden, um mehr Varianten der Produkte in immer kürzeren Abständen auf den Markt zu bringen. Die Kosten sollen weiter gesenkt werden, damit die Gesamtkosten des Produkts nicht eskalieren. Dabei wirken die E/E-Systeme in ihrer Funktion immer stärker auf den Menschen (z. B. wie die Fahrerassistenzsysteme), sodass die Produktqualität eine entscheidende Rolle erhält. Ambitionierte Zielvorgaben für eine Kostensenkung von bis zu 70 % bei gleichzeitiger Erhöhung der Produktqualität sind heutzutage angestrebte Ziele [BEND05; ADL07]. Hersteller und Lieferanten müssen Standardisierungen zeit- und kosteneffizient umsetzen, trotz wachsender Komplexität der Systeme [BIT08].

Im Engineering-Kontext der erwähnten Domänen werden eingebettete Systeme eigens für spezielle Anwendungen kollaborativ zwischen Hersteller und Zulieferer entworfen und führen dedizierte Produktfunktionen innerhalb des Gesamtsystems aus [FS08]. Die Teilsysteme des Produkts werden sowohl hardwaretechnisch als auch zunehmend softwaretechnisch überwacht, geregelt und gesteuert [REIF06]. Die Systemarchitektur kann

sowohl Mikroprozessoren, anwendungsspezifische Hardwareschaltungen als auch Funktionen mittels eingebetteter Software aufweisen [BAO03]. Bei E/E-Systemen muss die eingebettete Software in vielfältiger Weise mit der Umgebung, d. h. der Sensorik, Aktorik oder Mechanik interagieren [SCHÄ03], trägt entscheidend zur Produktfunktionalität bei und wirkt sich letztendlich auf die Produktqualität aus [ADAC09].

Heutzutage gelten aufgrund verschiedener Gesetze und technischer Normen auf nationaler wie internationaler Ebene immer stringentere Produktanforderungen hinsichtlich der Funktionalität und der Qualität solcher komplexer technischer Produkte [FLEI08]. Insbesondere in sicherheitskritischen Anwendungsentwicklungen, wie zum Beispiel das softwaregesteuerte elektronische Notbremssystem im Fahrzeug, sind für eine Marktfreigabe im Zulassungsprozess strenge Richtlinien und Auflagen für Engineering-Prozesse vorherrschend, die ein Hersteller beachten und mittels Zertifizierung auch nachweisen muss. Aus industriellen Normungen leiten sich Anforderungen ab, die sich früh und übergreifend auf die Entwicklungsphasen und deren Prozesse auswirken.

Um eine geregelte Zusammenarbeit in Engineering-Prozessen zu gewährleisten, erlassen branchenspezifische Standardisierungsgremien im Bereich der Entwicklung eingebetteter Systeme Empfehlungen für konforme Arbeitsweisen, wie das ASAM-Konsortium [ASAM09], das AUTOSAR-Konsortium [ASR09], die Hersteller Initiative Software [HIS09], das MathWorks Automotive Advisory Board [MAAB09; JMAAB09] oder die Richtlinien und Empfehlungen des VDA (Verband der Automobilindustrie e. V.) [VDA09]. Auch aktuelle Forschungsthemen adressieren Standardkonformität, wie beispielsweise eine vereinheitlichte Architekturbeschreibungssprache (‚*EAST-ADL*‘) [EAST08] oder ein standardisiertes Testverfahren (‚*TTCN-3 Embedded*‘) für eingebettete Systeme [SCHI09].

Ein bedeutender Aspekt dieser Arbeit und zugleich eine große ingenieurwissenschaftliche Herausforderung stellt der kollaborativ durchgeführte Engineering-Prozess dar, in dem viele Akteure des Produktentstehungsprozesses interdisziplinär involviert sind. Hier besteht die nicht triviale Problemstellung, wie die jeweilige Konformität zu geltenden Richtlinien aus Normungen (Erfüllung der Anforderungen) sowie die standardisierten Arbeitsergebnisse durch ein Verfahren übergreifend nachzuweisen sind. Das Verfahren muss dabei die Heterogenität einer bestehenden IT-Infrastruktur bewältigen sowie neben den elektronischen Arbeitsergebnissen auch prozessrelevante Informationen im Nachweis berücksichtigen.

Der Konformitätsnachweis ist für den Hersteller (Auftraggeber) von besonderem Interesse, da dieser von seinen Lieferanten (Auftragnehmern) die Bestätigung erhalten möchte, dass alle technischen Vorgaben für die abgelieferten Arbeitsprodukte (Daten, Dokumente, Modelle und Testspezifikationen usw.) befolgt wurden. Der Konformitätsnachweis ist aber auch für den jeweiligen Lieferanten (Auftragnehmer) von besonderem Interesse, da dieser dem Auftraggeber bereits vor einer Beauftragung nachweisen möchte, dass er dessen Anforderungen durch ein werkzeuggestütztes Verfahren nachweislich und auch zeitgerecht wird umsetzen können.

1.1 Richtlinien in Entwicklungsprozessen

Die Erfüllung festgelegter Anforderungen im Prozess ist die *Konformität* zu geltenden Vorschriften oder zu technischen Normen sowie zu betriebsinternen Konventionen. Konformität ist dabei ein betriebswirtschaftliches Kriterium für eine Organisation. Im Entwicklungsprozess existieren *Richtlinien* für Prozesse (Guidelines) und *Anordnungen* (Instructions) zur Umsetzung der Anforderungen aus einer Richtlinie.

In dieser Arbeit werden nur Richtlinien behandelt, die auch durch Anordnungen konkretisiert werden können. Neben gesetzlichen Regularien (EU-Normen, VDA-Richtlinien) gelten Regelwerke der Internationalen Organisation für Normung (ISO). Bekannt sind allgemeine Prozessrichtlinien für *Qualität* (das Qualitätsmodell) eines Unternehmens, z. B. vorgegeben durch ISO 9000er Regelwerke [ISO/DIN9000], domänenspezifische Vorgehensweisen, wie in der Automobiltechnik nach ISO/CD 26262 [ISO26262], technische Vorschriften im Bahnwesen nach DIN/EN 5012x [DIN50126], Richtlinien in der Luft- und Raumfahrttechnik nach RTCA DO-178B/DO-254 [DO178B] oder im militärischen Umfeld die Standards für Software nach DStan Def Stan 00-74 [DEFS08]. Des Weiteren regelt das Deutsche Institut für Normung (DIN) vielfache Standards [DIN09] auch für administrative Prozesse einer Organisation, wie z. B. die DIN 5007 Sortierregeln (Ordnen von Schriftzeichenfolgen) für die konforme Ablage, zur Anwendung im Schriftverkehr [DIN5007]. Für die Prozessverbesserung spezieller technischer Software-Entwicklungsprozesse gelten zudem noch Anforderungen aus Reifegradmodellen, wie dem international angewandten *‚Capability Maturity Model Integration‘* (CMMI) [CMMI12] oder dem europäischen ‚*Software Process Improvement and Capability Determination‘* (SPiCE) ISO-Standard [ISO15504].

Seitens der Unternehmensführung wird in den genannten Domänen die konforme Einhaltung der jeweilig geltenden Richtlinien im kollaborativen Produktentstehungsprozess, umzusetzen durch Mitarbeiter (Akteure), als grundlegend vorausgesetzt. Dies erfordert heutzutage viele Abstimmungsprozesse der involvierten Akteure im Prozess sowie einen sehr hohen manuellen Review-Aufwand für elektronische Arbeitsergebnisse, den *Artefakten*: Davon betroffen sind elektronische Dokumente und Formulare, Datenbestände (Datenbanken), Modelle, Codes bis hin zu Designs oder Konstruktionszeichnungen (CAD). Reviews erfolgen meist nur stichprobenartig, sodass Risiken unentdeckter Fehler stets bestehen. Schließlich erfolgt erst eine zu späte Aufdeckung von Fehlern während durchgeführter Prozessaudits bzw. Assessments zur Zertifizierung der Prozesse. Die prozessübergreifende Ursachenerkennung sowie die Nachverfolgung eines Fehlers sind oftmals schwierig bis gar unmöglich.

1.2 Aus Richtlinien abgeleitete Anforderungen an Artefakte

Im Produktentstehungsprozess wird eine Vielzahl von computergestützten Entwicklungs-, Simulations- und Testwerkzeugen eingesetzt [MUTZ05]. Hierbei entstehen elektronische Arbeitsergebnisse wie Produktanforderungen, Spezifikationen und später verbindliche Lasten-/Pflichtenhefte, auf denen nachgelagerte Prozesse (z. B. bei einem Zulieferer) aufbauen. Zusätzlich enthalten mitgeltende Dokumente begleitende Prozessinformationen, welche für die Erstellung der Arbeitsergebnisse übergreifend gelten.

Die *modellbasierte Entwicklung* ist ein zunehmend im Bereich der eingebetteten Systementwicklung verfolgtes, in der Automobilindustrie noch junges, Paradigma [CON04]. Hierbei geschieht eine Systementwicklung maßgeblich unter Verwendung von Operationen auf einem konzeptuellen Produktmodell, das alle in der Entwicklung erstellten Informationen integriert. Prozesslogisch bedingt stehen hierbei Arbeitsergebnisse in Form von Daten in den jeweiligen Arbeitsschritten unmittelbar in Abhängigkeit zueinander. Besonders durch Automatisierung verknüpfte oder gewonnene Daten hängen unmittelbar voneinander ab. Dies ist am Beispiel Codegenerierung besonders deutlich, wo der Code aus einem Modell heraus durch Transformationsvorschrift gewonnen wird [TKL06]. Die Konformität des Codes hängt dabei implizit vom Modell ab. Gelten als Anforderung zusätzlich in einem Glossar abgelegte bereichsübergreifende Konventionen für das Modell und den Code, wie

z. B. genormte Variablenbezeichner (Präfix, Akronym, Zeichensatz, Namenslänge, Typ), so müssen insgesamt Modell, Code und Glossar zueinander konsistent und konform zu der geltenden Konvention sein, also in einer Untersuchung mit einbezogen werden (vgl. auch [BOIS06]). Die Kenntnis von übergreifenden Informationen aus dem Engineering-Kontext (mitgeltende Dokumente) erlaubt erst eine Aussage über die Erfüllung der Anforderung aus einer Richtlinie. Es ist somit evident, dass die reine Modell- bzw. Codeanalyse für eine prozessübergreifende Richtlinienkonformität in einer kollaborativen Werkzeugkette nicht ausreichend ist.

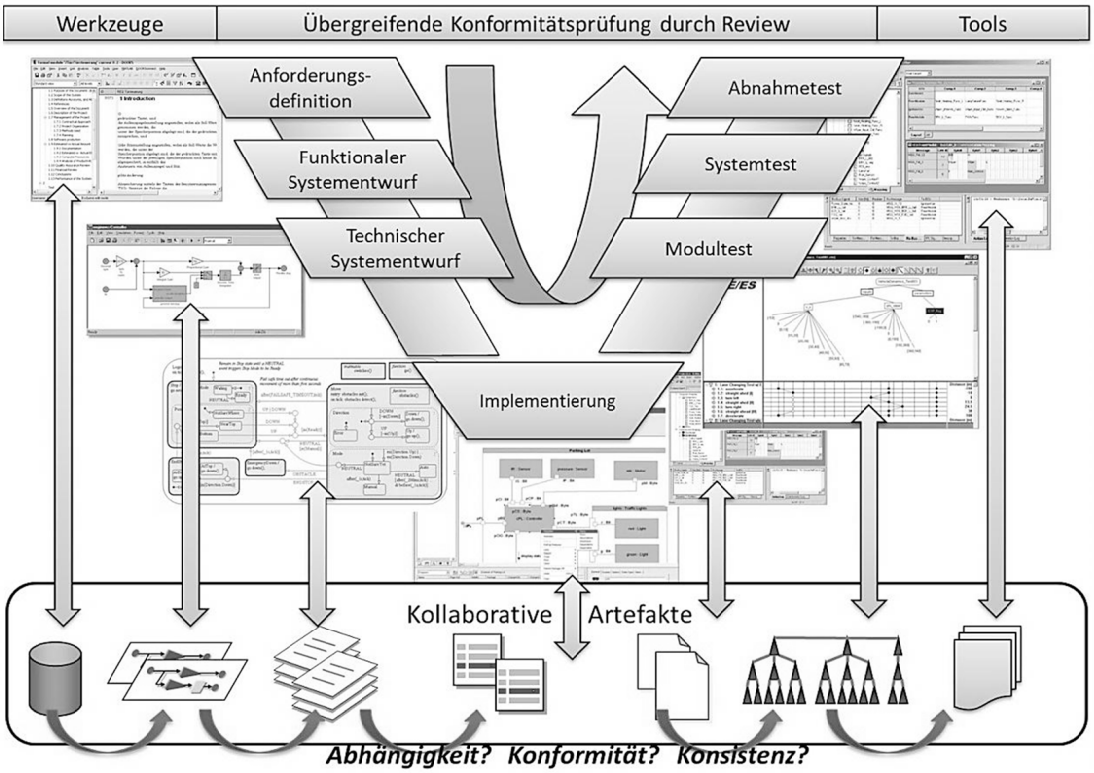


Abbildung 1: Logische Abhängigkeiten von Artefakten durch Richtlinien (V-Modell)

Prozessrichtlinien bringen Artefakte logisch in Abhängigkeit (Abbildung 1): *Kollaborative Artefakte* sind durch Richtlinien logisch verbundene, von mehreren Akteuren bearbeitete elektronische Arbeitsergebnisse. Entwicklungsrichtlinien legen für kollaborative Artefakte fest, wie Spezifikationen formuliert, Architekturen modelliert, Dokumente bearbeitet, Simulationen entworfen, eingebettete Software implementiert oder Daten in spezieller Formatierung für die Testautomatisierung abgelegt werden.

Die Bedeutung richtlinienkonformer Artefakte gerät somit insbesondere bei kollaborativ durchgeführten Prozessen in den Fokus, da hier Artefakte als Spezifikation – typisch im modellbasierten Entwicklungsprozess – für nachgelagerte Arbeitsschritte einwandfrei zur Weiterverarbeitung freigegeben werden sollen oder aber selbst eine oftmals zugelieferte Produktfunktion später umsetzen. Richtlinien betreffen somit die IT-Infrastruktur in der Handhabung von Programmen sowie *werkzeugübergreifend* alle durch Akteure bearbeiteten Arbeitsergebnisse (Architekturdesign, Systemlayout, Software, Testfälle usw.).

1.3 Aufwände, Risiken und Konformitätsprobleme

Der Komplexität kollaborativer Entwicklungsprozesse begegnen heutige Unternehmen mit dem Einsatz moderner Informationstechnologie (IT) sowie Prüfwerkzeugen. Durch die verschiedensten Konstellationen der jeweiligen Akteure in den Engineering-Prozessen sind jedoch die Methoden und die darin eingesetzten Werkzeuge (Computerprogramme) unterschiedlich. Daraus resultieren auch unterschiedliche Artefakte. So kann beispielsweise ein Implementierungsmodell eines eingebetteten Systems beim Hersteller in unterschiedlichen Modellierungssprachen vorliegen, je nachdem, welcher Lieferant für die Entwicklung zuständig war. Unabhängig von der Beschaffenheit des Artefakts ist jedoch gleichermaßen an allen heterogenen Artefakten nachzuweisen, dass geltende Richtlinien eingehalten wurden und somit die Arbeitsergebnisse fehlerfrei sind.

Gartner, ein Anbieter der Marktforschungsergebnisse und Analysen über die Entwicklungen in der IT durchführt, kommt zu der Aussage, dass mehr als 25 % der kritischen Daten in heutigen TOP-Unternehmen fehlerbehaftet sind [GAR07]. In einer von Gartner durchgeführten Studie *‚Magic Quadrant for Data Quality Tools‘* wird insbesondere belegt, dass Unternehmen vermehrt auf Datenqualität zu achten haben und der IT-Markt mit Prüflösungen zur Erhöhung der Datenqualität generell ansteigt [GAR10].

In [RUSS06], einer vom TDWI (The Data Warehousing Institute) durchgeführten IT-Studie *‚Consensus-Driven Data Definitions for Cross-Application Consistency‘,* nannten 83 % der befragten Teilnehmer, dass in ihrem Unternehmen bei werkzeugübergreifenden Arbeitsschritten die fehlerhaften Ausgangsdaten häufig eine Problemursache darstellen. Wiederum 54 % berichteten, dass ihre Prozesse aufgrund guter Ausgangsdaten profitierten. In der Studie sind weitere Umfrageergebnisse aufgeführt, welche zuallererst das ungenaue Berichtswesen (81 %) sowie die Eignung und Glaubwürdigkeit von Datenquellen (78 %) für die Folgen mangelnder Datenqualität in kollaborativen Prozessen verantwortlich machen. Aus den Aussagen der Studienteilnehmer lässt sich folgern, dass Konformitätsprüfungen bereits in den frühen Phasen (an den Ausgangsdaten) präventiv durchgeführt werden sollten, damit Folgefehler bzw. die Fehlerfortpflanzung vermieden werden.

In dieser Arbeit betrachten wir insbesondere den modellbasierten Entwicklungsprozess eingebetteter Systeme. In [ADL07], einer von Arthur D. Little und der TU-München durchgeführten Umfrage *‚Studie zur Kosten-/Nutzenanalyse der modellbasierten Softwareentwicklung im Automobil‘* kristallisiert sich eine noch drastischere Situation: 100 % der befragten Softwarehersteller äußern, dass die Modell-Konsistenz nur teilweise gegeben ist und 75 % äußern die Notwendigkeit für intensiver ausgearbeitete Konsistenzüberprüfungsmechanismen. Dies begründet den Bedarf und zugleich die Motivation, heute angewandte Konformitätsprüfungen kritisch zu hinterfragen und einen Evolutionsschritt zu schaffen.

In der modellbasierten Entwicklung eingebetteter Systeme hat der Einsatz von Software-Werkzeugen und generell der IT zur Folge, dass verschiedene gesetzliche wie auch unternehmensinterne Entwicklungsvorschriften zur Sicherung konformer Arbeitsergebnisse berücksichtigt und auf der Artefakt-Ebene (Datenqualität) eingehalten werden müssen. Der ISO-Standard ISO/CD 26262 [ISO26262] beispielsweise fordert im Band 8: *‚Supporting processes‘* eine Verifikation (Plan und Nachweis) für Arbeitsergebnisse sowie die nachgewiesenen Ergebnisse der zur Überprüfung angewandten Maßnahmen, um eine Software-Komponente auf Basis von Anforderungen zu qualifizieren. Aufgrund der engen Verzahnung von Produktentwicklungsprozess mit der IT erlangen zunehmend mehr Konformitätsanforderungen die Gültigkeit für logisch abhängige Artefakte. Die Schwierigkeit liegt in der Einhaltung von Konformität und Konsistenz bei werkzeug-

übergreifenden Prozessanforderungen (Information aus Prozess und Information aus Werkzeug), welche mit den aktuell verfügbaren Prüfwerkzeugen nicht automatisiert und methodisch unterstützt überprüfbar sind.

Die Tabelle 1 gibt typische Beispiele heutiger Probleme in der modellbasierten Entwicklung eingebetteter Systeme pro Prozessphase. Im Kapitel 3.5 wird die Konformitätsproblematik weiter vertieft.

Tabelle 1: Beobachtete Probleme in kollaborativen Prozessen

	Engineering-Prozess	***Richtlinie: Anforderung und resultierendes Problem***
(1)	Anforderungsdefinition	Zur Sicherstellung der Nachvollziehbarkeit muss der Bezug von Anforderungen zu Modellen und zu Testfällen hergestellt werden. Aus den daraus entstehenden logischen Beziehungen lässt sich pro Artefakt jederzeit ableiten, welches Modellelement und welcher Testfall welche Anforderung abdeckt. In der Praxis werden hierbei auf unterschiedlichste Art und Weise Bezugsreferenzen in den Artefakten hinterlegt, die werkzeugübergreifend funktionieren müssen. Dies geschieht durch konsistente Nummerierungen (Anforderungs- oder Bauteilenummern) sowie gleichartige Funktionsbezeichnungen und physikalische Einheiten gleicher Dimension, welche durchgängig im Entwicklungsprozess verwendet werden. Da jedoch alle beteiligten Artefakte in kollaborativen Prozessen Änderungen unterworfen sind, werden die Referenzen schnell inkonsistent und ein Entwickler kann Verstöße nur sehr mühsam ausfindig machen. Des Weiteren gibt es Regeln für die spezifizierte Sicherheits-anforderungsstufe sicherheitskritischer Funktionen in den Anforderungsdokumenten nach ASIL (Automotive Safety Integrity Level). Wird eine bestimmte Einstufung vorgenommen, müssen weitere sicherheitsrelevante Funktionen im Artefakt beschrieben und weitere Maßnahmen (Aktionen, Analysen, Abschätzungen) in separaten Dokumenten konsistent dokumentiert werden.
(2)	Modellbasierter Systementwurf	Zur Vermeidung von Problemen bei der Codegenerierung und zur Erhöhung von Verständlichkeit sollen einheitliche Namenskonventionen bereits im Entwurf angewendet werden. Die Konventionen werden für gewöhnlich mit vorgegebenen Worten oder Akronymen in einem Glossar hinterlegt. Problematisch ist hierbei der Abgleich pro Artefakt (z. B. Modell) zu dem Glossarinhalt, ob alle Bezeichner konform zum Glossar vergeben wurden. Ändert sich das Glossar (z. B. pro Projekt oder pro Kunde), werden die logischen Beziehungen zu den Artefakten schnell inkonsistent und der Entwickler muss alle ungültigen Bezeichner händisch auf Gleichheit prüfen.

(3) Modulspezifikation nach Systementwurf	Zur Sicherstellung funktionssicherer Software sowie zur Gewinnung effizienten Codes gelten unterschiedliche Anforderungen, z. B. dürfen gewisse Datentypen und reservierte Worte der Programmsprachen nicht verwendet werden sowie Normwerte, Wertebereiche von Variablen oder Varianzen nicht über- oder unterschritten werden. In der Praxis werden solche Vorgaben an die Modellierung in separaten Dokumenten begleitend festgehalten. Der manuelle Abgleich aller Variablen im Modell mit den Vorgabedokumenten stellt sich bei komplexen Modellen als überaus zeitaufwendig heraus. Auch Vorgaben zur effizienten Modellierung, wie z. B. eine festgelegte Rechenkette, die maximale Anzahl von Hierarchie-Ebenen im Modell oder die Anzahl an Schnittstellen können nicht im Modellierungswerkzeug begrenzt werden, sondern sind meistens in einem weiteren Artefakt hinterlegt. Auch hier muss der werkzeugübergreifende Bezug stets konform zur Anforderung (Vorgabe) sein und durch einen Review vollständig (meist manuell) geprüft werden.
(4) Implementierungsphase nach Systementwurf	Software-Entwicklungsprojekte bestehen immer aus mehreren Artefakten: Datentypdefinitionen, Schnittstellen, Bibliotheksfunktionen und Programmteile werden separat voneinander programmiert (z. B. AUTOSAR-Standard), referenzieren sich jedoch wechselseitig. Problematisch ist hier die Einhaltung der Konsistenz, ob nur die festgelegten Datentypen und Schnittstellen im Programm verwendet wurden, ob mehrfache Definitionen (ein Überladen) vorkommen oder wechselseitige Referenzen eine Rekursion hervorrufen können. Alle Referenzen zwischen den Artefakten (Modell zu Code und Code zu Code) müssen stets auf Fehler oder Anomalien überprüft werden.
(5) Testspezifikation	Bei der Testfallspezifikation besteht sofort eine logische Abhängigkeit zu anderen Artefakten: der Anforderung, dem zu testenden Modell oder dem zu testenden Code. Wenn in einer Anforderung eine Signalbeschreibung gefunden wird, sollte eine korrespondierte Schnittstelle (in Simulink: Inport- oder Outport-Block) die dieses Signal führt im Simulink-Modell existieren und dieses Signal sollte als Testfall-Klassifikation in mindestens einem Testfall auftreten. Hierbei müssen Testfall und das zu testende Artefakt stets konsistent zueinander sein. Ändert sich das zu testende Artefakt, so muss meistens auch die Testfallspezifikation angepasst werden. Auch bei der Auslegung und Parametrisierung von Testfällen gibt es technische Vorgaben für das Testsystem, welche in separaten Artefakten artefaktübergreifend hinterlegt sind.

In zusammenhängenden Prozessphasen sind hohe Review-Aufwände und fehleranfällige Tätigkeiten (nur Stichproben) zu beobachten. Informationen sind auf mehrere Dateien in unterschiedlichen Formaten verstreut. Die aufgeführten Praxisbeispiele treten jedoch nicht nur in der modellbasierten Entwicklung auf. Im Kapitel 6 ist daher auch ein Beispiel aus einem nichttechnischen Prozess (Rechnungsstellung) gegeben.

Eine Komplikation beobachtet man immer beim Einsatz unterschiedlicher Softwarewerkzeuge sowie den damit entwickelten kollaborativen Artefakten. Kurz umrissen resultieren Probleme aus den folgenden Prozessanforderungen:

- **Konsistente Datenbestände**, damit nachgelagerte Arbeitsschritte zyklenfrei ausgeführt werden können und schließlich die Nachvollziehbarkeit sichern.
- **Konforme Anwendung von Entwicklungswerkzeugen**, da diese teils einen hohen Kreativitätsgrad im Produktdesign/Parametrierung durch den Anwender erlauben.
- **Richtlinienkonform entwickelte Artefakte**, wie z. B. die Einhaltung von MISRA Software-Konventionen für eingebettete Software [WARD06; MISRA09].
- **Standardisierte Entwicklungsartefakte**, die einen konformen Austausch in kollaborativen Umgebungen erlauben, z. B. konformer Austausch von Anforderungsdokumenten mit *Requirements Interchange Format* (RIF) [RIF05].

Aus der analytischen Betrachtung im Kontext der Entwicklung eingebetteter Systeme geltender Richtlinien und implizierter Anforderungen an die kollaborativen Artefakte resultieren heutzutage Probleme in der Absicherung konformer Prozesse und insbesondere konformer Arbeitsergebnisse. Zusammengefasst sind dies die folgenden Probleme:

- Viele **interdisziplinäre Abstimmungen** von Akteuren sind erforderlich, um noch konsistente Arbeitsergebnisse in kollaborativer Umgebung zu ermöglichen.
- Durch **manuell durchgeführte Sichtprüfung** (*Reviews*) bleiben Fehler unentdeckt und bergen versteckte Risiken, insbesondere für nachgelagerte Prozesse.
- Ein **hoher, manueller Review-Aufwand** von elektronischen Dokumenten ist zur Sicherstellung von Konformität, über Werkzeuggrenzen hinaus, notwendig.
- Missverständnisse durch **Fehlinterpretation des Menschen**, bedingt durch Auslegungsfreiheiten von textuellen Richtlinien, führen zu Anwendungsfehlern.
- Review-Abläufe und daraus erfolgte Entscheidungen, z. B. bei Umgehung einer Richtlinie, sind nicht ausreichend dokumentiert. Dies erschwert später die **Nachvollziehbarkeit im Prozess**, wie Entscheidungspfade entstanden sind.
- Zum Bestehen von Prozessaudits ist der **Konformitätsnachweis** von Arbeitsschritten und -ergebnissen zu erbringen, der oftmals schwach ‚belastbar' ist.
- Die Forderung aus höheren **Reifegradstufen** (in CMMI, SPiCE) zur Etablierung sich selbst-optimierender Prozesse ist schwer durch manuelle Schritte erreichbar.

Folglich stellt aus prozessübergreifender Sicht, z. B. aus der eines Managers oder Leiters, die Prozessüberprüfung und die Konformitätsbewertung von Arbeitsergebnissen für eine belastbare Bestätigung der Richtlinienerfüllung und zur Einschätzung des Konformitätsgrades eine nicht triviale Herausforderung dar. Der stetige Nachweis, dass Entwicklungsprozesse konform zu geltenden Richtlinien ablaufen und somit auch erfüllt werden, wird langläufig auch durch den Begriff ‚*Compliance*' institutioniert. Compliance in der Informationstechnologie wird als eine weitreichende Unternehmensaufgabe verstanden

[KRAN09]. Bezogen auf die Entwicklung technischer Produkte betrifft dies eine ganzheitlich Einhaltung aller relevanten Vorschriften für die IT-gestützte Produktauslegung sowie die Beachtung gesetzlicher Vorgaben, der dazugehörigen internen Prozessrichtlinien, der konformen Produktionsverfahren sowie der Selbstverpflichtungen zur Erfüllung im Produktentstehungsprozess. Konformität − die Erfüllung von Anforderungen − ist heute ein wichtiges Thema für jedes produzierende Unternehmen (vgl. auch [BIT05; GRAM08; GRÜN09; KRAN09]).

1.4 Lösungsansatz und Zielstellung

Die Problematik heutiger Ansätze zur regelbasierten Konformitätsprüfung durch Prüfwerkzeuge (*Checker*) ist die fehlende Abstraktionsfähigkeit von dem Prüfling (Artefakt-Typ), welchen sie hinsichtlich Konformitätseigenschaften prüfen sowie die fehlenden Kontextinformationen, welche zusätzlich für einen Konformitätsnachweis erforderlich sind. Text-Analyzer prüfen Dokumente, Modell-Checker prüfen Modelle und Code-Checker prüfen Programmcode. Heutige Prüfprogramme, sofern sie für eine Prozessphase überhaupt existieren, sind Artefakt-spezifisch für einen speziellen Zweck ausgelegt. Sie verwenden eine werkzeuginterne Programmierung für die Analyse eines proprietären Dateiformats und sind in ihrer Wirkung auf nur einen Artefakt-Typ limitiert. Anforderungen aus Prozessen sind hingegen universeller, sind im Allgemeinen durch textuelle Richtlinien weitreichender verfasst, adressieren logische Zusammenhänge der Artefakte und implizieren konsistente Beziehungen zwischen ihnen − fernab von Grenzen oder Inkompatibilität einer IT-Umgebung.

Zusätzlich existieren in kollaborativen Prozessen artefaktübergreifende Konstrukte, wie z. B. das Konstrukt ‚*Fahrzeugfunktion*'. Diese sind nicht durch das Artefakt, sondern durch den Prozess bestimmt. Inhaltlich werden sie auf mehrere Artefakte abgebildet, z. B. in einer Anforderung, dem Funktionsmodell, der Softwarearchitektur sowie der Testspezifikation. Fordert eine Richtlinie (z. B. „*Namenskonvention für Fahrzeugfunktion*") die Konformität aller Bezeichner einer Fahrzeugfunktion, gilt diese Anforderung auf die verschiedenen Entwicklungsartefakte gleichermaßen und muss durch diese erfüllt werden. Die Konformität steht in unserer Betrachtung in Relation zu einer textuellen Spezifikation, dem logisch verbundenen Verhaltensmodell, dem zugehörigen Programmcode und der dazugehörigen Testspezifikation.

Heutzutage fehlt es demnach an der Möglichkeit, eine automatisierte Konformitätsprüfung an kollaborativen Artefakten durchzuführen, welche eine definierte Referenz von Richtlinie zu Artefakt-spezifischer bis hin zu logischer Ausprägung berücksichtigt. Im Spannungsfeld der meist sehr heterogenen IT-Infrastruktur im Entwicklungsprozess und der immer wichtiger werdenden Erfüllung von Konformität zu vordefinierten Entwicklungsrichtlinien orientiert sich die Arbeit an zwei ingenieurwissenschaftlichen Fragestellungen:

I. *Wie lassen sich prozesslogische Konstrukte, prozessbedingte Abhängigkeiten und virtuelle Konsistenzbeziehungen auf kollaborative Artefakten einer heterogenen IT-Infrastruktur beschreiben? Kann ferner eine Abstraktion von Artefakt-spezifischen Eigenarten hin zu prozessspezifischen Richtlinienformulierungen durchgeführt werden?*

II. *Wie lassen sich prozesslogische, nicht-funktionale Konformitätsforderungen aus Entwicklungsrichtlinien in einer heterogenen IT-Infrastruktur an kollaborativen Artefakten werkzeugübergreifend prüfen?*

Des Weiteren müssen tangierende wissenschaftliche wie auch technische Fragestellungen untersucht werden, wie die Automatisierung der Konformitätsprüfung an kollaborativen Artefakten bis hin zu einer Bewertung nicht-funktionaler Qualitätscharakteristika möglich sein kann, wie die regelbasierte Konformitätsprüfung im heterogenen Umfeld skaliert und wo Grenzen und Unlösbarkeiten eine Automatisierung verhindern. In dieser Arbeit wird der Lösungsansatz an den kollaborativen Prozessen und Arbeitsergebnissen (Daten) im Umfeld der modellbasierten Entwicklung eingebetteter Systeme demonstriert, gleichwohl auf die Übertragbarkeit des Ansatzes auf artfremde Prozesse und Domänen eingegangen wird. Ausgehend von regelbasierten Ansätzen zur statischen Analyse einzelner Artefakte basiert die Idee dieser Arbeit auf der Erweiterung der rein singulären Artefakt-Prüfung hin zu einer werkzeugübergreifenden, kollaborativen Artefakt-Prüfung mehrerer, prozesslogisch abhängiger Artefakte durch das Konzept des virtuellen, plattformunabhängigen Datenmodells, des so genannten *logischen Artefakts* (Kapitel 4.3).

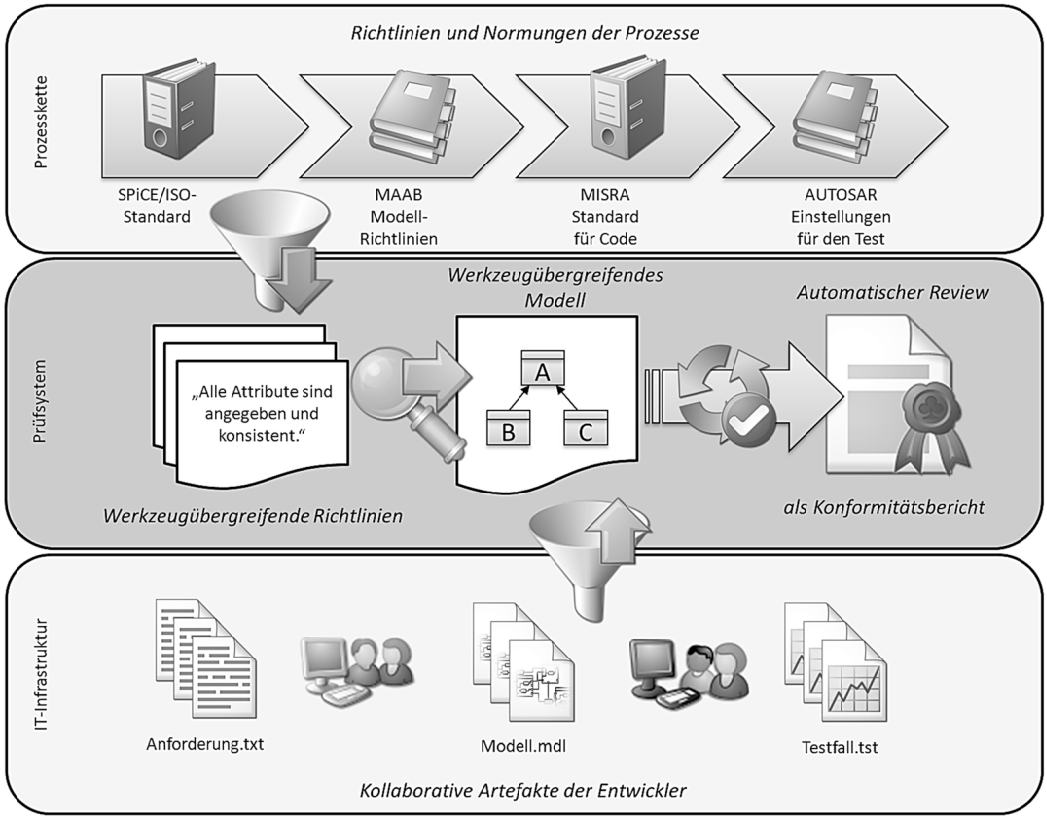


Abbildung 2: Regelbasierter Review an kollaborativen Artefakten

Durch Abstraktion und Formalisierung kollaborativer Artefakte und natürlich sprachlicher Richtlinien in plattformunabhängige Datenmodelle und digitale *Regeln* (computerausführbare Algorithmen) sollen interdisziplinäre Entwicklungsrichtlinien und ihre semantischen Abhängigkeiten auf dem *logischen Artefakt* (ein virtuelles Datenmodell) beschrieben, ausgeführt und schließlich Entwicklungsrichtlinien einer heterogenen IT-Infrastruktur hinsichtlich Konformität durch ein *Prüfsystem* automatisiert prüfbar werden.

In der Arbeit werden zunächst die *Analyse* sowie die *Abbildung* der Semantik von interdisziplinären Entwicklungsrichtlinien auf diverse IT-Programme und Daten im Themenkontext durchgeführt. In darauf folgenden Schritten werden *übergreifende Richtlinien* klassifiziert und exemplarisch formal als *Logikausdrücke* definiert. Durch die

nachfolgende Umsetzung der formalisierten Richtlinien in computerausführbare Algorithmen (*Regeln*) einer Programmiersprache kann die automatisierte Erkennung von Fehlern und Anomalien bei kollaborativen Artefakten an realen Beispielen erfolgen. In Erweiterung zu existenten Ansätzen statischer Analyse auf einzelnen Artefakten wird in dieser Arbeit ein erweiterter Prüfraum durch das Konzept des *logischen Artefakts* gebildet, das eine werkzeugübergreifende Konformitätsprüfung unabhängig von einer konkreten IT-Plattform erstmals ermöglicht.

Im Lösungsweg werden zunächst umgangssprachliche Prozessrichtlinien analysiert und durch syntaktische Dekomposition und semantische Attributierung mit kollaborativen Artefakten in Beziehung gesetzt. Hierfür wird eine spezielle Methodik entwickelt, welche ein *Abstraktionskonzept* und die *Metamodellierung* verwendet. Durch das Paradigma der Model Driven Architecture (MDA) werden *logische Artefakte* mittels direkter und indirekter *Transformation* gebildet. So wird aufgezeigt, wie sich auch komplexere Konformitäts-prüfungen auf einer abstrakteren Ebene durchführen sowie abgeleitete Aussagen auf spezifische Artefakte bestimmen lassen. Zudem wird eine *Methodik* erarbeitet, welche mathematische Beschreibungsmittel wie *Logik* für die Formalisierung der Richtlinien anwendet, um schließlich die Ableitung von umgangssprachlichen Richtlinien zu computerausführbaren Ausdrücken und Algorithmen für die statische Analyse *logischer Artefakte* zu ermöglichen.

Zur *Strukturierung* von Regelalgorithmen wird eine Auszeichnungssprache *Query-based Rule Description Language (QRDL)* eigens konzipiert, um hieraus die nachfolgende Regelimplementierung in diversen Programmiersprachen zu unterstützen. In den gewählten Programmiersprachen LINQ, OCL und M-Skript werden dann jeweils *Regeln* aus existenten Entwicklungsrichtlinien und Industrienormen abgeleitet und exemplarisch implementiert.

Zwecks Automatisierung des Ansatzes werden die Implementierungen zweier *Prüfsysteme* vorgestellt, welche die Regelimplementierungen anwenden. Das jeweilige Prüfsystem ermöglicht die werkzeuggestützte Konformitätsanalyse und Bewertung pro Prozessphase. Die Validierung der Ergebnisse erfolgt durch die probenhafte Umsetzung und Prüfung von Richtlinien an kollaborativen Artefakten aus einem modellbasierten Entwicklungsprozess.

1.5 Struktur der Arbeit

Die Arbeit strukturiert sich bezüglich der Aufgabenstellung und der Zielsetzung wie folgt:

Das **Kapitel 1** stellt den Kontext, die Problemstellung sowie den Zweck dieser Arbeit vor. Es werden die besonderen Herausforderungen für die Richtlinienkonformität bei der modellbasierten Entwicklung eingebetteter Systeme und die Anforderungen an kollaborative Artefakte skizziert. Aus der beobachteten Problematik zur Konformität von kollaborativen Prozessen und der Artefakte erwachsen ingenieurwissenschaftliche Fragestellungen, die Zielsetzung sowie der Lösungsansatz der Arbeit.

Das **Kapitel 2** vermittelt die Grundlagen zur Durchführung der Arbeit. Es werden grundlegende Begriffe eingeführt und der modellbasierte Entwicklungsprozess vorgestellt. Zusätzlich benötigte Formalisierungstechniken, mathematische Beschreibungsmittel, Abstraktionsparadigma und Abfragesprachen bilden das Fundament für den Lösungsweg. Auszeichnungs- und Abfragesprachen aus der Informatik werden vorgestellt und hinsichtlich ihrer Zweckmäßigkeit diskutiert, welche dann im achten Kapitel Anwendung finden.

Durch eine Situationsanalyse führt das **Kapitel 3** in den Kontext der modellbasierten Entwicklung in der Automotive-Domäne ein. Ausgesuchte Prozesse, Werkzeuge und kollaborative Artefakte werden vorgestellt. Typische Normen und Standards werden referenziert und Konformitätsprobleme ausgewiesen. Der Stand von Wissenschaft und Technik sowie die Einbettung des Ansatzes in das wissenschaftliche Umfeld schließen das dritte Kapitel ab.

Das **Kapitel 4** fundamentiert das Profil von Artefakten und Richtlinien. Nebst konkreten Definitionen werden Richtlinien anhand eines Qualitätsmodells klassifiziert und zu Prozessphasen eingruppiert. Aus aufgestellten Anforderungen an die regelbasierte Konformitätsprüfung werden der Ansatz und der Lösungsweg zur regelbasierten Konformitätsprüfung kollaborativer Artefakte vorgestellt. Der Geltungsbereich des Ansatzes schließt das vierte Kapitel ab.

Im **Kapitel 5** wird eine Regelsprache namens *‚Query-based Rule Description Language‘* (QRDL) vorgestellt, welche genau die strukturierte und abfrageorientierte Auszeichnungsgrammatik für Artefakt-übergreifende Konformitätsprüfungen konstatiert, mittels derer sich eine komplexere bzw. artefaktübergreifende Regel-Logik mittels einer Programmstruktur (Regelalgorithmus) aufbauen lässt.

Eine deduktive Methodik samt Vorgehensmodell (VR-Modell) zur Regelentwicklung basierend auf natürlichen sprachlichen Prozessrichtlinien ist Gegenstand im **Kapitel 6**. An Abstraktionsbeispielen wird aufgezeigt, wie sich Richtlinien formalisieren und kollaborative Artefakte mittels Transformationstechniken in logische Artefakte überführen lassen.

Die Konzipierung und Entwicklung von Prüfsystemen (Prüfwerkzeugen) zur automatischen Konformitätsprüfung kollaborativer Artefakte ist der Schwerpunkt von **Kapitel 7.** Technische Möglichkeiten und Randbedingungen werden flankierend erläutert sowie zwei Prüfwerkzeuge *ASD-Regel-Checker* und *Assessment Studio* vorgestellt.

In **Kapitel 8** wird die Validierung des Ansatzes an kollaborativen Artefakten in einem gewählten Szenario, der modellbasierten Entwicklung einer sicherheitsrelevanten Fahrzeugfunktion zum Insassenschutz, durchgeführt. Abgeleitet aus der eingeführten Strukturierung von Regeln werden jeweils Implementierungsbeispiele von Regeln in den Programmiersprachen LINQ, M-Skript und OCL in diesem Kapitel vorgestellt.

Das **Kapitel 9** fasst die Ergebnisse und Erkenntnisse der Arbeit zusammen und gibt einen Ausblick auf weiterführende Forschungs- und Entwicklungsfelder.

2 Kontext und Grundlagen

Das zweite Kapitel *Kontext und Grundlagen* beginnt mit der Definition und Präzisierung aller im Betrachtungsfeld liegender Begriffe und Terminologien dieser Arbeit. Ausgangspunkt für die Begriffsdefinitionen bildet die allgemeingültige Norm ISO/IEC 17000:2004 [ISO17000] zur Konformitätsbewertung in technischen Prozessen. Hieraus sind die Begriffe, das Verständnis und allgemeine Grundlagen in diesem Kapitel abgeleitet. Des Weiteren werden charakteristische Merkmale – Entwicklungsprozess und Prinzipien, Werkzeuge und Verfahrensweisen − in der modellbasierten Entwicklung eingebetteter Systeme vorgestellt. Die Analyse genereller Prinzipien und Entwicklungsschritte im Produktentstehungsprozess ist eine notwendige Voraussetzung für das Verständnis dieser Arbeit. Abschließend werden wesentliche Formalisierungstechniken, mathematische Beschreibungsmittel, Abstraktionsparadigma und Abfragesprachen beschrieben. Sie bilden das Fundament für den Lösungsweg. Da im ingenieursmäßigen Gebrauch von Begriffen häufig auch englische Bezeichnungen als Synonym Anwendung finden, werden zusätzlich auch die entsprechenden englischen Übersetzungen angegeben.

2.1 Einordnung der Grundbegriffe

Der Artefakt-Begriff entstammt dem lateinischen Wort ‚ars', ursprünglich für „Bearbeitung", und dem ‚facere' für „machen, herstellen". Er bezeichnet im Allgemeinen einen von Menschen hergestellten Gegenstand. In jedem nach einem Prozess durchgeführten Projekt entstehen diverse elektronische Arbeitsergebnisse, die als *Artefakte* bezeichnet werden. Die Einführung eines einheitlichen Prozesses dient oft dem Zweck, dass Artefakte möglichst einheitlich strukturiert sind, um sie vergleichbar und für jeden verständlich zu machen. Aus diesem Grund beschreiben Richtlinien für Artefakte, wie diese Artefakte auszusehen haben. In dem hier betrachteten Themenkontext sind Artefakte von einer oder von mehreren Personen in einem oder in mehreren Arbeitsschritten erstellte elektronische Dokumente, die Richtlinien unterliegen. Ein Artefakt wird demnach wir folgt definiert:

Artefakt ***(engl. artifact, work product, work item)*** **(2.1)**
„Von Personen bearbeitete, digitale Information in Form von Daten, welche in einem strukturierten Dateiformat vorliegen, werden als *Artefakt* bezeichnet. Wird ein Artefakt durch eine Anforderung aus einer Richtlinie prozesslogisch in Bezug zu einem weiteren Artefakt (z. B. einer anderen Person) gestellt, bezeichnen wir es als *kollaboratives Artefakt*."

Mit einem Computersystem erfasste Daten, welche in einem strukturierten Dateiformat vorliegen, gelten demnach als Artefakt. Ein Artefakt ist somit z. B. ein Office-Dokument, ein Architektur- oder Simulationsmodell, ein Quellcode oder generell ein beliebiger Datensatz, welcher als Zwischen- oder Endergebnis in einem Prozess den Arbeitsschritt darstellt. Ein Artefakt ist kein *Kompilat* (z. B. ausführbares Programm), da dies ursprünglich durch einen Compiler verarbeitet wurde und nicht durch eine Person. Artefakte sind aufgrund ihrer

Beschaffenheit und Heterogenität zu klassifizieren. Die Heterogenität (bzw. auch die Inhomogenität) von Artefakten bezeichnet die Uneinheitlichkeit der Elemente einer Menge hinsichtlich eines oder mehrerer Merkmale. Diese Merkmale sind bei einem Artefakt die Struktur des Artefakts (Metamodell) sowie die Form der Persistenz (das Dateiformat). Das Klassifizieren erfolgt aufgrund unterschiedlicher Dateiformen in (a) *Dokument*, (b) *Modell*, (c) *Software* (Code) und (d) *Datenbank*. Artefakte werden in mehreren Arbeitsschritten in einem Entwicklungsprozess kollaborativ, d. h. von einem oder auch von mehreren Personen, bearbeitet. Als *Artefakt-Vorlagen* bezeichnet man solche Artefakte, die vordefiniert im *Prozess* zur Verfügung stehen und von einer oder mehreren Person(en) auszufüllen sind (z. B. ein elektronisches Formular).

Prozess *(engl. process)* **(2.2)**
„Ein *Prozess* ***P*** ist eine zeitlich-sachlogische Abfolge von betriebswirtschaftlichen oder technischen Arbeitsschritten, die zur Bearbeitung eines Artefakts notwendig sind."

Wie in der Einleitung (vgl. Kapitel 1.1) beschrieben, gelten für einen Prozess allgemeine Arbeitsanweisungen, übliche Verfahrensweisen, so genannte *‚Best-Practices'*, für die Art der Bearbeitung von Artefakten. Folglich wird der Begriff *‚Richtlinie'* präzisiert.

Richtlinie *(engl. guideline)* **(2.3)**
„*Richtlinien* (Vorgaben, Vorschriften) sind unternehmensinterne, abstrakte oder sehr konkrete Handlungsanweisungen zur Befolgung im Prozess."

Die Definition der Richtlinie führt den Begriff *‚Handlungsanweisung'* ein. Dieser soll hierbei wie folgt als *‚Anweisung'* verstanden sein.

Anweisung *(engl. instruction)* **(2.4)**
„Eine *Anweisung* ist eine konkrete Handlungsvorschrift, wie eine Richtlinie umgesetzt wird."

Eine Anweisung stellt zugleich eine Anforderung aus der Richtlinie an den Prozess dar und ist als Synonym zu verstehen. Die *‚Prüfung'* von Entwicklungsrichtlinien ist ein zentrales Vorhaben dieser Arbeit, um eine Aussage zur Konformität eines kollaborativen Artefakts bzgl. der Erfüllung der Anforderung aus einer Richtlinie machen zu können.

Prüfung *(engl. examination)* **(2.5)**
„Eine *Prüfung* ist die Ermittlung eines oder mehrerer Merkmale an einem Artefakt zur Konformitätsbewertung nach einem Verfahren."

Da im Zuge einer durchgeführten Prüfung im Prozesskontext oftmals auch von *‚Review'* gesprochen wird, soll dieser Begriff nach [HOER06] ebenfalls konkretisiert werden.

Review *(engl. review)* **(2.6)**
„Formelle Prüfung eines Artefakts (z. B. Dokument, Modell oder Code) gegenüber Vorgaben und gültigen Richtlinien durch einen Gutachter mit dem Ziel, Fehler, Schwächen oder Lücken des Prüfobjekts aufzuzeigen, zu kommentieren und zu dokumentieren sowie den erwarteten Reifegrad des Artefakts festzustellen."

Untersuchungsverfahren, die dazu dienen, Engineering-Prozesse hinsichtlich der Erfüllung von Anforderungen aus Richtlinien zu bewerten, werden als Audit oder Assessment bezeichnet. In Qualitäts- und Reifegradmodellen (nach ISO; CMMI; SPiCE) werden Prozesse und deren Arbeitsergebnisse zu einem bestimmten Zeitpunkt überprüft (vgl. auch [KNEU06; HÖR06]). In den Qualitäts- und Reifegradmodellen sind hauptsächlich von Personen durchgeführte Untersuchungsverfahren gemeint. Wir verstehen ein Untersuchungsverfahren als Prüfung an einem kollaborativen Artefakt. Es ist daher erforderlich, die Begriffe *‚Audit‘* und *‚Assessment‘* einzuführen und zu präzisieren.

Audit ***(engl. audit)*** **(2.7)**
„Ein Audit ist ein systematischer, unabhängiger, dokumentierter Prozess zur Erlangung von Aufzeichnungen, Darlegungen von Fakten oder anderen relevanten Informationen und deren objektiver Begutachtung, um gezielt zu ermitteln, inwieweit festgelegte Anforderungen aus Richtlinien an einem kollaborativen Artefakt erfüllt sind.“

Assessment ***(engl. assessment)*** **(2.8)**
„Ein Assessment ist im Prozesskontext die Bewertung der Leistungsfähigkeit der Prozesse einer Organisation gegenüber einem geltenden Referenzmodell. Wird die Bewertung durch charakteristische Merkmale eines kollaborativen Artefakts beeinflusst, wird ein Assessment auf einem oder mehreren kollaborativen Artefakt(en) durchgeführt mit dem Ziel der Bewertung und Verbesserung des Prozesses zur Erstellung eines kollaborativen Artefakts.“

In dieser Arbeit werden demnach typische Begriffe aus der Prozesswelt hinsichtlich ihrer Anwendung auf kollaborative Artefakte betrachtet. Während in einem a) Audit strikte Prüfungen zur Ermittlung festgelegter Anforderungen aus Richtlinien an einem kollaborativen Artefakt durchgeführt werden, wird in einem b) Assessment eine Bewertung der Güte eines kollaborativen Artefakts unternommen. Die Güte ist typischerweise ein weiches Qualitätskriterium, wie Effizienz, Lesbarkeit oder Verständlichkeit des untersuchten Artefakts. Ziel von a) ist demnach die Erfüllung der Anforderung nachzuweisen, wobei Ziel von b) eine Bewertung zur Einschätzung der Güte darstellt.

Absicht dieser Arbeit ist die nachweisliche Überprüfung von Anforderungen aus den Prozessrichtlinien an kollaborativen Artefakten, d. h. deren Erfüllung zu ermitteln und zu dokumentieren. Die *‚Konformität‘* soll daher wie folgt verstanden werden:

Konformität ***(engl. conformance)*** **(2.9)**
„Die *Konformität* ist die Erfüllung einer Anforderung aus einer Prozessrichtlinie bezogen auf ein kollaboratives Artefakt.“

Die Anforderungen aus geltenden Prozessrichtlinien sind für kollaborative Artefakte häufig formale Kriterien an das Arbeitserzeugnis einer Person:

Es müssen Informationen in Abhängigkeit zu bestimmten Bedingungen angegeben sein oder es dürfen bestimmte Informationen eben nicht angegeben sein. Die Informationen müssen in ihrer Struktur und in ihrer Ausprägung einer bestimmten Vorgabe entsprechen. Informationen müssen über mehrere kollaborative Artefakte hinweg konsistent sein. Sofern möglich, müssen Informationen inhaltlich ausgewertet werden, um eine Bewertung zur Güte abgeben zu können. Die Erfüllung einer Anforderung bezieht sich nur auf Richtlinien und nicht etwa auf Systemspezifikationen, wo durch eine Validierung geprüft wird, ob eine textuelle Beschreibung durch ein Modell oder Code tatsächlich erfüllt wird.

Das Gegenteil von Konformität ist die Nonkonformität. Bezieht man Konformität auf die globale Einhaltung von Anforderungen (Richtlinien) im Unternehmen, wird von *‚Compliance‘* gesprochen. Für Unternehmen bedeutet dies die Einhaltung relevanter Gesetze sowie betriebsinterner Normen. Um ein präziseres Mittel zur Beschreibung von Konformität und Nichtkonformität zu etablieren, werden die Begriffe *‚Konformitätsprüfung‘*, *‚Fehler‘* und *‚Mangel‘* unterschieden.

Konformitätsprüfung/-bewertung ***(engl. conformance checking)*** **(2.10)**
„Darlegung durch ein Verfahren, dass die Konformität auf das kollaborative Artefakt zutrifft. Bei Bewertung, die Darlegung durch ein Verfahren der Beschaffenheit (mit Ermittlung der Güte und der Abweichung) bezogen auf die Anforderung.“

Eine allgemeingültige Verfahrensweise zur Konformitätsbewertung einer Einheit mit Nachweis der Erfüllung festgelegter Anforderungen wird nach [ISO17000], wie in der Abbildung 3 als Flussdiagramm dargestellt ist, beschrieben.

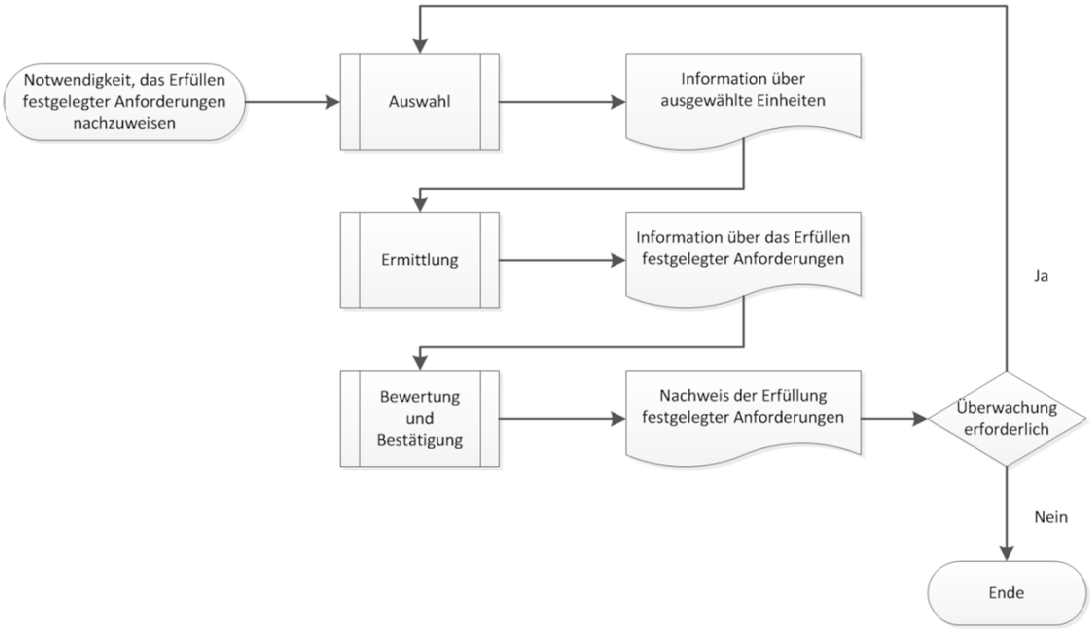


Abbildung 3: Funktionaler Ansatz für Konformitätsbewertungen, nach [ISO17000]

Ist die Konformität (Übereinstimmung mit einer Anforderung) nicht gegeben, d. h. verstößt ein Artefakt gegen eine im Regelwerk definierte Richtlinie, so ist das Artefakt *nonkonform*. Sprichwörtlich „verletzt“ es die Richtlinie. Das Artefakt hat einen *‚Fehler‘* oder *‚Mangel‘*.

Fehler, [Verstoß] ***(engl. error)*** **(2.11)**
„Nichterfüllung einer Anforderung.“

Mangel, [Schwachstelle, Lücke] ***(engl. defect)*** **(2.12)**
„Nichterfüllung einer Anforderung in Bezug auf einen beabsichtigten oder festgelegten Gebrauch.“

Die Unterscheidung von Fehler und Mangel wird hinsichtlich des Schweregrades getroffen. Ist z. B. die Angabe des Datums ein Pflichtattribut eines kollaborativen Artefakts, ist ein fehlender Datumseintrag ein Fehler. Wird nur das Jahr eingetragen, ist zwar die Anforderung formal erfüllt, jedoch in der Angabe unvollständig und stellt daher einen Mangel dar.

Der Übergang (Verbesserung) von Nichtkonformität zu Konformität eines Artefakts muss durch eine Maßnahme (Aktivität) erfolgen. Man spricht hierbei von *‚Korrektur‘*.

Korrektur ***(engl. correction)*** **(2.13)**
„Maßnahme zur Beseitigung eines erkannten Fehlers oder Mangels.“

Im Abschnitt 7.2 des Standard ISO 15489 (‚*Information and documentation – Records management standard‘*) [ISO15489] wird festgelegt, dass ein authentisches Dokument ein elektronisches Dokument ist, bei dem die Beweislage sicherstellt, dass a) es das Dokument ist, das es vorgibt zu sein; b) es von derjenigen Person erstellt oder versendet worden ist, die vorgibt, das Dokument erstellt oder versendet zu haben; c) es tatsächlich zu der angegebenen Zeit erstellt oder versendet wurde. Die Beurteilung der Authentizität eines Dokuments beinhaltet auch die Feststellung seiner *Identität* und die Demonstration seiner *Integrität*. Die Identität eines Dokuments bezieht sich auf seine charakteristischen Attribute (zum Beispiel: *Titel*, *Name* des Autors, *Datum* oder *Ort* der Dokumenterstellung), einschließlich seiner externen Attribute (zum Beispiel: *Kontext* und *Herkunft*), welche einzigartige Merkmale dieses Dokuments konstituieren und es von anderen Dokumenten unterscheidet.

Angelehnt an den Standard ISO 15489 [ISO15489] leitet sich die *‚Konsistenz‘* aus Integrität und Vollständigkeit eines elektronischen Dokuments ab und kann nun wie folgt für ein kollaboratives Artefakt so formuliert werden:

Konsistenz ***(engl. consistency)*** **(2.14)**
„Ein kollaboratives Artefakt ist *konsistent*, wenn es über den gesamten Zeitraum seiner Lebensdauer hinweg in allen seinen wesentlichen Merkmalen vollständig und unverfälscht ist. Dies bedeutet nicht, dass ein kollaboratives Artefakt genau dasselbe sein muss wie das ursprüngliche Artefakt, um seine Konsistenz sicherzustellen. Ein kollaboratives Artefakt kann als im Wesentlichen *vollständig* und *unverfälscht* betrachtet werden, wenn die für dessen Zweckbestimmung zu kommunizierende Botschaft (Information) unverändert bleibt. Mehrere kollaborative Artefakte sind *konsistent*, wenn sich festgelegte Informationen entsprechen.“

Konsistenz ist ein häufig auftretendes, notwendiges Qualitätskriterium für den Nachweis der Konformität. Zur Überprüfung von Konformität ist die Prüfung hinsichtlich der Konsistenzeigenschaften eines oder mehrerer kollaborativer Artefakte erforderlich.

Abschließend sollen noch zwei Arten einer Konformitätsprüfung unterschieden werden: Die *automatisierte Konformitätsprüfung* durch einen formalen Prüfalgorithmus, der von einer Maschine (einem Computer) ausgeführt wird und ein Prüfergebnis (Resultat) liefert. Die *manuelle Konformitätsprüfung*, oder *Sichtprüfung* durch einen Menschen, welcher aufgrund der Summe seiner Erfahrungen und seines Verstandes einen Gegenstand in Augenschein nimmt und ein Prüfergebnis (Resultat) erstellt. Die Auswertung einer Prüfung wird oftmals als *‚Kontrolle‘* oder *‚Überprüfung‘* bezeichnet und soll sinngemäß auch in dieser Arbeit so verstanden werden. Nachdem die Grundbegriffe nun eingeführt wurden, soll im folgenden Kapitel in den Kontext der Arbeit eingeführt werden.

2.2 Modellbasierter Entwicklungsprozess

Wie in der Einführung beschrieben, zeigt sich in der heutigen Automobilindustrie, Bahntechnik sowie im Luft- und Raumfahrtbereich ein deutlicher Trend: Die Elektronik im technischen Produkt spielt eine immer größer werdende Rolle. Immer mehr Funktionen verteilen sich auf immer mehr Steuergeräte – dabei gibt es Steuergeräte, auf denen mehrere Funktionen laufen, und Funktionen, die auf mehrere Steuergeräte verteilt sind [SCHÄ03]. Die Komplexität der Produktfunktionen wächst stetig an. Die Steuerung der komplexen Elektroniksysteme muss durch abgesicherte und überprüfte Softwarefunktionen erfolgen. Dies ist insbesondere für Ingenieure in der Entwicklung eingebetteter Systeme relevant, da sie die Verantwortung für das Systemkonzept und später für die Integration des eingebetteten Systems in das Gesamtsystem tragen. Mit der steigenden Anzahl an Steuergeräten müssen Entwickler auch ständig mehr und immer komplexere Funktionen entwerfen, codieren, testen und implementieren. Um in diesem zeit- und kostenkritischen Bereich, in dem es vor allem auf Qualität und Flexibilität ankommt, wettbewerbsfähig zu bleiben, ist der Entwickler auf die ständige Optimierung aller Entwicklungsaktivitäten angewiesen. Nachfolgend werden der Entwicklungsprozess nach dem V-Modell und die integrierten Prozessphasen der Systementwicklung eingebetteter Systeme eingeführt.

2.2.1 Prozessmodell (V-Modell)

Der gegenwärtig praktizierte modellbasierte Entwicklungsprozess für eingebettete Systeme ist in vielen Industriezweigen integriert in ein Vorgehensmodell (V-Modell) (vgl. MUTZ05). Das V-Modell (Abbildung 4) wird international als ein allgemeines Referenzmodell für den Entwicklungsprozess eingebetteter Systeme herangezogen und ist in [VM97] näher spezifiziert. Weiterentwicklungen des Modells sind das V-Modell XT für agile Prozesse und das W-Modell für Testprozesse, welche hier nicht näher betrachtet werden. Die Phasen werden dabei in allen Vorgehensmodellen iterativ durchlaufen und haben Beziehungen.

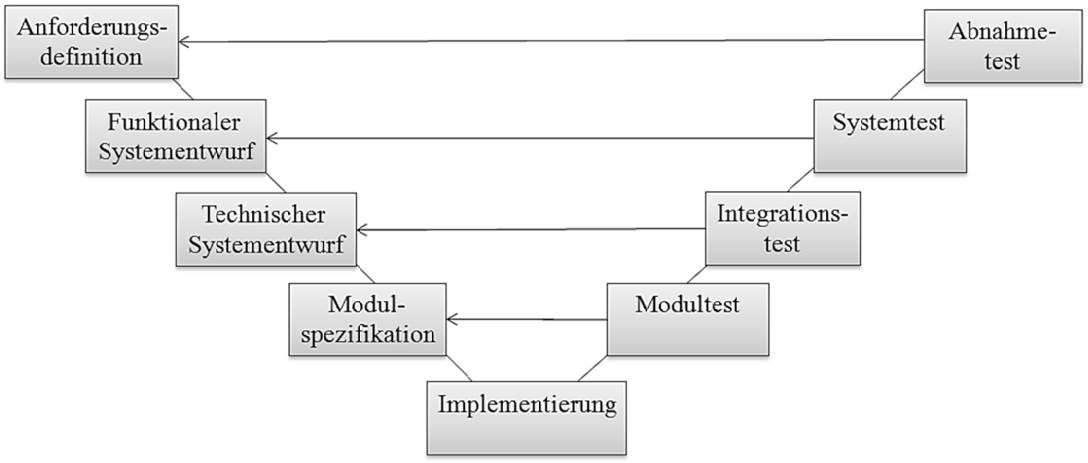


Abbildung 4: Das V-Modell definiert einen phasenorientierten Entwicklungsprozess

Der *Systementwurf* eingebetteter Systeme, welcher im besonderen Fokus dieser Arbeit liegt, gliedert sich in einzelne Teilprozesse des V-Modells, in den *funktionalen* und *technischen* Systementwurfsprozess. Er tangiert in der modellbasierten Entwicklung insbesondere auch die Anforderungsdefinition (Modelle als Anforderung) sowie die Modulspezifikation und ggf. sogar Implementierung bei angewandter Auto-Codegenerierung. Der rechte Ast wird

teils im W-Modell gespiegelt und erlaubt somit früh das modellbasierte Testen [ZAN08]. Aus dem V-Modell abgeleitet entspringen weitere, sehr viel detailliertere Prozessmodelle für die Systementwicklung, welche Zulieferer und mehrere Iterationsschritte zwischen Hersteller und Zulieferer wie auch zwischen den einzelnen Entwicklungsphasen mit berücksichtigen. So entstehen meist sehr unternehmensspezifische Entwicklungsprozesse, auf die in dieser Arbeit aufgrund der Vielfältigkeit nicht näher eingegangen wird. Eine gute Übersicht findet sich hierzu in [LIGG05; MUTZ05]. In den einzelnen Phasen des Entwicklungsprozesses kommen dabei unterschiedliche Computerprogramme, sogenannte *Werkzeuge* (englisch: Tools), zum Einsatz, mit denen Akteure pro Phase verschiedene *Artefakte* entwickeln.

2.2.2 Anforderungsdefinition

Die im V-Modell zu Beginn stehende *Anforderungsanalyse* (vgl. Kapitel 2.2.1) bezeichnet alle Aktivitäten zur Anforderungsdefinition und technischen Funktionsspezifikation eines eingebetteten Systems. Die Anforderungsanalyse beschäftigt sich dabei mit der Erhebung, Konsolidierung, Strukturierung und Verwaltung von Anforderungen an das zu erstellende System. Wie bereits beschrieben, bestehen eingebettete Systeme im Anwendungskontext der Automobilindustrie aus einer immer komplexer werdenden Architektur, zusammengesetzt aus Hardware und Software. Diese komplexe Architektur erfordert die Spezifikation von Anforderungen auf allen Abstraktionsebenen im Entwicklungsprozess. Die Art der zu erfassenden Anforderungen richtet sich hierbei nach der *Abstraktionsebene*, für die sie erstellt werden.

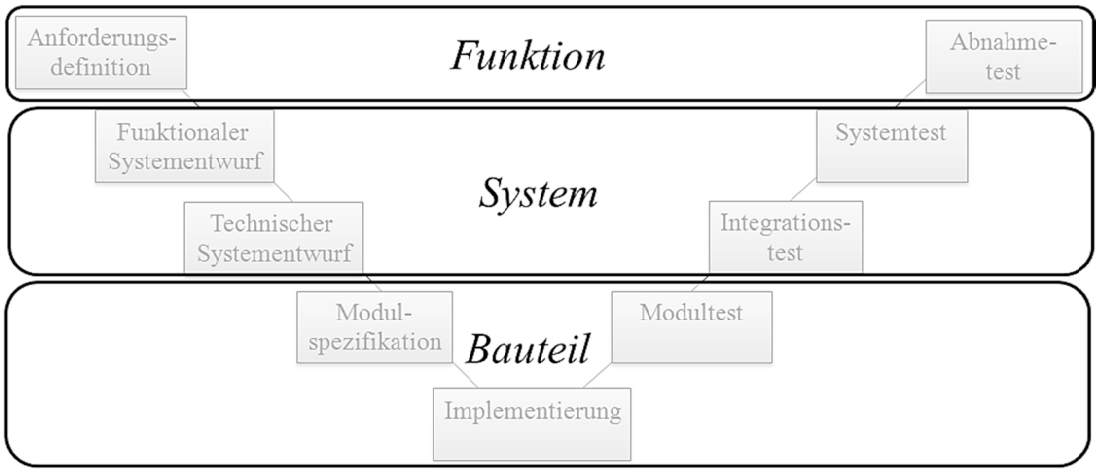


Abbildung 5: Abstraktionsebenen des V-Modells

Aus Sicht der Anforderungsanalyse erfolgt eine Aufteilung des V-Modells in die Abstraktionsebenen Funktionsebene, Systemebene und Bauteilebene.

- **Funktionsebene**: Auf der Funktionsebene werden Systemfunktionen spezifiziert, welche unabhängig von ihrer Umsetzung und jeglicher Systemarchitektur (hardwareunabhängig) eine logische Verhaltensbeschreibung aus Sicht des Systemanwenders darstellen. Solch abstrakte Systemfunktionen können in Unterfunktionen dekomponiert werden. Für Systemfunktionen werden keine Schnittstellen beschrieben. Der Schwerpunkt liegt allein auf Festlegung der umzusetzenden Funktionalität.

- **Systemebene**: Auf der Systemebene werden Basisfunktionen beschrieben. Dabei handelt es sich um technische Funktionen, die ohne weitere Teilung auf genau einem Bauteil realisiert werden, aber noch keinem Bauteil zugeordnet sind. Die Beschreibung ist unabhängig von einer konkreten technischen Architektur, also frei von technischen Terminologien oder einer konkreten Bauteilangabe. Allerdings besitzen Basisfunktionen eine *logische* Schnittstelle. Basisfunktionen dienen in den späteren Entwicklungsphasen dazu, um Systemfunktionen zu implementieren.
- **Bauteilebene**: Auf Bauteilebene erfolgt die Beschreibung von hardwarespezifischen Bauteilfunktionen, welche die konkreten Realisierungen von Basisfunktionen auf einem bestimmten Bauteil darstellen. Die *logische* Schnittstelle der zugehörigen Basisfunktion wird für eine Bauteilfunktion durch eine *technische* Schnittstelle realisiert. Diese Schnittstelle ist insofern architekturabhängig, als dass die beschriebenen Funktionen bauteilspezifische Datentypen besitzen und den Vorgaben für eine bestimmte Hardware entsprechen. Was durchaus möglich bleibt, ist die Wiederverwendung eines Bauteils und damit der auf diesem Bauteil realisierten Bauteilfunktion in verschiedenen technischen Systemen (Wiederverwendung).

Die in der Anforderungsanalyse entstehenden kollaborativen Artefakte sind natürlich sprachliche Dokumentationen (Spezifikationen in Textform) bzw. hierarchisch strukturierte Textfragmente in Tabellenform oder Textbausteine, welche in einem ‚Requirements Management System' (einer Datenbank) abgelegt bzw. gespeichert sind.

Diskussion:

Vorgehensweisen zur strukturierten Erfassung von Anforderungen existieren im wissenschaftlichen Umfeld, wie beispielsweise [KAPE06; HAGE06; CAFF04]. In der Praxis der eingebetteten Systementwicklung fehlt jedoch eine breite Akzeptanz. Vielmehr sind praxisorientierte und sehr unternehmensspezifische Vorgehensweisen vorzufinden. Eine solche Pragmatik wird in [SQLR08] zusammengetragen. Hier werden einige Prinzipien zusammengefasst, welche Anforderungen für eine hochwertige Anforderungsdefinition nach Tabelle 2 aufstellen und später in Richtlinien überführt werden können:

Tabelle 2: Prinzipien in der Anforderungsdefinition, nach [SQLR08]

Prinzipien für die Anforderungsdefinition:
− *Zuordnung von Anforderung zum Stakeholder (Akteur)* − *Klare und verständliche Formulierung* − *Prüfbarkeit zur Abnahme der Anforderung* − *Verständlichkeit durch eine einheitliche Terminologie* − *Identifikation (z. B. die eindeutige Nummerierung)* − *Konsistente Referenzen (auf andere Anforderungen, mitgeltende Dokumente)* − *Atomare Formulierung (Vermeidung von Verknüpfungen mit „und")* − *Positive Formulierung (Vermeidung der Wörter „nicht", „nie", „niemals")* − *Vermeidung von Füllwörtern, Weichmachern und Konjunktiven („eventuell", „soll", „sollte", „könnte", „wenn möglich" usw.)*

2.2.3 Modellbasierter Systementwurf

Oftmals ist ein systemumgebendes Problemszenario aus der Anforderungsanalyse zu komplex, um es gedanklich und mit allen Randbedingungen vollständig zu erfassen bzw. zu untersuchen. Untersuchungen mit allen Randbedingungen der realen Umgebung sind auch für computergestützte Verfahren aufgrund der Ressourcenknappheit mehrfach zu umfangreich und wären innerhalb der Systementwicklung sehr aufwendig. Aus diesem Grund ist das wesentliche Mittel auf Funktionsebene die Modellierung durch *Abstraktion*. Durch eine Reduktion auf die funktional wesentlichen und am besten fassbaren Parameter und Wechselwirkungen ist eine Systemfunktion einfacher zu beschreiben. Unter einem abstrakten Modell versteht man daher im Allgemeinen ein abstraktes *Abbild* eines Systems. Für Modelle soll im Allgemeinen folgendes Verständnis nach [BROY04] gelten:

Modell *(engl. model)* **(2.15)**

„Ein *Modell* ***M*** ist im Allgemeinen eine auf bestimmte Zwecke ausgerichtete, vereinfachende Abstraktion der Realität. Konkreter ist innerhalb der wissenschaftlichen Theoriebildung ein Modell das Resultat einer abstrahierenden und Relationen hervorhebenden Darstellung des behandelten Phänomens. Ein Modell entsteht, wenn Elemente aus dem Phänomen abstrahiert und zueinander in Beziehung gesetzt werden. Die Funktion des Modells besteht darin, aus den dargestellten Zusammenhängen Bedingungen und Prognosen bezüglich des Phänomens (oder Problems) ableiten zu können."

Man unterscheidet qualitative und quantitative Modelle, bei denen die Systemgrößen und ihre Wechselwirkungen beschrieben sind. Qualitative Modelle sind gekennzeichnet durch die verbale Beschreibung von Systemgrößen und ihre Wechselwirkungen. Quantitative Modelle hingegen sind charakterisiert durch eindeutige mathematische Formeln, Größen und Beziehungen. Funktionsmodelle (Artefakte) wiederum versuchen, das Verhalten eines Systems abstrakt zu beschreiben, um in einer Simulation die *Parametrierung* (Auslegung) zu bestimmen. Aufgrund anwachsender Komplexität im Entwurfsprozess eingebetteter Systeme wird das allgemeine Funktionsverständnis eines eingebetteten Systems, oder speziell seines Systemverhaltens, zunehmend anhand eines grafischen, teils simulierbaren Modells (Funktionsmodells) samt aller enthaltenen Systemkomponenten (Abbildung 6) erworben (vgl. [CON04; ZAN08]).

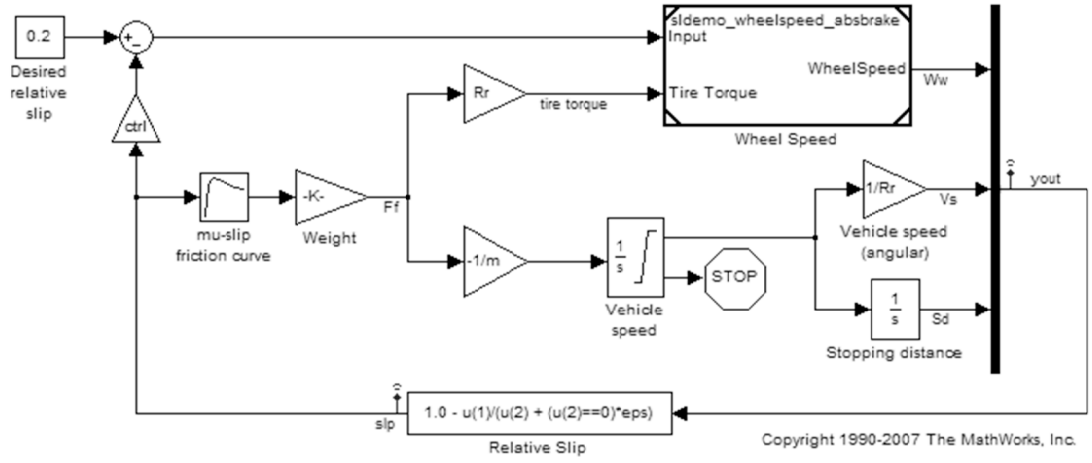


Abbildung 6: Funktionsmodell eines Anti-Blockier-Systems, aus [MLSL09]

Herkömmliche Anforderungsdefinitionen oder Softwarespezifikationen eingebetteter Systeme im Freitextformat werden daher zunehmend durch anwendungsspezifische, modellbasierte Entwicklungsmethoden für Software mit speziellen grafischen Notationen, wie Blockdiagrammen und Zustandsautomaten, ersetzt. Hier existiert demnach eine logische Artefakt-Beziehung von Anforderungsdefinition zum modellbasierten Systementwurf.

Die Notation ist durch verschiedene Modellierungssprachen bzw. Sprachstandards bestimmt, wie z. B. der *Unified Modeling Language* (UML) [UML22], oder herstellerspezifischen Modellnotationen wie der *ASCET-MD®* (ASCET) bzw. *MATLAB/Simulink/Stateflow®* (ML/SL/SF)-Modellierung [ASCT09; MLSL09]. Speziell im Bereich der Architektur- und Datenmodellierung eingebetteter Systeme wurden in diversen Forschungsprojekten UML-Derivate ausgebildet. Die bekanntesten Modellnotationen sind die *Systems Modeling Language* (SysML) [SYML11], die *Electronics Architecture and Software Technology Architecture Description Language* (EAST-ADL) [EAST08] als Vorläufer von AUTOSAR sowie die *Modeling & Analysis of Real-time and Embedded Systems* (MARTE) [MRTE08].

Der modellbasierte Systementwurf stellt Modelle in den Mittelpunkt der Systementwicklung. Modelle existieren dabei auch auf verschiedenen Abstraktionsebenen und beinhalten unterschiedlich abstrakte Informationen über das System, welches letztendlich entstehen soll. Diese Informationen reichen von Anforderungen, Spezifikationen, Programmcode bis hin zu Testfällen. Nachfolgend werden Modelle nach [KLEIN07] klassifiziert:

- **Konzeptionell**: Das Modell beschreibt eine mathematisch-physikalische Abbildung des Modellierungsgegenstandes. Es dient der regelungstechnischen bzw. der mathematischen Analyse zur Lösungsfindung. Es kann ebenfalls zur Ermittlung und zur Definition der wesentlichen Zustände einer abzubildenden Funktionalität sowie der möglichen Zustandsübergänge dienen. Im Allgemeinen sind physikalisch-konzeptionelle Modelle zeit- und wertekontinuierlich. Im Entwicklungsprozess werden physikalisch-konzeptionelle Modelle üblicherweise in der Systemanalyse und dem Systemdesign bzw. für den Systemtest als Strecke eingesetzt.
- **Verhaltensorientiert**: Verhaltensmodelle erfassen die vollständige Abbildung des Verhaltens einer Systemfunktionalität im algorithmischen Sinne. Dabei können ebenfalls Fehlererkennung bzw. Reaktionen auf Fehler modelliert werden, wie z. B. nicht plausible Eingangsgrößen oder das verspätete Empfangen von Signalen. Bei der Modellierung wird die spätere Umsetzung auf eine digitale Laufzeitumgebung durch zeitdiskrete Simulation berücksichtigt, wodurch Auswirkungen der Abtastzeit auf das funktionale Verhalten untersucht werden können. Im Allgemeinen wird eine ideale Umwelt in Bezug auf Rechnerressourcen (Prozessorzeit, Speicherbedarf) angenommen. Im Entwicklungsprozess wird das Verhaltensmodell vorwiegend in der Systemanalyse bzw. im Systemdesign eingesetzt und dient der Verifikation und Validation der System- bzw. Funktionsanforderungen. Hierzu werden sowohl Host-Simulationen auf dem PC als auch spezielle Rapid-Prototyping-Systeme eingesetzt. Ebenso können Verhaltensmodelle für die Testphase in Form von Strecken- und Testmodellen verwendet werden.
- **Implementierungsorientiert**: Ein implementierungsorientiertes Funktionsmodell beschreibt eine Systemfunktion mit dem Ziel der Ableitung einer ablauffähigen Software (Programmcode). Die Modellierung konzentriert sich dabei auf eine möglichst effektive Abbildung des Algorithmus mit Bezug auf den Ressourcenverbrauch in dem eingebetteten System. Typischerweise werden daher Datentypen verwendet, die ausgerichtet auf eine konkrete Ziel-Hardware (Plattform

und Betriebssystem) sind. Das Implementierungsmodell ist die Grundlage für die automatische Generierung eingebetteter Software. Im Entwicklungsprozess werden Implementierungsmodelle üblicherweise in den Phasen Modulspezifikation und Implementierung eingesetzt, wobei Letztere durch die automatische Codegenerierung größtenteils beeinflusst wird.

Der funktionale Systementwurf und der technische Systementwurf sind im Bereich der modellbasierten Entwicklung eingebetteter Systeme stark geprägt durch die funktionale Beschreibung eines *regelungstechnischen Systemverhaltens* [KLEIN07]. Die Grundkonzepte der Regelungstechnik stellen die Grundlage für die modellbasierte Entwicklungstechnik (*Modellbildung*) und für das Verständnis von *Funktions- und Verhaltensmodellen* dar. Da die umfassende Betrachtung geregelter Systeme aus der Regelungstechnik stammend nicht Gegenstand der Arbeit ist, sei hier auf detailliertere Einführung in [LUNZ08] verwiesen.

Im Prozess ist der Akteur (Funktionsentwickler) typischerweise konzentriert auf die Umsetzung der Systemfunktionen des eingebetteten Systems (*Funktionsdesign*), nicht jedoch auf den gesamten Funktionskontext. Dies birgt typischerweise Risiken, dass globale Richtlinien im Modell nicht eingehalten werden. Der gesamte Funktionskontext kann jedoch durch weitere Modelle vorgegeben sein, wie beim Modell des umgebenden Systems (*Streckenmodell*) und dessen Systemumgebung (*Umgebungsmodell*) [CON04]. Die Funktionalität und die physikalischen Zusammenhänge mit der Umgebung können hochkomplexe Funktionen ergeben, wie z. B. bei der Modellierung einer Motorensteuerung aus [BAU03]. Solche Modelle können mit einem hohen Detaillierungsgrad abgebildet und ihr Verhalten realitätsnah in Simulationsumgebungen simuliert werden. Hieraus wird deutlich, dass die Konformitätsprüfung einen umfassenden Betrachtungsraum besitzen muss, d. h. mehr als ein Modell zu untersuchen ist. Während also mehrere Artefakte, das Streckenmodell und das Umgebungsmodell, im weiteren Entwicklungsverlauf schrittweise durch das reale System und dessen reale Umgebung ersetzt werden, dient das Funktionsmodell als Basis für die Implementierung der eingebetteten Software auf dem eingebetteten System durch Codegenerierung. Ein Ergebnis des Funktionsdesigns ist typischerweise ein weiteres Artefakt: die generierte Software aus dem *Implementierungsmodell*. Implementierungsmodelle sind weniger abstrakt und berücksichtigen die Hardware-Plattform, auf der später der ‚Embedded Code' ausgeführt werden soll. Nachdem im Implementierungsmodell und im Codegenerierungs-Werkzeug alle Eigenschaften der Zielplattform erfasst wurden, kann Code für das gewünschte System generiert werden. Der Schritt vom Implementierungsmodell zum Embedded-Code ist heutzutage im modellbasierten Entwicklungsprozess bereits semiautomatisiert und je nach Anwendungsfall, in ideellen Fällen, auch voll automatisiert vorzufinden. Die Konformität zu spezifischen Codierungsnormen (wie z. B. MISRA) wird nur durch den Code-Generator selbst suggeriert und muss nachträglich durch einen Code-Checker geprüft werden. Für die Funktionsentwicklung ist entscheidend, welche Software auf welchem System ausgeführt wird.

Mit dem Ziel der Wiederverwertung einmal modellierter Funktionalität setzt das modellbasierte Funktionsdesign auf ein vergleichbares Konzept der Anforderungsanalyse auf. Basisfunktionsbeschreibungen aus der Anforderungsanalyse werden zunächst in sogenannten *Verhaltensmodellen* als Basismodelle (Basisfunktionsblöcke) modelliert. Diese Modelle nutzen die *logischen* Schnittstellenbeschreibungen und realisieren, unabhängig von einer möglichen späteren Systemarchitektur, das funktionale Verhalten der beschriebenen Basisfunktionen. Der Schritt von den Basisfunktionen zu den Bauteilfunktionen wird im Funktionsdesign mit dem Schritt zum Implementierungsmodell nachempfunden. Für die

übergreifende Konformitätsabsicherung müssen demnach die Basisfunktionsbeschreibungen und die logischen Schnittstellen konform zu den modellierten Basisfunktionsblöcken sein. Schließlich haben Fehler im Implementierungsmodell direkte Auswirkungen auf das generierte Artefakt (Embedded Code). Da Anforderungen einer kontinuierlichen Weiterentwicklung unterliegen, findet in der Praxis auch eine evolutionäre Weiterentwicklung der Funktionsmodelle von einem frühen logischen Modell hin zu einem Implementierungsmodell (*Modellevolution*) statt (vgl. [CON04]). Im Vergleich der Software-Entwicklung eingebetteter Systeme zur traditionellen Software-Entwicklung mit einer klaren Phasentrennung ist bei der modellbasierten Entwicklung ein stärkeres Zusammenspiel der Phasen Spezifikation, Design und Implementierung zu verzeichnen [ADL07]. In der Praxis wird häufig das Verhaltensmodell als ‚ausführbare Spezifikation', zusammen mit den textuellen Anforderungen, an den Zulieferer zur *Implementierung* des eingebetteten Systems weitergegeben. Dieser entwickelt dann ein spezifisches Implementierungsmodell oder alternativ auch einen handgeschriebenen Code, der dann auf einem eingebetteten System ausgeführt werden kann. Positive Effekte sind ein Effizienzgewinn durch die automatische Codegenerierung, wenn Modelle bereits vorhanden sind, sowie die frühe Festlegung von Testfällen im modellbasierten Test [ZAN08].

Diskussion:

Die Funktionsmodellierung, unabhängig von dem verwendeten Modellierungsstandard, bringen die genannten Vorteile aber auch Risiken mit sich. Die Nichteinhaltung von Richtlinienkonformität bei der Modellierung kann später zu ineffizienter bis fehlerhafter Software im eingebetteten System führen, so auch [CON06]. Werden Modelle als Spezifikation im kollaborativen Prozess verwendet, sind diese nicht immer eindeutig und widerspruchsfrei, da die Entwicklungs- und Modellierungswerkzeuge einen hohen Kreativitätsgrad erlauben. Zudem gibt es Abhängigkeiten zu anderen Tools in kollaborativen Prozessen (Anforderungsdokumentation, Architektur-Designs sowie Code-Compilern), die für den Modellierer nicht unbedingt offensichtlich bzw. einsehbar sind. Darüber hinaus können Spezifikationen auf dem Computer fehlerhaft implementiert und im Fahrzeug erst spät – wenn überhaupt – entdeckt werden. Aufgrund der genannten Risiken haben sich Entwicklungsrichtlinien (hier speziell: Modellierungsrichtlinien) allgemein durchgesetzt. Sie sind jedoch nicht für alle Modellierungsnotationen verfügbar. Die Entwicklungsrichtlinien können den gesamten Entwicklungsprozess mit Projektablauf, Modell-Design, Qualitätsmanagement, Release-Management, Change-Management und Programmierung übergreifend beschreiben oder aber auch hoch spezialisiert für eine bestimmte Modellierungssprache sein (Vertiefung in Kapitel 4.2). Globale Entwicklungsrichtlinien für kollaborative Prozesse beschreiben die geforderte Einhaltung und Verwendung von betriebsüblichen Vorlagen und unternehmensinternen Konventionen sowie von technischen Standards und Normenkatalogen, welche auch auf die Funktionsentwicklung wirken.

Nach [KLEIN07] werden für die modellbasierte Funktionsentwicklung folgende Qualitätskriterien (Anforderungen) identifiziert, aus denen Richtlinien abgeleitet werden:

- *Änderbarkeit, Wartbarkeit,*
- *Wiederverwendbarkeit,*
- *Verständlichkeit, Lesbarkeit,*
- *Testbarkeit,*
- *Nachvollziehbarkeit (Traceability) und Konsistenz.*

Hinsichtlich dieser Kriterien adressieren Prinzipien, aus denen Richtlinien abgeleitet werden können, die folgenden Aspekte (Tabelle 3) in der modellbasierten Funktionsentwicklung:

Tabelle 3: Prinzipien für den modellbasierten Systementwurf, nach [KLEIN07]

Prinzipien für den modellbasierten Systementwurf:
– *Einheitliches Layout/Gestaltung* – *Sprachumfang und -gebrauch* – *Modellarchitektur* – *Parametrisierung* – *Namensgebung* – *Dokumentation*

2.2.4 Modulspezifikation nach Systementwurf

In der modellbasierten Entwicklungskette kann nach dem Systementwurf die Phase der Modulspezifikation in sehr unterschiedlicher Art und Weise durch ein *Systemmodell* erfolgen. Systemmodelle, die rein einer Dokumentation dienen sollen, unterscheiden sich grundlegend von Systemmodellen, die zur Spezifizierung von Programmcode herangezogen werden. Möchte man schließlich eine grafische Repräsentanz einer vorhandenen Struktur aufzeigen oder aus diesen Modellen strukturierten Code automatisiert generieren, werden sich auch hier die Modelle in ihrer Zweckmäßigkeit logisch unterscheiden. In kollaborativen Prozessen ist der Personenkreis zu bedenken, dem diese Modelle in weiteren Entwicklungsschritten gezeigt oder zur weiteren Verarbeitung gegeben werden sollen, da die Verständlichkeit für verschiedene Domänenexperten nicht evident ist.

Tabelle 4: Vergleich modellbasierte und manuelle Software-Entwicklung

Vergleich	**Modellbasierte Entwicklung**	**Manuelle Software-Entwicklung**
Spezifikation	– als Modell, ausführbar	– als Text, nicht ausführbar
Richtlinien	– Modellierungsregeln, z. B. MAAB	– Programmierregeln, z. B. MISRA
Optimierung	– werkzeuggestützt, mit Simulation	– manuell, durch Erfahrung
Codierung	– automatisch, mit Sprachauswahl	– manuell, festgelegt ab Beginn
Wiederverwendung	– verifizierte Modell-Bibliotheken	– verifizierte Code-Bibliotheken
Qualitätssicherung	– Verifikation durch Modell-Checker	– Verifikation durch Code-Checker

Hierbei ist zu beachten, dass sich von den Softwareentwicklern angewöhnte Denkweisen der manuellen Entwicklung und Sprachkonventionen in den Modellen widerspiegeln, die später missverständlich auf Akteure (wie z. B. einen Funktionsentwickler) wirken, welche nicht programmcodeorientierte Denkweisen anwenden. Die Tabelle 4 zeigt einige Unterschiede.

Modelle sind meist durch ihren Zweck bestimmt. Sollte der Zweck eines Strukturmodells, Beschreibungsmodells, Spezifikationsmodells, Datenmodells oder Implementierungsmodells zu Beginn der Modulspezifikation nicht eindeutig ersichtlich sein, eignet sich eine Analyse des zu modellierenden Wissensgebietes mittels Interviews mit den Domänenexperten. Sollte als Ergebnis mehr als ein Zweck beabsichtigt werden, so ist es prinzipiell durch geeignete Abstraktion und Hierarchisierung des Informationsmodells möglich, diese unterschiedlichen Wünsche zu implementieren.

Signalorientierte Modellierung wurde bereits im vorangegangen Kapitel 2.2.3 vorgestellt. Im Gegensatz zur signalflussorientierten Modellierung mittels Blockdiagrammen finden in der Modulspezifikation objektorientierte Modellierungssprachen mittels Klassendiagrammen Verwendung. Während die signalflussorientierte Modellierung für das Funktionsdesign erstmals die geeignete Anwendung im industriellen Kontext findet, sind objektorientierte Modellierungssprachen während der Modulspezifikation eher üblich. In diesen wird der Softwareentwurf, also das Architektur-, das Komponenten- oder das Modul-Design für die Softwarearchitektur eines eingebetteten Systems modelliert.

Diskussion:

In der Prozessphase der Modulspezifikation kann im Softwareentwurf die Informations-modellierung durch *Metamodellierung* konform dem objektorientierten Paradigma nach der *Model Driven Architecture* (MDA) stattfinden [BORN05; PETRA06]. Ein Softwaremodell befindet sich in logischer Abhängigkeit zu einem Metamodell und muss konform zu diesem sein. In Kapitel 2.4 zur Abstraktionstechnik wird daher der Kerngedanke der MDA kurz eingeführt. Dieser bildet die Grundlage (bzw. die Spezifikation) zum Konformitätsnachweis für weitere Entwicklungsschritte, z. B. für die Implementierung in der nachfolgenden Prozessphase des V-Modells. In der Software-Entwicklung spielt der Softwareentwurf eine zentrale Rolle, so auch [LIGG02; OEST04].

Nach [REISS02] beeinflusst der Softwareentwurf die nachfolgenden Prozessphasen der Implementierung sowie der Wartung nachhaltig. Implementierung und Wartung sollen zusammengenommen etwa zwei Drittel des Gesamtaufwands ausmachen. In [REISS02] werden Richtlinien durch eine Bewertung des objektorientierten Entwurfs gegeben. Zusammengefasst sind die genannten Prinzipien in Tabelle 5 aufgeführt:

Tabelle 5: Prinzipien für den Softwareentwurf, nach [REISS02]

Prinzipien für den Softwareentwurf (Modulspezifikation):
− *Eindeutigkeit und Widerspruchsfreiheit* − *Abdeckung aller Stakeholder-Sichten des Systems* − *Vollständigkeit in Hinsicht auf die Aspekte der einzelnen Sichten* − *Vollständig quantifizierte Entwürfe (d. h., alle Aspekte sind messbar)* − *Entwurf unterstützt Bewertungsverfahren (Metriken)* − *Anpassbarkeit (Ausblendung von Aspekten; Bildung von Bewertungsschwerpunkten)*

2.2.5 Implementierungsphase nach Systementwurf

In der zum Systementwurf zugehörigen Implementierungsphase des V-Modells kombinieren Modellierungswerkzeuge objektbasierte Abstraktion und Codegenerierung miteinander. Hier existiert demnach eine logische Artefakt-Beziehung von Implementierungsmodell zum generierten Code. Durch die Wahl einer Softwareplattform, also eines eingebetteten Betriebssystems, kann plattformspezifischer Code erzeugt werden. Die Konformität zu Standards des Betriebssystems ist hierbei ein essenzieller Aspekt für die korrekte Ausführung des Codes zur Laufzeit. Ein wesentlicher Bestandteil im Systementwurf eines eingebetteten Systems ist demnach die Konformität zum zugrunde liegenden Echtzeit-Betriebssystem, das die Ausführung der verschiedenen Algorithmen und Berechnungen steuert. Populär verwendete Echtzeitbetriebssysteme im Anwendungsumfeld der Automobilindustrie sind die a) OSEK-konformen Echtzeitbetriebssysteme [OSEK09] bzw. die weiterentwickelten Betriebssysteme nach b) dem AUTOSAR-Standard [ASR09].

2.2.5.1 Konformität zu OSEK

Der Betriebssystem Standard OSEK wurde im Jahr 2005 als internationaler Standard in ISO 17356-3 veröffentlicht. Die OSEK-Spezifikation legt die Funktionsweise eines eingebetteten Betriebssystems in mehreren Substandards fest (vgl. auch [SCHN06]), wobei der Standard OSEK-OS in einer zugrunde liegenden Spezifikation (versionsabhängig) die Kernkomponente bildet. Typischerweise läuft das eingebettete Betriebssystem auf einer Ein-Prozessor-Architektur. Das OSEK-OS unterstützt die Echtzeitfähigkeit der eingebetteten Software: Anfragen von einzelnen Tasks oder auch vom Betriebssystem selbst werden zur Laufzeit in einem vordefinierten Zeitfenster beantwortet. Da OSEK die Ausführung mehrerer Tasks gleichzeitig erlaubt, die sich im Zeitfenster den Prozessor teilen, handelt es sich um ein präemptives Multitasking-Betriebssystem. Da die Ressourcenknappheit von eingebetteten Systemen typisch ist, wurden in OSEK-OS keine dynamischen Aspekte, wie eine dynamische Speicherallokation oder die Einbindung von Treibern, umgesetzt − da diese die Hardwareanforderungen maßgeblich erhöhen würden. Somit ist OSEK als statisches Betriebssystem konzipiert, sodass alle Betriebsmittel wie Speicher, Treiber oder Ressourcen samt Verbrauch durch die einzelnen Tasks bereits vor der Systemgenerierung bekannt und OSEK-konform festgelegt sein müssen. Die Anwendung wird zusammen mit dem Betriebssystem kompiliert und gelinkt. Implementierungen sind meist in der Programmiersprache C umgesetzt, sodass eine Betriebssystem-Implementierung in Form einer C-Bibliothek geliefert wird, die dann zu den eigens entwickelten Funktionen zugezogen werden kann.

2.2.5.2 Konformität zu AUTOSAR

Aufgrund der wachsenden Inhomogenität der umgesetzten OSEK-Lösungen am Markt waren Systementwickler im Laufe der Zeit zunehmend genötigt, den immer größer werdenden Teil der Entwicklungskapazitäten für die Integration von Einzelkomponenten aufzuwenden (vgl. [SCHN06]). Diese Kapazitäten standen der Entwicklung neuer Funktionen dementsprechend nicht mehr zur Verfügung. Zudem war ein weiterer Nachteil von OSEK-basierten Entwicklungen die Reduktion der Wiederverwendbarkeit von Software, da die zahlreichen Hardwarekomponenten zu verschieden waren, um Software mit nur leichten Anpassungen auf andere Hardwareplattformen oder Rechnerarchitekturen zu portieren. Dies waren Motivationspunkte zur Weiterentwicklung des OSEK-Standards in einem neuen Konsortium. Das AUTOSAR (AUTomotive Open System ARchitecture) Konsortium [ASR09] bildet einen internationalen Verbund von Herstellern und Zulieferern mit dem Ziel, einen offenen Standard für Software-Architekturen zu etablieren. Die Hauptidee ist dabei die klare Trennung der hardwareunabhängigen von den abhängigen

Komponenten der Steuergerätesoftware (vgl. Kapitel 3.3 zur Standardisierung). Hierzu wird eine Laufzeitumgebung als Abstraktionsschicht zwischen der Mikrocontroller-Hardware und den unabhängigen Anwendungen eingeführt. Die AUTOSAR-Software-Architektur lässt sich in drei Schichten unterteilen: die AUTOSAR-Software-Komponenten (SWC), die AUTOSAR Laufzeitumgebung (Runtime Environment, RTE) und die Basis-Software. Die Basis-Software enthält Module wie die Betriebssystem-Services sowie die Kommunikationsmodule. Die RTE abstrahiert die SWC-Schicht von allen Implementierungsdetails der Basis-Software und ist verantwortlich für die SWC-Kommunikation. Die Laufzeitumgebung RTE bildet die Basis für die oberste Abstraktionsebene, der AUTOSAR Application Software Components (Anwendungs-software). Unterhalb der RTE definiert der AUTOSAR-Standard mehrere Softwareebenen, welche die hardwarespezifischen bzw. steuergerätespezifischen Softwareanteile umsetzen, wie spezielle Kommunikationstreiber oder Betriebssystemfunktionen. Die RTE definiert zudem Schnittstellen für die Anwendungen zur Umgebung. Es werden für die Kommunikation die Modelle Client/Server und Sender/Receiver zur Verfügung gestellt. Sämtliche Interaktionen von Anwendungen erfolgen ausschließlich über die RTE. Der Vorteil in Gegenüberstellung zum OSEK-Standard ist eine Unabhängigkeit der Anwendung von der Steuergerätehardware. Des Weiteren sollen Applikationen von einem Steuergerät zu einem anderen verschoben werden können, um eine bestmögliche Nutzenoptimierung der Ressourcen zu erreichen.

2.2.5.3 Konformität von Modellen zu generiertem Code

Modellierungswerkzeuge berücksichtigen heutzutage die Einbindung sowohl von OSEK als auch von AUTOSAR Komponenten sowie die Festlegung standardisierter Softwaremodule bezüglich Variablen, Wertebereiche, Parameter, Konstanten, Speicherstrukturen und Datendarstellungen (vgl. [PINN09]). Durch die Verwendung von *Botschaften* (engl. Messages) kann bei der Festlegung von Echtzeit-Tasks das Zusammenwirken von Prozessen beschrieben werden. Mit diesem Verfahren kann die Datenkonformität während der Laufzeit gesichert werden. In den Modellierungswerkzeugen (Authoring Tools) werden oftmals die die Funktionslogik, die Ablaufplanung (Scheduling) in Echtzeit, die Parameterwerte und die gezielte Implementierung auf einem eingebetteten System voneinander separiert. Implementierungsabhängige Einstellungen können für die Bit-Auflösung, die Grenzwerte, die Umrechnungsformeln, die Speicherorte und Namensgebung für die Datenelemente vom Systementwickler eingestellt werden. Durch die Trennung von Modellelementen kann so die Anzahl der Modellvarianten während des Gesamtlebenszyklus einer Softwarefunktion reduziert und Basiskomponenten können wiederverwendet werden. Aus dem Modell wird in der Phase der Implementierung automatisiert der C-Code für das eingebettete System (*Mikrocontroller-Target*) spezifisch generiert. Des Weiteren wird basierend auf der Modellspezifikation meist eine vollständige Dokumentation der Embedded-Software mit generiert, welche mit dem generierten Code übereinstimmen sollte. Vor der Kompilierung muss der C-Code teilweise auch aus anderen Quellen (z. B. spezifische Mikrocontroller-Bibliotheken oder Treiber) integrieret werden, um die komplette Embedded-Software für das eingebettete System zu erstellen.

Diskussion:

Die Konformität zu AUTOSAR ist nicht nur auf Implementierungsebene gefordert (vgl. auch [WANG08]). Vom Konsortium veröffentlichte Dokumentationen wie ‚*Applying Simulink to AUTOSAR*' oder ‚*Applying ASCET to AUTOSAR*' beschreiben die AUTOSAR-gerechte Modellierung in ASCET oder in ML/SL/SF [APSA06; APPA06]. Modellierungs-

richtlinien helfen hierbei sicherzustellen, dass sich in der Regler-Entwurfsphase (während des Control-Designs) erstellte Modelle auch für die Implementierung in Form von standardkonformen C-Code eignen (vgl. auch [EISE06]). Modelle und Code befinden sich somit in Abhängigkeit, da Modelleigenschaften einen direkten Einfluss auf generierten Code und dessen Effizienz besitzen. Demnach ist es genauso wie bei Codierungsrichtlinien zur manuellen Programmierung sinnvoll, entsprechende Regeln für einen modellbasierten Entwicklungsprozess aufzustellen. In [EISE06] werden Modellierungsprinzipien für die automatische Codegenerierung mit TargetLink [TGTL09] (Tabelle 6) vorgestellt:

Tabelle 6: Prinzipien für die automatische Codegenerierung, nach [EISE06]

Prinzipien für die automatische Codegenerierung (Implementierungsphase):
− *Namenskonventionen für Blöcke und Signale* − *Geeignete Farbwahl und Blockanordnung (zur Vereinfachung von Reviews)* − *Konsistenter Datenfluss (üblicherweise von links nach rechts)* − *Hierarchische Modellstruktur, welche die Funktionalität und die Funktionspartitionierung des Code-Generators reflektiert* − *Aussagekräftige Blockkommentare, die vom Code-Generator auch in den Code übernommen werden*

Nach [EISE06; KLEIN07] wird durch Modellierungsrichtlinien die Durchgängigkeit des Entwicklungsprozesses sowie die Steigerung von Produktivität und Qualität erreicht. Zu den in den Richtlinien abgedeckten Aspekten gehören u. a. die Definition eines geeigneten Modell-Subsets, die Generierung von effizientem Code, die MISRA-Konformität sowie Fragen der Skalierung für Festkomma-Code. Durch die Anwendung der Richtlinien wird die Effizienz des modellbasierten Entwicklungsprozesses mit einem Code-Generator deutlich gesteigert, wobei diese durch den Einsatz von *‚Style Checkern‘* (regelbasierte Prüfwerkzeuge zur Sicherstellung eines konformen Modell-Layouts/-Designs) zur automatischen Überprüfung der Einhaltung der Richtlinien noch weiter erhöht werden kann [EISE06].

2.2.6 Testspezifikation im Systementwurf

Um das Funktionsdesign in den frühen Entwicklungsphasen des V-Modells abzusichern, werden anhand von Testspezifikationen während der modellbasierten Funktionsentwicklung systematische, automatisierte Tests durchgeführt und verwaltet (vgl. [CON04; ZAN08]). Eine Testfallbeschreibung gibt an, welches Testobjekt und welche seiner Funktionen geprüft wird, mit welchen Eingabedaten und somit Testdaten welche Ergebnisse erwartet werden. Vollständiges Testen ist in der Regel mit den gegebenen Ressourcen heutiger komplexer eingebetteter Systeme nahezu unmöglich. Die richtige Auswahl von Testfällen kann die Fehlerentdeckung effizient gestalten, erfordert aber viel Erfahrung und Intuition. Akteure im Prozess besitzen durch modellbasierte Tests heutzutage Möglichkeiten, um passende Testdaten automatisiert zu erhalten und diese auch in geeigneter Form zu präsentieren beziehungsweise die Tests in einen automatisch generierten Test-Bericht zu überführen. Die Werkzeugumgebungen für modellbasiertes Testen vereinigen verschiedene Phasen und bieten meist eine Testumgebung für den gesamten modellbasierten Testprozess. Sie ermöglicht die Speicherung, Verwaltung und Administration der größeren Anzahl von Tests, Testdaten und Testergebnissen. Bevor jedoch eine Testautomatisierung beginnen kann,

müssen Beschreibungsmittel für Testdaten angewendet werden, wie beispielsweise grafisch-tabellarische Editoren oder eine Möglichkeit zur Gewinnung bereits vorhandener Daten aus Simulationen − Testwerkzeuge unterstützen hierbei eine systematische Testfallermittlung. Ein mehrfach im Kontext der Entwicklung eingebetteter Systeme verwendetes Beschreibungsmittel ist die systematische Testfallermittlung mittels Klassifikations-baummethode (vgl. auch [KRUP05; KRUP06; ENGE07; NARD0]).

Die *Klassifikationsbaummethode* nach [GRO93] ist eine Methode der Funktionstests (Black-Box-Test), in denen die Eingangsgrößen und Kombinationen eines Testobjekts in relevante Aspekte unterteilt (klassifiziert) werden. Die Klassifikationsbaummethode startet mit der Analyse der funktionalen Spezifikation − dem Funktionsmodell aus der Prozess-phase des modellbasierten Systementwurfs. Hier existiert demnach eine logische Artefakt-Beziehung von Testfallbeschreibung zum Funktionsmodell. Die Zielstellung ist dabei, die testrelevanten Aspekte einer Funktion im Kontext zu erkennen und durch *Komposition* als Problem zu definieren. Anschließend werden die aus dem Modell abgeleiteten Klassen gezielt miteinander kombiniert, um daraus Testsequenzen zusammenzustellen (vgl. auch [KRUP05]). Die schrittweise Partitionierung der Eingangsdaten wird in einem Artefakt, strukturiert in Form eines hierarchischen Baums, dargestellt. Die Abbildung 7 zeigt einen solchen Klassifikationsbaum mit der Klassifikation *Eingang_1* und *Signalwert*.

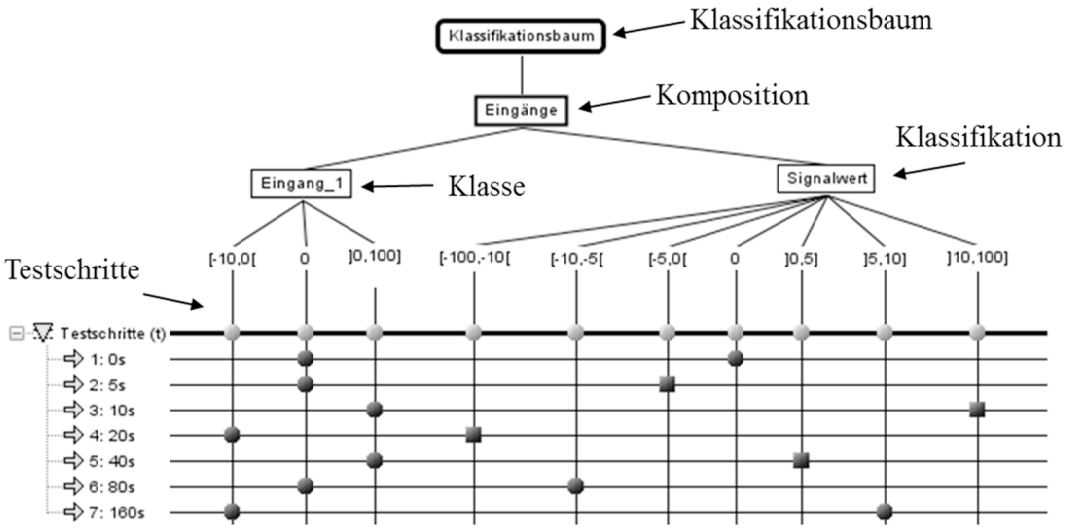


Abbildung 7: Klassifikationsbaum zur Testfallbeschreibung, nach [GRO93]

Die *Klassifikationen* werden dann weiter in einzelne *Klassen* unterteilt. Ziel ist es, die Klassen so zu wählen, dass sich jede einzelne Klasse in Bezug auf die Aufdeckung potenzieller Fehler konsistent verhält.

Das zu testende Artefakt, z. B. ein Funktionsmodell in ML/SL/SF, wird *Unit Under Test* (UUT) genannt. Die UUT verhält sich für alle Werte seiner Klasse entweder korrekt oder inkorrekt. Das bedeutet, dass repräsentative Sätze an Werten für das Testen der UUT (Uniformitätshypothese) eingesetzt werden können und nicht alle infrage kommenden Werte verwendet werden müssen. Am Anfang des Baumes steht dafür eine Kombinationstabelle, in der die *Testschritte* (Teststep Values) markiert sind. Testsequenzen werden durch Kombination unterschiedlicher Klassen erstellt. Die Referenzdaten und die Evaluierungsdaten können für den späteren Vergleich mit dem Ausgangsverhalten der UUT

definiert werden. Die Testszenarien können auf alle möglichen Artefakte angewendet werden: z. B. auf das Funktionsmodell, auf das Implementierungsmodell oder auf den Funktionscode. Um die Tests automatisiert durchzuführen, kann durch Einsatz einer Werkzeugumgebung, wie *MTest* [DSPA05] für ML/SL/SF, ein Testrahmen generiert werden, in den eine Kopie des Simulink- oder des TargetLink-Modells von der UUT eingefügt wird. Die Durchführung der Tests kann frei konfiguriert und für mögliche Wiederverwendung vor der Testdurchführung einzeln gespeichert werden. Die Testergebnisse werden für die Analyse in ein weiteres Artefakt (XML-Dokument) kompiliert. Das Ausgangsverhalten kann für den nächsten Test als Referenz verwendet werden. Abweichungen zwischen Referenzdaten und Ausgangsgrößen werden entdeckt und dokumentiert. Nach der Analyse können neue Tests beschrieben werden. Das Modell wird durch die wiederholten Testdurchläufe verfeinert.

In den späten Prozessphasen des V-Modells – nach Entwicklung und Implementierung des eingebetteten Systems auf der Ebene der Bauteile – erfolgt ein Modultest (vgl. Abbildung 4) mit dessen Hilfe die korrekte Umsetzung und Integration der für eine Komponente (Bauteilebene) festgelegten Funktionsanteile und sonstiger für die Komponente festgelegter Anforderungen erfolgt. Im Integrationstest wird anschließend die Integration mehrerer Bauteile beim so genannten *‚Breadboard-Test‘* durchgeführt. Das eingebettete System wird dabei in einer Laborumgebung (in einer sogenannten Breadboard-Anordnung) mit anderen Geräten verbunden, um das Verhalten im Umgebungskontext so funktionsnah wie möglich zu testen. Ziel der Untersuchungen ist es, die Wechselwirkung zwischen Funktionen verschiedener eingebetteter Systeme und der Kommunikation zu validieren bzw. festzustellen, ob sich die Systeme der Vorgabe entsprechend verhalten.

In dieser Phase werden im Automotive-Kontext nicht verfügbare bzw. nicht betreibbare Steuergeräte mittels Restbussimulation simuliert. Breadboard-Tests werden separat für Funktionen der Bus-Systeme wie z. B. Antriebs-CAN, Komfort-CAN und Infotainment-CAN durchgeführt (CAN, Controller Area Network). Teilweise wird dieser Prozess durch HiL-Echtzeitsimulatoren (HiL, Hardware-in-the-Loop) unterstützt. Bei der Hardware-in-the-Loop-Simulation werden reale elektronische Steuergeräte in eine Simulationsumgebung eingebunden. Fahrzeugkomponenten, wie z. B. Motor, Getriebe und Fahrwerk werden im Simulator als mathematische Modelle nachgebildet. Die Generierung der Sensorsignale sowie die Erfassung und Verarbeitung der Aktuator-Signale erfolgen dabei in echtem Zeit- und Werteverhalten. Der Test mittels HiL wird im Rahmen des Domänentests durchgeführt. HiL-Tests werden weitestgehend automatisiert durchgeführt, daher werden diese in dieser Arbeit nicht weiter betrachtet.

Schlussendlich wird auf der Systemtestebene der sogenannte Fahrzeugintensivtest (FIT) ausgeführt. Die Tests werden vor dem Übergang zur nächsten Projektphase durchgeführt und erfolgen an Fahrzeugen, die den letzten dokumentierten Bauzustand aufweisen. Ziel des FIT ist eine intensive Erprobung der kompletten elektrischen Funktionen am Fahrzeug durch Entwickler und unabhängige Testteams zur Ermittlung des aktuellen Entwicklungsprozesses aus Sicht der Elektrik/Elektronik. Typische Artefakte sind hierbei tabellarische Auswertungen, in denen die Ergebnisse aus dem FIT eingetragen werden.

Diskussion:

Durch die geschilderte Vorgehensweise in der modellbasierten Testfallspezifikation wird der Zusammenhang von Testspezifikation und Funktionsmodell bzw. dem generierten Code offensichtlich. Auch bzgl. der Anforderungen gibt es Abhängigkeiten. In [LIM08] wird die

systematische Ableitung von Testfällen aus den Anforderungen beschrieben. Es werden die kollaborativen Artefakte (Anforderung und Testfallbeschreibung) miteinander logisch verknüpft. Durch ein werkzeugtechnisch unterstütztes Verfahren werden zudem die Rückverfolgbarkeit (Tracing) und die Auswirkungsanalyse (Impact Analysis) geschildert. Zusätzlich ergeben sich Möglichkeiten zur Messung der Testabdeckung. In [CON6] wird vorgeschlagen, dass, wo immer möglich, statische Modellreviews automatisiert und mit dynamischen Testtechnologien verknüpft werden sollten, um die Qualität zu erhöhen.

Prinzipien zur Modellierung konformer Klassifikationsbäume sind bisher nicht ausreichend wissenschaftlich untersucht. Dies ist vermutlich darin begründet, dass sich ein Klassifikationsbaum logisch in Abhängigkeit zu der UUT befindet und konsistent zu dieser sein sollte (der Struktur, den Schnittstellen, den Parametern usw.). Aus Erfahrungsberichten und persönlichen Interviews im Kontext [LIM08] konnten jedoch folgende Prinzipien nach Tabelle 7 für Klassifikationsbäume aus Praxiserfahrungen zusammengetragen werden:

Tabelle 7: Prinzipien bei Testspezifikation, in Anlehnung an [LIM08]

Prinzipien für die Testspezifikation im Systementwurf:
− *Strukturierung der Kompositionen (z. B. Eingangs- und Ausgangssignale trennen)*
− *Einhaltung von Namenskonvention (hinsichtlich Testautomatisierung)*
− *Eindeutigkeit der Klassifikation (z. B. keine Namenswiederholungen)*
− *Übereinstimmung der Klassifikationen mit dem zu testenden Artefakt*
− *Korrekte Belegung der Testdaten mit gültigen Werten (z. B. nur Zahlenwerte)*
− *Sicherstellung fortlaufender Testschritte (z. B. konsistente Intervalle)*
− *Lückenlose Wertzuweisungen in der Kombinationstabelle*
− *Vermeidung unnötiger Wartezyklen für die Testausführung*
− *Übereinstimmung mit referenziertem Artefakt (Modell, Anforderung usw.)*

2.2.7 Prozessbewertungen durch Reifegradmodell

Ein Modell für die Bewertung von Entwicklungsprozessen bieten Reifegradmodelle. In Abgrenzung zu den ISO 9000er Regelwerken [ISO9000], welche das gesamte Unternehmen als Ganzes betrachten, sind Reifegradmodelle speziell für den Produktentwicklungsprozess entworfen worden. Nach [KNEU06; HOER06] sind das ‚*Capability Maturity Model Integration*' (CMMI) sowie das speziell für Software-Entwicklungsprozesse geltende ‚*Software Process Improvement and Capability Determination*' (SPiCE) industriell eingesetzte Standards zur Durchführung von Bewertungen von Entwicklungsprozessen, sogenannten Assessments. In [MUEL07] wird das domänenspezifische ‚*Automotive SPiCE*' Modell vorgestellt, welches ein ISO/IEC 15504-kompatibles, speziell auf die Automobilbranche zugeschnittenes Assessmentmodell ist. Herausgestellt sind hierbei Hilfestellungen zur Sicherung der Nachvollziehbarkeit (Traceability) sowie zur Erfüllung der Norm (ISO/IEC 61508) zur Funktionssicherheit elektronischer Produkte. Die Gemeinsamkeiten der Reifegradmodelle liegen in der Beurteilung und der Verbesserung der Qualität (Reife) von technischen Produktentwicklungsprozessen. Ein solches Prozessmodell wird immer dann angewandt, wenn die Stärken und die Schwächen einer Produktentwicklung objektiv

analysiert werden sollen. Aus der Bewertung sind Verbesserungsmaßnahmen zu bestimmen und diese sodann in die Prozesse zu integrieren. Nach [KNEU06] ist CMMI ein Mittel, um primär die Produktentwicklung nachhaltig zu verbessern. Sekundär verfolgt wird eine offizielle Überprüfung des Reifegrades (Appraisal) durch einen unabhängigen Gutachter (Assessor) mit anschließender Zertifizierung nach einer Reifegradstufe. Sechs aufeinanderfolgende, sich jeweils ergänzende Reifegradstufen werden definiert: (0) *Incomplete*, (1) *Performed*, (2) *Managed*, (3) *Defined*, (4) *Quantitatively Managed* und (5) *Optimizing*. Das primäre Ziel des Einsatzes von Reifegradmodellen in einem Unternehmen ist es, eine kontinuierliche Prozessverbesserung zu unterstützen, indem Anforderungen und Kriterien an den Produktentstehungsprozess definiert werden.

Diskussion:

Reifegradmodelle sind auf der Erfahrung basierende, sehr allgemein beschriebene Verfahrensweisen (Basispraktiken). Im Gegensatz zu einer konkreten Prozessbeschreibung (Handlungsvorschrift) werden strukturierte Vorschläge (Hilfestellungen) zur Erlangung einer effizienten und qualitativ hochwertigen Produktentwicklung erteilt. Die konkrete Definition des Entwicklungsprozesses ist jeweils spezifisch, d. h., es obliegt dem Unternehmen selbst. Reifegradmodelle verlangen die Aufstellung von Anforderungen an Prozesse und Arbeitsergebnisse, die stetige Durchführung von Reviews der Prozesse und von Arbeitsergebnissen sowie in den Stufen vier und fünf die Einführung von Messungen anhand von Messwerten (Metriken) zur kontinuierlichen Optimierung der Prozesse. Die Erfüllung der Anforderungen (Konformität) wird durch Audits bzw. Assessments bzgl. der Prozesse und Arbeitsergebnisse getätigt. Demnach kann eine regelbasierte Konformitätsprüfung an kollaborativen Artefakten ein Audit bzw. ein Assessment automatisiert unterstützen. In [KNEU06; HOER06; MUEL07] werden vielfältige Prinzipien für eine Ableitung von Richtlinien genannt, die aus Platzgründen nur auf die Entwicklung eingebetteter Systeme bezogen, grob zusammengefasst, in der Tabelle 8 aufgeführt sind.

Tabelle 8: Prinzipien der Reifegradmodelle, in Anlehnung an [KNEU06; HOER06]

Prinzipien aus Reifegradmodellen im Systementwurf:
Anforderungen: − *Dokumentationspflicht (Anfertigung von Methodenbeschreibung und Protokollen)* − *Datensammlung (über Reviews & Interviews durch Fragebögen & Umfragen)* − *Bewertungen (Ratings) der Prozesse und von deren Arbeitsergebnissen* − *Berichte über Prozessdurchführung und deren Arbeitsergebnissen* *Maßnahmen:* − *Einführung von Qualitätsaufzeichnungen (z. B. Prozess- und Artefakt-Prüfung)* − *Entwicklung von Richtlinien auf Basis von Kriterien (Testkriterien, Metriken, usw.)* − *Einführung von Verifikationsverfahren (z. B. für Softwaremodule durch Reviews)* − *Sicherstellung von Testbarkeit (Testplanung, Testfallspezifikation)* − *Sicherstellung von Konsistenz (besonders bei prozessübergreifenden Tätigkeiten)* − *Mittel zur Sicherung der Nachvollziehbarkeit (Traceability)*

2.3 Formalisierungstechniken

Zur Erfassung und zur konkreten Beschreibung natürlich sprachlicher Anforderungen und Schlussfolgerungen dienen Formalisierungstechniken. In der Informatik kennt man formale Logiken als ein Teil der formalen Sprachen, bei denen durch Symbolnotationen Schlüsse gezogen werden können. Damit solche Schlüsse möglich sind, müssen eine formale Syntax und eine Semantik, die Struktur und die Bedeutung der Sprache wohl definiert worden sein. Es werden daher für die Arbeit verwendete Beschreibungsmittel zur Formalisierung von natürlich sprachlichen Richtlinien durch die Mengentheorie, die Aussagenlogik sowie als dessen Erweiterung durch die Prädikatenlogik erster Ordnung nach [STAE05] vorgestellt. Da bei der statischen Analyse kollaborativer Artefakte ein zeitlicher Verlauf nicht relevant ist und somit als weitere Dimension nicht in die Betrachtung einbezogen wird, werden beispielsweise Automaten oder Temporallogiken in diesem Kontext nicht betrachtet.

2.3.1 Mengen

Anforderungen in Richtlinien beziehen sich auf eine Ansammlung von Elementen in kollaborativen Artefakten bzw. schließen wohldefinierte Elemente aus einer *logischen Zusammenfassung* aus. In der Mathematik kennt man nach (G. Cantor, 1895) die *‚Menge‘*:

Menge ***(engl. set)*** **(2.16)**
„Unter einer *Menge* ***M*** verstehen wir jede Zusammenfassung von bestimmten wohlunterschiedenen Elementen unserer Anschauung oder unseres Denkens (welche *Elemente* der Menge genannt werden) zu einem Ganzen.“

Dies entspricht auch der in der Informatik üblichen Ansichtsweise, wie der in [BACH92; PEPP95; PEPP00; WEST96]. Die elementare *Mengenlehre* wird als bekannt vorausgesetzt. Eine ausführliche Definition findet sich in [BRON01]. Mengen werden in dieser Arbeit mit griechischen Großbuchstaben bezeichnet. Für die Elementbezeichnung und Notation gilt:

$\boldsymbol{a} \in \boldsymbol{M}$	Das Element ***a*** ist in der Menge ***M*** enthalten.
$\boldsymbol{a} \notin \boldsymbol{M}$	Das Element ***a*** ist nicht in der Menge ***M*** enthalten.
$\varnothing$	Eine Menge ohne ein Element ist die *leere Menge*.
$\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$	*Aufzählung*: Eine Menge bestehend aus den indizierten Elementen.
$\boldsymbol{A} \subseteq \boldsymbol{B}$	*Teilmenge*: Die Menge ***A*** ist eine Teilmenge von ***B*** genau dann, wenn jedes Element von ***A*** auch zu ***B*** gehört.
$\boldsymbol{A} \cup \boldsymbol{B}$	*Vereinigung*: Menge aller Elemente, die in ***A*** *oder* in ***B*** vorkommen.
$\boldsymbol{A} \cap \boldsymbol{B}$	*Durchschnitt*: Menge aller Elemente, die in ***A*** als auch in ***B*** vorkommen.
$\boldsymbol{A} \setminus \boldsymbol{B}$	*Differenz*: Menge aller Elemente von ***A***, die nicht auch in ***B*** liegen.
$\{\boldsymbol{a} \in \boldsymbol{M}: \varphi(\boldsymbol{a})\}$	*Aussonderung*: Menge aller Elemente mit der Eigenschaft $\boldsymbol{\varphi}$.

Des Weiteren verwenden wir bekannte Definitionen der Mengenlehre, wie *geordnetes Paar*, *Tupel, Relation, Produkt* und *Funktion* auf den Elementen von Mengen, nach [BRON01].

2.3.2 Aussagenlogik

Die *Aussagenlogik* ist eine in der Informatik bekannte Form für die Formalisierung von Bedingungen. Sie wird in dieser Arbeit für die formale Beschreibung einer natürlich sprachlichen Anforderung aus einer Richtlinie verwendet und soll kurz eingeführt werden. In der klassischen Aussagenlogik geht man von einer Menge von Aussagenvariablen aus. Aus diesen werden mithilfe logischer Verknüpfungen (*Junktoren*) kompliziertere Aussagen (*Formeln*) aufgebaut und ausgewertet. Aussagenvariablen stehen für nicht weiter spezifizierte Aussagen, die wahr oder falsch sein können.

Beispiel:
Beispiele solcher Aussagen bzgl. Artefakteigenschaften sind „*Die Hintergrundfarbe ist weiß.*“ und „*Die Vordergrundfarbe ist nicht weiß.*“ bzw. bekannte Aussagen aus dem Bereich der Mathematik sind: „5 > 4“ oder „2 + 3 = 5“.

In diesem Kapitel werden atomare Aussagenvariablen mit Kleinbuchstaben (wie ***x***, **y**) und Formeln mit lateinischen Großbuchstaben (wie ***Q***, ***R***) bezeichnet. Die Aussagenlogik definiert Symbole als logische Verknüpfungen (Junktoren) mit folgenden Bedeutungen:

- ¬ *Negation* („nicht“)
- ∧ *Konjunktion* („und“)
- ∨ *Disjunktion* („oder“)
- ⇒ *Implikation* („dann“)
- ⇐ *Bedingte Implikation* („wenn“)
- ⇔ *Äquivalenz* („genau dann, wenn“)

Neben Junktoren existieren logische Symbole für die Wahrheit und den Widerspruch:

- ⊤ *Wahrheit* („wahr“)
- ⊥ *Widerspruch* („falsch“)

Die *Interpretation* der Formeln ergibt für wahre Aussagen die Erfüllung einer Anforderung (Konformität) und für falsche Aussagen einen Fehler oder Mangel (die Nonkonformität).

Beispiel: Ein logischer Vergleich zweier Variablen ***x*** und ***y*** mit unterschiedlicher Wertebelegung und der Auswertung (Ergebnis) des logischen Ausdrucks bzgl. Konformität.

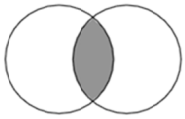
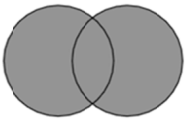
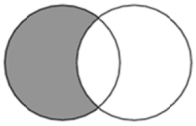
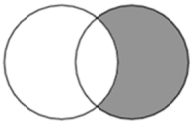
Tabelle 9: Auswertung logischer Ausdrücke

x	***y***	***Logischer Ausdruck***	***Ergebnisinterpretation***
0	0	***x*** > -1 ∧ **y** == 0	*„wahr“* ⇒ *konform*
1	-1	***x*** <= **y** ∨ **y** >= 0	*„falsch“* ⇒ *nonkonform*
-1	0	***x*** ∧ ¬ **y**	*„wahr“* ⇒ *konform*

2.3.2.1 Aussagen auf Mengen

Zwischen Aussagen und Mengen existiert eine Analogie: Die *Konjunktion* von Aussagen entspricht dem Durchschnitt von Mengen, die *Disjunktion* der Vereinigung von Mengen und die *Negation* der Mengendifferenz.

Tabelle 10: Aussagen auf Mengen

$A \wedge B$	$A \vee B$	$A \wedge \neg B$	$\neg A \vee B$
			

Für Konformitätsrichtlinien sind Aussagen auf Mengen insofern relevant, da durch sie ein *Geltungsbereich*, bestehend aus Teilmengen, festgelegt werden kann (Ausschnitt).

Beispiel:

Sei ein Funktionsmodell nach Kapitel 2.2.3 vorliegend. Die Menge $\boldsymbol{M}$ umfasst dann alle Schnittstellen des Modells. Mit $\boldsymbol{A} \subseteq \boldsymbol{M}$ sind die Menge aller Eingänge $\boldsymbol{a} \in \boldsymbol{A}$ sowie mit $\boldsymbol{B} \subseteq \boldsymbol{M}$ die Menge aller Ausgänge $\boldsymbol{b} \in \boldsymbol{B}$ im Funktionsmodell gegeben. Eine Richtlinie fordert eine Namenskonvention für Ausgänge oder Eingänge, vorausgesetzt sie haben einen Bezeichner. Die Menge $\boldsymbol{C} \subseteq \boldsymbol{M}$ umfasst dann alle Schnittstellen mit einem Bezeichner. Für die Anforderung der Richtlinie gilt somit der Geltungsbereich:

$$\boldsymbol{C} \wedge (\boldsymbol{A} \vee \boldsymbol{B})$$

2.3.2.2 Implikationen in Richtlinien

Implikationen sind für Richtlinien ein wichtiges Hilfsmittel, um eine Schlussfolgerung zu ziehen oder eine Abhängigkeit zu beschreiben. Nehmen wir an, dass eine Richtlinie fordert:

„*Wenn die Namenskonvention für Schnittstellen eingehalten wird, werden keine Probleme bei der Codegenerierung aus dem Implementierungsmodell auftreten.*"

Falls nun die Namenskonvention der Schnittstellen im Funktionsmodell eingehalten wurde und keine Probleme bei der Codegenerierung auftraten, ist die Richtlinie wahr, d. h., folglich die Konformität erfüllt. Dies ist der Fall für die Implikation:

„*wahr* $\Rightarrow$ *wahr*"

Die Richtlinie wäre nicht erfüllt (Nonkonformität), wenn die Namenskonvention eingehalten wurde und Probleme bei der Codegenerierung auftraten. In diesem Fall wäre die Implikation:

„*wahr* $\Rightarrow$ *falsch*"

Für den besonderen Fall, wenn die Namenskonvention nicht eingehalten wurde und keine Probleme bei der Codegenerierung auftraten, sind zwei Implikationen möglich:

„*falsch* $\Rightarrow$ *wahr*" oder „*falsch* $\Rightarrow$ *falsch*"

In beiden Fällen ist die Konformität zur Richtlinie nicht erfüllt, gleichwohl die Richtlinie im Ersten der beiden Fälle erfüllt erscheint.

2.3.2.3 Syntax der Aussagenlogik

In der Aussagenlogik ist die Syntax von Formeln prinzipiell aus Zeichenketten aufgebaut, was die Symbolik festlegt. Die folgende Grammatik beschreibt die exakte Syntax von Formeln.

$$\begin{aligned}
\textbf{\textit{Formel}} &= \textit{Proposition} \mid \text{‚}\top\text{‘} \mid \text{‚}\bot\text{‘} \mid \text{‚(‘,‚}\neg\text{ ‘ }\textit{Formel}\text{ ,)‘} \mid \\
&\quad \text{‚(‘ }\textit{Formel Operator Formel}\text{ ,)‘} \\
\textbf{\textit{Operator}} &= \text{‚}\wedge\text{‘} \mid \text{‚}\vee\text{‘} \mid \text{‚}\Rightarrow\text{‘} \mid \text{‚}\Leftarrow\text{‘} \mid \text{‚}\Leftrightarrow\text{ ‘} \\
\textbf{\textit{Proposition}} &= \text{‚p‘ }\textit{Zahl}\text{ \{ }\textit{Zahl}\text{ \}} \\
\textbf{\textit{Zahl}} &= \text{‚0‘} \mid \text{‚1‘} \mid \text{‚2‘} \mid \text{‚3‘} \mid \text{‚4‘} \mid \text{‚5‘} \mid \text{‚6‘} \mid \text{‚7‘} \mid \text{‚8‘} \mid \text{‚9‘}
\end{aligned}$$

Die Symbole auf den linken Seiten der Gleichungen werden in der Fachliteratur als ‚Nicht-Terminalsymbole‘ benannt. Die Symbole, die in Hochkommas eingeschlossen sind, heißen demnach ‚Terminalsymbole‘. Die Grammatik ist so zu verstehen, dass ein Nicht-Terminalsymbol durch eine der mit einem vertikalen Strich (|) getrennten Alternativen auf der rechten Seite der Gleichung ersetzt werden darf. Die Terminalsymbole bleiben in der ursprünglichen Form bestehen. Dabei steht { ... } für beliebig viele Wiederholungen der Symbole innerhalb der geschweiften Klammern.

Beispiel: Formeln der Aussagenlogik sind z. B. $\boldsymbol{A} = \boldsymbol{x_{20}}$ und $\boldsymbol{B} = (\neg \boldsymbol{x_1} \Rightarrow (\boldsymbol{x_1} \wedge \boldsymbol{x_2}))$

2.3.2.4 Erfüllbarkeit, Allgemeingültigkeit, logische Konsequenz

Wesentliche *Aussagen* in den Richtlinien können durch logische Begriffe wie *erfüllbar*, *allgemeingültig*, *logisch äquivalent* und *logische Konsequenz* überführt werden. Wie auch bei der in Kapitel 2.3.1 für Mengen eingeführten Schreibweise werden mit großen griechischen Buchstaben die Mengen von Formeln bezeichnet, also $\Phi = \{\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots\}$. Die Mengen können endlich oder theoretisch unendlich sein. In der Praxis haben wir bei kollaborativen Artefakten jedoch immer eine endliche Menge an Elementen vorliegen.

Erfüllbarkeit, Allgemeingültigkeit, logische Konsequenz **(2.17)**

(1) Eine Formel $\boldsymbol{A}$ heißt *erfüllbar*, falls es eine Belegung α gibt mit $[\boldsymbol{A}]_\alpha = 1$.
(2) Eine Formel $\boldsymbol{A}$ heißt *allgemeingültig* ($\models \boldsymbol{A}$), falls für alle Belegungen α gilt $[\boldsymbol{A}]_\alpha = 1$.
(3) Die Formeln $\boldsymbol{A}$, $\boldsymbol{B}$ heißen *logisch äquivalent,* falls für alle Belegungen α gilt $[\boldsymbol{A}]_\alpha = [\boldsymbol{B}]_\alpha$.
(4) Eine Formel $\boldsymbol{A}$ ist eine *logische Konsequenz* einer Menge Φ ($\Phi \models \boldsymbol{A}$), falls für alle Belegungen α gilt: Falls $[\boldsymbol{B}]_\alpha = 1$ für alle $\boldsymbol{B} \in \Phi$, dann ist $[\boldsymbol{A}]_\alpha = 1$. ($\boldsymbol{A}$ folgt logisch aus Φ)
(5) Eine Formelmenge Ψ *folgt logisch* aus Φ, falls für alle Formeln $\boldsymbol{A}$ aus Ψ gilt $\Phi \models \boldsymbol{A}$.

Eine *formalisierte Richtlinie* besteht beispielsweise aus einer Formel $\boldsymbol{A}$**.** Sie ist *erfüllbar*, wenn es möglich ist, sie ‚*wahr*‘ zu machen. Eine Formel $\boldsymbol{A}$ ist *allgemeingültig*, wenn sie immer ‚*wahr*‘ ist, unabhängig davon, wie man die Aussagenvariablen belegt. Eine Formel $\boldsymbol{A}$ ist *allgemeingültig* genau dann, wenn die Negation $\neg\boldsymbol{A}$ nicht erfüllbar ist. Zwei Formeln $\boldsymbol{A}$ und $\boldsymbol{B}$ sind *logisch äquivalent* genau dann, wenn die Formel $\boldsymbol{A} \Leftrightarrow \boldsymbol{B}$ eine Tautologie ist. Eine

Formel ***A*** *folgt logisch* aus einer Menge Φ von Annahmen, wenn sie ‚*wahr*' ist in jeder Situation, wo alle Annahmen in Φ ‚*wahr*' sind. Falls Φ endlich ist – wovon wir ausgehen − etwa $\Phi = \{\boldsymbol{b_1}, \ldots, \boldsymbol{b_n}\}$, dann folgt ***A*** logisch aus Φ genau dann, wenn die Formel $(\boldsymbol{b_1} \wedge \ldots \wedge \boldsymbol{b_n}) \Rightarrow \boldsymbol{A}$ allgemeingültig (eine Tautologie) ist.

2.3.2.5 Boolesche Algebra

Die *Boolesche Algebra* bildet eine Verallgemeinerungsform der Aussagenlogik. Die logischen Verknüpfungen werden als Rechenoperationen aufgefasst. Anstelle der Symbole (‚∧' | ‚⊥' | ‚…' | ‚¬' | ‚=') werden die Rechenoperationen (‚*' | ‚+ ' | ' | ‚0' | ‚1') benutzt. Dabei beschränkt sich der Bereich der Rechenoperationen nicht nur auf die Wahrheitswerte 0 und 1, sondern kann auch Mengen und deren Elemente umfassen. Der Bereich der Rechenoperationen wird daher bewusst abstrakt gelassen. Nur die Gesetze, welche die Rechenoperationen erfüllen müssen, sind hierbei von Bedeutung. In der Logik wird dies auch als Prinzip des ‚Information Hiding' benannt, welches bei den abstrakten Datentypen und in der objektorientierten Programmierung eine Rolle spielt. Das Prinzip des Information Hiding ist in der Mathematik unter dem Namen ‚algebraische Struktur' bekannt. In einer algebraischen Struktur gibt es eine abstrakte Menge von Objekten zusammen mit Operationen auf den Objekten. Die Namen der Operationen, die Anzahl von Argumenten der Operationen und die Gesetze, welche die Operationen erfüllen müssen, sind festgelegt. Die Menge der Objekte ist abstrakt. Erst in konkreten Beispielen wird die Menge der Objekte genau beschrieben und die Wirkung der Operationen auf den Objekten genau festgelegt. Die Boolesche Algebra ist ein Beispiel für das Konzept der algebraischen Struktur. Andere Beispiele von algebraischen Strukturen aus der Mathematik (Algebra) sind Gruppen, Ringe, Körper, Verbände oder Vektorräume (in [BRON01]).

Boolesche Algebra **(2.18)**

Eine *Boolesche Algebra* ist eine Struktur ⟨A, ',+, *, 0, 1⟩, wobei gilt:

(1) ***A*** ist eine nicht leere Menge.

(2) + und * sind zweistellige Operationen auf ***A*** (Funktionen $\boldsymbol{A}\times\boldsymbol{A}$ nach ***A***).

(3) ' ist eine einstellige Operation auf ***A*** (eine Funktion ***A*** nach ***A***).

(4) 0 und 1 sind zwei ausgezeichnete Elemente von ***A***
und die Axiome der Booleschen Algebra für alle $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \boldsymbol{A}$ gelten.

Axiome der Booleschen Algebra sind:

(I.) *Kommutativität:* $x * y = y * x$ und $x + y = y + x$

(II.) *Assoziativität:* $x * (y * z) = (x * y) * z$ und $x + (y + z) = (x + y) + z$

(III.) *Verschmelzung:* $x * (x + y) = x$ und $x + (x * y) = x$

(IV.) *Distributivität:* $x * (y + z) = (x * y) + (x * z)$ und $x + (y * z) = (x + y) * (x + z)$

(V.) *Komplement:* $x * x' = 0$ und $x + x' = 1$

Aus den Axiomen für die Boolesche Algebra folgen weitere Rechengesetze wie etwa das Gesetz der doppelten Negation oder die aus der Schaltungslogik bekannten DeMorgan'schen Gesetze. Aus mangelnden Platzgründen sei hierzu auf eine anschauliche Einführung in [LIEB96] verwiesen.

Die Formulierung von Richtlinien unterliegt folgenden Rechengesetzen:

(1) *Idempotenz:* $x * x = x$ und $x + x = x$

(2) *Null:* $x * 0 = 0$ und $x + 0 = 0$

(3) *Eins:* $x * 1 = x$ und $x + 1 = 1$

(4) *Komplement:* $0' = 1$ und $1' = 0$

(5) *Negation:* $x'' = x$

(6) *DeMorgan-Gesetz:* $(x * y)' = x' + y'$ und $(x + y)' = x' * y'$

Da die Gesetze in jeder Booleschen Algebra gelten, sagt man, dass sie aus den Axiomen logisch folgen.

2.3.2.6 Gleichungen

Eine *Gleichung* zwischen zwei Ausdrücken gilt in jeder Booleschen Algebra genau dann, wenn sie auch in der Algebra (aus der Mathematik) gilt. Man kann darum sagen, dass die Axiome der Booleschen Algebra eine algebraische Charakterisierung der Aussagenlogik sind. Insbesondere gilt eine Gleichung zwischen zwei Mengenausdrücken genau dann, wenn die entsprechenden Formeln der Aussagenlogik logisch äquivalent sind.

Beispiel:
In der Aussagenlogik gilt die folgende logische Äquivalenz für die Aussagen ***X***, ***Y***:

$$\neg (\boldsymbol{X} \wedge \neg \boldsymbol{Y}) \Leftrightarrow \neg \boldsymbol{X} \vee \boldsymbol{Y}$$

Also gilt für zwei Mengen ***A, B*** die folgende Gleichung:

$$\neg (\boldsymbol{A} \cap \neg \boldsymbol{B}) = \neg \boldsymbol{A} \cup \boldsymbol{B}$$

In diesem Abschnitt wurden elementare Grundlagen der Aussagenlogik eingeführt, die das Fundament für die Formalisierung von Anforderungen in Richtlinien als Aussagen bilden.

2.3.3 Prädikatenlogik 1. Ordnung

Die Prädikatenlogik (First-Order Logic, FOL) gilt als eine Erweiterung der Aussagenlogik. Sie erlaubt es, einen umfassenden Bereich von Argumenten zu formalisieren und auf ihre Gültigkeit zu überprüfen. Hierbei wird eine klare Trennung zwischen ‚Daten' einerseits und ‚Logik' andererseits vorgenommen. Die Prädikatenlogik kennt *Objekte* (Objects) mit unterschiedlichen *Eigenschaften* (Properties) sowie deren *Beziehungen* (Relations) untereinander. Die Objekte sind in diesem Kontext *Artefakte* (Daten und Struktur) mit den Eigenschaften z. B. *leer*, *gleich* oder *verschieden*. Als Beziehungen gelten z. B. *größer als*, *hat* (Eigenschaft), *besitzt nicht* (Eigenschaft).

Satz: „*Aufgrund der Charakteristik spielt die Prädikatenlogik erster Ordnung eine zentrale Rolle für den Lösungsweg, da wir für kollaborative Artefakte logische Beziehungen zwischen einzelnen Artefakten und deren Eigenschaften formal ausdrücken wollen*."

Demnach scheint die FOL ein adäquates Beschreibungsmittel zur Formalisierung von Richtlinien zu sein. In der Prädikatenlogik ist das zentrale Konzept das Prädikat, eben die Eigenschaft oder Beziehung. Welche Prädikate genau zur Verfügung stehen, was der Bereich

der Objekte ist und welche Eigenschaften angewandt werden können, wird im Voraus kontextabhängig (Diskursbereich) festgelegt. Des Weiteren gibt es *Funktionen* (Beziehungen), die Objekten andere Objekte zuordnen. Im vorherigen Kapitel 2.3.2 wurden zusammengesetzte Aussagen der Aussagenlogik daraufhin untersucht, aus welchen einfacheren Aussagen sie mithilfe von Aussage verknüpfenden Bindewörtern, den Junktoren, zusammengesetzt sind. In Erweiterung betrachten wir daher nun Junktoren im Zusammenhang mit Objekten und versuchen, die Prädikate auf die Junktoren anzuwenden.

Des Weiteren definiert die Prädikatenlogik *Variablen* und *Quantoren*. Variable bezeichnen beliebige Objekte eines Diskursbereichs. Quantoren erlauben es, Aussagen wie *„für alle **x** gilt…"* und *„es gibt ein **x** mit...*" zu machen. Quantoren ermöglichen es demnach, Aussagen darüber zu treffen, auf wie viele Elemente ein Prädikat zutrifft. Der *Existenzquantor* sagt aus, dass ein Prädikat auf mindestens ein Element zutrifft, beschreibt also die Existenz mindestens eines unter das Prädikat fallenden Gegenstandes. Der *Allquantor* sagt aus, dass ein Prädikat auf alle Elemente zutrifft.

(1) *Allquantor:* $\forall \boldsymbol{x}$ ("für alle $\boldsymbol{x}$ gilt")

(2) *Existenzquantor:* $\exists \boldsymbol{x}$ ("es gibt ein $\boldsymbol{x}$ mit")

Aus den Variablen und Konstanten werden mithilfe der Funktionen die Terme der Prädikatenlogik aufgebaut. Die *atomaren Formeln* der Prädikatenlogik sind entweder Gleichungen zwischen Termen oder Prädikate angewandt auf Terme. Aus den atomaren Formeln werden mithilfe der aussagenlogischen Junktoren und den Quantoren so genannte *komplexe Formeln* aufgebaut.

2.3.3.1 Syntax der Prädikatenlogik

Die erste Stufe bedeutet für die Prädikatenlogik, dass die Variablen nur für Individuen stehen können, nicht aber für Funktionen oder Prädikate.

In der Prädikatenlogik werden Terme und Formeln durch die folgende Grammatik erzeugt:

Term = *Variable* | *Konstante* | *Funktion* ‚(' *TermListe* ‚)'

TermListe = *Term* { ‚,' *Term* }

Formel = ‚⊤' | ‚⊥' | *Relation* ‚(' *TermListe* ‚)' | ‚(' *Term* '≈' *Term* ‚)'
‚(',¬ ' *Formel* ‚)' | ‚(' *Formel Operator Formel* ‚)' |
Quantor Variable Formel

Operator = ‚∧' | ‚∨' | ‚⇒' | ‚⇐' | ‚⇔'

Quantor = ‚∀' | ‚∃'

Variable = ‚*x*' *Zahl* { *Zahl* }

Konstante = *Bezeichner*

Funktion = *Bezeichner*

Relation = *Bezeichner*

Bei der Prädikatenlogik erster Stufe haben die Rechengesetze, die Anwendung der Booleschen Algebra sowie die Sätze von DeMorgan ihre Entsprechung zur Aussagenlogik. Daher müssen diese nicht weiter untersucht werden und besitzen ihre Gültigkeit. In der Prädikatenlogik erster Stufe können folgende Notationen verwendet werden:

(1) ***x, y*:** für *Variable*

(2) ***Q, R*:** für *Prädikate (Relationssymbole)*

(3) ***f*:** als *Funktionssymbol*

(4) ***c*:** als *Konstante*

(5) ***t*:** als *Term*

(6) ***A, B, C*:** für *Formeln*

(7) **Φ, Γ, Ψ:** für *Mengen von Formeln*

Beispiele:

Beispiele von Termen sind: $\exists\, \boldsymbol{x}\, \boldsymbol{Q}(\boldsymbol{x}) \wedge \boldsymbol{R}(\boldsymbol{x}) \mid \mathit{ist\_Modell}(\boldsymbol{x}_1, \boldsymbol{x}_2)$

Beispiel für eine Formel ist: $\boldsymbol{C}\colon \forall \boldsymbol{x}\, (\mathit{ist\_Elternteil}(\boldsymbol{x}) \Rightarrow \exists \boldsymbol{y}(\mathit{ist\_Kind}(\boldsymbol{y}) \wedge \mathit{Elternteil}(\boldsymbol{x}, \boldsymbol{y})))$

Folgende atomare und zusammengesetzte Aussagen gelten nach der Prädikatenlogik:

Tabelle 11: Formulierung von Prädikaten

	Textuelle Richtlinie	***Logischer Ausdruck***
(a)	*„Es gibt mindestens ein Modell im Prozess.* *Ein Prozess ist genau dann ein Entwicklungsprozess wenn er mindestens ein Modell enthält.“*	Der „Prozess“ wird dargestellt durch eine Menge ***P*** von Artefakten: $\exists\, \boldsymbol{x} \in \boldsymbol{P}\colon \mathit{ist\_Modell}(\boldsymbol{x})$ $\mathit{ist\_Entwicklunsprozess}(\boldsymbol{P}) \Leftrightarrow (\exists\, \boldsymbol{x} \in \boldsymbol{P}\colon \mathit{ist\_Modell}(\boldsymbol{x}))$
(b)	*„Im Prozess werden aus Spezifikationen Modelle für die Evaluierung entworfen.* *Jedes Modell entsteht durch Evaluierung aus genau einer Spezifikation, und umgekehrt entsteht aus jeder Spezifikation durch Evaluierung genau ein Modell.“*	Menge aller Spezifikationen: $\boldsymbol{S} := \{\, \boldsymbol{x} \in \boldsymbol{P}\colon \mathit{ist\_Spezifikation}(\boldsymbol{x}) \,\}$ Menge aller Modelle: $\boldsymbol{M} := \{\, \boldsymbol{x} \in \boldsymbol{P} : \mathit{ist\_Modell}(\boldsymbol{x}) \,\}$ Evaluierung wird dargestellt durch Abbildung: $\boldsymbol{Eval}\colon \boldsymbol{S} \rightarrow \boldsymbol{M}$ Abbildung ist bijektiv: $\mathit{ist\_bijektiv}(\boldsymbol{Eval})$

(c)	*„Der Name eines Modells darf die maximale Textlänge von 550 Zeichen nicht überschreiten.“*	$\boldsymbol{c}$:= 550 $\forall \boldsymbol{x} \in \boldsymbol{M}$: (*Länge*(*Name*($\boldsymbol{x}$)) ≤ $\boldsymbol{c}$)
(d)	*„In einer Spezifikation soll der Gliederungstext aufsteigend sortiert sein.“*	Eine Spezifikation ist eine Menge von nummerierten Gliederungstexten. Das wird dargestellt durch eine Menge $\boldsymbol{S}$ von Teilartefakten und einer Abbildung: $\boldsymbol{Num}: \boldsymbol{S} \rightarrow \mathbb{N}$ Aufsteigend nummeriert: $\forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in \boldsymbol{S}$: *ist_Nachfolger*($\boldsymbol{x}_1$, $\boldsymbol{x}_2$) $\Leftrightarrow$ *Num*($\boldsymbol{x}_2$) = 1 + *Num*($\boldsymbol{x}_1$)
(e)	*„Kollaborative Artefakte sind Spezifikationen und Modelle mit gleicher Referenznummer oder sind nicht vom gleichen Autor erstellt worden.“*	$\forall \boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{S} \cup \boldsymbol{M}$: *ist_kollaborativ*($\boldsymbol{x}, \boldsymbol{y}$) $\Leftrightarrow$ ((*hat_RefNum*($\boldsymbol{x}$) = *hat_RefNum*($\boldsymbol{y}$) $\vee \neg$ (*hat_Autor*($\boldsymbol{x}$) = *hat_Autor*($\boldsymbol{y}$)))

2.4 Abstraktionsmittel

Das Kapitel 2.3 hat Formalisierungstechniken vorgestellt, welche mit ganz allgemeinen Objekten, Eigenschaften und Beziehungen zur logischen Beschreibung auskamen. Welche Prädikate genau zur Verfügung stehen, was der Bereich der Objekte ist und welche Eigenschaften angewandt werden können, muss jedoch im Voraus kontextabhängig für die betrachteten Artefakte festgelegt werden. Das heißt, die Semantiken der Artefakte und deren Eigenschaften bzgl. ihrer Daten und Struktur sowie der Beziehung zwischen ihnen müssen ebenfalls formal und abstrakt beschrieben werden können.

Beispiel:
Im Prozessverlauf sind Artefakte durch ihre Eigenschaften bestimmt, d. h., ihre Bedeutung wird daraus abgeleitet. Sei ein Artefakt in der Tabellenkalkulation (z. B. in Excel) erstellt worden, so kann es entweder ein *Angebot* oder eine *Rechnung* sein, je nachdem, welche Eigenschaft (Daten), z. B. welcher Titel, angegeben ist. Ein Modell kann ein *Funktions-modell* oder ein *Implementierungsmodell* sein, je nachdem, wie feingranular der Abstraktionsgrad (Struktur) modelliert worden ist. Eine Testfallbeschreibung ist direkt zu einem Funktionsmodell entwickelt worden, welches genau die Schnittstellen und deren Parameter für den modellbasierten Test beschreibt (Daten und Struktur).

Um Artefakte zur Semantik formalisierter Richtlinien in Bezug zu setzen und einen hohen Abstraktionsgrad vom werkzeugspezifischen Format eines Artefakts erreichen zu können, ist ein generisch anwendbares Beschreibungsmittel zur Abstraktion erforderlich. Idealerweise ermöglicht es die Transformation werkzeugspezifischer Aspekte in werkzeugunabhängige Aspekte. Dieses Kapitel führt das Paradigma der *Model Driven Architecture* (MDA) ein und stellt die Technik *Metamodellierung* anhand der *Meta Object Facility* (MOF) vor. Des Weiteren wird die grafische Notation der *Unified Modeling Language* (UML) für die formale Modellierung von Artefakten, Artefakt-Eigenschaften sowie der Strukturen eines Datenmodells eingeführt, mit denen sich diverse Abstraktionen formulieren lassen.

2.4.1 Model Driven Architecture (MDA)

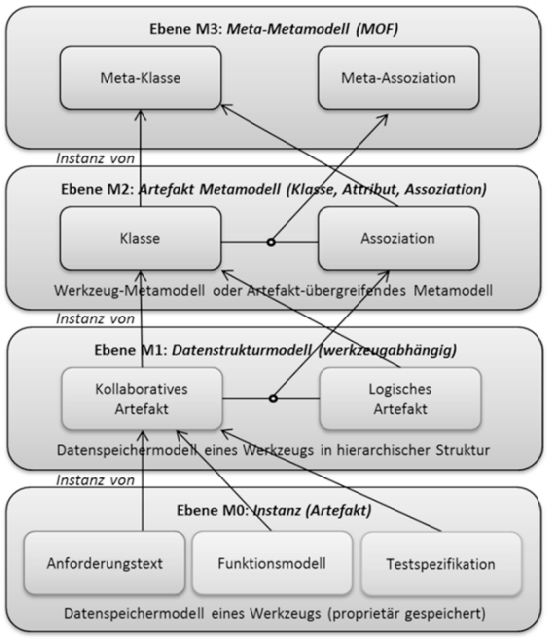
In der Informatik bekannt ist die *Model Driven Architecture* (MDA), ein Abstraktions-Paradigma standardisiert von der *Object Management Group* (OMG). Ursprünglich wurde es für die modellgetriebene Software-Entwicklung standardisiert [MDA00]. Die MDA wurde im Jahre 2000 erstmalig innerhalb der OMG vorgestellt und befindet sich seitdem in der Weiterentwicklung in verschiedenen Versionen. Nach [BORN05; PETRA06] unterscheidet man in der MDA Modelle hinsichtlich der in ihnen enthaltenen Informationen (des Abstraktionsgrades) in Modelle, die unabhängig von einer Plattform definiert sind (Platform Independant Model, PIM), und Modelle, die spezifisch für eine Plattform (Platform Specific Model, PSM) definiert sind. Unter einer Plattform werden hierbei entweder eine Hardwareinfrastruktur oder eine Softwareinfrastruktur verstanden, auf deren Basis das zu erstellende Softwaresystem betrieben wird. Grund für die Einführung der MDA, die die seit Anfang der Neunziger Jahre bestehende *Object Management Architecture* (OMA) abgelöst hat, war die Erkenntnis, dass nicht mehr nur die Interoperabilität von Komponenten eines Softwaresystems sichergestellt werden muss, sondern auch die Interoperabilität von Informationen über diese Komponenten (Modellen) und damit die Interoperabilität und Kooperation von Entwicklungsteams bei der Entwicklung selbst. Wir erkennen, dass der Hintergrund bereits die Kollaboration darstellt.

Der Kerngedanke in der MDA ist die Formalisierung der in Modellen enthaltenen Informationen. Dies führt uns zu der Idee, dass auch die in den kollaborativen Artefakten enthaltenen Eigenschaften durch Informationen und deren Beziehungen als Modell beschreibbar sind. Eine solche Formalisierung erfolgt konsequent in einem Datenmodell für ein kollaboratives Artefakt, dem *Metamodell des kollaborativen Artefakts*. Eine generische Technologie zur Unterstützung der Definition und der automatischen Verarbeitung von Modellen und deren Metamodellen bietet der OMG-Standard *Meta Object Facility* (MOF), welcher auch für die Spezifikation der bekannten Modellierungssprache *Unified Modeling Language* (UML) verwendet wurde [BORN05]. Hierdurch wird eine formale Beschreibung objektorientierter Paradigmen zur Definition von Datenmodellen möglich. Datenmodelle definieren Begriffe, Typen und Beziehungen sowie die Präsenz von Regeln (*Constraints*), welche sogar die automatische Ableitung von Programmcode ermöglichen [PETRA06]. Auf der Grundlage des MOF-Standards bauen diverse Modellierungswerkzeuge für die Modellrepräsentation, die Modelltransformation (Transformationsvorschriften) oder Programmiersprachen auf. Die nachfolgenden Kapitel 2.4.2 sowie 2.4.3 stellen MOF sowie die Metamodellierung anhand der UML vor und übertragen die Konzepte auf die Problemstellung der Arbeit.

2.4.2 Meta Object Facility (MOF)

Die Modellierung nach MOF [MOF20] erfolgt auf vier Abstraktionsebenen, wie in der Abbildung 8 skizziert. In der Ebene ***M0*** befinden sich die tatsächlichen existierenden kollaborativen Artefakte, dies sind durch ein Werkzeug (Computerprogramm) geladene Daten. Die Datenstruktur ist proprietär durch das Werkzeug bestimmt und im Speicher des Rechners abgelegt. Auf der nächsthöheren Ebene ***M1*** ist das konkrete Datenmodell dieser kollaborativen Artefakte definiert. Das *‚logische Artefakt‘* ist eine spezielle Ausprägung kollaborativer Artefakte und wird im Kapitel 4.3 eingeführt. Die Ebene ***M2*** spezifiziert die zur Modellbildung auf Ebene ***M1*** verwendbaren Modellelemente. Dies ermöglicht uns, eine eigene Semantik für kollaborativer Artefakte zu definieren und sogar Prozessinformationen

mit einzuschließen, welche für die Konformitätsprüfung relevant sind. Die Ebene ***M3*** wiederum definiert Sprachelemente, die zur Modellbildung auf Ebene ***M2*** einsetzbar sind.



M3 – Meta-Metamodellebene: Definiert Meta-Klasse, Meta-Assoziation.

M2 – Metamodellebene: Instanziiert Meta-Klasse, Meta-Assoziation und definiert Klasse sowie Assoziation.

M1 – Modellebene: Instanziiert die Klassen kollaboratives Artefakt und die Assoziation zwischen ihnen.

M0 – Instanz (Datenebene): Instanziiert kollaborative Artefakte mit konkreten Daten der angegebenen Klasse.

Abbildung 8: Abstraktionsprinzip, nach MOF

Umrissen definiert der MOF-Standard für den Lösungsansatz wichtige Elemente:

- Eine Menge von Meta-Metamodellkonzepten (ein Meta-Metamodell), welche zur Definition konkreter, MOF-konformer Metamodelle einsetzbar ist. Konkrete Metamodelle kollaborativer Artefakte werden im Kapitel 6.4 aufgeführt.
- Ableitungsregeln für die Erzeugung von CORBA-IDL-Interfaces zum Zugriff auf ein Repository (Modelldatenbank) aus MOF-konformen Metamodellen, auf die weiter in Kapitel 7.2 eingegangen wird.
- Regeln zur Erzeugung (Transformation) von XML-Dokumentenformaten (DTD, Document Type Definition) aus MOF-konformen Metamodellen sowie das kollaborative Austauschformat für Modelle mittels XMI (XMI, XML Metadata Interchange), auf die in den Kapiteln 2.5 (Technologie) und 7.3.3 (Transformationen) ausführlich eingegangen wird.

Die Informations- bzw. Metamodellierung basiert auf dem Klasse-Objekt-Paradigma, welches auch von der objektorientierten Modellierung bekannt ist. Eine *Klasse* beschreibt dabei eine Menge von Objekten mit gleichen *Eigenschaften*. Dies lässt sich analog übertragen auf eine Menge von Elementen in kollaborativen Artefakten mit gleichen Eigenschaften. Objekte derselben Klasse haben denselben *Zustandsraum* und stellen dieselben *Operationen* zur Verfügung. Allgemein können Klassen und Objekte als *Klassifizierer* (Classifier) und *Instanzen* (Instances) beschrieben werden. Ein Klassifizierer beschreibt eine Menge von Instanzen, also eine Menge von Daten oder Informationen, die der Beschreibung des Klassifizierers entsprechen. Die Klassifikationsbeschreibung besteht selbst aus Daten, den *Metadaten*, die weiter klassifiziert werden können. Klassifizierer können wiederum Instanzen eines (Meta-) Klassifizierers sein. Bei Klassifizierern oder Instanzen handelt es sich um Rollen, die durch eine Instanziierungsrelation zwischen zwei

Elementen bestimmt wird. Insofern stellt die Metamodellierung eine Verallgemeinerung der auf die zwei Ebenen beschränkten objektorientierten Modellierung dar, in der die Rollen von Klasse und Objekt klar unterschieden werden. Eine Klasse ist immer nur ein Klassifizierer und ein Objekt immer nur eine Instanz. Die *Instanziierungsrelation* stellt die Grundlage für eine Metamodellierungsarchitektur dar. Sie beschreibt eine Klassifikationshierarchie, indem sie Elemente gemäß einer Instanziierungsrelation wohldefinierten Ebenen zuordnet, und zwar so, dass die Elemente einer Ebene die Instanzen der Elemente der darüberliegenden Ebene bilden. Die Elemente der nächsthöheren Ebene klassifizieren also die Elemente der darunterliegenden Ebene. Auf der untersten Ebene befinden sich die Elemente, die nicht weiter instanziiert werden können. Dies sind also reale Objekte bzw. die kollaborativen Artefakte im Werkzeug (vgl. Abbildung 8). Die Klassifikationshierarchie ist einseitig begrenzt. „Nach oben" kann die Klassifikation prinzipiell beliebig fortgesetzt werden. Metamodellarchitekturen legen aber meist eine vorher bestimmte Anzahl von Ebenen fest. Die oberste Ebene ist selbstreferenziell, d. h., die Elemente der obersten Ebene klassifizieren sich selbst.

Ein *Modell* ist eine Zusammenfassung von Elementen, die derselben Ebene angehören und die Instanzen von einem Modell der nächsthöheren Ebene sind, dem sogenannten *Metamodell*. Die Instanziierungsrelation der Metamodellierungsarchitektur zwischen einzelnen Elementen wird auf Modelle übertragen. In einer nach oben begrenzten Architektur wird demzufolge ein Modell der obersten Ebene durch sich selbst definiert. Die Instanziierungsrelationen zwischen den Elementen, wie Modellen, verlaufen orthogonal zu den Ebenen der Architektur immer in eine Richtung. Es wird also niemals ein Element in Ebene ***n*** Instanz eines Elements aus der Ebene ***n***-1 sein. Nach [BORN05] existieren unterschiedliche Sprachen und Notationsformen, um Metamodelle zu beschreiben. In dieser Arbeit werden nur die MOF basierenden Metamodelle betrachtet, da sie für den genannten Zweck ausreichend ist. Metamodelle dienen uns zur formalen Beschreibung einer werkzeugunabhängigen Datenstruktur, welche idealerweise auch Prozessinformationen mit beinhalten können. Durch diese zusätzlichen, informellen Attribute können wir später sehr geschickt eine Konformitätsprüfung ausführen und den Konformitätsnachweis erbringen. Im Spezialfall stellen Metamodelle selbst wieder kollaborative Artefakte dar, d. h., sie können ebenso in die Konformitätsprüfung mit einbezogen werden, falls Modellierungsrichtlinien für Datenmodelle Anwendung finden sollen (vgl. [AMB05]). Nachfolgend wird noch eine formale Notationstechnik benötigt, um die Metamodelle grafisch zu beschrieben.

2.4.3 Metamodellierung mit der UML

Nach [BORN05; PETRA06] spezifiziert MOF eine Modellierungssprache, mit der die abstrakte Syntax eines Metamodells modelliert werden kann. Die konkrete Syntax der MOF ist eine Teilmenge der *Unified Modeling Language* (UML) [UML22], wie sie in der Ebene ***M2*** (vgl. Abbildung 8) spezifiziert ist.

Zur Beschreibung von werkzeugspezifischen sowie werkzeugunabhängigen Datenmodellen kollaborativer Artefakte können als Notations-elemente die Klassendiagramme und Strukturierungselemente (Package-Konzepte) aus der UML verwendet werden. Bei der Metamodellierung müssen dann ausschließlich die in MOF spezifizierten Beschreibungsmittel verwendet werden, die nachfolgend tabellarisch nach [PETRA06] aufgezählt werden.

Tabelle 12: Elemente der Metamodellierung, nach MOF

Paket	Strukturelemente, welche der Partitionierung und Gliederung innerhalb eines Modells dienen. Ähnlich einem Dateiordner kann eine hierarchische Paketstruktur durch Verschachtelung entstehen. Abhängigkeiten zwischen Paketen ergeben sich oft implizit durch ihre jeweiligen Inhalte und deren Beziehungen zueinander. Sie können im Metamodell durch Abhängigkeitsbeziehungen dargestellt werden.
Klasse	Logische Informationseinheiten werden zu Klassen zusammengefasst. Hierbei wird die Semantik von gleichartigen Objekten festgelegt. Über Assoziationsbeziehung, Spezialisierung und Generalisierung können Klassen miteinander in Verbindung stehen. Hierdurch erweitert sich deren jeweilige Semantik.
Attribut	Eigenschaften eines Objekts, welche sich mit konkreter Datenbelegung beziffern lassen, werden bei der Metamodellierung durch Attribute abgebildet. Dabei erhält jedes Attribut einen definierten Datentyp (primitiver Datentyp, vgl. Kapitel 4.1.1). Auch komplexe Datentypen lassen sich realisieren (Tagged Values). Die Sichtbarkeit der Attribute im Modell ist angegeben durch *public* oder durch *private* eingeschränkt.
Assoziation	Generell können Klassen über die Assoziationsarten Assoziation, Aggregation und Komposition miteinander in Beziehung stehen. Eine Assoziation zwischen zwei Klassen drückt aus, dass diese beiden unmittelbar miteinander in Beziehung stehen.
Aggregation	Mit Aggregation und Komposition wird ausgedrückt, ob ein Objekt aus anderen Objekten zusammengestellt ist.
Komposition	Eine Komposition drückt einen Besitz zwischen zwei Objekten aus. Dies bedeutet, dass ein Objekt nicht ohne das andere existieren darf. Daraus folgt, dass ein Objekt nicht mehrere Besitzer haben kann. Jede modellierte Beziehung wird mit den Eigenschaften wie Kardinalität, Rollenname, Navigierbarkeit, Assoziationsname und Ordnung ergänzt.
Generalisierung, Spezialisierung	Generalisierungen ermöglichen die Vererbung von Beziehungen und Eigenschaften (nur *public* sichtbare) von Objekten. Mehrfachvererbung (Generalisierungsbeziehungen zu mehreren Klassen) ist prinzipiell erlaubt und dient vor allem dazu, ganze Konzepte (Eigenschaften/ Strukturen) auf erbende Klassen anzuwenden.

Metamodelle werden in *Klassendiagrammen* durch die Angabe von Klassen und deren Beziehungen entworfen. Sie dienen uns daher als eigentliche semantische Beschreibung und Strukturangabe von kollaborativen Artefakten. In einem Metamodell kann es ein Vererbungsdiagramm geben, das alle Generalisierungsbeziehungen zwischen den Metaklassen darstellt. In einem weiteren Klassendiagramm können alle Kompositionen modelliert werden, um schließlich den Modellbaum aufzuspannen. In jedem *Paket* kann ein Übersichtsdiagramm existieren, das den Inhalt des Pakets aufzeigt. Annotiert werden

Diagramme durch Kommentare. Dies sind Texte, die bestimmten Modellelementen zugeordnet oder in einem Diagramm frei platziert werden können. Der Inhalt eines Kommentars kann dabei ein beliebig strukturierter Text oder ein formaler Ausdruck (Invariante) sein – z. B. eine (Vor-)Bedingung oder Anforderung aus einer Richtlinie.

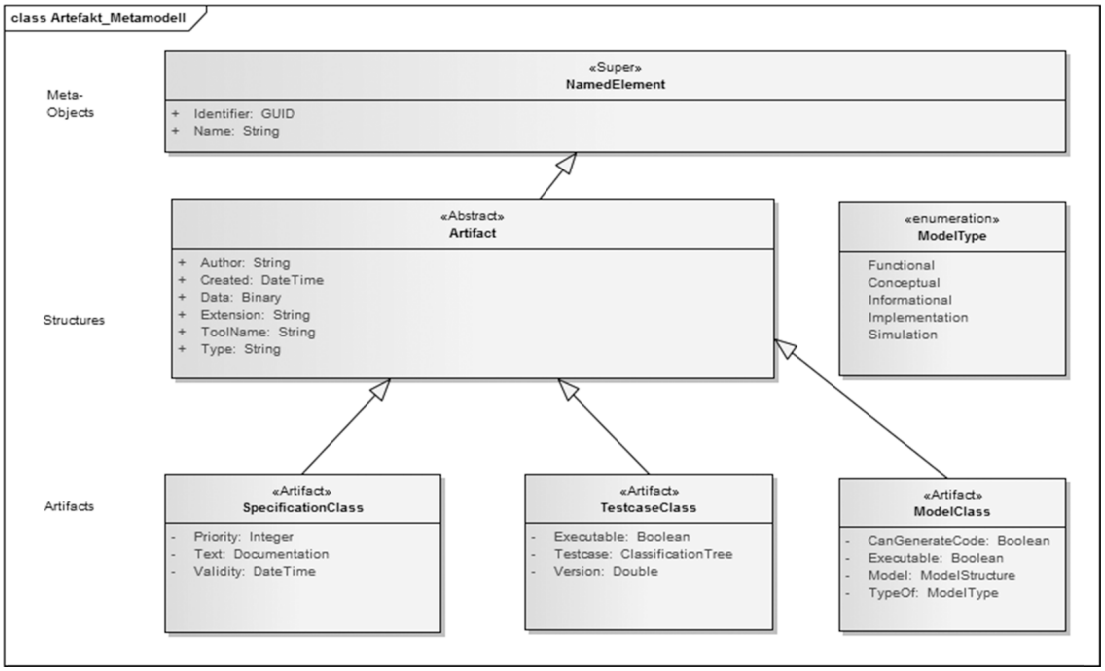


Abbildung 9: Grundlegendes Metamodell eines Artefakts (Auszug)

Beispiel: Das Metamodell in der Abbildung 9 liefert eine generische Beschreibung für Artefakte (*Artifact*), abgeleitet aus einem atomaren Super-Objekt (*NamedElement*). Von diesem darf nur geerbt werden. Des Weiteren werden spezielle Artefakt-Typen für Anforderungen, Modelle und Testfallbeschreibungen (*SpecificationClass*, *ModelClass* und *TestcaseClass*) mit konkreten Eigenschaften modelliert. Die Klassifikation der unterschiedlichen Modell-Typen wird durch eine Enumeration, d. h. eine referenzierte Aufzählung (*ModelType*) erreicht.

Abstraktionen (wie z. B. *Meta-Objects*, *Structures* und *Artifacts*) dienen zur Strukturierung innerhalb des Metamodells. Durch Verschachtelung von Paketen können *Ebenen* modelliert werden. Es entsteht somit eine baumartige Hierarchisierung durch Strukturbäume, welche sehr gut zu den typischen Artefakt-Strukturen korrespondiert. Die Tiefe dieser Bäume steht im Zusammenhang mit der Komplexität des abzubildenden Artefakts. Es ist daher schon frühzeitig zu bedenken (bzw. das kollaborative Artefakt einer Analyse zu unterziehen), wie viele Abstraktionsebenen in einem Modell erwünscht sind und wie sinnvoll oder zweckgebunden jede einzelne Ebene ist.

Ein *Glossar* definiert die Semantik der verwendeten Terminologie und enthält vorrangig Artefakt-spezifische Begriffe und Definitionen. Es ist empfehlenswert, während der Metamodellierung eines Artefakts die Termini in das Glossar einzupflegen. Nützlich ist im Glossar ebenso die Referenz der Begriffe auf ein Artefakt, welche insbesondere dann auch die automatische Ableitung aus einem Metamodell erlaubt.

2.5 Sprachen zur Formalisierung

Für die Konformitätsprüfung ist die Kenntnis von Beschaffenheit und Abfragemöglichkeit von Artefakten relevant sowie die Möglichkeit Bedingungen durch eine Ausdruckssprache zu formulieren. Die Datenstruktur einzelner Artefakte ist in unterschiedlichen Entwicklungsschritten und durch verschiedene Softwarewerkzeuge bedingt meist heterogen. Sie ist wesentlich durch verschiedenartige menschen- oder maschinenlesbare Datenformate bestimmt. Auszeichnungssprachen legen dabei die Artefakt-Struktur der menschenlesbaren Datenformate für eine Persistenz fest. Der Persistenz-Begriff wird verstanden als die Fähigkeit, Daten in nichtflüchtigen Speichermedien, wie bei einem Dateisystem oder einer Datenbank, im binär-proprietären oder im textuellen Datenformat zu speichern. In dieser Arbeit stehen persistente, strukturierte, menschenlesbare Datenformate im Vordergrund, da im kollaborativen Umfeld ein Trend zu diesen als „offen" geltenden Austauschformaten ersichtlich ist (vgl. [ECKE09]). Die nachfolgenden Kapitel geben einen kurzen Überblick über die in dieser Arbeit referenzierten und angewandten Technologie-Standards zur Formulierung von Bedingungen, Datenstrukturierung und zur Datenabfrage von Artefakten.

2.5.1 Extensible Markup Language (XML)

Die *Extensible Markup Language* (XML) ist eine Auszeichnungssprache für Daten und mittlerweile ein De-facto-Standard für die Datenpersistenz in kollaborativen Einsatzbereichen. XML wird vom *World Wide Web Consortium* (W3C) standardisiert und ist in [XML09] näher spezifiziert. Mittels XML lässt sich – unter anderen Möglichkeiten – ein universelles Dateiformat umsetzen, welches hierarchisch abhängige Daten textbasiert strukturiert. Hierbei werden Daten und Struktur voneinander getrennt. Nach [ECKS04] sind die Hauptstrukturierungsmittel für Dokumentinstanzen die *Elemente*. Sie bestehen aus je einer Anfangs- und einer Endemarke (‚*start tag*' und ‚*end tag*'). Zwischen den *Tags* ist der Elementinhalt eingeschlossen (Beispiel: `<datum>12.10.1974</datum>`). Elemente können ineinander geschachtelt werden, wodurch XML-Dokumente eine hierarchische Struktur erhalten. Elemente dürfen auch leer sein (keine Wertebelegung), dann ist die Anfangsmarke gleich der Endemarke (Beispiel: `<leeresElement/>`). Der Name eines Tags gibt gleichzeitig den *Typ* des Elements an. Zusätzlich können *Attribute* innerhalb der Tags mit angegeben werden, welche zusätzliche Typ-Eigenschaften mit angeben (Beispiel: `<element id="1"></element>`). XML-Dokumente sind durch den Menschen lesbar, es können aber auch Binärdaten zwischen den Tags eingebettet werden.

Eine *Dokumenttypdefinition* (*Document Type Definition*, DTD) ist ein in XML-Struktur formulierter Regelsatz, welcher Dokumente eines bestimmten Typs repräsentiert. Ein *Dokumenttyp* ist dabei eine *Klasse* ähnlicher Dokumente, wie beispielsweise Anforderungen, Modelle oder Testdatensätze. Die DTD besteht dabei aus *Elementtypen*, *Attributen von Elementen*, *Entitäten* und *Notationen*. In einer DTD werden die Reihenfolge, die Verschachtelung der Elemente und die Art des Inhalts von Attributen festgelegt, um die Struktur des Dokuments zu bestimmen.

Nach [XML09] ist die Syntax und die Semantik einer DTD ein Bestandteil der XML-Spezifikation. Mit *Document Schema Definition Languages* existiert eine eigene Spezifikation zur Definition von Dokumentstrukturen, Datentypen und Datenbeziehungen in strukturierten Informationsquellen. Neben der einfachen DTD bietet XML-Schema ein etwas mächtigeres Beschreibungsmittel, um inhaltliche Zusammenhänge eines Datenmodells zu beschreiben. Mittels XML-Schema wird die Struktur von XML-Dokumenten vorgegeben. Es wird ein Mittel zur Verfügung gestellt, den Inhalt von Elementen und Attributen in einem

Struktur-Modell anzugeben, z. B. vordefinierte Attribute, Häufigkeit, Beschränkung von Datentypen, formatierte Datumsangaben oder Texte mittels regulärer Ausdrücke. Nachfolgendes Beispiel zeigt ein solches XML-Dokument, welches das XML-Schema für die Klasse *Artifact* des Metamodells aus Abbildung 9 definiert.

Beispiel: XML-Schema der Metamodell-Klasse *Artifact* (Abbildung 9)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Artifact" type="Artifact"/>
 <xs:complexType name="Artifact">
  <xs:complexContent>
   <xs:extension base="NamedElement">
    <xs:sequence>
     <xs:element name="Author" type="xs:String" maxOccurs="1"/>
     <xs:element name="Created" type="xs:DateTime" maxOccurs="1"/>
     <xs:element name="Data" type="xs:String" maxOccurs="1"/>
     <xs:element name="Extension" type="xs:String" minOccurs="1"/>
     <xs:element name="ToolName" type="xs:String" minOccurs="1" />
     <xs:element name="Type" type="xs:String" maxOccurs="1"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:schema>
```

Das XML-Schema führt alle Klassen-Attribute `Author`, `Created`, `Data`, `Extension`, `ToolName` und `Type` samt ihrer Datentypen und der Auftretungshäufigkeit (`minOccurs`, `maxOccurs`) auf. Durch Einrückung ist die hierarchische Verschachtelung der Tags zu erkennen. Prinzipiell lässt sich somit ein komplettes Metamodell als XML-Schema ableiten.

Diskussion:

Ein XML-basiertes Artefakt stellt eine Instanz eines XML-Schemas dar. Die Vorgaben des Schemas zur inhaltlichen Strukturierung sind quasi Anforderungen an das XML-basierte kollaborative Artefakt. Die Konformitätsprüfung, ob ein XML-basiertes, kollaboratives Artefakt einer Anforderung entspricht, wird häufig auch als ‚Schema-Validation' bezeichnet. Allerdings wird hierbei nur strukturell und nicht die Semantik der Daten (Elemente) geprüft. Durch die XML-Auszeichnungssprache wird ein Standard zur Verfügung gestellt, der flexibel genug ist, um an diverse Anwendungsfelder angepasst zu werden. In der modellbasierten Entwicklung ist beispielsweise das durch die OMG standardisierte *XML Metadata Interchange* (XMI) ein XML-basiertes Austauschformat für Modelle.

2.5.2 XML Metadata Interchange (XMI)

Nach [XMI21] definiert die *XML Metadata Interchange* (XMI) ein XML-Integrations-Framework mit der definierten Ableitung von MOF-Modellen zu XML-Artefakten, standardisiert durch die OMG. Die nach XML serialisierten Modelle dienen vielfach für den Austausch von Modellen zwischen verschiedenen Modellierungswerkzeugen. XMI ist wie XML ein textbasiertes Austauschformat für Metadaten und Daten, die auf einem Metamodell (MOF-Modell, vgl. Kapitel 2.4.2) basieren. Ein auf der MOF basierendes Modell ist üblicherweise ein grafisches Modell, welches durch den XMI-Standard in eine textbasierte Form gebracht wird. Es entsteht eine auf einer XML-DTD basierenden Repräsentation, die von einem Transformator automatisch verarbeitet werden kann. Es bietet sich in Hinblick auf die MDA an, durch mehrere Transformationsregeln, Modelle von Ebene zu Ebene zu transformieren bzw. ein PIM in ein PSM umzuwandeln oder umgekehrt (Abstraktion).

2.5.1 XML Path Language (XPath)

Die *XML Path Language* (XPath) ist eine Sprache zur Adressierung angegebener Stellen in XML-Dokumenten sowie zur Navigation innerhalb komplexer XML-Dokumente (vgl. auch [ECKS04]). Sie bildet die Grundlage für weitere XML-Abfragesprachen und soll daher kurz eingeführt werden. XPath basiert dabei auf einem Datenmodell. Ist dieses im Speicher eines Rechners geladen, wird ein XML-basiertes Artefakt als Baum gemäß den Knotentypen abgebildet. Ein XPath Ausdruck wird ausgehend von einem bestimmten Knoten (Kontextknoten) ausgewertet und liefert als Ergebnis eine Knotenmenge, welche auch die leere Menge sein darf. Der Kontextknoten ist entweder der Wurzelknoten oder ein anderer Knoten des XML-basierten Artefakts, welcher sich aus dem Kontext der *Abfrage* (Query) ergibt. Die mengenorientierte Ausdrucksform und Ergebnisrückgabe erlaubt die Verwendung als eine Art Abfragesprache auf Dokumenten-Ebene. Durch Angabe von Prädikaten kann die Ergebnisrückgabe weiter eingeschränkt werden. Prädikate werden in einem Ausdruck als eckige Klammern eingeschlossen und können in beliebiger Zahl hintereinander aufgeführt werden, wobei die Reihenfolge von Bedeutung ist. Prädikate können XPath-Ausdrücke enthalten, außerdem kann eine Vielzahl von Funktionen verwendet werden. Eine speziell für XML-Datenbanken entwickelte Abfragesprache ist die *XML Query Language* (XQuery) und benutzt eine an SQL (Kapitel 2.5.3) angelehnte Syntax. Technisch basiert sie auf XPath-Technologie und XML-Schema (Kapitel 2.5.1) als hinterlegtes Datenmodell sowie einer umfangreichen Funktionsbibliothek.

2.5.2 Object Constraint Language (OCL)

Die *Object Constraint Language* (OCL) [OCL20] ist eine Ausdruckssprache zur textuellen Spezifikation von Bedingungen in den UML-Diagrammen. Eine gute Einführung gibt [CLAR02]. Die OCL ist für uns interessant, da sie formalisierte Bedingungen auf Modellebene erlaubt. Die OCL verwendet Elemente der Mengentheorie (Kapitel 2.3.1) sowie der Prädikatenlogik (Kapitel 2.3.3), hat jedoch keine Sprachkonstrukte zur Programmablaufsteuerung. Die mengenorientierte Ausdrucksform (Collections) erlaubt die Formulierung von Bedingungen auf einem Bereich von Modellelementen (Kontext) und deren jeweilige Auswertung. In der OCL werden verschiedene *Zusicherungen* (Constraints) klassifiziert. Hierzu zählen (a) Invarianten, die zu jeder Zeit für eine Instanz oder Assoziation gelten; (b) Vor- und Nachbedingungen, welche zu jedem Zeitpunkt gelten müssen, an dem die Ausführung der zugehörigen Operation beginnt oder endet; (c) initiale oder abgeleitete Werte. Diese stellen Bedingungen für Ausgangs- und abgeleitete Werte dar; (d) die Festlegung von Attributen und Operationen, die nicht im Modell enthalten sind.

Eine Zusicherung (Schlüsselwort: `inv`) gilt im Rahmen eines angegebenen *Kontextes* (Schlüsselwort: `context`). Dies kann ein beliebiges Modell-Element (Entity) wie z. B. eine Klasse, ein Typ, eine Schnittstelle (Interface) oder eine Komponente sein, auf welches durch das Schlüsselwort `self` optional referenziert wird. Man unterscheidet den Kontexttyp und die Kontextinstanz. Auf Letztere beziehen sich die Angaben einer Zusicherung.

In der OCL werden Basis-Operatoren für den Zugriff auf Attribute zur Verfügung gestellt, z. B. *max*(), *min*(), *string.size*(), *string.substring*(*int*, *int*) usw. – auf die mittels der Pfeil-Notation ‚→' zugegriffen wird. OCL-Editoren bieten teilweise noch weitreichendere Funktionalitäten an.

Beispiel:
Als *Anforderung* kann eine Zusicherung festlegen, dass für eine Instanz der Klasse *Artifact* (nach Abbildung 9) der Wert des Attributs `Extension` mindestens aus drei Zeichen bestehen muss und nicht leer sein darf. Folgende Zusicherung muss demnach in OCL gelten:

```
context Artifact inv:
self.Extension->string.size() >= 3 and self.Extension->notEmpty()
```

Der OCL-Ausdruck als Invariante ist immer vom Typ Boolean, d. h. wir erhalten einen Wahrheitswert, der uns eine Ergebnisinterpretation für die Konformität erlaubt. Die OCL bietet uns somit ein geeignetes Ausdrucksmittel zur Darstellung semi-formaler Anforderungen bezogen auf einen Geltungsbereich. Die Erfüllung der Anforderung ist somit die Sicherstellung der Konformität eines Modell-Attributs durch einen formalen Ausdruck.

2.5.3 Structured Query Language (SQL)

Neben den bisher vorgestellten Datenabfragemöglichkeiten auf Dokumenten- bzw. Modell-Ebene existiert für Datenbank-basierte Artefakte ein De-facto-Standard für relationale Datenbanksysteme (DBS). Die *Structured Query Language* (SQL) ist eine weitverbreitete Sprache zur Definition, Abfrage und Manipulation von Daten in relationalen Datenbanken, standardisiert von der *International Organisation for Standardization* (ISO) [ISO9075]. Die SQL ermöglicht es dem Anwender, Datenbank- und Relationsstrukturen zu erstellen sowie Datensätze in der Datenbank abzufragen, zu filtern oder zu verwalten. Ferner ist die Ausführung von einfachen und komplexen Abfragen bis hin zu einer Rechteverwaltung auf Datenbankmanagementebene möglich. Mittlerweile existieren auch Standards (wie z. B. ISO/IEC 9075-14) für die Integration und auch Ableitung von SQL zu XML. Wie auch beim XML-Schema aus Kapitel 2.5.1 unterliegt eine Datenbank einem Daten-Schema (ISO/IEC 9075-11). Nach [KEMP99] legt das Schema dabei fest, welche Daten in einer Datenbank in welcher Form gespeichert werden können und welche Beziehungen zwischen den Daten bestehen.

Die SQL besteht aus der Integration einer *Data Definition Language* (DDL) und einer *Data Manipulation Language* (DML). Durch die DDL ist das Schreiben, Ändern und Entfernen von Datenbankstrukturen möglich. Die DML ermöglicht das Auslesen, Schreiben, Ändern und Löschen von Informationen in einer Datenbank. Es ist mittels SQL möglich, Daten auch zu transformieren. In diesem Falle stellen mehrere aufeinanderfolgende SQL-Ausdrücke die *Transformationsvorschrift* dar. Eine solche Abfolge von SQL-Ausdrücken kann durch ein DBS als *gespeicherte Prozedur* (Stored Procedure) benannt abgelegt werden und stellt somit einen neuen Datenbankbefehl dar. Dies wird beispielsweise auch zur Erstellung *virtueller Tabellen* (Views) genutzt. Mit einer virtuellen Tabelle oder *Sicht* können Daten aus einer oder mehreren Tabellen oder Sichten in einer anderen Weise dargestellt werden. *Datenbanktrigger* (Trigger) stellen eine Sonderform von gespeicherten Prozeduren dar. Es werden wiederum DML-Trigger und DDL-Trigger unterschieden. Trigger werden automatisch ausgeführt, wenn ein Ereignis auf dem Datenbankserver eintritt (ereignisgesteuerte Aktion).

Die SQL bietet ein Ausdrucksmittel zur Darstellung semiformaler Anforderungen bezogen auf Datenbanken und einen definierten Geltungsbereich. Die Erfüllung der Anforderung ist die Sicherstellung der Konformität eines Datenbank-Attributs durch einen formalen Ausdruck.

2.5.4 Language-Integrated Query (LINQ)

Abfragesprachen von XML-basierten Dokumenten, Modellen sowie für Datenbank-basierte Artefakte wurden in den vorangegangenen Kapiteln vorgestellt. Schließlich fehlt noch eine Abfragesprache für ein im Speicher eines Rechners geladenen Objekt-Modells. Dieses Objekt-Modell ist wiederum auch eine baumartige Struktur, gerade dann, wenn es sich um ein in den Speicher eingelesenes XML-Artefakt handelt. Eine Abfragesprache für Objekt-Modelle sowie ein Technologie-Framework zur Integration aller Abfragesprachen für XML-basierte Dokumente, Modelle sowie für Datenbank-basierte Artefakte bietet das LINQ-Framework [LINQ07]. Nach [PIAL07; MARG08] handelt es sich bei *Language-Integrated Queries* (LINQ) um eine von Microsoft entwickelte Methodik, welche ein Programmier- und Zugriffsmodell für alle vorher genannten Datenformate und -formen ist.

LINQ ermöglicht das Abfragen und Ändern (Transformation) von Daten über eine Abstraktionsschicht, welche den Zugriff und die Operationen auf Daten unabhängig von einer bestimmten Datenquelle realisiert. Insbesondere wird auch die Transformation von Daten und Datenstrukturen unterstützt. Es handelt sich um eine im .NET-Framework integrierte Software-Bibliothek, mit deren Hilfe man SQL-ähnliche Abfragen in einer deklarativen Sprache in den Programmcode einbetten kann. Auf eine Hinterlegung in einem ‚String' wird verzichtet, was beim Kompilieren zu einer Optimierung oder einer frühzeitigen Fehlererkennung führt. Nach [PIAL07; MARG08] lassen sich durch LINQ drei Arten von Datenquellen abfragen und modifizieren:

(a) *LINQ to XML* (für XML-Daten); (b) *LINQ to SQL* (für relationale Datenbanken); (c) *LINQ to DataSet bzw. Collections* im Hauptspeicher (für ADO.NET-DataSets). *LINQ to XML*, früher bekannt als XLINQ, stellt eine Schnittstelle zum Lesen und Bearbeiten von XML-Dokumenten zur Verfügung. Durch *LINQ to SQL*, ehemals bekannt als DLINQ, ist es möglich, auf relationalen Datenbanken zu arbeiten. *LINQ to DataSet* ermöglicht es, ADO.NET-DataSets zu verwenden.

Ein weiteres wichtiges Merkmal von LINQ ist die Funktionalität, auf *Objekt-Collections* im Hauptspeicher des Programms navigieren und mengenorientiert zugreifen zu können.

Im nachfolgenden Beispiel wird eine *Anforderung* als LINQ-Ausdruck vorgestellt.

Beispiel:
Sei $\boldsymbol{M}$ die Menge der Zahlen mit $\boldsymbol{M}$ = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 }. Die Teilmenge $\boldsymbol{A} \subseteq \boldsymbol{M}$ sei dann definiert durch die Anforderung $\boldsymbol{A} = \{ \boldsymbol{a} \in \boldsymbol{M}: \boldsymbol{a} < 5 \}$.

```
int[] M = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

   var A = from a in M
           where a < 5
   select a;
```

Ergebnis dieses LINQ-Ausdruckes ist die im Speicher gebildete Menge $\boldsymbol{A} = \{ 4, 1, 3, 2, 0 \}$.

LINQ stellt somit eine Sammlung von Standardabfrageoperationen zur Verfügung, welche die Funktionen der darunterliegenden Datenschicht steuert. So kann eine Navigation, Filterung und Ausführung weiterer *Abfragen* (Queries) für unterschiedliche Datenquellen durchgeführt werden. LINQ bietet ein Ausdrucksmittel zur Darstellung semi-formaler Anforderungen bezogen auf Datenquellen (Objekt, XML, Modell oder Datenbank) sowie einen definierten Geltungsbereich.

Diskussion:

Die SQL-Technologie wurde entwickelt, um auf Daten zu agieren, die durch ein relationales Datenbankmodell repräsentiert werden. Auf diese Art deklarativer Sprache baut LINQ auf, um als DML die referenzierten Daten durch eine einheitliche Schnittstelle zu verschiedenen Datenquellen herzustellen. Dies muss im Gegensatz zu SQL keine Datenbank sein, sondern kann eine Objekt-, XML- oder Modell-Struktur sein. Da LINQ in erster Linie nur zum Lesen, Schreiben und Ändern von Daten gedacht ist, fehlen im Gegensatz zu SQL Kontrollmechanismen für ein spezielles DBS. Es ist somit nicht möglich, Benutzer auf Datenbankmanagementebene anzulegen oder eine Rechteverwaltung vorzunehmen. Es ist ferner nicht vorgesehen, LINQ als DDL zu verwenden. Es besteht also nicht die Möglichkeit, wie bei der Metamodellierung (Kapitel 2.4.3), ein Datenstrukturmodell zu definieren. LINQ ist als Brücken-Technologie zwischen persistentem Datenspeicher und einem objektorientierten Programm zu verstehen, welche die einheitliche Abfrage und Bearbeitung verschiedener Datenquellen ermöglicht.

Formal handelt es sich bei LINQ und SQL um eine deklarative Sprache im erweiterten Sinne zu OCL oder XPath, welche keine Kontrollflusssteuerung vorsehen. Syntaktisch ähneln sich beide Sprachen LINQ und SQL sehr. Beide Abfragesprachen sind für uns interessant, da sie aus Elementen der Mengentheorie, Aussagenlogik sowie der Prädikatenlogik bestehen (Kapitel 2.3.1-3) und die Formalisierung von Anforderungen durch ihre Sprachmächtigkeit (Befehle und Operationen) an kollaborativen Artefakten unterstützen.

Für die Arbeit erforderlich, haben wir in den vorangegangenen Kapiteln 2.5.1-4 einen kurzen Überblick über grundlegende Technologie-Standards zur Datenstrukturierung in kollaborativen Umgebungen mittels XML und XMI erhalten, können nun auswertbare Bedingungen aus Anforderungen mittels OCL formulieren und haben Technologien zur werkzeugunabhängigen Datenabfrage und Transformation auf kollaborativen Artefakten kennengelernt.

3 Stand der Technik im Bereich der Konformitätsprüfung

In diesem dritten Kapitel wird, ausgehend von einer Prozessanalyse im Kontext der Entwicklung eingebetteter Systeme, der derzeitige Stand von Wissenschaft und Technik regelbasierter Konformitätsprüfung vorgestellt. Analysen bezüglich relevanter Richtlinien aus Normungen oder Standardisierungen, praktizierte Prüfverfahren und eingesetzte Software-Werkzeuge im Bereich modellbasierter Entwicklung eingebetteter Systeme vertiefen das Kapitel. Zudem werden Entwicklungsrichtlinien aus dem Engineering-Prozess am Beispiel der Automobilentwicklung genauer beleuchtet. Diese sind relevant bzgl. der in dieser Arbeit durchgeführten Validierung (Fallstudie in Kapitel 7) des Ansatzes. Andere technische Domänen wie Luft- und Raumfahrt, Bahntechnik oder der Maschinenbau unterscheiden sich in den hier betrachteten Prozessphasen nur marginal und zumeist nur in den geltenden Qualitätsmodellen (Qualitätsstandards), weniger in den regelungstechnisch-orientierten Implementierungsvarianten für eingebettete Systeme. Da der gesamte Produktentstehungsprozess eines komplexen Steuergerätes (z. B. einer Motorsteuerung im Kfz) mehrere komplexe Phasen umfassen kann, wird in dieser Arbeit der Fokus auf die kollaborativen Phasen des Systementwurfs gelegt. Diese Phasen umspannen die Anforderungsdefinition, die modellbasierte Funktionsspezifikation, den modellbasierten Systementwurf und Modulspezifikation sowie die Testspezifikation für Modelle in der frühen Phase des V-Modells.

3.1 Prozessanalyse

Eine in der Elektronikentwicklung eines Automobils durchgeführte Prozessanalyse aus vorangegangenen Forschungsarbeiten [Kap. 11; Nr.23] sowie unter Miteinbeziehung anderer Untersuchungen [MUTZ05; GOTT06; ADL07] soll zu Beginn des Kapitels einen Einblick in die heutzutage praktizierten Entwicklungsprozesse eingebetteter Systeme schaffen.

Heute gerät der Entwicklungsprozess durch mehrere Aspekte unter Druck:

(1) *Kostenoptimierung* und *Reifegradabsicherung* durch Reduktion der Steuergeräteanzahl und der damit verbundenen Schnittstellen,

(2) Integration von *Standardisierungen* im Bereich der Elektronik und Software durch firmenübergreifende Kooperationen,

(3) *Wiederverwendung* und *Skalierbarkeit* hinsichtlich Funktionen und Modulen.

Aus Aspekt (1) ergibt sich, dass mit steigender Funktionalität die Verteilung von Funktionen von mehreren Steuergeräten auf die begrenzte oder gar abnehmende Menge abgebildet werden muss. Wettbewerbsdifferenzierende Funktionen möchte ein Produkthersteller möglichst unabhängig vom Zulieferer entwickeln. So begünstigt Aspekt (2), dass die an der Entwicklung beteiligten Personen in der Lage sein müssen, von Dritten entwickelte

Software- und Hardwarekomponenten zuverlässig in ihre technischen Produkte integrieren zu können. Dieser Schritt gelingt nur, wenn alle am Entwicklungsprozess involvierten Personen das gleiche Verständnis von einer Spezifikation, Systemfunktion und Systemtest besitzen und die ausgetauschten Entwicklungsartefakte interne Qualitätsrichtlinien sowie eine abgestimmte Hersteller-Zulieferer-Konvention einhalten. Tatsächlich divergieren die angestrebten Ziele mit den praktizierten Vorgehensweisen in den Engineering-Prozessen und werden demnach nur unzureichend erfüllt (vgl. [EKKA06]). Zum einen ist dies bedingt durch (a) das Vorgehen in der Systementwicklung und (b) dem Pflegeaufwand von Lasten- und Pflichtenheften, zum anderen (c) durch die Schnittstelle zwischen Hersteller und Zulieferer.

(a) In [LEDE04] wird das traditionelle Vorgehen in der Systementwicklung am Anwendungsbeispiel der Steuergeräteentwicklung in der Automobilindustrie wie folgt bemängelt: Zunächst erfolgt die Erstellung des Lastenheftes bei Projektbeginn, welches die Anforderungen an das System „Fahrzeug-Elektronik“ und seine einzelnen Komponenten aus Sicht des Auftraggebers verbal beschreibt. Das Lastenheft stellt die Basis für die Projektbearbeitung dar, die in den Musterphasen (A-, B-, C-Muster) durchgeführt wird. Dies bedeutet, dass sich die funktionalen Eigenschaften eines Systems in der C-Musterphase nicht mehr ändern dürfen, sondern lediglich Fehlerkorrekturen erfolgen – so die Theorie. Die Praxis sieht jedoch anders aus, da viele Änderungen in der Software des eingebetteten Systems selbst, darunter wesentliche funktionale Erweiterungen, noch in der C-Musterphase anfallen. Dieses Vorgehen bringt daher ein hohes Risiko für die Projekte mit sich, da der SOP (Start of Production) deutlich verzögert wird, Rückruf-Aktionen erforderlich sind oder das Projekt vollständig scheitert.

(b) Anforderungen sind häufig Änderungen unterworfen, die sogar erst kurz vor der Serieneinführung der Steuergeräte auftreten. Der daraus resultierende Aufwand für die Pflege der Anforderungsdokumente (Lasten- und Pflichtenhefte) ist nicht zu unterschätzen. Des Weiteren sind widersprüchliche Aussagen bei Anforderungsdokumenten in reiner Textform nur schwer zu erkennen. Schon in der frühen Phase der Entwicklung können demnach Inkonsistenzen auftreten, die den Entwicklungsprozess durch ständige Nachbesserung verlangsamen oder gar zu einem nicht funktionierenden Gesamtsystem führen. Lasten- und Pflichtenhefte werden immer noch mit unterschiedlichen Werkzeugen erstellt. Resultate sind unterschiedliche Dateiformate und verschiedene Repräsentationsformen wie Blockdiagramme, Zustandsautomaten oder Spezifikationen in Prosaform. Wiederverwendung findet teilweise nur durch einfaches Kopieren von Dokumenten statt.

(c) Bedingt durch die verteilte Zuordnung der Zuständigkeit von Automobilhersteller und verschiedener Zulieferer zu einzelnen Steuergeräten treten häufig Probleme bei der Schnittstelle zwischen Hersteller und Zulieferer auf, wenn die logische Zusammenarbeit von Steuergeräten aufgrund mehrdeutiger Anforderungen, inkonsistenter Funktionsmodelle oder unvollständiger Testfallbeschreibungen nicht hinreichend genau spezifiziert ist. Die Prüfung auf Vollständigkeit der benötigten Entwicklungsartefakte stellt eine schwierige Aufgabe dar. So wird in [WAGN03] festgestellt: „*Werden Definitionslücken in den einzelnen Lastenheften oder Pflichtenheften im Hinblick auf Vollständigkeit, Eindeutigkeit und Widerspruchsfreiheit nicht durch eine intensive Zusammenarbeit zwischen Hersteller und Zulieferern während der Fahrzeugentwicklung identifiziert und behoben, kann dies zu signifikanten Abweichungen und erheblichen Problemen bei der Integration des Gesamtnetzwerks führen.*“

Die Phasen innerhalb der Entwicklungsprozesse erfolgen demnach oftmals keiner stringenten Vorgehensweise oder Richtlinien. Bisher sind Systementwickler, die sich um

Einhaltung der übergreifenden Konsistenz von Entwicklungsartefakten (wie Dateien, Dokumenten, Modellen) bemühen, auf sich alleine gestellt. Das Erfüllen von Konformitätskriterien kann in kollaborativen Prozessen von ihnen häufig nur durch manuelles Vergleichen einer großen Menge von Daten erreicht werden. Diese Tätigkeiten geschehen heute meist manuell, da sie typischerweise einen Übergang zwischen Artefakten betreffen, die mit verschiedenen Werkzeugen erstellt wurden. Eine nähere Betrachtung der Tätigkeiten, die zur werkzeugübergreifenden Konformitätssicherung durchgeführt werden müssen zeigt, dass diese bezüglich des intellektuellen Anspruchs oftmals sehr einfach sind. Die Belastung für den Entwickler entsteht durch die große Anzahl an Wiederholungen, mit der er diese Tätigkeiten ausführen muss. Gerade derartige Tätigkeiten sind es aber auch, die sich besonders gut durch die regelbasierte Konformitätsprüfung automatisieren lassen.

3.2 Normungen

Gerade im Entwicklungsbereich sicherheitsrelevanter Produkte, wie in der Einführung beschrieben, unterliegen kollaborative Prozesse heutzutage einer möglichst kontinuierlichen Sicherstellung von Produktqualität und Systemzuverlässigkeit (so auch [CON06+]). Bei der Entwicklung funktionssicherer Anwendungen, die von eingebetteter Software im Auto gesteuert werden, kommen eine Vielzahl an unterschiedlichen internationalen Richtlinien, Industriestandards und Industrienormen bezüglich Qualitäts-, Sicherheits- und Zuverlässigkeitsabsicherung sowie rechtlicher Bestimmungen der Gesetzgebung bzgl. Produkthaftung innerhalb der Systementwicklung in Betracht. Andere Domänen, wie z. B. die Luftfahrt, unterliegen schon seit langer Zeit sehr hohen Anforderungen bezüglich funktionaler Sicherheit elektronischer Systeme, die im zugrunde liegenden Standard DO-178B [DO178B] definiert sind. In den letzten Jahren sind die Anforderungen bezüglich der Anwendung dieses Standards stetig gestiegen. Das hat dazu geführt, dass selbst elektronische Systeme ohne Sicherheitsverantwortung, wie z. B. Systeme im Bereich Komfortelektronik, heutzutage grundlegende Anforderungen bezüglich funktionaler Sicherheit erfüllen müssen, bevor sie in einem Flugzeug verbaut werden dürfen. Die Umsetzung solcher Richtlinien ist domänenspezifisch und sogar teils unternehmens-spezifisch. Je nach Anwendungsdomäne gibt es in der Nachweisführung unterschiedliche Schwerpunkte, und es sind verschiedene Maßnahmen zum Erreichen der Sicherheits- und Verfügbarkeitsziele erforderlich. Ein Beispiel ist die Domänen übergreifende Norm IEC 61508 [IEC615] sowie die teils daraus abgeleiteten branchenspezifischen Normen für die Domänen:

- *Bahnwesen*, nach DIN EN 50126 / 50128 / 50129 / 50159
- *Luft-/Raumfahrttechnik*, nach RTCA DO-178B / DO-254
- *Defence*, nach Def Stan 00-56
- *Automotive*, nach ISO CD 26262
- *Prozessindustrie*, nach DIN IEC 61511
- *Maschinenbau*, nach DIN EN 62061
- *Medizinische elektrische Geräte*, nach DIN EN 60601

Die IEC 61508 ist eine internationale Norm zur Schaffung von elektrischen, elektronischen und programmierbar elektronischen (E/E/PE) Systemen, die eine Sicherheitsfunktion ausführen, und wird von der *International Electrotechnical Commission* (IEC) heraus-gegeben. Die Norm besteht aus sieben Teilen und trägt den Titel ‚Funktionale Sicherheit

sicherheitsbezogener elektrischer/elektronischer/programmierbar elektronischer Systeme'. Sie wurde 1998 veröffentlicht, wovon einige Teile im Jahr 2000 in einer überarbeiteten Fassung neu veröffentlicht wurden. Vom Europäischen Komitee für Normung (CEN) wurde die Norm im Jahr 2001 inhaltsgleich als EN 61508 übernommen. In Deutschland hat sie als deutsche Fassung unter den Namen DIN EN 61508 und VDE 0803 ihre Gültigkeit. Entwicklungsrichtlinien, meist natürlich sprachlich dokumentiert, sind häufig von solch einer Norm abgeleitet worden. Das Hauptanliegen von Entwicklungsrichtlinien in der modellbasierten Entwicklung von eingebetteten Systemen ist die Verbesserung und die Absicherung des Spezifikations- und Entwurfsprozesses bis hin zur ihrer Implementierung, der darin verwendeten Methoden sowie der eingesetzten Werkzeuge. Die Einhaltung bewirkt, dass die resultierende Qualität bei Entwicklung eines eingebetteten Systems durch regelkonforme Artefakte bereits in frühen Phasen signifikant erhöht und die Komplexität in der Entwicklung zukünftig beherrschbar bleibt.

3.3 Standardisierungen

Für die Kollaboration in einer unternehmensübergreifenden Zusammenarbeit bei der Entwicklung eingebetteter Systeme und zum Zwecke der Prozessoptimierung in technischen Domänen sind viele Zusammenschlüsse von Interessengruppen mit unterschiedlicher Zweckvorgabe in verschiedenen Konsortien (Gremien) für verschiedene Domänen und Bereiche entstanden. In dieser Arbeit soll aus Platzgründen nur Bezug auf den Systementwurf und den relevanten Konsortien innerhalb der Automobilindustrie eingegangen werden. Wie bereits erwähnt, existieren für jede andere technische Domäne ebenso Interessengruppen und Konsortien mit ähnlicher Zielrichtung. In diesem Abschnitt werden Gremien für Qualitätssicherung und Standardisierung von Entwicklungsprozessen/ Artefakten aufgeführt, welche maßgeblich die Richtlinien-Basis bzw. die Artefakt-Basis für den modellbasierten Systementwurf im Kontext des Entwicklungs-prozesses eingebetteter Systeme in den späteren Kapiteln stellen und auf die nachfolgend in der Arbeit mehrfach referenziert wird.

- **AUTOSAR**: Das Industrie-Konsortium *Automotive Open System Architecture* (AUTOSAR) [ASR09] ist ein internationaler Verbund von Automobilherstellern und -zulieferern mit dem Ziel, einen offenen Standard für Software-Architekturen speziell in Kraftfahrzeugen zu etablieren (siehe auch Kapitel 2.2.5.2 zur Betriebssystem-Plattform). Die Hauptidee des AUTOSAR-Standards ist dabei die klare Trennung der hardwareunabhängigen von den hardwareabhängigen Komponenten der Steuergerätesoftware. Dies gelingt durch die dedizierte und formale Beschreibung (Artefakte in XML-Format) jeder einzelnen Architektur und Funktion des eingebetteten Systems (Hardware und Software) inklusive ihrer Schnittstellen. Diese Beschreibung liegt in Form einer AUTOSAR-XML-Datei zusammen mit implementierter Funktion (C-Code) vor und wird als Software-Komponente bezeichnet. Neben den implementierungsbezogenen Standards gibt das AUTOSAR-Konsortium auch Entwicklungsrichtlinien für die modellbasierte Funktionsentwicklung vor. Die herausgegebenen ‚*Styleguides*' gelten für unterschiedliche Modellierungswerkzeuge und Modellierungssprachen.
- **ASAM**: Die *Association for Standardization of Automation and Measuring Systems* (ASAM) ist ein Verein, dessen Mitglieder sich im Wesentlichen aus internationalen Automobilbauern, deren Zulieferern und deren Dienstleistern zusammensetzen [ASAM09]. Ziel ist die Standardisierung von Schnittstellen, Protokollen und

Datenformaten für den Automobilbau mit Schwerpunkt auf Elektrik-/Elektronik-Systeme, um diese in Artefakten wie Software, Treiber und Hardware einzusetzen. ASAM definiert verschiedene Standards für die unterschiedlichsten Aufgaben: Das ASAM ODS (Open Data Service) definiert ein generisches Datenmodell für die Interpretation von Daten, Schnittstellen für das Modellmanagement, Datenablage, Datenzugriffe sowie für den Datenaustausch (Syntax und Struktur). Ein weiterer wesentlicher Standard für die Diagnose von eingebetteten Systemen ist der ODX-Standard (Open Diagnostic Data Exchange, ASAM-Standard MCD-2D). ODX steht für eine formale Beschreibungssprache im XML-Format. Als Norm ISO/DIS 22901-1 enthält er alle Informationen zu Anforderungen und zur Dokumentation, welche in der Fahrzeug- oder Steuergerätediagnose relevant sind. Dies ist z. B. die Bedatung des Werkstatttesters oder die konforme Softwarekonfiguration. Das ASAM-Konsortium arbeitet hierbei eng mit der ISO zusammen.

- **ISO**: Die *International Organization for Standardization* (ISO) [ISO17000] ist eine international operierende Vereinigung von Normungsorganisationen und erarbeitet internationale Normen in allen Bereichen (insbesondere der Prozessnormen wie ISO 9001, CMMI oder SPiCE) mit Ausnahme des Spezialfalls der Elektrik-/Elektronik-Systeme, für die die *Internationale Elektrotechnische Kommission* (IEC) zuständig ist, und mit Ausnahme der Telekommunikation, für die die *Internationale Fernmeldeunion* (ITU) zuständig ist. Gemeinsam bilden diese drei Organisationen die *World Standards Cooperation* (WSC).
- **HIS**: Die in der *Hersteller Initiative Software* (HIS) [RIF05] zusammengeschlossenen deutschen Automobilhersteller Audi, BMW, Daimler, Porsche und Volkswagen arbeiten auf einer Reihe von Gebieten in den Arbeitsgruppen ‚Standard Software', ‚Software Test', ‚Process Assessment', ‚Simulation and Tools' sowie ‚Flash Programming' zusammen. Kernziel der HIS ist die Vereinheitlichung von Anforderungen an Komponenten und Prozessen, um den Aufwand der Zulieferer zu reduzieren, der durch die Berücksichtigung unterschiedlicher Anforderungen der Automobilhersteller entsteht.
- **MAAB:** Das werkzeugspezifische Gremium *MathWorks Automotive Advisory Board* (MAAB) [MAAB09] wurde gegründet, um die Anforderungen von Anwendergruppen aus der Automobilindustrie in Bezug auf die Produkte des Software-Herstellers The MathWorks zu koordinieren. Derzeit beteiligen sich über 45 der größten Hersteller und Zulieferer aus der Automobilbranche am MAAB-Gremium und erarbeiten dort in jährlichen Abständen vor allem Richtlinien für den Steuerungs- und Regelungsentwurf, die Simulation und die Codegenerierung mit ML/SL/SF (Modellierung/Simulation) und Real-Time Workshop (Codegenerierung). Die aktuelle Version der MAAB-Modellierungsrichtlinien für die Anwendung von MATLAB, Simulink und Stateflow stellt eine Basis für die Qualitätsvorgaben und die Richtlinien für modellbasierte Projekte in der Automobilindustrie dar.
- **VDA:** Der Verband der Automobilindustrie (VDA) ist ein Interessenverband der deutschen Automobilhersteller und -zulieferer [VDA09]. Der Verband ist Gründungsmitglied in europäischen Organisationen, welche Standardisierungsaufgaben im Bereich der unternehmensübergreifenden Zusammenarbeit in der Automobilindustrie wahrnehmen. Der VDA entwickelt in Arbeitskreisen Richtlinien aus regulatorischen Anforderungen und veröffentlicht Leitfäden, welche

Empfehlungen für die Automobilindustrie enthalten und dort durch konkrete Anforderungen im Engineering-Prozess umgesetzt werden müssen.

3.4 Prüfmethoden

In den kollaborativen Prozessphasen Anforderungsdefinition, funktionaler und technischer Systementwurf, Modulspezifikation, Implementierung und Testspezifikation gelten im V-Modell unterschiedliche Entwicklungsrichtlinien bzw. Hersteller-Zulieferer-Konventionen. Meistens sind die Entwicklungsrichtlinien systematisch in Richtlinienkatalogen textuell dokumentiert. Ansonsten gelten mündliche Abstimmungen. Gelebte Konventionen sind evolutionär und nur aus den Artefakten heraus ersichtlich (Beispiel: Art der Namensgebung, Art der Farbwahl, Gliederung, Design-Struktur usw.). In den heute praktizierten Methoden zur Konformitätsprüfung für die Einhaltung von Konventionen und Entwicklungsrichtlinien werden unterschiedliche Prüfmethoden eingesetzt. Für die Durchführung von Code-Reviews sind bereits eine Reihe von Vorgehensweisen im wissenschaftlichen Kontext vorgeschlagen worden. In die Thematik führt [FAGA02] ein. Üblicherweise sind vom Qualitätspersonal durchgeführte Prüfmethoden unternehmensabhängig und nicht immer öffentlich. Einen Einblick gibt die NASA mit den ‚*Procedure for Developing and Implementing Software Quality Programs*' [NASA07]. Während für Code-Reviews bereits viele Automatisierungstechniken durch Werkzeuge verfügbar und Modell-Analyse-Werkzeuge langsam auf dem Vormarsch sind (siehe Kapitel 3.7.1), werden für alle anderen Artefakte in der Praxis noch manuelle Prüfmethoden durchgeführt.

Eine *Prüfmethode* beschreibt, wie ein Merkmalsprüfergebnis ermittelt wird (vgl. DIN) und basiert in der Regel auf einer Prüfspezifikation. Die heute in manueller Tätigkeit durchgeführten Methoden, um Konformität zu sichern, sind die Review-Methode, die Walk-Through-Methode sowie die Inspektions-Methode.

- ***Review-Methode*:** Der Entwickler nimmt das Artefakt manuell in Augenschein. Er untersucht alle sich im Artefakt befindlichen Elemente auf Einhaltung der Qualitätsvorgaben. Manuelle Vergleiche mit dem Artefakt gegen den Richtlinienkatalog oder gegen ein weiteres Artefakt müssen zur Sicherstellung der Konformität von ihm durch Sichtprüfung durchgeführt werden.
- ***Walk-Through-Methode*:** Der Entwickler nimmt zusammen mit einem Sachverständigen das Artefakt manuell in Augenschein. Sie untersuchen gemeinsam alle oder partielle sich im Artefakt befindlichen Elemente auf Einhaltung der Qualitätsvorgaben. Manuelle Vergleiche mit dem Artefakt gegen den Richtlinienkatalog oder gegen ein weiteres Artefakt müssen zur Sicherstellung der Konformität von beiden Personen durchgeführt werden. Die Dualität fördert die Aufdeckung von menschlichen Fehlern durch die Kontrolle einer zweiten Person. Diese Methode wird nach [HRH02] in der agilen Software-Entwicklung angewandt.
- ***Inspektions-Methode*:** Das Entwicklungsartefakt wird von einer dritten, nicht an der Entwicklung beteiligten Person (Prüfer) manuell in Augenschein genommen. Diese Person untersucht alle sich im Artefakt befindlichen Elemente auf Einhaltung der Qualitätsvorgaben und protokolliert Verstöße gegen die Richtlinienvorgaben. Manuelle Vergleiche mit dem Artefakt gegen den Richtlinienkatalog oder gegen ein weiteres Artefakt müssen zur Sicherstellung der Konformität vom Prüfer durchgeführt werden. Im iterativen Prozess geht das Artefakt zusammen mit dem Prüfprotokoll zur Nachbesserung an den Entwickler.

3.5 Einordnung der Begriffe verwandter Ansätze

Ausgewählte wissenschaftliche Arbeiten im Kontext der Entwicklung eingebetteter Systeme zur regelbasierten Konformitätsprüfung sollen in diesem Kapitel diskutiert und abgegrenzt werden, welche im Zusammenhang mit dem Lösungsweg stehen.

Die Dissertation von [MUTZ05] stellt eine durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich vor. Insbesondere sind hier folgende Aspekte für diese Arbeit relevant: Die Erstellung von Fahrzeugfunktionen findet gewöhnlich in Zusammenarbeit mit den Zulieferern statt (Kollaboration). Festgestellte Schwächen in den Anforderungsspezifikationen (Artefakte) haben zur Folge, dass es häufig zu Unstimmigkeiten zwischen Auftraggebern und Auftragnehmern kommt. Als Anforderungsspezifikationen verfolgt die Methodik ausführbare Modelle. Als Schwächen werden Unvollständigkeit, Inkonsistenz, Mehrdeutigkeit, Redundanz, Überspezifikation und Informationsvermischung genannt. Die Arbeit schlägt eine Methodik vor, die den Einsatz modellbasierter Techniken in den Vordergrund stellt. Gleichzeitig wird jedoch herausgefunden, dass die genannten Schwächen in der Kollaboration auch in der modellbasierten Entwicklungsmethodik – insbesondere betrachtet auf Zustandsautomaten – zutreffen. In der Arbeit wird weiter erkannt, dass unterschiedliche Modellnotationen und Werkzeuge allein für die Zustandsautomaten-Modellierung eingesetzt werden. Als Vorschlag wird erarbeitet, dass ein zentralisiertes und werkzeugunabhängiges Prüfverfahren, umgesetzt als Prüfwerkzeug, notwendig ist, um die Schwächen im Modellierungsprozess aufzudecken. Als werkzeugunabhängiges Datenformat wird pro Werkzeug ein XML-Export (durch Modelltransformation aus den Modellierungswerkzeugen) angewendet. Auf diesem Export prüft ein Checker-Programm aufgestellte Modellierungsrichtlinien. Die technische Erläuterung folgt im Kapitel 3.7. Die Dissertation bestätigt somit unsere Beobachtung im kollaborativen Prozessen: Kollaborative Artefakte, durch unterschiedliche Werkzeuge verschiedener Akteure entwickelt, sind durch Schwächen aus dem Modellierungsprozess und den kreativen Freiheiten der Werkzeuge geprägt. Eine werkzeugunabhängige Konformitätsprüfung (Prüfung der Anforderungen aus Richtlinien) soll die Schwächen in den Modellen aufdecken und zu einem durchgängigen Entwicklungsprozess beitragen. Da die Arbeit sich sehr stark auf Modellierungsrichtlinien konzentriert, wird nicht erkannt, dass auch andere kollaborative Artefakte als Kontextinformation im Bezug zum Modell stehen und die Grundlage zu einer Konformitätsaussage hinsichtlich der Richtlinienerfüllung stellen. Es wird eine werkzeugunabhängige aber keine werkzeugübergreifende Prüfung mit dem vorgestellten Prüfwerkzeug durchgeführt. Modellierungsrichtlinien werden in der Arbeit bzgl. der Exporte programmiert. Es können keine Prozessinformationen mit abgebildet oder zusätzlich mit abgeprüft werden. Schließlich überwindet der vorgestellte Ansatz zwar technische Barrieren, kann jedoch komplexere werkzeugübergreifende Prozessrichtlinien nicht abdecken und nicht ohne Weiteres auf andere Artefakt-Typen angewendet werden.

Prozesslogische Abhängigkeit von Artefakten wird in verwandten Ansätzen und in den Reifegradmodellen durch den Begriff ‚*Traceability*' (Rückverfolgbarkeit, Nachvollziehbarkeit) belegt. Dabei liegt der Ursprung dieser Anforderung in der Prozesswelt: Es existiert für Prozessphasen und deren Arbeitsergebnissen eine horizontale und eine vertikale Traceability im V-Modell, je nachdem ob ein nachfolgender Prozess durch andere Akteure nachvollziehbar sein soll oder ein Fehler aus den späten Testphasen in die Entwurfsphase zurückverfolgt werden soll. Beispielhaft erfordert ein CMMI-konformer Prozess [CMMI12] in seiner ‚*Specific Practice SP 1.4*' die Sicherstellung der Verfolgbarkeit von Anforderungen auf andere Arbeitsergebnisse, ohne Einschränkungen. Die Traceability ist Voraussetzung für die ‚*Specific Practice SP 1.5*', die das Thema Konsistenzprüfung zwischen verschiedenen

Artefakten behandelt. Andere verwandte wissenschaftliche Arbeiten greifen die nicht triviale Problematik dieser Anforderungen auf. Prozesslogische Abhängigkeit von Artefakten im modellbasierten Entwicklungsprozess eingebetteter Systeme wird beispielsweise in [ALT02; DIJK05] festgestellt. In [ALT02] wird versucht, bidirektionale Konsistenzbeziehungen von Artefakten nach [GW06] aus unterschiedlichen Prozessen und Werkzeugen stammend, durch die Metamodellierung abstrakt und formalisiert zu beschreiben sowie werkzeuggestützt durch den Einsatz einer ‚Cockpit-Anwendung' (Toolnet-Framework) abzuprüfen. Der Ansatz verfolgt primär jedoch das Abhängigkeitsmanagement in der Systementwicklung und nicht die Prüfung auf Erfüllung einer Anforderung aus Richtlinien. Dieser Ansatz tangiert nur insofern, dass die halb automatische Überwachung von Konsistenzbeziehungen und Propagation von Änderungen zwischen ML/SL/SF-Modellen einerseits und den in DOORS verwalteten Anforderungen andererseits (vgl. [MATE07+]) erfolgt. Der Ansatz ist pragmatisch nachvollziehbar, jedoch nicht vollständig zielführend. Die Verbindungen von übergreifenden Artefaktinformationen obliegen dem Akteur (Menschen) und sind somit generell fehleranfällig. Ferner ist nur systematisch prüfbar, ob vitale Konsistenzbeziehungen noch existieren, nicht ob inhaltliche Datentypdefinitionen tatsächlich mit den in der Anforderungsspezifikation geforderten Datentypdefinitionen übereinstimmen. Die Anforderung zum Nachweis einer Verbindung zur Traceability ist zwar geschaffen, im Vergleich jedoch wird keine werkzeugübergreifende Konformitätsprüfung inhaltlicher Kriterien (vgl. die Schwächen von [MUTZ05]) möglich. Trotzdem begründen auch die Arbeiten aus Toolnet zum Abhängigkeitsmanagement unsere Beobachtungen: Kollaborative Artefakte, durch unterschiedliche Werkzeuge verschiedener Akteure entstanden, stehen durch ihren Kontext logisch in Abhängigkeit. Eine werkzeugunabhängige Konformitäts-prüfung (Prüfung der Anforderungen aus Richtlinien) muss inhaltliche Bezüge abstrakt erfassen, abprüfen können und zu einem durchgängigen Entwicklungsprozess beitragen. In der Praxis werden solche Bezüge heute durch inhaltliche Vermerke, Verlinkung und durch zusätzliche Informationen (Referenznummern der Artefakt-Inhalte) manifestiert. Die inhaltliche Überprüfung der Traceability-Anforderung erfolgt heute oftmals durch die in Kapitel 3.4 genannten Prüfmethoden bzw. durch eng verzahnte Werkzeugkopplungen, welche jedoch einen Idealfall darstellen (Heterogenität der IT-Infrastruktur wird vernachlässigt).

Dass Informationen aus dem Entwicklungskontext überhaupt zur Prüfung notwendig werden, erkennt beispielsweise die Arbeit von [SCHIN09] ‚*The Rhapsody UML Verification Environment*'. Hierbei behandelt die Arbeit wie bei [MUTZ05] auch Zustandsautomaten, jedoch speziell die der UML. Es wird ein Modell-Checker beschrieben (VIS, Verification Interacting with Synthesis) welcher das Modell (FSM, Finite State Machine) in Zusammenhang mit einer formalisierten Anforderung (CTL, Computation Tree Logic) auf einem abstrahierten, durch mehrere Transformationen (auf Basis XMI-Export) erzeugten Artefakts überprüft. Die Überprüfung geht jedoch in die Ausführung und temporale Logiken über und versucht, den Nachweis der Funktionalität aus der Anforderung zu erbringen. Dies soll in dieser Arbeit abgegrenzt werden, da die Prüfung sich hierbei auf die Validierung von Anforderungen bezieht und nicht wie in dieser Arbeit verfolgt, werkzeugübergreifende Prozessrichtlinien versucht zu prüfen. Trotzdem begründen die Arbeiten zum *Rhapsody UML Verification Environment* unsere Beobachtungen: Kollaborative Artefakte, durch unterschiedliche Werkzeuge verschiedener Akteure entstanden, stehen auch inhaltlich in Abhängigkeit und müssen bzgl. einer Überprüfung mit einbezogen werden. Die Arbeit führt uns jedoch zu einer Aussage und zu einer wichtigen Abgrenzung: Ein dynamisches Systemverhalten wird durch die regelbasierte Konformitätsprüfung kollaborativer artefakte

nicht versucht nachzuweisen. Es geht hierbei nur um formale Entwurfs- und Designkriterien eines Artefakts, was auch kein Funktionsmodell ist, sondern eine Excel-Tabelle sein kann.

Dies führt uns auch zur Abgrenzung des in diesem Zusammenhang belegten Begriffs ‚*Model-Checking*'. Mithilfe der Verifikationstechnik Model-Checking ist es durch verschiedene Ansätze verwandter Arbeiten möglich, die Modellierung eines Systems auf Basis eines oder mehrerer Modelle, beispielsweise ein das System beschreibende Programm oder ein Zustandsautomat, daraufhin zu überprüfen (Checking), ob sie die Eigenschaften aus einer zugehörigen Spezifikation (Beschreibung der Eigenschaften des Modells) beispielsweise durch logische Formeln erfüllt. Das Mittel der Formalisierung in Logik-Ausdrücke ist daher eine Tangente zu unserem Ansatz. In solchen Fällen hat das Model-Checking zum Zweck, ein Ergebnisnachweis durch ein möglichst automatisiertes Verfahren zu liefern, ob eine Systemanforderung durch das Modell (bzw. die Modelle) erfüllt ist. Dies ist die Abgrenzung, da unsere Zielstellung die formale Anforderung aus einer Prozessrichtlinie durch geforderte Artefakt-Kriterien nachweisen soll (die Konformität) und neben Modellen auch für Dokumente oder Datenbanken anwendbar sein soll. Modelle sollen hinsichtlich von Modellierungsrichtlinien statisch prüfbar sein, ohne jedoch eine Ausführungsumgebung für ein Artefakt zu fordern.

Natürlich sprachliche Anforderungen (Nebenbedingungen oder Richtlinien) als Regel zu formalisieren, ist in der Informatik ein bekannter Ansatz. Die Arbeiten von [GROS00; STRA09] verfolgen beispielsweise eine regelbasierte Konformitätsprüfung im Bereich der Business-to-Business Anwendungen (B2B), wobei [GROS00] Logikprogramme für Geschäftsabläufe auf XML-basierten Artefakten vorstellt. Im eingrenzenden Vergleich zum Kontext dieser Arbeit widmen sich die Arbeiten von [DINE08; MARP08] der Formalisierung von Richtlinien durch Logikausdrücke. Insbesondere werden diese verwendet, um Nebenbedingungen bzw. Konsistenzverweise innerhalb des Modellierungsprozesses bzw. Modellierungswerkzeugs sicherzustellen. Hierzu verwandte Arbeiten unterscheiden sich meistens in der Art der Anwendung, entweder durch reine Annotation von Modellen oder aber durch Prüfwerkzeuge auf Modellebene. Im Vergleich werden kollaborative Aspekte und verschiedenartige (heterogene) Artefakte nicht beleuchtet.

Die Abstraktion von Daten von der realen Welt ist in der Informatik ein lange praktiziertes und bekanntes Verfahren. Bereits bei der Datenmodellierung von Datenbanksystemen, nach Chen-Notation mittels Entity-Relationship-Diagramm (ERD) oder bei der Metamodellierung nach dem MDA-Paradigma wird dies verfolgt. Bereits erwähnt wurden solche Abstraktionen im RIF- und AUTOSAR-Standard [BESS06; ASR09] sowie bei MDA nach [PETRA06]. Eine Ausführung in diesem sehr breiten Feld würde zu weit ausschweifen. Es sei jedoch auf ähnliche Arbeiten in der modellbasierten Entwicklung mit ML/SL/SF nach [NEEM01] verwiesen, worin die Abstraktion von einem proprietären Datenmodell in ein semantisches Datenmodell (Metamodell) vorgestellt wird. Es wird beschrieben, wie direkt von Funktionsmodellen zu abstrahieren ist und diese als logisches/semantisches Modell durch Metamodellierung zu beschreiben sind, um dieses in anderen Programmen für die weitere Verarbeitung zu instanziieren. Das Kapitel ist auf ergänzende Aspekte des gewählten Ansatzes eingegangen, wie die prozesslogische Abhängigkeit von Artefakten, die Formalisierung von Bedingungen durch Logikausdrücke und schließlich die Bildung eines abstrakten (logischen) Artefakt-Modells. Um jedoch die Zielstellung der Arbeit weiter zu konkretisieren und die Einzigartigkeit und Verschiedenheit zu exsistierenden Ansätzen aufzuzeigen, werden in den nachfolgenden Kapiteln noch mal aktuell nicht lösbare Probleme und die Eingeschränktheit verfügbarere Werkzeuge beschrieben.

3.6 Konformitätsprobleme und Kriterien

Jede Prüfmethode führt zu einem Ergebnis. Ein Konformitätsproblem liegt immer dann vor, wenn in einem Review oder während eines Audits/Assessments festgestellt wird, dass die Anforderungen oder die Ziele aus den Prozessrichtlinien nicht erfüllt worden sind. Meistens wird daraufhin versucht, den Fehler in dem Prozess zu entdecken bzw. die betroffenen Artefakte zu identifizieren. An der Stelle sei ein sehr typisches Beispiel aus der Praxis modellbasierter Systementwicklung eingebetteter Systeme gegeben:

Es existiert die Prozessrichtlinie zur konformen Vergabe der Bezeichner von technischen Produktfunktionen, also eine mit verschiedenen Prozessen abgestimmte Namenskonvention. Aus einem Architekturmodell sowie einem Verhaltensmodell wird in nachgelagerten Schritten aus den Modellen Software für das eingebettete System erzeugt. Code-Compiler verbieten einige spezifische Namensausprägungen, die jedoch in dem vorgelagerten Modellierungsprozess keine negative Wirkung hätten. Im Systementwurf liegen nun jedoch Architekturmodelle (Designs) in einer objektorientierten Modellierungssprache, z. B. in der UML vor, Funktionsmodelle aus dem Simulationsprozess hingegen liegen z. B. als Simulink-Modell vor. Die Namenskonvention selbst liegt in Form von Bezeichnern in einer Microsoft-Excel-Tabelle und in den Köpfen der Entwickler bzgl. des syntaktischen Aufbaus der Bezeichner vor. Es ist evident, dass in diesem (oft auftretenden) Beispiel, die Prüfung auf Namenskonformität durch den Abgleich heterogener Artefakte ein aufwendiger manueller Schritt ist und zugleich von einem Modell-Checker/Style-Checker nicht alleine lösbar wäre. Für alle betroffenen Artefakte einen separaten Artefakt-Checker zur Automatisierung der Reviews zur Verfügung zu stellen, der in der jeweils werkzeugspezifischen Ausprägung die Namenskonformität prüft und dabei stets dedizierte Konventionen einzelner Artefakte berücksichtigt, ist kein gangbarer Lösungsweg.

Für Prüfwerkzeuge (Checker) müssen diverse Kriterien eines regelbasierten Verfahrens zur Konformitätsprüfung von Artefakten aufgestellt werden, die verschiedene Aspekte beleuchten. Zum einen sind das *Artefakt* vom Typ her sowie das *Werkzeug* relevant, für das ein Checker entwickelt wurde. Das vom Werkzeug verwendete *Prüfverfahren* ist hinsichtlich der *Art* und des *Automatisierungsgrades* zu bewerten. Des Weiteren ist zu bewerten, welche *Norm* bzw. Richtlinien damit prüfbar sind und in welcher Form der Benutzer eine *Auswertung* als Konformitätsnachweis erhält. Schließlich ist der wichtigste Aspekt, ob kollaborative Artefakte, also ein *werkzeugübergreifender* Prüfraum, berücksichtigt sind. Zur Lösung von lokalen sowie prozessübergreifenden Konformitätsproblemen werden daher folgende Kriterien an verfügbare Checker angesetzt, wie in der Abbildung 10 gezeigt.

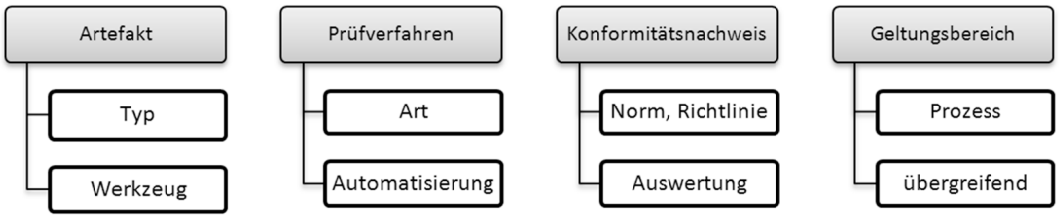


Abbildung 10: Kriterien für regelbasierte Konformitätsprüfung

Das nachfolgende Kapitel stellt den heutigen Stand der Technik zur Lösung von Konformitätsproblemen an Artefakten der modellbasierten Entwicklung eingebetteter Systeme vor und bewertet eine repräsentative Prüfwerkzeug-Auswahl anhand der Kriterien.

3.7 Stand der Technik

Wie bereits in der Einführung geschildert, steht die Konformität von Arbeitsergebnissen in unmittelbarem Zusammenhang mit konformen Arbeitsprozessen. Ausgangspunkt für die Arbeit sind daher die in den Prozessen durch Akteure bereits etablierten Verfahren zur manuellen Sicherstellung von konformen Arbeitsergebnissen, verwandte wissenschaftliche Arbeiten zum Thema Konformitätsprüfung im Prozess sowie der Stand der Technik bezogen auf die bereits eingesetzten Prüfwerkzeuge und geforderte Richtlinien. Schließlich müssen der Stand der Technik und praktizierte Richtlinienprüfungen analysiert werden.

Für werkzeuggestützte, statische Analysen existieren für vereinzelte Prozesse seit einigen Jahren Prüfwerkzeuge am Markt. Sie sind bekannt unter den Begriffen wie z. B. Source-Code-Analyzer für Softwareanalyse oder Modell-Checker bzw. Style-Checker für die Modellanalyse. Sie sind jeweils beschränkt auf eine hoch spezielle Aufgabenstellung im Entwicklungsprozess. Kollaborative Wechselwirkungsaspekte, logische Zusammenhänge zwischen den Artefakten und verschiedenartige Dateiformate einer heterogenen IT-Infrastruktur werden durch die verfügbaren Ansätze der statischen Analyse nicht adressiert. Die werkzeuggestützten Verfahren statischer Analyse erfassen somit bisher keine übergreifenden Entwicklungsrichtlinien. Hinsichtlich kollaborativer Umgebungen sind daher die heute eingesetzten Mittel zur Sicherstellung der Konformität manuell durch Entwicklungsingenieure durchgeführte Reviews, stichprobenartige Inspektionen sowie die Ausfüllung vordefinierter *Formulare* zur einheitlichen Erfassung von durchgeführten *Stichproben*.

Statische Analyse **(3.1)**
„Allgemein versteht man unter der *statischen Analyse* die Untersuchung des statischen Aufbaus eines Prüflings auf die Erfüllung vorgegebener Kriterien (Anforderungen).“

In der Informatik wird die statische Analyse meist durch computerausführbare Algorithmen (Regeln) auf Artefakten ausgeführt. Beispielsweise checkt ein Modell-Checker programmierte Regeln auf dem Modell, indem er sie innerhalb der Programmumgebung lädt und ausführt. Hier spricht man von *regelbasierter Analyse*.

Regelbasierte Analyse, Regelbasierte Prüfung **(3.2)**
„Unter der *regelbasierten Analyse* (bzw. Prüfung) wird die werkzeuggestützte Untersuchung des statischen Aufbaus eines Prüflings (Artefakts) auf die Erfüllung vorgegebener Kriterien (Anforderungen) verstanden. Die Untersuchung basiert dabei auf programmierten Prüf-algorithmen und ihrer Ausführung durch ein Programm/Prüfwerkzeug (Software).“

Um mittels regelbasierter Analyse, z. B. durch ein Prüfwerkzeug, die Konformität eines Artefakts nachzuweisen, spricht man von *regelbasierter Konformitätsprüfung*. Diese wird in dem Kapitel 4 genauer spezifiziert.

In der Entwicklung eingebetteter Systeme wird der modellbasierte Entwicklungsansatz in dem Bereich des Systementwurfs heutzutage zunehmend eingesetzt [ADL07]. Somit ist die Modellierung im Systementwurf ein zunehmender Trend, jedoch abhängig von der Unternehmensgröße, da in kleineren Unternehmen noch viel handgeschriebener Code programmiert wird.

Wird die statische Analyse auf den modellbasierten Prozess übertragen, sind die Prüflinge die Artefakte (z. B. das Modell) im modellbasierten Entwicklungsprozess und die

vorgegebenen Kriterien sind die Richtliniendokumente für die Erstellung der Artefakte. Betrachtet man Artefakte genauer, beinhaltet der Systementwurf Modelle, die Transformation zwischen Modellen untereinander (z. B. zwischen Implementierungs- und physikalischem Modell) sowie die automatische Generierung von Programmcode von Modellen (Embedded Software). Dies ist nach dem MDA-Paradigma letztendlich auch eine Modelltransformation. Demnach ist die statische Analyse auf Code-Ebene im modellbasierten Entwicklungsansatz nicht mehr ausreichend. Auch Modelle müssen früh einer statischen Analyse unterzogen werden. Folglich sind auch Anforderungsspezifikationen einer statischen Analyse zu unterziehen, da Modelle wiederum aus Anforderungsspezifikationen entwickelt werden. Zuletzt ist auch die Analyse von Testspezifikationen im modellbasierten Entwurf bedeutend, da dieses im modellbasierten Entwicklungsprozess aus einem Modell abgeleitet oder auf einem Modell durchgeführt wird, wie beim Model-based Testing [ZAN08].

Nachfolgend werden im industriellen Kontext praktizierte Prüfmethoden und am Markt verfügbare Prüfwerkzeuge vorgestellt und für einen Vergleich ausgesucht.

3.7.1 Richtlinien und Prüfwerkzeuge

Um die im Kapitel 3.4 genannten Prüfmethoden entstehenden manuellen Prüfaufwände zur Konformitätsprüfung in Teilprozessen, wie bei Modellierung und in der Softwareentwicklung, möglichst zu minimieren, werden im Kontext der Entwicklung eingebetteter Systeme werkzeuggestützte Prüfverfahren eingesetzt. Beispielsweise ist im Rahmen eines Softwaretestverfahrens die statische Code-Analyse üblich. Für dieses Verfahren wird der Quellcode, also das Artefakt, benötigt. Die Analyse erfolgt entweder durch eine manuell durchgeführte Code-Inspektion oder automatisiert durch ein regelbasiertes Prüfwerkzeug. Man spricht in diesem Falle von einer statischen Analyse eines Artefakts, da die zu testende Embedded Software in Form von Algorithmen und Daten in ihrer Formulierung und Beschaffenheit (statisch) als Artefakt dem Prüfer (oder Werkzeug) vorliegt.

Ein weiteres statisches Analyseverfahren sind die sogenannten Style-Checker Programme. Es handelt sich hierbei um werkzeuggestützte Prüfverfahren der normierten Programmierung zur Erkennung von Code- bzw. Modell-Mustern. Hier wird ein Konformitätsabgleich zwischen einem Muster (Vorlage) und dem Artefakt-Fragment durchgeführt, der primär die Lesbarkeit des Artefakts vereinfacht. Analyse-Werkzeuge prüfen Programmstrukturen und Datentypen beispielsweise auf:

- ***Formale Eigenschaften***:
  Einhaltung von Entwurf, Stil, Art und Menge
- ***Strukturkomplexität des Artefakts:***
  Einhaltung von Strukturierungsregeln, Tiefe
- ***Strukturfehler bzw. –anomalie:***
  Vorhandene, aber nicht benutzte Operationen; benutzte, aber nicht vorhandene Operationen; Redundanzen
- ***Fehler oder Anomalien in den Daten***:
  Fehlende Variablen-Deklaration, nicht benutzte Variablen, Benutzung nicht initialisierter Bezeichner
- ***Parametereigenschaften***:
  Belegung und Verwendung von Variablen

Für die Werkzeuge im Anforderungsmanagement, wie dem ausgewählten Werkzeug MS Word und dem Werkzeug DOORS, wird im modellbasierten Systementwurf eingebetteter Systeme das Hilfstool DESIRe® zur Auffindung von nicht erwünschten Worten oder Begriffen (Weak Words) in Anforderungsdokumenten verwendet [HOOD09]. Dies ist insbesondere nützlich, da im Anforderungsmanagement viel Wiederverwendung stattfindet, jedoch übersehene Worte oder Namen vertraulicher Informationen in alten Textfragmenten schnell übersehen und in die neue Anforderungsdokumentation mit übernommen werden und somit bei ungewollter Wiederverwendung einen erheblichen Schaden verursachen würden. Generelle Anforderungsrichtlinien und komplexe Strukturprüfungen (Gliederung, Referenzen usw.) sind nicht implementiert bzw. prüfbar.

Im modellbasierten Systementwurf eingebetteter Systeme haben Entwurfsrichtlinien eine höhere Verbreitung als in dem Anforderungsmanagement. Bekannte Modellierungsrichtlinien sind aus der MISRA-Normung stammende ‚*Generic modelling design and style guidelines*' [MISRA09], herstellerbezogen nach ‚*MISRA-Konformität von ASCET-Autocode*' [MACK07] oder dem ‚*Modelling design and style guidelines for the application of Simulink and Stateflow*' [MISRA09+], nach dem englischen MAAB-Standard ‚*Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*' [MAAB09] bzw. speziell für die Codegenerierung ‚*dSPACE - Modeling Guidelines for MATLAB, Simulink, Stateflow, and TargetLink*' [STUE09] oder speziell auf den Anwendungsfall bezogen die japanischen ‚*Plant Modeling Guidelines using MATLAB*' [OKA09].

Modellierungsrichtlinien werden teils frei und teils kommerziell angeboten, wie z. B. durch das Online-Richtlinienportal *e-Guidelines* für die konforme Modellierung [EGUID09]. Die Richtlinien entstanden im Kontext des BMBF-Forschungsprojekts IMMOS [IMMOS06] und wurden später weiterentwickelt durch Forschungsarbeiten [MATE07; MATE07+]. Vereinzelt existieren Prüfwerkzeuge am Markt. Eine Auswahl ist z. B. der *Model Advisor*® des Herstellers The MathWorks [MODA09], *mint*® des Herstellers Ricardo [MINT09], *Model Examiner* des Herstellers MES [MODE09] oder der *Embedded Validator* des Herstellers BTC [EMBV09]. Für die Werkzeuge ML/SL/SF sind die genannten Werkzeuge kürzlich auf den Markt gekommen, ausgenommen *mint,* welches bereits länger verfügbar ist.

Die Vielfalt an regelbasierten Prüfwerkzeugen bekräftigt den Bedarf von Richtlinienprüfung neben der Code-Analyse auch auf Modellebene. Für weitere Modellierungswerkzeuge, wie z. B. *SCADE*® des Herstellers Esterel [SCAD09] oder *ASCET-MD*® des Herstellers ETAS [ASCT09] fehlen Modell-Checker am Markt.

In [MUTZ05] werden Richtlinien speziell für die modellbasierte Analyse und Metriken-basierte Bewertung zustandsbasierter Softwaresysteme (Zustandsdiagramme) vorgestellt, welche jedoch keine bestimmte Norm umsetzen, sondern auf Erfahrungswerten im industriellen Kontext beruhen. Wie bereits erwähnt, ist ein in [MUTZ05] vorgestelltes Prüfwerkzeug *Regel-Checker* in der Lage, über einen XML-basierten Modell-Export der Zustandsdiagramme als Artefakte aus den Werkzeugen Rhapsody® [RATA09], ASCET® [ASCT09] sowie ML/SL/SF [MLSL09] zu importieren und hinsichtlich ausprogrammierter Regeln in Java zu prüfen.

Einen ähnlichen Ansatz verfolgt auch ‚*The Rhapsody UML Verification Environment*' aus Arbeiten in [SCHIN09]. Hier wird XML-basierte Modelltransformation angewendet, um schließlich auf einem transformierten Artefakt, welches zusätzlich durch Informationen aus der Umgebung (den Anforderungen) angereichert, abgeprüft wird. Hierbei stehen allerdings keine Modellierungsrichtlinien im Vordergrund.

Kommerzielle Prüfwerkzeuge für Zustandsdiagramme sind außerdem im Embedded-Bereich der *Statemate Design Checker*® des Herstellers Quality Park [STAT09] oder der *Sate Mate Model Checker*® des Herstellers BTC [EMBV09] für das zustandsbasierte Modellierungswerkzeug Statemate, [RATA09].

Im industriellen Kontext eingebetteter Entwicklung sind für den objektorientierten Entwurf noch keine industrietypischen Entwicklungsrichtlinien zu beobachten. Dies hat vermutlich den Hintergrund, dass die objektorientierte Modellierung bisher eher als visuelles Beschreibungsmittel, als zur systematischen Architekturmodellierung oder Embedded-Code Generierung im Automotive-Kontext verwendet wurde. Der Wandel hin zum Einsatz objektorientierter Programmiersprachen für das Embedded-Umfeld wird jedoch zu einer strategischeren Anwendung und somit auch zu abgestimmten Konventionen führen.

Für die Modulspezifikation mittels objektorientierter Modellierung mit der UML [UML22] schlägt die OMG ‚*Well-Formedness Rules*' vor. Design Guidelines und Erfahrungsberichte in MDD/MDA werden in [SOMM07] berichtet. Richtlinien für die Modellierung werden in ‚*Description and Implementation of a UML Style Guide*' vorgeschlagen [HIND09].

Schließlich ist eine Überprüfung des modellbasierten Entwurfs in ‚*Verification of Good Design Style of UML Models*' [HNAT07] erläutert. Üblicherweise werden Richtlinien und Konsistenzchecks durch die Modellierungswerkzeuge selbst berücksichtigt und sind dadurch teils auch werkzeugintern verfügbar. Weitere integrierte Prüfungen, wie z. B. einen *Spell-Checker,* bietet der Enterprise Architect [SPAX09] integriert an. Innerhalb der UML-Modellierungswerkzeuge existieren verschiedene Modell-Notationen (Diagramme), welche nur logisch miteinander verknüpft sind (eine andere Sicht auf das Modell darstellen). Wenn von übergreifender Konsistenzsicherung gesprochen wird, ist meist genau die Überprüfung der ‚logischen Konsistenz' zwischen den Modellen gemeint. Auch wenn verschiedene Modelle dabei untersucht werden, handelt es sich immer noch um die Prüfung eines Artefakts eines Modellierungswerkzeuges. Ein werkzeuggestütztes Prüf-Verfahren nennt [ALEX07] in ‚*UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models*'. Im wissenschaftlichen Kontext existieren sehr viele Arbeiten zur Prüfung von UML-Modellen, besonders von Zustandsdiagrammen, wie in [PAPM01; KNAPP07] vorgestellt wird. Diese sind jedoch meist domänenunspezifisch und adressieren Konformität bzgl. des OMG MOF-Standards. Andere, in den Modellierungswerkzeugen integrierte Prüfmodule basieren auf der Auswertung von OCL Richtlinien (vgl. [CABO09]) oder annotieren Konsistenzsicherungen in der OCL bereits während der UML-Spezifikation, wie in [USE09]. Zur Wahrung der Konsistenz und Vollständigkeit industrieller UML Modelle werden in [LANG04] Vorschläge gemacht und weitere Untersuchungen zu dem Thema referenziert. In [REISS02] werden domänenunspezifische Kriterien und Richtlinien für die Bewertung objektorientierter Entwürfe vorgestellt und weiterführende Literatur genannt. *UMLtoCSP* [UMLT09] ist ein externes Prüfwerkzeug für die objektorientierte Modellierung mit der UML auf Basis von OCL-Regeln und ist beschrieben in [CABO08].

In der Implementierungsphase ist die Konformität zum AUTOSAR-Standard sowie auf Code-Ebene zu dem MISRA-Standard bedeutend. Die Einhaltung ist aber nicht nur auf Code-Ebene gefordert: Vom AUTOSAR-Konsortium veröffentlichte Dokumentationen wie ‚*Applying Simulink to AUTOSAR*' oder ‚*Applying ASCET to AUTOSAR*' beschreiben die konforme Modellierung in ASCET-MD oder ML/SL/SF [APSA06; APPA06]. Des Weiteren existieren allgemeine Programmierrichtlinien wie die ‚*Coding Rules for C++'* [CODE04]. Ein reiches Angebot an Code-Analyse-Werkzeugen ist für Embedded-Anwendungen am Markt zu beobachten: *PC-lint for C/C++* [PCLi09] oder *CodeCheck C/C++* [CODE04],

QA-C/MISRA [QACM09], *Cosmic Software MISRA Checker* [COSM09], *LDRA Tool Suite* [LDRA09], *Flawfinder* [FLAW09] und *Rough Auditing Tool for Security* [RATS09] sind gängige Werkzeuge für die Codeanalyse (ohne den Anspruch auf Vollständigkeit). Diese Werkzeuge sollen (C/C++) Code optimieren, statisch auf Fehler prüfen und Performance analysieren. Es ist abzusehen, dass die modellbasierte Entwicklung den handgeschriebenen Code ersetzen wird, da Code-Generatoren und Modell-Compiler wesentlich effizienteren (C/C++) Code generieren können.

Für die Testspezifikation mit der Klassifikationsbaummethode mit CTE/ES des Herstellers Razorcat [CTES09] bzw. CTE/XL [CTEX09] sind bisher auch noch keine Prüfwerkzeuge im praktischen Industrieeinsatz identifiziert worden. Hier scheint sich eine Lücke aufzutun.

Diskussion:

Bei der Betrachtung der technischen Umsetzungen einer regelbasierter Prüfung werden mehrere Auffälligkeiten und auch Lücken offensichtlich. Auffällig ist, dass die regelbasierten Prüfungen, wenn diese vorhanden sind, als Prüfling immer nur einen Artefakt-Typ anwenden. Die Verfahren gehen also immer davon aus, dass alle benötigten Informationen, aus denen Schlüsse gezogen werden können, in dem Artefakt selbst vorliegen. Die Verfahren berücksichtigen nicht, falls Informationen auf andere Artefakte verstreut sind. Des Weiteren wird nicht berücksichtigt, dass auch eigene Semantiken der Prozesswelt sowie die erkannte, prozesslogische Abhängigkeit der Artefakte als Information vorliegen und prüfbar sein müssen. Wie auch ein spezielles Modell erfüllt daher auch ein Checker immer nur einen bestimmten Zweck. Dieser ist auch nicht anzuzweifeln, da hoch spezialisierte Prüfungen stets eine Berechtigung im Engineering-Prozess haben. Im Gegenteil ist hier ein Evolutionsschritt notwendig, sodass Prozessinformationen, logische Abhängigkeiten sowie die Heterogenität in der Beschaffenheit der Artefakte bisher keine Berücksichtigung finden. Dies resultiert in den im Kapitel 3.6 geschilderten Konformitäts-problemen und macht ungünstige Prüfmethoden erforderlich. Auch die Lücken sind offensichtlich: Da es mitunter sehr aufwendig ist, für jedes einzelne Werkzeug ein dediziertes Prüfverfahren im Engineering-Prozess einzusetzen und gleichzeitig auch die Richtlinienpflege für alle diese auftretenden Artefakt-Typen zu gewährleisten, muss eine regelbasierte Prüfung hinsichtlich *kollaborativer Artefakte* erforscht werden. Dies ist nicht nur eine Erweiterung zu bekannten Ansätzen, sondern schafft eine Neuerung bzgl. der Möglichkeit, Schlüsse auf Grundlage von Informationen aus verschiedensten Informations-quellen zu ziehen. Gerade die Abstraktion von Werkzeugen und die Integration von Prozessinformationen vervollständigt einen Konformitätsnachweis im Engineering-Prozess. Ein Beispiel ist, dass ein Prüfverfahren zur Analyse von Unterschieden zweier Modellversionen eine sehr komplizierte Differenzanalyse erforderlich macht, wobei bei Miteinbeziehung eines weiteren Artefaktes (tabellarische Änderungshistorie, welche sowieso im Prozess begleitend als Journal mitgeführt wird) in diesem bereits die Differenzen dokumentiert und mit dem Modell konsistent sein müssen. Wir erkennen daher die Zweckmäßigkeit zur Erweiterung der regelbasierten Prüfung.

Nachfolgende Tabelle 13 fasst noch mal den Stand der Technik mit einem Vergleich von Prüfwerkzeugen pro Phase anhand der im Kapitel 3.6 eingeführten Kriterien zusammen.

Tabelle 13: Vergleich ausgewählter Prüfwerkzeuge

Prüf-werkzeug	*Prüfkriterien*								
	Artefakt		*Prüfverfahren*		*Konformitätsnachweis*		*Geltungsbereich*		
	Typ	*Werkzeug*	*Art*	*Automatisierung*	*Norm, Richtlinie*	*Auswertung*	*Prozess*	*Übergreifend*	
DESIRe [HOOD09]	-externes Programm (Text oder Datenbank)	- MS Word, DOORS	- statische Analyse (Weak-Words-Check) durch Wortvergleich (interner Algorithmus)	- semi-automatisiert, keine Regelentwicklung	- benutzerspezifisch, Konsistenz von Wortlisten	- Meldung an den Prüfer (Nachricht)	- Anforderungsdefinition	- Kombination mehrerer Artefakte nicht möglich	
Model Advisor [MODA09]	-integriertes Programm (Modell)	- Simulink, Stateflow	- statische Analyse durch Regelausführung (M-Skript)	- voll-automatisiert, Implementierung anwenderspezifischer Regeln	- benutzerspezifisch, sowie DO-178B, IEC 61508, MAAB	- Prüfbericht (tabellarisch als HTML-Report), verbindet Meldungen direkt mit dem Modell	- signalflussorientierte und zustandsbasierte Modellierung	- nur Prüfung verwaister Referenzen zu DOORS (Traceability Ceck)	
Model Examiner [MODE09]	-externes Programm (Modell)	- Targetlink, Simulink	- statische Analyse durch Regelausführung (M-Skript)	- voll-automatisiert, Implementierung anwenderspezifischer Regeln	- benutzerspezifisch, sowie MISRA/TargetLink, dSPACE TargetLink, MAAB	- Prüfbericht (tabellarisch als HTML-Report)	- signalflussorientierte und zustandsbasierte Modellierung	- Kombination mehrerer Artefakte nicht möglich	
Regel Checker [MUTZ05]	-externes Programm (Zustandsdiagramme über XML-Datei)	- ASCET-SD, Rhapsody, Stateflow	- statische Analyse durch Regelausführung (Java oder OCL)	- semi-automatisiert, Implementierung anwenderspezifischer Regeln	- benutzerspezifisch	- Prüfbericht (grafisch, generierter Fehlerbaum)	- zustandsbasierte Modellierung	- mehrere Artefakte nur isoliert prüfbar, keine Kombination	
UMLtoCSP [UMLT09]	-externes Programm (Modell, XMI oder Ecore Datei)	- Modellierungswerkzeug, UML (nur Klassen-diagramme)	- statische Analyse durch Regelausführung (OCL)	- semi-automatisiert, Implementierung anwenderspezifischer Regeln	- benutzerspezifisch	- Prüfbericht (grafisch, Objekt-diagramm)	- objektorientierte Modellierung	- Kombination mehrerer Artefakte nicht möglich	
QA-C/MISRA [QACM09]	-externes Programm (Code, Datei)	- Softwaretools mit ANSI-C/C++ Code	- statische Codeanalyse durch Regelausführung (Tool-intern)	- voll-automatisiert, konfigurierbare Regeln	- benutzerspezifisch, sowie CMM, ISO9003/EN29003, ISO 9126, IEC 61508, DO-178B, Def Stan 00-55	- Prüfbericht, verbindet Meldungen direkt mit dem Quellcode	- Implementierung	- mehrere Artefakte nur isoliert prüfbar, keine Kombination	

3.8 Ausgewählte Artefakt-Typen

Die ausgewählten Entwicklungsumgebungen sowie die zugehörigen Prüfwerkzeuge wurden im Kontext der Arbeiten in (Veröffentlichungen, Nr. 28) analysiert. Typischerweise bieten die von den Prüfwerkzeugen angebotenen Modellierungsrichtlinien nur ein Basis-Angebot aus Erfahrungswerten. Sie werden meistens unternehmensspezifisch auf die konkreten Werkzeugketten und Anforderungen der Prozesse angepasst, so auch [MUTZ05; MUTZ05+; CON05+; OKA09].

In den folgenden Unterkapiteln findet sich eine Klassifizierung typischer im modellbasierten Entwicklungsprozess eingesetzter Werkzeuge und derer Artefakte. Es werden jeweils für die einzelnen Phasen gängige Software-Werkzeuge analysiert sowie die damit entwickelten Artefakte genauer vorgestellt und für diese Arbeit ausgewählt.

3.8.1 Artefakte der Anforderungsdefinition

Das Anforderungsmanagement ist an sich ein breites Forschungs- und Entwicklungsfeld. In dieser Arbeit soll daher nur eine Auswahl an industriell gängigen IT-Werkzeugen zur Anforderungsdefinition und zur technischen Funktionsspezifikation im Bereich der eingebetteten Systementwicklung sowie deren Artefakte vorgenommen werden. Funktionen eingebetteter Systeme werden üblicherweise in *Lastenheften* als Artefakte beschrieben, deren Format und Aufbau von Zulieferer zu Zulieferer variiert. Unter dem Begriff *Lastenheft* werden alle Artefakte (Dokumente der Spezifikation und des Entwurfs) auf System- und Komponentenebenen zusammengefasst. Grob ist ein Lastenheft charakterisiert durch Zielbestimmung, Produkteinsatz, Produktübersicht, Produktfunktionen, Produktdaten, Produktleistungen, Qualitätsanforderungen und sonstige Ergänzungen.

Die populärsten, dokumentenbasierten Tools zur Entwicklung von Lastenheften sind die Werkzeuge Microsoft Word® (Textverarbeitung), Microsoft Excel® (Tabellenkalkulation) (kurz: MS Word sowie MS Excel, [MSOF07]) und das spezifisch für das Anforderungsmanagement entworfene Werkzeug IBM Rational® DOORS (kurz: DOORS, [DOOR09]). Die aufgeführten Werkzeuge sind insbesondere in den Domänen Automobilindustrie, Luft- und Raumfahrt sowie Bahntechnik und Maschinenbau ein De-facto-Standard. Andere verwendete Werkzeuge wie ARCWAY/Cockpit, Borland/CaliberRM, QA-Systems/IRqA, MKS/Integrity, TREND/Analyst u. a. sowie deren Dateiformate können in dieser Arbeit aus Platzgründen nicht näher betrachtet werden. Sie sind jedoch technisch ähnlich beschaffen.

Aus der dokumentenbasierten Anforderungsdefinition existieren im Entwicklungsprozess die Artefakte Microsoft Word Dokument (Dateiendung: .doc; .docx) sowie Microsoft Excel Tabelle (Dateiendung: .xls; .xlsx). Das Artefakt-Metamodell ist nach [OOXML06] als XML-Schema spezifiziert. Nachfolgendes Beispiel zeigt das MS Office XML-Schema der *coreProperties*-Klasse für die Angabe von Metainformationen eines *Packages*.

Beispiel: MS Office XML-Schema (*coreProperties*-Klasse) nach OOXML

```
<?xml version="1.0" encoding="UTF-8"?><xs:schema targetNamespace=
            "http://schemas.openxmlformats.org/package/2006/metadata/core-properties"
   xmlns   ="http://schemas.openxmlformats.org/package/2006/metadata/core-properties"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:dcterms= "http://purl.org/dc/terms/"
   elementFormDefault="qualified" blockDefault="#all">
<xs:import namespace="http://purl.org/dc/elements/1.1/"
         schemaLocation="http://dublincore.org/schemas/xmls/qdc/2003/04/02/dc.xsd" />
<xs:import namespace="http://purl.org/dc/terms/"
    schemaLocation="http://dublincore.org/schemas/xmls/qdc/2003/04/02/dcterms.xsd" />
<xs:element name="coreProperties" type="CT_CoreProperties" />
```

```
<xs:complexType name="CT_CoreProperties">
 <xs:all>
  <xs:element name="category" minOccurs="0" maxOccurs="1" type="xs:string" />
   <xs:element name="contentStatus" minOccurs="0" maxOccurs="1" type="xs:string" />
   <xs:element name="contentType" minOccurs="0" maxOccurs="1" type="xs:string" />
   <xs:element ref="dcterms:created" minOccurs="0" maxOccurs="1" />
   <xs:element ref="dc:creator" minOccurs="0" maxOccurs="1" />
   <xs:element ref="dc:description" minOccurs="0" maxOccurs="1" />
   <xs:element ref="dc:identifier" minOccurs="0" maxOccurs="1" />
   <xs:element name="keywords" minOccurs="0" maxOccurs="1" type="xs:string" />
   <xs:element ref="dc:language" minOccurs="0" maxOccurs="1" />
   <xs:element name="lastModifiedBy" minOccurs="0" maxOccurs="1" type="xs:string" />
   <xs:element name="lastPrinted" minOccurs="0" maxOccurs="1" type="xs:dateTime" />
   <xs:element ref="dcterms:modified" minOccurs="0" maxOccurs="1" />
   <xs:element name="revision" minOccurs="0" maxOccurs="1" type="xs:string" />
   <xs:element ref="dc:subject" minOccurs="0" maxOccurs="1" />
   <xs:element ref="dc:title" minOccurs="0" maxOccurs="1" />
   <xs:element name="version" minOccurs="0" maxOccurs="1" type="xs:string" />
  </xs:all>
</xs:complexType></xs:schema>
```

In den MS Office-Werkzeugen vor der Version 2007 sind diese Dokumente in einem proprietären Dateiformat gespeichert, können jedoch über die Office-API in das XML-Format exportiert werden. Erst ab der Version 2007 führte der Hersteller OOXML mit den Werkzeugen ein, welcher mittlerweile als ISO-Standard DIS 29500 vorliegt. Das Dokumentenformat ist durch die XML-Repräsentation offen, wodurch die gesamte (Struktur-)Dokumentation zugänglich ist und zur weiteren Entwicklung kompatibler Anwendungen verwendet werden kann (vgl. auch [ECKE09]).

Ein weiteres geläufiges Format für Bürosoftware ist das *Open Document Format for Office Applications* (ODF) nach [ISO26300]. Der ODF-Standard wurde ursprünglich von Sun entwickelt, durch die Organisation OASIS als Standard spezifiziert und 2006 als internationale Norm ISO/IEC 26300:2006 veröffentlicht. ODF ist auch ein offener Standard für Dateiformate von Werkzeugen der Open Office Suite wie von Texten (Writer), Tabellen (Calc), Präsentationen (Impress), Zeichnungen mit Bildern und Diagrammen (Draw). ODF verwendet wie OOXML eine XML-basierte Formatierungssprache zur Strukturierung des Artefakts. Für mathematische Formeln (Calc) wird eine Untermenge von MathML verwendet. ODF kann mit beliebigen weiteren XML-Sprachen angereichert werden.

Beispiel: Open Office XML-Schema (*meta*-Klasse) nach ODF

```
<?xml version="1.0" encoding="UTF-8"?>
<office:document-meta
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:xlink="http://www.w3.org/1999/xlink">
          <office:meta>
           <meta:generator>OpenOffice.org/1.9.118
             $Win32 OpenOffice.org_project/680m118$Build-8936</meta:generator>
           <meta:initial-creator>Vorname Nachname</meta:initial-creator>
           <meta:creation-date>2010-01-09T10:00:00</meta:creation-date>
           <dc:creator>Vorname Nachname</dc:creator>
           <dc:date>2010-01-09T10:00:00</dc:date>
           <meta:printed-by>Vorname Nachname</meta:printed-by>
           <meta:print-date>2010-01-09T10:00:00</meta:print-date>
           <dc:language>de-DE</dc:language>
           <meta:editing-cycles>25</meta:editing-cycles>
           <meta:editing-duration>PT6H11M44S</meta:editing-duration>
           <meta:user-defined meta:name="Meta-Info 1"/>
           <meta:user-defined meta:name="Meta-Info 2"/>
           <meta:user-defined meta:name="Meta-Info 3"/>
           <meta:user-defined meta:name="Meta-Info 4"/>
           <meta:document-statistic
                   meta:table-count="1"
                   meta:image-count="2"
                   meta:object-count="3"
```

```
                    meta:page-count="10"
                    meta:paragraph-count="100"
                    meta:word-count="12345"
                    meta:character-count="18034"/>
          </office:meta>
</office:document-meta>
```

Für eine Mehrbenutzer-Anforderungsdefinition wird im Entwicklungsprozess häufig das datenbankbasierte Werkzeug DOORS eingesetzt. DOORS ist ein mehrplatzfähiges Client-Server-Tool, welches für die Weitergabe von Anforderungen an Zulieferer z. B. Microsoft Word Dokumente generiert. Es setzt auf einem proprietären Datenbankformat auf und stellt eine minimale Programmierschnittstelle bereit. Mit dem Werkzeug DOORS können beliebige Anforderungen in Textform hierarchisch im Baum strukturiert erfasst und zentralisiert verwaltet (Rechte, Sichten, Versionen) werden. Auch ist es möglich, innerhalb textueller Informationen grafische Repräsentationen als Abbildungen einzubetten.

Im Anforderungsmanagement fehlte lange Zeit ein werkzeug- und herstellerübergreifender Austauschstandard für Anforderungen (vgl. [RIF05]). Externen Autoren einen Zugriff auf die unternehmensinterne Anforderungsdatenbank zu gewähren, war aus politischen wie auch aus praktischen Gründen technisch aufwendig und ferner nicht erwünscht. Während bei den Automobilherstellern vielfach DOORS im Einsatz ist, sieht die Situation bei den Zulieferern deutlich inhomogener aus. Bei kleineren Unternehmen sind meist aus Kostengründen dokumentenbasierte Werkzeuge im Einsatz. Anforderungen konnten technisch somit nicht in einem konsistenten, übergreifenden, digitalen Format ausgetauscht werden.

Das von der HIS daraufhin entwickelte *Requirements Interchange Format* (RIF) ermöglicht den Austausch von Anforderungsdefinitionen zwischen den Werkzeugen verschiedener Hersteller [RIF05]. Die Basis einer Artefakt-Struktur bildet das Metamodell von RIF (v1.1a), welches auszugsweise in der Abbildung 11 dargestellt ist.

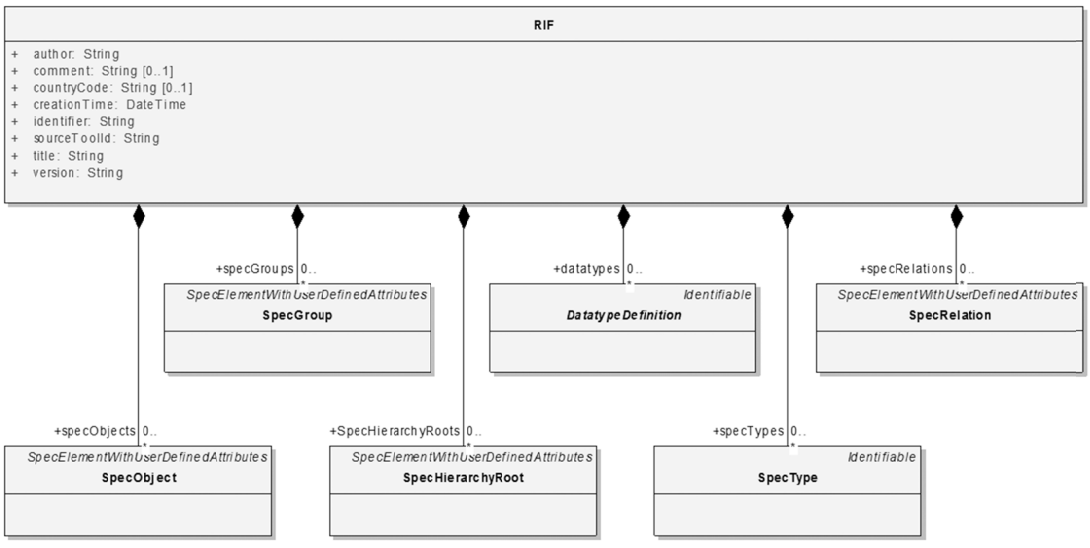


Abbildung 11: Metamodell einer Anforderungsdefinition, nach [RIF05]

Um Anforderungen in das RIF-Format zu transformieren, müssen die Produkthersteller von Anforderungsmanagement-Software entsprechende Export- und Importschnittstellen in ihre Werkzeuge integrieren. Eine Reihe von Werkzeug-Herstellern (wie z. B. IBM) hat dies bereits in die Werkzeugkette integriert. Die momentane Situation lässt vermuten, dass RIF bald von den gängigsten Werkzeugen im Anforderungsmanagement verarbeitet werden

kann. Angelehnt an die AUTOSAR-Standardisierung nutzt auch RIF das XML-Format. Jeder konkrete Informationstyp im RIF-Modell wird auf ein XML-Element abgebildet (vgl. auch [BESS06]). Der Name des XML-Elements besteht ausschließlich aus Großbuchstaben. Bei zusammengesetzten Namen im RIF-Modell werden diese in XML durch einen Bindestrich getrennt. Abgeleitet aus dem Metamodell besteht ein RIF-basiertes Artefakt aus dem Wurzel-Element *RIF*, in welchem Metadaten des Artefakts sowie die in der Datei verwendeten Datentypen und Anforderungen eingebettet sind. Die Datentypen basieren auf primitiven Datentypen, können jedoch projektspezifisch in ihrem Wertebereich eingeschränkt werden. Anforderungs-Objekte werden als einzelne *SpecObjects* zu einer Spezifikationen (*SpecGroup*) gruppiert. Die *SpecGroup* kann wiederum hierarchisch geschachtelt werden, womit sich auch die Gliederung von komplexen Spezifikationsstrukturen abbilden lässt. Um beispielsweise den RIF-Informationstyp *SpecObject* auf ein XML-Element mit dem Namen SPECOBJECT abzuleiten, wird aus dem RIF-Metamodell ein XML-Schema extrahiert.

Das folgende Beispiel zeigt einen Ausschnitt aus der XML-Schema-Definition für die *SpecObject-Klasse.*

Beispiel: Anforderungsdokumentations-XML-Schema (S*pecObject*-Klasse) nach RIF

```
<xsd:complexType name="SPEC-OBJECT">
<xsd:annotation>
 <xsd:documentation>The atomic specification object.</xsd:documentation>
</xsd:annotation>
<xsd:all>
<xsd:element name="DESC" type="xsd:string" minOccurs="0" maxOccurs="1" />
<xsd:element name="IDENTIFIER" type="xsd:string" minOccurs="1" maxOccurs="1" />
<xsd:element name="LAST-CHANGE" type="xsd:dateTime" minOccurs="1" maxOccurs="1" />
<xsd:element name="LONG-NAME" type="xsd:string" minOccurs="0" maxOccurs="1" />
<xsd:element name="TYPE" minOccurs="1" maxOccurs="1">
 <xsd:complexType>
  <xsd:choice minOccurs="1" maxOccurs="1">
   <xsd:element name="SPEC-TYPE-REF" type="RIF:REF"/>
  </xsd:choice>
 </xsd:complexType>
</xsd:element>
<xsd:element name="VALUES" minOccurs="0" maxOccurs="1">
<xsd:complexType>
 <xsd:choice minOccurs="0" maxOccurs="unbounded">
   <xsd:element name="ATTRIBUTE-VALUE-EMBEDDED-DOCUMENT"
                type="RIF:ATTRIBUTE-VALUE-EMBEDDED-DOCUMENT"/>
   <xsd:element name="ATTRIBUTE-VALUE-EMBEDDED-FILE"
                type="RIF:ATTRIBUTE-VALUE-EMBEDDED-FILE"/>
   <xsd:element name="ATTRIBUTE-VALUE-ENUMERATION"
                type="RIF:ATTRIBUTE-VALUE-ENUMERATION"/>
   <xsd:element name="ATTRIBUTE-VALUE-FILE-REFERENCE"
                type="RIF:ATTRIBUTE-VALUE-FILE-REFERENCE"/>
   <xsd:element name="ATTRIBUTE-VALUE-SIMPLE" type="RIF:ATTRIBUTE-VALUE-SIMPLE"/>
   <xsd:element name="ATTRIBUTE-VALUE-XML-DATA" type="RIF:ATTRIBUTE-VALUE-XML-DATA"/>
 </xsd:choice>
</xsd:complexType>
</xsd:element>
</xsd:all>
```

Wir stellen daher fest, dass für alle in dieser Arbeit betrachteten Werkzeuge MS Word, Writer und DOORS und derer Artefakte in der Anforderungsdefinition folgender Satz gilt:

Artefakte aus der Anforderungsdefinition **(3.3)**

„Artefakte aus der *Anforderungsdefinition* liegen meist in herstellerspezifischen, proprietären Datenformaten oder in einem Datenbanksystem vor. Eingesetzte Werkzeuge (Tools) bzw. Werkzeug-Erweiterungen wie MS Word, Writer und DOORS mit RIF-Export ermöglichen für die Anforderungsdefinition jedoch eine Ableitung von textuellen Anforderungen in den offenen Standard nach OOXML, ODF bzw. RIF als XML-basiertes Artefakt.“

3.8.2 Artefakte des modellbasierten Systementwurfs

Der in dieser Arbeit betrachtete Kontext schränkt die Anzahl der im modellbasierten Systementwurf verwendeten Modellierungssprachen und Modellierungswerkzeuge, demzufolge auch die damit entwickelten Artefakt-Typen, auf folgende drei Klassen ein:

(1) *Objektorientierte Modellierungssprachen,*

(2) *Signalflussorientierte Modellierungssprachen*,

(3) *Implementierungsorientierte Modellierungssprachen*.

Im Bereich der Entwicklung von eingebetteten Systemen werden *Architecture Description Languages* (ADL) im Systementwurf für das Architektur-Design eingesetzt. Diese sind spezialisierte semi-formale Sprachen mit textuellen und grafischen Notationen zum Entwurf, der Analyse und Simulation von Software-Architekturen, ähnlich der Notationselemente der UML (vgl. Kapitel 2.4.3). Sprachmittel sind Komponenten, Verbinder sowie Mittel zur Darstellung der Software-Architektur. Einige ADLs können direkt in Code übersetzt werden, für andere ist die Implementierung der spezifizierten Architektur offen.

In (1) werden folglich im Systementwurf Anwendungsfälle (Use Cases), System- bzw. Software-Architekturbeschreibungen (Class Diagrams) sowie Ablaufdiagramme (Activity Diagrams) objektorientiert modelliert. Eine hierfür oft verwendete Modellierungssprache ist die UML [UML22]. Die UML und ihre Modelle (UML-Modelle) werden in dieser Arbeit als repräsentative Artefakte für die objektorientierte Modellierung im Systementwurf betrachtet. Dies resultiert aus der Beobachtung, dass auch vom UML-Standard abweichende Profile, wie die EAST-ADL [EAST08] oder MARTE [MRTE08] bis hin zur SysML [SYSML11] im Systems Engineering, identische Modell- bzw. Artefakt-Typen (XMI-Dateien) entstehen. Speziell für den Software-Architektur-Entwurf sind auch softwareentwicklungsgetriebene Entwurfssprachen im Einsatz, wie z. B. das *Eclipse Modeling Framework* (EMF) im Bereich der Java-Entwicklung auf Basis von Eclipse [STEIN09].

Bei den Modellierungswerkzeugen sind generell drei Kategorien zu beobachten:

M1) *Modellierungswerkzeuge mit dem Hauptfokus auf den Systementwurf,*

M2) *Modellierungswerkzeuge mit dem Hauptfokus auf Software-Entwicklung,*

M3) *Modellierungswerkzeuge mit dem Hauptfokus auf Embedded-Software-Entwicklung.*

Unter (***M1***) lassen sich Werkzeuge wie Enterprise Architect® von Sparx Systems [SPAX09] oder Rational® System Architect Suite von IBM [RATI09] einordnen. Mittels Enterprise Architect® wurden das AUTOSAR-Metamodell sowie das RIF-Metamodell modelliert.

In der Klasse (***M2***) lassen sich Rational® Rose Modeler von IBM [RATR09] und Together® Designer [TOGE09] von Borland sowie Eclipse/EMF [STEIN09] einordnen.

Unter (***M3***) fallen Modellierungswerkzeuge wie Artisan Studio® von Artisan Software Tools [ARTI09] und Rational® Rhapsody von IBM [RATA09]. Der starke Fokus auf Werkzeuge von IBM resultiert aus den Aufkäufen des Herstellers vieler Modellierungswerkzeuge in den letzten Jahren. Die Klassifizierung ist auch nicht ausschließlich – teilweise verschmieren Funktionalitäten über diese Kategorien (***M1-M3***).

Regelbasierte Modellprüfungen fehlen jedoch in allen Werkzeugen oder sind nur rudimentär durch OCL-Auswertung integriert. Ein Beispiel für einen objektorientierten Modell-Entwurf wurde bereits in Kapitel 2.4.3 (Abbildung 9) gegeben.

In (***M2***) werden im Systementwurf signalflussorientierte Verhaltensmodelle in Blockdiagrammen erstellt, wie in der Abbildung 12 gezeigt. Die angelegte Motorspannung *U_IN*, der Motorwiderstand *R* sowie die Motordrehzahl *RPM* sind Funktionsparameter und können vor einer Simulation gesetzt werden. Aus dem ausgeführten (simulierten) Verhaltensmodell eines einfachen E-Motors können so der Stromwert des Motors *I_OUT* und das Drehmoment *TAU_OUT* errechnet und auf dem internen Speicher (Workspace) abgelegt werden.

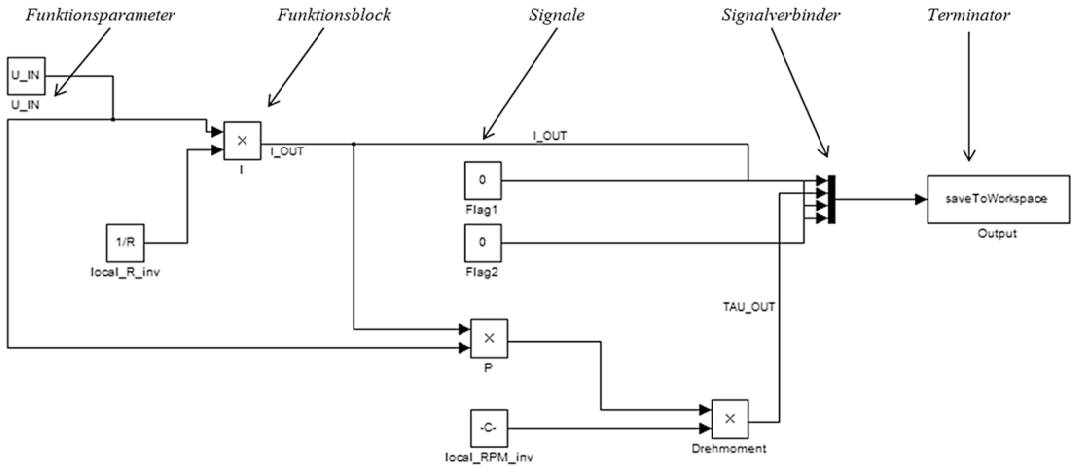


Abbildung 12: Verhaltensmodell (E-Motor) als Simulink-Modell

Nach [ANGE05; ANGE07] basiert die Modellierung auf dem blockorientierten Ansatz (Blockdiagramme), der auf die Beschreibung von gerichtetem Ein-Ausgangs-Verhalten von *Funktionsblöcken* orientiert ist. Modelle folgen nicht der Objektorientierung, sondern dem Paradigma *Quelle-Funktion-Senke* (Source-Function-Sink), also einem Erzeuger-zu-Funktion-zu-Verbraucher-Prinzip. Unter einem *Source Block* ist ein Wertgeneratorblock zu verstehen, der die Eingangssignale für die nachfolgenden Funktionsblöcke erzeugt. *Signale* werden durch *Signalverbinder* verbunden. Die Funktionsblöcke sind die eigentlichen Algorithmen im Modell, indem sie ihre Eingangssignale anhand beliebiger mathematischer Funktionen verarbeiten. *Terminatoren* ‚erden' das Signal bzw. den errechneten Wert und speichern diesen flüchtig oder nicht-flüchtig nach einer Simulation ab.

Im Modellierungswerkzeug werden die Funktionsblöcke aus einer Blockbibliothek auf die Zeichenfläche gezogen und mit Signalverläufen verbunden sowie über Kontextmenüs in ihren Parametern bestimmt. Verhaltensmodelle unterstützen den funktionalen Systementwurf zur Definition von regelungstechnischen Funktionskomponenten des Systems (Kapitel 2.2.3), ihrer Verbindungen und Schnittstellen durch semi-formale grafische Spezifikationen des Modells mittels editierbarer, hierarchischer Blockschaltbilder und auch integrierbaren Zustandsübergangsdiagrammen (State Charts). In den frühen Phasen des Systementwurfs werden signalflussorientierte Modelle häufig zur Algorithmen-Entwicklung eingesetzt, da bedingt durch eine große Zahl an verfügbaren Spezialbibliotheken zu allen mechatronischen Problemstellungen die Algorithmen-Entwicklung und die Funktionserprobung mittels Simulation und Rapid Prototyping sehr schnell durchgeführt werden kann.

Für die Auslegung und Systemsimulation bei der Entwicklung von Funktionen eines eingebetteten Systems ist eine Vielzahl von Simulationswerkzeugen im wissenschaftlichen, wie auch im industriellen Einsatz. Dies sind traditionell den Regler-Entwurf unterstützende

Programme wie SCADE [SACD09], Octave [GNUO09], Scilab [INRI09], MATRIXx [MATR09] oder MATLAB/Simulink/Stateflow [MLSL09] (kurz: ML/SL/SF).

Lässt sich ein System nicht ohne Weiteres durch explizite, gewöhnliche Differentialgleichungen beschreiben, d. h. kommen die für elektrische und regelungstechnische Systeme typischen algebraischen Nebenbedingungen und implizite Zusammenhänge hinzu, so müssen implizite Differential-Algebraischen-Gleichungen mit Werkzeugen wie SimulationX [SIMX09], SystemVision [SYST09] oder Dymola [DYMO09] gelöst werden. Dies wird auch durch entsprechende Modellierungssprachen wie Modelica [MODL09] und teils sehr umfassende Modellbibliotheken unterstützt. Die genannten Systeme bieten auch Ansätze zur Visualisierung der Simulationsergebnisse mittels Codegeneratoren.

Ein populäres Modellierungs- und Simulationswerkzeug im Bereich der Entwicklung eingebetteter Systeme ist ML/SL/SF, welches in dieser Arbeit exemplarisch als Referenzwerkzeug für das Fallbeispiel (Kapitel 8) ausgewählt wird. Die MATLAB-Produktfamilie unterstützt die Entwicklung und Analyse linearer wie nichtlinearer dynamischer Systeme. Während MATLAB ein mathematisches Framework mit integrierter, prozeduraler Programmiersprache (*M-Skript*) bereitstellt, stellen die Produkterweiterungen (*Toolboxen*) ‚Simulink' und ‚Stateflow' jeweils Simulatoren für Blockschaltbilder und Zustandsdiagramme zur Verfügung. Ein grafischer Editor erlaubt dabei eine visuelle Darstellung und durch Verschachtelung die Entwicklung komplexer Modelle (System-Subsystem-Prinzip). Regelbasierte Modellprüfungen sind nicht durch das Zusatzwerkzeug ‚Model Advisor' [MODA09] durchführbar.

Des Weiteren sind im Elektronik-Bereich rechnergestützte Entwurfsmethoden, speziell beim Hardware-Entwurf mittels integrierter Schaltungen fest etabliert. Im Bereich digitaler Schaltungen ist der Entwurfsprozess durch einen hohen Grad der Automatisierung charakterisiert. Umfangreiche Toolumgebungen unterstützten die High-Level-Beschreibung mit Sprachen wie VHDL [VHDL09] und Verilog HDL [VERI09], die automatisierte Umsetzung in Gatterschaltungen (Synthese) und die Überführung in das Chip-Layout.

Im für die Mechatronik relevanteren Bereich der analogen und gemischt analogen und digitalen Schaltungstechnik haben sich in den letzten Jahren Sprachen etabliert, die aus der digitalen Welt kommend auch die Verhaltensbeschreibung für analoge Komponenten unterstützten. Der zugrunde liegende mathematische Ansatz ist auch hier ein System Differential-Algebraischer-Gleichungen. Mit weitverbreiteten Modellierungssprachen wie SABER VHDL-AMS [SABE09], Verilog-AMS [VAMS09] oder SystemC-AMS [SYSC09] ist ebenso die Beschreibung nichtelektrischer Systeme möglich. Eine Simulationsvisualisierung ist in diesem Bereich noch unüblich. Qualitätsorientierte Modellprüfungen nach Richtlinien durch Prüfwerkzeuge fehlen außerdem.

In die Klasse (***M3***) der implementierungsorientierten Modellierungssprachen lässt sich hierbei speziell im Bereich der Fahrzeugelektronik neben ML/SL/SF insbesondere die ASCET-SD Produktfamilie einordnen, welche in dieser Arbeit exemplarisch als zusätzliches Referenzwerkzeug für das Fallbeispiel (Kapitel 8) ausgewählt wird.

Nach [ASCT09] sind die Werkzeuge ASCET-MD (Modellierung und Funktionsdesign), ASCET-RP (Rapid Prototyping) und ASCET-SE (Software Engineering) für den Entwicklungsprozess von Software für eingebettete Steuergeräte durch das modellbasierte Design von Steuer-, Regelungs- und Diagnosefunktionen konzipiert. Die ASCET-Werkzeuge beschreiben eingebettete Softwarefunktionen in Form von Rechenrastern, sogenannten *Prozessen*, *Klassen* und *Instanzen*. Das Verhalten von Klassen wird mittels Datenflussdiagrammen und endlichen Automaten beschrieben. Die Abbildung 13 zeigt das

Implementierungsmodell einer Integrator-Funktion in ASCET. Der zeitliche Verlauf der Ausgangsgröße einspricht dem zeitlichen Integral des Verlaufs der Eingangsgröße.

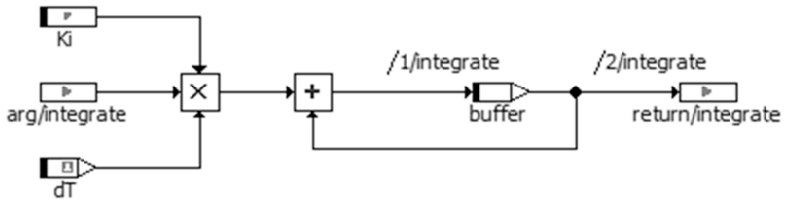


Abbildung 13: Integrator-Funktion als ASCET-Modell, nach [ASCT09]

Innerhalb von Prozessen wird die Ausführungsreihenfolge durch die Spezifikation von Teildatenflüssen mittels *Ausdrücken* definiert. Ausdrücke werden in Blockdiagrammen (vgl. ML/SL/SF) durch das Verbinden von *Elementen* oder anderen Ausdrücken mit *Operatoren* gebildet. Sie werden grafisch durch das Verbinden der Rückgabeanschlüsse von Elementen oder Operatoren mit den Argumentanschlüssen von Methoden oder anderen Operatoren aufgebaut. Die Abbildung 14 zeigt unterschiedliche Additionsarten von Elementen.

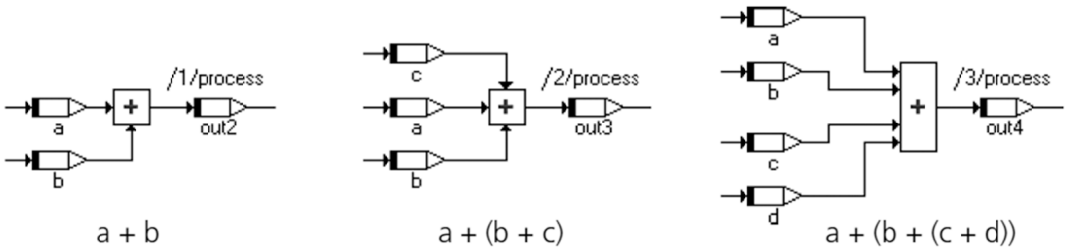


Abbildung 14: Spezifikation von Ausdrücken in Blockdiagrammen, nach [ASCT09]

Der *Ablauf* (Schedule) von Prozessen und die damit verbundene Zuordnung auf Betriebssystem-Tasks (vgl. auch Kapitel 2.2.5) bestimmt die Ausführungsreihenfolge innerhalb eines eingebetteten Systems. Diese Art der Softwaremodellierung erlaubt einerseits eine implementierungsorientierte Darstellung von eingebetteter Echtzeitsoftware, andererseits eine plattformspezifische Implementierung auf Mikrocontrollern mittels automatischer C-Code-Generierung.

Die grafische Entwicklungsumgebung von ASCET-MD (Modellierung und Funktionsdesign) ermöglicht eine modellbasierte Funktionsspezifikation unabhängig von dem später implementierten eingebetteten System (Target) und bietet infolgedessen dem Entwickler eine Abstraktion von der Zielplattform. Hierfür setzt ASCET-RP (*Rapid Prototyping*) für den Entwickler das zielidentische Rapid-Prototyping automatisiert um, sodass hiermit Softwaretests durchgeführt werden können.

Das Werkzeug ASCET-SE (Software Engineering) ermöglicht das automatische Generieren von Steuergerätesoftware auf der Basis von Blockdiagrammen und Zustandsautomaten, ähnlich wie bei ML/SL/SF durch die Zusatzprogramme ‚Real-Time Workshop Embedded Coder' [MLSL09] oder ‚TargetLink' [TGTL09].

Wie im Kapitel 2.2.4 beschrieben wurde, kann aus Implementierungsmodellen plattformspezifischer Code entwickelt werden. Beide Modellierungswerkzeuge, ASCET und

ML/SL/SF, werden in verschiedenen Phasen der Software-Entwicklung mit unterschiedlicher Zielsetzung verwendet. Beim Übergang von der Algorithmen-Entwicklung hin zur Feinentwicklung von Funktionen wird im Entwurfsprozess häufig ein Werkzeugwechsel vollzogen, da die tiefere Abstraktionsebene der Serienentwicklung zusätzliche Anforderungen an die Werkzeugumgebung stellt. Ein Werkzeugübergang nach ASCET bietet hier – im Gegensatz zur hohen Abstraktion mit ML/SL/SF – eine realistischere Umgebung. Diese erlaubt eine Integration auf Basis des OSEK-Betriebssystems (vgl. Kapitel 2.2.5.1) bzw. in Richtung der Steuergeräte-Applikation mit Kalibrierung und Diagnose sowie die automatische Generierung von Seriencode. Qualitätsorientierte Modellprüfungen nach Richtlinien sind in den ASCET-Werkzeugen nicht enthalten.

3.8.2.1 Auswertung der Untersuchung

Bisher war der Modelldatenaustausch zwischen den Modellierungswerkzeugen vom Typ (***M1***) sehr zeitintensiv, da die Übertragung manuell von Hand durchgeführt werden musste. Wie im Kapitel 2.5.2 eingeführt, wurde hierfür der XMI-Standard zum Modellaustausch für MOF-, UML- und SysML-Modelle geschaffen, der heutzutage in unterschiedlichen Versionen vorliegt und stetig erweitert wird.

Wir stellen daher fest, dass für alle in dieser Arbeit betrachteten Werkzeuge und deren Artefakte objektorientierter Modellierung folgender Satz gilt:

Artefakte objektorientierter Modellierung **(3.4)**
„Artefakte *objektorientierter Modellierungssprachen* liegen meist in herstellerspezifischen, proprietären Datenformaten vor. Verfügbare Modellierungswerkzeuge, wie der Enterprise Architect bzw. Werkzeug-Erweiterungen (Tools) ermöglichen jedoch die Ableitung eines MOF-/UML-/SysML-Modells in den offenen Standard XMI als XML-basiertes Artefakt.“

Ausgenommen ist diese Feststellung für die signalflussorientierte Modellierungssprache ML/SL/SF und die implementierungsorientierte Modellierungssprache ASCET-MD. ML/SL/SF-Modelle werden in einem *M-Skript* (ein textuelles, proprietäres Dateiformat) in ASCII-Codierung gespeichert. Zudem kompiliert MATLAB in M-Skript entwickelte Modelle und Programmcode in einen intern repräsentierten Zwischencode (P-Code) im Binärformat (bei Modellen im MDL-Dateiformat), der erst zur Ausführung von MATLAB interpretiert wird. Standardmäßig wird der Zwischencode nicht auf der Festplatte gespeichert, sondern nur zur Laufzeit im Arbeitsspeicher gehalten. Er kann jedoch persistent in einem verschlüsselten Binärformat auf der Festplatte abgelegt werden. Bei der Weitergabe von Modellen werden diese proprietären Dateien durch Kopieren verteilt.

ML/SL/SF verfügt derzeit über keine standardisierten Schnittstellen für die Ableitung eines Modells in das XML-Format. Um nun die in Simulink-Modellen enthaltenen Informationen in ein standardisiertes Format konvertieren zu können, bietet das Werkzeug SimEx eine Funktionserweiterung für ML/SL/SF an [SIME09].

Ein Algorithmus im Werkzeug SimEx konvertiert dabei die Modelldaten aus Simulink und Stateflow in das universelle XML-Format (Dateiendung: .xml) und definiert selbst ein eigenes XML-Schema (DTD) für Simulink. Der Kern von SimEx ist ein Java-basiertes Konvertierungsmodul, welches ein im MDL-Format gespeichertes Simulink-Artefakt in das universelle XML-Format übersetzt (Abbildung 15), wobei die hierarchische Artefakt-Struktur und der Inhalt der Modelldaten bidirektional erhalten bleiben.

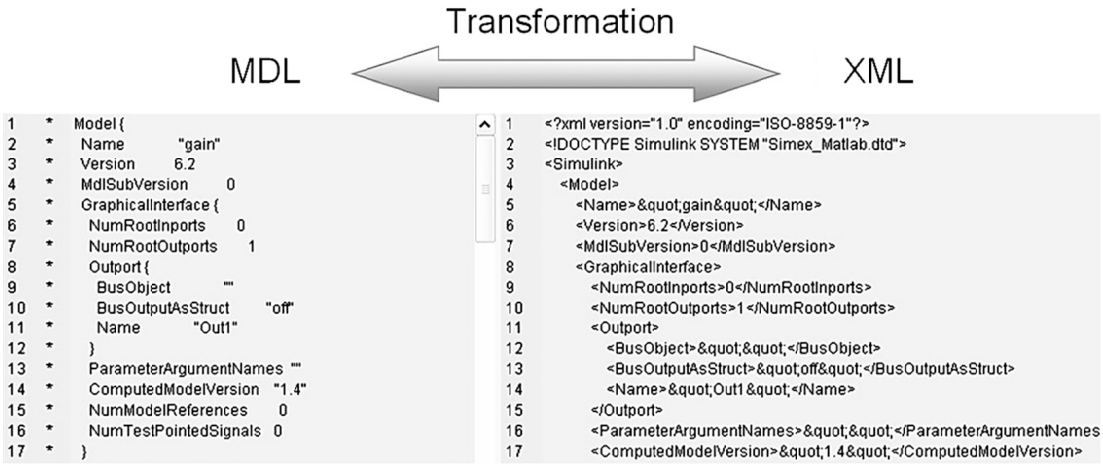


Abbildung 15: Transformiertes Simulink-Artefakt im XML-Format (Ausschnitt)

Darüber hinaus ermöglicht das Werkzeug die Rückübersetzung der XML-Daten in das MDL-Format, welches wieder in ML/SL/SF ladbar ist. Durch den Rückimport können Erweiterungen oder Änderungen des Modells durch andere XML-basierte Werkzeuge nach ML/SL/SF übernommen werden.

Wir stellen daher fest, dass für das in dieser Arbeit betrachtete Werkzeug ML/SL/SF und dessen Artefakte signalflussorientierter Modellierung folgender Satz gilt:

Artefakte signalflussorientierter Modellierung **(3.5)**
„Artefakte *signalflussorientierter Modellierungssprachen* liegen im Falle von ML/SL/SF in einem herstellerspezifischen, proprietären Datenformat vor. Werkzeuge bzw. Werkzeug-Erweiterungen (Tools) zur Modelltransformation (z. B. SimEx) ermöglichen jedoch die Ableitung eines Simulink-Modells in den offenen Standard XML."

Die implementierungsorientierte Modellierungssprache ASCET-MD basiert auf einer Datenbank mit proprietärer Datenstruktur. Modelle werden in einem oder mehreren Repositorien (Modell-Datenbanken) gespeichert. Über die Programmschnittstellen (ASCET-API) erfolgt die Modell-Manipulation, -Transformation und -Persistenz. Der Hersteller plant die komplette Umstellung auf eine XML-basierte, offene Persistenz von Repositorien in den kommenden Tool-Versionen. Bis dahin bietet die ASCET-API einen Export-Mechanismus zur vollständigen Ableitung der Modelle in das XML-Format [ETAS08].

Wir stellen daher fest, dass für das in dieser Arbeit betrachtete Werkzeug ASCET-MD und dessen Artefakte implementierungsorientierter Modellierung folgender Satz gilt:

Artefakte implementierungsorientierter Modellierung **(3.6)**
„Artefakte *implementierungsorientierter Modellierungssprachen* liegen im Falle von ASCET-MD in einem herstellerspezifischen, proprietären Datenbankformat vor. Die Programmschnittstelle (API) zur Modelltransformation ermöglicht jedoch die Ableitung eines ASCET-Modells in den offenen Standard XML."

3.8.3 Artefakte der Modulspezifikation

Präzise Modulspezifikationen für die Entwicklung einer konformen Software-Architektur eingebetteter Systeme nach dem OSEK- oder nach dem AUTOSAR-Standard, bestehend aus Beschreibungen für eingebettetes System und Software, benötigen Werkzeuge zur modellbasierten Modulspezifikation. Die aktuelle Version 3.1 des AUTOSAR-Standards (vgl. Kapitel 2.2.5.2) enthält präzise Spezifikationsmittel für die Entwicklung einer standardisierten Elektrik-/Elektronikarchitektur. Am Markt sind immer mehr Werkzeuge zur Unterstützung einer AUTOSAR-Entwicklung verfügbar.

Der Hersteller dSPACE bietet seine AUTOSAR-Unterstützung mit der Architektur-Software SystemDesk [SYSD09] zur modellbasierten Planung, Implementierung und Integration komplexer Systemarchitekturen und verteilter Softwaresysteme am Markt an, die das AUTOSAR-Datenformat unterstützt. Laut dSPACE wird die modellbasierte Modulspezifikation funktionaler Netzwerke und Software-Architekturen gemäß AUTOSAR-Standard, die Formalisierung der Hardware-Topologien und Netzwerkkommunikation, die Integration von Steuergeräte-Code, die Generierung einer AUTOSAR Runtime Environment (RTE) sowie die Simulation einzelner Software-Komponenten oder eines ganzen Steuergeräte-Verbundes ermöglicht.

Die DaVinci Tool Suite des Herstellers Vector Informatik bietet ebenfalls eine hardwarenahe Entwicklungsmethodik zur modellbasierten Modulspezifikation, welche den Entwurf von Elektronikfunktionen, z. B. für den Kfz-Innenraum, realisiert [DAVI09]. Laut Hersteller wird der funktionale Entwurf von Software erlaubt, sowohl die Entwicklung verteilter Systeme als auch die Integration von Anwendungssoftware auf Steuergeräten. Durch die Bereitstellung und Verwaltung modular aufgebauter Softwarekomponenten mit definierten Signalschnittstellen wird die Wiederverwendung von in sich getesteten Fahrzeugfunktionen in verschiedenen Steuergerätenetzwerken ermöglicht. Die DaVinci Tool Suite bietet zwei kooperierende Werkzeuge, die DaVinci DEV (Developer) Version als eine Entwicklungs- und Integrationsplattform für einzelne Steuergeräte und den DaVinci SAR (System Architect) als ein Werkzeug für den funktionalen Entwurf und die Integration von verteilten Systemen.

Eine Eclipse-basierte AUTOSAR-Entwicklungsplattform ist der AUTOSAR Builder von Geensys. Nach [GEEN09] handelt es sich um eine Entwickler-Plattform für AUTOSAR-konforme Software, in die sich Design-Tools von Drittanbietern und Plugin-Tools mit offener Schnittstelle integrieren lassen. Die Suite besteht aus den Komponenten AUTOSAR Authoring Tool (AAT), ECU Extract (ECE), Software Component Conformance Validation Tool (SCVT) und Generic ECU Configuration Editor (GCE). Das AAT soll die Entwicklung von AUTOSAR-Softwarekomponenten, die Steuergeräte und Systembeschreibungen auf Anwendungsebene unterstützen. Die Beschreibungsdaten werden in *Templates* und *ARPackages* organisiert dargestellt. Für jedes AUTOSAR-Element werden die Eigenschaften in Formularen aufgeführt. Ein grafischer Editor dient dazu, die Elemente als Diagramme zu visualisieren und neue Elemente zu erstellen. Mit ECE lässt sich eine einzelne hardwarenahe Beschreibung aus der Gesamtbeschreibung extrahieren. Das SCVT arbeitet auf Anwendungsebene und verifiziert die Beschreibung von SW-Komponenten (SWCs). Es bietet Analysen auf der Basis von Beschreibungsregeln für SWCs, statischer Analyse von Anwendungscode und des erzeugten RTE-Codes, wobei die Stimmigkeit zwischen SWC-Beschreibungen und zugehörigem Code geprüft wird. Außerdem liefert es Reports über Analyse und Validierung des Codes. Der GCE ermöglicht das Erstellen und Konfigurieren von Basis-Software-Modulparametern (BSW). Mit dem ECU-Parameter

Definition-Plugin (EPD) lassen sich Parameterdefinitionen erstellen. Damit können auch Definitionen im XML-Format für Softwaremodule erstellt werden, die keine AUTOSAR-BSW-Module sind. Das BSW Module Description-Plugin (BMD) dient zum Erstellen oder Modifizieren von BSW-Moduldefinitionen und Konfigurationen durch BSW-Anbieter. Schließlich wird das ECU Parameter Configuration-Plugin (EPC) verwendet, um ECU-Parameterkonfigurationen zu erstellen und um BSW-Modulkonfigurationen zu integrieren.

Die Basis des AUTOSAR-Standards ist das AUTOSAR-Metamodell [ASR09]. Nach den *Model Persistance Rules* (MPR) [WANG08] wird eine konforme Ableitung des Metamodells in das XML-Schema definiert. Das Beispiel (*SWC-IMPLEMENTATION*-Klasse) zeigt einen XML-Schema-Ausschnitt nach AUTOSAR Version 3.1.

Beispiel: XML-Schema (*SWC-IMPLEMENTATION*-Klasse) nach AUTOSAR 3.1

```
[...]
<!-- Element group for class SwcImplementation::SwcImplementation -->
<xsd:group name="SWC-IMPLEMENTATION">
 <xsd:sequence>
  <xsd:element name="BEHAVIOR-REF" minOccurs="0" maxOccurs="1">
   <xsd:complexType>
    <xsd:simpleContent>
     <xsd:extension base="AR:REF">
      <xsd:attribute name="DEST" type="AR:INTERNAL-BEHAVIOR--SUBTYPES-ENUM"/>
     </xsd:extension>
    </xsd:simpleContent>
   </xsd:complexType>
  </xsd:element>
 <xsd:element name="PER-INSTANCE-MEMORY-SIZES" minOccurs="0" maxOccurs="1">
<xsd:complexType>
 <xsd:choice minOccurs="0" maxOccurs="unbounded">
  <xsd:element name="PER-INSTANCE-MEMORY-SIZE" type="AR:PER-INSTANCE-MEMORY-SIZE" />
 </xsd:choice>
</xsd:complexType>
</xsd:element>
 <xsd:element name="REQUIRED-RTE-VENDOR" type="xsd:string" minOccurs="0" >
  <xsd:annotation>
    <xsd:documentation>Text</xsd:documentation>
      </xsd:annotation>
     </xsd:element>
  </xsd:sequence>
</xsd:group>
[...]
```

In der modellbasierten Modulspezifikation eingebetteter Systeme sind derzeit Artefakte die Beschreibungsdateien für AUTOSAR-konforme Softwarekomponenten, ggf. die RTE-Generierungs-Daten oder die Konfigurationen der AUTOSAR-Basis-Software. Sie sind mit den beschriebenen Werkzeugen bearbeitbar, importier- sowie exportierfähig. Die Artefakte der modellbasierten Modulspezifikation sind demnach alle XML-Beschreibungsdateien (Dateiendung: .axml), welche Informationen zur Struktur und internem Verhalten der SWCs, zur Steuergeräte-Konfiguration oder zum Gesamtsystem enthalten.

Wir stellen daher fest, dass für alle in dieser Arbeit betrachteten Werkzeuge und deren Artefakte in der Modulspezifikation nach AUTOSAR-Standard folgender Satz gilt:

Artefakte der Modulspezifikation **(3.7)**
„Artefakte der *Modulspezifikation* liegen im Falle des AUTOSAR-Standards (Version 3.1) als AXML-Artefakte nach dem offenen Standard XML vor.“

Im Folgenden wird für das Fallbeispiel (Kapitel 8) das Werkzeug SystemDesk als Referenz-Werkzeug für die Modulspezifikationen nach AUTOSAR-Standard und der Enterprise Architekt für die UML/SysML-Modellierung nach XMI-Standard weiter verfolgt, da beide die XML-basierte Persistenz von Modulspezifikationen hinreichend unterstützen.

3.8.4 Artefakte der Testspezifikation

Die Artefakte der Testspezifikation im modellbasierten Entwurf eingebetteter Systeme adressieren den Test der Verhaltens- oder Implementierungsmodelle. Aus Forschungsarbeiten der Daimler AG stammend, wurde zur systematischen Ermittlung redundanzarmer und fehlersensitiver Testfälle die *Klassifikationsbaum-Methode* (vgl. Kapitel 2.2.6) werkzeugtechnisch erstmals unterstützt [CON04]. Für den effizienten Einsatz dieser Methode wurde der *Klassifikationsbaum-Editor* (Classification Tree Editor, CTE) entwickelt. Es handelt sich um einen syntaxgesteuerten, grafischen Editor, der eine visuelle Unterstützung für die Testfallermittlung mit der Klassifikationsbaum-Methode bietet. Nach prototypischer Implementierung in Forschungsprojekten entsprangen zwei kommerzielle Versionen, CTE/XL und CT/ES, für den Einsatz am Markt. Der ‚*Classification Tree Editor CTE/XL*' [CTEX09] wurde seit Anfang 1990 entwickelt und seitdem unabhängig von modellbasierter Entwicklung zum Testen von C-Code eingesetzt. Er unterscheidet sich in der Funktionalität unwesentlich zum ‚*Classification Tree Editor for Embedded Systems* (CTE/ES)' [CTES09]. CTE/ES ist integraler Bestandteil der *MTest*-Suite [DSPA05]. MTest stellt ein weiteres, auf ML/SL/SF ausgerichtetes Werkzeug zur Erstellung von systematischen Tests für Simulink-Modelle, zur Verfügung. MTest ermöglicht das Black-Box-Testen von Simulink Subsystemen (System Under Test, SUT). Diese Subsysteme werden dafür in ein automatisch erstelltes Test-Framework (Umgebung) eingefügt [ZAN08]. Der eigentliche Test simuliert das Test-Framework mit bestimmten Eingangssignalen und zeichnet die entsprechenden Reaktionen (Ausgangssignale) des Subsystems (SUT) auf.

Entwicklungsartefakte der Testspezifikation sind mit dem Werkzeug CTE/XL gespeicherte Klassifikationsbäume im XML-Format vorliegend. Sie folgen in der inhaltlichen Strukturierung der herstellerspezifischen Umsetzung des CTE-Metamodells. Nachfolgendes Beispiel zeigt einen Ausschnitt des persistenten Klassifikationsbaums als XSD-Schema nach dem CTE/XL Metamodell.

Beispiel: Klassifikationsbaum-Editor XML-Schema (*tree*-Klasse), nach CTE/XL

```
[...]
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="layoutStyle">
   <xs:attribute name="fgcolor" type="xs:string" use="required"/>
   <xs:attribute name="fontstyle" type="xs:string" use="required"/>
   <xs:attribute name="fontfamily" type="xs:string" use="required"/>
   <xs:attribute name="bgcolor" type="xs:string" use="required"/>
   <xs:attribute name="fontsize" type="xs:int" use="required"/>
  </xs:complexType>
[...]
<xs:complexType name="tree">
 <xs:sequence>
  <xs:element name="nodelayout" maxOccurs="unbounded">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="layoutstyle" type="layoutStyle"/>
    </xs:sequence>
    <xs:attribute name="xpos" type="xs:int" use="required"/>
    <xs:attribute name="ypos" type="xs:int" use="required"/>
    <xs:attribute name="pagepid" type="xs:string" use="required"/>
   </xs:complexType>
  </xs:element>
<xs:element name="activetag" type="activetag" maxOccurs="unbounded"/>
<xs:element name="tree" type="tree" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="type" type="xs:string" use="required"/>
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="pid" type="xs:string" use="required"/>
</xs:complexType>
[...]
```

In der Testspezifikation werden oftmals auch Werkzeuge mit tabellarischer Notationsform zur Beschreibung der Testfälle verwendet. Das Tabellenkalkulationsprogramm Microsoft Excel ist hierbei aufgrund der Universalität heutzutage die meistverwendete Software in technischen Domänen. Sie stellt eine weitere Klasse für wesentliche, in dieser Arbeit betrachtete Artefakte dar. Das Artefakt-Metamodell von MS Excel ist, wie bereits geschildert, nach [OOXML06] als XML-Schema spezifiziert. Somit gelten die gleichen Möglichkeiten, wie wir es in der Anforderungsdefinition als Satz definiert hatten.

Zum Austausch von Testfallbeschreibungen zwischen verschiedenen Werkzeugen schlägt [IMMOPS06] die *Test Markup Language* (TestML) vor. Das XML-Schema definiert eine eigene Sprachsemantik für den Austausch von Testfällen auf Basis einer XML-Syntax.

Wir stellen daher fest, dass für die in dieser Arbeit betrachteten Werkzeuge MS Excel sowie CTE/XL und derer Artefakte (z. B. auch als TestML) der modellbasierten Testspezifikation folgender Satz gilt:

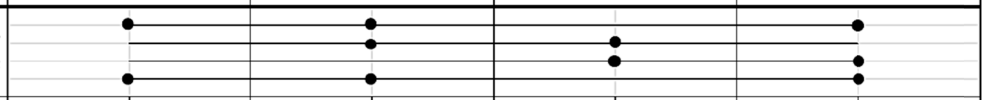
Artefakte der Testpezifikation **(3.8)**
„Artefakte der *Testspezifikation* liegen im Falle des Klassifikationsbaum-Editors CTE/XL bereits nach dem offenen Standard XML vor oder lassen sich durch eine Auszeichnungssprache (TestML) beschreiben. Artefakte im Falle von MS Excel liegen nach OOXML-Standard als XML-Artefakte vor.“

Im Folgenden wird in dieser Arbeit für Fallbeispiele der Testspezifikation der CTE/XL und MS Excel als Referenz-Werkzeug für modellbasierte Testspezifikationen im Systementwurf weiter verfolgt, da beide die XML-basierte Persistenz, von Klassifikationsbäumen bei CTE/XL bzw. tabellarischer Testspezifikation bei MS Excel, unterstützen.

3.8.5 Zusammenfassung und Auswahl

Zusammenfassend wird das Ergebnis der Klassifizierung und der Auswahl typischer im modellbasierten Entwicklungsprozess eingesetzter Werkzeuge und der damit erzeugten Artefakte in der Tabelle 14 dargestellt.

Tabelle 14: Auswahl betrachteter Werkzeuge & Artefakte

Prozessphase	***Anforderungsdefinition***		***Modellbasierter Systementwurf***		***Implementierungsphase***		***Testspezifikation***	
Logische Relation								
Werkzeug	*MS Word*	*DOORS*	*ML/SL/SF*	*ASCET-MD*	*SystemDesk*	*Enterprise Architect*	*MS Excel*	*CTE/XL*
Artefakt-Klasse	Text, Dokument	Text, Datenbank	Modell, Dokument	Modell, Datenbank	Modell, Dokument	Modell, Datenbank	Tabelle, Dokument	Modell, Dokument
Artefakt-Struktur	OOXML	RIF	proprietär	proprietär	AXML	XMI	OOXML	proprietär
Normung	ISO	HIS	-	-	AUTOSAR	OMG	ISO	-
Metamodell	Microsoft	HIS	MathWorks	ETAS	AUTOSAR	MOF	Microsoft	Berner& Mattner
Serialisier-barkeit	ja, nach XML	ja, nach XML	ja, nach XML	ja, nach XML	ist XML	ja, nach XML	ja, nach XML	ist XML

3.9 Zusammenfassung

Das Erfüllen von Konformitätsanforderungen kann von Entwicklungs-Teams in kollaborativer Umgebung heutzutage nur durch ein mühsames, kostenaufwendiges und fehleranfälliges Vergleichen einer großen Menge von Daten erreicht werden. Diese Tätigkeiten geschehen durch die im Kapitel 3.4 beschriebenen Prüfmethoden vielfach manuell, da sie typischerweise einen Übergang zwischen verschiedenen Dokumenten, Artefakten oder Prozessen betreffen – wobei Artefakte mit verschiedenen Werkzeugen erstellt werden. Bei immer komplexer werdenden Entwicklungsartefakten, wie ein Komplexitätsbeispiel nach [TKL06] am Fahrzeugfunktionsmodell eines Komfort-steuergerätes mit 3750 zu prüfenden Objekten verdeutlicht, sind Entwickler mit manuellen Prüfungen überfordert und brauchen einen zu hohen Zeitaufwand, was hohe Kosten verursacht. Auch ist es menschlich, dass bei den manuellen Prüfmethoden einige Fehler übersehen werden und dadurch die Fehlerrate nicht weiter gesenkt werden kann.

Die in Kapitel 3.7.1 vorgestellten Prüfwerkzeuge helfen, durch eine regelbasierte Konformitätsprüfung einen Automatismus der statischen Analyse zu schaffen, und sind auch am Markt vermehrt vorzufinden. Für die statische Artefakt-Analyse werden derzeit für die betrachteten Entwicklungsphasen eingebetteter Systeme vorwiegend im Bereich der Modell- und Code-Analyse Prüfwerkzeuge angeboten, wie die Übersicht in der Tabelle 13 zeigt. Für die Phase des Anforderungsmanagements sowie der Testfallspezifikation ist auffällig, dass hier in der Konformitätsprüfung noch wenig Standardisierung sowie Automatisierung vorzuherrschen scheint. Bei objektorientierter Modulspezifikation wird häufig Konformität zum Sprachstandard durch Prüf-Tools selbst geprüft bzw. die Konsistenz zwischen verschiedenen Modelldiagrammen sichergestellt. Einen weiteren Bereich wissenschaftlicher Arbeiten bilden die Überprüfungen der zustandsbasierten Modellierung auf Regeln für die Modellierung von Zustandsautomaten, aus denen später der Embedded-Code abgeleitet wird.

Zusammenfassend stellt man durch den Vergleich von Prüfwerkzeugen pro Phase fest, dass vereinzelte produktreife oder experimentelle Prüfwerkzeuge zwar vorhanden sind, jedoch diese immer nur einen lokalen Anwendungsbereich im kollaborativen Prozess abdecken (Kriterium: Geltungsbereich/Übergreifend nach Tabelle 13). Übergreifende, prozesslogisch verbindende Prüflösungen zur Umsetzung komplexerer, werkzeugübergreifender Prüfungen werden von den Prüfwerkzeugen, egal in welcher Phase, nicht unterstützt. Auch unter der Annahme, dass für ***n*** Artefakte auch ***n*** verschiedene Prüfwerkzeuge in den Prozessen eingesetzt werden könnten, wäre weder die Durchgängigkeit noch eine Praktikabilität der Konformitätsprüfung gewährleistet, da verschiedene statische Analyseverfahren mit unterschiedlicher Bedienung, Programmierbarkeit und Funktionsweise nicht effizient handhabbar sind.

Aus persönlichen Gesprächen kristallisierte sich die Wunschvorstellung heraus, dass die Unternehmen für ihre Vielzahl an eingesetzten Entwicklungswerkzeugen möglichst nur ein zentrales Prüfwerkzeug einsetzen möchten, welches sowohl das jeweilige Artefakt einzeln sowie auch werkzeugübergreifende Beziehungen mehrerer Artefakte überprüfen kann.

4 Konformität kollaborativer Artefakte

Nachdem im zweiten Kapitel die Grundlagen dieser Arbeit gelegt und im dritten Kapitel Prozesse, Methoden sowie der Stand der Wissenschaft und Technik vorgestellt wurden, können nun im vierten Kapitel das formalisierte Profil von Artefakten und Richtlinien aus prozessübergreifender Sicht festgelegt und die regelbasierte Konformitätsprüfung kollaborativer Artefakte erarbeitet werden. Aus den Einsatzgebieten der regelbasierten Konformitätsprüfung sowie der gängigen Verfahrensweisen werden zunächst Prozess- sowie Prüf-Anforderungen an die regelbasierte Konformitätsprüfung kollaborativer Artefakte erarbeitet. Danach werden Artefakte bzgl. Struktur und Inhalten genau definiert. Des Weiteren wird das Prinzip des logischen Artefakts konstruiert, damit eine übergreifende Konformitätsprüfung möglich wird. Sodann werden auch Richtlinien hinsichtlich eines industriellen Qualitätsmodells klassifiziert und zu Prozessphasen eingruppiert. Aus den Definitionen und den aufgestellten Anforderungen an die regelbasierte Konformitätsprüfung werden der Ansatz und der Lösungsweg zur regelbasierten Konformitätsprüfung kollaborativer Artefakte sowie der Geltungsbereich vorgestellt.

4.1 Anforderungen

Aus der Analyse von Prüfwerkzeugen, den im Kapitel 3.6 aufgestellten Kriterien sowie der Eingruppierung nach Prozessphasen können nun Anforderungen an die regelbasierte Konformitätsprüfung kollaborativer Artefakte gestellt werden.

Ein werkzeugübergreifendes Prüfverfahren muss folgenden Nutzen erbringen:

- Artefakt-Fehler und auftretende Folgefehler, die auch rückgängig gemacht werden müssten, werden durch dieses Verfahren deutlich und minimieren Risiken.
- Beim Review mehrerer Dokumente wird die Prüfzeit verkürzt. Produkte können wegen kürzerer Testphasen bei gleicher oder besserer Qualität schneller auf den Markt gebracht werden. Das bedeutet u. a. einen Vorsprung vor Konkurrenten.
- Die Bugfixing-Phase wird verkürzt und damit billiger, da Fehler wie „Vertipper“, die Verletzung von Namenskonventionen oder „Copy-Paste-Fehler“ oft die Ursachen für Integrationsprobleme im kollaborativen Prozess sind.

Ein hohes Qualitätsniveau bei der Entwicklung eingebetteter Systeme erfordert, dass dieses bereits in frühen Entwicklungsphasen, im Systementwurf eines eingebetteten Systems, Anwendung findet. Zudem sollen die Zeitaufwände für Prüfungen zur Erreichung des Qualitätsniveaus aus Kostengründen minimiert werden und die gesetzten Qualitätsziele bis zu hundert Prozent eingehalten werden können. Zur Qualitätssicherung interdisziplinärer Entwicklungsprozesse sowie zur stetigen Verbesserung von prozessspezifischen Qualitätszielen, wie durch Entwicklungsrichtlinien und Unternehmenskonventionen formuliert, ist eine regelbasierte Konformitätsprüfung von Artefakten erstrebenswert, falls diese eine frühzeitige, übergreifende und automatisierte Prüfung erlaubt. Insbesondere im interdisziplinären Umfeld, wenn Artefakte durch Uneinheitlichkeit der Elemente, einer

Menge von Dokumenten und Modellen, hinsichtlich eines oder mehrerer Merkmale, bei Herstellern und Zulieferern erstellt und ausgetauscht werden, ist dies erwünscht.

Bevor das Konzept einer kollaborativen Konformitätsprüfung an kollaborativen Artefakten hinreichend definiert werden kann, müssen Anforderungen an die Art der statischen Analyse, bezogen auf prozesslogisch abhängige heterogene Artefakte aufgestellt werden.

Die Kriterien zur Einordnung der Anforderungen stammen ursprünglich zum einen aus dem Prozessumfeld und zum anderen aus der geforderten Mächtigkeit, die ein Prüfverfahren besitzen muss, um kollaborativ prüfen zu können.

Die Anforderungen teilen sich in zwei Partitionen auf:

(1) Prozessanforderungen

(2) Prüfanforderungen

Nachfolgend werden beide Partitionen untersucht und daraus die allgemeinen Kriterien für eine kollaborative Konformitätsprüfung abgeleitet.

4.1.1 Prozessanforderungen

In der regelbasierten Konformitätsprüfung ist die erste Anforderung aus folgendem Prozess-Szenario (Abbildung 16) abgeleitet:

„*Ein Mitarbeiter (z. B. Entwickler, Tester) muss die Möglichkeit erhalten, direkt am Arbeitsplatz durch ein Computerprogramm aktuelle Konformitätsanforderungen von einem Server zu laden, um diese direkt in seiner Programmumgebung auf seinen Artefakten lokal auszuführen. Der Server hält für die Abteilung oder für das Unternehmen alle gültigen Regelkataloge in aktueller Fassung vor.*“

Durch diese Forderung wird eine *erste Konformitätsstufe* in den Entwicklungsprozess eingebaut, die den Mitarbeiter durch regelbasierte Prüfung und ggf. umgehende Korrektur zeitlich entlastet und vermeidet, dass im kollaborativen Prozess auf fehlerhaften Daten weitergearbeitet wird.

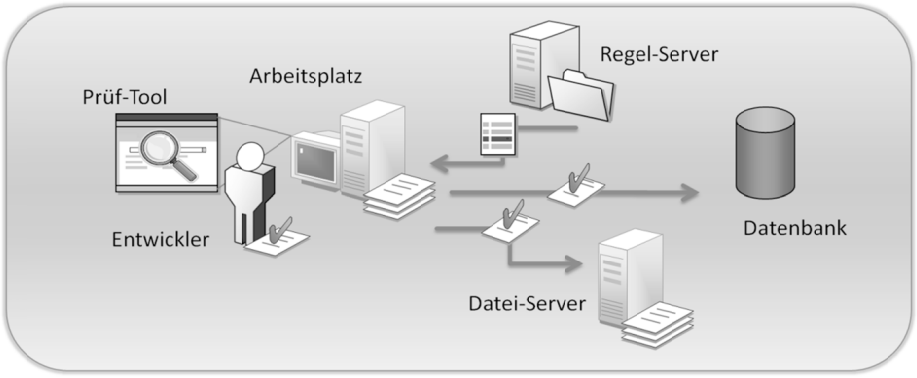


Abbildung 16: Prozesslokale Konformitätsprüfung (Compliance-at-the-Desk)

Erst wenn der Mitarbeiter die geprüften Artefakte freigibt, werden diese in die Datenbank (Produktdatenmanagementsystem o. Ä.) oder auf einem zentralen Datei-Server, ggf. mit Versionierung, bereits geprüft abgelegt. Alle prozesslokalen Aktivitäten und Vorgänge einer Konformitätsprüfung werden in [KRAN09] unter dem Begriff ‚Compliance-at-the-Desk‘ subsumiert.

Sind nun nachgelagerte Prozesse im übergreifenden Entwicklungsprozess involviert, basieren diese auf zuvor prozesslokal erstellten Artefakte. Die Prüfung, an einem bestimmten Zeitpunkt an prozesslogisch abhängigen Artefakten durchgeführt, wird durch den Begriff ‚Compliance Gate' bezeichnet (in der Literatur oftmals auch als ‚Quality Gate' bezeichnet, z. B. falls durch eine/mehrere Person(en) durchgeführt). Ein Compliance Gate beinhaltet eine regelbasierte Konformitätsprüfung kollaborativer Artefakte aus der Vorphase und kann in einer Prozesskette an unterschiedlichen Stellen erfolgen, wie die Abbildung 17 skizziert.

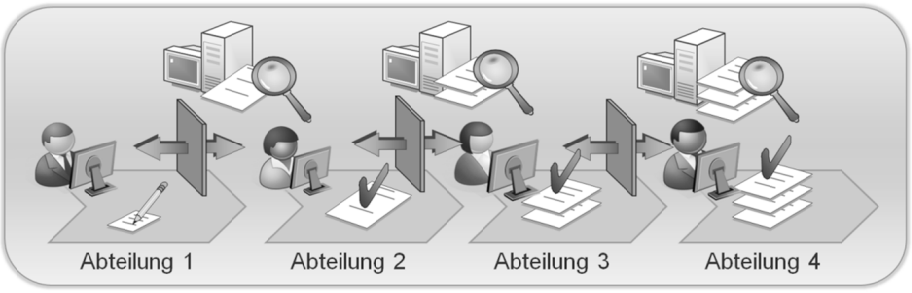


Abbildung 17: Prozessintegrierte Konformitätsprüfung (Compliance Gates)

Hierdurch wird sofort eine *zweite Konformitätsstufe* möglich, da die kollaborativen Artefakte und deren Zusammenhänge zwischen Prozess-Schnittstellen (Abteilungen) geprüft werden. Nur konsistente Artefakte werden zur weiteren Verarbeitung an den Compliance Gates für die nächste Phase geprüft und freigegeben. Erst die Platzierung der Compliance Gates an Informations- und Synchronisationspunkten von Daten sorgt für einen qualitativ abgesicherten Prozessverlauf. Hierbei ist ein wesentlicher vorbereitender Analyseschritt die Identifikation der kritischen Phasen bzw. der Prüf-Stellen in dem Entwicklungsprozess oder innerhalb von Projekten.

Die Steuerung und die Überwachung einer regelbasierten Konformitätsprüfung sind für die nachträgliche Analyse von Prozessen und die spätere Optimierung, bis hin zu Voraussagen (Predictions), wesentlich von Bedeutung. Mit ‚Compliance Monitoring & Control' kann demnach die *dritte Konformitätsstufe* durch eine autonome, externe Inspektion etabliert (Abbildung 18) werden.

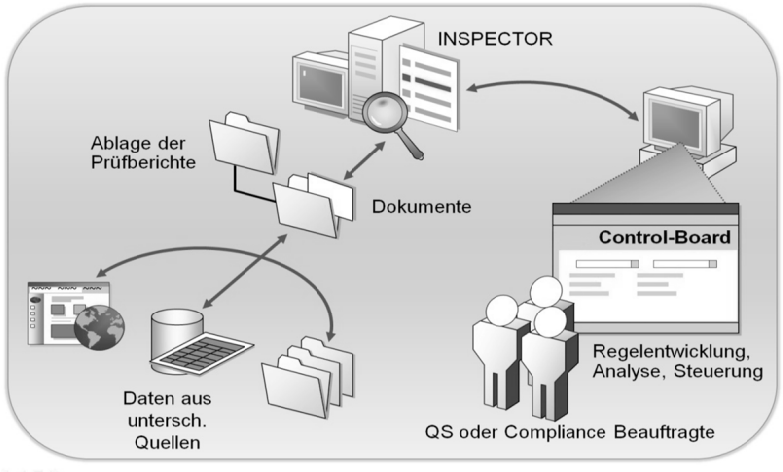


Abbildung 18: Autonome Konformitätsüberwachung (Monitoring & Control)

Ein serverbasiertes System zur Steuerung und zur Überwachung (hier als INSPECTOR bezeichnet) kann sodann zeitgesteuerte Prüfungen (z. B. über Nacht) durchführen, Prüfberichte der Konformitätsstufen 1 und 2 aus unterschiedlichen Quellen in einem ‚Quality Repository' (Datenbank) archivieren und erlaubt nachträglich eine datenbankgestützte Analyse über bestimmte Zeiträume hinweg.

Mittels regelbasierter Konformitätsüberwachung in der dritten Stufe werden vorbeugende Maßnahmen ermöglicht (Forderungen aus höheren CMMI Level), die auf Messdaten, Berechnungen sowie auf der Verwendung historischer Prüfdaten basieren. Die Anforderungen aus einer *präventiven Qualitätssicherung* ergänzen somit die *integrierte Qualitätssicherung* (Konformitätsstufe 1) sowie die nachgelagerte (Konformitätsstufe 2) und infolgedessen das ganzheitliche Qualitätsmanagement um folgende Aspekte:

- *Informationsschutz*, nicht alle Mitarbeiter dürfen alle Daten sehen.
- *Risikomanagement*, eine Einstufung von Risiken ist teils zwingend.
- *Informationsmanagement*, Nachvollziehbarkeit der Zusammenhänge.
- *Internes Kontroll- und Bewertungssystem*, automatisierte Prüfung.
- *Mitwirkungs- und Informationspflicht*, Meldewesen und Berichterstattung.

Eine besondere Anforderung aus Reifegradmodellen (höhere Stufen) liegt jedoch nicht ausschließlich in den vorbeugenden Maßnahmen, sondern ferner in der Möglichkeit, die Qualität im Voraus durch Personen (Compliance-Beauftragte) abschätzen, berechnen und für wiederkehrende Prozesse vorhersagen zu können. Aufgrund der Prognosen können dann präventive Maßnahmen begründet, neue Regeln entwickelt oder angepasst bzw. angewendet und nach der Implementierung ausgewertet werden, um die kontinuierliche Verbesserung von Entwicklungsprozessen zu ermöglichen. Die nachfolgende Tabelle 15 fasst die beschriebenen prozessgetriebenen Anforderungen zusammen.

Tabelle 15: Prozessanforderungen

	Anforderung	***Beschreibung***
(1)	Etablierung der ersten Konformitätsstufe	Die regelbasierte Konformitätsprüfung ist durch ein werkzeuggestütztes Verfahren prozesslokal am Arbeitsplatz an einem Artefakt, beliebig wiederholbar, durchführbar.
(2)	Etablierung der zweiten Konformitätsstufe	Die regelbasierte Konformitätsprüfung wird zu einem bestimmten Zeitpunkt prozessübergreifend durch ein werkzeuggestütztes Verfahren in der Prozesskette an kollaborativen Artefakten durchgeführt.
(3)	Etablierung der dritten Konformitätsstufe	Die regelbasierte Konformitätsprüfung wird autonom (Ereignis/Zeitplan gesteuert) durch einen Server-Dienst prozessübergreifend durch ein werkzeuggestütztes Verfahren an kollaborativen Artefakten durchgeführt. Prüfprotokolle werden archiviert und für statistische Auswertungen gespeichert.

4.1.2 Prüfanforderungen

Die Anforderungen an eine regelbasierte Prüfung ergeben sich im Kontext der modellbasierten Entwicklung eingebetteter Systeme aus den im Kapitel 2 und im Kapitel 3 vorgestellten Artefakten und den Richtlinien. Um Richtlinien als Bedingungen formulieren zu können, bedarf es einer Richtliniensprache. Mit dieser sollen invariante Bedingungen, Vor- und Nachbedingungen, Abfrageausdrücke, Ableitungsausdrücke sowie einfache Berechnungen beschreibbar sein. Dies ist ein Thema des fünften Kapitels. Die gewählten Sprachen für die Regelimplementierung werden im Kapitel 7 untersucht.

Zunächst werden die allgemeinen Prüfanforderungen für die regelbasierte Konformitätsprüfung basierend auf nicht-funktionalen Qualitätscharakteristiken sowie auf der Basis struktureller Eigenschaften heterogener Artefakte in der Tabelle 16 aufgestellt.

Tabelle 16: Prüfanforderungen

	Anforderung	***Beschreibung***
(1)	Attributvergleich	Eine regelbasierte Konformitätsprüfung muss prüfen können, ob ein Datum oder mehrere Daten eines/mehrerer Artefakt/e konform zu einer Vorgabe ist/sind. In Erweiterung zu kollaborativen Artefakten muss dies beinhalten, dass zwei oder mehr Artefakte auf Gleichheit ihrer Daten prüfbar sind. Die Gleichheit beschränkt sich dabei auf Wert- oder Typgleichheit des Datums.
(2)	Strukturvergleich	Eine regelbasierte Konformitätsprüfung muss prüfen können, in welcher Struktur (Reihenfolge) Daten eines Artefakts konform zu einer Vorgabe sind. In Erweiterung zu kollaborativen Artefakten muss dies beinhalten, dass zwei oder mehr Artefakte auf Gleichheit ihrer internen Struktur prüfbar sein müssen. Die Heterogenität in Persistenz des Artefakts ist dabei nicht ausschlaggebend.
(3)	Komplexitätserkennung	Eine regelbasierte Konformitätsprüfung muss prüfen können, in welcher Komplexität Daten eines Artefakts konform zu einer Vorgabe sind. In Erweiterung zu kollaborativen Artefakten muss dies beinhalten, dass zwei oder mehr Artefakte auf Komplexität ihrer internen Daten und deren Struktur prüfbar sein müssen. Dabei sollen Prüfprotokolle archiviert und für statistische Auswertungen berechnet und gespeichert werden können.
(4)	Erkennung von Anomalien	Eine regelbasierte Konformitätsprüfung muss prüfen können, ob in einem Artefakt ein unerwünschter Zustand (falsche oder fehlende Anordnung eines Datums) vorliegt, der vom Erwarteten (der Vorgabe) abweicht. In Erweiterung zu kollaborativen Artefakten muss dies beinhalten, dass zwei oder mehr Artefakte auf Anomalie prüfbar sein müssen.

Durch die Anforderungen können typische, auf anderem Wege teilweise nur schwer zu findende Fehler im kollaborativen Prozess gefunden und identifiziert werden, bevor diese publik werden, was wiederum höheres Vertrauen in der Prozesskette schafft. Alleine durch das Nachdenken bei der Definition von Richtlinien entsteht eine größere Klarheit über notwendige Zusammenhänge, was sich wiederum positiv auf die Qualität der Arbeitsergebnisse auswirkt, was in ähnlicher Weise für die modellbasierte Entwicklung an sich gilt und dort eine weitgehend anerkannte Tatsache ist.

4.2 Kollaborative Artefakte

Im Kapitel 3.8 wurden ausgewählte Artefakte aus dem modellbasierten Entwicklungsprozess beispielhaft vorgestellt und im Kapitel 2.1 eine Definition gegeben, die prinzipiell klärt, was ein Artefakt ist. Nun ist es erforderlich, genau zu definieren, welche Struktur ein Artefakt aufweisen muss und was genau ein ‚kollaboratives Artefakt' ist.

4.2.1 Herleitung

In der Mathematik kennt man zur Beschreibung einer Struktur zunächst den Begriff des Graphen. An der Stelle soll uns die aus der Mathematik stammende Graphentheorie unterstützen, die einen Graphen (angelehnt an [SCHN91]) wie folgt definiert:

Graph *(engl. graph)* **(4.1)**
„Ein *Graph* ***G*** besteht aus einer Menge von Punkten, zwischen denen Linien verlaufen. Die Punkte nennt man *Knoten*, die Linien nennt man meist *Kanten*, in einigen Fällen auch Bögen. Die Form der Knoten und Kanten ist im Allgemeinen nicht von Bedeutung. Die Knoten und die Kanten können auch mit Namen versehen sein, dann spricht man von einem *benannten Graphen*."

Beispiel:

$\boldsymbol{E_G}$ = {"A", "B", "C", "D", "E", "F"} sei die Menge der Buchstaben "A" bis "F".

Demzufolge ist

$\boldsymbol{K_G}$ = { ("A", "B"), ("B", "C"), ("D", "B"), ("C", "E"), ("E", "D"), ("E", "F")}.

Dann ist ($\boldsymbol{E_G}$, $\boldsymbol{K_G}$) ein ***Graph***. Man kann ihn wie folgt (Abbildung 19) veranschaulichen:

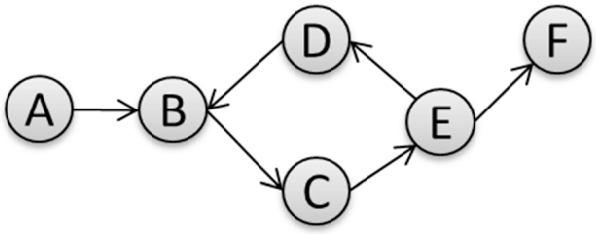


Abbildung 19: Gerichteter Graph

Der entstandene Graph $\boldsymbol{G_G}$ ist somit das Tupel ($\boldsymbol{E_G}$, $\boldsymbol{K_G}$), wobei $\boldsymbol{E_G}$ eine beliebige Menge und $\boldsymbol{K_G}$ eine *Relation* darstellt. Eine Relation wiederum ist definiert als eine Teilmenge des kartesischen Produktes $\boldsymbol{E_G} \times \boldsymbol{E_G}$. Letzteres schließlich, $\boldsymbol{E_G} \times \boldsymbol{E_G}$, ist definiert als die Menge

aller Paare (***a, b***) von Elementen aus $\boldsymbol{E_G}$. In gerichteten Graphen oder orientierten Graphen werden Kanten statt durch Linien durch Pfeile gekennzeichnet, wobei der Pfeil vom ersten zum zweiten Knoten zeigt. In sogenannten Multigraphen können zwei Knoten auch durch mehrere Kanten verbunden sein, was in einfachen Graphen nicht erlaubt und auch nicht notwendig ist.

Was bedeutet dies nun in Bezug auf die Festlegung unserer Artefakt-Struktur? Über die Beschaffenheit von $\boldsymbol{E_G}$ macht der Graphen-Begriff keine Aussage außer der, dass es sich um eine beliebige Menge von *benannten Punkten* handelt, welche in Relation stehen. Dies muss in unser zu entwickelndes Konzept übertragen und genauer festgelegt werden.

Als zusätzliche Forderung wird daher festgelegt, dass die *benannten Punkte* genau die *Eigenschaften* eines Artefakts darstellen, aus denen ein Artefakt beschaffen ist. Im Vergleich zum Paradigma der Objektorientierung ist eine Objekteigenschaft (Attribut) durch einen *Wert* (Value) und einen *Typ* (Type) bestimmt. Dass die einzelnen Artefakt-Eigenschaften jeweils aus einem *Attribut-Werte-Paar* bestehen, wird nun durch den Begriff ‚Datum' wie folgt eingeführt:

Datum, Daten ***(engl. value, values)*** **(4.2)**
„Ein *Datum* ***d*** ist ein ***n***-Tupel von Eigenschaften $\boldsymbol{a_0, a_1, ..., a_n}$, genannt *Attribute* des Datums. Ferner ist jedes Attribut einem primären Datentyp ***a*T** zugeordnet (typisiert) und besitzt einen textuellen Bezeichner ***a*B**. Eine Datenmenge (*Daten*) wird kurz als ***D*** geschrieben."

Immer lässt sich auch der primäre Datentyp zu einem Attribut bestimmen, daher ist dieser in der Definition aufgenommen und später bedeutend, da Entwicklungsrichtlinien die konforme Verwendung von Datentypen innerhalb eines Artefakts fordern können.

Als primäre Datentypen ***a*T** definieren wir nach Tabelle 17.

Tabelle 17: Datentypen der Artefakt-Attribute

I.	*Buchstaben* (*char*)	, z. B. ‚a', ‚b', ‚c'
II.	*Zeichenketten* (*string*)	, z. B. ‚Zeichenkette'
III.	*Wahrheitswerte* (*bool*)	, z. B. ‚true', ‚false'
IV.	*Ganze Zahlen* (*int, long*)	, z. B. ‚1', ‚5', ‚9'
V.	*Dezimale Zahlen* (*float, double*)	, z. B. ‚1.2', ‚3.14'
VI.	*Bytefolgen* (*n byte*)	, z. B. ‚00110'
VII.	*Datum* (*datetime*)	, z. B. ‚2009-11-02T16:25:42'
VIII.	*Nicht festgelegt* (*object*)	, z. B. ‚null'

Eine Sonderstellung nimmt das nicht typisierte Objekt (*object*) ein. Sowohl die primitiven Verweis- als auch die Wertetypen werden von der Basisklasse *object* abgeleitet. In Fällen, in denen ein Wertetyp als *object* fungieren muss, wird ein sogenannter Container (*Box*), der den Wertetyp als Verweisobjekt erscheinen lässt, dem Heap zugeordnet. Der Wert des Wertetyps wird kopiert. Dieser Vorgang wird als *Boxing* und der umgekehrte Vorgang als *Unboxing*

bezeichnet. Durch das *Boxing* und *Unboxing* kann ein beliebiger Datentyp als *object* behandelt werden.

Nun lässt sich das Artefakt exakter definieren:

Artefakt *(engl. artifact)* **(4.3)**
„Ein *Artefakt* ***A*** besteht aus Daten und ist ein Paar (***D, K***), bestehend aus einer endlichen, nicht-leeren Menge ***D*** von Daten und einer Relation ***K*** in ***D***."

Jedes Artefakt $\boldsymbol{A} = (\boldsymbol{D}, \boldsymbol{K})$ ist damit ein gerichteter *Graph* im Sinne der Graphentheorie der Mathematik mit den Knoten ***D*** und den Kanten ***K***. Die Begriffe der Graphentheorie, wie beispielsweise *Vorgänger* (Predecessor) und *Nachfolger* (Descendant), sind damit definiert und unmittelbar auf Artefakte und den enthaltenen Daten anwendbar.

Im Kapitel 3.8 wurden ausgewählte Artefakte aus dem modellbasierten Entwicklungsprozess beispielhaft vorgestellt. Auffällig ist, dass die Datenmodelle der Artefakte meist eine hierarchische Struktur aufweisen. Typischerweise sind gerade XML-basierte Dokumente in einer Baumstruktur persistent gespeichert und folgen sogar einem XML-Schema (Metamodell). Aus dieser Beobachtung soll daher nachfolgend auch noch eine weitere Eigenschaft hinsichtlich der Beschaffenheit eines Artefakts gelten:

Hierarchisches Artefakt *(engl. hierarchically artifact)* **(4.4)**
„Ein Artefakt $\boldsymbol{A_H} = (\boldsymbol{D}, \boldsymbol{K})$ heißt *hierarchisch*, wenn folgende Eigenschaften gelten:

(1) Es gibt genau einen Knoten ***w*** aus ***D***, der keinen Vorgänger hat. Dann heißt ***w*** die Wurzel (englisch: *root*) des Baumes.

(2) Jeder Knoten ***k*** aus ***D*** mit Ausnahme von ***w*** hat genau einen Vorgänger.

(3) Für jeden von der Wurzel ***w*** verschiedenen Knoten ***k*** existiert eine Folge mit $\boldsymbol{w} = \boldsymbol{k}_0, \boldsymbol{k}_1, \ldots, \boldsymbol{k}_n = \mathbf{k}$ $(\boldsymbol{n} \geq \mathbf{1})$ von Knoten, bei der $\boldsymbol{k}_i$ der Nachfolger von $\boldsymbol{k}_{i-1}$ ist $(1 \leq \boldsymbol{i} \leq \boldsymbol{n})$."

Durch die Folge von Knoten gibt es ab der Wurzel zu jedem Knoten genau einen *Pfad*.

Ein hierarchisches Artefakt heißt auch *baumartiges Artefakt* bzw. *Artefakt-Baum*. Die Abbildung 20 zeigt ein Beispiel für ein hierarchisches Artefakt.

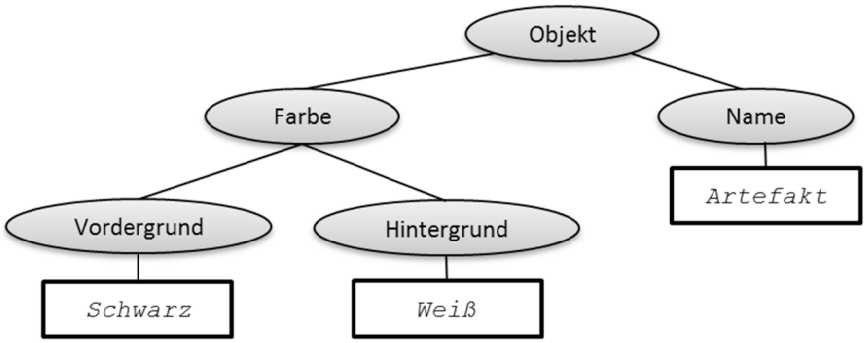


Abbildung 20: Struktur eines hierarchischen Artefakts

Die *Traversierung* ist in der Graphentheorie der Name für Verfahren, bei der jeder Knoten und jede Kante eines baumförmigen Graphen genau einmal besucht wird, d. h. durchläuft ein Algorithmus die Knoten eines Artefakts, so spricht man von *traversieren* (parsen). Zur Konformitätsprüfung ist immer eine Traversierung notwendig, um eine bestimmte Menge an Daten eines Artefakts zu untersuchen bzw. nur ein bestimmtes Attribut aufzufinden.

4.2.2 Definition

Wie im Kapitel 2.2 festgestellt wurde, haben aus der prozessorientierten Sicht hierarchische Artefakte verschiedene prozesslogische Querbeziehungen zueinander. Beispielsweise ist eine Testspezifikation einem bestimmten Funktionsmodell logisch zugeordnet, die erlaubte Verwendung von Datentypen oder Fachworten ist in einem separaten Glossar verzeichnet, oder hierarchische Artefakte haben im Entwicklungsprozess durch mehrere aufeinander aufbauende Arbeitsschritte mehrerer Akteure (Personen) oder Programme (Automatismen) logische Bezüge zueinander, z. B. aus einem Implementierungsmodell generierter Code. Durch die kollaborativ im Prozess bearbeiteten, erstellten, veränderten oder weitergegebenen Dateien existieren eben virtuelle Beziehungen, die oftmals nicht in dem Artefakt selbst als Information enthalten sind. Diese beobachtete Besonderheit soll folgende Definition eines ‚kollaborativen Artefakts' ausdrücken:

Kollaboratives Artefakt ***(engl. collaborative artifact)*** **(4.5)**
„Im Kontextbezug eines Prozesses $\boldsymbol{P}$ nennt man ein hierarchisches Artefakt $\boldsymbol{A_H}$ ein *kollaboratives Artefakt* $\boldsymbol{A_K}$, wenn folgende charakteristischen Merkmale von $\boldsymbol{A_H}$ vorliegen:

(1) Ein Datum ist logisch abhängig von einem Datum eines anderen Artefakts.

(2) Ein Artefakt steht virtuell im prozesslogischen Bezug zu weiteren Artefakten.

Es liegt genau dann ein kollaboratives Artefakt vor, wenn mindestens eines der aufgeführten Merkmale (1) oder (2) zutrifft."

Alle sich in einem beliebigen Prozess befindlichen kollaborativen Artefakte stellen somit eine nicht leere Menge von hierarchischen Artefakten dar, für diese dann Anforderungen aus Entwicklungsrichtlinien gelten können.

Ein Prozess umfasst einen Ausschnitt der realen Welt und aller darin vorkommenden Daten. Hierbei spricht man von dem *Diskursbereich*. Nun muss dieser Ausschnitt auf die zugrunde liegende IT-Infrastruktur und alle damit verbundenen kollaborativen Artefakte abgebildet werden. Übertragen auf die IT-Infrastruktur ist der Diskursbereich somit die Menge aller in einem informationsverarbeitenden Prozess logisch referenzierten kollaborativen Artefakte. Dieser Begriff soll als ‚Prüfraum' eingeführt werden:

Prüfraum ***(engl. scope of examination)*** **(4.6)**
„Der durch einen Prozess $\boldsymbol{P}$ festgelegte Ausschnitt der realen Welt (Diskursbereich) referenziert eine endliche Menge aller dem Prozess logisch zugehörigen kollaborativen Artefakte $\boldsymbol{A_K}$ ($\boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_n \mid \boldsymbol{n} \in |\mathrm{R}$). Die aus dem Diskursbereich zu $\boldsymbol{P}$ resultierende Menge spannt den *Prüfraum* $\boldsymbol{\Omega}$ zu $\boldsymbol{P}$ auf."

Beispielsweise kann ein Prozess zur Angebotserstellung ein Anschreiben, das Angebot und die dazugehörige Kalkulation als kollaborative Artefakte logisch referenzieren. Eine dazugehörige Rechnung und ein Lieferschein sind kollaborative Artefakte eines anderen Prozesses und gehören nicht mehr zum Diskursbereich der Angebotserstellung.

Typischerweise adressiert eine Entwicklungsrichtlinie nicht unbedingt alle durch einen Prozess referenzierten kollaborativen Artefakte. Beispielsweise kann eine Richtlinie nur die Konsistenz von Angebot zu der zugehörigen Kalkulation einfordern und die Form des Anschreibens beliebig offen lassen. Für die Konformitätsprüfung ist somit immer nur ein zeitbezogener Ausschnitt (Snapshot) kollaborativer Artefakte in einem Prozess von Relevanz. Eine Prüfung wird daher jeweils immer auf einer endlichen Teilmenge des eigentlichen Prüfraums $\boldsymbol{P}$ ausgeführt. Dies wird im Folgenden als ‚Prüfaufbau' verstanden und wird nun definiert:

Prüfaufbau ***(engl. composition of examination)*** **(4.7)**
„Die abgeschlossene Menge $\Delta \subseteq \boldsymbol{\Omega}$ eines beliebigen Prüfraums $\boldsymbol{\Omega}$ nennt man *Prüfaufbau*, in dem alle kollaborativen Artefakte $\boldsymbol{A_K}$ $(\boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_x \mid \boldsymbol{x} \leq \boldsymbol{n})$ für eine regelbasierte Konformitätsprüfung zu einem bestimmten Zeitpunkt enthalten sind."

Der Prüfaufbau zieht ein notwendiges Kriterium für die Durchführung einer regelbasierten Konformitätsprüfung nach sich:

➢ **Notwendiges Kriterium:** $\Delta \neq \varnothing$ (*Der Prüfaufbau darf zur Durchführung einer regelbasierten Konformitätsprüfung nicht leer sein.*)

Für diesen speziellen Fall eines leeren Prüfaufbaus wird festgelegt, dass die Erfüllung der Anforderung aus einer Richtlinie nicht nachgewiesen werden kann und somit die Konformität nicht gegeben ist.

4.3 Logisches Artefakt

In werkzeugübergreifenden Prozessen und den darin geltenden Richtlinien existieren logische Konstrukte, die zunächst rein imaginär sind – oftmals auch bedingt durch eine produktorientierte Terminologie. Eine *Basisfunktion* oder eine *Komfortfunktion* ist solch ein fiktives Konstrukt aus einem Prozess und charakterisiert bereits bestimmte Eigenschaften des eingebetteten Systems. Sie hat in frühen Entwicklungsphasen noch keine konkrete Artefakt-Ausprägung, sehr wohl aber eine festgelegte Bedeutung im Prozess. Abgebildet auf die kollaborativen Artefakte, wie das beschriebene Konstrukt *Basisfunktion*, besitzen diese selbst kein Datum für eine *Basisfunktion*, sondern nur eine wohldefinierte Struktur von Daten oder eine bestimmte Eigenschaft oder Abhängigkeit im Artefakt-Kontext. Das fiktive Konstrukt entsteht vielmehr aus einer Kombination von Daten in mehreren kollaborativen Artefakten, die durch mehrere Werkzeuge festgelegt oder bearbeitet werden.

Beispiel:

Zur Verdeutlichung sei folgendes Beispiel gegeben: Eine *Basisfunktion* aus dem gewählten Beispielszenario (Kapitel 8) ist hierbei logisch charakterisiert durch ein kollaboratives Artefakt (DOORS-Anforderungsobjekt) mit der Eigenschaft „*Object Heading*" = „*Basisfunktion <Name der Funktion>*". Sämtliche untergeordneten Daten (Anforderungsobjekte) müssen die Überschriften „*Beschreibung*", „*Eingaben*", „*Ausgaben*" usw. in einer festgelegten Reihenfolge beinhalten. Die „*Eingaben*", „*Ausgaben*" sind wiederum benannte

Signalenden (*Inports*, *Outports*) in ML/SL/SF und dort modelliert. Dieselben „*Eingaben*“, „*Ausgaben*“ sind wiederum die Schnittstellen für den modellbasierten Test und die zugehörige Testspezifikation in CTE/XL. Es besteht somit eine wechselseitige, logische Abhängigkeit zwischen Anforderung, Modell und Test. Erst die komplette Struktur spezifischer Anforderungsobjekte und deren werkzeugübergreifender Eigenschaften erzeugt dann eine „*Basisfunktion*“ oder eine „*Komfortfunktion*“.

Prozesse führen demnach fiktive Konstrukte ein, welche semantische Beziehungen, eine definierte Artefakt-Struktur oder werkzeugübergreifende Eigenschaften von kollaborativen Artefakten besitzen. Durch diese Abstraktion kommen sie in den kollaborativen Artefakten nicht zwangsläufig als Information vor. Es sind vielmehr einige Ausschnitte einzelner oder Zusammenfassungen verschiedener kollaborativer Artefakte. Wir nennen solche fiktiven Konstrukte *logische Artefakte* und führen sie wie folgt ein:

Logisches Artefakt ***(engl. logical artifact)*** **(4.8)**
„Bezogen auf einen Prozess ***P*** nennt man ein kollaboratives Artefakt A_K immer dann ein *logisches Artefakt* A_I, wenn mehrere charakteristische Merkmale kollaborativer Artefakte eine logische Relation ***R*** in ***P*** zum Artefakt bilden. Dies ist immer dann der Fall, wenn:

(1) eine Artefakt-übergreifende Strukturbeschreibung die logische Relation bildet,

(2) Artefakt-übergreifende Daten die logische Relation bildet.

Logische Artefakte sind demnach eine kombinierte Instanz von Struktur und Daten, die in mehreren kollaborativen Artefakten vorhanden sind. Es liegt genau dann ein *logisches Artefakt* A_I vor, wenn mindestens eines der Merkmale (1) oder (2) zutrifft.“

Wird demnach ein logisches Artefakt A_I in einem Prozess ***P*** betrachtet, ist dies abgeleitet aus einer gewählten Teilmenge Δ kollaborativer Artefakte aus dem Prüfraum Ω.

➢ **Hinreichendes Kriterium:** *Die logischen Artefakte bilden immer den Prüfaufbau.*

Zur Verdeutlichung der oben genannten Definition sei ein weiteres Beispiel aufgeführt.

Beispiel:

Gegeben sind zwei kollaborative Artefakte ***A*** und ***X*** in einem Prüfaufbau:

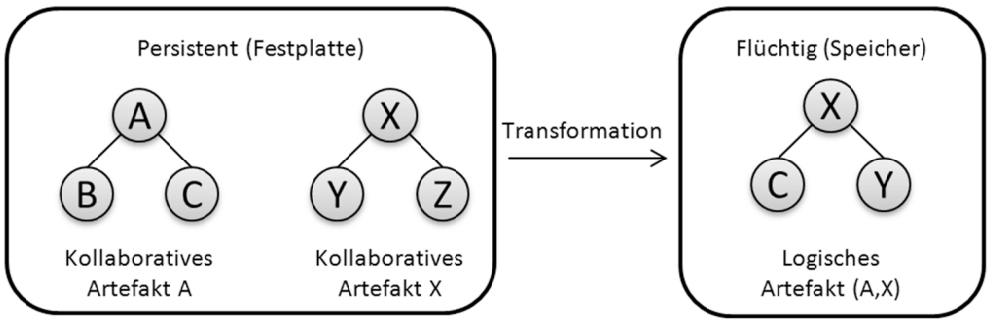


Abbildung 21: Bildung eines logischen Artefakts aus dem Prüfaufbau

Durch direkte oder indirekte Transformation lassen sich die Datenmodelle (Metamodelle) der einzelnen kollaborativen Artefakte in eine oder mehrere logische Instanz(en) $\boldsymbol{I}$ überführen, welche ein kombiniertes Datenmodell ergeben. Die Abbildung 21 zeigt die Aggregation in ein logisches Artefakt $\boldsymbol{A_I} = (\boldsymbol{A}, \boldsymbol{X})$ mittels indirekter Transformation.

Im Beispiel kann demnach ein neues Metamodell die Grundlage des logischen Artefakts bilden. Dieses Metamodell kann die Vereinigungsmenge aus den Daten-Definitionen und den Strukturen der beiden kollaborativen Artefakte bilden. Zusätzlich können aber auch neue, virtuelle Konstrukte oder Beziehungen in dem Metamodell des logischen Artefakts aufgenommen werden, die eigentlich nicht in den kollaborativen Artefakten auftreten.

Somit lassen sich durch verschiedenartige Transformationstechniken folgende drei mögliche Fälle zur Bildung eines logischen Artefakts realisieren:

(1) $\boldsymbol{A_K} \cup \boldsymbol{A_I} \mid \boldsymbol{A_K} \in \Delta$: Ein logisches Artefakt $\boldsymbol{I}$ kann identisch mit dem kollaborativen Artefakt $\boldsymbol{A}$ sein, welches persistent auf der Festplatte vorliegt und dann als logisches Artefakt instanziiert wird. In diesem Falle handelt es sich um eine *Kopie* im Hauptspeicher eines Rechners oder in einem Repository (Modelldatenbank).

(2) $\boldsymbol{A_K} \subseteq \boldsymbol{A_I} \mid \boldsymbol{A_K} \in \Delta$: Ein logisches Artefakt kann Teile des kollaborativen Artefakts enthalten. Dann bildet es virtuell einen *Ausschnitt* des ursprünglichen persistenten Artefakts als Datenmodell im Hauptspeicher eines Rechners oder in einem Repository (Modelldatenbank) ab.

(3) $\boldsymbol{A_{K1}} \subseteq \boldsymbol{A_I} \wedge \boldsymbol{A_{K2}} \subseteq \boldsymbol{A_I} \mid \boldsymbol{A_{K1}}, \boldsymbol{A_{K2}} \in \Delta$: Ein logisches Artefakt (z. B. $\boldsymbol{A}$, $\boldsymbol{X}$) kann als *Vereinigung* aus ursprünglichen kollaborativen Artefakten instanziiert werden. Dann bildet es ein integriertes Datenmodell im Hauptspeicher eines Rechners oder in einem Repository (Modelldatenbank) ab.

Durch die Erzeugung eines logischen Artefakts als ein virtuelles und temporäres Datenmodell können auf diese Weise prozesslogische Beziehungen und abstrakte Konstrukte gebildet und schließlich hinsichtlich Konformität auf abstrakterer Ebene abgeprüft werden.

Satz: *„Durch die Abstraktion von reellen Daten und Strukturen der kollaborativen Artefakte in ein virtuelles, temporäres Datenmodell kann eine regelbasierte Konformitätsprüfung zu einem Zeitpunkt werkzeugübergreifend möglich werden.“*

Dieser Ansatz ist angelehnt an das MDA-Paradigma, welches plattformspezifische Aspekte und plattformunabhängige Aspekte trennt, gleichwohl die Transformation beider zulässt, ohne die Bezüge zu verlieren.

Zusätzlich können durch einen Abstraktionsschritt mittels Metamodellierung eigene, prozesslogische Semantiken und Beziehungen im Metamodell des logischen Artefakts definiert werden, die sogleich auch eine Referenz auf die Datenmodelle der kollaborativen Artefakte bilden.

Es stellt sich abschließend die Frage, wie ein logisches Artefakt für eine regelbasierte Konformitätsprüfung methodisch und technisch konkret gebildet werden kann. Hierzu wird in dieser Arbeit die *Metamodellierung* verwendet, um logische Artefakte beschreiben und für die automatisierte Prüfung auch technisch abbilden zu können. Das Kapitel 6 stellt hierfür die nötige Methodik und das Kapitel 7 die technische Realisierung (Implementierung) vor.

4.3.1 Virtuelle Sicht auf Daten

In der Welt der relationalen Datenbanken kennt man für imaginäre Konstrukte der Prozesswelt bzw. der temporär transformierten, zweckgebundenen Darstellung von Daten den Begriff *Sicht*. Relationale Datenbanksysteme bieten Sichten an, mit deren Hilfe Prozesse die für sie relevanten Daten einer Datenbank nach ihren wesentlichen Bedürfnissen selektieren, kombinieren und umstrukturieren können (vgl. auch [KEMP99; KEMP07]). Eine Sicht auf eine Datenbank ist somit eine ausgewählte Menge von Daten, die von den Basisdaten der Datenbank abgeleitet sind. Sichten schränken zunächst nur die Datenmengen ein, sie können aber nicht die unterschiedlichen Darstellungsarten in Applikationsprogrammen und Datenbanksystemen überbrücken – hierzu dienen technische Mittel wie beispielsweise *temporäre Tabellen*.

In relationalen Datenbanksystemen wird mit der SQL (Structured Query Language, vgl. Kapitel 2.5.3) als Datendefinitionssprache eine Sicht durch eine SQL-Anfrage definiert. In Analogie zu den relationalen Datenbanksystemen wird folgende Annahme auch für die Bildung des logischen Artefakts übernommen: Bei einer Abstraktion (Kapitel 6.4) hin zu logischen Artefakten ermöglichen Transformationsvorgänge eine ‚Sicht auf kollaborative Artefakte', welche unterschiedliche Ergebnisse liefern kann:

- Eine *Selektionssicht* filtert bestimmte Daten aus den Artefakten heraus.
- Eine *Projektionssicht* filtert bestimmte Datenstrukturen aus den Artefakten heraus.
- Eine *Verbundsicht* verknüpft Daten mehrerer kollaborativer Artefakte.
- Eine *Aggregationssicht* wendet Aggregationsfunktionen (wie Minimum, Maximum, Summe, Mittelwert usw.) auf Daten der Artefakte an.

Eine Sicht aus der Datenbanktechnik lässt sich mit einer Transformation in ein logisches Artefakt vergleichen und wurde dabei entweder aus einem Artefakt oder aus mehreren kollaborativen Artefakten heraus gebildet. Entwicklungsrichtlinien, welche werkzeugübergreifende Konsistenzbeziehungen einfordern, können somit als virtuelle Beziehung in ein logisches Artefakt abgebildet werden. Zusätzlich ist die Abstraktion der Problembeschreibung in Form von Anforderungstext, Modellen oder Testfällen und von den eingesetzten Entwicklungswerkzeugen möglich. Dies sagt jedoch nicht aus, dass werkzeugübergreifende Konsistenzbeziehungen unabhängig von den eingesetzten Werkzeugen sind. Logische Artefakte bilden nur die Möglichkeit, diese übergreifenden Beziehungen effizient darzustellen und von der Heterogenität der realen Welt abstrahieren zu können.

Satz: *„Das logische Artefakt stellt eine logische Sicht auf kollaborative Artefakte dar. Es werden nur die für eine werkzeugübergreifende Konformitätsprüfung relevanten Daten und Strukturen innerhalb der Sicht benötigt. Zusätzliche bzw. in den kollaborativen Artefakten fehlende Informationen zu prozesslogischen Konstrukten und Beziehungen können im logischen Artefakt abgebildet werden und stehen für die regelbasierte Konformitätsprüfung temporär zur Verfügung."*

4.4 Ableitung von Richtlinien zu Regeln

Nachdem eine konkrete Definition für ein hierarchisches, kollaboratives und logisches Artefakt gegeben wurde, ist es nun notwendig, die Richtlinien konkreter zu analysieren und zu beschreiben. Wie bereits eingeführt wurde, versteht man im Allgemeinen unter

Richtlinien (Vorschriften) unternehmensinterne, abstrakte oder sehr konkrete Handlungsanweisungen zur Befolgung im Prozess. Gelten Richtlinien speziell für einen bestimmten Prozess, z. B. für die Entwicklung, spricht man von Entwicklungsrichtlinien. Noch konkreter wird eine Einschränkung vorgenommen, wenn sich Richtlinien beispielsweise nur auf die konforme Erstellung von Modellen, die Modellierung, beziehen; in diesem Fall spricht man von Modellierungsrichtlinien. Es scheint nun zweckmäßig, konkrete Definitionen zu den spezifischen Richtlinien zu geben. Die nachfolgenden Kapitel definieren Richtlinien und daraus abgeleitete Regeln.

4.4.1 Definitionen

Richtlinien stellen Anforderungen an die Beschaffenheit kollaborativer Artefakte an Inhalt, Form und Struktur. Da die hier betrachteten Richtlinienanforderungen natürlich sprachlich formuliert sind, bedarf es einer Formalisierung zur Präzisierung der enthaltenen Anforderung. Eine Regel wird demnach wie folgt verstanden:

Regel *(engl. rule)* **(4.9)**
„Eine *Regel* Ψ ist der formalisierte Ausdruck aus einer natürlich sprachlich dokumentierten Richtlinienanforderung."

Die Formalisierung hilft zunächst, die Anforderung aus der Richtlinie zu präzisieren. Dabei kann eine Richtlinieanforderung auch in mehrere Regeln partitioniert werden.

Zur automatisierten Konformitätsprüfung muss zudem der formalisierte Regel-Ausdruck als computerausführbarer Prüfalgorithmus weiterentwickelt werden. Ein Regel-Algorithmus ist im Allgemeinen eine Rechenvorschrift, die aus mehreren elementaren Anweisungen (von Programmiersprachen), Befehlen (bei Maschinensprachen), den sogenannten *Instruktionen* besteht. Instruktionen müssen in einer bestimmten Reihenfolge ausgeführt werden und nach einer endlichen Anzahl von Schritten zu einem Ergebnis führen. In diesem Fall spricht man von der Transformation einer Richtlinie in eine *ausführbare Regel* bzw. einen *Regel-Algorithmus*, also einem Algorithmus der die Erfüllung der Anforderung auf kollaborativen Artefakten überprüft.

Ausführbare Regel *(engl. executable rule)* **(4.10)**
„Eine Regel Ψ heißt *ausführbar*, wenn sich eine Regel als terminierender Algorithmus formulieren und sich dieser durch ein Softwareprogramm ausführen lässt."

Der in dieser Definition enthaltene Begriff ‚Softwareprogramm' ist im gebräuchlichen Sinne zu verstehen. Ein *Programm* ist ein für den Computer in einer formalen Sprache aufbereiteter Algorithmus. Die verwendete Sprache wird dabei Programmiersprache genannt, welche durch ihre Syntax und Semantik spezifiziert ist. Der Begriff Softwareprogramm wird sowohl für Quelltexte in einer speziellen Programmiersprache verwendet als auch für Maschinencode. Nach der Definition ist die ausführbare Regel zwar selbst ein Programm, wir differenzieren hier jedoch und verstehen das Prüfwerkzeug als Programm.

Wie bereits erwähnt, kann eine Richtlinie in mehrere einzelne atomare Regeln zerfallen. Richtliniendokumente stellen ein Sammelwerk an Richtlinien dar, aus denen vielfältige Regeln abgeleitet werden können. Demnach müssen bei einer Konformitätsprüfung mehrere,

für eine Prüfung geltende Regeln aus einem ‚Regel-Pool' vor der Prüfausführung ausgewählt werden. Diese Zusammenfassung an Regeln verstehen wir unter dem Begriff ‚Regelkatalog'.

Regelkatalog *(engl. guideline catalog)* **(4.11)**
„Die abgeschlossene Menge Ξ umfasst alle sequenziell ausführbaren Regeln einer Prüfung. Ein ***n***-Tupel ausführbarer Regeln Ψ_n aus Ξ wird als ein *Regelkatalog* bezeichnet."

Der Regelkatalog enthält alle für eine Konformitätsprüfung relevanten Regeln. Die sequenzielle Ausführung jeder einzelnen Regel ist Inhalt der Prüfung, welche auf den durch den Anwender ausgewählten Regeln basiert.

Unter ‚regelbasierter Prüfung' verstehen wir somit die automatisierte Ermittlung eines oder mehrerer Merkmale an kollaborativen Artefakten zur Konformitätsbewertung mittels eines automatisierten Verfahrens. Da die Prüfung durch Ausführung eines Regelkatalogs am Prüfaufbau erfolgt, wird diese nun spezifiziert:

Regelbasierte Prüfung *(engl. guideline checking)* **(4.12)**
„Unter einer *regelbasierten Prüfung* versteht man die programmtechnische Ausführung und Auswertung aller in der Menge Ξ enthaltenen, ausführbaren Regeln eines Regelkatalogs an einem festgelegten Prüfaufbau Δ."

Schließlich muss nach jeder ausgeführten Prüfung das jeweilige Prüfergebnis einzeln ermittelt werden. Das Prüfergebnis oder mehrere nach der Booleschen Algebra verbundene Prüfergebnisse müssen die Auswertung zulassen, ob das logische Artefakt konform zur Richtlinie bzw. der Regel(n) ist oder eben gegen diese verstößt. Wir definieren das ‚Prüfergebnis' daher wie folgt:

Prüfergebnis *(engl. examination result)* **(4.13)**
„Sei die Menge Ξ als eine Sammlung von ***n***-Regeln mit (Ψ_0, Ψ_1, …, Ψ_n | $\boldsymbol{n} \in$ |R) gegeben, dann ist das *Prüfergebnis* ein ***m+1***-Tupel ($\boldsymbol{\Sigma}_0$, $\boldsymbol{\Sigma}_1$, …, $\boldsymbol{\Sigma}_m$), wobei $\boldsymbol{\Sigma}_m$ die Menge Π aller Daten des logischen Artefakts darstellt, die der Richtlinie Ψ_n nicht entsprechen. Das Prüfergebnis ist *konform* zu einer Richtlinie, falls $\boldsymbol{\Sigma}_m = \varnothing$."

Nach ausgeführter, regelbasierter Prüfung wird jeder ausführbaren Regel ein eigenes *Prüfergebnis* zugeordnet. Dabei erfolgt eine *wahrheitswertorientierte Ermittlung* des Prüfergebnisses gemäß den in Kapitel 2.3.2 vorgestellten *Wahrheitswerten* und den *Implikationen*. Die einzelnen Prüfergebnisse können mittels Aussagenlogik zu einem Gesamtergebnis (Wahrheitswert) verkettet werden, aus diesem die Konformitätsaussage abgeleitet wird.

Eine Alternative stellt die *mengenorientierte Ergebnisauswertung* dar. Findet hierbei ein Regelalgorithmus ein nicht konformes Datum im Prüfaufbau, wird dieses in einer Ergebnismenge als Prüfergebnis festgehalten. Folglich gibt es bei einer leeren Menge keine Beanstandungen und somit lässt sich aufgrund der leeren Menge die Aussage auf Einhaltung der Konformität (*wahr* oder *falsch*) ableiten. Im umgekehrten Fall erhält das Prüfergebnis alle Daten des logischen Artefakts, welche nicht konform zur gegebenen Richtlinie sind.

4.4.2 Kriterien

In einem nach Reifegradmodell durchgeführten modellbasierten Entwicklungsprozess eingebetteter Systeme haben bei der Einschätzung ihrer Engineering-Prozesse die aus dem Bereich der Qualitätssicherung und des Qualitätsmanagements verantwortlichen Personen erhebliche Schwierigkeiten folgende Fragestellungen zu beantworten:

- *„Wie können prozesslogische Abhängigkeiten der Artefakte erkannt und bei prozessübergreifenden Arbeitsschritten noch effizient überprüft werden?“*
- *„Wie lassen sich von Personen manuell durchgeführte Dokumenten-Reviews prozessübergreifend automatisieren und Überprüfungsergebnisse dokumentieren?“*
- *„Wie regelkonform werden die Artefakte von einzelnen Personen erstellt? Wie können getroffene Entscheidungen gegen Richtlinien erfasst werden?“*
- *„Werden auch nicht-funktionale Prozessrichtlinien nachweislich umgesetzt? Wie können die Personen vom notwendigen Dokumentationsaufwand für Protokollierung oder Berichterstattung entlastet werden?“*

Diese und weitere in persönlichen Gesprächen geäußerten Fragestellungen lassen erkennen, dass der Konformitätsgrad von Artefakten einen deutlichen Bezug zu den Prozessen hat. Außerdem werden Ziele zur Erreichung eines höheren Qualitätsniveaus in Reifegradmodellen durch Richtlinien mit Artefakten in Verbindung gebracht. Artefakte müssen demnach allgemeine Qualitätskriterien durch Richtlinienkonformität erfüllen – nur welche Arten von Richtlinien müssen hierbei differenziert werden?

Aus dem ISO-Standard für Softwarequalität nach ISO/IEC 9126 (Software Engineering - Product Quality, [ISO9126]) sind allgemeine Qualitätskriterien aus dem Software-Engineering-Prozess bekannt, die neben der reinen Funktionsabsicherung auch weitere, nicht-funktionale Qualitätskriterien einführen. Dies sind Kriterien wie Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit (Abbildung 22), welche die Gesamtqualität eines Programms entscheidend beeinflussen.

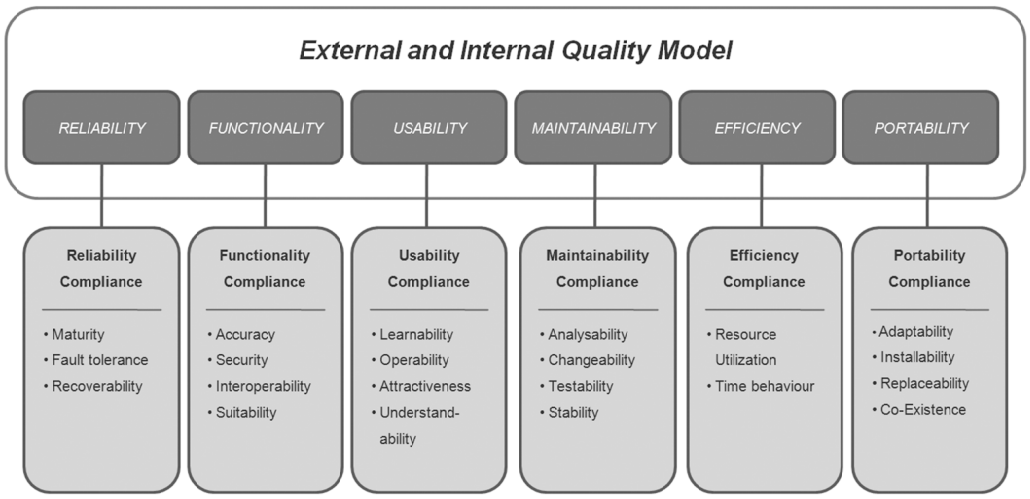


Abbildung 22: Qualitätsmodell für Software Qualität, nach ISO/IEC 9126

Es stellt sich hierbei die Frage, ob sich diese Kriterien auch für die Klassifizierung von Richtlinien bzgl. kollaborativer Artefakte eignen? Besonders in den nicht funktionalen

Eigenschaften liegt ein hohes Potenzial an Fehleranfälligkeit bei kollaborativen Arbeitsprozessen, da beispielsweise Verständlichkeit, Änderbarkeit oder Übertragbarkeit wichtige Aspekte darstellen. Nach [ISO9126] werden in der Abbildung 22 wesentliche Kriterien definiert:

Konformität zu vorgegebenen Prozessrichtlinien für eine Absicherung der genannten Kriterien unterstützt in jedem Aspekt die Erreichung nicht-funktionaler Qualitätsziele. Konformität ist somit ein notwendiges Kriterium für Artefakt-, Prozess- und schließlich für Produktqualität. Die wissenschaftliche Auswertung ergibt, dass kollaborative Prozesse für Qualitäts- und Reifegradmodelle zukünftig die Erbringung von Methoden zur strategischen Fehlervermeidung, zur stetigen Effizienzsteigerung sowie der Optimierung von Entwicklungsprozessen und Arbeitserzeugnissen bei gleichzeitiger Kostenreduktion erfordern. Anhand der gewählten Kriterien wird nun eine allgemeine Klassifizierung im nachfolgenden Kapitel erarbeitet.

4.4.3 Klassifizierung

Zur Strukturierung, zur Standardisierung und zur Verbesserung der im Entwicklungsprozess erzeugten kollaborativen Artefakte werden von den im Kapitel 3.3 genannten Konsortien normative Richtlinien durch Standardisierungsvorgaben (Normen) in natürlich sprachlicher Form vorgegeben. Resultierend werden daraus Entwicklungsrichtlinien abgeleitet, welche die Umsetzung für den jeweiligen Prozess beschreiben. Die Entwicklungsrichtlinien sind in ihrer Art, in ihrem Abstraktionsniveau und in ihrem Geltungsbereich äußerst divers. Daher muss zunächst die Klassifizierung von textuell dokumentierten Richtlinien bezüglich ihrer Art, ihrer Anwendung und ihrer Wirkungsweise vorgenommen werden. Anhand der im Kapitel 4.4.2 vorgeschlagenen Kriterien können folgende Kriterien zur Klassifizierung von Richtlinien aufgestellt werden.

Tabelle 18: Klassifizierung von Richtlinien nach Qualitätskriterien

Nr.	***Klasse***
(1)	*Funktionsabsicherung* (Functionality)
(2)	*Zuverlässigkeit* (Reliability)
(3)	*Benutzbarkeit* (Usability)
(4)	*Effizienz* (Efficiency)
(5)	*Änderbarkeit* (Maintainability)
(6)	*Übertragbarkeit* (Portability)

Im Kontext der modellbasierten Systementwicklung können für die Konformitätsprüfung nun die Entwicklungsrichtlinien anhand ihrer Klassifizierung beschrieben und auch hinsichtlich ihrer Prozessphase eingruppiert werden. Dies betrifft die in dieser Arbeit betrachteten Prozessphasen des Anforderungsmanagements, des modellbasierten Systementwurfs und der Testspezifikation. Die nachfolgenden Kapitel beschreiben die einzelnen Richtlinien-Klassen.

4.4.3.1 Funktionsabsicherung und Zuverlässigkeit

Die Entwicklungswerkzeuge in den Prozessphasen des V-Modells (vgl. Kapitel 3.8) erlauben aufgrund ihres breiten Funktionsspektrums einen hohen Freiheitsgrad an Anwendungsmöglichkeiten für einen Entwickler. Dies entspringt zum einen daraus, dass die Entwicklungswerkzeuge nicht auf eine spezifische Domäne (wie z. B. den Anforderungen der Automobilindustrie) angepasst sind, oder zum anderen daraus, dass sie unabhängig vom dedizierten Anwendungszweck (wie z. B. der Antriebselektronik oder Komfortelektronik) einsetzbar sind. Die Erzeugung von syntaktisch und semantisch korrektem sowie auch effizientem Code aus Implementierungsmodellen, im modellbasierten Entwicklungsprozess eingebetteter Systeme beispielsweise, ist nur durch eine unikonforme Modellierung und Anwendung des Entwicklungswerkzeugs gegeben. Syntaktische Fehler, falsche Annahmen des Modellierers (z. B. überdimensionierte bzw. unterdimensionierte Wertebereiche für Variable, Signalverläufe, Kalibrierung) im Modell können spätere Fehlfunktionen des implementierten Systems hervorrufen und die Zuverlässigkeit des Gesamtsystems beeinträchtigen. Auch syntaktisch und semantisch korrekter Auto-Code (generierter Code) kann zur Laufzeit fehlerhaft sein, erfüllt er nicht die Echtzeitanforderungen − somit ist auch die Effizienz von Code ein Zuverlässigkeitskriterium speziell im Bereich eingebetteter Systeme.

4.4.3.2 Benutzbarkeit und Verständlichkeit

Anforderungsspezifikationen in Lasten- und Pflichtenheften, in modellierter Form sowie durch Testspezifikationen unterfüttert, beinhalten Entwicklungsartefakte verschiedenster Ausprägungen sowie diverser technischer Dateiformate (vgl. Kapitel 3.8). Sie dienen im Produktentstehungsprozess wesentlich dem Informationsaustausch zwischen Hersteller und externen Zulieferern. Je nach Anwendungszweck, welcher mit dem kollaborativen Artefakt verbunden ist, stellen diese einen essenziellen Teil in der frühen Systementwicklung im V-Modell dar. Entweder sie spezifizieren die vom Zulieferer zu implementierende Funktion oder sie selbst sind die Basis für die Implementierung durch Codegenerierung oder den Test. In der übergreifenden Entwicklung mit mehreren Beteiligten (Hersteller, Zulieferer, Dienstleister) ist es von Bedeutung, ein gemeinsames Verständnis der zu entwickelnden Systemfunktion zu erlangen, dass durch ein für alle Beteiligten nachvollziehbares Erscheinungsbild und eine gemeinsame Strukturierung gefördert wird. Hierdurch wird die Verständlichkeit und Benutzbarkeit gewährleistet. Durch eine konzernweite sowie eine Hersteller und Zulieferer umfassende Vereinheitlichung von Entwicklungsartefakten kann dieses auch die Möglichkeit eines Wechsels zwischen verschiedenen Lieferanten unterstützen. Üblicherweise werden zur Erhöhung der Verständlichkeit Synonyme vermieden und weichmachende Wörter (z. B. Konjunktive) inhaltlich in den Artefakten ausgeschlossen. Durch festgelegte Terminologie wird die Benutzbarkeit zudem erhöht.

4.4.3.3 Änderbarkeit und Wiederverwendbarkeit

Die wesentlichen Ziele des Systementwurfs in der modellbasierten Entwicklung eingebetteter Systeme sind die Steigerung von Effizienz durch Abstraktion, Reduktion von Entwicklungszeiten durch frühzeitige Simulation, Komponentenbildung und Auto-Codegenerierung sowie eine Erhöhung der Qualität der aus den Modellen entstehenden Software. Diese Ziele sind unmittelbar mit einem hohen Grad an wiederverwendbaren und hinsichtlich ihrer Eigenschaften getesteten Modellkomponenten verknüpft. Normative Richtlinien zur Wiederverwendbarkeit von Modellkomponenten für verschiedene Generationen und Fahrzeugklassen bieten daher die Unterstützung, die entstehenden Entwicklungsartefakte durch Vereinheitlichung ihres Umfangs, ihrer inneren Struktur, ihrer

standardisierten äußeren Schnittstellen, ihrer Referenzen und ihrer Darstellung wiederverwendbar zu gestalten. Leichte Änderbarkeit ist durch Komponentenbildung und Definition von Schnittstellen möglich. Änderungshistorien und Versionierung wirken positiv auf die Änderbarkeit und sind zudem typische Richtlinien für Artefakte.

4.4.3.4 Verbesserung der Übertragbarkeit

Das Ziel einer hardwareunabhängigen Funktionsentwicklung bei eingebetteter Software hängt unmittelbar mit dem Wunsch der Übertragbarkeit innerhalb unterschiedlicher Fahrzeuggenerationen und -klassen zusammen. Dies ist aufgrund der unterschiedlichen Topologie von Elektrik-/Elektroniksystemen im Fahrzeug nicht trivial. Nach [REIF06] sind verschiedene Hardwareplattformen der Steuergeräte, der Sensorik, der Aktorik sowie ihrer Vernetzungsart durch unterschiedliche Feldbusse (vgl. [ETSB02]) verbaut. Laut [SCHÄ03] ist es in verschiedenen Fahrzeugklassen und -generationen wesentlich, dass Artefakte in der frühen Entwurfsphase zunächst von Hardwareressourcen abstrahieren. Des Weiteren werden in frühen Phasen keine Annahmen über die Signalherkunft oder das Steuergerät, für das die Funktion implementiert werden soll, vorweggenommen. Durch normative Richtlinien zur Verbesserung der Übertragbarkeit können diese steuergeräte-unabhängigen Artefakte die Übertragbarkeit unterstützen und somit auch zur Verteilbarkeit einer Funktion in einem Steuergerätenetzwerk beitragen.

4.4.3.5 Sicherstellung der Effizienz

Entwicklungsartefakte sind nur dann zweckmäßig, wenn sie für weitere Entwicklungsschritte verständlich beschaffen und auch effizient anwendbar sind. Die Anwendbarkeit sieht daher die Effizienz in einer Verhältnismäßigkeit in der Richtlinienumsetzung, d. h. auch dessen Prüfungsaufwand vor, sodass eine Überprüfung der Richtlinie mit angemessenem Ressourcenaufwand am Entwicklungsartefakt praktikabel durchführbar ist. Zudem ist ein wesentliches Effizienz-Ziel, die Anwendbarkeit der Entwicklungsartefakte nach Möglichkeit in einem breiten Anwendungskontext zu erlauben. Dabei ist ebenfalls die Einschränkung von Regeln auf einzelne Entwicklungswerkzeuge, wie beispielsweise Modellierungswerkzeuge, als differenziert zu betrachten, da spezifische Modellierungsrichtlinien zur Konsistenzsicherung eine Gefahr der Abhängigkeit vom Werkzeughersteller hervorrufen und dass die Heterogenität der beim Hersteller oder bei unterschiedlichen Zulieferern eingesetzten Werkzeuglandschaft nicht ausreichend beachtet wird. Normative Richtlinien zur Sicherstellung der Effizienz betreffen demnach Teile von Entwicklungsartefakten, welche auf werkzeugunabhängige Eigenschaften überprüfbar sind.

4.5 Richtlinien in Relation zu kollaborativen Artefakten

Im Anforderungsmanagement wird nach [VERS04] eine kontinuierliche Aufnahmeanalyse von existenten Anforderungen durchgeführt. Dies umfasst sowohl die strukturierte Auswertung von Artefakten, die auf einzelne Funktionen und deren softwaretechnische Implementierung ausgerichtet sind (z. B. funktionale Anforderungen als Artefakt), als auch die Gesamtheit der nicht-funktionalen Anforderungen als kollaborative Artefakte im System-, Umgebungs- bzw. Anwendungskontext.

Im Prozess der modellbasierten Entwicklung eingebetteter Systeme wird das allgemeine Funktionsverständnis häufig anhand eines grafischen Funktionsmodells aller enthaltenen Systemkomponenten als Artefakt erworben. Idealerweise erlaubt dies auch die frühzeitige Simulation der Systemfunktion, welche in Teilen als Steuercode generiert werden können. Herkömmliche Softwarespezifikationen im Freitextformat werden daher zunehmend durch

modellbasierte Artefakte mit Erweiterungen, z. B. durch Blockdiagramme und Zustandsautomaten, ersetzt. Beim modellbasierten Testen werden aus den Teilmodellen des Systems, die das Sollverhalten der Software beschreiben, Testfälle als Artefakte abgeleitet bzw. diese spezifiziert. In diesem Zusammenhang werden in der modellbasierten Entwicklung eingebetteter Systeme eine Reihe von Richtlinien pro Prozessphase (Anforderung, Modellierung und Test) aufgestellt und geben Anforderungen an die Herstellungsweise sowie der Beschaffenheit der Artefakte vor.

Die Richtlinien haben typischerweise meist unternehmensspezifischen Charakter, da es bisher für den Systementwurf in der modellbasierten Entwicklung eingebetteter Systeme keine allgemeingültige Normvorgabe gibt.

Aus den geschilderten Ursachen heraus sind Qualitätsprobleme in kollaborativen Artefakten zu beobachten – sie haben semantische und syntaktische Ursprünge.

Häufig zu beobachten sind:

a) *Formulierungsprobleme*,

b) *Inhalts- und Verständnisprobleme*,

c) *Strukturierungsprobleme,*

d) *Konsistenzprobleme*

e) *sowie Verfolgbarkeitsprobleme.*

Die genannten Probleme bedingen damit die konkreten Anforderungen (zur Beseitigung der Probleme) in den Entwicklungsrichtlinien.

Beispiele:

Ein Beispiel für (a) bzgl. der Ungenauigkeiten in der Formulierungsweise liegt z. B. in folgender Systemanforderung: „Die Fahrzeugfunktion Blinken *kann* bei Fahrspurwechsel durch Antippen des Lenkstocks *bis* höchstens drei Blinkzyklen durchlaufen.“ Hieraus wird nicht deutlich, ob nur genau drei oder bis zu drei Blinkzyklen implementiert werden müssen und ob es vor, in oder nach der Situation Fahrspurwechsel ausgeführt werden darf.

Ein Beispiel für (b) bzgl. der Inhaltsprobleme in der Darstellungsweise liegt z. B. in folgender Systemanforderung: „Die Fahrzeugfunktion Blinken soll bei Fahrspurwechsel bis höchstens drei Blinkzyklen durchlaufen.“ Hieraus ist nicht ersichtlich, welcher Akteur oder welches Ereignis im Modell eintreffen bzw. getestet werden muss, damit die Fahrzeugfunktion Blinken korrekt ausgeführt bzw. getestet wird.

Beispiele für (c) sind Strukturierungsprobleme im Aufbau von Anforderungstexten oder Modellarchitekturen sowie Klassifikationsbäumen, wenn die Artefakt-Struktur nicht einheitlich, nicht vollständig oder unklar (interpretierbar) ist, sinnvolle Abstraktionsgrade nicht eingehalten werden oder prozesslogische Referenzen auf andere kollaborative Artefakte (Glossar bzw. verwendete Termini) fehlerhaft sind.

Ein Beispiel für (d) sind Konsistenzprobleme, welche in allen *abhängigen* Artefakten von Anforderungstexten über Modellarchitekturen bis hin zu Klassifikationsbäumen auftreten können. Eine Anforderung einer Systemfunktion, die nicht modelliert wurde, bzw. eine modellierte Systemfunktion, die nicht in der Anforderung beschrieben wurde, stellt eine prozesslogische Inkonsistenz dar. Dies gilt auch für Testspezifikationen, die gewisse Anforderungen nicht durch Testfälle abdecken. Insbesondere, falls Artefakte auf andere Artefakte referenzieren und diese ggf. zusammengefasst wurden oder nicht mehr existieren,

liegen Konsistenzprobleme vor und begünstigt Verfolgbarkeitsprobleme. Konsistenzprobleme treten häufig durch Änderungen in den übergreifenden Prozessphasen auf oder dann, wenn Artefakte parallel bearbeitet werden und sich divergent fortentwickeln.

Beispiele für (e) sind etwas verzögert auftretende Verfolgbarkeitsprobleme, indem aufgrund der wachsenden Komplexität einer Anforderungsspezifikation, eines komplexen Modells (unter Verwendung vieler Bibliotheken) oder eines sehr umfangreichen Klassifikationsbaums nicht mehr überschaubar ist, ob die Systemfunktion (bzw. ein Anforderungsteil) implizit auf ein anderes Artefakt insofern referenziert, als dass es sich in Abhängigkeit befindet. Ein weiteres Beispielszenario ist, dass Inkonsistenzen (Lücken) in Modellen und der verwendeten Referenzen auf andere kollaborative Artefakte (ggf. auch nur durch Tippfehler verursacht) auftreten.

4.5.1 Wirkung auf kollaborative Artefakte

Nach Klassifizierung der Entwicklungsrichtlinien soll nun der Bezug zu den kollaborativen Artefakten hergestellt werden. Essenzielle Anforderungen an die Bearbeitungsweise und die Beschaffenheit kollaborativer Artefakte stellen die Grundlage einer Richtliniendokumentation dar. Nachfolgend wird beleuchtet, was die Richtlinien für Anforderungen an Artefakte im Allgemeinen stellen, ohne den Anspruch auf Vollständigkeit zu erheben.

4.5.1.1 Richtlinien zur Funktionsabsicherung und Zuverlässigkeit

Kollaborative Artefakte können, insbesondere wenn Sie unabhängig von verschiedenen Personen erstellt werden, sich gegenseitig widersprechen, sich ausschließen oder sich gegenseitig sachlogisch behindern. Dies hat direkten Einfluss auf die Funktionsabsicherung und Zuverlässigkeit der Systemauslegung und ist nicht auf funktionale Anforderungen beschränkt. Bei nicht-funktionalen Anforderungen ist zu beobachten, dass verschiedene Bereiche der Anforderungsanalyse, wie Kunden- oder Projekt-Anforderungen, sich gegenseitig beeinflussende Anforderungen aufstellen. Funktionale Anforderungen können den nicht-funktionalen beispielsweise durch Budget- und Zeit-Restriktionen in Konflikt stehen. Neben den Anforderungen, die sich direkt und sichtbar in der Software eines eingebetteten Systems auswirken, gibt es weitere Anforderungen, wie z. B. sicherheitskritische Anforderungen. Teilweise ragen diese umgebenden Anforderungen auch noch in die konkrete Softwareerstellung bzw. die Modellierung und Beschaffenheit der Testfallspezifikation hinein. Die organisatorischen Anforderungen stellen allerdings im Normalfall so genannte nicht-funktionale Anwendungen dar. Sie sollten als solche kenntlich gemacht und an einer zentralen Stelle gesammelt und gepflegt werden.

4.5.1.2 Richtlinien zur Verbesserung der Benutzbarkeit & Verständlichkeit

Die einheitliche und ausgewogene Strukturierung kollaborativer Artefakte erhöht die Benutzbarkeit und Verständlichkeit eines komplexen Systems. In Texten entspricht dies der Gliederung, in Modellen des Architektur-Designs und in Testspezifikationen der Klassifizierung gleicher Testfälle. Je nach Entwicklungsphase kann z. B. ein Modell in unterschiedlichen Ausprägungen modelliert werden. Dabei bezieht sich die Ausprägung auf den Abstraktionsgrad des zu entwickelnden Modellierungsgegenstandes. Zur Identifizierung und Nachvollziehbarkeit existieren Richtlinien, welche eine eindeutige Identifikation der Ausprägungen eines Artefakts einfordern (vgl. Forderungen aus ISO/IEC 61508 zur Unterstützung der Traceability oder Reifegradmodellen wie SPiCE).

Die Umsetzung der Anforderung bedeutet z. B. die Zuweisung von eindeutigen Schlüsseln (Bezeichnern) in den kollaborativen Artefakten. Dies erleichtert die technische Verarbeitung, die Suche und die Referenz von kollaborativen Artefakten, insbesondere dann, wenn sie einer Versionierung unterliegen. An kollaborativen Artefakten annotierte Metainformationen (Ersteller, Abteilung, Gruppierung usw.) helfen der Benutzbarkeit bzw. im Austausch mit anderen Personen und werden oft bzgl. der Angabe (Pflichtangabe) in Richtlinien eingefordert. Zur weiteren Systematisierung wird in Richtlinien die Verwendung von Glossaren gefordert, welche die verwendeten Termini in den kollaborativen Artefakten klären. Dieses erleichtert es allen anderen Prozessbeteiligten, welche eine Terminologie einer Prozessphase nicht nutzen, die Verständlichkeit der kollaborativen Artefakte zu erhöhen. Manche Richtlinien fordern eine Zielstellung des kollaborativen Artefakts. In Anforderungsdokumenten beispielsweise können in einer Zusammenfassung die Ziele bekannt gemacht werden, in Modellen ein zusätzliches Dokument zur Beschreibung erstellt werden. Dies fördert die Verständlichkeit eines Lesers bzw. eines Modellierers.

4.5.1.3 Richtlinien zur Erhöhung der Änderbarkeit und Wiederverwendbarkeit

Spezifikationsdokumente, wie Lasten-/Pflichtenhefte, sind kollaborative Artefakte des Anforderungsmanagements. Im modellbasierten Systementwurf werden Anforderungen in Modelle überführt und durch Testfälle abgesichert. Der Modellierungsgegenstand charakterisiert ein Modell hinsichtlich seiner Zugehörigkeit zu einem Teilsystem in einem allgemeinen steuerungs- oder regelungstechnischen Gesamtsystem. Richtlinien zur einheitlichen Strukturierung aller kollaborativen Artefakte zielen auf die Erhöhung der Änderbarkeit sowie Wiederverwendbarkeit ab. Im kollaborativen Artefakt sind dies geforderte Strukturierungseigenschaften zur Orientierung im Artefakt wie auch zur Partitionierung von Anforderungs-, Funktions- oder Testteilen. Die strukturierten Artefakte sollten darüber hinaus auch die Eigenschaft besitzen, leicht änderbar zu sein, da nicht jedes Projekt eine komplett neue Spezifikation, ein neues Modell oder neue Testfälle benötigt. Werden nur Systemerweiterungen vorgenommen, ist eine hohe Änderbarkeit erforderlich. Den Einsatz von gängigen Software-Werkzeugen, die innerhalb des gesamten Entwicklungsprozesses zur Verfügung stehen, prägen die Artefakt-Eigenschaften bzgl. Änderbarkeit und Wiederverwendbarkeit. Hier sind häufig sehr werkzeugorientierte Richtlinien zu beobachten, die aus Erfahrungen zur richtigen Anwendung der Software-Werkzeuge resultieren.

4.5.1.4 Richtlinien zur Verbesserung der Übertragbarkeit

Im funktionalen Systementwurf werden Modelle mit unterschiedlichen Zielen/Aspekten entworfen und sind daher bereits im Ursprung schlecht übertragbar. Dementsprechend können die entwickelten Modelle in ihrer Ausprägung sehr divers aufgebaut sein und müssen unterschiedlichen Systemanforderungen entsprechen. Während die Wiederverwendbarkeit den klaren Fokus auf Teile eines bestimmten kollaborativen Artefakts in einem Prozess legt, liegt der Schwerpunkt bei der Übertragbarkeit auf dem interdisziplinären Austausch an kollaborativen Artefakten zur weiteren Bearbeitung. Gleichwohl kann die Übertragbarkeit natürlich auch auf Teile eines bestimmten Artefakts in einem Prozess durch eine Richtlinie gefordert werden. Übertragbarkeit erweitert somit die Wiederverwendbarkeit um wesentliche Aspekte: Kollaborative Artefakte sollten einheitlich partitioniert und gruppiert werden, bestehend aus möglichst atomaren Basisbausteinen. Da meistens nur Teile eines Prozesses zur Weiterverarbeitung weitergegeben werden, wird die Übertragbarkeit dadurch erleichtert. Richtlinien zum Geltungsbereich oder Gültigkeitsbereich sind oftmals in der Übertragbarkeit zu finden. Kollaborative Artefakte sollten dahin gehend mit einem Zeitstempel (z. B. Datum) gekennzeichnet sein, einer Sichtbarkeitsinformation (z. B. ob sie öffentlichen

Charakter besitzen) oder einem Hinweis, für welchen Prozess-Bereich sie gelten bzw. verwendet werden. Bei prozessübergreifenden Richtlinien dienen systematisierte Schnittstellen dazu, ein Artefakt bereits im Prozesskontext modular zu betrachten und dieses auch so wahrzunehmen.

4.5.1.5 Richtlinien zur Sicherstellung der Effizienz

Vereinheitlichte Beschreibungsmittel und Arbeitstechniken sollten in kreativ durchgeführten Arbeitsschritten zum Einsatz kommen, da Sie eine effiziente Einarbeitung und auch Abarbeitung in nachgelagerten Prozessphasen ermöglichen. Standardisierte Beschreibungstechniken sind beispielsweise Struktur-, Text- oder Modellvorlagen, die zentral abgelegt sind. Freitexte lassen sich durch ein einheitliches Schema strukturieren. Modelle lassen sich durch eine bekannte Modellierungsnotation und vordefinierte Patterns in ein einheitliches Schema strukturieren. Modelle können genau dann personenunabhängig verstanden werden, wenn nur eine begrenzte (abgestimmte) Auswahl von Modellierungstechniken zum Einsatz kommt. Richtlinien zur Sicherstellung der Effizienz fordern daher eine konforme Bearbeitung durch ein festgelegtes Werkzeug sowie die konforme Auslegung (Parametrierung) durch einheitliche Programmeinstellungen.

4.6 Ansatz, Definition und Lösungsweg

Nach der im Kapitel 3.6 beschriebenen Beobachtung ist die Problematik heutiger Ansätze die fehlende Abstraktionsfähigkeit, die fehlenden Prozessinformationen sowie die fehlende Möglichkeit zur Darstellung prozesslogischer Zusammenhänge von den Prüflingen, welche hinsichtlich Konformitätseigenschaften zu Richtlinien geprüft werden sollen. In den vorangegangenen Kapiteln wurden daher die Anforderungen (in Kapitel 4.1) aus den Richtlinien nebst Klassifizierung (in Kapitel 4.4) gesammelt, welche eine regelbasierte Konformitätsprüfung an kollaborativen Artefakten erfüllen muss. An diesen orientiert sich der in diesem Kapitel beschriebene Lösungsweg.

Der Ausgangspunkt des hier vorgestellten Ansatzes ist zunächst die Einordnung in die Taxonomie der Testmethoden, angelehnt an [SCHI09+]. Wie in der Abbildung 23 farblich hervorgehoben, wird in unserem Ansatz ein statisches Testverfahren verfolgt, welches sich durch Regelbildung automatisiert durchführen lässt.

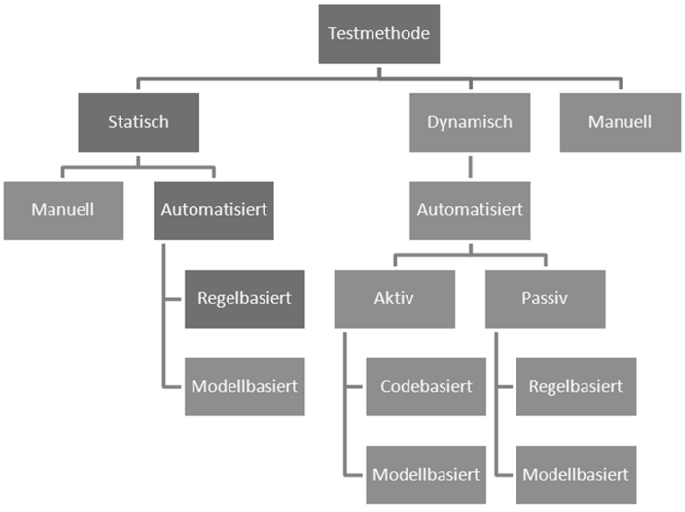


Abbildung 23: Taxonomie der Testmethoden

Das statische Testverfahren ist eine Prüfung und hat primär zum Ziel, Fehler in einem Artefakt zu finden. Als Erweiterung zu bisherigen Ansätzen werden in unserem Ansatz jedoch zusätzliche Informationen aus der Prozesswelt und aus logisch im Prozess mitgeltenden kollaborativen Artefakten mit in eine Prüfung einbezogen. Zusätzlich wollen wir vordergründig nicht einen Fehler im Artefakt nachweisen, sondern ob die Anforderung aus einer Richtlinie durch das Artefakt erfüllt ist oder nicht erfüllt ist. Die statische Konformitätsanalyse an kollaborativen Artefakten wird ebenso mit entsprechender Werkzeugunterstützung durchgeführt. Demnach lässt sich die regelbasierte Richtlinienprüfung an kollaborativen Artefakten wie folgt definieren:

Regelbasierte Konformitätsprüfung *(engl. rule-based compliance examination)* **(4.14)**
„Die *regelbasierte Konformitätsprüfung* ist ein automatisiertes Verfahren, nach dem eine Prüfung eines Artefakts bescheinigt, dass die Daten und/oder die Struktur des kollaborativen Artefakts den aufgestellten Anforderungen einer Richtlinie entsprechen."

In der regelbasierten Konformitätsprüfung kollaborativer Artefakte werden natürlich sprachliche Prozessrichtlinien semantisch auf mehrere Artefakte angewendet. Daher erfolgt die Erweiterung der regelbasierten Konformitätsprüfung durch folgende Definition:

Regelbasierte Konformitätsprüfung kollaborativer Artefakte
(engl. rule-based compliance examination of collaborative artifacts) **(4.15)**
„Die *regelbasierte Konformitätsprüfung kollaborativer Artefakte* ist eine statische Testmethode, welche sich durch ein regelbasiertes Verfahren automatisieren lässt, durch das eine übergreifende Prüfung mehrerer, sich in prozesslogischer Abhängigkeit befindlicher Artefakte bescheinigt, dass die Daten und/oder die Struktur der kollaborativen Artefakte den festgelegten Anforderungen einer Richtlinie entsprechen."

Folgende Sätze 1-5 nennen die Voraussetzungen für die regelbasierte Konformitätsprüfung kollaborativer Artefakte:

(1) Anforderungen aus einem Prozess/einer Normung müssen als Richtlinie formuliert worden sein.

(2) Die Anforderungen jeder textuellen Richtlinie lassen sich durch Logik formalisieren.

(3) Aus mehreren Logikausdrücken (Logikprogramm) können ein oder mehrere Regelalgorithmen entwickelt werden.

(4) Für das kollaborative Artefakt ist eine Struktur durch ein Metamodell definiert, sodass sich eine abstrakte Struktur (logisches Artefakt) daraus transformieren lässt.

(5) Auf dem logischen Artefakt kann der Regelalgorithmus durch ein Computerprogramm (Prüfwerkzeug) ausgeführt werden, das dem Logikprogramm entspricht.

Die regelbasierte Konformitätsprüfung kollaborativer Artefakte verwendet Logik als Grundlage für eine erste Formalisierung der Anforderung aus einer Richtlinie. Die Miteinbeziehung mehrerer Artefakte ist demnach als eine Erweiterung der regelbasierten Konformitätsprüfung zu verstehen. Neben der Erweiterung, dass mehrere Artefakte zugelassen werden, ist auch gefordert, dass sich die Struktureigenschaften eines Artefakts in eine abstraktere Form transformieren lassen. Zusätzlich ist anzumerken, dass durch die regelbasierte Konformitätsprüfung die kollaborativen Artefakte nicht verändert werden

dürfen. Nehmen wir zunächst an, dass die regelbasierte Konformitätsprüfung kollaborativer Artefakte ein abstraktes Prüfsystem Φ ist. Das Prüfsystem benötigt wenigstens eine Richtlinie als logischen Ausdruck und ein logisches Artefakt. Zudem muss das Prüfsystem auch eine Art Resultat als Prüfergebnis liefern. Das Prüfsystem hat also zwei Eingangsgrößen (Γ, ϑ) und es muss mindestens eine Ausgangsgröße Λ liefern.

Das *Prüfsystem* **Φ** hat zwei Eingangsgrößen **Γ** und **ϑ**: **(4.16)**
(1) Die Menge aller formalisierten Richtlinien als Eingangsgröße **Γ**.
(2) Die Menge aller logischen Artefakte als Eingangsgröße **ϑ**.
Das *Prüfsystem* **Φ** hat eine definierte Ausgangsgröße **Λ = *f* (Γ, ϑ):**
(3) *Prüfergebnis*, eine Funktion ***f*** über die Eingangsgrößen ist die Ausgangsgröße ***f* (Γ, ϑ)**.

Die Ausgangsgröße Λ ist das Prüfergebnis. Das Prüfergebnis als Funktion der Eingänge hat mindestens ein Ergebnis (Resultat) nach ausgeführter Prüfung, kann jedoch die Zwischenergebnisse mehrerer Teilprüfungen beinhalten. Beide Eingangsgrößen sind als Vorbedingung in einem Vorbereitungsschritt anhand einer Methodik, welche im Kapitel 6 vorgestellt wird, für das Prüfsystem Φ in eine *formale Form* und in das *logische Artefakt* zu bringen.

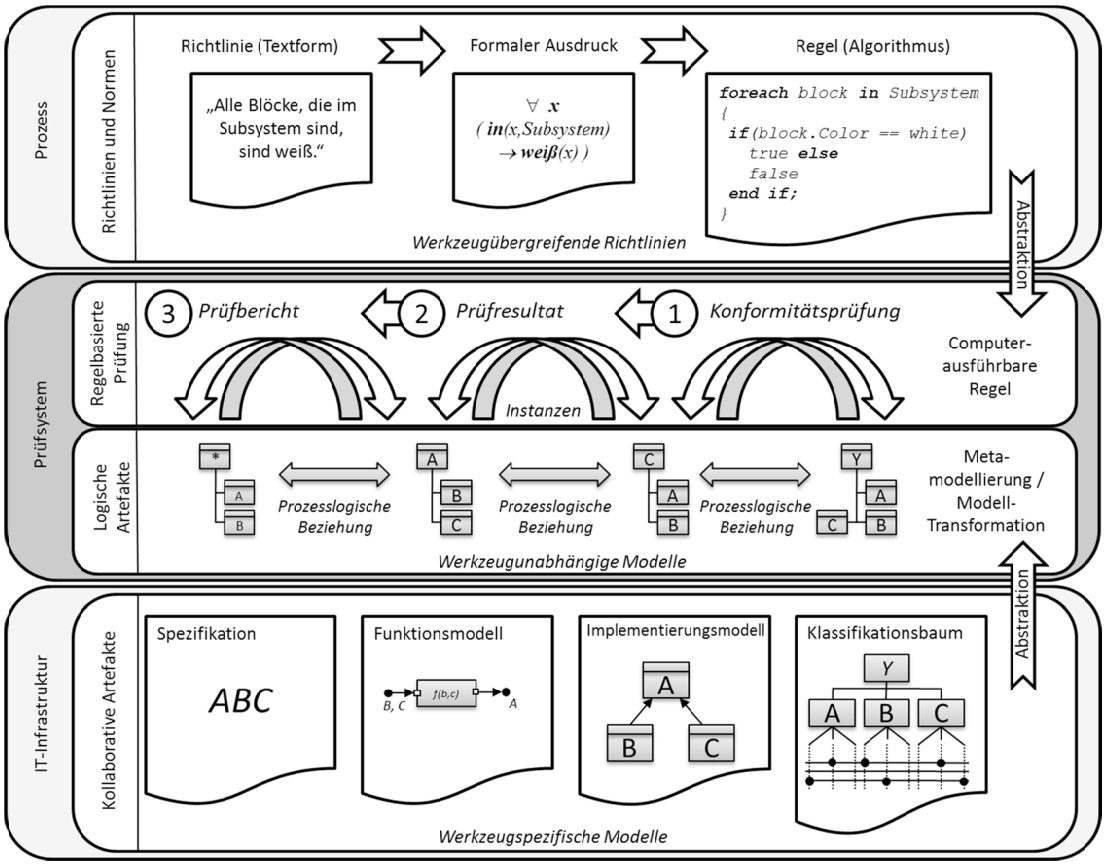


Abbildung 24: Regelbasierte Konformitätsprüfung kollaborativer Artefakte

Die Anforderungen aus natürlich sprachlichen Richtlinien (Eingangsgröße Γ) eines Prozesses müssen deshalb zunächst als Regel mittels Logikausdrücken formalisiert werden, damit sich diese in einem weiteren Schritt in einen Prüfalgorithmus überführen lassen. Die Abbildung 24 zeigt die Idee, wie eine regelbasierte Konformitätsprüfung Artefakt-übergreifend unter Integration der prozesslogischen Sicht auf kollaborative Artefakte funktionieren kann. Die organisatorischen Ebenen a) Prozess, b) Prüfsystem und c) IT-Infrastruktur eines Unternehmens beinhalten die Richtlinien und Normen aus dem Prozess, das Prüfsystem zur regelbasierten Konformitätsprüfung kollaborativer Artefakte sowie die kollaborativen Artefakte der IT-Infrastruktur.

***a) Anforderungen aus Prozessrichtlinien oder Normen*:**

Die Anforderungen werden aus den Richtlinien ermittelt und in Form von Logikausrücken mittels Prädikatenlogik erster Ordnung sowie mit Aussagenlogik formalisiert bzw. kombiniert. Aus den entwickelten Logikausdrücken werden sodann ein (oder mehrere) computerausführbare Regelalgorithmen abgeleitet, die schließlich durch ein Prüfsystem ausgeführt und ausgewertet werden können.

b) Prüfsystem:

Das Prüfsystem kann die Regelalgorithmen auf dem oder den logischen Artefakt(en) ausführen und zur Laufzeit evaluieren. Zudem wird ein Prüfergebnis in Form eines Prüfresultats durch das Prüfsystem festgelegt. Das Prüfsystem kann auch mehrere Teilergebnisse zu einem Gesamtergebnis zusammenfassen und wieder die Relation auf die ursprünglichen kollaborativen Artefakte herstellen.

c) Kollaborative Artefakte:

Durch Transformation erfolgt eine Generalisierung von den durch Werkzeuge instanziierte kollaborative Artefakte in ein generisches Datenformat (werkzeugspezifisches Modell). Zur Beschreibung prozesslogischer Abhängigkeiten oder Informationen dient zusätzlich das aus den kollaborativen Artefakten transformierte oder eigens modellierte Metamodell des logischen Artefakts als werkzeugübergreifendes Hilfsmittel für eine Konformitätsprüfung (werkzeugunabhängiges Modell).

Der vorgeschlagene Ansatz hat verschiedene Komponenten des Lösungsweges eingeführt, welche in den nachfolgenden Kapiteln detaillierter beschrieben werden. Hierzu zählen die Abschnitte:

i. Kollaboratives Artefakt, beschrieben in dem Kapitel 4.2 und logisches Artefakt, beschrieben im Kapitel 4.3.

ii. Prüfalgorithmus und Prüfresultat für die automatisierte Ausführung und Auswertung des Prüfsystems beschrieben in Kapitel 4.

iii. Methodik für die Formalisierung der Richtlinie zu Logikprogrammen sowie die Transformation der Artefakte, beschrieben in Kapitel 6.

iv. Prüfsystem (Computerprogramm) als eine Ausführungsplattform, ausführlich beschrieben und vorgestellt in Kapitel 7.

4.7 Geltungsbereich

Um einen regelbasierten Konformitätsnachweis werkzeugübergreifend führen zu können, müssen kollaborative Artefakte erstellt, verarbeitet und persistent gespeichert werden. Es wird für den Lösungsweg daher die generelle Annahme getroffen, dass in einem Unternehmen bereits viele Entwicklungsprozesse durch Dokumente, Modelle, Dateien, Datenbanken und Programme auf IT-Systemen injektiv abgebildet sind. Somit liegen viele relevante Prozessinformationen in den IT-Systemen bereits immanent und inhärent als kollaborative Artefakte unserer Vorstellung vor.

Die in dieser Arbeit betrachteten kollaborativen Artefakte einer IT-Infrastruktur bilden die Ausgangslage und grenzen Richtlinien für die Konformität physikalischer Artefakte aus. Alle Informationen, die nicht auf eine digitale Datenbasis zurückzuführen sind, z. B. ob ein Mitarbeiter eine Schutzbrille bei der Arbeit trägt oder ein Transistor im eingebetteten System thermisch überhitzt, liegen außerhalb des Geltungsbereiches. Wird in dem genannten Fall jedoch ein Fühler (Sensor) angebracht, der die Temperatur misst und digital erfasst, liegt dies wieder im Geltungsbereich, da nun die konforme Temperatur als Datum gespeichert und somit regelbasiert prüfbar würde.

Im Geltungsbereich liegen demnach alle elektronisch bearbeitbaren Dokumente einer IT-Umgebung, die strukturierte Daten aufweisen. Sie werden in Prozessen erstellt, kollaborativ bearbeitet und bilden die Grundlage von Prozessen oder Transaktionen im modellbasierten Entwicklungsprozess. Dabei ist der Einsatzbereich prinzipiell domänenunspezifisch, da die Semantik der Daten hierbei nicht von Bedeutung ist. Die Menge kollaborativer Artefakte spannt sodann den theoretisch abbildbaren Prüfraum auf. Gleichwohl lassen sich auch aus atomaren Daten abgeleitete Aussagen formulieren, die sonst nicht als Daten explizit vorliegen. Beispielsweise kann durch die serverseitig gespeicherte Check-In- und Check-Out-Zeit eines Dokumentenmanagementsystems die Bearbeitungszeit für ein Artefakt leicht errechnet werden.

Das Verfahren dient zum Zweck dem Konformitätsnachweis bezogen auf eine Anforderung einer Prozessrichtlinie auf Grundlage vorhandener kollaborativer Artefakte. Der Zweck zur Validierung von Anforderungen aus Systemspezifikationen wird dabei nicht verfolgt. Abgrenzen lässt sich somit das Verfahren bezogen auf eine Validierung, d. h. zu einer dokumentierten Beweisführung, dass ein System die Anforderungen in der Praxis erfüllen wird.

5 Algorithmisierung und Strukturierung von Regeln

Im zweiten Kapitel wurde die Formalisierung natürlich sprachlicher Regeln und der Artefakte durch einzelne Logikaussagen vorgestellt und der darauf aufbauende Ansatz in dem vierten Kapitel beschrieben. Nach dem Lösungsweg dieser Arbeit ist der nächste Schritt zu einer automatisierten Richtlinienprüfung die Umsetzung jedes formalen Logikausdrucks in einen computerausführbaren Algorithmus als Regel. Aus den Anforderungen natürlich sprachlicher Richtlinien lassen sich solche Regeln methodisch entweder mittels einer gewählten Ausdruckssprache (hierbei OCL) oder durch eine gewählte Programmiersprache als formale Regeln beschreiben (Kapitel 6). Diese Regeln können sodann als interpretierter OCL-Ausdruck oder aber auch durch einen Algorithmus einer anderen Programmiersprache von einem Prüfsystem ausgeführt und die Aussagen bzgl. der Konformität automatisiert ausgewertet werden (Kapitel 7).

Praktiziert wird eine direkte Umsetzung einer Richtlinienanforderung in eine Programmiersprache (z. B. in Java oder M-Skript) auch in verwandten Ansätzen zur regelbasierten Prüfung für ein spezielles Artefakt, wie Kapitel 3.7 eingeführt hat. Bei der regelbasierten Prüfung kollaborativer Artefakte jedoch möchte man die werkzeugspezifische Programmierung nicht auf eine Technologie bzw. nur auf bestimmte Programmierkonzepte von vornherein festlegen. In einigen Prozessen muss man flexibel und technologieunabhängig die Richtlinien zunächst formal strukturieren und beschreiben können, da man beispielsweise nicht weiß, ob ein Werkzeug im Prozess über einen längeren Zeitraum noch weiter eingesetzt werden wird (Wechsel des Programms) oder durch einen anderen Systemlieferanten eine Technologieänderung notwendig werden wird (Wechsel des Akteurs). Hinsichtlich der Algorithmisierung und zur strukturierten Beschreibung von komplexeren Logikausdrücken, verschachtelten sowie bedingt geltenden Logikausdrücken, wird in diesem Kapitel daher eine Struktur- und Auszeichnungssprache vorgeschlagen, welche für solche besonderen Prozessanforderungen eine flexible Möglichkeit darstellt, Regeln vor der Umsetzung (Implementierung) eigens zu spezifizieren. Es wird daher eine Regelsprache namens *Query-based Rule Description Language* (QRDL) erarbeitet, welche genau die strukturierte und abfrageorientierte Auszeichnungsgrammatik für artefakt-übergreifende Konformitätsprüfungen konstatiert, mittels derer sich eine komplexere bzw. artefaktübergreifende Regel-Logik aufbauen und schließlich als Algorithmus programmieren lässt. Durch Anwendung der Grammatik und Struktur der QRDL können mehrere prädikatenlogische Ausdrücke zu einem gesamtheitlichen Wahrheitsausdruck durch Aussagenlogik ausgewertet werden. Auch ist die QRDL prinzipiell offen für Erweiterungen, sofern diese für einen bestimmten Einsatzzweck begründet notwendig erscheinen.

Das fünfte Kapitel leitet zum sechsten Kapitel (insbesondere Kapitel 6.5.3) über, wo ein ausformulierter QRDL-Regelalgorithmus zur Konformitätsprüfung kollaborativer Artefakte exemplarisch angewendet wird. Einleitend bildet die QRDL eine Sammlung formaler Auszeichnungsmittel zum Ausdruck komplexer, mengenorientierter Logikausdrücke als

Abfragen auf Variable oder Mengen und deren logischer Verknüpfung. Zudem bietet ein QRDL-Regelalgorithmus eine Vorlage zur späteren Implementierung von Regeln in den interpretierten Ausdrucks- bzw. Programmiersprachen wie in dieser Arbeit OCL, LINQ und M-Skript, welche im Anhang - B der Arbeit aus der QRDL abgeleitet implementiert wurden.

5.1 Überblick

In den gängigen Verfahren zur statischen Analyse (vgl. Tabelle 13) im Bereich der modellbasierten Entwicklung eingebetteter Systeme wird das Spektrum des potenziell spezifizierbaren reaktiven Prüfverhaltens ganz allgemein durch eine *Regelsprache* bestimmt. Die Programmierung in einer solchen Regelsprache kann in imperativer oder in deklarativer Form erfolgen – meistens ist sie jedoch deklarativ. Die *deklarative Programmierung* ist ein Programmierparadigma, bei dem die Beschreibung des Problems im Vordergrund steht. Der Lösungsweg wird durch einen Algorithmus automatisch ermittelt. Zur Klasse der deklarativen Programmiersprachen gehören beispielsweise:

- *funktionale Sprachen* (z. B. Lisp, Miranda, Gofer, Haskell, Erlang)
- *logische Sprachen* (z. B. Prolog oder Datalog)
- *funktional-logische Sprachen* (z. B. Babel, Escher, Curry, Oz)
- *Datenflusssprachen* (z. B. Val oder Linda)
- *synchrone Programmiersprachen* (z. B. Lustre)
- *Ablaufwerkzeuge* (z. B. Make oder Ant)
- *Transformationssprachen* (z. B. XSLT, QVT)
- *Abfragesprachen* (z. B. SQL, LINQ)

Im Gegensatz zu einer imperativen Regelsprache, bei der das ‚*Wie*' im Vordergrund steht, fragt man in der deklarativen Programmierung nach dem ‚*Was*', das berechnet werden soll. Für die Grammatikbildung der QRDL-Regelsprache stellen folgende Eigenschaften die Anforderungen dar:

- Algorithmus und logisches Artefakt sind getrennt
- Die Schreibweise ist angelehnt an übliche Schreibweisen der Informatik
- Der Algorithmus ist eine Folge von Funktionsdefinitionen mit einem Ausdruck
- Unterschiedlich gestaltete Funktions- bzw. Modulkonzepte sind möglich
- Es existieren primitive Datentypen, Listen (Arrays),
  Terme und darauf aufbauend ein komplexerer Typverbund
- Starke Typisierung, Verwendung von Typinferenz
- Mehrere Definitionen des gleichen Funktionsbezeichners sind möglich
- Auswertung von Argumenten
- Parametrische Polymorphie
- Mengenorientierte Abfragen (z. B. SELECT *a* WHERE *c* ORDER BY *b*)
- Reguläre Ausdrücke (z. B. Mustervergleiche wie [A-Za-z0-9])
- Grundlegende Funktionen (z. B. Sum, Min, Max, InStr, Count, Group, Join)

Um die regelbasierte Konformitätsprüfung auf kollaborativen Artefakten logisch erfassen und gleichzeitig auch komplexe Anforderungen der Entwicklungsrichtlinien im modellbasierten Entwicklungsprozess eingebetteter Systeme prüfen zu können, bedarf es einer höherwertigen Regelsprache, in der sich die verschiedenen Logikausdrücke und Mengenabfragen in Form eines *Algorithmus* beschreiben lassen.

Nach [BALZ01] wird der *Algorithmus* als eine eindeutige und endliche Beschreibung eines allgemeinen, endlichen Verfahrens zur schrittweisen Ermittlung gesuchter Größen aus gegebenen Größen definiert. In imperativen Programmiersprachen besteht ein Algorithmus aus einer festgelegten Folge von Befehlen, welche eine genaue Vorschrift befolgen, die aus Eingangsgrößen in sequenziellen Schritten eine Ausgabe berechnet. In der Logikprogrammierung nach [STAE05] besteht dagegen ein Algorithmus aus einer Menge von Fakten und Regeln. Die Fakten und Regeln beschreiben die Beziehungen (*Relationen*) zwischen den Daten. Ein Logikprogramm kann als logische Theorie aufgefasst werden. Das System versucht, Antworten auf Fragen (*Abfragen* bzw. englisch *Queries*) zu berechnen. Die Antworten sind logische Konsequenzen der Fakten und Regeln. Ein Logikprogramm hat demnach sowohl eine prozedurale als auch eine logische Bedeutung für unseren Lösungsweg.

5.2 Anforderungen

Im Forschungsprojekt MESA (Veröffentlichungen, Nr. 28) wurden Anforderungen an eine Regelsprache für die werkzeugübergreifende statische Analyse untersucht, mit der sich unternehmerische Entwicklungsrichtlinien formal erfassen lassen [WANG08; JANS09]. Das Ergebnis ist, dass sich für die Erfassung heterogener Daten die Eigenschaften der mengenorientierter Abfragesprachen, wie sie bereits im Kapitel 2.5 vorgestellt wurden, für die regelbasierte Konformitätsprüfung gut eignen.

Aus den vorangegangenen Untersuchungen wird in dieser Arbeit zusammenfassend eine abfrageorientierte Regelsprache entwickelt, mit der sich prozessorientierte Richtlinien formal auf der Grundlage von Abfragen, Bedingungen und Aktionen erfassen lassen. Zunächst werden die Anforderungen an einen QRDL-Algorithmus aufgestellt.

Anforderungen an die QRDL:

- *Grammatik*: Der QRDL-Algorithmus basiert auf grundlegende Sprach- und Strukturierungselemente der QRDL.
- *Finitheit*: Der QRDL-Algorithmus muss in einem endlichen Text eindeutig beschreibbar sein.
- *Ausführbarkeit*: Jeder Schritt im QRDL-Algorithmus muss ausführbar sein.
- *Komplexität*: Ein QRDL-Algorithmus darf zu jedem Zeitpunkt nur endlich viele Ressourcen (wie Speicherplatz) benötigen.
- *Terminierung*: Der QRDL-Algorithmus darf nur endlich viele Schritte benötigen.
- *Determiniertheit*: Der QRDL-Algorithmus muss bei denselben Voraussetzungen das gleiche Ergebnis liefern.
- *Determinismus*: Die nächste anzuwendende Regel im QRDL-Algorithmus ist zu jedem Zeitpunkt eindeutig definiert.
- *Mutation*: Der QRDL-Algorithmus ändert ein logisches Artefakt nicht.

Für die Automatisierung von textuellen Richtlinien bedarf es solch einer formalen Regelsprache, deren Mächtigkeit hinreichende primitive und komplexe Aussagen erlauben soll.

QRDL erlaubt primitive Ausdrücke (atomar und nicht weiter zerlegbar) durch:

- *Arithmetik*: Addition, Subtraktion, Multiplikation, Division und Rechengesetze
- *Boolesche Algebra*: Algebraische Struktur, welche die Eigenschaften der logischen Operatoren UND, ODER, NICHT sowie die Eigenschaften der mengentheoretischen Verknüpfungen Durchschnitt, Vereinigung, Komplement beschreibt
- *Aussagenlogik*: Aussagen und deren Verknüpfung, ausgehend von strukturlosen Elementaraussagen, denen ein Wahrheitswert zugeordnet werden kann

QRDL erlaubt komplexe Ausdrücke (höherwertig, erweitert) durch:

- *Prädikatenlogik* der ersten Stufe: Prädikate und Quantoren (Allquantor, Existenzquantor) sowie Relationen
- *Relationenalgebra*: Projektion, Selektion (Informatik) als Auswahl von Datenobjekten aus einer Datenmenge (SELECT *a* WHERE *c*) sowie die alphanumerische Sortierung einer Menge (ORDER BY *b*)
- *Reguläre Ausdrücke*: Beliebige Zeichenketten, die der Beschreibung von Mengen beziehungsweise Untermengen von Zeichenketten mithilfe bestimmter syntaktischer Regeln für Mustervergleiche dienen (wie [A-Za-z0-9])

Die QRDL muss die Anforderungen nach Unternehmens- und Branchenunabhängigkeit erfüllen, da gefordert wird, dass aus einer formalisierten Richtlinie sich jeder beliebige Algorithmus

(1) auf einem Artefakt und

(2) auf mehreren Artefakten

als ein QRDL-Algorithmus formal definieren und ausführen lässt.

Aus den genannten Anforderungen wird in dieser Arbeit in den folgenden Kapiteln eine solche abfrageorientierte, höherwertige Regelsprache entwickelt.

5.3 Struktur von Regeln aus Richtlinien

Neben den aufgestellten Anforderungen soll die QRDL auch zur Strukturierung als Auszeichnungssprache dienen. Dafür ist es notwendig, zunächst die Strukturierung von Entwicklungsrichtlinien zu verstehen, um diese Strukturelemente dann auch im QRDL-Regelalgorithmus anwenden zu können. Es wird daher ein abstraktes Metamodell aus Richtlinien modelliert und beschrieben.

Entwicklungsrichtlinien sollten mittels Metadaten attribuiert werden, um eine effiziente Verwaltung (im Regelkatalog) und Strukturierung (in der Anwendung) zu schaffen. Jede Richtlinie ist demnach bezüglich Information, Inhalts und Gültigkeitsbereich in unterschiedliche Teile strukturiert. Hierzu besitzt jede Richtlinie optionale und nicht-optionale Attribute, die in (a) *informative*, (b) *inhaltliche* und (c) *klassifizierende* Attribute unterteilt werden können. Inhaltliche Attribute geben neben dem Richtlinientext Informationen zur Notwendigkeit, Anwendung und Quelle der Richtlinie an.

Klassifizierende Attribute haben im Wesentlichen die Aufgabe, Richtlinien bezüglich ihres Gültigkeitsbereichs und ihres Status zu charakterisieren.

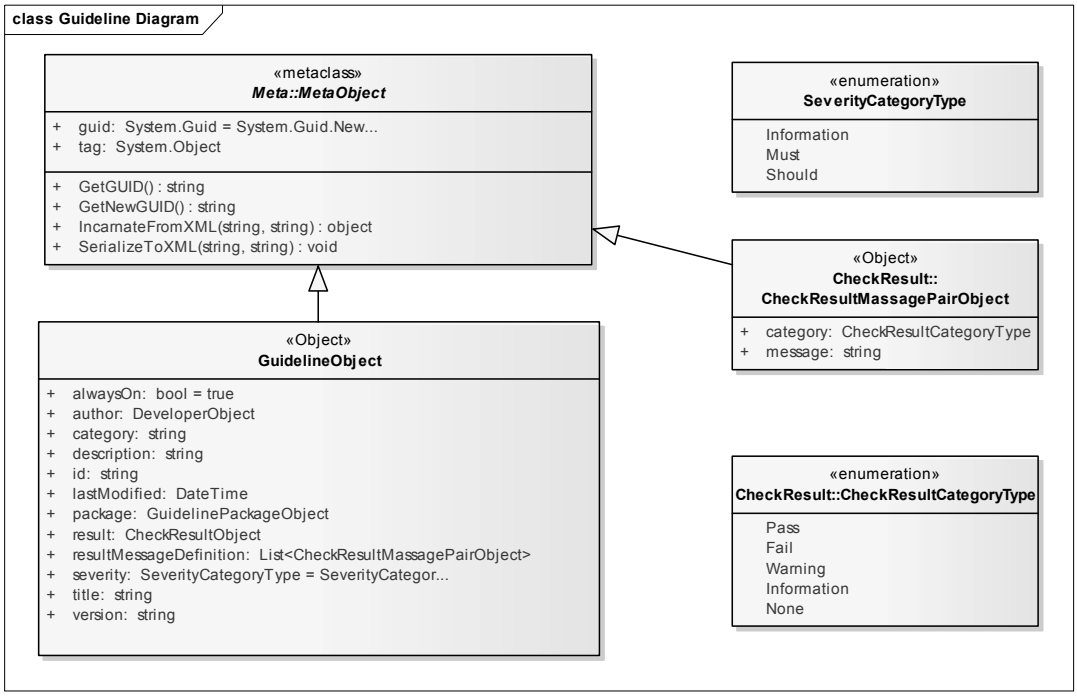


Abbildung 25: Metamodell einer Richtlinie

Die Abbildung 25 zeigt das Metamodell (Auszug) als Beispiel für inhaltliche und klassifizierende Attribute. Im Folgenden werden die wesentlichen Attribute kurz vorgestellt. Nachfolgend werden dabei die einzelnen Attribute der Metaklassen erläutert.

a) Informative Attribute einer Richtlinie

Autor, Titel, Version der Richtlinie **(**`author, title, version`**):**

Das Attribut zum Auto gibt den Urheber (*Author*) der Richtlinie an bzw. kann auch verwendet werden, um Änderungen der Richtlinie durch eine Person informativ festzuhalten. Der Inhalt dieses Attributs dient auch als Kontaktinformation bei Fragen und Änderungswünschen und legt die Verantwortlichkeit für die Richtlinie fest (Urheberschaft). Das Versions-Attribut (*Version*) notiert Revisionen bei einer Überarbeitung. Die Überschrift (*Titel*) gibt, kurz und möglichst prägnant, den Inhalt der Richtlinie wieder. Die Überschrift sollte ebenfalls eindeutig gewählt werden, um Verwechselungen zu vermeiden.

Ergänzende Attribute: Beispiele, Begründungen oder Verweise (Referenzen)**:**

Zusätzlich können zu jeder Richtlinie ergänzende Informationen als Beispiele in Form von Grafiken, Begründungen oder Verweise (Referenzen) auf weitere Richtlinien bzw. Vor- oder Nachbedingungen (*Pre-/Post-Conditions*) gegeben werden. Der Inhalt dieser Begründung gibt das Ziel und die mit der Nutzung der Richtlinie verbundenen Vorteile wieder. Somit trägt sie maßgeblich zum Verständnis und zur Akzeptanz bezüglich der Anwendung im Prozess der Konformitätsprüfung bei. Zusätzlich können hier Gefahren bei Missachtung der Richtlinie aufgeführt werden. Die Angabe eines Beispiels (Verweis auf Grafik, Bild) dokumentiert und erläutert die Umsetzung der Richtlinie. Beispielsweise werden hierzu Ausschnitte aus Modellen in Form von Screenshots benutzt (vgl. Anhang - B).

b) Inhaltliche Attribute einer Richtlinie

Beschreibung **(**`description`**):**

Die Beschreibung gibt den Inhalt (in textueller Form) der Richtlinie im Detail wieder. Mit Bezug auf die Handhabbarkeit des Richtlinienkatalogs in der Praxis sollte auch hier eine kurze und prägnante Formulierung gewählt werden. Bei zu langen Texten sollte stets geprüft werden, ob nicht eventuell mehrere unterschiedliche Aspekte in einer Regel verarbeitet wurden und stattdessen eine Aufteilung (Partitionierung) in einzelne Richtlinien sinnvoll ist.

Formaler Regelausdruck **(**`package`**):**

Hinter diesem wohl wichtigsten Attribut verbirgt sich die Angabe des Regelausdrucks zur computertechnischen Ausführung der Richtlinie (ausführbarer Regelalgorithmus).

Prüfergebnis **(**`result`**,** `resultMessageDefinition`**):**

Eine ausführbare Regel liefert ein Prüfergebnis zurück. Prüfergebnisse können nach Fehlerkategorien klassifiziert werden (`CheckResultCategoryType`) in:

- ***Pass***: Prüfung fand keine Auffälligkeit,
- ***Fail***: Prüfung fand eine schwerwiegende Anomalie,
- ***Warning***: Prüfung fand eine Anomalie mit geringer Auswirkung,
- ***Information***: Prüfung fand eine Anomalie,
- ***None***: Das Prüfergebnis kam zu keiner Aussage.

Inhaltlich kann das Prüfergebnis wie folgt beschaffen sein:

- ***Standardisierter Text***:
  Das Prüfergebnis ist „konform“ oder „non-konform“.
- ***Dynamisch generierte Fehlermeldung***:
  Das Prüfergebnis setzt sich aus einem dynamisch während der Prüfung zusammengestellten und verketteten Ergebnistext zusammen.
- ***Wahrheitswert***: *true*, *false*
- ***Numerischer Wert***: { 1 .. ***n*** }
- ***Objektliste***: $\boldsymbol{x}\begin{cases} Menge\ von\ Objekten \\ \varnothing, \text{ sonst} \end{cases}$

c) Klassifizierende Attribute einer Richtlinie

Eindeutige ID **(**`guid, id`**):**

Jede Richtlinie ist durch einen eindeutigen Bezeichner (ID) gekennzeichnet. Diese ID besteht aus einer formatierten Zahlenfolge. Eine vollständige ID ist beispielsweise durch die {9906BA94-383A-4524-89D5-CC0F22FD4BDC} Zahlencodierung gegeben. Die ID dient im Wesentlichen zur eindeutigen, computertechnischen Referenz zu Richtlinien und wird bei der Verwaltung (Laden/Speichern), bei Prüf-Ausführung oder bei Änderungsanfragen verwendet. Von einer alphabetischen Reihenfolge oder sequenziellen Nummerierung { 1..***n*** } der Richtlinien ist generell abzuraten, da Richtlinien in Katalogen später sehr unterschiedlich (in Reihenfolge und Klassifizierung) zur regelbasierten Konformitätsprüfung in Abhängigkeit und Reihenfolge stehen werden (vgl. Kapitel 6.6).

Optionaler Ausschluss **(**`alwaysOn`**):**

Jede Richtlinie ist durch das Attribut gekennzeichnet, damit festgestellt werden kann, ob die Richtlinie zwingend erforderlich ist (eingehalten werden muss) oder einen optionalen Charakter besitzt. Optionaler Charakter bedeutet, dass die Richtlinie von einer Prüfung ausgeschlossen werden kann (Ausnahme).

Schweregrad **(**`severety`**):**

Das Attribut ist durch die Enumerationsklasse `SeveretyCategoryType` gegeben. Diese hat Informationsattribute *Must*, *Should* und *Information.* Sie klassifizieren semantisch den Schweregrad einer fehlgeschlagenen Richtlinie nach einem Prüfdurchlauf.

Gültigkeitsbereich **(**`category`**):**

Dieses Attribut kennzeichnet die ursprüngliche Herkunft der Richtlinie bzw. zeigt ihren Gültigkeitsbereich an. Dies kann entweder der Name des Entwicklungswerkzeugs (wie z. B. MATLAB/Simulink) sein oder eine Referenz auf ein zugrunde liegenden Qualitätsstandard (z. B. Modellierungsrichtlinien nach MAAB-Style-Guide).

5.4 Query-based Rule Description Language

Nachdem Anforderung und Struktur in den vorangegangenen Kapiteln eingeführt wurden, kann nun die Query-based Rule Description Language (QRDL) durch eine Grammatik und durch eine einfache Struktur definiert werden. Wie der Name schon verrät, basiert diese Ablaufbeschreibung auf Regelalgorithmen, welche die Konformität von kollaborativen Artefakten mittels Logikausdrücken auf Datenmengen prüfen und mit einer Aktion (dem Prüfergebnis) quittieren. Am einfachsten ist die Idee der Beschreibungssprache durch das sogenannte LCA-Paradigma (Load-Condition-Action) nach abarbeitungsorientierter Logik zu vermitteln:

ON	***L**oad*
IF	***C**ondition*
DO	***A**ction*

ON: Ein Artefakt kann aus verschiedenen Daten konstruiert sein, die beim Laden (*Load*) des Artefakts einen Objektbaum zu einem bestimmten Zeitpunkt im Speicher repräsentieren.

IF: Die Bedingung (*Condition*) bezieht sich auf den für das Artefakt geltenden Zustand oder auf den aktuellen Wert von Daten, der durch bedingte Logikausdrücke geprüft wird.

DO: Die Aktion (*Action*) liefert ein aus der Bedingung resultierendes Prüfergebnis, kann aber auch ein Zwischenergebnis sein.

Die sequenzielle Abarbeitung der Schritte *ON*, *IF*, *DO* liefert eine regelbasierte Konformitätsprüfung auf einem logischen Artefakt oder an mehreren logischen Artefakten. Neben der abarbeitungsorientierten Logik muss außerdem die Grammatik festgelegt werden:

QRDL-Grammatik:

Algorithm = { *Declarations Rule Result* }

Declarations = *Variable* | *Constant ';'*

Variable = *Ident ":=" ValueType ';'*

Constant = *Ident ":=" Value | Char{ Char } ';'*

Ident = *Char{ Char }*

Char = 'A' - 'Z' | 'a' - 'z'

ValueType = 'ARTIFACT' | 'CHAR' | 'STRING' | 'BOOL' | 'INT' | 'FLOAT' | 'BYTE' | 'DATETIME' | 'OBJECT'

Value = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Rule = { QUERY<*Ident*> "⊆" *Operator Element Operator Artifact Condition { Condition }* SELECT *Element* '.' }

Operator = '∈' | '∉' | '∀' | '∃' | ,IF' | ,THEN' | ,ELSE' | ,RETURN'

Logic = ,∧' | ,∨' | ,→' | ,←' | ,↔'

Query = <*Ident*>

Element = <*Ident*>

Artifact = <*Ident*>

Condition = *RuleAlgorithm | RuleSemantic*

RuleAlgorithm = *Algorithm* | { *Expression Logic Expression* }

RuleSemantic = *Variable Logic Variable*

Result = *Operator* ,True' | ,False' | *Element{ Element }* | *Value{ Value }* | *Char{ Char }*

Comment = *"/*" Value{ Value } | Char{ Char } "*/"*

Mit der Grammatik soll ausdrückt werden können, dass ein Algorithmus aus einer *Deklaration*, der *Regel* sowie der *Ergebnisfeststellung* aufgebaut ist. Die Deklaration umfasst die Definition von Konstanten- und Variablennamen mit Wertzuweisung für den Algorithmus. Dies sind Angaben, welche Werte, Zeichen oder Zeichenfolgen Variable oder Konstante aufnehmen können und dass sie einen primitiven Datentyp bzw. die Referenz auf ein Artefakt besitzen. Danach folgt die abfrageorientierte Regel. Diese ist die Teilmenge, die durch Abfrage mittels einer Nebenbedingung (Constraint) für alle Elemente eines referenzierten Artefakts gebildet wird. Die Nebenbedingung selbst kann wieder durch einen Algorithmus ausgedrückt werden oder einen Logikausdruck, welcher die Semantik der Nebenbedingung festlegt. Schließlich muss die Ergebnisfeststellung Werte liefern. Diese sind ein Wahrheitswert (wahr, falsch), ein Element bzw. eine Menge von Elementen, ein Wert bzw. eine Wertefolge, ein Zeichen oder eine Zeichenkette. Zusätzlich sind Kommentare erlaubt, welche besonders gekennzeichnet werden.

Bei der Regeldefinition sind die Struktur, der Inhalt und die Semantik zu spezifizieren. Die Sprache setzt sich daher aus einem *strukturellen* und einem *semantischen* Teil zusammen.

Der strukturelle Teil (<*rule_algorithm*>) ist ein Prüfalgorithmus, der die Daten eines geladenen Artefakts prüft. Für diesen Teil gelten alle im Kapitel 5.2 aufgeführten Bedingungen. Im semantischen Teil (<*rule_semantic*>) kann zusätzlich festgelegt werden, welche Bedeutung das Ergebnis des Algorithmus für das Resultat hat. Ein grundlegender QRDL-Ausdruck weist die folgende deklarative Syntax für ein Artefakt auf:

Syntax einer atomaren QRDL-Regel:

[< *declarations*>]

<*query*> := <*element*> ∈ <*artifact*>

<*rule_algorithm*>

<*rule_semantic*>

<*result*>

Für die Regeldefinition werden optional im <*declarations*> Abschnitt alle benötigten Variablen eingeführt oder die Konstanten festgelegt. Eine Regel setzt sich demnach aus einem strukturellen und einem semantischen Teil zusammen. Die Regel <*query*> ist eine Abfrage auf ein referenziertes Element im Artefakt (Objektbaum) <*artefact*>, welches einer Bedingung unterliegt. Diese Bedingung wird im Abschnitt <*rule_algorithm*> algorithmisch festgelegt. Die semantische Auswertung <*rule_semantic*> legt eine Fehlerklasse fest. Die Bewertung wird durch <*result*> durchgeführt.

Die Regelsprache QRDL erlaubt für die werkzeugübergreifende Prüfung die sequenzielle Abarbeitung mehrerer Abfragen mit anschließender Auswertung. Wenn nun Artefakt-übergreifend geprüft werden soll, müssen für jede Artefakt-Referenz sodann ***n***..***n***+1 Teil-Regeln <*query_n, query_n+1*> je Artefakt formuliert werden. Diese liefern jeweils eine Konformitätsaussage pro Artefakt. Das heißt, die Gesamtaussage muss noch durch die Aussagenlogik (logische Verkettung der Teilergebnisse) gefunden werden. Dies geschieht im Abschnitt der Auswertung <*evaluation*>, in dem die Teilregeln algorithmisch und semantisch auf dem logischen Artefakt ausgewertet werden. Die semantische Auswertung <*rule_semantic*> legt die übergreifende Fehlerklasse fest. Schließlich wird die globale Bewertung durch <*result*> durchgeführt.

Ein grundlegender QRDL-Ausdruck weist die folgende deklarative Syntax für mehrere Artefakte auf:

Syntax einer komplexen QRDL-Regel:

[< *declarations*>]

<*query_n*> := <*element*> ∈ <*artifact_n*>

<*rule_algorithm*>

<*rule_semantic*>

. . .

<*query_n+1*> := <*element*> ∈ <*artifact_n+1*>

<*rule_algorithm*>

<*rule_semantic*>

<*evaluation*> := <*query_n*> ∨ <*query_n+1*>

<*result*>

Die QRDL-Grammatik erlaubt die Formulierung von *Logikprogrammen* als komplexer Prüfalgorithmus, welche Artefakt-übergreifend prüfen können.

Zur Anschaulichkeit soll nun ein Beispiel einer atomaren QRDL-Regel dienen. Hierbei soll für ein logisches Artefakt geprüft werden, ob alle enthaltenen Elemente und deren Datenwerte größer als der vorgegebene Schwellwert sind. Falls kein Element einen Datenwert unter dem Schwellwert besitzt, ist dieses Artefakt konform zur Richtlinie, ansonsten wurde die Richtlinie verletzt.

Beispiel: ***Atomare QRDL-Regel***

```
/* Declaration */
INT i := 10; ARTIFACT Data;
/* PRE-CONDITION: x == null */
QUERY_X ⊆ ∀ x ∈ Data
/* PRE-CONDITION: x <> null */
        x > i /* constraint */
SELECT x
 /* Result */
IF x != null THEN
   RETRUN False;  /* non-compliance */
ELSE
   RETRUN True;  /* compliance */
END IF
```

Hinsichtlich der strukturierten Beschreibung von komplexeren Logikausdrücken in den Schritten *ON*, *IF*, *DO* werden Auszeichnungselemente eingeführt, welche verschachtelte sowie bedingt geltende Logikausdrücke definieren oder zu einem Logikprogramm der QRDL zusammenfassen.

Die QRDL-Struktur ist eine an die XML angelehnte Auszeichnungssprache, bestehend aus Elementen markiert durch Tags (vgl. Kapitel 2.5.1), deren Verschachtelungsregeln und aus Attributen mit erlaubten Wertzuweisungen. Platzhalter (Tags) werden durch Spitze Klammern < > eingeschlossen und markieren einen Abschnitt. Zwischen den Klammern lassen sich aus der Grammatik der QRDL, z. B. für einen Logikausdruck, verschiedene Ausdrücke strukturieren.

Beispiel: ***Struktur einer QRDL-Regel***

```
<BEGIN RULE>
        [ <DECLARATIONS> Declarations </DECLARATIONS> ]
         <EXAMINATIONS>
                 <QUERY> Rule A </QUERY>
                    .
                    .
                    .
                 <QUERY> Rule Z </QUERY>
         </EXAMINATIONS>
         <ESTIMATION> RuleAlgorithm </ESTIMATION>
        <COMPLIANCE RESULT> Result </COMPLIANCE RESULT>
<END RULE>
```

Zunächst wird der Regelalgorithmus durch die Tags *<BEGIN RULE>* und *<END RULE>* syntaktisch eingegrenzt. Es folgt ein Deklarationsteil *<DECLERATIONS>* von Konstanten und Variablen. Da dieser optional ist, ist er zur Kennzeichnung mit eckigen Klammern [ ] eingeschlossen. Es folgt ein Abschnitt *<EXAMINATIONS>*, der alle Logikausdrücke in einer mengenorientierten Abfrage sequenziell durch die *<QUERY>*-Tags gruppiert. Im Abschnitt *<EXAMINATIONS>* wird der übergreifende Logikausdruck zu einer Menge durch die logische Verkettung der einzelnen Abfragen zusammengeführt. Schließlich wird im Abschnitt *<COMPLIANCE RESULT>* das ermittelte Prüfergebnis ausgewertet, einer Fehlerkategorie zugeordnet und als Rückgabewert festgehalten.

Ein ausführliches Beispiel eines ausformulierten, komplexen QRDL-Regelalgorithmus zur Konformitätsprüfung kollaborativer Artefakte wird im Kapitel 6.5.3 vorgestellt.

6 Methodik der Regelentwicklung

Die Ableitung von Richtlinien zu Regeln mittels Deduktion erfordert einen nicht trivialen Denkprozess, nachdem mehrere aufeinander aufbauende Arbeitsschritte für eine regelbasierte Konformitätsprüfung an kollaborativen Artefakten notwendig sind. In der Deduktion werden Aussagen (Logik) aufgestellt, die von anderen Aussagen abgeleitet sind, d. h., es werden Schlussfolgerungen vom Allgemeinen zum Besonderen gebildet. Dieses Kapitel stellt eine deduktive Methodik für die Entwicklung von Regeln vor. Beginnend wird der allgemeine Entstehungsprozess von Regeln beschrieben. Anschließend wird auf Grundlage des Kapitels 4 und des Kapitels 5 schrittweise ein pragmatisches Vorgehensmodell (VR-Modell) aus sechs Phasen konzipiert, welches die mathematischen Beschreibungs- und Abstraktionsmittel aus dem Kapitel 2 für die Formalisierung der Richtlinien anwendet, um schließlich die Ableitung von umgangssprachlichen Richtlinien zu computerausführbaren Ausdrücken und Algorithmen für die statische Analyse an kollaborativen Artefakten zu ermöglichen. Zur Beschreibung und Strukturierung von Regelalgorithmen wird die Regelsprache *Query-based Rule Description Language* (QRDL) eingesetzt, welche die formale Grundlage für die spätere Implementierung von Regelalgorithmen (Anhang - B) in beliebige Programmiersprachen darstellt.

6.1 Entstehungsprozess von Richtlinien

Der generelle Ursprung von Richtlinien – wie auch von Normungen – sind (teils negative) gesammelte Erfahrungswerte im Entwicklungsprozess. Hinzu kommen aus einem kreativen Denkprozess gewonnene Ideen zur Vereinfachung, Optimierung oder Standardisierung im Entwicklungsprozess bearbeiteter Artefakte. Schließlich sind auch durch Interviews entstandene Meinungsbilder in Richtlinien vertreten, um ein ausgedehntes Verständnis und eine vereinheitlichte Denkweise der Entwickler und Reviewer zu fördern. Grundlegend für den Entstehungsprozess der Richtlinien sollte die Einbeziehung einer möglichst breiten Basis an vorhandenem Wissen sowie bereits bestehender Richtlinien im Unternehmen sein. Unter Miteinbeziehung von weiteren Interessengruppen in Konsortien, wie im Kapitel 3.3 beschrieben, können Richtlinien abgeleitet werden, um insbesondere auf existierenden Erfahrungen aufzubauen und somit die Akzeptanz zu fördern sowie die Einführung in den kollaborativen Prozessen zu erleichtern. In der Abbildung 26 ist ein Ablaufdiagramm gezeigt, welches den iterativen Entstehungsprozess eines textuellen Richtlinienkatalogs für den modellbasierten Entwicklungsprozess aufzeigt.

Akteuren (Modellierern) liegen Modelle vor. Methodische und werkzeugspezifische Vorgaben sind Ausgangspunkt des Prozesses. Zur Erfassung von Erfahrungswerten werden mit Entwicklern Interviews geführt sowie Best Practices der Personen berücksichtigt. Dies bildet die Grundlage für den generellen Entwurf der Richtlinien. Ist dieser freigegeben, wird er durch ein Steuerungskomitee (Change Control Board) verifiziert und ggf. mit anderen Abteilungen oder Zulieferern abgestimmt. Hier werden deren Feedbacks zum Entwurf und Änderungswünsche berücksichtigt.

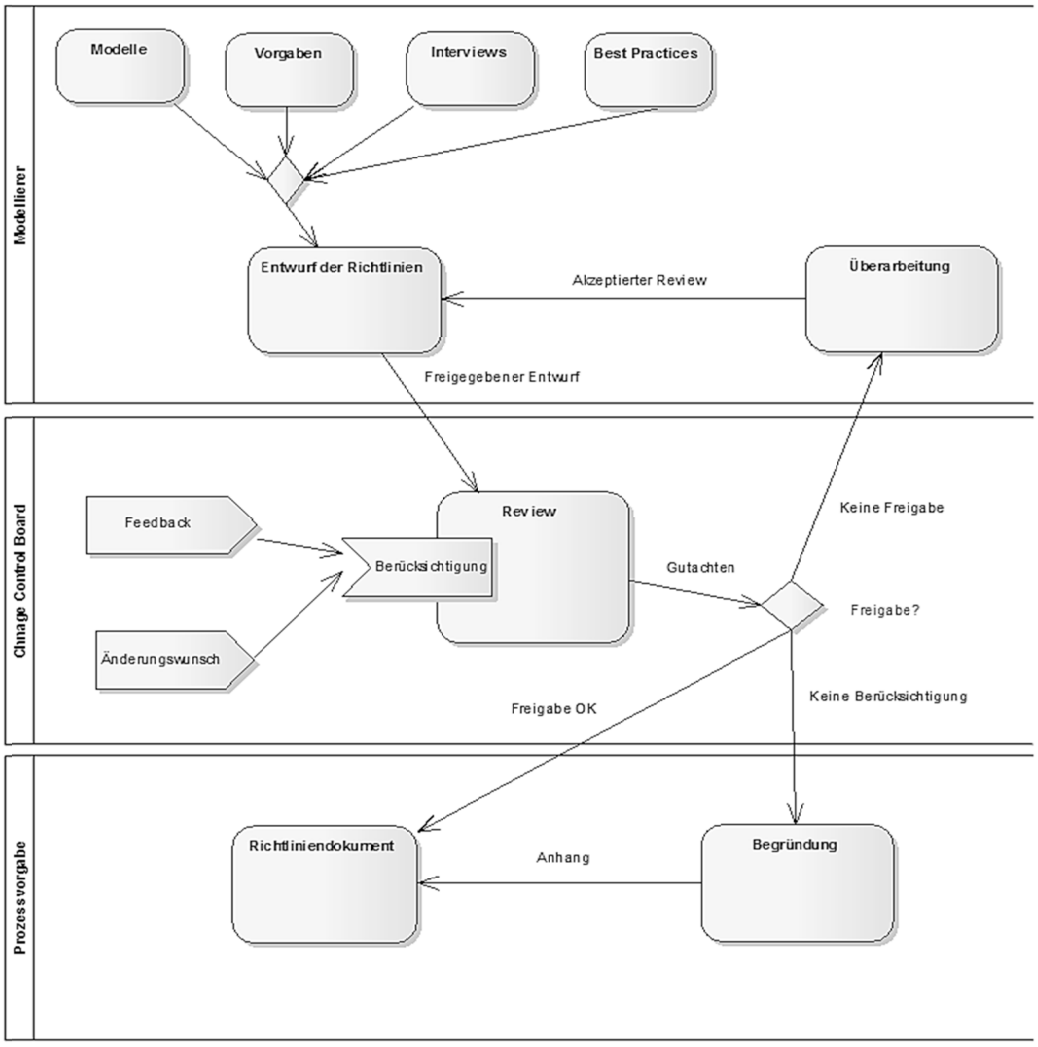


Abbildung 26: Entstehungsprozess eines Richtlinienkatalogs

Schließlich entscheidet der Review-Prozess nach Begutachtung über eine Freigabe oder gibt den Entwurf zur Überarbeitung an die Modellierer zurück, die dann die hinzugekommenen Änderungen iterativ einarbeiten. Finden Feedbacks zum Entwurf und Änderungswünsche keine Berücksichtigung, werden diese in einem (fortgeschriebenen) Begründungsdokument erfasst, damit abgelehnte Feedbacks zum Entwurf und Änderungswünsche nicht zu einem späteren Zeitpunkt erneut diskutiert werden müssen. Dieses Begründungsdokument kann sodann als Anhang dem freigegebenen Richtliniendokument angefügt werden. Das Richtliniendokument erhält eine feste Versionsnummer und ist sodann gültig für den Entwicklungsprozess.

6.2 Vorgehensmodell (VR-Modell)

Um die regelbasierte Konformitätsprüfung auf kollaborativen Artefakten durchführen zu können, bedarf es eines eignen Vorgehensmodells für die Regelentwicklung, welches die einzelnen Schritte (Phasen) und insbesondere die Verfahrenstechniken für eine automatisierte Richtlinienprüfung (im Kapitel 7) vorstellt. Das Vorgehensmodell für die Richtlinienerstellung kollaborativer Artefakte (kurz: VR-Modell) lässt sich in sechs

iterativen Prozessschritten (siehe Abbildung 27) beschreiben, bestehend aus: *Analyse, Abstraktion, Formalisierung, Konformitätsprüfung, Auswertung* und *Optimierung.*

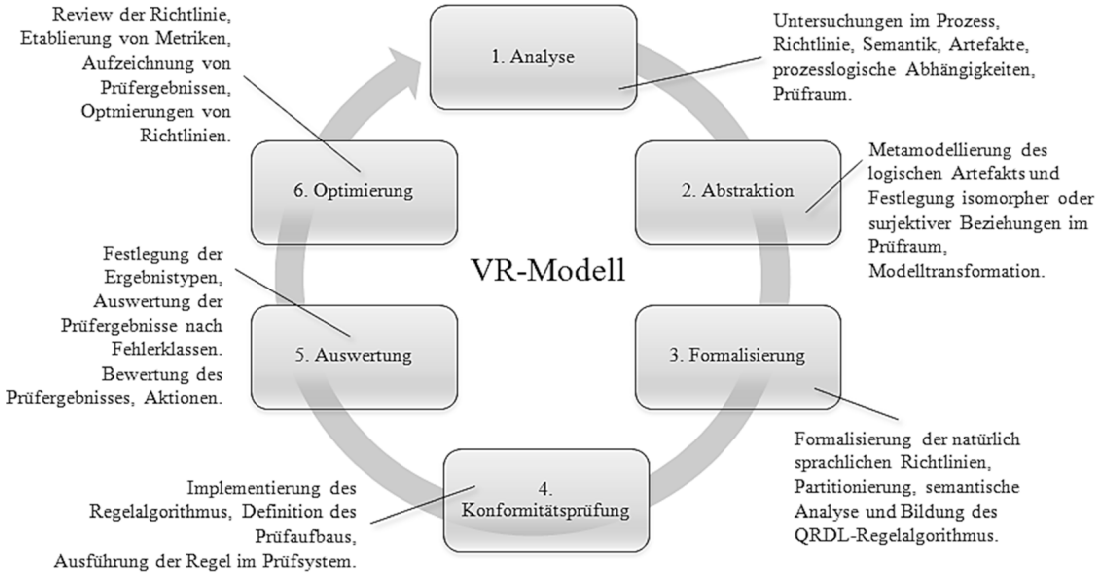


Abbildung 27: Vorgehensmodell Regelentwicklung

Die im VR-Modell dargestellten Phasen 1-6 werden zunächst kurz beschrieben und in den nachfolgenden Kapiteln 6.3 bis 6.8 ausführlich vorgestellt.

In der ersten Phase (*Analyse*) werden natürlich sprachliche Richtlinien hinsichtlich Automatisierbarkeit untersucht. In der zweiten optionalen Phase (*Abstraktion*) folgt nach der Analyse die Festlegung isomorpher oder surjektiver Beziehungen zwischen kollaborativen Artefakten, wahlweise die Metamodellierung des logischen Artefakts und die Wahl der Modelltransformation. In der dritten Phase (*Formalisierung*) werden die ausgewählten Richtlinien als Regel formalisiert sowie pro Regel ein Prüfalgorithmus in der QRDL entwickelt. Hierzu sind insbesondere die Kenntnisse des Artefakt-Metamodells bzw. des logischen Artefakts notwendig. Die Implementierung der Regel geschieht nachfolgend in Phase vier (*Konformitätsprüfung*). Hierbei wird der QRDL-Regelalgorithmus aus der vorherigen Phase implementiert und auf dem(n) logischen Artefakt(en) ausgeführt. In der fünften Phase (*Auswertung*) wird nach abgeschlossener Prüfung bestimmt, wie das Prüfergebnis zu bewerten ist. Es erfolgt die Festlegung einer Fehlerkategorie. Schließlich wird in der Phase sechs (*Optimierung*) eine spezielle Untersuchung zur Verbesserung der Richtlinie oder des Prozesses durchgeführt. Es kann nötig sein, dass Vorbedingungen oder Nachbedingungen vor einer Prüfung gelten müssen oder Aktionen auf ein bestimmtes Resultat im Prozess nach Prüfdurchlauf (z. B. Korrekturmaßnahmen) vorzunehmen sind. Dies kann wieder eine erneute Analyse erfordern, ob sich der Prozess zwischenzeitlich geändert hat oder ob es neue oder verwaiste Querbeziehungen zu anderen Artefakten gibt. Zudem kann in der Optimierung eine Auswertung mehrerer Prüfdurchläufe über die Zeit (Historie) durchgeführt werden, welche zusätzliche, metrikbasierte Regelprüfungen erfordert und somit den Ablauf erneut anstößt.

Das in der Arbeit evolutionär entwickelte VR-Modell ist selbstoptimierend, sofern es mehrfach iterativ durchlaufen wird. In den nachfolgenden Unterkapiteln werden die einzelnen Phasen Analyse, Abstraktion, Formalisierung, Konformitätsprüfung, Auswertung und Optimierung detailliert vorgestellt.

6.3 Analysephase

Im vorherigen Kapitel 6.1 wurde der Entstehungsprozess von Richtlinien überblickartig eingeführt. Dieser Prozess findet typischerweise vor der ersten Phase im VR-Modell statt. In dieser Phase (*Analyse*) werden bereits vorhandene, natürlich sprachliche Richtlinien hinsichtlich ihrer Automatisierbarkeit nach einer bestimmten Vorgehensweise (Abbildung 28) untersucht sowie hinsichtlich Geltungsbereich, prozesslogischen Abhängigkeiten und ihrer Automatisierbarkeit eingeschätzt. Neben bereits existenten Richtliniendokumenten oder technischen Normenwerken lassen sich zudem aus der analytischen Betrachtung des Entwicklungsprozesses und der angrenzenden Prozesse, den darin eingesetzten Werkzeugen und den damit erzeugten Artefakten die Konventionen, Best-Practices und Vorgaben im Prüfraum ableiten.

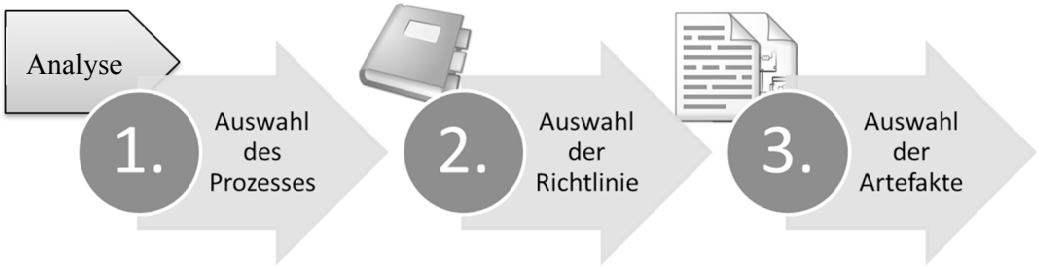


Abbildung 28: Analyseschritte im VR-Modell

Die Abbildung 28 zeigt die wesentlichen drei Schritte in der Analysephase:

- *Auswahl des Prozesses*: In diesem Schritt wird der Prozess analysiert. Es werden die Prozessgrenzen genau definiert und angrenzende (vorgelagerte oder nachgelagerte) Prozesse bestimmt. Für die kollaborative Artefakt-Prüfung sind hier insbesondere die prozesslogischen Wechselwirkungsaspekte, bedingt durch Arbeitsabläufe im Prozess, wichtig. Wird beispielsweise aus einem Artefakt ein anderes automatisch generiert (wie bei Modell zu generiertem Code), ist die Konformität des Codes abhängig von den konformen Einstellungen im Programm (Modellierungswerkzeug), der Transformationsvorschrift (Parameter des Codegenerators) sowie den vorher im Modell festgelegten Parametern (Variable, Konstante und Wertebereiche). Wird zusätzlich eine Namenskonvention für Bezeichner aus einem Glossar gefordert, sollte dies mit einbezogen werden. Auch die Spezifikationen der Testphase sind zu berücksichtigen, schließlich ist der gesamte Diskursbereich im Produktentstehungsprozess bedeutend.
- *Auswahl der Richtlinie*: In diesem Schritt wird entweder eine Richtlinie natürlich sprachlich formuliert oder eine bereits vorliegende Richtlinie aus einer Menge von geltenden Prozessrichtlinien ausgewählt. Es erfolgt die semantische Analyse durch Ermittlung von Homonymen und Synonymen sowie deren Bezug zu den kollaborativen Artefakten. Hierbei kann es vorkommen, dass Richtlinien (z. B. aus rechtlichen oder regulativen Anforderungen) keine oder nur sehr universelle Empfehlungen bzw. Handlungsanweisungen für die Umsetzung und somit ihre Erfüllung beinhalten. Auch können abstrakte Leitfäden für ein generisches IT-Compliance-Management vorhanden sein, die nicht konkret ausgearbeitet sind. In diesen Fällen muss die Präzisierung der Richtlinie in diesem Schritt erfolgen, um die

Wirkung auf Artefakte und die Abhängigkeiten im Prozess mit Bezugnahme auf die vorherigen Schritte festzulegen.

- *Auswahl der Artefakte*: In diesem Schritt erfolgt die Festlegung des Prüfraums (vgl. Kapitel 4.2). Aus dem vorherigen Schritt wurde der Diskursbereich ermittelt und alle sich darin befindlichen kollaborativen Artefakte identifiziert. Es liegt nun eine nicht leere Menge von Artefakten vor, für die Entwicklungsrichtlinien gelten können. Prozesslogische Beziehungen und Wechselwirkungsaspekte zwischen den kollaborativen Artefakten müssen analysiert und beschrieben werden. Nicht in den Artefakten enthaltene prozesslogische Informationen müssen erfasst werden und dienen für die Abstraktionsphase (vgl. Kapitel 6.4) als
- Auch ist zu prüfen, ob nur die Artefakte oder aber auch die Programmeinstellungen für die Konformitätsprüfung von Bedeutung sind. In diesem Fall müssen auch alle Werkzeuge und deren Einstellungsdateien identifiziert und zusätzlich in den Prüfraum aufgenommen werden.

Die Abfolge der Schritte kann wahlweise auch in einer anderen Reigenfolge durchgeführt werden. Ob die Schritte sequenziell oder parallel ausgeführt werden, ist ebenso frei wählbar. Schließlich müssen am Ende der Analysephase der *Prüfraum* sowie die *Selektion der Richtlinien* eindeutig festgelegt sein.

Anmerkung: In der Analysephase wird meistens schon ein Schritt hinsichtlich der Qualitätsverbesserung im Entwicklungsprozess vorgenommen. Wechselwirkungsaspekte, Analysen bzgl. der Nachvollziehbarkeit sowie aufgedeckte Auswirkungen des Prozesses auf die IT-Infrastruktur helfen, die Qualität der Richtliniendokumente bis hin zu den Prozessen entscheidend zu verbessern.

6.4 Abstraktionsphase

In der ersten Phase des VR-Modells wurden Prozesse, Richtlinien und Bezüge zu Artefakten sowie deren Abhängigkeiten untersucht. In der nun folgenden Phase werden die kollaborativen Artefakte des Prüfraums genau analysiert und für die regelbasierte Konformitätsprüfung in ein *logisches Artefakt* transformiert.

Die Transformation in ein logisches Artefakt hat mehrere Hintergründe und verfolgt unterschiedliche Zwecke. Um eine werkzeugübergreifende Prüfung an kollaborativen Artefakten durchführen zu können, müssen die ausgewählten Artefakte technisch normiert werden, d. h. einem identischen Dateiformat entsprechen und in dieses überführt werden, sofern sie sich nicht schon in diesem befinden. Beispiele für solch eine Transformation finden sich im Kapitel 7.3.3. Zusätzlich kann es aufgrund der vorangegangenen Analyse erforderlich werden, dass prozessspezifische oder prozesslogische Attribute (Semantik) zusätzlich zu den gewählten kollaborativen Artefakten angegeben werden, welche die Beziehungen zwischen den einzelnen Artefakten (vgl. logisches Artefakt, Kapitel 4.3) ausdrücken. Ein Artefakt kann beispielsweise zusätzlich eine Versionsnummer und eine Abteilungsinformation erhalten, welche wichtig für die übergreifende Prüfung sein können, da aufgrund dieser Zusatzinformationen andere Artefakte eine bestimmte Struktur aufweisen müssen.

Auch die Umkehrung kann gefordert sein, nämlich dass die Konformitätsprüfung auf einer Selektion, Projektion oder Aggregation von Artefakten durchgeführt wird (vgl. Kapitel 4.3.1). Ist es gewünscht, innerhalb der Regelsprache eine eigene, von den Daten des

originalen Artefakts verschiedene Semantik zu verwenden, die nicht von der Beschaffenheit des Artefakts (Metamodellstruktur) direkt abhängt, muss zusätzlich eine Semantik im logischen Artefakt definiert werden.

Die Abstraktion kollaborativer Artefakte in ein logisches Artefakt soll mittels *Transformation* erfolgen. Grundsätzlich verfolgt der in dieser Arbeit vorgestellte Ansatz zwei Möglichkeiten:

a) Eine *direkte* Transformation kollaborativer Artefakte.

b) Eine *indirekte* Transformation kollaborativer Artefakte.

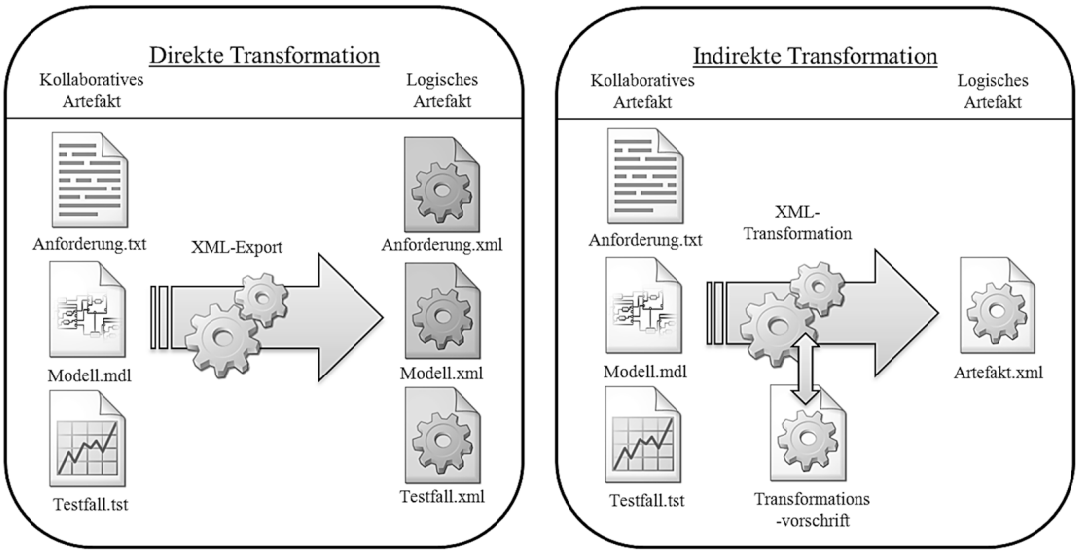


Abbildung 29: Direkte und indirekte Artefakt-Transformation

In beiden Fällen ergibt sich eine Abstraktion sowie Format-Vereinheitlichung zum plattformunabhängigen Modell (PIM) des logischen Artefakts/der logischen Artefakte. Diese unterscheiden sich jedoch durch die spezielle Transformationsvorschrift (vgl. Kapitel 2.4).

- Bei *direkter* Transformation wird nur eine Datenformatumsetzung vorgenommen. Das logische Artefakt ist somit im Metamodell und in der Semantik gleich mit dem ursprünglichen Artefakt. Alle kollaborativen Artefakte sind danach in einem identischen Dateiformat. Die Transformation kann typischerweise automatisiert durch einen Generator (Programm zur Transformation) erfolgen. Gängige Werkzeuge besitzen solche Exportmechanismen in ein offenes Format wie XML.
- Bei *indirekter* Transformation wird zunächst eine Transformationsvorschrift erstellt. Zur Beschreibung der Transformationsvorschrift wird die Metamodellierung aus Kapitel 2.4 verwendet. Die *indirekte* Transformation ist somit eine *Abstraktion* von den ursprünglichen Artefakten. Es ist ein manueller Schritt der Metamodellierung durch einen Experten. Das Metamodell bildet das logische Artefakt. Durch Werkzeugadapter kann dann eine Datenformatumsetzung aus dem Werkzeug in das logische Artefakt erfolgen.

Anmerkung: Viele Entwicklungswerkzeuge werden im Laufe der Jahre fortentwickelt (wie z. B. das Softwarepaket MS Office Version XP, Version 2003, Version 2007 usw.). Durch die Fortentwicklung ergeben sich Änderungen in den Metamodellen der Artefakte pro Programmversion. Dies würde auch die Struktur des logischen Artefakts und somit auch die

Regelalgorithmen betreffen. Schließlich müssten bei jeder neuen Programmversion ein neues logisches Artefakt und sogar der Regelkatalog überarbeitet werden. Diesen Aufwand vermeidet man, indem man das logische Artefakt konstant belässt und nur die Transformationsvorschrift an das neue Artefakt-Metamodell anpasst. Mit diesem Verfahren lässt sich eine regelbasierte Konformitätsprüfung über einen längeren Zeitraum gegen Versionsänderungen der Werkzeuge konstant halten.

Nach erfolgter Festlegung der Richtlinie und der Artefakte kann in der zweiten Phase des VR-Modells ein weiterer, optional durchzuführender Abstraktionsschritt für kollaborative Artefakte erfolgen, sofern die *indirekte* Transformation angewandt wird. Dies geschieht wahlweise durch die folgenden Schritte:

(1) Mittels *Selektion*, *Aggregation* oder *Deklaration* (semantische Neuzuordnung) artefaktspezifischer Daten soll von Daten des Artefakts abstrahiert werden. Es entsteht aus einem Artefakt ein transformiertes $Artefakt^T$.

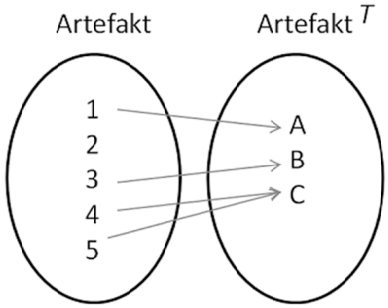


Abbildung 30: Surjektive Abbildung in ein logisches Artefakt

(2) Es soll ein logisches Artefakt aus definierten Eigenschaften oder Strukturen kollaborativer Artefakte für die Prüfung durch *Transformation* gebildet werden. Es entsteht aus Artefakten A, B ein transformiertes $Artefakt^T$.

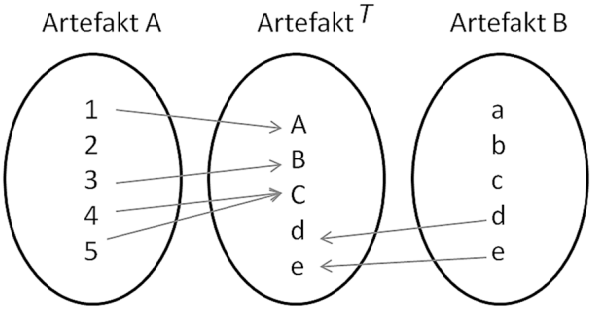


Abbildung 31: Multiple Abbildung in ein logisches Artefakt

Artefakte folgen einer inhärenten Struktur, sind in dieser persistent gespeichert. Man sagt, sie sind eine Instanz eines Metamodells (vgl. Kapitel 2.4). Das Metamodell gibt die Struktur vor, während die Instanz eine tatsächliche Ausprägung des Metamodells mit Wertebelegung darstellt (z. B. eine Datei, ein Modell oder ein Testfall).

In der Abbildung 32 wird prinzipiell veranschaulicht, dass jedem in einem Entwicklungswerkzeug geladenen Artefakt ein logisches Artefakt nach Metamodell bereits durch das verwendete Software-Werkzeug vorgegeben ist. Eine Artefakt-Instanz wird durch eine

direkte Transformation automatisiert erzeugt (Werkzeug-Exportfunktion) und folgt dem werkzeugspezifischen Datenmodell. Hierbei ist das logische Artefakt die direkte Abbildung (Instanz) des ursprünglichen Metamodells.

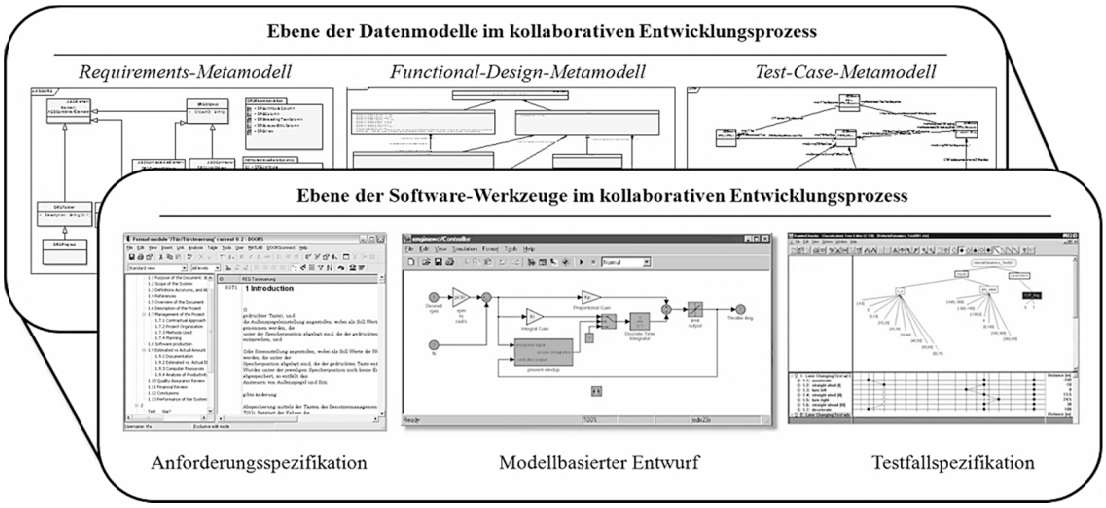


Abbildung 32: Artefakte haben werkzeugspezifische Metamodelle

Im zweiten Fall wird eine Instanz durch eine indirekte Transformation automatisiert erzeugt (Werkzeug-Wrapperfunktion, vgl. Kapitel 7.3) und folgt nicht mehr dem werkzeugspezifischen Datenmodell, sondern einem eigens vorgegebenen Metamodell. Das logische Artefakt ist entweder eine isomorphe oder eine surjektive Abbildung des ursprünglichen Metamodells. Die Abbildung 33 skizziert die beiden Wege der Generalisierung.

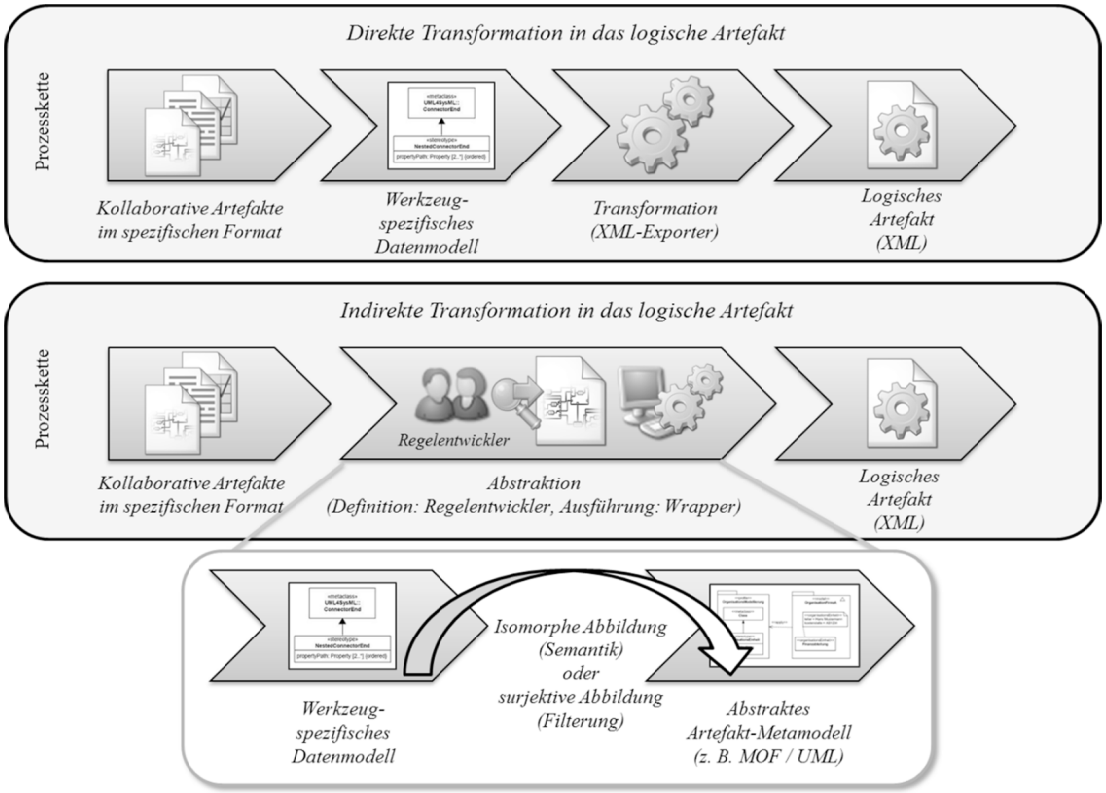


Abbildung 33: Artefakt- oder Benutzer-spezifische Generalisierung

Die Abbildung 33 zeigt ein direkt und ein indirekt transformiertes logisches Artefakt entstanden in einer Prozesskette. Ausgangspunkt sind in beiden Fällen die kollaborativen Artefakte im spezifischen Format.

In der oberen Prozesskette liegen zunächst ein oder mehrere kollaborative Artefakte im proprietären Dateiformat ihres Metamodells vor. Mit direkter Transformation wird dann das vom Artefakt vorgegebene Metamodell für das logische Artefakt verwendet und jede Artefakt-Instanz in eine XML-Struktur durch einen Exportmechanismus abgebildet. Es findet lediglich eine Transformation in eine offene Datenstruktur (hier XML) statt. Datum und Struktur sind für das logische Artefakt durch die Beschaffenheit des ursprünglichen Artefakts vorgegeben und müssen seitens der Regelprogrammierung (vgl. Kapitel 6.5.4) beachtet werden.

In der unteren Prozesskette wird ein zusätzlicher Verfahrensschritt der Abstraktion vorgenommen. Die Regelentwickler können für jedes kollaborative Artefakt ein eigenes Metamodell des logischen Artefakts definieren. Mit der Definition eines eigenen Metamodells muss auch die Transformationsvorschrift für das neue Metamodell festgelegt werden. Die von einem Automatismus (Werkzeug-Wrapper) ausgeführte Transformationsvorschrift ermöglicht eine isomorphe oder surjektive Abbildung des Quellmodells auf das logische Artefakt. Diese Prozesskette erlaubt es, auf einer höheren Abstraktionsstufe einen Regelalgorithmus auf dem logischen Artefakt zu formulieren. In dem eigenen Metamodell lassen sich auch prozesslogische Beziehungen oder Attribute abbilden, die sonst durch die kollaborativen Artefakte nicht gegeben sind. Wie bei direkter Transformation findet danach eine automatisierte Transformation in eine offene Datenstruktur (hier XML) statt.

6.4.1 Modellierung des logischen Artefakts

Auf dem *indirekten* Weg wird jedes Artefakt-spezifische Metamodell analysiert und ein abstraktes Metamodell des logischen Artefakts modelliert. Dies enthält sodann alle für die Richtlinienprüfung benötigten Elemente des ursprünglichen Artefakts und optional auch alle weiteren Informationen der kollaborativen Artefakte des Prüfraums. Durch Metamodelldefinition findet eine isomorphe oder surjektive Abbildung (Abstraktion) in das logische Artefakt-Metamodell statt, aus dem eine generalisierte Datenstruktur (hier XML) abgeleitet wird. Datum und Struktur sind durch das benutzerspezifische Metamodell vorgegeben und müssen seitens der Regelimplementierung beachtet werden. Im indirekten Weg wird zur Beschreibung der Transformationsvorschrift die Metamodellierung (vgl. Kapitel 2.4) verwendet und ein Metamodell des logischen Artefakts modelliert.

Während dieser Arbeit entstanden exemplarisch Metamodelle als logische Artefakte für die Werkzeuge DOORS, ML/SL/SF und CTE/XL. Diese konnten sowohl einzeln als auch in einem Super-Metamodell zusammengefasst werden (Automotive Software Development Metamodel, ASD). Für Beispiele der Metamodelle und Richtlinien sei auf vorherige Arbeiten (vgl. Veröffentlichungen) verwiesen. Zur Veranschaulichung sei dennoch ein kurzes Beispiel gezeigt. Es beschreibt eine werkzeugübergreifende Richtlinie mit prozesslogischer Artefaktverknüpfung und zeigt Auszüge aus dem entwickelten ASD-Metamodell.

Beispiel aus der modellbasierten Funktionsentwicklung

Zur Veranschaulichung der werkzeugübergreifenden Prüfung soll nun folgende Richtlinie umgesetzt werden: *„Jede im Anforderungsdokument beschriebene Systemfunktion des eingebetteten Systems muss im modellbasierten Entwurf (Funktionsdesign) durch mindestens*

einen Funktionsblock repräsentiert sein." Aus dieser Prozessvorgabe als Richtlinie erkennen wir eine prozesslogische Verknüpfung von dem Anforderungsartefakt und dem Artefakt des Funktionsdesigns.

Die Abstraktion vom werkzeugspezifischen Metamodell ist ein kreativer Denkprozess, der durch die Regelentwickler durchgeführt wird. Er ist anfangs einmalig erforderlich und bildet die Grundlage für eine zu entwickelnde Transformationsvorschrift, die durch einen (zu implementierenden) Werkzeug-Wrapper automatisiert ausgeführt werden kann.

Zu Beginn der Modellierung des eigenen Metamodells bietet es sich an, allgemeine Meta-Elemente (Klassen, Datentypen, Enumerationen usw.) zu modellieren, von denen jede weitere Artefakt-Klasse erben kann. Hat beispielsweise jedes Datum eines Artefakts einen eindeutigen Bezeichner oder im Allgemeinen eine Vorder- und Hintergrundfarbe, so kann dies grundlegend in einem werkzeugübergreifenden Metamodell (Kernel-Package) definiert werden. Die Abbildung 34 zeigt die *ASDElement-Klasse* von der andere Elemente erben können und der eine eigene Repräsentation zugeordnet wird.

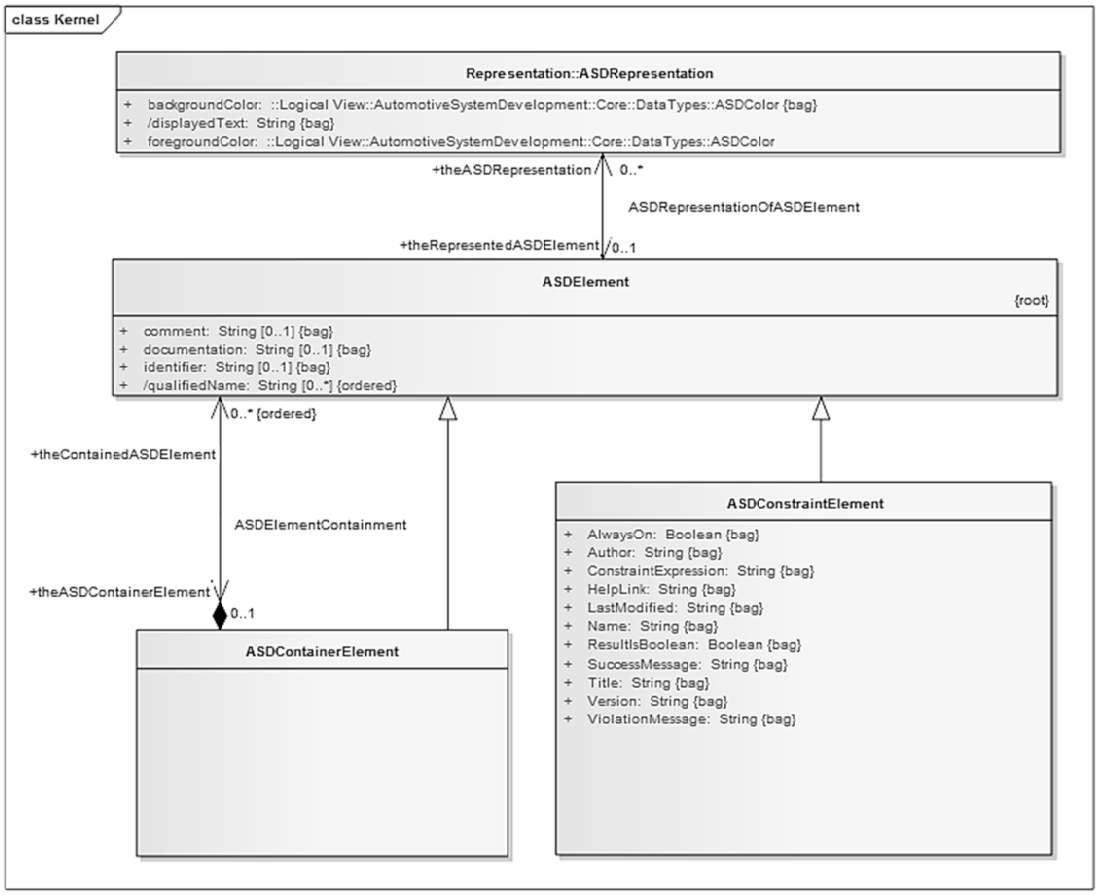


Abbildung 34: Metamodell für werkzeugübergreifende Eigenschaften

Für unser Beispiel ist hier das Attribut *identifier* ein wenig später von Bedeutung. Es repräsentiert den Bezeichner (Funktionsnamen) eines beliebigen Systemelements. Das *ASDConstraintElement* beschreibt alle Metainformationen zu einer Regel. Im Attribut *ConstraintExpression* wird der Regelalgorithmus in der Instanz gespeichert.

Sei in unserem Richtlinien-Beispiel nun die Anforderung mit dem Werkzeug DOORS (vgl. Kapitel 3.8.1) und das Funktionsdesign mit dem Modellierungswerkzeug Simulink erarbeitet (vgl. Kapitel 3.8.2) worden, so haben wir als Ausgangspunkt zwei Metamodelle

der beiden kollaborativen Artefakte für das logische Artefakt zu modellieren. Zur Metamodellierung benötigen wir zunächst nur eine recht reduzierte Artefakt-Struktur, welche zur Überprüfung der Richtlinie ausreichen soll. Wir müssen zunächst verstehen, wie der Funktionsname in einer Anforderung und wie er in einem Funktionsmodell im internen Datenmodell des jeweiligen Artefakts repräsentiert wird. Schließlich muss die Regel dann die Funktionsnamen in jedem Artefakt ausfindig machen und jeweils eine Übereinstimmung prüfen, um die Konformität zur Prozessvorgabe zu bestätigen.

Im nächsten Schritt würde nun für das DOORS-Artefakt ein Metamodell definiert werden. Die Struktur von DOORS gliedert Anforderungen in Module – es gibt hierbei unterschiedliche, auf die nicht weiter eingegangen werden soll. Wichtig für unsere Richtlinie sind die Attribute (*DRSAttribute*) einer Anforderung. In diesem ist der Funktionsname jeder Systemfunktion enthalten und in einer Artefakt-Instanz gespeichert. *DRSAttribute* steht in der Vererbungsbeziehung zu *ASDElement,* besitzt also das Attribut *identifier*.

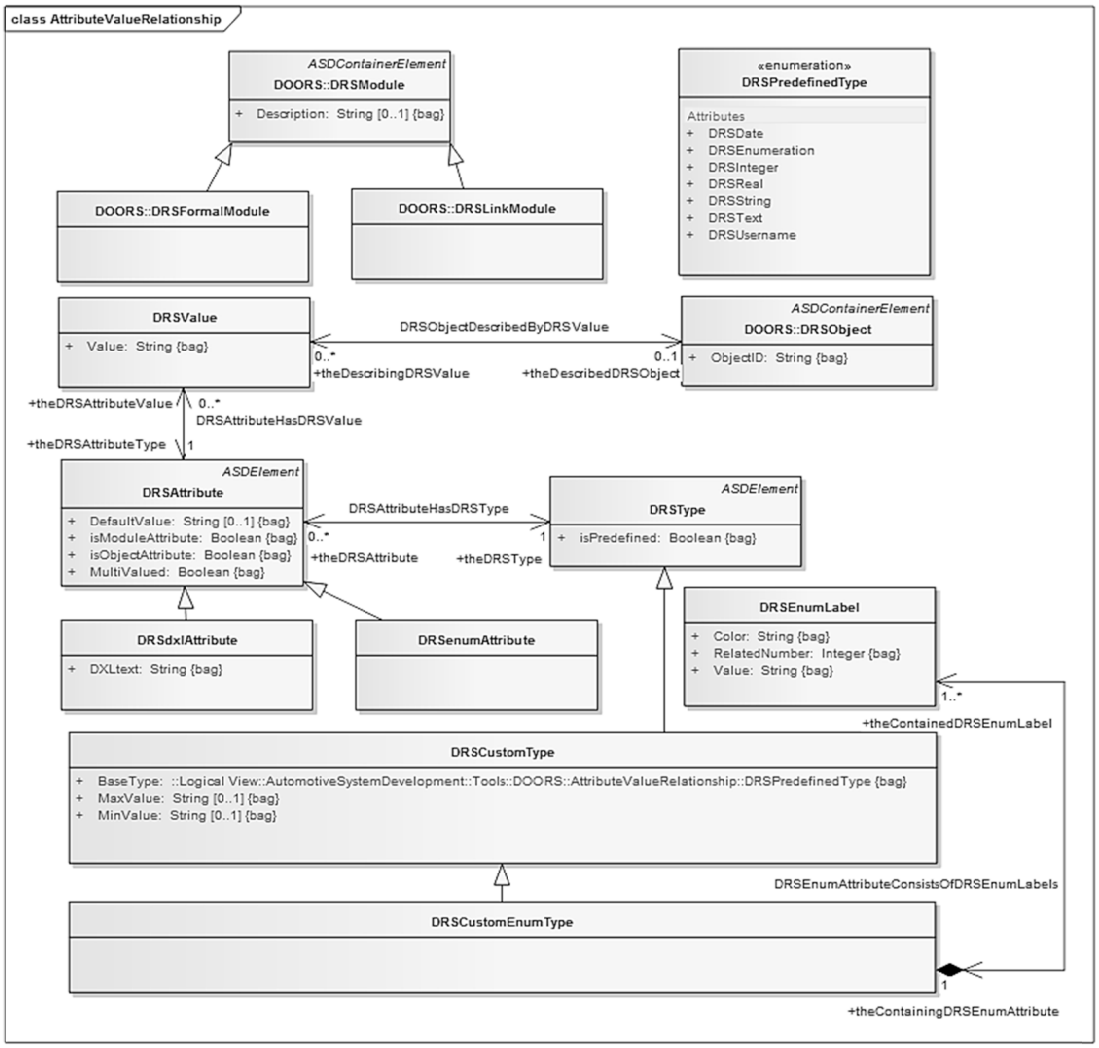


Abbildung 35: Metamodell des logischen Artefakts für DOORS (Auszug)

Im folgenden Schritt würde nun für Simulink ein Metamodell definiert werden. Die Struktur von Simulink (*SLStructure*) gliedert ein Funktionsmodell (*SLModel*) in eine Subsystemstruktur (*SLSystem*, *SLRootSystem*, *SLSubSystem*). In einem System können Funktionsblöcke (*SLBlock*), Signalverbinder (*SLLine*) sowie Schnittstellen (*SLPort*) auftreten. Für unsere

Richtlinie ist hierbei der Name der Funktionsblöcke entscheidend. Funktionsblöcke werden durch die Klasse *SLBlock* repräsentiert und erben ebenso das Attribut *identifier* vom *ASDElement*. Die Abbildung 34 zeigt ein benutzerspezifisches Metamodell (Auszug aus dem ASD-Metamodell) für das logische Artefakt eines Simulink-Modells. Nur für die Regelalgorithmen notwendige Elemente wurden hierbei modelliert. Zusätzliche Attribute oder Klassen wurden modelliert. Diese konnten logische Sachverhalte im Artefakt zusätzlich für andere Richtlinien beschreiben.

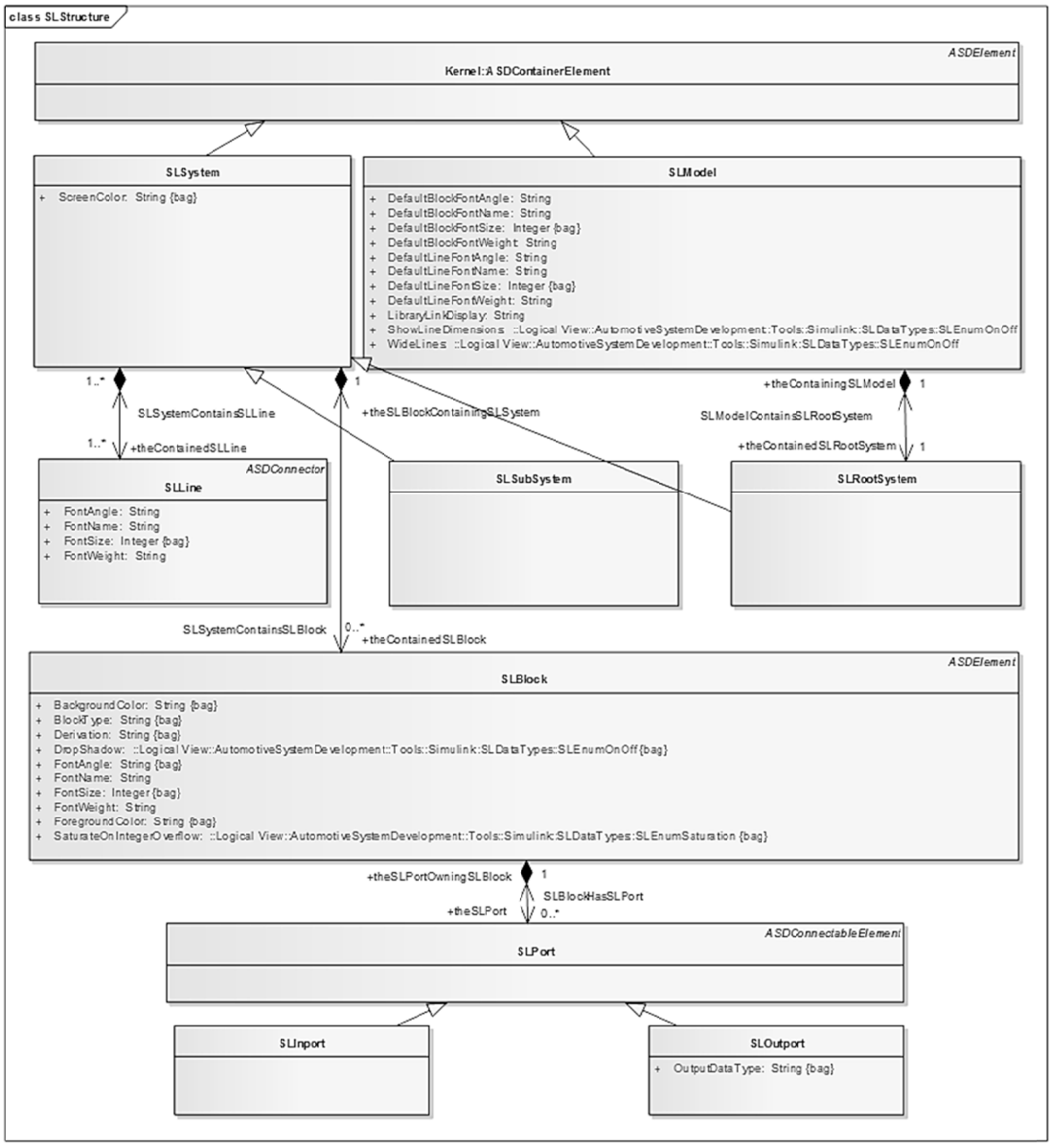


Abbildung 36: Metamodell des logischen Artefakts für Simulink (Auszug)

Da wir nun beide kollaborativen Artefakte durch ein logisches Artefakt als Metamodell definiert haben, müssen auf Instanzebene die beiden Artefakte (Anforderung und Funktionsmodell) in das logische Artefakt für eine Prüfung überführt werden. Dies kann auf Grundlage der Metamodelle automatisiert erfolgen und wird im Kapitel 7 näher erläutert.

Abschließend wollen wir die informale, werkzeugübergreifende Richtlinie als formale Regel (OCL-Ausdruck) im ASD-Metamodell beschreiben. Die Regel soll alle betreffenden

Elemente im Anforderungs-Artefakt und im Funktionsmodell jeweils auf Übereinstimmung durch Namensgleichheit prüfen.

Werkzeugübergreifende Konformitätsregel:

```
context AutomotiveSystemDevelopment inv:

 AutomotiveSystemDevelopment::Tools::DOORS::AttributeValueRelationship::
 DRSAttribute.allInstances()->select(x|x.identifier = 'Object
 Heading').theDRSAttributeValue->select(x|x.Value.substring(1,14) =
 'Systemfunktion ').Value->collect(x|x.substring(15, x.size()))->
 forAll(x|AutomotiveSystemDevelopment::Tools::Simulink::SLStructure::
 SLBlock.allInstances()->exists(y|y.identifier = x))
```

Zunächst gilt unsere Invariante im Kontext des gesamten logischen Artefakts (Metamodells). Wir fordern die Namensgleichheit mit dem *exists*-Schlüsselwort und vergleichen alle Attribute (*identifier*) der Instanzen von Simulink-Blöcken mit einer Auswahl an Attributen der Instanzen von Anforderungen. Um aus der Vielzahl von Anforderungsobjekten die richtigen Objekte herauszufinden, bei denen es sich tatsächlich auch um die Definition einer Systemfunktion handelt, benötigen wir eine Einschränkung. Diese wird dadurch erreicht, in dem nur die Anforderungsüberschriften (`identifier='Object Heading'`) mit einem Präfix (`x.Value.substring(1,14)='Systemfunktion'`) berücksichtigt werden sollen. Mit der gemachten Einschränkung wird sichergestellt, dass die Regel nur für Namen von Systemfunktionen greift.

Im Anhang – B werden weitere Artefakt-spezifische und -übergreifende Beispiele gezeigt.

Die *indirekte* Transformation ist somit eine *Abstraktion* und zugleich die *Generalisierung* von dem/den ursprünglichen proprietären Artefakt(en). Das Metamodell bildet die Struktur des logischen Artefakts. Durch Transformatoren (Werkzeugadapter) kann dann eine programmtechnische Umsetzung aus einem beliebigen Werkzeug in das logische Artefakt erfolgen (vgl. Kapitel 7.3). Prinzipiell ist dieser Ansatz vergleichbar mit dem RIF-Metamodell nach [RIF05] für das Anforderungsmanagement, in dem versucht wird, alle werkzeugspezifischen Aspekte durch prozesslogische Konstrukte zu ersetzen und daraus ein offenes Dateiformat zu gewinnen. Einen ähnlichen Ansatz verfolgten auch AUTOSAR [ASR09] und ASAM ODX [ASAM09] mit der Metamodellierung.

6.4.2 Auswahl der Transformation

Vor der Entwicklung eines Regelkatalogs ist in der Abstraktionsphase des V-Modells pro Artefakt die geeignete Transformationsart zu wählen. Direkt und indirekt transformierte Artefakte erlauben die Konstruktion eines logischen Artefakts. Vor- und Nachteile beider Verfahren müssen jedoch abgewägt werden. Der Vorteil der direkten Transformation liegt in der sehr schnellen Abarbeitung der Abstraktionsphase. Günstigerweise unterstützt bereits das Werkzeug einen XML-Export, sodass Artefakte im XML-Format vorliegen. In diesem Fall folgen sie dem herstellerspezifischen Metamodell. Der Aufwand für die Implementierung eines geeigneten XML-Wrapper (Wrapper beschrieben in Kapitel 7.3.1) für das Prüfsystem ist gering und kann für viele weitere Artefakte verwendet werden. Implementierungskosten und Wartungsaufwand sind hierbei niedrig. Nachteilig ist in diesem Fall jedoch, dass die

Regelentwicklung sich strikt an das herstellerspezifische Metamodell halten muss. Dadurch lassen sich die Regelalgorithmen nicht immer optimal formulieren, und die Regelalgorithmen sind sehr stark abhängig von dem verwendeten Werkzeug. Sollte dieses irgendwann durch eine aktuellere Version ersetzt werden (bei ML/SL/SF erscheinen fast halbjährlich neue Versionen), muss auch immer geprüft werden, ob alle Regelalgorithmen noch funktionieren bzw. kompatibel zum neuen Format sind. Eine Portierung von z. B. MISRA-Regeln vom Modellierungswerkzeug ML/SL/SF hin zu ACSET würde nur durch eine Neuentwicklung der Regeln funktionieren.

Dies bildet wiederum die Stärke eines eigenen Metamodells und der indirekten Transformation eines Artefakts in das logische Artefakt. Das logische Artefakt kann auch bei einer Versionsänderung eines herstellerspezifischen Metamodells konstant bleiben. Dadurch ist die Kompatibilität des Regelkatalogs stets gewährleistet. Trotzdem muss beim Versionssprung untersucht werden, ob die Transformationsvorschrift noch zum gleichen Resultat führt. Dies ist jedoch sehr viel einfacher, als einen umfangreichen Regelkatalog auf Kompatibilität zu überprüfen. Durch indirekte Transformation lassen sich die Regelalgorithmen optimal formulieren (im Metamodell können zusätzliche Informationen modelliert und unnötige ausgelassen werden), und die Regelalgorithmen sind unabhängig von der verwendeten Werkzeug-Version. Ein Nachteil bei diesem Verfahren kristallisiert sich in der technischen Umsetzung heraus. Meist muss ein spezieller Wrapper entwickelt werden, der die Transformationsvorschrift auf dem Artefakt ausführt. Ändert sich die Transformationsvorschrift, muss auch der Wrapper angepasst werden. Die Wartung und die Anpassungen können bei vielen zu unterstützenden Artefakten schnell einen hohen Aufwand bedingen. Aus den im Forschungsprojekt MESA gewonnenen Erfahrungen ist es zur Minderung des Nachteils bei indirekter Transformation immer ratsam, die Transformationsvorschrift von der technischen Implementierung weitestgehend zu entkoppeln. Ein Beispiel ist ein regelbasierter Transformationsansatz, bei dem immer nur eine aktualisierte Regel dynamisch vom Wrapper geladen und ausgeführt wird. Weiterführende technische Aspekte von Transformationen beleuchtet das Kapitel 7.3.

6.5 Formalisierungsphase

In der vorherigen Phase wurde mittels Abstraktion und Generalisierung eine Formalisierung kollaborativer Artefakte in ein logisches Artefakt vorgenommen. Für die automatisierte Konformitätsprüfung von Entwicklungsrichtlinien an dem logischen Artefakt ist für die Bildung eines Prüfalgorithmus nach diesem Ansatz auch eine Formalisierung der natürlich sprachlichen Richtlinien erforderlich. Dies geschieht in der Formalisierungsphase des VR-Modells in den jeweiligen Schritten Partitionierung, semantische Analyse und Bildung des Prüfalgorithmus.

6.5.1 Partitionierung

Zunächst wird in der Analyse anhand einer gegebenen, natürlich sprachlich formulierten Richtlinie analysiert, ob diese sich in mehrere logische Teile partitionieren lässt. Eine Richtlinie zur einheitlichen Farbwahl in Modellen (Darstellung von Modellen) kann beispielsweise eine separate Richtlinie zur Prüfung der Vordergrundfarbe und zur Prüfung der Hintergrundfarbe zerfallen. Noch feingranularer wäre dies sogar der Fall, wenn die Farbcodierung eine bestimmte Semantik für Objekte darstellt. Zudem sind Vor- und Nachbedingungen und Auswirkungen der Richtlinie voneinander zu separieren. Somit kann

eine dokumentierte Richtlinie für die regelbasierte Konformitätsprüfung in mehrere Richtlinien für die Ausführung partitioniert werden.

6.5.2 Semantische Analyse

In dieser Phase des VR-Modells wird eine semantische Textanalyse der natürlich sprachlichen Richtlinie durchgeführt. In der Untersuchung werden der Richtlinienkontext sowie die resultierende Aufgabenausführung ermittelt. Es werden zusätzliche Anforderungen von der Richtlinie aufgestellt, welche für die Prüfkriterien gelten. Bei der semantischen Analyse werden semantische Bausteine im Richtlinientext gesucht, wie

- *Termini, Nomen, Verben,*
- *Synonyme und Homonyme.*

Insbesondere ist hierbei eine *Klassifizierung* vorzunehmen. Kollaborative Artefakte werden implizit durch die Regel genannt und müssen zur tatsächlichen IT-Infrastruktur in Bezug gesetzt werden.

Beispiel:

Eine Richtlinie gilt für den Prozess *Rechnungsstellung*. Ein ‚Angebot' kann in diesem Prozesskontext beispielsweise immer eine MS Excel Tabelle (Datei) sein. Daher bietet sich eine Klassifizierung der Bausteine eines Richtlinientextes an. Eine aus der Praxis gewonnene Klassifizierung von Bausteinen in natürlich sprachlichen Richtlinien ist folgende: *Prozess, Vorgang, Artefakt (Prädikat), typisierte Artefakt-Eigenschaft, Aktion, Ereignis, Konsistenz-relation, Bedingungstyp (Prädikatenlogik), Quantor (Existenz-/All-quantor) und Resultat.*

Die Beziehungen von Richtlinie zu Artefakt sowie die verborgene Semantik zeigt die Abbildung 37. Hier wird durch die farbliche Unterstreichung eine Klassifizierung vorgenommen. Füllwörter und Satzverbinder entfallen, da sie für die spätere Formalisierung keine Bedeutung besitzen.

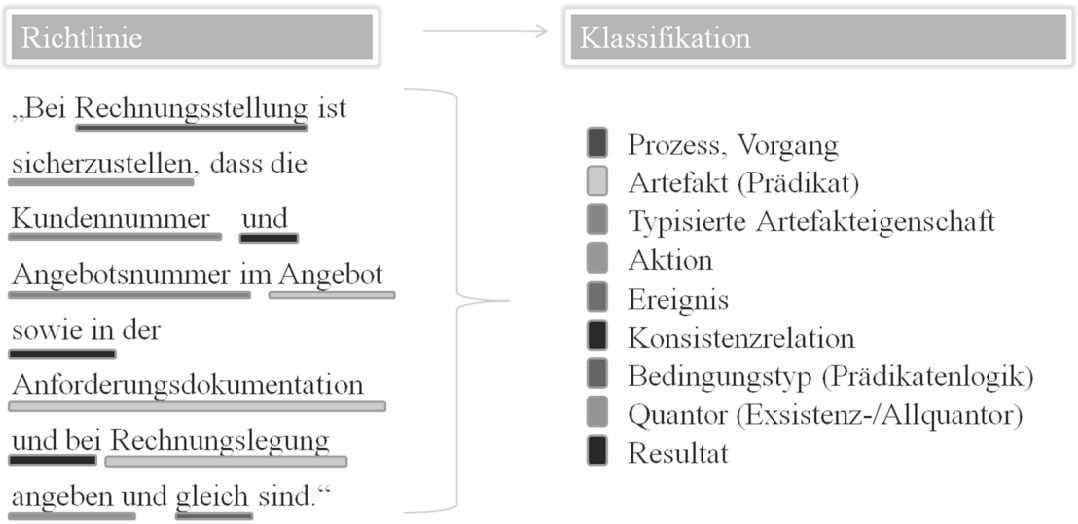


Abbildung 37: Beispiel - Semantische Analyse einer textuellen Richtlinie

Nach den vorbereitenden Klassifikationsschritten können nun der Prüfraum exakter sowie das Prüfergebnis festgelegt werden. Mittels Prädikatenlogik erster Ordnung (vgl. Kapitel 2.3.3) wird die textuelle Richtlinie in eine formalisierte Richtlinie gemäß unserer semantischen Klassifizierung durchgeführt:

Prüfraum, Prüfaufbau:	Die „Rechnungsstellung“ ist ein Prozess $\boldsymbol{P}$ mit einem Prüfraum $\boldsymbol{\Omega}$ in dem alle kollaborativen Artefakte enthalten sind. So besteht der Prüfaufbau $\boldsymbol{\Delta}$ aus ($\boldsymbol{A} \times \boldsymbol{D} \times \boldsymbol{R}$), wobei für $\boldsymbol{A}$, $\boldsymbol{D}$ und $\boldsymbol{R}$ gilt: Menge aller Angebote: $\boldsymbol{A} := \{ \boldsymbol{x}_i \in \boldsymbol{\Omega}: ist\_Angebot(\boldsymbol{x}) \}$ Menge aller Dokumentationen: $\boldsymbol{D} := \{ \boldsymbol{x}_j \in \boldsymbol{\Omega}: ist\_Dokumentation(\boldsymbol{x}) \}$ Menge aller Rechnungen: $\boldsymbol{R} := \{ \boldsymbol{x}_k \in \boldsymbol{\Omega}: ist\_Rechnung(\boldsymbol{x}) \}$ mit Indizes $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k} \in \mathbb{N}$.
Formalisierte Richtlinie:	$\forall \boldsymbol{x} \in \boldsymbol{\Delta}$: $((hat\_AngebotNr(\boldsymbol{x}_i) = hat\_AngebotNr (\boldsymbol{x}_j) = hat\_AngebotNr (\boldsymbol{x}_k)$ $\wedge (hat\_KundenNr(\boldsymbol{x}_i) = hat\_KundenNr(\boldsymbol{x}_j) = hat\_KundenNr(\boldsymbol{x}_k)))$
Prüfergebnis:	$Prüfung(x) = \begin{cases} \emptyset, & konform \\ sonst, & nonkonform \end{cases}$

Nach Definition des Prüfraums und des Prüfaufbaus wurde die natürlich sprachliche Richtlinie in einen logischen Ausdruck überführt. Aus den Anforderungen einer Richtlinie liegt nun eine *Regel* vor. Wie bereits erwähnt, können am Ende dieser Schritte auch mehrere Regeln aus der Anforderung einer Richtlinie entstehen.

6.5.3 QRDL-Regelalgorithmus

Der letzte Schritt der Formalisierungsphase besteht aus der Bildung des Prüfalgorithmus aus den formalisierten Logikausdrücken der textuellen Richtlinie. Der Algorithmus wird gemäß der in den vorangegangenen Schritten gewonnenen Vorbedingungen, der Nachbedingungen und der Abhängigkeiten deduktiv entwickelt. Der folgende Prüfalgorithmus ist dabei abfrageorientiert durch die QRDL strukturiert:

Beispiel: ***QRDL-Regelalgorithmus der Konformitätsprüfung kollaborativer Artefakte***

```
01. <BEGIN RULE>
02.
03.      <DECLARATIONS>
04.
05.      ARTIFACT A := ANGEBOT; D := DOKUMENTATION; R := RECHNUNG;
06.
07.      OBJECT  (PRÜFUNG, QUERY_1, QUERY_2, QUERY_3) := null;
08.      OBJECT  (x, y, z) := null;
09.
10.      STRING  (Angebotsnummer) := '1099';
11.      STRING  (Kundennummer) := '54');
```

```
12.
13.      </DECLARATIONS>
14.
15.      <EXAMINATIONS>
16.
17.      <QUERY>
18.      /* PRE-CONDITION: x == null */
19.      QUERY_1 ⊆ ∀ x ∈ A
20.      /* PRE-CONDITION: x != null, sonst Konsistenzverletzung */
21.               Angebotnr(A) == Angebotsnummer ∧
22.               Rechnungnr(A) == Kundennummer
23.      SELECT x
24.      /* POST-CONDITION: QUERY_1 = x */
25.      </QUERY>
26.
27.      <QUERY>
28.      /* PRE-CONDITION: y == null */
29.      QUERY_2 ⊆ ∀ y ∈ D
30.      /* PRE-CONDITION: y != null, sonst Konsistenzverletzung */
31.               Angebotnr(D) == Angebotsnummer ∧
32.               Rechnungnr(D) == Kundennummer
33.      SELECT y
34.      /* POST-CONDITION: QUERY_2 = y */
35.      </QUERY>
36.
37.      <QUERY>
38.      /* PRE-CONDITION: z == null */
39.      QUERY_3 ⊆ ∀ z ∈ R
40.      /* PRE-CONDITION: z != null, sonst Konsistenzverletzung */
41.               Angebotnr(R) == Angebotsnummer ∧
42.               Rechnungnr(R) == Kundennummer
43.      SELECT z
44.      /* POST-CONDITION: QUERY_3 = z */
45.      </QUERY>
46.
47.      </EXAMINATIONS>
48.
49.      <ESTIMATION> /* OVERALL COMPLIANCE CHECK */
50.               PRÜFUNG := QUERY_1 ∧ QUERY_2 ∧ QUERY_3;
51.      </ESTIMATION>
52.
53.      <COMPLIANCE RESULT> /* RESULT */
54.
55.      IF   Elements(PRÜFUNG) == null THEN
56.               RETRUN Pass;  /* compliance */
57.      ELSE
58.               RETRUN Fail; /* non-compliance */
59.      END IF
60.
61.      </COMPLIANCE RESULT>
62.
63. <END RULE>
```

Der QRDL-Regelalgorithmus wird durch eine Klammerung *<BEGIN RULE>* und *<END RULE>* syntaktisch eingegrenzt. Eine *Variable* ist ein Bezeichner für einen Platzhalter im Speicher, welcher typabhängig eine bestimmte maximale Größe hat und stets verändert werden kann. Eine *Konstante* ist ein Bezeichner für einen Platzhalter im Speicher, welcher typabhängig eine bestimmte maximale Größe hat und nicht mehr verändert werden kann. Es folgen zu Beginn allgemeine Variable- und Konstante-Deklarationen für die Regel, wie die zunächst leeren Mengen ***PRÜFUNG***, ***QUERY_1***, ***QUERY_2***, ***QUERY_3***, variable Objekte ***x***, ***y***, ***z***, welche jeweils ein variables Datum zur Laufzeit des Prüfalgorithmus darstellen, die typisierten logischen Artefakte ***A, D, R*** mit Referenz zu ihnen sowie die

Konstanten ***Angebotsnummer*** und ***Kundennummer***, welche für Vergleichsoperationen innerhalb des Prüfalgorithmus benötigt werden.

Für jedes logische Artefakt erfolgt dann sequenziell eine separate, mengenorientierte Abfrage (*QUERY_1, QUERY_2, QUERY_3*) auf die Elemente des jeweiligen logischen Artefakts, welche die Regel erfüllen müssen. Zusätzlich sind Vor- und Nachbedingungen für die Abfrage in Form von Kommentaren angegeben, welche später auch implementiert werden können. In dem Teilausdruck der Abfrage wird die Konformitätsprüfung (logischer Ausdruck) formal beschrieben.

In diesem Beispiel wird für jedes beliebige Element ***x*** in dem jeweiligen logischen Artefakt geprüft, ob die vorher definierte ***Angebotsnummer*** und ***Kundennummer*** konform zueinander sind bzw. hierbei übereinstimmen (Äquivalenz). Die Vorbedingung, dass das jeweilige logische Artefakt überhaupt Elemente besitzt, ist eine Konsistenzbedingung, die nicht gegeben wäre, wenn ***x != null***, also eine leere Menge auf dem logischen Artefakt ist, wie auch bei den anderen Abfragen für ***y*** und ***z***. Dies ist als Kommentar eingefügt und dient später bei den Regelimplementierungen als Hinweis für die Vorprüfung, dass die im logischen Artefakt enthaltenen Elemente mindestens ein Datum für die gültige Konformitätsprüfung besitzen. Ist die jeweilige Konformitätsprüfung erfüllt, existiert also mindestens ein Datum im logischen Artefakt, das jeweils pro Artefakt im Objekt ***x***, ***y***, ***z*** gespeichert wird. Die jeweilige Abfrage ist demnach erfüllt, wenn sie mindestens ein Datum enthält, und die Regel ist nicht erfüllt, wenn sie eine leere Menge ist. Nach den drei Abfragen erfolgt nun die Prüfung durch *Aussagenlogik* im Abschnitt `<ESTIMATION>`. Sind alle Abfragen verschieden von der leeren Menge und enthalten dieselben Daten, wird durch logische Und-Verknüpfung (*AND*) die Mengengleichheit gebildet und das Prüfergebnis ***PRÜFUNG*** ermittelt. ***PRÜFUNG*** ist somit die Teilmenge der Menge aller Daten in den logischen Artefakten, für welche die Konformität zutraf.

Im letzten QRDL-Abschnitt `<COMPLIANCE RESULT>` muss das Prüfresultat mittels Fallunterscheidung IF, ELSE, END IF gebildet werden. Dies geschieht durch einen *prädikatenlogischen Ausdruck* ***Elements(PRÜFUNG) == null***, welcher unterscheidet, ob das Prüfergebnis ***PRÜFUNG*** entweder gleich oder verschieden der leeren Menge ist. Bei Gleichheit wird der Entschluss gefasst, dass die Konformität gegeben ist. Es wird als Rückgabewert RETURN die Fehlerkategorie [PASS] festgelegt. Bei Ungleichheit wird der Entschluss gefasst, dass die Konformität nicht gegeben ist. Es wird als Rückgabewert RETURN die Fehlerkategorie [FAIL] festgelegt. Hier sind auch andere, aus der Analysephase ermittelte, Fehlerkategorien möglich (z. B. [ERROR], [WRANING], [INFORMATION] usw.).

In dieser Phase des VR-Modells wird jeder Algorithmus in einem Prüfalgorithmus in der QRDL als *Regel* implementiert. Die hier gezeigte QRDL-Regel zeigt exemplarisch die formalisierte Konformitätsprüfung kollaborativer Artefakte. Im nächsten Schritt kann aus der QRDL-Regel eine beliebige, hinreichend mächtige Prüfsprache (Programmiersprache) für die Regelimplementierung verwendet werden.

6.5.4 Regelprogrammierung

Die anschließende Implementierung der Richtlinie aus der QRDL-Regel überführt den Algorithmus der QRDL-Regel in eine computerausführbare Programmiersprache, damit die Regel automatisiert durch einen Interpreter und Compiler ausgeführt werden kann (vgl. Kapitel 7.2). Die Programmiersprache kann dabei beliebig gewählt werden. Sie muss jedoch die aufgestellten Prüfanforderungen (vgl. Kapitel 4.1.2 und 5.2) erfüllen.

In dieser Arbeit wurden die Sprachen LINQ, M-Skript und interpretierte OCL-Ausdrücke für die Regelprogrammierung exemplarisch gewählt und erfolgreich für den Einsatz erprobt. Mit jeder Sprache ließen sich umfangreiche Regelkataloge erstellen. Die Versuche mit M-Skript und OCL sowie die Ausdrucksmächtigkeit beider Sprachen sind in vorangegangenen Arbeiten untersucht und beschrieben worden (vgl. Veröffentlichungen, Nr. **1**, 2, 23, 24). Die Ausdrucksmächtigkeit von LINQ wird ausführlich im Kapitel 7.4 untersucht.

Gegen Ende der Formalisierungsphase des VR-Modells erfolgt somit die Wahl einer geeigneten Implementierungssprache für die Regelalgorithmen. Die verschiedenen Technologien bieten empfundene Vor- und Nachteile. Eine Auswahlhilfe sei daher aufgrund der verschiedenen Sprachaspekte mit nachfolgender Tabelle 19 gegeben.

Tabelle 19: Gewählte Sprachen zur Regelimplementierung im Vergleich

Eigenschaft	*LINQ*	*OCL*	*M-Skript*
Typ	Abfragesprache	Ausdruckssprache	Programmiersprache
Schlüsselwörter	vordefiniert	vordefiniert	vordefiniert
Boolesche Ausdrücke	ja	ja	ja
Abhängigkeit	kontextabhängig	kontextabhängig	Kontext unabhängig
Invariante	abfrageorientiert	objektorientiert	funktionsorientiert
Pre-/Postconditions	nicht integriert	integriert	nicht integriert
Vergleichsoperatoren	vorhanden	vorhanden	vorhanden
Traversierung	über Objektbeziehungen möglich	über Objektbeziehungen möglich	nur bei selbstdefinierten Strukturen (eigene Datentypen) möglich
Variablendeklaration	ja	ja	ja
Typsicherheit	ja	ja	nein
Typwandlung	ja	ja	ja
Kollektionen	Datentyp, Operationen	Datentyp, Operationen	Iterator-Operationen
Seiteneffekte	kann Objektzustand verändern	kann Objektzustand nicht verändern	kann Objektzustand verändern
Berechnung	integriert, ausreichend ausgeprägt	integriert, wenig ausgeprägt	integriert, sehr gut ausgeprägt
Kontrollstrukturen	ja	ja	ja
Empfundene Vorteile	- gute Lesbarkeit, - Abstraktion vom Datenmodell, - schnelle Ausführung	- kompakte Form der Ausdrücke, - rekursive Ausdrücke	- umfangreiche Berechnungsfunktionen, - Funktionsbibliotheken
Empfundene Nachteile	- Abfragen können bei steigender Komplexität schwer nachvollziehbar werden, - starke Abhängigkeit zur .NET-Sprache / Vermischung	- Abhängigkeit zum Datenmodell, - beschränkt auf lokalisierte Elemente, - keine selbstdefinierte Funktionsbibliotheken - schlechte Lesbarkeit bei komplexen Ausdrücken	- kein Zugriff auf ein objektorientiertes Datenmodell möglich, - Artefaktspezifische Regelimplementierung (nur ML/SL/SF), - mäßige Unterstützung von XML-Integration

Ein Vergleich der Sprachmächtigkeit der drei Sprachen bedarf sicherlich eines eigenen Kapitels und kann aus Platzgründen hier nicht weiter vertieft werden. Zusammengefasst lässt sich feststellen, dass eine formale Beschreibung der in dieser Arbeit ausgewählten Richtlinien mit den Sprachen möglich war. Der einfache Zugriff auf das zugrunde liegende Datenmodell (Traversierung der Objektbeziehungen) des logischen Artefakts ist ein großer Vorteil bei den Sprachen LINQ und OCL und sollte deshalb noch einmal genau gegen die

Nachteile der eingeschränkten Verständlichkeit bei komplexen Abfragen abgewogen werden. Die größte Flexibilität offeriert jedoch LINQ durch seine Abstraktionsfähigkeit vom Datenmodell. In OCL und M-Skript muss das zugrunde liegende Datenmodell zunächst sehr umständlich aufbereitet werden, bevor sich der Prüfalgorithmus effizient implementieren lässt.

6.6 Konformitätsprüfungsphase

Mehrere Regeln können, hierarchisch in einer Menge in Relation stehend, verknüpft werden. Das Ergebnis ist ein digitaler Regelkatalog (vgl. Kapitel 4.4.1), der durch ein Prüfwerkzeug auf dem logischen Artefakt ausgeführt werden kann. Dies ist im Kapitel 7 weiter beschrieben.

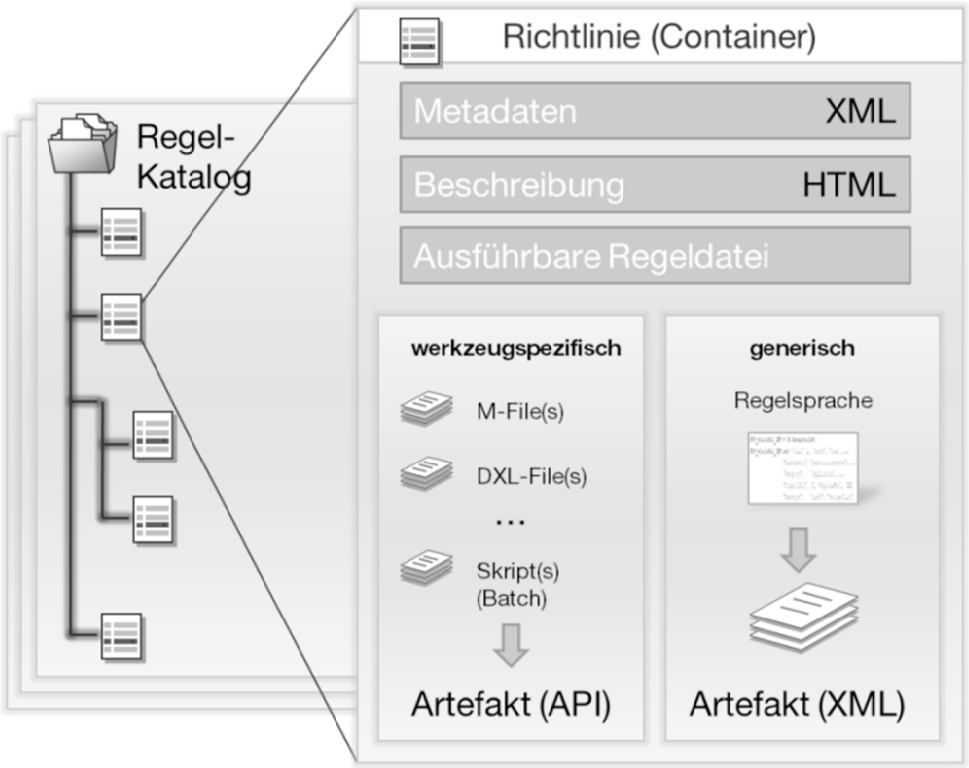


Abbildung 38: Aufbau eines Regelkatalogs

In diesem Ansatz ist die ausführbare Richtlinie als eine Art ‚Container' aufgebaut. Dieser beinhaltet zwecks der späteren Regel-Verwaltung die zugehörigen Metadaten (z. B. nicht in der Regel enthaltene Zusatzinformationen), eine Beschreibung der Richtlinie (natürlich sprachlicher Text aus der Analysephase) und der ausführbaren Regeldatei(en) in Form der implementierten QRDL-Regel (Algorithmus).

Zur Durchführung der Konformitätsprüfung wird zunächst aus einem gewählten Prüfraum die Prüfkonfiguration festgelegt. Der Prüfraum ergibt sich aus der Analyse-Phase (VR-Modell), der vorherigen Prozess- und Artefakt-Analyse. Die Prüfkonfiguration besteht aus einem festgelegten Satz an kollaborativen Artefakten, die vor einer Prüfung ausgewählt und festgelegt wird. Des Weiteren werden in der Konformitätsprüfungsphase ein digitaler Regelkatalog und eine ausgewählte Teilmenge enthaltener Regeln bestimmt. Durch eine Ausführungsumgebung wird sodann das logische Artefakt durch Transformation gebildet und die gewählten Regeln aus dem digitalen Regelkatalog auf dem logischen Artefakt

ausgeführt. Die automatische Ausführung des digitalen Regelkatalogs mittels Interpreter bzw. Compiler ist ein wesentlicher Arbeitsschritt im Teilschritt der Konformitätsprüfung.

6.7 Auswertungsphase

Nach einem Prüfdurchlauf werden die Ergebnisse in einem *Prüfbericht* (Report) durch eine möglichst automatisierte Berichterstattung zusammengefasst und protokollartig aufgeführt. Ein Anwender erhält hierdurch die Möglichkeit zu sehen, bei welchem Artefakt eine Konformitätsverletzung vorlag oder bei welchem Artefakt diese erfolgreich bewertet wurden. Bei einem negativen Ergebnis wird durch die Auswertung eine Liste der problematischen Artefakt-Elemente aufgebaut. Im Falle einer direkten Verknüpfung mit dem logischen Artefakt kann der Anwender diese Elemente durch Anklicken automatisiert im ursprünglichen Artefakt (Dokument, Modell usw.) auffinden.

Für die Auswertung der Prüfergebnisse eignen sich *Fehlerklassen*. Tabelle 20 führt sinnvolle Ergebniskategorien durch die Definition von Fehlerklassen ein, ohne den Anspruch auf Vollständigkeit zu erheben.

Tabelle 20: Fehlerklassen

Fehlerklasse	*Bedeutung*
PASS	Die Konformität wurde nachgewiesen.
FAIL	Die Konformität wurde nicht nachgewiesen.
WARNING	Die Konformität wurde mit Einschränkung nachgewiesen.
INFO	Die Konformitätsprüfung wurde durchgeführt. Alle Vor- und Nachbedingungen wurden erfüllt.
ERROR	Die Konformitätsprüfung konnte nicht durchgeführt werden, da eine Vor- oder Nachbedingung nicht eingehalten wurde.
CUSTOM	Eine durch den Regelentwickler festgelegte, spezielle Prüfnachricht.
NONE	Die Konformität konnte nicht nachgewiesen werden aufgrund einer Konsistenzverletzung (z. B. logisches Artefakt leer usw.).

Eine tabellarische und grafische Auswertung zeigt Abbildung 39.

Aus den Prüfdurchläufen ergeben sich jeweils einzelne Prüfprotokolle, aus denen Konformitätsverletzungen durch Fehler oder Hinweise ersichtlich werden, die wiederum Aktionen in der Prozesskette bedingen oder die nachträgliche Anpassung der Richtlinie erfordern. In der gezeigten Auswertung (Abbildung 39) wird dargestellt, wie viele Richtlinien im Prüfdurchlauf ausgeführt wurden, welches Ergebnis sie hatten, wie viele Daten einen Konformitätsverstoß aufweisen und wie die prozentuale Gewichtung (Regel zu Kategorie) ausfällt.

Ergebnis	Umfang	Prozent (%)	Anzahl der Verstöße	Prozentsatz (%)
Bestanden	17	77.27	0	0.00
Information	3	13.64	3	60.00
Warnung	2	9.09	2	40.00

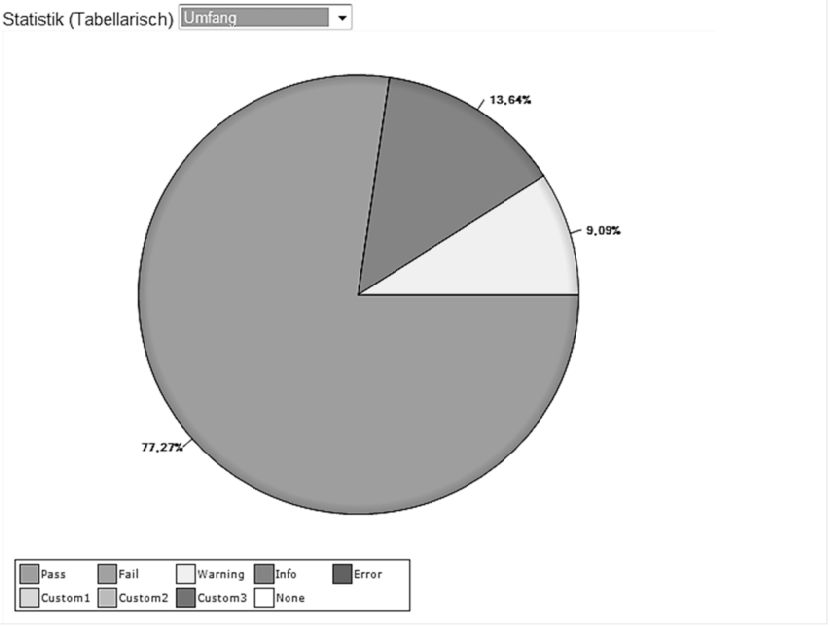


Abbildung 39: Auswertung eines Prüfdurchlaufs nach Fehlerklassen

6.8 Optimierungsphase

Während traditionelle Verfahren zur Qualitätssicherung in der Software-Entwicklung häufig Qualitätserfordernisse an Artefakten im Nachhinein prüfen, liegt der Kerngedanke der präventiven Qualitätssicherung in den vorbeugenden Maßnahmen zur frühestmöglichen Vermeidung von Qualitätseinbußen auf Grundlage von Erfahrungswerten aus der Vergangenheit. Der wesentliche Nutzen ist demzufolge die Reduktion von Kosten, welche mit exponentieller Steigung zunehmen, je später sie im Entwicklungsprozess bemerkt werden. Nach [VERS06] ist aus der Praxis eine sogenannte 10er-Regel ableitbar, welche den potenziellen Anstieg des Kostenfaktors um das Zehnfache pro Phase charakterisiert, in der ein Fehler aufgedeckt wird.

6.8.1 Metriken

Um den vorher genannten Aspekten entgegenzuwirken, muss in der Optimierungsphase ein präventiver Mechanismus vorgesehen sein. Das VR-Modell erlaubt (als solchen Mechanismus) die Kennzahlen (*Metriken*) aus Erfahrungswerten festzulegen und mit diesen das logische Artefakt nach einstellbaren Kategorien, in Analogie zu den Fehlerklassen, automatisch zu bewerten. Die regelbasierte Konformitätsprüfung ermöglicht durch die Algorithmen eine Artefakt-Berechnung auf Basis von Kennzahlen. Dies dient der Unterstützung bei der Bewertung von Entwicklungsprozessen und Arbeitsergebnissen für die vorhersagbare Qualitätsoptimierung. Hierdurch lässt sich ermitteln, ob beispielsweise ein logisches Artefakt in einer konzeptionellen Stufe oder bereits in einer Fassung zur

Weitergabe bzw. Freigabe vorliegt. Bewertungen hinsichtlich Effizienz oder Stabilität sowie Wartbarkeit sind weitere sehr nützliche Metriken, um eine Einschätzung nicht-funktionaler Qualitätseigenschaften vorzunehmen.

In der Arbeit wurden Metriken festgelegt und die Ermittlung als Regel-Algorithmus entwickelt, welcher jeweils eine Aussage über die Effizienz von generiertem Code aus Implementierungsmodellen – also präventiv vor der Codegenerierung – zulässt. Hierzu zählen die im Modell verwendeten Datentypen (Anzahl und Bytegröße), die einstellbaren Wertebereiche der Datentypen sowie die Berechnung von mathematischen Operationen, wie die Vermeidung von Mehrfachberechnungen oder die ungünstige Verwendung von Divisionen anstelle der Multiplikationsoperationen. Beispiele sind im Anhang - B gegeben.

6.8.2 Erfahrungswerte

In der Phase der Optimierung (VR-Modell) wird versucht, Richtlinienkataloge aus gewonnenen Erfahrungswerten der Prüfdurchläufe zu verbessern. Dies geschieht in Form der Berechnung von Metriken, die den Effekt der Fehlerfortpflanzung bereits im Ursprung vermeiden können. Zudem sollen vorbeugende Maßnahmen gefunden werden, welche auf den Messdaten, den Metrik-Berechnungen sowie der Verwendung historischer Auswertungen (digital archivierte Prüfberichte) basieren. Die Sammlung von Prüfergebnissen über mehrere Prüfdurchläufe eines längeren Zeitraums hinweg hilft bei der Entwicklung von neuen Kennzahlen (Metriken) und schafft damit in einem Unternehmen einen erweiterten Erfahrungsschatz an Konformität, der wiederum die Richtlinien wie auch die Prozesse optimiert.

Ein besonderes Potenzial der letzten Phase des VR-Modells liegt demzufolge in der Möglichkeit zur Entwicklung verbesserter Prüfalgorithmen, welche die Konformität im Voraus abschätzen, berechnen und vorherbestimmen zu können. Aufgrund der Prognosen können dann präventive Maßnahmen (Richtlinienanpassungen oder Formatvorlagen für Artefakte) begründet, angewendet und nach der Implementierung ausgewertet werden, um die kontinuierliche Verbesserung von Entwicklungsprozessen zu ermöglichen. Die präventive Qualitätssicherung ergänzt somit die nachgelagerte und die integrierte Qualitätssicherung und infolgedessen die ganzheitliche Konformitätsprüfung um wesentliche Aspekte.

7 Automatisierte Konformitätsprüfung

Im Rahmen der fünfjährigen Forschungsarbeit am Fraunhofer Institut für offene Kommunikationssysteme FOKUS entstanden nach dem vorgestellten Ansatz anhand der Anforderungen der Volkswagen AG und in Zusammenarbeit mit der Carmeq GmbH prototypische Prüfwerkzeuge für die Automatisierung der regelbasierten Konformitäts-prüfung unter Federführung des Autors.

Zunächst wurde eine Konformitätsprüfung an Modellen durch den *ASD-Regel-Checker* (Veröffentlichungen, Nr. 2, 23, 24) implementiert und nach erfolgreichen Ergebnissen später in eigener Forschungsarbeit die Ausführungsumgebung *Assessment Studio* (Veröffentlichungen, Nr. 11) für die Konformitätsprüfung kollaborativer Artefakte entwickelt. Beide Werkzeuge bauen auf den in dieser Arbeit beschriebenen Grundkonzepten auf und automatisieren die Konformitätsprüfung sowohl für ein einzelnes wie auch für multiple Artefakte**.** Während Teilaspekte innerhalb der Forschungsarbeiten zum Forschungsprojekt MESA (Veröffentlichungen, Nr. 27) bereits zu werkzeugspezifischen Themen unter Anwendung eines allumfassenden Metamodells *(Automotive Software Development Metamodell)* als logisches Artefakt mit der technischen Instanziierung des Konzepts auf der Repository-basierten Plattformtechnologie *medini* [IKV09] unter Anwendung der OCL [OCL20] für Richtlinienformalisierung mehrfach veröffentlicht wurden (siehe Kapitel: Veröffentlichungen), legt diese Arbeit erstmals ein neues Fundament für die generische Automatisierung übergreifender Konformitätsprüfung an kollaborativen Artefakten.

Auf Basis verschiedenster Artefakt-Metamodelle, deren Persistenz auf dem offenen XML-Technologiestandard (Kapitel 2.5.1) basiert, wird in diesem Kapitel ein erweiterter und praktikablerer Weg mit einer industriell weitverbreiteten Technologie beschritten, um auch genügend Offenheit gegenüber anderen Anwendungsfeldern und Prozessen zu gewähren, wo der XML-Technologiestandard ebenfalls Verwendung findet.

7.1 Anforderungen

Eine Ausführungsumgebung für die regelbasierte Konformitätsprüfung kollaborativer Artefakte muss mehrere Anforderungen erfüllen. Dies sind technische wie auch prozessrelevante Aspekte sowie die in dieser Arbeit vorgestellten Kriterien für die Methodik der kollaborativen Konformitätsprüfung. Nachfolgend werden durch Anwendungsfälle Systemanforderungen ermittelt, welche an das Prüfsystem gestellt werden.

Die nachfolgende Abbildung 40 zeigt die Anwendungsfälle des Prüfsystems, modelliert in einem Use-Case-Diagramm (UML).

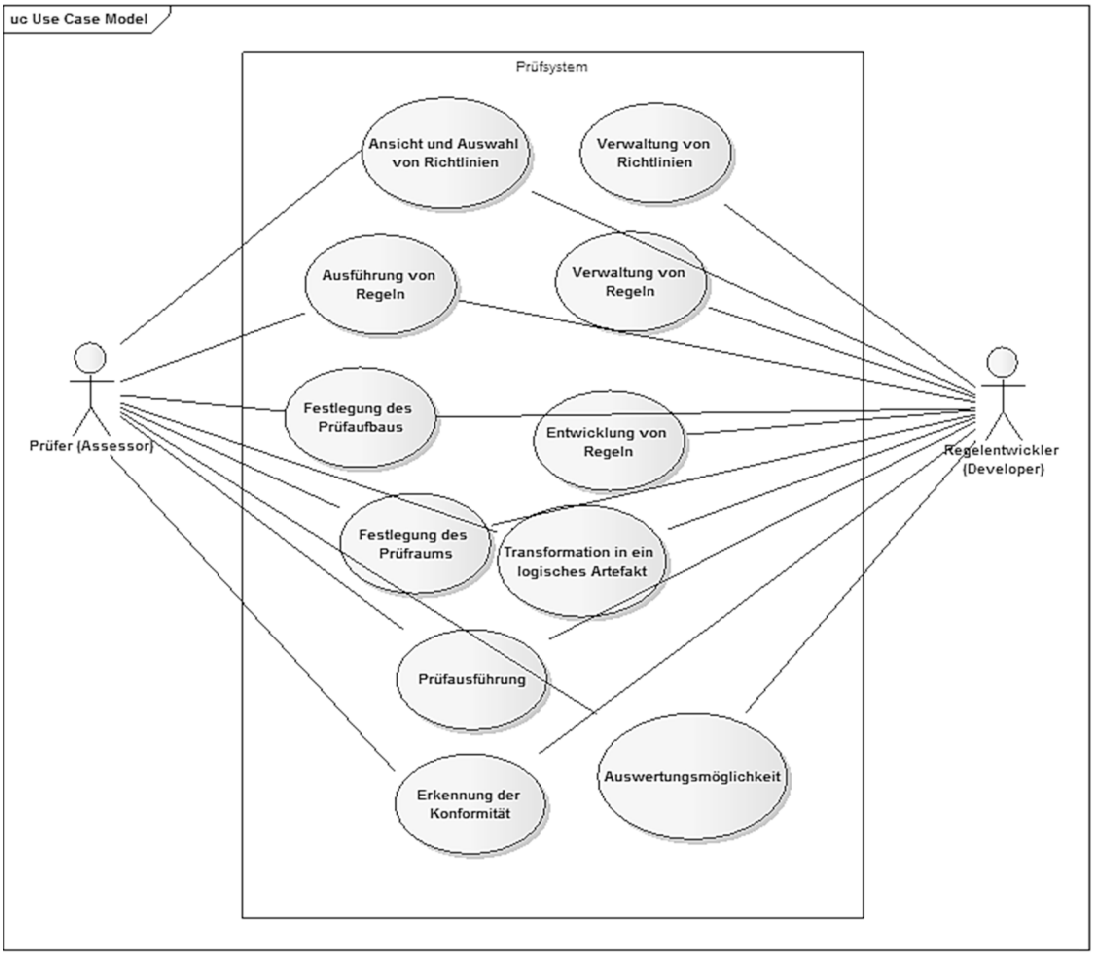


Abbildung 40: Anwendungsfälle für das Prüfsystem

Beschreibung der Rollen:

Das Prüfsystem kann durch verschiedene Anwender bzw. *Akteure* bedient werden. Es werden zwei Akteure identifiziert, für die dementsprechende Anwendungsfälle durch das Prüfsystem konfiguriert werden:

- ***Prüfer*** (Assessor) ist ein Anwender, der die Richtlinienprüfung ausführt. Ein Prüfer kann ein Mitarbeiter eines Unternehmens (z. B. eines Zulieferers) sein, der z. B. die Artefakte entwickelt und regelmäßig kollaborative Artefakte im Entwicklungsprozess gegen vorgegebene Entwicklungsrichtlinien überprüft. Ein Prüfer kann aber auch ein Qualitätsbeauftragter (z. B. eines Herstellers) sein, der die entwickelten kollaborativen Artefakte vom Zulieferer abnimmt. Dazu benötigt der Prüfer ein Prüfsystem mit übersichtlichen Funktionen zu Richtlinienausführung und Reportgenerierung.
- ***Regelentwickler*** (Developer) ist ein Programmierer, der die QRDL-Regelalgorithmen sowie die digitalen Richtlinienkataloge für das Prüfsystem implementiert. Dazu muss der Entwickler die Implementierung der Richtlinien nach dem VR-Modell vornehmen und Programmierkenntnisse besitzen, in welchen der QRDL-Regelalgorithmus implementiert ist. Eine Entwicklungsumgebung mit umfangreichen Funktionen für die Erstellung und Bearbeitung der Regelalgorithmen erleichtert die Arbeit des Entwicklers erheblich.

Beschreibung der Anwendungsfälle:

- *Ansicht, Auswahl und Verwaltung von Richtlinien*: Natürlich sprachliche Richtlinien müssen in einer digitalen Form erfasst werden, sodass ein Anwender sich jederzeit die textuelle Fassung einer Richtlinie ansehen kann. Zudem muss es die Möglichkeit geben, Richtlinien aufgrund ihrer Klassifizierung zu strukturieren und auch hierarchisch zu ordnen. Idealerweise ist die technische Anordnung gleich der der Richtliniendokumentation aufgebaut, um hier eine Transparenz zu schaffen.
- *Ausführung und Verwaltung von Regeln*: Neben der textuellen Fassung einer Richtlinie muss auch der dazugehörige implementierte QRDL-Regelalgorithmus assoziiert werden. Dieser ist in einer beliebigen Programmiersprache verfasst, der durch eine Laufzeitumgebung ausgeführt werden kann.
- *Entwicklung von Regeln*: Eine Entwicklungsumgebung erlaubt die Entwicklung oder die Anpassung eines ausgewählten implementierten QRDL-Regelalgorithmus.
- *Festlegung des Prüfraums*: Es ist sinnvoll, wenn ein Anwender nicht immer alle Regeln eines Regelwerks prüft, sondern Teile nach Bedarf auswählen kann. Hierzu wird ein Mechanismus zur selektiven Auswahl von Richtlinien erforderlich, die zu einer Prüfausführung herangezogen werden sollen.
- *Festlegung des Prüfaufbaus*: Vor einer Prüfung muss die Auswahl kollaborativer Artefakte erfolgen, auf der die regelbasierte Prüfung ausgeführt werden soll.
- *Transformation in ein logisches Artefakt*: Der Prüfaufbau muss durch direkte oder indirekte Transformation das logische Artefakt aus den kollaborativen Artefakt(en) bilden und in den Speicher zur Prüfausführung laden. Ist nur ein einzelnes Artefakt zu prüfen, so ist es wünschenswert, wenn eine direkte Verbindung zum Werkzeug hergestellt wird, worin das Artefakt geladen ist.
- *Prüfausführung*: Eine Prüfausführungsumgebung muss die selektierten Regeln auf dem logischen Artefakt in endlicher Zeit prüfen. Die ermittelten Prüfergebnisse müssen nach einem Prüfdurchlauf nicht-flüchtig gespeichert und auch für die spätere Einsicht erhalten bleiben. Ein Mechanismus zum Laden und Speichern von Prüfungen erleichtert den Umgang für den Anwender.
- *Erkennung der Konformität*: Als Piktogramm stilisierte Elemente der aufgestellten Prüfkriterien erlauben eine schnelle visuelle Auswertung durch den Anwender, welche Richtlinien nach Prüfdurchlauf nicht erfüllt worden sind bzw. an welchen Stellen die Mängel durch den Prüfdurchlauf ermittelt wurden.
- *Auswertungsmöglichkeit*: Ein Bericht (*Report*), der die Auswertung eines Prüfdurchlaufs enthält, gibt dem Anwender die Möglichkeit, den Prüfdurchlauf zu dokumentieren und im Nachgang auszuwerten. Sollten spezielle Prüfnachrichten durch den Regelentwickler im Regelalgorithmus hinterlegt worden sein, erhält ein Prüfer einen Hinweis zur Fehlerbehebung.

Die Auswahl der Rollen sowie die genannten Anwendungsfälle stellen Anforderungen an das zu entwickelnde Prüfsystem − einem Werkzeug zur regelbasierten Konformitätsprüfung kollaborativer Artefakte − dar. In den folgenden Kapiteln wird dieses sukzessive entworfen.

7.2 Konzipierung einer Ausführungsumgebung

Eine Ausführungsumgebung besteht grundlegend aus der Möglichkeit, mehrere Prüfalgorithmen durch einen *Interpreter* bzw. *Compiler* an kollaborativen Artefakten automatisiert auszuführen. Ein Interpreter ist ein spezielles Computerprogramm, das eine in einer beliebigen Programmiersprache beschriebene QRDL-Regel einliest, analysiert und auf einem PC mithilfe einer *Laufzeitumgebung* ausführt.

Laufzeitumgebung *(engl. runtime environment)* **(7.1)**
„Eine *Laufzeitumgebung* ist eine Softwareschicht (Softwarebibliothek), die sich zwischen der Anwendungs- und der Betriebssystem-Schicht befindet. Sie stellt einem ausgeführten Prüfwerkzeug zur regelbasierten Prüfung an kollaborativen Artefakten die benötigte Funktionalität zur Verfügung."

Die Analyse des Quellcodes erfolgt also zur Laufzeit des Programms durch eine Laufzeitumgebung. Die Laufzeitumgebung hat meist selbst einen plattformspezifischen *Compiler* integriert. Ein Compiler ist (wie in der Informatik langläufig bekannt) ein Computerprogramm, welches den in einer Programmiersprache verfassten Algorithmus in ein semantisch äquivalentes Programm einer Zielsprache (plattformspezifischer Code) überführt. Üblicherweise handelt es sich dabei um die Übersetzung eines von einem Programmierer in einer Programmiersprache geschriebenen Quelltextes in eine Maschinensprache (Assemblersprache oder Bytecode). Das Übersetzen der implementierten QRDL-Regel in ein ausführbares Programm durch einen Compiler wird geläufig als *Kompilierung* bezeichnet.

Eine Umgebung zur programmatischen Ausführung von Regelalgorithmen durch einen Interpreter bzw. Compiler nennen wir im weiteren Verlauf *Ausführungsumgebung*.

Ausführungsumgebung *(engl. framework)* **(7.2)**
„Eine Umgebung zur programmatischen Ausführung von QRDL-Regelalgorithmen durch einen Interpreter bzw. Compiler wird als *Ausführungsumgebung* bezeichnet."

7.2.1 Technische Anforderungen an die Ausführungsumgebung

Die Ausführungsumgebung muss zudem die Möglichkeit besitzen, das logische Artefakt aus dem Prüfaufbau in den Hauptspeicher des Rechners als Objektbaum zu laden. Nach [XML09] ist das *Document Object Model* (DOM) eine vom *Word Wide Web Consortium* (W3C) standardisierte Programmierschnittstelle für den Zugriff auf hierarchisch strukturierte Dokumente. Dies sind Dokumente wie XML-Dokumente oder Mapper auf relationale Datenbanken mittels *Open Database Connectivity* (ODBC) Technologie. ODBC ist eine standardisierte Datenbankschnittstelle, welche die SQL als Datenbanksprache verwendet. ODBC bietet also eine Programmierschnittstelle, die es einem Programmierer erlaubt, die Datenstrukturen im internen Hauptspeicher des Rechners relativ unabhängig vom verwendeten *Datenbankmanagementsystem* (DBMS) abzufragen und diese zu verändern, sofern für das gewählte DBMS ein ODBC-Treiber existiert. Beim DOM werden XML-Artefakte gemäß ihrer hierarchischen Anordnung als Objektbaum im Hauptspeicher des Rechners geladen und verwaltet. Weitverbreitete Laufzeitumgebungen, wie das *Java Runtime Environment* (JRE) vom Hersteller Sun Microsystems (Oracle) oder das .NET-Framework (.NET) von Microsoft stellen mehrere Klassen für DOM und die Verarbeitung

von XML-Daten als Laufzeitumgebung zur Verfügung, um einen geladenen Objektbaum im Hauptspeicher zu verwalten.

7.2.2 Auswahl einer Laufzeitumgebung und Programmiersprache

In vorangegangenen Forschungsarbeiten aus MESA (Veröffentlichungen, Nr. 28) wurde mit den im Kapitel 3.8 vorgestellten Werkzeugen aus der Funktionsmodellierung (DOORS, ML/SL/SF und CTE-XL), den Laufzeitumgebungen JRE und .NET, der OCL-Ausdrucks-sprache sowie den Programmiersprachen C# und M-Skript experimentiert. Zum Einsatz in MESA kam pro Werkzeug ein eigens entwickelter, werkzeugintegrierter Tool-Adapter, mit dessen Hilfe ein beliebiges Artefakt (z. B. ML/SL/SF-Modell) aus dem Werkzeugspeicher (z. B. MATLAB-Workspace) über eine COM-Schnittstelle ausgelesen wurde. Anschließend konnte dieses Modell durch den jeweiligen Tool-Adapter, mittels Zugriff (CORBA-Technologie, [CRB30]) über das Netzwerk, als eine Modellinstanz des logischen Artefakts in ein objektorientiertes Repository *medini* erzeugt und gespeichert werden. In diesem Szenario wurde die Transformation des Artefakts in die Struktur des logischen Metamodells durch den Tool-Adapter selbst durchgeführt. Die Transformation wurde dabei statisch in der Programmiersprache des Tool-Adapters ausprogrammiert. Ansätze zur flexiblen Anpassung der Transformationsvorschrift waren weitere Forschungsarbeiten rund um den Standard *Query View Transformation* (QVT) der OMG, nach [QVT10]. Eine auf dem Microsoft .NET-Framework 2.0 basierte Applikation *ASD-Regel-Checker* (Abbildung 41) für die Richtlinienprüfung konnte sodann die netzwerkfähige OCL-Laufzeitumgebung OSLO [OSLO09] nutzen und die in OCL definierten Richtlinien auf einem oder mehreren Instanzmodellen im Repository ausführen.

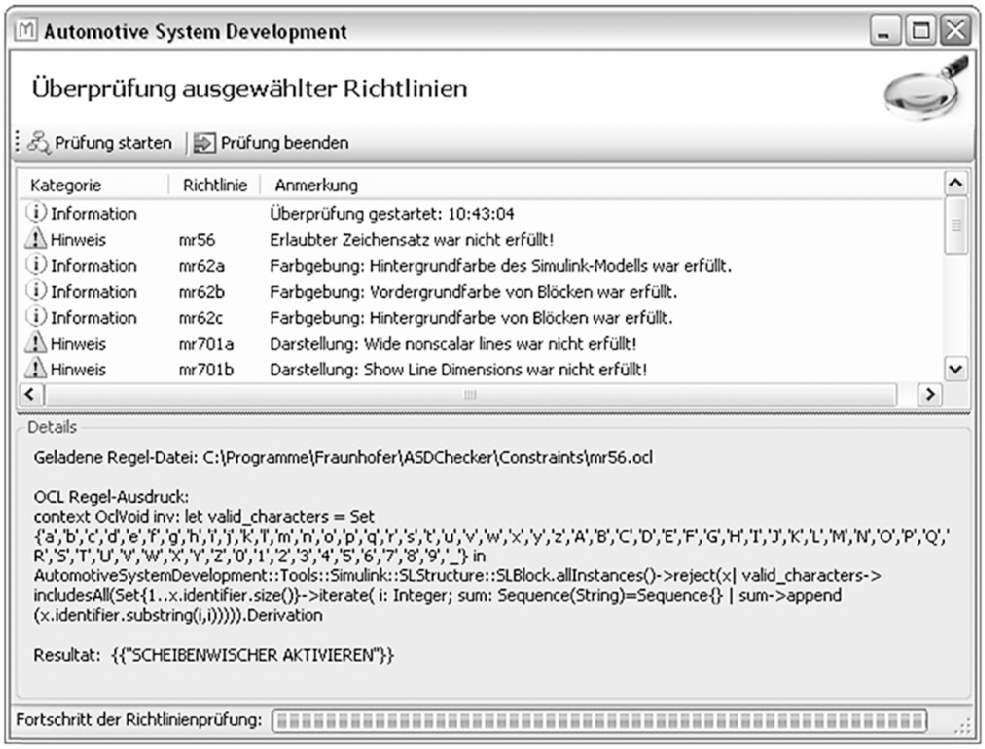


Abbildung 41: ASD-Regel-Checker mit Regelsprache OCL

Die Auswertung erfolgte einfach durch boolesche Ausdrücke (TRUE | FALSE) oder durch die Rückgabe fehlerhafter Elemente, z. B. wie in der Abbildung 41 gezeigt, die nicht

konforme Benennung der Funktion *{{"SCHEIBENWISCHER AKTIVIEREN"}}* im Simulink-Modell, die ein unerlaubtes Leerzeichen enthält. Die Anbindung weiterer Entwicklungswerkzeuge durch Tool-Adapter, wie die bereits eingeführten Werkzeuge DOORS oder CTE-XL, funktionierten analog. In MESA wurden zwar bereits kollaborative Aspekte erkannt und es konnte auch eine erste artefaktübergreifende Konformitätsprüfung werkzeugtechnisch demonstriert werden. Versuche in der Praxis zeigten jedoch, dass die Mächtigkeit der zugrunde liegenden OCL-Technologie sowie die Performance der Regelprüfung einige Einschränkungen aufwiesen.

Auch wurde der datenbankorientierte Ansatz beleuchtet. Es ist prinzipiell möglich, die logischen Artefakte in ein relationales Datenbankmanagementsystem (DBMS) zu instanziieren und dort die Abfragesprache SQL, wie in Kapitel 2.5.3 vorgestellt, für die Implementierung des QRDL-Algorithmus zu verwenden. Dies ist ein gangbarer Weg, hat jedoch den entscheidenden Nachteil, das kollaborative Artefakte auf ein hierarchisch strukturiertes, objektorientiertes Datenmodell aufbauen, welches sich durch einen weiteren Transformationsschritt nur umständlich in das flache relationale Datenmodell überführen lässt. Es existieren auch objektorientierte DBMS – diese wurden jedoch nicht weiter untersucht, da der datenbankorientierte Ansatz generell noch einen weiteren eklatanten Nachteil bezüglich der Skalierbarkeit in sich birgt. Da es sich bei DBMS-Lösungen um integrierte Systeme handelt, an die dann mittels Tool-Adapter-Konzept die Transformation kollaborativer Artefakte in logische Artefakte erfolgt, müsste für jedes verschiedenartige Artefakt ein neuer Tool-Adapter mit spezifischer Transformationsvorschrift entwickelt werden. Somit ist für diese technische Lösung immer mindestens ein Tool-Adapter zu entwickeln und zusätzlich die Lizenz eines DBMS auf einem Client notwendig.

Diese Erfahrungen und Vorüberlegungen haben einen neuen dateibasierten Lösungsweg erforderlich gemacht. Im weiteren Verlauf dieser Arbeit wird daher eine Weiterentwicklung des technischen Konzepts vorgestellt, welches auf XML-Technologie und LINQ basiert. Das neue Konzept vermeidet Einschränkungen vorheriger Ansätze, skaliert besser (vgl. Kapitel 7.7) und ist praxistauglicher.

Für den Aufbau der Ausführungsumgebung wird als Basis die Laufzeitumgebung .NET-Framework (.NET) in der Version 3.5 von Microsoft gewählt. Die Laufzeitumgebung wird als *Common Language Infrastructure* (CLI) bezeichnet, ein ISO/IEC/ECMA-Standard. Kerninhalt dieses Standards ist nach [RICH06] die Spezifikation eines Systems, das eine sprach- und plattformunabhängige Anwendungsentwicklung und -ausführung ermöglicht. Die Common Intermediate Language (CIL) ist eine Zwischensprache, welche von der *Common Language Runtime* (CLR) in Maschinencode übersetzt und zur Ausführung an das Betriebssystem übergeben wird. Die .NET-Sprachcompiler übersetzen den Quellcode der Assemblies (Distributionseinheit) nicht direkt in nativen Maschinencode, sondern in die CIL als Zwischensprache. Dieses Konzept erlaubt zum einen eine angestrebte Plattform-unabhängigkeit der Programmiersprachen und zum anderen den Einsatz verschiedener Programmiersprachen für eine Ausführungsumgebung. CIL wird in der Literatur auch häufig als MSIL (Microsoft Intermediate Language) oder einfach als IL (Intermediate Language) bezeichnet. Neben der Umsetzung dieses Standards durch Microsoft als .NET-Framework existieren weitere Implementierungen z. B. für Unix/Linux-Derivate und Mac OS-Systeme. Die bekannteste ist das Mono-Framework [MONO09] als Portierung für Linux/Unix.

Das .NET-Framework 3.5 erfüllt insbesondere die in Kapitel 7.1 genannten Anforderungen an das zu entwickelnde Prüfsystem und bietet Interpreter und Compiler für die Ausführungsumgebung auf Basis verschiedener Programmiersprachen. Der neu gewählte

Ansatz, die XML-Technologie einzusetzen, erfordert zudem kein gesondertes Repository für die Speicherung eines logischen Artefakts, sondern funktioniert rein auf Dateibasis sowie verteilt durch Ordnerfreigaben. Das logische Artefakt wird dabei durch die Ausführungsumgebung geladen und direkt im Hauptspeicher gehalten. Anstelle OCL oder M-Skript als Programmiersprachen für den QRDL-Regelalgorithmus zu verwenden bietet das .NET-Framework die Möglichkeit zum Ausführen von XPath-Ausdrücken mit *XmlDocument* oder neuerdings auch eine integrierte Abfragesprache LINQ an, die eingebettet in der Programmiersprache C# auf Objektbäumen im Hauptspeicher arbeitet, wie im Kapitel 2.5.4 vorgestellt wurde. Dabei ist C# eine objektorientierte, mit dem .NET-Framework eingeführte, .NET-Programmiersprache. Sie bietet im Gegensatz zu OCL oder M-Skript eine weitreichende Sprachmächtigkeit, die allen seitens der QRDL geforderten Eigenschaften genügt. Die Syntax sowie viele Sprachkonstrukte und -konzepte von C# ähneln denen von C++ oder Java. Im Gegensatz zu interpretierten XPath-Ausdrücken sind LINQ-Ausdrücke kompiliert und dadurch schneller ausführbar.

7.2.3 Analyse der benötigten Framework-Komponenten

Nach Auswahl der Laufzeitumgebung muss das .NET-Framework hinsichtlich der benötigten Funktionalitäten mithilfe von [LOUI09] untersucht werden. Die .NET-Klassen *XmlNode* und *XmlDocument* aus dem Namespace *System.Xml* ermöglichen den Zugriff auf und die Manipulation von XML-basierten Artefakten, während die Klasse *HtmlDocument* aus dem Namespace *System.Windows.Forms* den Zugriff auf HTML-Dokumente für die Darstellung und die Verwaltung natürlich sprachlicher Richtlinien (in HTML-Format) ermöglicht. Für die DOM-Repräsentation kann das gesamte logische Artefakt samt Struktur in den Hauptspeicher geladen werden. Für einen lesenden Zugriff eignet sich die *XmlReader*-Klasse. Der *System.Linq*-Namespace stellt die Klassen und Schnittstellen bereit, welche die Algorithmen mit sprachintegrierter Abfrage (Language-Integrated Query, LINQ) unterstützen. Die wesentlichen LINQ-Klassen zur Traversierung der Objektbäume im Speicher sowie zur Formulierung der Regelalgorithmen finden sich in den Namensräumen *System.Linq.**, *System.Data.Linq* sowie *System.Xml.Linq*.

7.2.4 Auswahl der Technologien

Aus den gesammelten Anforderungen an eine Ausführungsumgebung sowie der Analyse der Laufzeitumgebung und vorangegangener prototypischer Entwicklungen wird nun eine flexible Ausführungsumgebung mit Anbindungsmöglichkeiten für Datei, für Datenbank und für Werkzeuge entwickelt, welche die Möglichkeit besitzt, **1..*n*** kollaborative Artefakte aus dem Prüfaufbau in den Hauptspeicher des Rechners als logisches Artefakt zu instanziieren und schließlich zu traversieren. Als kollaborative Artefakte wird technisch folgende Klassifizierung (Klassen) festgelegt:

i. *Kollaboratives Artefakt in einem Entwicklungswerkzeug*

ii. *Kollaboratives Artefakt als proprietäres Dateiformat*

iii. *Kollaboratives Artefakt als XML-Datei*

iv. *Kollaboratives Artefakt als Datenmenge eines DBMS*

Das logische Artefakt ist durch direkte oder indirekte Transformation in Form eines hierarchisch strukturierten, objektorientierten Datenmodells im Hauptspeicher des Rechners vorhanden und muss dafür hinsichtlich der technischen Klassifizierung unterschiedlich

gebildet werden. Die Bildung und der Zugriff erfolgten durch die Vereinheitlichung der technischen Unterschiede auf XML-Basis und durch unterschiedliche Technologien nach [PIAL07] der Laufzeitumgebung des .NET-Frameworks:

- *Component Object Model (COM)*: Das Component Object Model ist eine Plattformtechnologie, um unter dem Betriebssystem Windows Interprozesskommunikation und dynamische Objekterzeugung zu ermöglichen. Jede COM-Komponente bietet eine Schnittstelle (Interface) an, welche dazu verwendet werden kann, die angebotenen Funktionen der COM-Komponente einzusetzen.
- *LINQ-to-Dataset*: Technologie zur Abfragemöglichkeit von Objekten, die in einem Datenbankobjekt (*DataSet*-Objekt) gespeichert sind und durch ein integriertes DBMS zur Verfügung stehen (wie z. B. Microsoft SQL Server / MS SQL Server).
- *LINQ-to-SQL (DLINQ)*: Technologie zur Übersetzung nativer LINQ-Abfragen in die SQL. Zusätzlich wird eine Kommunikation zu einer relationalen Datenbank via ODBC hergestellt. Von der Datenbank nach Abfrage gelieferte Ergebnisse werden dann entsprechend wieder in das interne Objektmodell übersetzt.
- *LINQ-to-XML (XLINQ)*: Technologie für den Zugriff auf XML-Dokumente (DOM-Schnittstelle und Manipulation) sowie eine XML-Programmierschnittstelle für XML-Objekte im Arbeitsspeicher.
- *LINQ-to-Objects*: Technologie für Abfragemöglichkeiten eines speicherinternen Datenmodells (.NET-Objektmengen), die in Listen gespeichert sind, z. B. in den Datentypen *List*, *Array* oder *Dictionary*. Voraussetzung ist, dass ein Datentyp die Schnittstelle (Interface) *IEnumerable* bzw. *IEnumerable<T>* implementiert.

Es folgt nun die Untersuchung, welche der vorgestellten Technologien für welche der technischen Klassen ***i***)-***iv***) einsetzbar ist:

Im Falle ***i***) ist das kollaborative Artefakt in einem Entwicklungswerkzeug geladen. Um dies regelbasiert zu prüfen, muss das Entwicklungswerkzeug eine Programmierschnittstelle (API) zur Verfügung stellen, die es erlaubt, auf das werkzeuginterne Datenmodell im Hauptspeicher zuzugreifen. So wäre der Zugang zum logischen Artefakt gleich gegeben und es kann eine direkte oder indirekte Transformation eines kollaborativen Artefakts in das logische Artefakt erfolgen. Bei einer Vielzahl an Entwicklungswerkzeugen geschieht der API-Zugriff mittels COM-Technologie. Ist dies z. B. durch eine eingeschränkte API nicht möglich, sollte das Artefakt im Hintergrund vor Prüfdurchlauf automatisch in eine XML-Datei exportiert werden. Falls nötig, müssen Komponenten von Drittherstellern hinzugezogen werden, die eine proprietäre Lösung für die XML-Konvertierung anbieten. Auf Grundlage der XML-Datei kann dann LINQ-to-XML eingesetzt werden, um das logische Artefakt im Speicher zu erzeugen.

Im Falle ***ii***) liegt das kollaborative Artefakt in einem proprietären Dateiformat vor. Um dies regelbasiert zu prüfen, muss das Artefakt im Hintergrund vor Prüfdurchlauf automatisch durch direkte oder indirekte Transformation in eine XML-Datei konvertiert werden. Viele Werkzeuge bieten heutzutage die Möglichkeit zum Export in das XML-Format an. Auf Grundlage der XML-Datei kann dann LINQ-to-XML eingesetzt werden, um das logische Artefakt im Speicher zu erzeugen.

Im Falle ***iii***) ist das kollaborative Artefakt bereits in einem XML-Dateiformat. Um dies regelbasiert zu prüfen, muss das Artefakt im Hintergrund vor Prüfdurchlauf automatisch als logisches Artefakt in den Hauptspeicher geladen werden. Die direkte Transformation

entfällt. Eine Wrapper-Technologie muss lediglich ein logisches Artefakt im Hauptspeicher allozieren. Auf Grundlage der XML-Datei kann dann LINQ-to-XML eingesetzt werden, um das logische Artefakt im Speicher zu erzeugen. Nur bei indirekter Transformation ist zusätzlich ein Transformationsschritt nötig.

Im Falle ***iv***) ist das kollaborative Artefakt in einer Datenbank abgelegt, verwaltet durch ein relationales DBMS. Hierfür wird eine Wrapper-Technologie erforderlich, die es ermöglicht, auf das interne Datenmodell der Datenbank zuzugreifen. Die LINQ-to-SQL-Technologie (DLINQ) der Laufzeitumgebung übersetzt die native LINQ-Abfrage in SQL und sendet diese an eine relationale Datenbank. Von der Datenbank gelieferte Ergebnisse werden dann entsprechend wieder in ein Objektmodell übersetzt. Da der SQL-Provider nur mit MS SQL-Servern funktioniert, ist es möglich, mit der verwandten Technologie LINQ-to-Dataset mittels DBMS-Provider die Daten aus Data-Sets zu beziehen. Die Datasets können mittels ADO .NET-Technologie, beschrieben in [LOUI09], auch aus anderen Datenbanken gefüllt werden, wodurch es nach [PIAL07] indirekt möglich ist, mittels LINQ-to-SQL auf beliebige relationale Datenbanksysteme zuzugreifen.

Wurden die kollaborativen Artefakte schließlich erfolgreich durch eine der genannten Technologien geladen, muss das logische Artefakt im Hauptspeicher durch eine Technologie und Abfragesprache traversiert und ausgewertet werden können. LINQ-to-Objects ermöglicht dies. Es können referenzierte Datenbestände durch Abfragen verwaltet werden, die in Quellcode der Programmiersprache C# eingebettet werden. Diese Abfragen werden dabei in einer speziellen Syntax formuliert, die der SQL ähnelt (vgl. Kapitel 2.5.3). Während SQL aber nur im Zusammenhang mit relationalen Datenbanken verwendet werden kann, können LINQ-Abfragen auf unterschiedliche Datenbestände durch die vorher erwähnten Technologien angewandt werden.

Somit bietet C# mit LINQ ein Instrument an, um beliebige Daten auf eine Art und Weise zu verarbeiten, wie sie vielen Entwicklern mit SQL vertraut ist. Für das logische Artefakt wird daher LINQ-to-Objects als geeignete Technologie identifiziert. Im nachfolgenden Kapitel 7.3 wird die Sprachmächtigkeit genauer untersucht.

7.2.5 Ausführungsumgebung

Schließlich lässt sich aus den Anforderungen an eine Ausführungsumgebung sowie der Analyse vorangegangener Entwicklungen und verfügbarer Technologien eine hinreichend genügende Ausführungsumgebung mit Anbindungsmöglichkeiten für Datei, für Datenbank und für Werkzeuge konzipieren. Diese muss insbesondere die Möglichkeit besitzen, ***1..n*** kollaborative Artefakte aus dem Prüfaufbau in den Hauptspeicher des Rechners als logisches Artefakt zu instanziieren und schließlich zu traversieren. Die nachfolgende Abbildung 42 skizziert den prinzipiellen Aufbau einer solchen Ausführungsumgebung.

Die Laufzeitumgebung des .NET-Frameworks stellt ausgewählte Technologien für den Zugriff auf unterschiedliche Datenquellen bereit, um mehrere kollaborative Artefakte in den Hauptspeicher eines Rechners zu laden. Zusätzlich verwendet auch das Prüfsystem die LINQ-Technologie samt Interpreter und Compiler der Laufzeitumgebung, um sowohl logische Artefakte im Hauptspeicher eines Rechners zu traversieren und außerdem Regelalgorithmen auf diesen auszuführen.

Das Prüfsystem (beschrieben in Kapitel 7.5) gibt nach Prüfdurchlauf eine Auswertung in dokumentierter Form als Bericht. Zusätzlich lassen sich in Werkzeugen geladene Artefakte durch die COM-Technologie an die Ausführungsumgebung technologisch anbinden. Die

Zahnräder in der Abbildung 42 symbolisieren zusätzliche Transformationsalgorithmen, welche durch Tool-Adapter automatisiert ausgeführt werden können. Die Tool-Adapter sind ein Thema des nachfolgenden Kapitels 7.3.

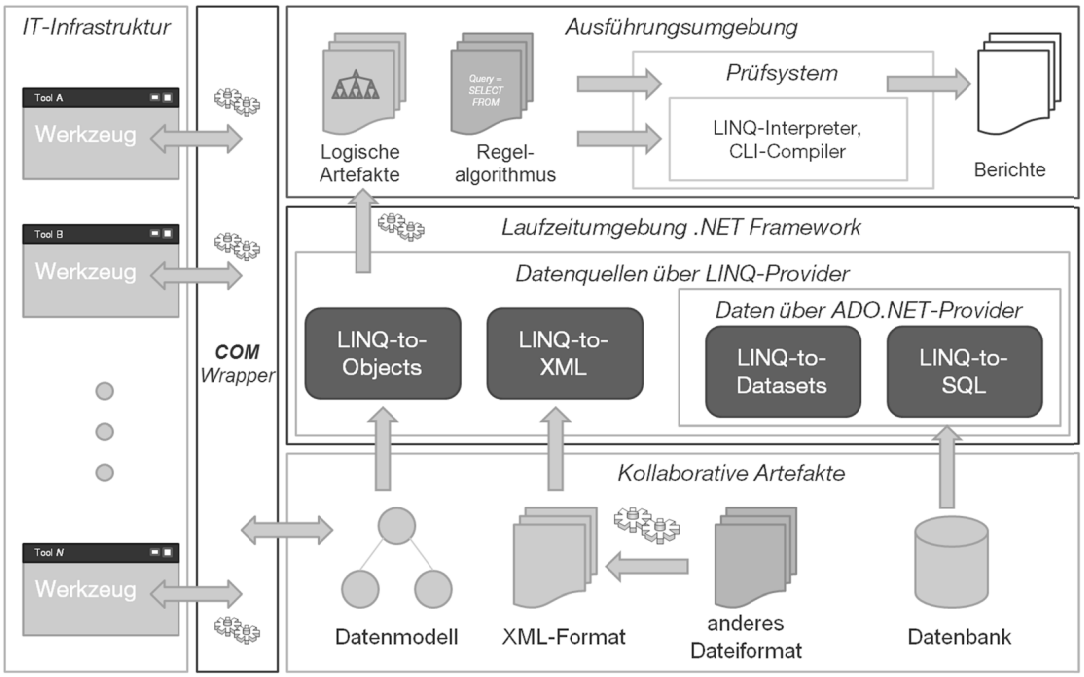


Abbildung 42: Ausführungsumgebung und Prüfsystem

7.3 Konstruktion logischer Artefakte

Wie bereits zuvor beschrieben, sind zwei Wege möglich, um ein logisches Artefakt für die regelbasierte Konformitätsprüfung zu erhalten. Durch eine direkte oder eine indirekte Transformationsvorschrift muss es nun technisch möglich sein, kollaborative Artefakte durch die Ausführungsumgebung in den Hauptspeicher eines Rechners automatisiert zu laden. Hierfür wird das technische Konzept eines Wrapper-Mechanismus verwendet, im Folgenden *Wrapper* genannt. Zunächst wird kurz der Wrapper-Mechanismus konzipiert, bevor technische Beispiele für die direkte und die indirekte Transformation vorgestellt werden.

7.3.1 Wrapper-Mechanismus

Damit unterschiedliche Entwicklungswerkzeuge (Programme) verschiedener Hersteller für eine regelbasierte Konformitätsprüfung ein gleich strukturiertes Datenmodell als logisches Artefakt in den Hauptspeicher eines Rechners für die Prüfausführung laden können, muss es ein Prinzip einer einheitlichen Werkzeug-Kapselung geben. Dieses Prinzip wird technisch *Wrapper*-Mechanismus genannt und existiert pro Werkzeug (Programmschnittstelle) oder pro Technologie (XML-Wrapper).

Wrapper ***(engl. wrapper)*** **(7.3)**
„*Wrapper* beschreiben eine Technik, die dazu gedacht ist, Werkzeuge flexibel und auch nachträglich in eine Ausführungsumgebung zu integrieren.“

Ein Wrapper (vgl. Abbildung 42) legt bildlich eine Schale um das jeweils zu integrierende Werkzeug oder proprietäre Artefakt. Der Sinn dieses Konzepts besteht darin, einerseits die vorhandene Schnittstelle des eingekapselten Werkzeugs für den Zugriff auf das interne Datenmodell zu nutzen und andererseits der Ausführungsumgebung, die mit einem eingekapselten Werkzeug integriert werden soll, eine saubere und integrierbare Schnittstelle (z. B. für XML-Datenexport, der Verwendung einer werkzeuginternen Programmiersprache als Regelsprache oder zum Referenzieren in das interne Datenmodell) anzubieten. Die jeweilige Schale wird direkt mit dem eingekapselten Werkzeug abgestimmt und nimmt dann, als eigentliche Funktionalität, eine Umsetzung einzelner Schnittstellen aufeinander vor.

Wie weitreichend die Integration der Werkzeuge mit diesem Verfahren ist, hängt in erster Linie von den zu integrierenden Werkzeugen ab (vgl. Kapitel 3.8). Diese bestimmen, wie viel von ihrer Funktionalität nach außen gegeben wird, bzw. wie viel Zugriff sie auf ihr internes Datenmodell gewähren. Der Wrapper-Mechanismus ist zunächst von der konkreten Implementierung unabhängig, da nur eine Technik auf eine andere abgebildet wird. Es ist daher möglich, verschiedene Wrapper-Implementierungen in der Ausführungsumgebung zu integrieren.

Um die prinzipielle Offenheit und leichte Erweiterbarkeit bei der Datenstrukturierung des logischen Artefakts zu erreichen, kann entweder direkt das Datenmodell des Werkzeugs oder des proprietären Artefakts für das logische Artefakt verwendet werden. Dies wäre die *direkte* Transformation. Andernfalls wird die *indirekte* Transformation verfolgt, bei der mittels Metamodell eine spezifische Repräsentation der Daten und Datentypen, die zwischen den einzelnen Werkzeugen auszutauschen sind, modelliert wird. Auch eine Transformation in ein eigenes Datentypformat ist somit möglich. In diesem Fall führt ein Wrapper die Konvertierung der spezifischen Werkzeug-Datenformate in das universelle Datenformat des logischen Artefakts vor.

Demnach sieht der technische Ansatz zur Realisierung *werkzeugübergreifender Prüfung* vor, für jedes kollaborative Artefakt einen Wrapper-Mechanismus zu verwenden, um das jeweilige Datenmodell in eine werkzeugunabhängige Repräsentation (logisches Artefakt) zu überführen. Wie bei früheren Arbeiten in MESA festgestellt, führt dies in der Praxis zu einem relativ hohen Aufwand – betrachtet man die Vielzahl an verfügbaren Werkzeugen am Markt. Daher wird in diesem neuen technischen Lösungsansatz die weitverbreitete XML-Technologie als generische Auszeichnungssprache und Interimsformat für das logische Artefakt verwendet, da mit dieser Technologie nur *ein* singulärer XML-Wrapper erforderlich ist, der eine Vielzahl an bereits verfügbaren Dateiformaten in der Praxis abdeckt. Zudem kann die direkte und die indirekte Transformation durch XSLT (nach [WYKE02]) oder LINQ (nach [MARG08]) durch den Wrapper selbst realisiert werden, sodass das logische Artefakt aus einer XML-Datei als Datenmodell im Speicher aufgebaut werden kann.

Während dieser Arbeit sind Wrapper für die Werkzeuge DOORS, MATLAB, ASCET-MD sowie ein generischer XML-Wrapper in verschiedenen Forschungsprojekten implementiert worden. Sie wurden teils in Skriptsprachen DXL und M-Script wie auch in der Programmiersprache C# als externe Komponenten entwickelt, welche auf dem derzeit aktuellen Microsoft .NET-Framework in der Version 3.5 implementiert wurden und auch unter Version 4.0 lauffähig sein werden. Die nachfolgenden Kapitel stellen die Technologie-Konzepte für DOORS und MATLAB sowie zu XML vor.

7.3.2 Technische Realisierung

Im Folgenden wird der konzeptionelle Aufbau eines Wrappers für die ausgewählten Werkzeuge DOORS und ML/SL/SF sowie für beliebige XML-basierte Dokumente erklärt.

Das Werkzeug für das Anforderungsmanagement DOORS und die Werkzeugfamilie ML/SL/SF (vgl. Kapitel 3.8) verwenden zum Austausch von Informationen und Daten durchgehend die Common Object Model (COM-Schnittstelle, *Interface*). Das Werkzeug zur Testspezifikation CTE/XL weist keine COM-Schnittstelle auf. Es speichert seine Artefakte im XML-Format und wird daher später betrachtet.

COM ist eine gemeinsam von den beiden Herstellern Microsoft und DEC entwickelte, objektorientierte offene Systemarchitektur, über die Client-Server-Anwendungen von verschiedenen Plattformen miteinander kommunizieren können. Die Schnittstelle, das COM-Interface, dient der Kommunikation vom COM-Client zum COM-Server. Eine COM-Komponente kann dazu über allgemein definierte und vorgegebene Schnittstellen (*IUnknown, IDispatch*) sowie über spezielle Schnittstellen angesprochen werden. Für beide Werkzeuge wurde festgestellt, dass ein Zugriff auf das interne Datenmodell über keine Schnittstelle möglich ist. Jedoch werden ein Datenimport und ein Datenexport ermöglicht sowie auch die automatisierte Steuerung des Programmablaufs, z. B. das Werkzeug durch Fernzugriff starten, das Artefakt laden, das Artefakt schließen, das Werkzeug wieder schließen.

7.3.2.1 Beispiel eines DOORS-Wrappers

Die Schnittstelle zur Anwendungsprogrammierung (Programmierschnittstelle, API) von DOORS ist eine spezielle COM-Schnittstelle, auf die von einem Betriebssystem oder von einer anderen Softwareapplikation (Werkzeug oder Adapter) zugegriffen werden kann. Je nach Mächtigkeit erlaubt sie es, Funktionen aus anderen Programmen aufzurufen oder über diese zu kommunizieren. Über definierte Schnittstellen wird so ein Informations- und Datenaustausch erst möglich. Ein DOORS-Wrapper muss eine Applikation sein, die den Zugriff auf die COM-API zulässt. Die gewählte Laufzeitumgebung .NET-Framework erlaubt diesen Zugriff. Für die Steuerung von DOORS zum Export von Artefakten wurde ein Wrapper entwickelt, der die COM-basierten Funktionen *runFile(string)* und *runStr(string)* verwendet. Die Parameter sind im ersten Fall der Name der DXL-Datei (DXL: DOORS Programmiersprache) und im letzteren Fall ein auszuführendes Programm in DXL-Programmiersyntax. Um das Datenmodell aus DOORS auszulesen, ist zwingend eine DOORS-interne Wrapper-Komponente erforderlich, welche das geladene Datenmodell in das XML-Format persistent überführt. Für die Transformation des kollaborativen Artefakts in ein logisches Artefakt, zunächst in XML, wurde während der Arbeit ein DXL-Programm entwickelt, welches eine Teilkomponente des DOORS-Wrappers realisiert. Ausgeführt wird dieses über eine externe Teilkomponente des DOORS-Wrappers in der Programmiersprache C#, welche über das genannte COM-Interface die beiden Methodenaufrufe zur Ausführung der internen Wrapper-Komponente steuert.

7.3.2.2 Beispiel eines MATLAB-Wrappers

Das Werkzeug ML/SL/SF unterstützt ebenfalls die COM-Schnittstelle. In Erweiterung sogar die Distributed-COM-Schnittstelle (DCOM) für den entfernten Zugriff über ein Windows-basiertes Netzwerk. Zudem ist die API dieser Schnittstelle in der Lage, im Gegensatz zur Java-Laufzeitumgebungsintegration in ML/SL/SF, sowohl den Import von Informationen aus Dritt- als auch den Informationsexport an Drittanwendungen zu realisieren. Die Tabelle 21 listet wesentliche COM-Funktionen zur entfernten Interaktion, Steuerung und

Datenübertragung für die Transformation in ein logisches Artefakt auf, die sich über die COM-Programmierschnittstelle ansprechen lassen.

Tabelle 21: COM Programmierschnittstelle ML/SL/SF

MATLAB COM Server Funktionen	
Enableservice	*Aktivierung des COM-Automation-Servers*
Execute	*Ausführung eines MATLAB-Kommandos*
Feval	*Ausführung einer MATLAB-Funktion*
GetCharArray	*Auslesen eines Zeichenfelds*
GetFullMatrix	*Auslesen eines Datenfelds (Matrix)*
GetVariable	*Auslesen einer Variablen des internen Speichers*
GetWorkspaceData	*Auslesen von Daten des internen Speichers*
MaximizeCommandWindow	*Anzeige des Server-Programmfenster auf dem Desktop*
MinimizeCommandWindow	*Minimieren des Server-Programmfensters*
PutCharArray	*Setzen eines Zeichenfelds*
PutFullMatrix	*Setzen eines Datenfelds (Matrix)*
PutWorkspaceData	*Setzen einer Variablen des internen Speichers*
Quit	*Beenden des COM-Automation-Servers*

In Analogie zum DOORS-Wrapper ist auch hier keine Möglichkeit gegeben, das Datenmodell direkt auszulesen. Über einen Umweg wurde jedoch eine Lösung erarbeitet. MATLAB selbst hat einen integrierten Speicher (Workspace) und diesen partitioniert in mehrere Bereiche.

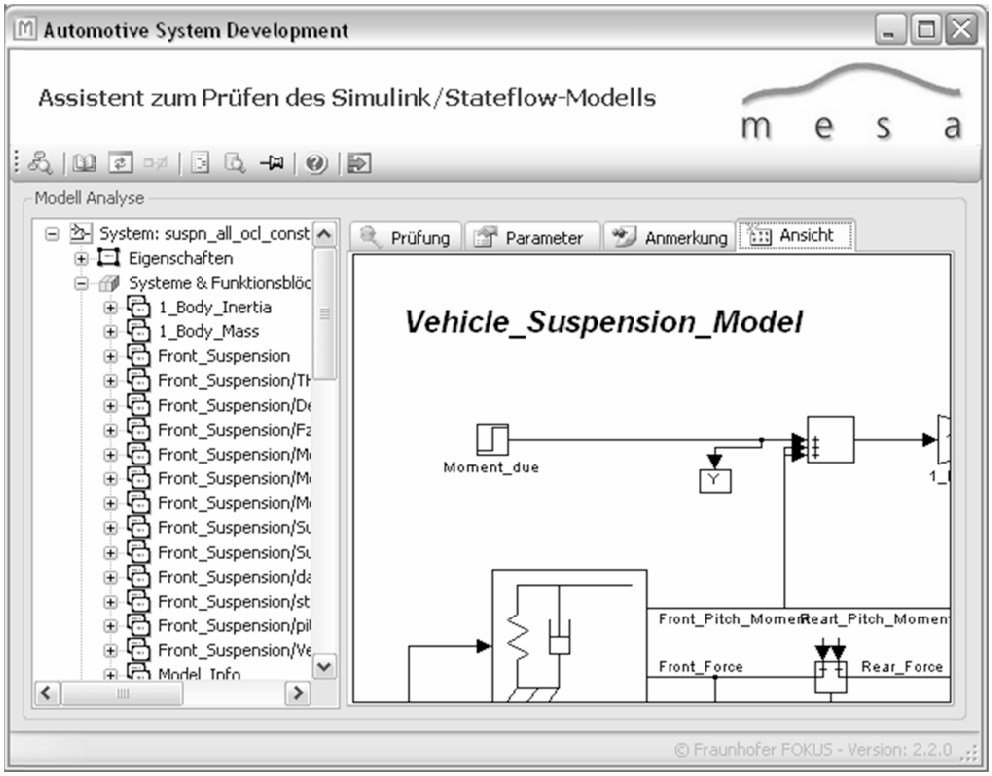


Abbildung 43: Simulink-Modell als logisches Artefakt

Der interne Workspace dient zur Ablage der aus einem Modell ausgelesenen Daten. Er wird mittels der COM-Schnittstelle systematisch ausgelesen und im Repository als logisches Artefakt geladen. Dies hierarchisch strukturierte logische Artefakt ist in der Abbildung 43 im Navigationsbaum links zu sehen. Identisch muss hier auch eine zweiteilige interne und externe Wrapper-Komponente technisch realisiert werden, welche ein internes Programm durch die integrierte ML/SL/SF-Programmiersprache M-Skript aufruft, um die Datentransformation über den MATLAB-internen Workspace, dem werkzeugspezifischen Programmspeicher, durchzuführen.

Ein zweiter technischer Lösungsansatz wurde mittels XML-Export versucht. Für diese interne Wrapper-Komponente wurde ein am Markt erhältliches Programm *SimEx* [SIME09] verwendet. Von der externen Wrapper-Komponente werden *SimEx*-Schnittstellen aufgerufen und führen den XML-Export eines geladenen Simulink-Modells unter Angabe eines Speicherpfades durch. Ausgeführt wird diese Wrapper-Komponente über eine externe Teilkomponente des MATLAB-Wrappers in der Programmiersprache C#, welche über das genannte COM-Interface die drei Methodenaufrufe *Execute(args)*, *GetVariable(args)* und *GetWorkspaceData(args)* zur Ausführung der internen Wrapper-Komponente steuert.

7.3.2.3 Beispiel eines generischen XML-Wrappers

Die bisher vorgestellten Lösungsansätze haben gezeigt, dass XML als Interimsformat für ein logisches Artefakt verwendet werden kann. Werkzeuge wie CTE/XL (vgl. Kapitel 3.8.4) und viele weitere speichern sogar direkt ihre Artefakte in das XML-Format. Technische Standardisierungen aus dem Automotive-Umfeld (vgl. Kapitel 3.2), wie Requirements Interchange Format (RIF), AUTOSAR-XML-Beschreibungsdateien für Steuergerätesoftware sowie ASAM-ODX (Open Diagnostic Data Exchange, ODX-Standard) für Diagnosenachrichten bestätigen den Trend. Auch im Office-Umfeld sind Standardisierungen, wie das OASIS Open Document Format (ISO/IEC 26300) oder Office Open XML (ISO/IEC 29500) allgemeine Beispiele, wie [ECKE09] einführt.

Nehmen wir ein XML-Beispiel, das Informationen über Funktionsmodelle in einem Datenmodell hält, welches in eine XML-Datei mit dem Namen ***APR-Models.xml*** transformiert wurde.

Beispiel: ***APR-Models.xml***

```
<?xml version="1.0" encoding="utf-8" ?>

<Models>

  <Model>
    <Author>Max Mustermann</Author>
    <Title>
      Fensterhebersteuergerät
    </Title>
    <Acronym>SGFH</Acronym>
    <Date>12/01/2009</Date>
  </Model>

  <Model>
    <Author>Otto Beispielhaft</Author>
    <Title>
      Batteriemanagement-Steuergerät
    </Title>
        <Acronym>SGB</Acronym>
    <Date>12/02/2009</Date>
  </Model>
```

```
  <Model>
    <Author>Max Mustermann</Author>
    <Title>
      Steuergerät für Lichtsteuerung
    </Title>
        <Acronym>SGL</Acronym>
    <Date>12/03/2009</Date>
  </Model>
</Models>
```

Um weitreichende, werkzeugübergreifende Prüfungen durchführen zu können, erfordert dies einen XML-Wrapper, der die Datenmodelle unterschiedlicher Werkzeuge und Standards durch eine XML-Datei einlesen und im Hauptspeicher eines Rechners als DOM verarbeiten kann.

Ein XML-Wrapper kann durch die Verwendung von LINQ-Sprachsyntax und .NET-Framework-Komponenten diese Datei in den Hauptspeicher als DOM laden und die Instanz des Objektmodells durch einen LINQ-Abfrageausdruck traversieren.

Einen grundlegenden XML-Wrapper zeigt der folgende Code-Ausschnitt:

Beispiel: ***Generischer XML-Wrapper***

```
using System;
using System.Linq;
using System.Xml.Linq;
using System.Collections.Generic;

XDocument xmlDoc = XDocument.Load("APR-Models.xml");

var objects = from Model in xmlDoc.Descendants("Model")

                select new
                {
                  Author    = Model.Element("Author").Value,
                  Title     = Model.Element("Title").Value,
                  Acronym   = Model.Element("Acronym").Value,
                  Date      = Model.Element("Date").Value
                };
```

In dem gezeigten Beispiel wird die Datei mit ***APR-Models.xml*** in den Hauptspeicher als Datenmodell `xmlDoc` *direkt* transformiert und anschließend durch eine LINQ-Abfrage das logische Artefakt `objects` aufgebaut.

Das Beispiel setzt voraus, dass das zugrunde liegende Datenmodell des kollaborativen Artefakts bekannt ist. Es ist auch möglich, hier *reflexiv* zu programmieren und neue Objekte aus unbekannten Datenmodellen im Speicher aufzubauen. Implementierungen wurden hierfür durchgeführt, sodass die Machbarkeit demonstriert werden konnte.

7.3.3 Transformationsvorschrift

Unter einer *Transformation* versteht man in der Informatik allgemein eine strukturverträgliche Abbildung zwischen zwei gleich strukturierten Datenmengen oder spezieller auch eine Selbstabbildung auf einer Datenmenge.

Bisherige Beispiele zeigten solche Transformationen als *direkte* Transformation eines Datenmodells kollaborativer Artefakte in ein logisches Artefakt, bei der das Datenmodell des logischen Artefakts ganz oder in Teilen dem ursprünglichen Datenmodell entsprach. Verfahren und Technologien für eine Transformation zur Abbildung von Quelldaten auf Zieldaten sind im Allgemeinen sicherlich ein separat zu betrachtender Themenkomplex. Eine

umfangreiche Betrachtung ist aber auch nicht erforderlich – hierfür sei auf andere Forschungsarbeiten verwiesen.

In diesem Kapitel soll prinzipiell aufgezeigt werden, wie sich ein logisches Artefakt mittels *indirekter* Transformation durch eine Transformationsvorschrift bilden lässt, bei dem das Datenmodell des logischen Artefakts nicht dem ursprünglichen Datenmodell des ursprünglichen kollaborativen Artefakts entspricht.

Eine indirekte Transformation ist demnach technisch charakterisiert durch

a) *eine formale Beschreibung des Zieldatenmodells,*

b) *eine Transformationsvorschrift für die Umformung von Quelldaten*

c) *sowie eine Technologie zur automatischen Ausführung der Transformation.*

Im ersten Schritt a) wird das Metamodell des logischen Artefakts durch eine Person modelliert. Dies ist ein kreativer Denkprozess, der nicht ohne Weiteres automatisierbar ist. Er ist jedoch nur initial erforderlich und daher nur einmalig für die Definition des logischen Artefakts durchzuführen. Hierfür wird die in Kapitel 2.4 beschriebene Metamodellierung grafisch mit der UML-Notation werkzeuggestützt durchgeführt. In den Versuchen wurde dafür der *Enterprise Architect* (Sparx Systems, [SPAX09]) verwendet. Bei der Metamodellierung werden Prozessbegriffe und weitere technische Termini jeweils einem konkreten Datum des Quellartefakts zugeordnet. Bei mehreren kollaborativen Artefakten können auch Datenmodelle von zwei oder mehreren ursprünglichen Quellmodellen im logischen Artefakt verschmelzen.

Die Abbildung 44 zeigt das ursprüngliche Artefakt-Metamodell aus dem Beispiel ***APR-Models.xml*** und das Metamodell des konstruierten, logischen Artefakts. Durch Transformation werden vor der Prüfung die entsprechenden Elemente aus dem ursprünglichen Artefakt in das logische Artefakt umgesetzt.

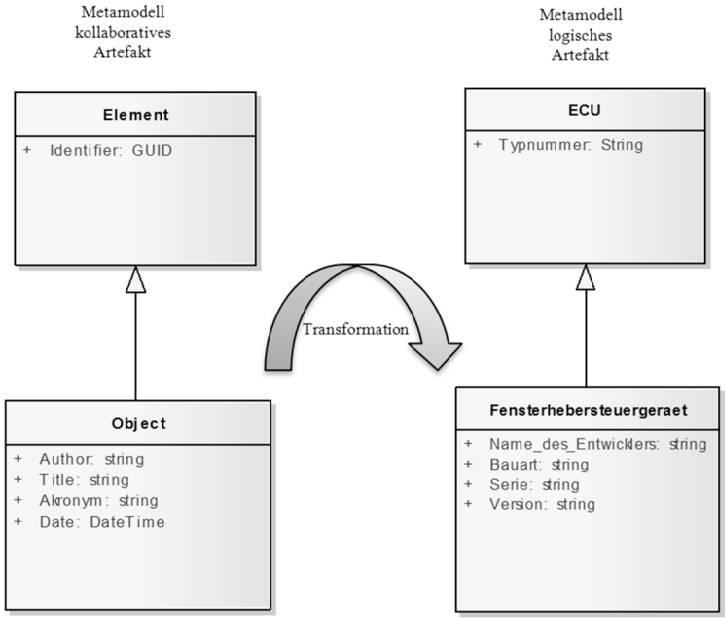


Abbildung 44: Indirekte Transformation mittels Metamodellierung

Die regelbasierte Konformitätsprüfung kann nun auf dem logischen Artefakt semantische Zusammenhänge oder modellierte Beziehungen algorithmisch prüfen. Bei der indirekten

Transformation ist zu beachten, dass häufig auch Datentypkonvertierungen vorgenommen werden müssen. In dem Beispiel ist die Datentypkonvertierungen GUID nach String oder DateTime nach String. Bei der Umwandlung von Datentypen ist daher technisch auf die Genauigkeit (z. B. Rundungsfehler) und die Typverträglichkeit (Abbildbarkeit) der Datentypen zu achten. Das konstruierte Metamodell definiert somit die Struktur des logischen Artefakts und dient als Vorlage für eine Transformationsvorschrift.

Die regelbasierte Prüfung erfolgt auf dem logischen Artefakt. Das bedeutet, dass die in einem Metamodell verwendete Semantik einen direkten Bezug zum Regel-Algorithmus hat. Es ist ferner möglich, so fast natürlich sprachliche Regel-Algorithmen auf einer abstrakteren Ebene zu formulieren, wie nachfolgender Beispiel-Code (LINQ) illustriert.

Beispiel: ***LINQ-Abfrage auf dem logischen Artefakt***

```
        Query = from Elements in Collection
                where Fensterhebersteuergeraet.Serie == „SGFH“
        select Elements;
```

Wurde das logische Artefakt durch das Metamodell definiert, muss nun eine Abbildungsregel in b) gefunden werden. Die Transformation kann nun ebenfalls regelbasiert erfolgen, d. h. durch computerausführbare Algorithmen für die Umformung des ursprünglichen Datenmodells in das logische Artefakt.

Folgendes Beispiel zeigt eine Transformationsvorschrift in LINQ, wie aus dem Quellartefakt `Objects` ein neues, logisches Artefakt gebildet wird, welches die Elemente der ursprünglichen Menge in zwei transformierten Mengen `Upper` und `Lower` überführt. Dabei werden die Elemente der Ursprungsmenge `Objects` durch den Transformationsalgorithmus in Großschreibweise und in Kleinschreibweise umgewandelt sowie auf zwei Zielmengen `Upper` und `Lower` abgebildet.

Beispiel: ***LINQ-basierte Transformation***

```
public void TransformationRule() {

string[] Objects =
         {"FensterheberSteuergeraet","BatterieManagement","Lichtsteuerung"};

    var logicArtifact =

        from Element in Objects
        select new {
            Upper = Element.ToUpper(),
            Lower = Element.ToLower()
        };

    foreach (var lE in logicArtifact)
    {
     Console.WriteLine("Upper:{0},Lower:{1}", lE.Upper, lE.Lower);
    }
}
```

Die *foreach*-Schleife gibt auf der Konsole die Inhalte der Mengen wie folgt aus:

```
Upper: FENSTERHEBERSTEUERGERAET,    Lower: fensterhebersteuergeraet
Upper: BATTERIEMANAGEMENT,          Lower: batteriemanagement
Upper: LICHTSTEUERUNG,              Lower: lichtsteuerung
```

Im Hauptspeicher des Rechners wird das logische Modell, bestehend aus zwei transformierten Mengen, gehalten und kann beliebig abgefragt werden. Das Quellartefakt `Objects` kann analog zum Beispiel auch zunächst aus einer XML-Datei eingelesen werden.

Neben der LINQ-Technologie, die Transformationen erlaubt, existieren auch andere gängige Verfahren, insbesondere wenn die XML-basierte Transformnation betrachtet wird. Denkbar ist hier die Verwendung von *Query View Transformation* (QVT) als deklarative Transformationssprache und Transformationswerkzeug, wie in [UMTQ09] gezeigt. Transformatoren basieren meist selbst auf einer Programmiersprache oder auf einer XSLT-Transformationsvorschrift, welche das .NET-Framework neben LINQ ebenfalls zur Verfügung stellt (wie auch die Java-Laufzeitumgebung JRE).

Versuchsaufbauten wurden auch zur XSLT-Transformation unternommen, speziell wenn ein umfangreiches kollaboratives Artefakt vor einem Prüfdurchlauf gefiltert werden sollte oder wenn ein Metamodell sich bezüglich der Version ändert (neue Programmversion eines Werkzeugs) und eine Instanz vorab korrigiert werden muss. Hierbei ist eine XSLT-Transformation sehr effizient.

Das nachfolgende Beispiel ist eine XSLT-Transformationsvorschrift, welche das vorher eingeführte Artefakt ***APR-Models.xml*** in eine HTML-basierte Struktur überführt.

Beispiel: ***XSLT-basierte Transformation***

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
 <xsl:template match="/">

  <html>
        <head>
         <title>
           Bauart: <xsl:value-of select="/Models/Model/Title"/>
         </title>
         </head>
        <body>
         <h2>
           Name des Entwicklers: <br/>
           <xsl:value-of select="/Models/Model/Author"/>
         </h2>
         <hr>
          <b>Details</b>
          <br/><br/><br/>
          Serie:<br/>
          <xsl:value-of select="/Models/Model/Acronym"/>
          <br/>
         <hr>
          <br/>
           Version:<br/>
          <xsl:value-of select="/company/Date"/>
         <br/>
        </body>
   </html>

  </xsl:template>
</xsl:stylesheet>
```

Die `<xsl:value-of select="<<Reference>>"/>` Anweisungen werden jeweils durch das referenzierte Datum `<<Reference>>` im ursprünglichen Artefakt gefüllt und somit in eine neue Datenmodellstruktur des logischen Artefakts überführt.

Der Automatisierungsaspekt im Punkt c) kann demnach eine in beliebiger Sprache formulierte Transformationsvorschrift sein, welche durch einen computerausführbaren Algorithmus einer Programmiersprache ausgeführt wird.

Der Algorithmus kann in der Ausführungsumgebung entweder als Programmteil eines Wrappers implementiert oder eine Teilkomponente des Prüfsystems selbst sein. Im weiteren Verlauf wird Letzteres verfolgt.

Das Beispielprogramm `XML2HTML` (C#) führt die XSLT-Transformationsvorschrift aus.

Beispiel: *Algorithmus einer Transformation mit XSLT-Transformationsvorschrift*

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Xsl;
using System.Xml.XPath;

public class XML2HTML{

    public static void Main(string[] args)
         {
        if (args.Length == 2){ Transform(args[0], args[1]); }
    }

    public static void Transform
           (string sXmlPath, string sXsltPath){

            //Quell-Artefakt und logisches Artefakt initieren
            XPathDocument myXPathArtifact =
                                  new XPathDocument(sXmlPath) ;

            XslTransform myXslTrans = new XslTransform();

            //Transformationsvorschrift laden
            myXslTrans.Load(sXsltPath);

            //Transformation und logisches Artefakt schreiben
            XmlTextWriter myWriter =
            new XmlTextWriter("LogicalArtifact.html", null);

            myXslTrans.Transform(myXPathArtifact,null, myWriter);

            myWriter.Close() ;
    }
}
```

Als Programm *XML2HTML.exe* kompiliert und ausgeführt, erwartet der Algorithmus zwei Eingangsparameter: Die Referenz zum kollaborativen Artefakt `<sXmlPath>` sowie zur Transformationsvorschrift `<sXsltPath>`. Das Ergebnis der Transformation ist das logische Artefakt `LogicalArtifact.html` (Webseite).

7.4 Untersuchung der Sprachmächtigkeit LINQ

Die in diesem technischen Ansatz gewählte Abfragesprache ist LINQ als eine Teilsprache der .NET-Programmiersprache C#. Es ist zu untersuchen, ob sich alle Kriterien (vgl. Kapitel 5.2), die durch die QRDL vorgegeben sind, durch die LINQ-Abfragesprache als Regeln ausdrücken bzw. sich für die automatische Ausführung implementieren lassen. Daher wird in den folgenden Abschnitten eine Untersuchung vorgenommen und jeweils mit einem kleinen Beispiel untermauert. Eine ausführliche Untersuchung ist aus Platzgründen nicht vorgesehen. Es sei hier daher auf weiterführende Literatur verwiesen, wie [PIAL07; MARG09].

7.4.1 Allgemeine Abfragesyntax

Methoden und Klassen befinden sich vorrangig im Namensraum *System.Linq* die für die Formulierung einer Regel zunächst als Basis zur Verfügung stehen. Die wichtigsten Klassen sind dabei *Enumerable* und *Queryable*, welche zahlreiche Methoden anbieten, um interne Datenobjektstrukturen im Speicher zu verarbeiten. Während es möglich ist, die Methoden dieser Klassen direkt in der Regel zu verwenden, unterstützt C# eine spezielle Abfragesyntax, mit der SQL-ähnliche Abfragen in C#-Code eingebettet werden können. Diese Abfragesyntax ist ein Sprachmerkmal der Programmiersprache C#, um mengenorientierte Abfragen auf Objekten im Speicher durchzuführen.

Ergebnis:

„Dies führt zum Schluss, dass *mengenorientierte Abfragen* auf ein logisches Artefakt durchführbar sind und das komplexe Kriterium der QRDL erfüllen."

Im Folgenden sei dazu ein Beispiel gegeben, das eine beliebige Objektmenge mit der Bezeichnung `ObjectsOrderedByName` dem Namen nach aufsteigend (*ascending*) sortiert als LINQ-Abfrage formuliert.

Beispiel: *Sortier-Algorithmus als Regel*

```
using System;
using System.Linq;
using System.Collections.Generic;

public class ObjectManager
{
  List<Object> objects = new List<Object>();

  public Object[] ObjectsOrderedByName
  {
    get
    {
      IEnumerable<Object> objQuery = from obj in objects
                                     orderby obj.Name ascending
                                     select obj;
      return objQuery.ToArray();
    }
  }
}
```

LINQ bietet Operationen und Funktionen für die Bildung, Traversierung und Verwaltung von Objektmengen an. Durch die Sprachmächtigkeit sind verschiedene Sortierungen und Sortierreihenfolgen möglich.

7.4.2 Wert- und Typabfragen, Selektion, Filter

LINQ arbeitet in Bezugnahme auf eine angegebene Datenquelle. Dies ist in unserem Falle jeweils das logische Artefakt.

Eine LINQ-Abfrage beginnt mit dem Schlüsselwort ***from***, um eine Datenquelle zu referenzieren. Außerdem wird eine allgemeine Variable definiert, über die auf Datensätze verwiesen werden kann, um diese in irgendeiner Form abzufragen. Dies geschieht durch Wertabfragen oder Typabfragen und logische Operatoren (==, &&, <=, >=, ||). Somit lässt

sich ein Datum mit einem anderen Datum zu einem logischen Ausdruck in Beziehung setzen.

Zusätzlich können auch arithmetische Operationen in LINQ als Bruchoperation (Addition, Subtraktion, Division, Multiplikation) mit in die LINQ-Abfrage eingebaut werden.

Ergebnis:

Die primitiven Bedingungen der QRDL, wie

a) Arithmetik (Addition, Subtraktion, Multiplikation, Division),

b) Boolesche Ausdrücke und Boolesche Algebra,

c) Aussagenlogik: Aussagen und Prädikate

sind somit durch LINQ umsetzbar.

Neben der Möglichkeit zur Attributabfrage gelingt es mit LINQ-Abfragen auch, eine Selektion von Objekten (Filter-Mechanismus) aufgrund eben einer geltenden Bedingung zu erstellen. Folgendes Beispiel illustriert die bisherige Erkenntnis.

Beispiel: *Filter-Algorithmus als Regel*

```
using System;
using System.Linq;
using System.Collections.Generic;

public class ObjectManager
{
  List<Object> objects = new List<Object>();

  public Object[] ObjectsFilteredByArgument
  {
    get
    {
      IEnumerable<Object> objQuery = from obj in objects
           where obj.GUID == "C8F2355C-63CB-4051-AA68-7739A947589A"
           select obj;
      return objQuery.ToArray();
    }
  }
}
```

Im Beispiel wird aus einer beliebigen Objektmenge `ObjectsFilteredByArgument` genau das Datum aus einem logischen Artefakt ausgewählt, das die eindeutige Bezeichnung `C8F2355C-63CB-4051-AA68-7739A947589A` (GUID) aufweist.

7.4.3 Variable, Gruppierung

Die Grammatik von LINQ kennt Schlüsselwörter ***group***, ***by*** und ***into*** für die Gruppierung von Daten. Während mittels ***group*** auf die Variable zugegriffen wird, die vormals mittels ***from*** definiert wurde, wird hinter dem Schlüsselwort ***by*** das Kriterium angegeben, welches bei der Gruppenbildung gelten soll. Mittels ***into*** wird eine neue Variable definiert, um Gruppen zu bearbeiten. Mittels der gefundenen Funktionen lassen sich aus der Prädikatenlogik bekannte Konzepte wie Variablen und Quantoren (Allquantor, Existenzquantor) implementieren.

Ergebnis:

„Dies führt zum Schluss, dass Konzepte der *Prädikatenlogik erster Ordnung*, nämlich Variable und Quantoren (Allquantor, Existenzquantor), sich in LINQ implementieren lassen und auf ein logisches Artefakt durchführbar sind (komplexes Kriterium der QRDL)."

Folgendes Beispiel veranschaulicht die Gruppierung von Objekten anhand eines Datums (GUID) und die Bildung einer neuen, geordneten Menge `groupedObjects`.

Beispiel: *Gruppier-Algorithmus als Regel*

```
using System;
using System.Linq;
using System.Collections.Generic;

public class ObjectManager
{
  List<Object> objects = new List<Object>();

  public Object[] ObjectsGroupedByArgument
  {
    get
    {
      IEnumerable<IGrouping<string, Object>>
         objQuery = from obj in objects
         group obj by obj.GUID into objHosts
        select objHosts;

      List<Object> groupedObjects = new List<Object>();
      foreach (var group in objQuery)
      {
        foreach (var obj in group)
        {
          groupedObjects.Add(obj);
        }
      }
      return groupedObjects.ToArray();
    }
  }
}
```

Im obigen Beispielcode gibt die LINQ-Abfrage Gruppen von Objekten zurück, die jeweils einen gleichen Bezeichner (GUID) aufweisen. Dies ist im Datentyp der Variable `objQuery`, der nun `IEnumerable<IGrouping<string,Object>>` lautet. Beim generischen Interface `IGrouping`, welches im Namensraum *System.Linq* definiert ist, handelt es sich um eine Beschreibung von Gruppen. Die Datentypen, die als Parameter in spitzen Klammern übergeben werden müssen, kennzeichnen einen Index-Schlüssel und die jeweiligen Elemente der Gruppe.

Gruppen lassen sich mittels bekannter Programmkontrollstrukturen (z. B. einer ***foreach***-Schleife) verarbeiten. Während die äußere Schleife Objekte zurückgibt, die `IGrouping` implementieren, ermöglicht die innere Schleife es, Zugriff auf die Objekte in den Gruppen zu nehmen bzw. diese zu ändern.

7.4.4 Aggregation

Aggregationsoperationen werden auf die Ergebnisse von LINQ-Abfragen angewandt. Es handelt sich hierbei um herkömmliche Methodenaufrufe, welche anstatt einer Menge von Objekten eine skalare Größe als Ergebnis liefern. Dies ist beispielsweise die Anzahl aller gefundenen Daten mit einem identischen Datum oder Datentyp oder die Summe aller Elemente einer Menge.

Ergebnis:

„Dies führt zum Schluss, dass Konzepte der Prädikatenlogik erster Ordnung, nämlich *Aggregationen* (wie z. B. Sum, Min, Max, InStr, Count, Group, Join), sich in LINQ implementieren lassen und auf ein logisches Artefakt durchführbar sind (komplexes Kriterium der QRDL).“

Nachfolgendes Beispiel demonstriert die Ermittlung der Anzahl der Elemente einer Menge als skalare Größe.

Beispiel: *Aggregation-Algorithmus als Regel*

```
using System;
using System.Linq;
using System.Collections.Generic;

public class ObjectManager
{
  List<Object> objects = new List<Object>();

  public int Length
  {
    get
    {
      return (from obj in objects
              select obj).Count();
    }
  }
}
```

Der Rückgabewert ist eine skalare Größe (Datentyp: Integer), welche die Menge aller Objekte enthält. Für Vergleiche zur Ermittlung der Konformität (z. B. zu einem vorgegebenen Schwellwert) sind solche Sprachmittel zwingend erforderlich. Zudem lässt sich das hier gezeigte Beispiel auch noch weiter ausbauen, indem zusätzliche Abfrageeigenschaften durch das ***where***-Schlüsselwort mit angegeben werden.

7.4.5 Sortierung

Es wurde bereits demonstriert, dass mit dem Schlüsselwort ***orderby*** eine aufsteigende (*ascending*) oder absteigende (*descending*) Sortierung von Elementen einer Menge vorgenommen werden kann.

Ergebnis:

„Dies führt zum Schluss, dass *mengenorientierte Sortierungen* durch Kriterien auf ein logisches Artefakt durchführbar sind (komplexes Kriterium der QRDL).“

Als weiteres Kriterium soll im folgenden Beispiel die Sortierung jedoch in Abhängigkeit eines Attributwertes erfolgen. Dies ist ein sehr typisches Verfahren für eine Regel.

Beispiel: *Parameterabhängiger Sortier-Algorithmus als Regel*

```
using System;
using System.Linq;
using System.Collections.Generic;

public class ObjectManager
{
  List<Object> objects = new List<Object>();

  public Object[]OrderedAndFilteredObjects
  {
    get
    {
      IEnumerable<Object> objQuery =

        from obj in objects
        where obj.GUID == "C8F2355C-63CB-4051-AA68-7739A947589A"
        orderby obj.Name, obj.GUID
       select obj;

      return objQuery.ToArray();
    }
  }
}
```

Im Beispiel wird gezeigt, wie sich zunächst eine Teilmenge von den Objekten einer Menge bildet, welche den gleichen Bezeichner (GUID) aufweisen. Zusätzlich wird diese Teilmenge sortiert nach dem Datum `Name` und nach dem Datum `GUID`.

7.4.6 Projektion

Bei einer Projektion werden Elemente einer Menge aufgrund ihrer Eigenschaft klassifiziert. Genau die klassifizierte Teilmenge (z. B. ein einzelnes Datum) wird aus der Ursprungsmenge extrahiert und als eine neue Menge zurückgegeben.

Ergebnis:

„Dies führt zum Schluss, dass mengenorientierte *Projektionen* durch Kriterien auf ein logisches Artefakt durchführbar sind (komplexes Kriterium der QRDL).“

Folgendes Beispiel extrahiert nur die Bezeichner (*Name*) eines Objekts aus einer Menge von Objekten durch die Klassifizierung des Datums *Name* und wird als Liste zurückgegeben.

Beispiel: *Sortier-Algorithmus als Regel*

```
using System;
using System.Linq;
using System.Collections.Generic;

public class ObjectManager
{
  List<Object> objects = new List<Object>();

  public Object[] SelectedObjects
  {
```

```
    get
    {
      IEnumerable<string> objQuery = from obj in objects
                                     orderby obj.Name
                                     select obj.Name;
      return objQuery.ToArray();
    }
}}
```

7.4.7 Fazit

Die Untersuchung hat im Ergebnis gezeigt, dass sich sowohl primitive Anforderungen sowie komplexe Anforderungen der QRDL in der Abfragesprache LINQ formulieren und als ein Regel-Algorithmus für die Konformitätsprüfung implementieren lassen.

Da LINQ als eine Erweiterung der objektorientierten Programmiersprache C# konzipiert und folglich in dieser eingebettet ist, stehen auch alle Klassen und Funktionen aus dem mächtigen Sprachumfang von C# für den Regelalgorithmus zur Verfügung. Dies sind insbesondere benötigte Operationen für Zeichenkettenverarbeitung (reguläre Ausdrücke) aus dem Namespace *System.Text.RegularExpressions* sowie umfangreiche Programm-kontrollstrukturen für die effiziente Abarbeitung auch komplexerer, rekursiver oder verschachtelter Algorithmen.

7.5 Offene Prüfsysteme kollaborativer Artefakte

Das Prüfsystem muss sowohl das logische Artefakt in den Hauptspeicher des Rechners laden können als auch QRDL-Regelalgorithmen in einer Programmiersprache formuliert auf diesem ausführen können. Schließlich wird zur Auswertung noch ein Ausgabemechanismus für das ermittelte Prüfergebnis (Reporting) benötigt. Während dieser Arbeit wurden hierfür MATLAB als Prüfsystem für M-Skript-basierte Prüf-Skripte angewandt, OSLO für OCL-basierte Prüf-Skripte verwendet und das .NET-Framework 3.5 für LINQ-basierte Prüf-Abfragen umgesetzt. Die beiden Prüfsysteme, mittels OCL-Regelsprache durch das Werkzeug *ASD-Regel-Checker* sowie LINQ durch das *Assessment Studio*, werden in den nachfolgenden Kapiteln vorgestellt. Das *Assessment Studio* erlaubt zudem die Anbindung des MATLAB-Prüfsystems durch einen Wrapper-Mechanismus zur Ausführung von M-Skript programmierten Regelalgorithmen. Hierdurch wird der im Kapitel 7.3.1 konzipierte Wrapper-Mechanismus demonstriert.

7.5.1 Prüfsystem des ASD-Regel-Checker

Im internen Prüfsystem des *ASD-Regel-Checker* sollten formulierte Regelausdrücke in der OCL-Sprachsyntax artikuliert und auf dem logischen Artefakt ausgeführt werden können. Technisch erfolgte die Auswertung von OCL-Regeln durch eine OCL-Engine mit der Bezeichnung OSLO. Die *Open Source Library for OCL* (OSLO) ist eine durch das Fraunhofer Institut FOKUS vorangetriebene Weiterentwicklung der Kent OCL Bibliothek.

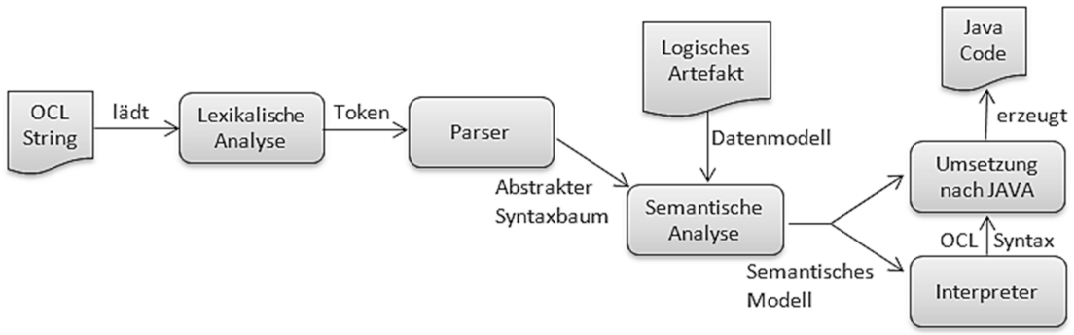


Abbildung 45: Strukturbild des Prüfsystems mit OCL, nach [OSLO09]

Sie implementiert den Sprachstandard OCL Version 2.0 und erlaubt es, OCL-Ausdrücke auf beliebigen Modellen des *Eclipse Modeling Framework* (EMF) auszuwerten. Die Abbildung 45 skizziert die prinzipielle Funktionsweise der OSLO-Engine, nach [OSLO09].

OSLO besteht aus einer zentralen Programm-Bibliothek *OCLCommon*, die vordefinierte OCL-Standardoperationen gemäß OMG Spezifikation unabhängig von der zugrundeliegenden Metamodellierungstechnik bereitstellt. Hinzu kommen verschiedene Brücken, welche die Basisfunktionen jeweils für eine bestimmte Metamodellierungstechnik adaptieren.

Im Forschungsprojekt MESA wurde *OCL4EMF* gewählt und somit das logische Artefakt in einer EMF-Modelldatenbank (Repository) erzeugt. Das Strukturbild von OSLO zeigt die Struktur von *OCLCommon*. In dem Prüfwerkzeug *ASD-Regel-Checker* wurde ein repräsentierter OCL-Ausdruck durch einen String angegeben, sodass dieser die typischen Phasen eines Interpreters durchlaufen musste. Zunächst wird beim Interpreter die Eingabezeichenkette (OCL-Ausdruck) in der lexikalischen Analyse in zusammenhängende elementare Bestandteile, so genannte *Token*, zerlegt. Die Zeichen im OCL-String werden separiert in Leerzeichen, reservierte Schlüsselwörter, Bezeichner und Strukturierungssymbole (z. B. Klammern) nach Grammatik der OCL.

Beispiel: *Tokens eines OCL-Ausdrucks*

```
Schlüsselwort(„context“) Bezeichner(„ECU“) Schlüsselwort(„inv“)
Strukturierungssymbol(„:“) Bezeichner(„MaxSpannung“) Bezeichner(„<“)
Zahl(12) Strukturierungssymbol(„.“)
```

Die syntaktische Struktur von OCL-Ausdrücken ist durch eine formale Grammatik beschrieben (vgl. dazu die OCL-Spezifikation). Nach lexikalischer Analyse durchläuft ein Parser den OCL-String und fasst identifizierte Token zu einem Element im abstrakten Syntaxbaum zusammen.

Der abstrakte Syntaxbaum ist dabei eine baumartige Darstellung der syntaktischen Struktur eines OCL-Ausdrucks. Für den einfachen OCL-Ausdruck würde sich ein Syntaxbaum wie in der Abbildung ergeben.

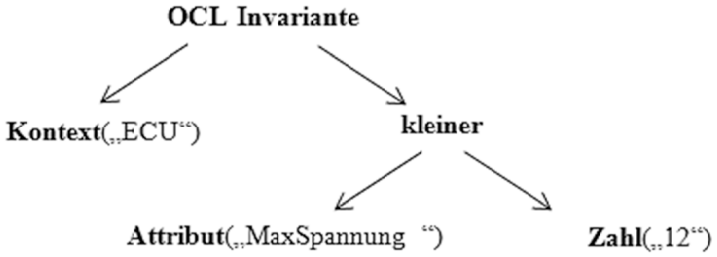


Abbildung 46: Beispiel eines abstrakten Syntaxbaums in OCL

Im Wurzelknoten wird die OCL-Invariante behandelt. Der linke Unterknoten beschreibt für den nächsten Arbeitsschritt, für welche Elemente („*Kontext“*) im Datenmodell der Regelausdruck gelten soll. Im rechten Unterknoten wird die vordefinierte Operation „*kleiner*“ identifiziert, welche in zwei Vergleichsattribute zerfällt. In den Blättern des Syntaxbaums findet sich schließlich die Aussage, dass der Wert des Attributs „*MaxSpannung*“ mit der Konstanten „*12*“ zu vergleichen ist.

Im nächsten Schritt erfolgt die semantische Analyse des Syntaxbaums. Dabei wird von unten nach oben geprüft, ob alle benutzten Operationen existieren und die Typen der Argumente mit der Operationsdeklaration mit der OCL-Grammatik übereinstimmen. Zusätzlich muss die Kenntnis über das Metamodell des logischen Artefakts einfließen. Im gezeigten Beispiel würde erkannt werden, dass das Attribut „*MaxSpannung*“ gemäß dem Metamodell ebenso wie die Konstante „*12*“ dem Integer-Datentyp zugeordnet ist. Des Weiteren existiert die Vergleichsoperation „*kleiner“*, welche die zwei Attribute vom Integer-Datentyp *(int)* verarbeitet. Diese Operation wird durch die semantische Analyse ermittelt. Sie ist zwar nicht explizit deklariert worden, steht jedoch durch die OSLO-Bibliothek *OCLCommon* gemäß ihrer Definition in der OCL-Spezifikation zur Verfügung.

Der Funktionsdeklaration ist der zugeordnete Rückgabewert zu entnehmen. Die Vergleichsoperation liefert erwartungsgemäß einen booleschen Rückgabewert (*Wahrheitswert*). Dies ist wiederum konform zu der Invarianten, welche stets zu *true* oder *false* ausgewertet werden muss. Damit ist die semantische Analyse erfolgreich ohne Fehler beendet und es folgt schließlich die Interpretation des Ausdrucks, d. h. die Bestimmung des tatsächlichen Ergebniswertes. Dazu wird der abstrakte Syntaxbaum wiederum von unten nach oben durchlaufen und in jedem Schritt werden die Werte der Blattknoten bestimmt und nach oben propagiert. Im vorliegenden Beispiel handelt es sich um eine OCL-Invariante, d. h., die Interpretation wird einmal für jede, durch den Kontext referenzierte, Klasse durchgeführt. Demzufolge werden zuerst alle Instanzen der Klasse „*ECU*“ aus dem logischen Artefakt im Repository geladen und für jede Klasse eine Kopie des abstrakten Syntaxbaums als Auswertungsbaum angelegt. Dies bedeutet, es gibt nun ebenso viele auszuwertende Bäume wie Klasseninstanzen von „*MaxSpannung*“.

Im Syntaxbaum und im Auswertungsbaum wird nun anhand der zugeordneten Klasseninstanz der linke Blattknoten durch den Attributwert ersetzt und der rechte Blattknoten durch die Konstante ausgetauscht. Zudem muss bestimmt werden, ob der Wert im linken Blatt echt kleiner ist als der im rechten Blatt und der Wurzelknoten „*kleiner*“ durch das Ergebnis ersetzt werden kann. Damit sind sämtliche Auswertungsbäume auf einen einzigen booleschen Wert, einen Wahrheitswert, reduziert worden. Da es sich um eine OCL-Invariante handelt, wird abschließend die konjunktive Verknüpfung jeglicher dieser Wahrheitswerte nach Boolescher Algebra bestimmt. Schließlich liefert die

Zusammenfassung der Wahrheitswerte das Ergebnis des OCL-Ausdrucks. Dies sind nach OCL-Sprachstandard die Warheitswerte *true* oder *false.* Dies ist zwar ausreichend für die Ermittlung auf Konformität des logischen Artefakts, jedoch ist für die Auswertung auch von Interesse, *welche* Elemente oder Attribute des logischen Artefakts nicht konform sind.

Hierfür wurde eine Erweiterung des OCL-Sprachstandards im *ASD-Regel-Checker* vorgenommen, dass eine Invariante, wenn als Kontext *OclVoid* angegeben ist, nicht unbedingt einen booleschen Wert liefern muss, sondern einen beliebigen OCL-Typen zurückliefern darf. In diesem Falle würde, nach der Kompression aller Auswertungsbäume auf jeweils einen einzigen Knoten, die Ergebnismenge aus der Vereinigung aller dieser Knoten zu einer Menge bestehen.

Für weitere technische Umsetzungsdetails, unter anderem wie der abstrakte Syntaxbaum während der semantischen Analyse mit hilfreichen Zusatzinformationen annotiert wird und wie die Brücke zwischen *OCLCommon* und *OCL4EMF* durch das Mapping von Metamodellen zueinander funktioniert, sei auf (Veröffentlichungen, Nr. 28, 29) verwiesen sowie die gängige Literatur zum Compilerbau empfohlen.

7.5.2 Prüfsystem des Assessment Studio

Im Falle OSLO wurde ein Interpreter eingesetzt, der den OCL-Regelausdruck Anweisung für Anweisung übersetzt und ihn danach ausführt. Die Analyse der Regel erfolgt also zur Laufzeit des Prüfsystems. Es wird im Unterschied zu Compilern und Assemblern keine auf dem System direkt ausführbare Datei erstellt. Nachteil von Interpretern ist die geringere Ausführungsgeschwindigkeit. So müssen beispielsweise Anweisungen in Schleifen bei jeder Iteration neu übersetzt werden. *Just-in-Time*-Compiler, wie in der .NET-Laufzeitumgebung vorhanden, sind Zwischenlösungen und können einen Geschwindigkeitsvorteil erbringen. Auch hier wird das Programm zur Laufzeit übersetzt, jedoch direkt in Maschinencode, der direkt vom Prozessor ausgeführt wird. Da der Maschinencode zwischengespeichert wird, müssen mehrfach zu durchlaufende Programmteile nur einmal übersetzt werden. Ein Bytecode-Interpreter wiederum übersetzt den Quelltext zur Laufzeit vor seiner Ausführung in den Zwischencode. Dieser Zwischencode wird von einem Interpreter ausgeführt.

Die Einbettung in eine beliebige Programmiersprache der .NET-Laufzeitumgebung, wie Visual Basic oder C#, ist ein grundlegender Aspekt von LINQ. Der wesentliche Unterschied von LINQ-to-XML in Gegenüberstellung zur Auswertung von OCL-Ausdrücken mittels Interpreter besteht darin, dass Abfragen in LINQ vor ihrer Ausführung statisch mit dem restlichen Programm (LINQ-Klassenbibliotheken) vom *Just-in-Time*-Compiler kompiliert werden müssen. Sie sind förmlich in die .NET-Programmiersprache eingebettet. OCL-Regelausdrücke wie auch andere Ausdrücke anderer Technologien (wie z. B. XPath-Abfragen) müssen dagegen zur Laufzeit des Prüfsystems interpretiert werden, wie im Falle der OCL-Regelsprache durch die OSLO-Umgebung geschehen. Da für die Interpretation von komplexeren Regelalgorithmen zur lexikalischen Analyse, insbesondere bei *Rekursion*, deutlich mehr Rechenzeit für die regelbasierte Konformitätsprüfung aufgewendet werden muss, sind im Gegensatz dazu kompilierte LINQ-Ausdrücke effizienter im Ablauf. Nachteilig ist jedoch, dass die Regelalgorithmen *statisch* im Programmcode eingebettet sind und nicht *dynamisch* während der Laufzeit als ‚String' geladen und interpretiert werden können. Dies bedingt immer einen vollen Compilerdurchlauf und Start des resultierenden Programms zur Ausführung der regelbasierten Konformitätsprüfung am kollaborativen Artefakt.

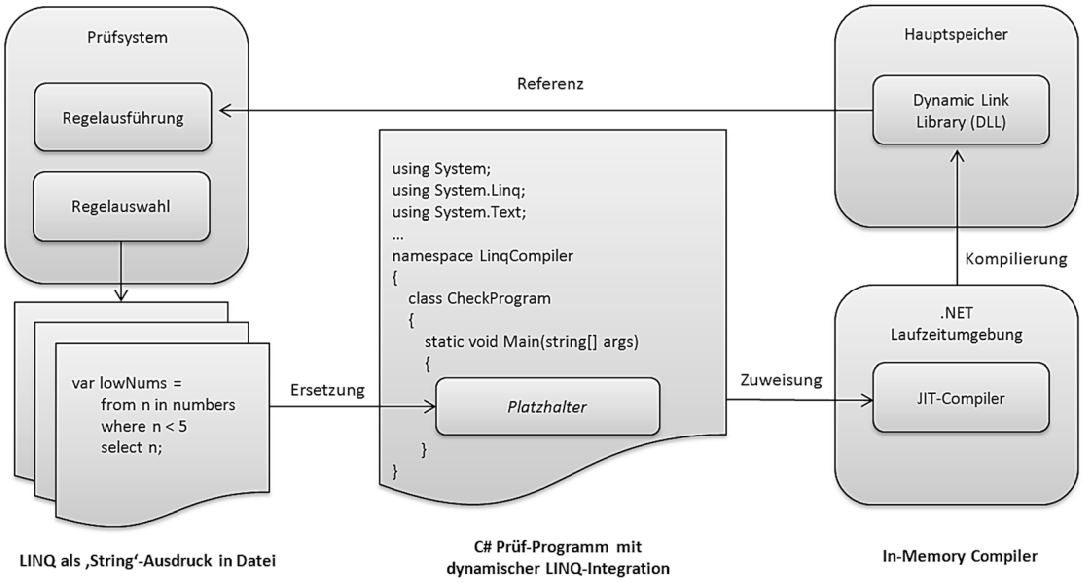


Abbildung 47: .NET Erweiterung - Dynamisches Laden, Ausführen der LINQ-Query

Das *dynamische Laden* einer QRDL-Regel im *String*-Format, wie es für OCL-Abfragen vorgestellt wurde und auch für gängige SQL-Abfragen technisch verfügbar ist, wird von der verwendeten .NET-Laufzeitumgebung 3.5 nicht unterstützt. Die deskriptive Auslagerung einer Regel in einen LINQ-String, der zur Laufzeit geladen und interpretiert würde, könnte eine höhere Flexibilität insbesondere bei der Entwicklung der Regeln erlauben.

Um diese Einschränkung zu umgehen, wurde in dieser Arbeit die LINQ-Technologie des .NET-Frameworks 3.5 eigens durch einen Mechanismus programmatisch erweitert, um im Prüfsystem die deklarative Beschreibung von Regeln in LINQ (als *String*-Ausdruck) dynamisch zu laden sowie die Kompilierung und Ausführung zur Laufzeit der Ausführungsumgebung zu ermöglichen. Die Abbildung 47 zeigt das Strukturbild zum dynamischen Laden und Ausführen von LINQ-Queries.

7.6 Prüfwerkzeug (Assessment Studio)

Zur werkzeuggestützten Unterstützung der vorgestellten Methodik nach dem VR-Modell sowie zur Automatisierung des in dieser Arbeit vorgestellten Ansatzes zur regelbasierten Konformitätsprüfung kollaborativer Artefakte wurde nach dem ersten Prototypen *ASD-Regel-Checker* (Prüfung mittels OCL und Modell-Repository) das Prüfwerkzeug *Assessment Studio* (Abbildung 48) mit einer alternativen Ausführungsumgebung (logisches Artefakt und Regelsprache) auf einer anderen Technologie-Plattform entwickelt.

Das *Assessment Studio* erlaubt die umgesetzte Werkzeugunterstützung des Lösungswegs mit einer regelbasierten Prüfung von Konformitätsrichtlinien mittels LINQ an kollaborativen Artefakten durch automatische Konformitätsprüfung und einer Prüfberichterstattung (Auswertung). Dabei wird, durch das im Kapitel 7.3.1 vorgestellte Wrapper-Konzept, das Prüfsystem von der Laufzeitumgebung und der Auswertung insofern entkoppelt, sodass sich theoretisch beliebige Programmiersprachen für die Implementierung der QRDL-strukturierten Regeln anwenden lassen. Dies wurde exemplarisch für das Prüfsystem mit XML durch LINQ sowie für das Prüfsystem mit MATLAB durch M-Skript erprobt.

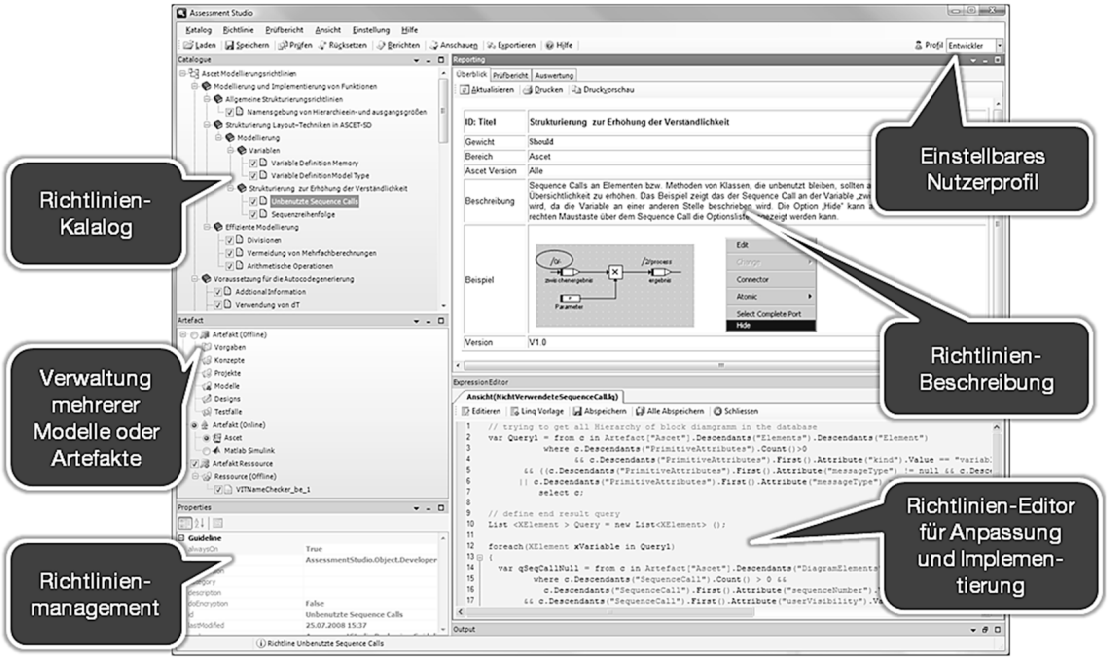


Abbildung 48: Prüfwerkzeug Assessment Studio

Die Prüfung kollaborativer Artefakte basiert im Prüfsystem des *Assessment Studio* auf dem offenen XML-Format, in dem die Daten bereits vorliegen oder aber durch das *Assessment Studio* direkt transformiert (konvertiert) werden können. Das zugrunde liegende Prüfsystem ist nach dem im Kapitel 7.5.2 erläuterten Prinzip umgesetzt worden.

7.6.1 Ausführungsumgebung

Die Ausführungsumgebung des *Assessment Studio* hat eine grafische Benutzeroberfläche mit zwei Sichten der beiden Akteure: a) *Prüfer* (Assessor) und b) *Regelentwickler* (Developer), wie aus der Anforderungsanalyse im Kapitel 7.1 ermittelt.

Die Benutzeroberfläche (Abbildung 48) bestehet aus einem Richtlinienbaum, dem digitalen Richtlinienkatalog und einem strukturierten Artefakt-Baum (dargestellt auf der linken Seite); zudem den Kommandoschaltflächen zum Verwalten, Selektieren, Starten oder Drucken von Richtlinien (obere Menüleiste) sowie der Anzeige von HTML-basierten, natürlich sprachlichen Richtlinienbeschreibungen (rechte Seite). Ein Editor ist für die Entwicklung oder Anpassung von Regelalgorithmen (untere Seite) integriert. Er ist stets mit der selektierten Regel im Richtlinienbaum verbunden, sodass die zu einer Richtlinie assoziierte Regel in der jeweiligen Programmiersprache erstellt, angesehen (in der Developer-Sicht) oder verändert sowie gespeichert werden kann. Der Richtlinienbaum enthält alle von einem Regelentwickler erstellten Richtlinien zur Laufzeit. Eine Richtlinie enthält neben der textuellen Beschreibung einige Meta-Informationen und jeweils eine Referenz auf eine Implementierungsdatei der Regel. In den nachfolgenden Kapiteln werden die wesentlichen Komponenten des *Assessment Studio* im Detail vorgestellt.

7.6.2 Richtlinien

Natürlich sprachliche Entwicklungsrichtlinien können aus DOORS, aus einem Word- oder PDF-Dokument in eine HTML-Seite konvertiert (vgl. Beispiele in Kapitel 7.3.3) und in der Benutzeroberfläche für den Anwender angezeigt werden.

Dabei wird jeweils die im linken Regelkatalog mit der Maus selektierte Richtlinie in dem Übersichtsfenster (rot eingerahmt) dargestellt. Der Anwender erhält so die Möglichkeit, sich über die Anforderung der Richtlinie zu informieren und ggf. Positiv- und Negativ-Beispiele einzusehen. Des Weiteren kann eine zusätzlich angegebene Begründung zur Richtlinie helfen, den Sinn und Zweck sowie den Anwendungskontext zu verstehen.

Die hier gewählte HTML-Formatierung wurde gewählt, da mehrere positive Aspekte dadurch erreicht werden:

- *Verteilung*: Die Richtlinie lässt sich auch mittels URL-Referenz einbinden, sodass diese zentral über einen Richtlinien-Server im Intranet/Extranet zur Verfügung gestellt werden kann (Verteilung im Unternehmen).
- *Dynamik*: Das HTML-Format erlaubt die dynamische Einbettung von Hyperlinks.
- *Darstellung*: Es kann eine dynamische Skalierung an die Fenstergröße erfolgen. Dies ist auch für eine Druckausgabe der Richtlinie relevant. Eine implementierte Zoomfunktion erlaubt die Vergrößerung oder die Verkleinerung der Abbildungen (z. B. der dem Benutzer gezeigten Konformitätsbeispiele).
- Anpassbarkeit: Die HTML-Vorlage ist mittels *Cascading Style Sheet* (CSS) flexibel an verschiedene Layouts von Unternehmen anpassbar.

Die Abbildung 49 zeigt die Einbettung der konvertierten Richtlinienbeschreibung in die Benutzeroberfläche (GUI).

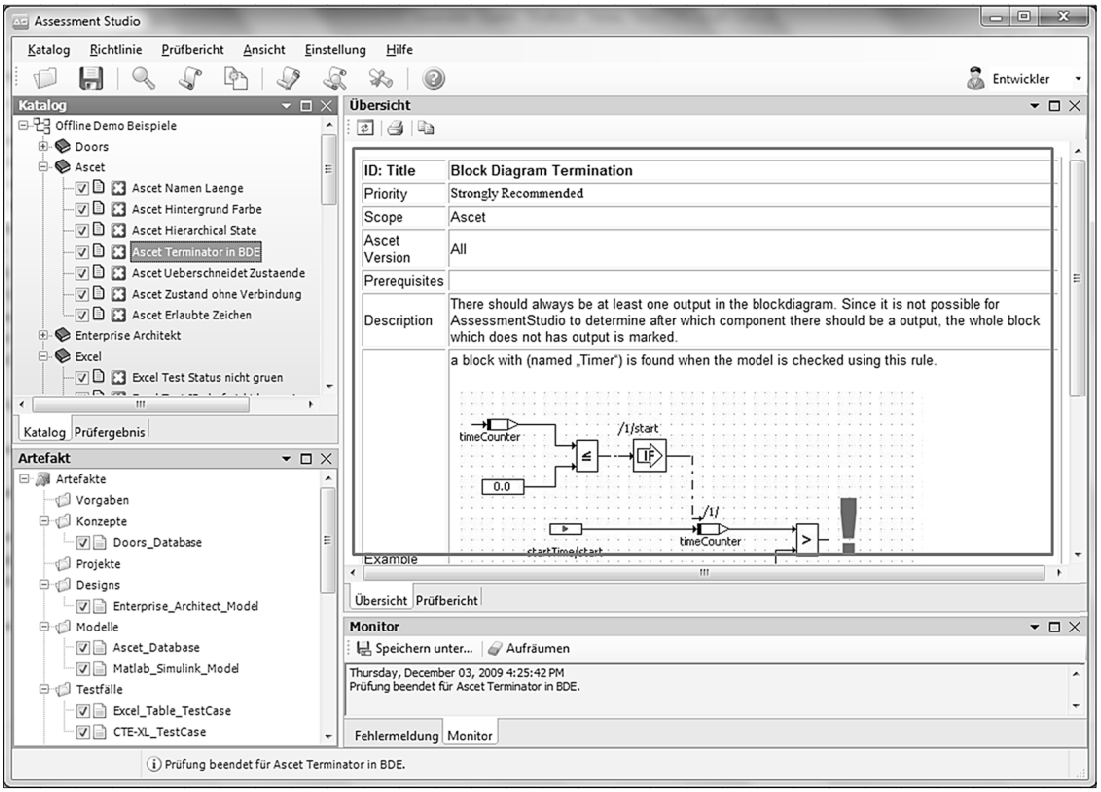


Abbildung 49: Richtlinien-Selektion und HTML-Ansicht

7.6.3 Regelkatalog

In einem hierarchisch strukturierten Regelkatalog können die einzelnen Richtlinien zusammengefasst und thematisch gruppiert werden. Die Gruppierung erfolgt in Ordnern, wie es z. B. bei einem Dateisystem üblich ist. Eingruppierungen erfolgen sodann unter frei wählbaren Themen wie Darstellung, Layout, Namensgebung oder Codegenerierung. Nachfolgende Abbildung 50 zeigt einen Ausschnitt aus dem hierarchischen Regelkatalog mit der MAAB-Richtlinienstruktur für ML/SL/SF als Beispiel.

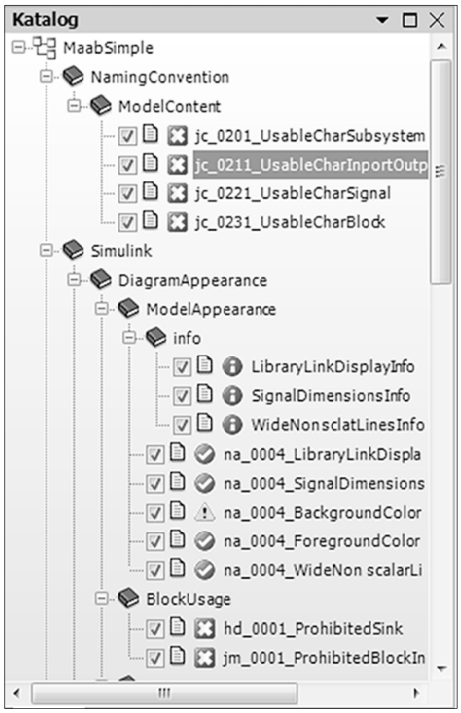


Abbildung 50: Umsetzung eines Regelkatalogs (MAAB)

Es besteht die Möglichkeit zur Auswahl der jeweiligen Richtlinien aus dem Regelkatalog vor einem Prüfdurchlauf, in dem diese selektiert oder abgewählt werden (durch ein Häkchen dargestellt). Zusätzlich wird mittels stilisierter *Icons* im Baum neben dem Namen der Richtlinie angezeigt, ob die Richtlinie bereits geprüft wurde und falls ja, welches Prüfergebnis (z. B. Information, Warnung, Hinweis, Abbruch) erreicht wurde. Die jeweilig zuletzt konfigurierte Einstellung wird vom Prüfwerkzeug gespeichert, sodass Auswahl und Status eines letzten Prüfdurchlaufs erhalten bleiben.

Funktionen zum Zurücksetzen (Reset) des Baums auf den ursprünglichen Zustand wurden für die Benutzerfreundlichkeit (Usability) ebenfalls implementiert. Mehrere Richtlinien können sodann im werkzeugspezifischen Format als ein hierarchischer Baum (Regelkatalog) strukturiert, gespeichert und wieder für eine Prüfung geladen werden. Die Wrapper sind als modulare Plug-In-Komponenten in das Prüfwerkzeug integrierbar. Dieses Wrapper-Konzept erlaubt die spätere Anbindung anderer Prüfsysteme nach Kapitel 7.3.1. Während der Arbeit ist ein Adapter für kollaborative Artefakte in XML sowie in ML/SL/SF entwickelt worden. Die Ausführung der Entwicklungsrichtlinien kann dadurch mittels der Programmiersprachen LINQ oder M-Skript erfolgen. Sie sind verknüpft mit Programmskripten (z. B. M-Skript oder LINQ Dateien), welche mit dem Prüfwerkzeug durch den Anwender ausgeführt und ausgewertet werden können.

7.6.4 Entwicklungsumgebung für Regeln

Zu jeder Richtlinie im Regelkatalog ist jeweils die Regel selbst als computerausführbarer Regelalgorithmus in der jeweiligen Programmiersprache (M-Skript oder LINQ) hinterlegt. Dies erfolgt durch die Angabe des Dateipfades zur Regelimplementierungsdatei. Diese kann im Prüfwerkzeug in der Entwicklungsumgebung für Regeln geladen, angesehen, verändert und gespeichert werden. Nachfolgende Abbildung 51 zeigt die Entwicklungsumgebung für Regeln.

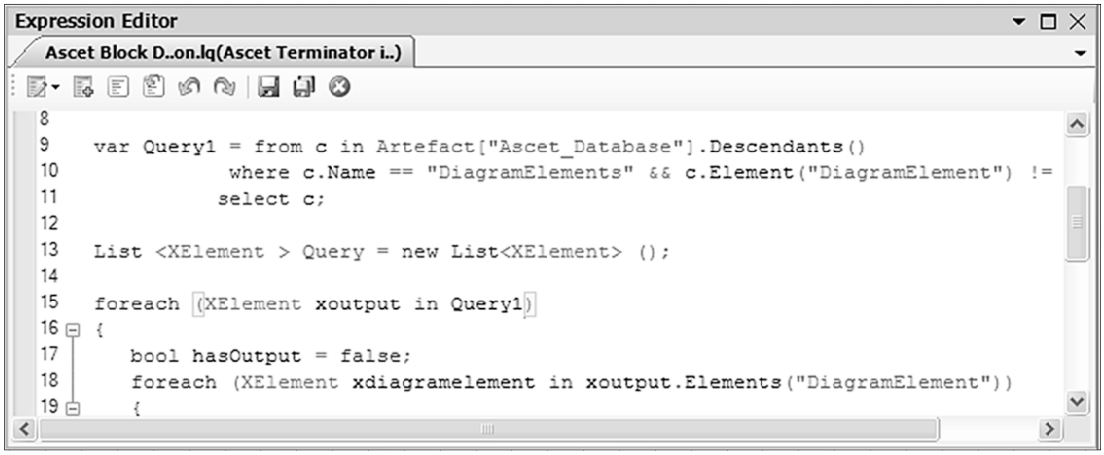


Abbildung 51: Entwicklungsumgebung für Regeln

Durch die implementierte Syntaxhervorhebung werden vordefinierte Schlüsselwörter und andere Sprachelemente der Programmiersprache im Editor farblich hervorgehoben. Zusätzliche Funktionsschaltflächen (*New, Copy, Paste, Undo, Redo, Save, Save all*) erleichtern die Programmierung sowie die Speicherung eines veränderten Regelalgorithmus.

7.6.5 Kollaborative Artefakte

Zur Festlegung des Prüfaufbaus müssen aus dem Prüfraum verschiedene kollaborative Artefakte vor einem Prüfdurchlauf selektiert werden können. Hierzu wird das Konzept des Regelkatalogs auf Artefakte übertragen und somit ein Artefakt-Baum durch das Prüfwerkzeug bereitgestellt. Es kann pro Prozessphase ein Ordner angelegt und dieser benannt sowie die dazu gehörigen Artefakte unter diesem eingruppiert werden. Nachfolgende Abbildung 52 zeigt einen Ausschnitt aus dem hierarchischen Artefakt-Baum.

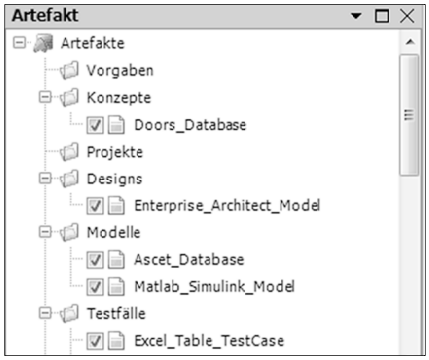


Abbildung 52: Kollaborative Artefakte im Baum

Die im Artefakt-Baum referenzierten Artefakte werden – sofern sie nicht bereits als XML-Format vorliegen – in das XML-Format direkt transformiert (konvertiert). Somit wird es möglich, auf den referenzierten Artefakten werkzeugübergreifende Konformitätsprüfungen durchzuführen. Hierfür wurde eine Anzeigeumgebung (Viewer) implementiert (Abbildung 53) welche es erlaubt, das Datenmodell des Artefakts im XML-Format einzusehen und einen gefunden Fehler visuell im XML-Artefakt einzufärben.

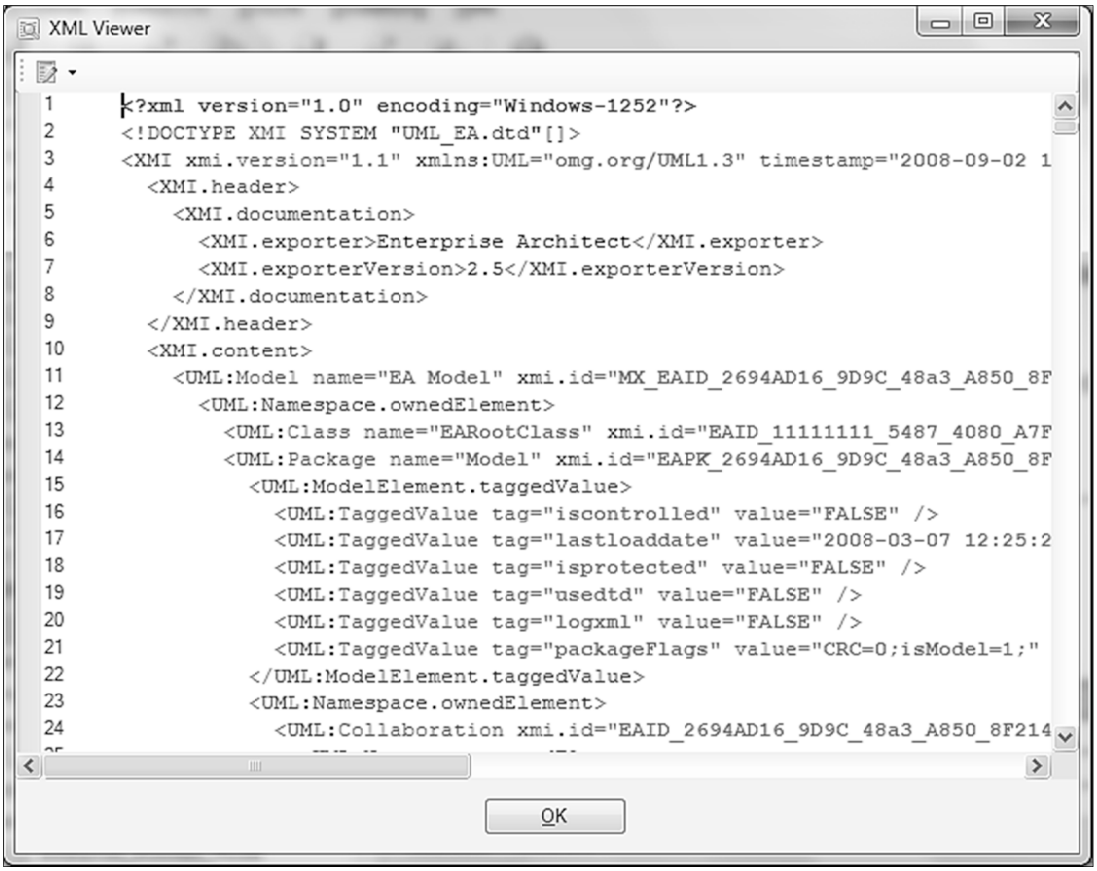


Abbildung 53: Datenmodell des logischen Artefakts in XML

Zum Nachweis der Machbarkeit wurden solche automatische Konvertierungen für kollaborative Artefakte implementiert: ein Rational DOORS-XML-Konverter und ein Microsoft Excel-XML-Konverter. Für andere, nicht im XML-Format vorliegende Artefakte, wurden XML-Konverter von Drittherstellern verwendet: SimEx [SIME09] für die MATLAB/Simulink-Konvertierung und Enterprise Architect [SPAX09] für die UML-Modellkonvertierung.

7.6.6 Prüfung und Auswertung

Durch die Kommandoschaltflächen der oberen Menüleiste kann die regelbasierte Konformitätsprüfung der im Prüfaufbau befindlichen kollaborativen Artefakte erfolgen. Der Regelkatalog wird durchlaufen und es wird pro selektierter Richtlinie, jeweils nach einer Prüfung, ein Prüfresultat in Form eines Prüfberichts zusammengestellt. Das Prüfresultat enthält die Information mit:

- *einem Bezeichner (ID) der geprüften Richtlinie,*
- *der Kategorie (Ordner des Regelkatalogs) der geprüften Richtlinie,*

- *einer Kurzbezeichnung (Titel) geprüften Richtlinie,*
- *einer optionalen Beschreibung, die ggf. im Regelalgorithmus hinterlegt wurde,*
- *sowie das Prüfergebnis, z. B. bei Nichtkonformität das Ergebnis „Fehlgeschlagen" samt einer vorher definierten Begründung und der bemängelten Elemente als Liste.*

Die nachfolgende Abbildung 54 zeigt solch ein Prüfergebnis zur Richtlinie der konformen Namenskonvention aus dem MAAB-Regelkatalog und die im Simulink-Modell bemängelten Modellelemente als dynamische Fehlerliste mit Referenz (*Uniform Resource Locator*, URL) auf das jeweilige Modellelement.

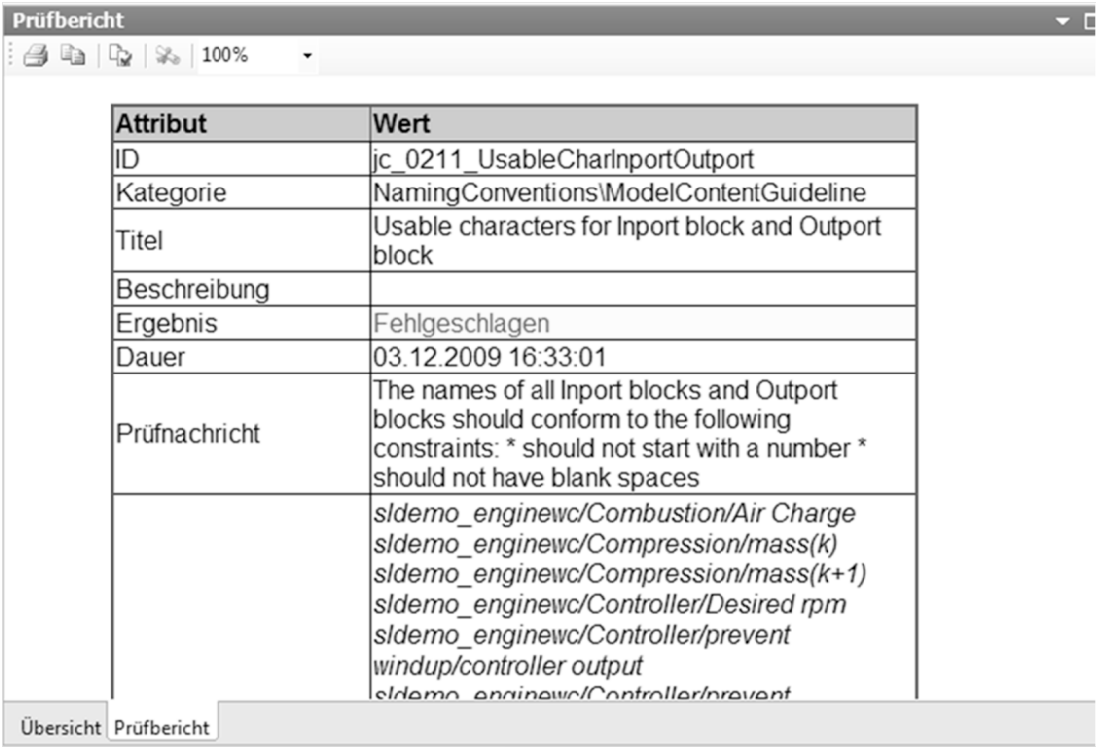


Abbildung 54: Prüfbericht pro Regel nach Prüfdurchlauf

Die Menge nicht konformer Modellelemente enthält den Namen und einen eindeutigen Pfad zum Modellelement. Mittels aktivem Hyperlink (Modellelement-Referenz) lässt sich jedes bemängelte Modellelement anhand der URL auch im Werkzeug selbst auffinden, wenn das Modell in der MATLAB-Umgebung geladen ist. Hierzu wird über die COM-Schnittstelle die Referenz auf das Modellelement übergeben und kann mittels MATLAB-Befehl (M-Skript) im geladenen Modell gefunden und sogar farblich markiert werden. Das Auffinden nicht konformer Modellelemente zeigt die Abbildung 55, in der das nicht konforme Element farblich (dunkle Einfärbung) automatisch markiert wurde.

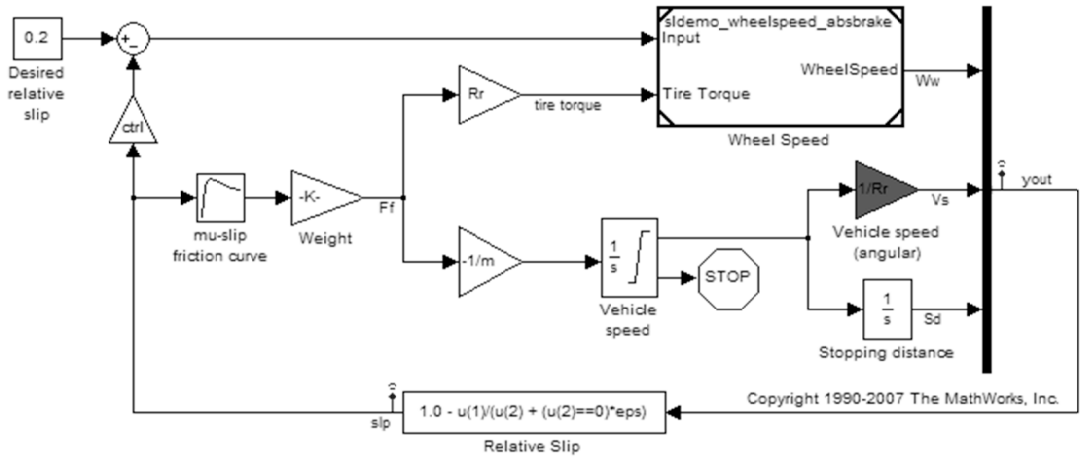


Abbildung 55: Auffinden nicht konformer Modellelemente im Artefakt

Das Beispiel zeigt das bereits bekannte Funktionsmodell eines Anti-Blockier-Systems aus den MATLAB-Beispielen [MLSL09], welches ein farblich eingefärbtes, nicht MAAB-konformes Element bzgl. der Namenskonvention aufzeigt. Der Modellierer erhält somit die Möglichkeit zum effizienten Auffinden des Mangels im Modell.

7.6.7 Ausnahmeregelungen

Erfahrungen in Anwendungsfällen haben gezeigt, dass zu jeder Richtlinie auch gewollte Ausnahmeregelungen für bestimmte Daten in einem kollaborativen Artefakt erwünscht sind. Die trifft im Falle eines Modells z. B. für verwendete Bibliotheksfunktionen (höherwertig zusammengesetzte Subsysteme) zu, die nicht in die Konformitätsaussage einfließen sollen.

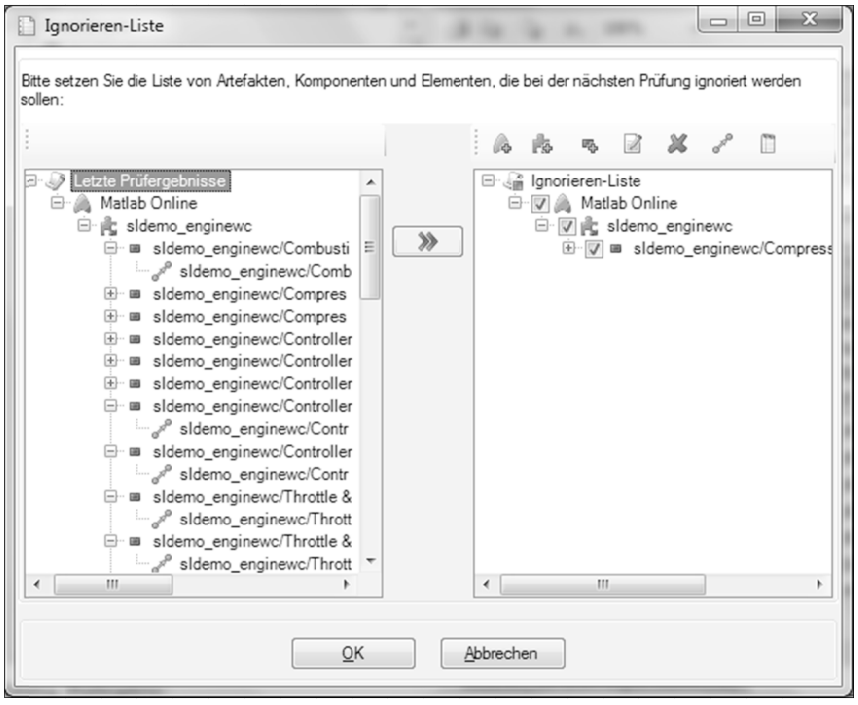


Abbildung 56: Festlegung von Ausnahmen

Aufgrund dieser zusätzlichen Anforderung wurde das Konzept des Regelkatalogs um eine *Ausnahmemenge* erweitert. In dieser Menge befinden sich vor einem Prüfdurchlauf festgelegte Modellelemente, die generell von jeder Konformitätsprüfung ausgeschlossen werden. Technisch werden die Modellelemente der Ausnahmemenge zwar durch den Prüfalgorithmus selbst gefunden und erfasst, jedoch nicht im Prüfergebnis zur Konformitätsaussage berücksichtigt.

Für den Prüfprozess selbst ist zudem von Bedeutung, dass jede Ausnahmeregelung auch dokumentiert und ggf. begründet wird. Hierzu wurde im Prüfwerkzeug *Assessment Studio* eine Lösung geschaffen, die es erlaubt, jedem Element der Ausnahmemenge eine natürlich sprachliche Begründung zu hinterlegen, die im Prüfergebnis zusätzlich erscheint.

Die Abbildung 56 zeigt im linken Fenster nach Prüfdurchlauf bemängelte Modellelemente in einem hierarchischen Baum. Der Anwender kann nun Elemente auswählen, die in die Ausnahmemenge aufgenommen werden sollen und der jeweiligen Richtlinie für alle folgenden Prüfungen dann fest zugeordnet sind. Zudem lassen sich mittels zusätzlichen Eingabefenstern (Aufruf durch die Kommandoschaltflächen) auf der rechten Seite (oben) selbst definierte Begründungen zur Ausnahme hinterlegen.

7.6.8 Automatische Korrektur

Aus den gewonnenen Erfahrungen mit der werkzeuggestützten, regelbasierten Konformitätsprüfung kollaborativer Artefakte resultierte eine weitere konzeptionelle sowie technische Herausforderung, die bemängelten Elemente aus einer Prüfung nicht nur automatisiert zu finden, sondern – falls möglich – auch automatisch korrigieren zu können. Konzeptionell steht dieser Herausforderung entgegen, dass eine Abfragesprache (Regel) kollaborative Artefakte auf dem logischen Artefakt zwar untersucht, nicht aber verändert. Dies ist ein wichtiges Merkmal der regelbasierten Konformitätsprüfung, da diese niemals ein Artefakt des Prüfaufbaus verändern darf. Eine Veränderung entspräche ja einer Verfälschung des Ergebnisses, da das kollaborative Artefakt durch die Regel verändert wäre.

Eine Korrektur entspricht einer geringfügigen Transformation eines ursprünglichen kollaborativen Artefakts aus dem Prüfaufbau. Das Konzept der automatisierten Korrektur muss daher

- aus dem logischen Artefakt eine *Relation* zum kollaborativen Artefakt herstellen,
- das ursprüngliche Artefakt mittels *Transformationsvorschrift* transformieren
- und schließlich im Prüfergebnis die *Korrekturmaßnahmen* wieder anzeigen.

Diese Anforderung kann prinzipiell so gelöst werden, dass der Regelkatalog um eine computerausführbare Transformationsvorschrift – also eine Transformationsregel – erweitert wird, die auf dem ursprünglichen kollaborativen Artefakt des Prüfaufbaus ausgeführt wird. Die Transformationsregel, selbst ein Algorithmus programmiert in einer Programmiersprache, erhält die Referenz vom Prüfergebnis (z. B. den Pfad zum Modellelement aus der Ergebnismenge) und die Referenz auf das kollaborative Artefakt aus dem Prüfaufbau. Technisch wird somit die Korrektur automatisiert, indem die Transformationsregel ausgeführt wird.

Die Möglichkeiten zur Transformation der kollaborativen Artefakte wurden bisher schon zur Bildung des logischen Artefakts im Kapitel 7.3.3 untersucht und können auch für die automatisierte Ausführung einer Transformationsregel dienen.

Die Autokorrektur wurde exemplarisch im Falle ML/SL/SF an verschiedenen Regeln implementiert, in dem das Konzept des Regelkatalogs um eine optional zu hinterlegende Transformationsregel erweitert wurde. Das folgende Beispiel zeigt eine Transformationsregel in M-Skript, welche die konforme Namenskonvention an Modellelementen vornimmt.

Beispiel: *Automatischer Korrektur-Algorithmus für Namenskonformität*

```
function [Result ResultDetail] = fixInOutportNames(system)
    global fixIoports;
    ResultDetail ={};
    for i = 1: length(fixIoports)
        iopname = get_param(fixIoports(i),'Name');
        validname = regexprep(iopname, '\W|[_]{2,}|^[\d_]|_$', '');
        set_param(fixIoports(i),'Name',validname);
        ResultDetail{end +1} = getfullname(fixIoports(i));
    end
   Result ='pass';
end.
```

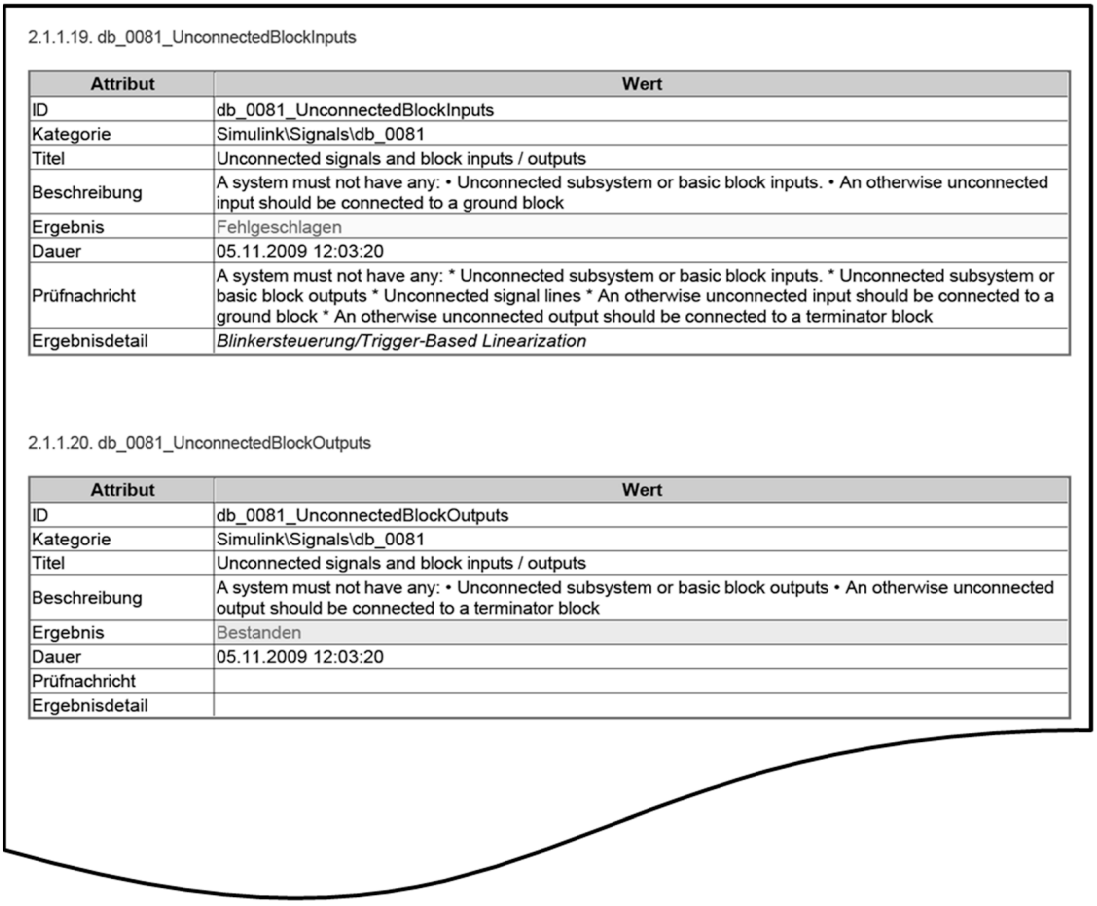
7.6.9 Auswertung

Für die Berichterstattung werden alle gewonnenen Teilergebnisse aus den regelbasierten Konformitätsprüfungen zusammengefasst und in einem Prüfbericht nachvollziehbar dokumentiert. Der Anwender erhält somit die Möglichkeit, einen schriftlichen Nachweis über die Konformität kollaborativer Artefakte zu erhalten. Dies ist insbesondere bei der Einhaltung höherer Reifegradstufen in Reifegradmodellen (wie CMMI oder SPiCE) notwendig, wo eine dokumentierte Berichtspflicht zur Befolgung gegebener Regeln gefordert ist und mit dem hier vorgestellten Verfahren automatisiert erfolgen kann.

Der Prüfbericht besteht aus den folgenden Abschnitten:

- *Titel und Datum mit Angabe der Uhrzeit und des Prüflings (Artefakts)*
- *Inhaltsverzeichnis aller Richtlinienprüfungen*
- *Tabellarische Statistik, Übersicht der Prüfergebnisse*
- *Grafische Statistik, Kreisdiagramm nach Prüfkategorie*
- *Detaillierte Prüfergebisse pro überprüfter Richtlinie*

Der Prüfbericht selbst ist eine im HTML-Format ausgegebene Auswertung, welche selbst durch eine XSLT-Transformation nach Kapitel 7.3.3 erzeugt wird. Hierdurch lässt sich der Bericht nach speziellen Formatierungsoptionen leicht durch ein Stylesheet (CSS) visuell anpassen. Das HTML-Format erlaubt den Ausdruck auf Papier sowie die Konvertierung in ein unveränderliches PDF-Format für Archivierungszwecke.



2.1.1.19. db_0081_UnconnectedBlockInputs

Attribut	Wert
ID	db_0081_UnconnectedBlockInputs
Kategorie	Simulink\Signals\db_0081
Titel	Unconnected signals and block inputs / outputs
Beschreibung	A system must not have any: • Unconnected subsystem or basic block inputs. • An otherwise unconnected input should be connected to a ground block
Ergebnis	Fehlgeschlagen
Dauer	05.11.2009 12:03:20
Prüfnachricht	A system must not have any: * Unconnected subsystem or basic block inputs. * Unconnected subsystem or basic block outputs * Unconnected signal lines * An otherwise unconnected input should be connected to a ground block * An otherwise unconnected output should be connected to a terminator block
Ergebnisdetail	*Blinkersteuerung/Trigger-Based Linearization*

2.1.1.20. db_0081_UnconnectedBlockOutputs

Attribut	Wert
ID	db_0081_UnconnectedBlockOutputs
Kategorie	Simulink\Signals\db_0081
Titel	Unconnected signals and block inputs / outputs
Beschreibung	A system must not have any: • Unconnected subsystem or basic block outputs • An otherwise unconnected output should be connected to a terminator block
Ergebnis	Bestanden
Dauer	05.11.2009 12:03:20
Prüfnachricht	
Ergebnisdetail	

Abbildung 57: Auswertung aller Konformitätsergebnisse im Prüfbericht (Auszug)

Werden Prüfergebnisse in einer Datenbank (Quality Repository) archiviert, können über eine Zeitspanne hinweg nachgelagerte Analysen der Prüfergebnisse Rückschlüsse auf Auffälligkeiten über einen Zeitraum zulassen. Wird beispielsweise festgestellt, dass die konforme Namenskonvention innerhalb der modellbasierten Entwicklung eingebetteter Systeme stets über einen längeren Zeitraum verletzt wurde, wobei die Bezeichnungen der Namen ursprünglich aus dem Anforderungsmanagement übernommen wurden, muss die Konformität aller kollaborativen Artefakte bereits in dem vorgelagerten Prozess (Anforderungsmanagement) gefordert werden. Es ist daher zu prüfen, ob nicht bereits im Anforderungsmanagement Fehler gemacht werden, die vorab bereits geprüft werden sollen. Die Konformitätsprüfung kollaborativer Artefakte hilft bei der Aufdeckung solcher Inhomogenität über Prozessgrenzen hinweg.

7.7 Untersuchung der Skalierbarkeit

In der Informatik ist bei jedem automatisiert durchführbaren Verfahren von Interesse, ob das gesamte System auch für eine hohe Last ausgelegt ist und sprichwörtlich „gut skaliert“. Nachfolgend wird daher eine Untersuchung des hier entwickelten Prüfsystems durchgeführt.

Unter dem Begriff *Skalierbarkeit* versteht man in der Informatik generell die Fähigkeit eines Systems, bestehend aus Software und/oder Hardware, sich wachsenden Anforderungen aus der realen Welt anzupassen. Die wachsenden Ansprüche adressieren somit das Verhalten von Programmausführung (Software) bzw. der Algorithmen bezüglich ihres Ressourcenbedarfs bei wachsenden Eingabemengen. Der ansteigende Ressourcenbedarf macht eine

Aussage über die zu erwartende Komplexität sowie die resultierende Ausführungszeit (Performance) des Systems. Für gewöhnlich spricht man von einer „guten Skalierbarkeit“ eines Systems, wenn der Ressourcenbedarf nur linear oder quadratisch mit den Eingabemengen (Last) anwächst, d. h., sich nahezu proportional zur erhöhten Systemlast verhält. Führt andernfalls ein exponentieller Anstieg des Ressourcenbedarfs bei linearer Erhöhung der Systemlast zu deutlichen Einbußen des Systemverhaltens, z. B. in der langen Ausführungszeit eines Algorithmus, spricht man von einem „schlecht skalierbaren System“. Droht mitunter ein Systemausfall oder eine nicht terminierende Programmausführung, „skaliert das System nicht“. Da der Ressourcenbedarf physikalisch bereits durch ein Rechnersystem (Hardware) begrenzt ist, wird in der Praxis durch Verteilung der Last (Load balancing) ein System auf mehrere Prozessoren oder entsprechend mehrere Rechnersysteme verteilt. Die Systemlast wird durch die Eingänge am Prüfsystem gesteuert. Diese kann demnach durch eine Vielzahl an Regeln oder eine Vielzahl an Elementen der kollaborativen Artefakte erzeugt werden.

Ein Ziel der Arbeit ist die Validierung des Ansatzes (Kapitel 8). Voraussetzung ist zunächst der Nachweis, dass das Prüfsystem im praktischen Anwendungsumfeld gut skaliert. Zur Bewertung der Skalierbarkeit des hier vorgestellten Ansatzes zur regelbasierten Konformitätsprüfung kollaborativer Artefakte wurden beide Prüfsysteme (*ASD-Regel-Checker* sowie *Assessment Studio*) hinsichtlich des Skalierbarkeitsverhaltens bei steigender Last experimentell untersucht. Nachfolgend werden die Skalierbarkeitsuntersuchungen zum *Assessment Studio* vorgestellt, die zum gleichen Resultat führten wie beim Prüfsystem *ASD-Regel-Checker*. Die Untersuchung geht dabei von den Umfängen realer Modelle und Regelwerke aus sowie einer üblichen Hardwareausstattung (Büro-PC), wie diese für gewöhnlich im Einsatz sind. Das Verhalten bei ungewöhnlichen großen Modellen oder überaus großen Regelwerken sowie bei Ausführung auf Hochleistungsrechnern wurde bei der nachfolgenden Untersuchung nicht berücksichtigt, da theoretisch konstruierte Extrema in diesem Fall wenig Aussagekraft bzgl. der praktischen Anwendbarkeit haben.

Die Länge der Zeitspanne, die zur Lösung einer Prüfung benötigt wird, lässt sich hierbei durch Ausprobieren bestimmen. Zur Bestimmung des Skalierbarkeitsverhaltens wurden zunächst fünf Versuchsreihen mit unterschiedlichen kollaborativen Artefakten angelegt, welche durch Transformation fünf verschieden umfangreiche logische Artefakte (System-Eingangsgröße ***A***) erzeugten.

Tabelle 22: Versuchsreihen - Logisches Artefakt

Versuchsreihe (Nummer)	***Logisches Artefakt-Strukturgröße (Σ Knoten)***	***Logisches Artefakt-Modellgröße (Σ Elemente)***	***Logisches Artefakt-Dateigröße (Kilobyte)***
1.	826,0	34,0	64,0
2.	4.538,0	236,0	383,0
3.	24.309,0	1.496,0	2.271,0
4.	55.348,0	3.795,0	5.277,0
5.	169.642,0	9.563,0	15.051,0

Da bei einer Konformitätsprüfung die gesamte Strukturgröße des logischen Artefakts die Ausführungsdauer des Prüfalgorithmus pro Regel beeinflusst, ergibt sich der Skalierungsfaktor des logischen Artefakts pro Versuchsreihe, wie aus Tabelle 23 folgt.

Tabelle 23: Skalierungsfaktor - Logisches Artefakt

Versuchsreihe (Nummer)	*Skalierungsfaktor*
1.	1,0-fach
2.	5,5-fach
3.	29,4-fach
4.	67,0-fach
5.	205,4-fach

In jeder Versuchsreihe wurden sechs unterschiedlich große Regelkataloge Ξ an dem logischen Artefakt angewandt:

$$\Xi = \{1, 5, 10, 15, 20, 52\}$$

Die verschiedenen Lasten wurden in fünf Versuchsreihen an das Prüfsystem angelegt, in dem die System-Eingangsgrößen ***A, B*** unabhängig voneinander und stetig erhöht werden. Der Ressourcenbedarf (Ausführungszeit und Speicherverbrauch) zur Ausführung der regelbasierten Prüfung wurde dabei gemessen, welcher für das Prüfergebnis (System-Ausgangsgröße ***Y***) samt Auswertung und Protokollierung aufgebracht werden musste.

Untersuchung:

Die Untersuchungen basieren auf der Annahme, dass alle Regeln (Prüfalgorithmen) den gleichen Ressourcenbedarf im Mittel benötigen (Average-Case-Laufzeit) und jeweils bzgl. ihrer Ausführung terminieren. Folgende drei Untersuchungen wurden durchgeführt:

a) Auf einem Artefakt werden ***n***-Regeln ausgeführt.

b) Auf ***m***-Artefakten wird *eine* Regel ausgeführt.

c) Auf ***m***-Artefakten werden ***n***-Regeln ausgeführt.

Im ersten Fall wird jeweils ein Regel-Tupel aus dem Regelkatalog Ξ (MISRA-Regelsatz) für die Analyse ausgewählt sowie gleichbleibend große logische Artefakte (ASCET-Modell, Glossar und Datei für Vorgabewerte) für die übergreifende Konformitätsprüfung gewählt. Im zweiten Fall wird das logische Artefakt (ASCET-Modell) bezüglich seines Umfangs skaliert und ein gleichbleibend großes Regel-Tupel aus dem Regelkatalog Ξ ausgewählt. Der dritte Fall kombiniert die ersten beiden Fälle. Für die regelbasierte Konformitätsprüfung auf dem logischen Artefakt ist es gleichbedeutend, ob das logische Artefakt aus ***m***-Artefakt-Quellen transformiert wurde oder durch ein einzelnes, anwachsendes logisches Artefakt repräsentiert wird. Die Modellelemente des ASCET-Modells wurden mit einem Skalierungsfaktor aus Tabelle 23 skaliert. Die Versuchsreihen stellen reale Szenarien aus der Praxis dar, sodass konstruierte Worst-Case-Szenarien (z. B. alle Modellelemente verstoßen gegen eine Regel) nicht in der Untersuchung enthalten sind. Die Messungen der Prüfzeit sind als Funktion über die Regel-Tupel jeweils pro Versuchsreihe grafisch in Abbildung 58 als Punkte abgetragen. Eine eingefügte Trendlinie pro Versuchsreihe veranschaulicht das Verhalten.

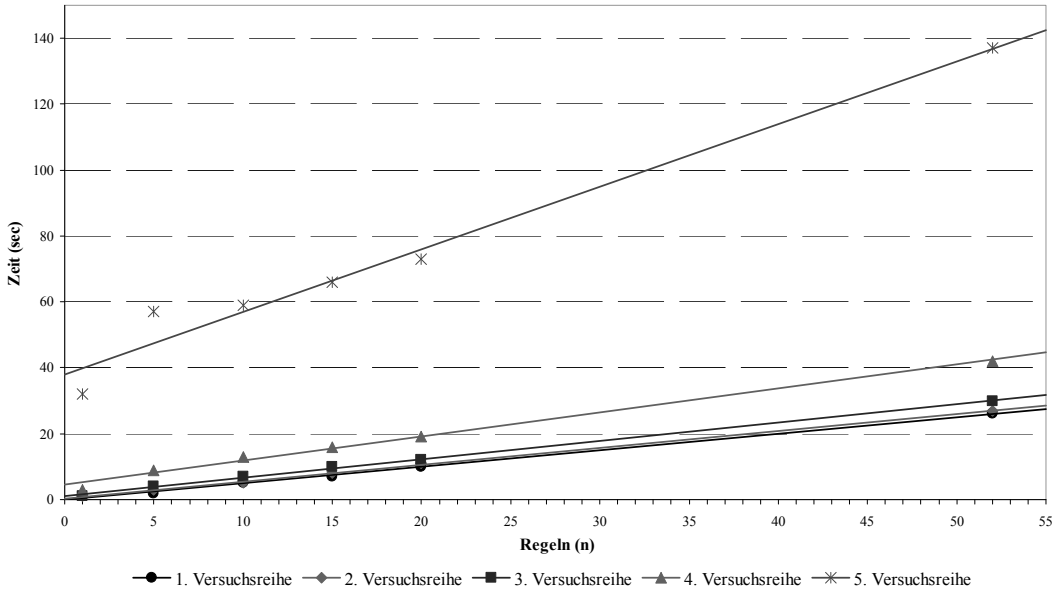


Abbildung 58: Prüfaufwand pro Regelkatalog und Artefaktgröße

Die Messungen wurden mit einem gewöhnlich ausgestatteten Bürorechner durchgeführt, der folgende Merkmale besaß: AMD Athlon 64 X2 5000+ 2.6 GHz Dual-Core Socket CPU, 1000 MHz Bus-Speed, 1 MByte Cache, 2 GByte RAM, OS Windows XP 32-Bit.

Die Messungen spiegeln folgendes Verhalten wieder:

Der Umfang des gewählten Regelkatalogs sowie der Umfang der logischen Artefakte verhalten sich in diesen Versuchsaufbauten proportional zueinander. In jeder Versuchsreihe wurde ein linearer Anstieg von Ressourcenbedarf, Ausführungszeit und auch Speicherverbrauch (nicht dargestellt) bei Erhöhung der Systemlast gemessen. Das *lineare Wachstum* der Laufzeit als dauer der Ausführung einer Prüfung ist somit $\boldsymbol{O}(\boldsymbol{n})$ (Landau-Notation). Die geringen Abweichungen der fünften Versuchsreihe sind im komplexeren Aufbau (Strukturkomplexität) des größeren Artefakts begründet. Die nicht gleichartige Strukturveränderung eines Modells fällt ggf. optimaler oder aber schlechter für einen Regelalgorithmus aus.

Betrachtung von Sonderfällen:

Werden für eine Prüfung zwei oder mehrere logische Artefakte notwendig, können höhere Laufzeiten auftreten. Zur Betrachtung sei folgendes Beispiel gewählt: Eine MISRA-Regel prüft, ob im Modell alle verwendeten Variablen einen gültigen Datentyp zur Codegenerierung aufweisen. Das eine logische Artefakt $\boldsymbol{A}$ enthält das Modell (Variable als Elemente), das zweite logische Artefakt $\boldsymbol{B}$ eine Auflistung gültiger Datentypen (Datentypen als Elemente). Im normalen Fall ist die Kardinalität der Mengen $|\boldsymbol{A}| >> |\boldsymbol{B}|$, da gültige Datentypen eine sehr kleine Anzahl an Elementen im Vergleich zu den Variablen im Modell darstellen. Ein Prüfalgorithmus, welcher bei jedem Element aus $\boldsymbol{A}$ eine gültige Entsprechung in $\boldsymbol{B}$ sucht, wird in diesem Fall auch eine lineare Prüfdauer mit $\boldsymbol{O}(\boldsymbol{n})$ aufweisen, wie die Abbildung 58 zeigt. Ein Sonderfall liegt jedoch bei $|\boldsymbol{A}| <= |\boldsymbol{B}|$ vor, z. B. wenn der Prüfalgorithmus im Mittel immer als gültige Entsprechung das letzte Element aus $\boldsymbol{B}$ findet oder ein Prüfalgorithmus sogar rekursiv programmiert wurde (z. B. eine Prüfung erfordert einen Sortieralgorithmus). In diesen Fällen ist ein nahezu *quadratischer Anstieg* der Prüfzeit (Worst-Case-Laufzeit) möglich und die obere Schranke zur Laufzeit $\boldsymbol{O}(\boldsymbol{n}^2)$.

Das Laufzeitverhalten wurde für beide implementierten Prüfsysteme *ASD-Regel-Checker* (hier nicht gezeigt) sowie *Assessment Studio* gleichermaßen beobachtet, obwohl beim *Assessment Studio* die Prüfgeschwindigkeit aufgrund anderer Technologie deutlich höher ist.

Das gemessene Verhalten an den gewählten Beispielen lässt folgenden Schluss zu:

Ergebnis: „Die regelbasierte Konformitätsprüfung kollaborativer Artefakte *skaliert* für die beiden umgesetzten Prüfsysteme *gut*."

Der Speicherverbrauch beeinflusst die Prüfzeit nicht direkt und ist daher in der Grafik (Abbildung 58) nicht gezeigt. Für den Ressourcenanstieg ist es jedoch trotzdem von Interesse, wie sich der Speicherverbrauch verhält.

Der Speicherbedarf ϑ ist abhängig von dem Speicherbedarf des Prüfsystems S des im Speicher geladenen logischen Artefakts A und dem allozierten Speicherplatz a_n eines Prüfalgorithmus bei Ausführung pro Regel (verwendete Variablen, Datentypen usw.).

Nachfolgende Formel ergibt den gesamten Speicherverbrauch für einen Prüfdurchlauf:

$$\vartheta = S + A + \sum_{n=1}^{\infty} a_n$$

Durch die Linearität skaliert folglich auch der Speicherverbrauch gut. Um einen Performancegewinn bei umfangreichen Berechnungen oder großen Mengen von kollaborativen Artefakten erreichen zu können, kann die Prüfausführung mittels Lastverteilung auf mehrere parallel arbeitende Systeme (Prozessoren oder Rechner) verlagert werden. Durch Aufteilung des Regelkatalogs kann die Konformitätsprüfung auf einem eigenen Prozessor ausgeführt werden und somit schneller ein Zwischenergebnis ermitteln.

Die Parallelisierung ist auch ein Thema von *Parallel Language Integrated Query* (P-LINQ), weiterführend beschrieben in ‚*Running Queries On Multi-Core Processors*' nach [DUFF07] und wird Bestandteil zukünftiger .NET-Framework-Versionen sein.

8 Fallbeispiele im modellbasierten Systementwurf des APR

Zur Evaluierung des in dieser Arbeit entwickelten Ansatzes (Kapitel 4-6) wird in den nachfolgenden Unterkapiteln ein praxisnahes Entwicklungsszenario aus dem industrienahen Forschungsprojekt ‚*AAES – Automotive Application Evaluation System*' im modellbasierten Systementwurf einer sicherheitskritischen Fahrzeugfunktion beschrieben, welches während dieser Arbeit am Fraunhofer Institut für offene Kommunikationssysteme (FOKUS) in Zusammenarbeit mit dem Steuergerätehersteller iSYS RTS GmbH durchgeführt wurde. Abgeleitet aus der durch die QRDL eingeführten Strukturierung der Regeln werden zudem jeweils Implementierungsbeispiele von Regeln in den Programmiersprachen LINQ, M-Skript und OCL in diesem Kapitel vorgestellt. In dem Anwendungsbeispiel sollen eine softwaregesteuerte Sicherheitsfunktion mit der Bezeichnung ‚*Active Passenger Rescue*'-Funktion (kurz: APR) entworfen und schließlich die entstehenden kollaborativen Artefakte auf geltende Konformitätsregeln durch ausführbare Regeln geprüft werden.

8.1 Systementwurf des APR-Systems

Das APR ist eine steuergeräteübergreifende Fahrzeugfunktion zum Insassenschutz und soll zur funktionalen Sicherheit von Personen beitragen. Das APR soll im Falle eines Unfalls (Crash) den Insassenschutz dabei aktiv unterstützen, indem

a) die Fenster im Fahrzeug automatisch geöffnet werden,

b) eine Überspannung im Bordnetz des Fahrzeuges bei Unfall verhindert wird

c) sowie ein sehr helles Bremslicht den Unfall dem Verkehr visualisiert.

Die Funktion (a) ermöglicht, dass durch die Explosion der Airbags hervorgerufenes Gas und Feinstaub aus dem Fahrzeug-Innenraum ausweichen können sowie den Insassen eine bessere Flucht- bzw. Bergungsmöglichkeit nach einem Unfall geboten wird.

Die Funktion (b) verhindert, dass die Insassen nicht durch Überspannung oder Kurzschlüsse weiteren Risiken bzw. Fehlfunktionen anderer eingebetteter Systeme ausgesetzt werden.

Schließlich unterstützt (c) die Risiko-Minimierung von Auffahrunfällen, indem ein sehr grelles Rücklicht auffahrende Fahrzeuge visuell warnt bzw. die Unfallstelle anderen Verkehrsteilnehmern kenntlich macht.

Die Fahrzeugfunktion APR integriert drei Einzelfunktionen eines Fahrzeuges im Crash-Fall und wird durch eine Verbund-Applikation von drei eingebetteten Systemen realisiert:

- ‚Fenster öffnen' durch das Fenstersteuergerät (SGFH)
- ‚Abtrennen der Fahrzeugbatterie' durch das Batteriemanagement-Steuergerät (SGB)
- ‚Helles Bremslicht' durch das Steuergerät für Lichtsteuerung (SGL)

Wird auf dem CAN-Bus (beschrieben in [ETSB02]) die Nachricht des Airbag-Auslösens von den drei Steuergeräten erhalten, wird das Fenster geöffnet, die Fahrzeugbatterie abgetrennt sowie anschließend die Batterie vom Bordnetz entkoppelt. Die drei ECUs sind daher jeweils durch CAN-Busleitungen verbunden. Die Schnittstellen bestehen somit aus:

- der 12V-Spannungsversorgung der Steuergeräte,
- der 12V-Leistungsversorgung für die Fensterheber-Motoren.

Nachfolgend werden die einzelnen Steuergeräte SGFH, SGB und SGL der iSYS RTS GmbH und die verbundene Sensorik und Aktorik aus dem Forschungsprojekt kurz eingeführt, bevor der modellbasierte Systementwurf und die darin existenten kollaborativen Artefakte im Kapitel 8.2 sowie angewendete Richtlinien im Kapitel 8.3 vorgestellt werden.

8.1.1 Fenstersteuergerät (SGFH)

Elektrische Fensterheber werden durch ein zentrales Steuergerät oder durch vier verteilte Steuergeräte in modernen Fahrzeugen in Serie verbaut. In [BAU02] wird die Autoelektrik weiterführend erläutert. Die Hauptaufgabe der Systemfunktion liegt in der automatischen Steuerung der Fensterstellung im Fahrzeug. Neben den Basis-Steuerungen sind diese heutzutage weitaus komplexer, da viele Randbedingungen (Einklemmschutz u. a.) und weitere Komfortfunktionen (Keyless Entry u. a.) heutzutage durch Softwarefunktionen implementiert werden. Die elektrischen Fensterheber bestehen aus Sensorik (Schalter), Aktorik (Elektromotor), Mechanik (Gestänge) sowie dem Steuergerät (ECU), wie in der Abbildung 59 der Brettaufbau (Breadboard) zeigt.

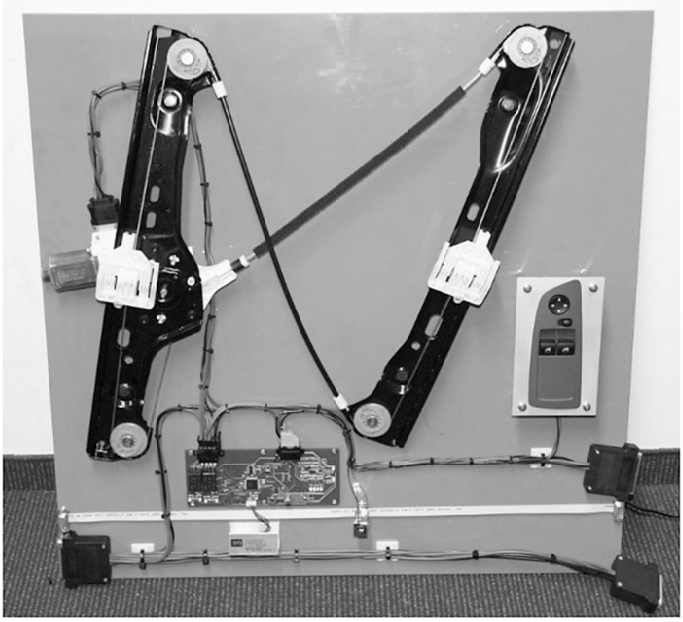


Abbildung 59: Fensterheber - Sensorik, Aktoren und Steuergerät

Die meisten elektrischen Fensterheber arbeiten mit Seilzügen. Angetrieben werden diese durch einen Elektromotor mit integrierten Hall-Sensoren, die über ein Stirnradgetriebe eine Seiltrommel steuern. An der Seiltrommel sind die beiden Enden eines Stahlseils so befestigt, dass bei Drehung das eine Ende auf- und das andere Ende abgewickelt wird. Das Zugseil zieht über einen Bowdenzug und Umlenkrollen auf zwei Führungsschienen laufende Fensterbefestigungen. Das Seilende ohne Zug wird wieder auf die Seiltrommel aufgewickelt.

Die Enden der beiden Bowdenzug-Mäntel sind an dem Gehäuse des Antriebs federnd gehalten, um eine gewisse Nachgiebigkeit in das System zu bringen.

8.1.2 Batteriemanagement-Steuergerät (SGB)

Das Batteriemanagement-Steuergerät (SGB) ist an die Fahrzeugbatterie angeschlossen und wertet gemessene Eingangssignale der Batteriespannung aus. Zudem steuert es das Trennrelais gemäß eingestellter Algorithmen (z. B. Spannungsschwellwert) an. Weiterhin ist die Kommunikation mit anderen Steuergeräten über CAN-Bus möglich. Die Abbildung 60 zeigt Sensorik, Aktorik sowie das SGB (Steuergerät).

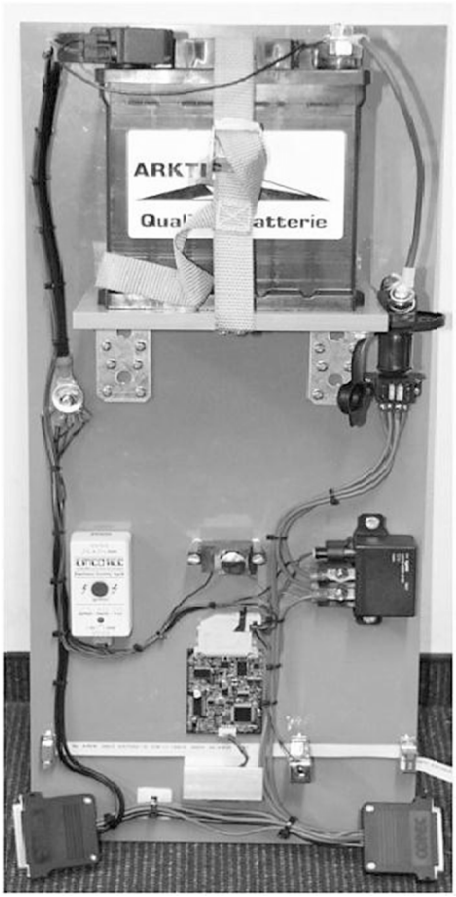


Abbildung 60: Batteriemanagement - Sensorik, Aktorik und Steuergerät

Das verwendete Trennrelais dient im Fahrzeug dazu, eine Batterie physikalisch abzutrennen oder zwei Bordbatterien zu 24 Volt zu verbinden. Aufgrund der großen Stromstärken kommt ein Hochstromrelais für 150 A zum Einsatz. Im Demonstrationsaufbau ist nur eine Batterie vorhanden. Die zweite Batterie kann durch ein geeignetes externes Netzteil simuliert werden.

Die verwendete 12V-Fahrzeugbatterie dient der Versorgung des gesamten Aufbaus, sofern der kombinierte Betrieb des Batteriesteuerungs-ECU mit einem oder beiden anderen Modulen erfolgt. Die Daten der Batterie werden durch Sensorik erfasst und vom SGB ausgewertet. Da die echte Zündpille der Batterie-Anschlussleitung nur einmalig und irreversibel ausgelöst werden kann, ist diese im Versuchsaufbau nicht angeschlossen. Stattdessen wird eine rücksetzbare Zündpillen-Simulation des Herstellers ‚Emcotec‘

eingesetzt. Diese signalisiert das Auslösen durch den roten Leuchtmelder und kann danach wieder in den Ursprungszustand rückgesetzt werden (Reset).

8.1.3 Steuergerät für Lichtsteuerung (SGL)

Das Steuergerät für Lichtsteuerung (SGL) hat eine Einheit zur Auswertung der Sensorik für ein einstellbares Bedienelement und steuert die Aktorik (Bremsrückleuchten) durch zwei LED-Ausgangskanale an, wie in Abbildung 61 dargestellt.

Abbildung 61: Lichtsteuerung - Sensorik, Aktorik und Steuergerät

Als Lichteinheit kommt ein Rücklicht-Modul des BMW E63 (6er Baureihe) zum Einsatz. Dieses hat eine Rückleuchte (TAIL) und eine Bremsleuchte (STOP) integriert, ebenso enthält sie eine hellere Bremsleuchte (*Brake Force Detection*, BFD) zur Signalisierung starker Bremsvorgänge (Notbremsungen). Das Steuergerät kann zwei Kanäle ansteuern, hierzu sind Rücklicht und BFD mit dem SGL verbunden. Die dunklere, normale Bremsleuchte wird nicht angeschlossen. Diese Funktion kann durch reduzierte BFD-Ansteuerung durch das SG nachgebildet werden. Das Bedienpanel mit Taster und Potenziometer liefert die Signale für die Helligkeits-Steuerung an das Lichtsteuergerät. Sowohl Potenziometer als auch der Taster sind am SGL angeschlossen und können per Software ausgewertet werden.

8.2 Kollaborative Artefakte der APR-Systemfunktion

Im modellbasierten Systementwurf des APR sind durch die im V-Modell bekannten Prozessphasen der Anforderungsdefinition, des Architektur-Designs, der funktionalen Spezifikation, der modellbasierten Modulspezifikation sowie der Testspezifikation kollaborative Artefakte entstanden, welche in den nachfolgenden Kapiteln vorgestellt werden. Durch die steuergeräteübergreifende Funktionalität sind die Artefakte inhaltlich sowie auch prozesslogisch in Abhängigkeit zueinander. Das Kriterium aus ***Definition 4.5*** (Kapitel 4.2.2) ist

demnach in beiden Merkmalen erfüllt. Zusätzlich zu den während der APR-Entwicklung erzeugten kollaborativen Artefakten werden regelbasierte Konformitätsprüfungen in dem folgenden Kapitel 8.3 exemplarisch vorgestellt und an den Artefakten evaluiert.

8.2.1 Artefakte der Anforderungsdefinition

Im APR sind die Anforderungen für die Systementwickler in den ‚Systemanforderungen' festgelegt, während die vom Kunden gewünschten Merkmale in den ‚Kundenanforderungen' beschrieben sind. Die Systemanforderungen beschreiben, was das APR können soll, d. h., welche Funktionalität von den Entwicklern umgesetzt werden soll. Im Unterschied zu Kundenanforderungen sind hier nur technische Spezifikationen enthalten. Beide Artefakte, System- und Kundenanforderungen, referenzieren sich logisch. Die Spezifikation des gesamten APR-Systems ist verbal in der Kundenanforderungsdokumentation erfasst. Die Abbildung 62 zeigt einen Ausschnitt aus den Kundenanforderungen für das Fensterhebersteuergerät (SFH) des APR.

ID	Anforderung	Systembezeichnung	Autor	Status
2	Fenstersteuergerät (SGFH)	SGFH	TFA	Imlementiert
2.1.1	**Eingänge Fahrzeug-Klemmen**	Klemme 30		
1.2.1.1	Das SGFH ist an Klemme 30 angeschlossen. Eine Absicherung gegen Verpolung, Überspannung und Störungen ist vorgesehen.			
1.2.2	**Spannungsmessung KL30**	SGFH_Modul		
1.2.2.1	Mittels Spannungsteiler wird die aktuelle Bordnetzspannung der Klemme auf einen Spannungswert gebracht, der vom A/D-Wandler des µC gemessen werden kann.			
1.2.3	**CAN Transceiver**	CAN_Transceiver		
	Die Anbindung an einen Fahrzeugbus erfolgt mittels CAN. Dabei wird der Fault Tolerant CAN-Transceiver TJA1054 der Firma Philips verwendet, sowie zwei Abschlußwiderstände von je 820Ω.			
1.2.3	**Diskrete Hallelementauswertung**	SGFH_Modul		
1.2.3.1	Durch ein SGFH werden insgesamt vier Hallelemente ausgewertet.			
1.2.3.2	Pro Motor sind zwei zweipolige Hallelemente verbaut. Sie stellen Kanal A und B dar.			
1.2.3.2	Mittels einer Komparator Schaltung wird der durch die Hallelemente ingeprägte Strom mit einem Schwellwert verglichen. Aus den so erzeugten Rechtecksignalen kann die Drehrichtung bestimmt werden, da sie je nach Richtung einen Phasenversatz von nominell +90° oder -			
1.3	**Ansteuerung und Auswertung FH-Relais**			
1.3.1	Es werden zwei FH-Motoren angesteuert und ausgewertet. Zu den auszuwertenden Signalen gehören der Strom durch den jeweiligen Motor und die Schaltzustände des Doppelrelais. Zudem ist die Möglichkeit gegeben, den Motor mittels PWM auch „langsam" anfahren zu können. Die Ansteuerung der Relais erfolgt mittels Doppeltransistoren. Die Auswertung des Motorstroms wird über einen Strommesswiderstand vorgenommen. Die Spannungen an den Relais-Ausgängen werden als Digitalsignal ausgewertet. Die PWM			

Abbildung 62: Kundenanforderungen des APR in MS Excel (Auszug)

Die Abbildung 63 wiederum zeigt einen Ausschnitt aus den Systemanforderungen für das Fensterhebersteuergerät (SFH) des APR, welche sich aus den Kundenanforderungen ableiten und ggf. für die Systementwicklung erweitert und mit Prozessinformationen annotiert werden. In diesem Szenario wird demnach Systemfunktionalität des APR einmal im

Werkzeug Microsoft Excel [MSOF07] und einmal in IBM Rational DOORS [DOOR09] beschrieben.

Es entstehen kollaborative Artefakte aus dem jeweiligen Werkzeug. Anschließend wird durch eine werkzeuggestützte Modelltransformation jedes kollaborative Artefakt in ein logisches Artefakt (XML-basiert nach OOXML und RIF) überführt.

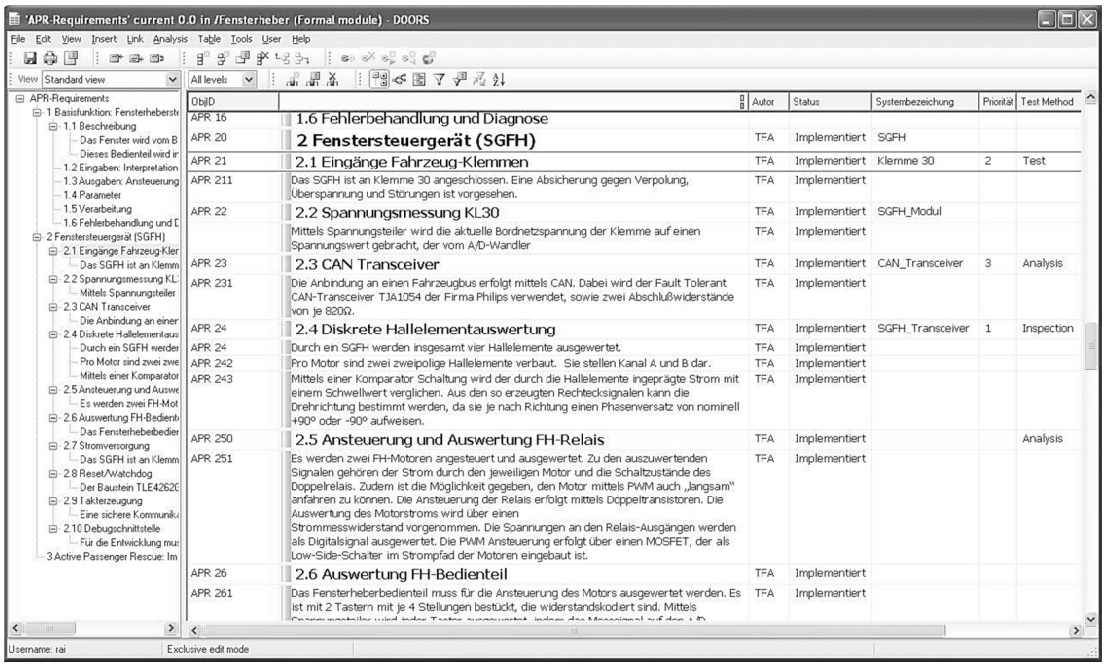


Abbildung 63: Systemanforderung des APR in DOORS (Auszug)

8.2.2 E/E-Systemarchitektur

Die modellbasierte E/E-Systemarchitektur des APR beschreibt architekturrelevante Aspekte des Systems. Es muss ein funktionierendes Gesamtsystem mit allen beteiligten Komponenten beschrieben bzw. modelliert werden (Anhang – A zeigt eine Vergrößerung).

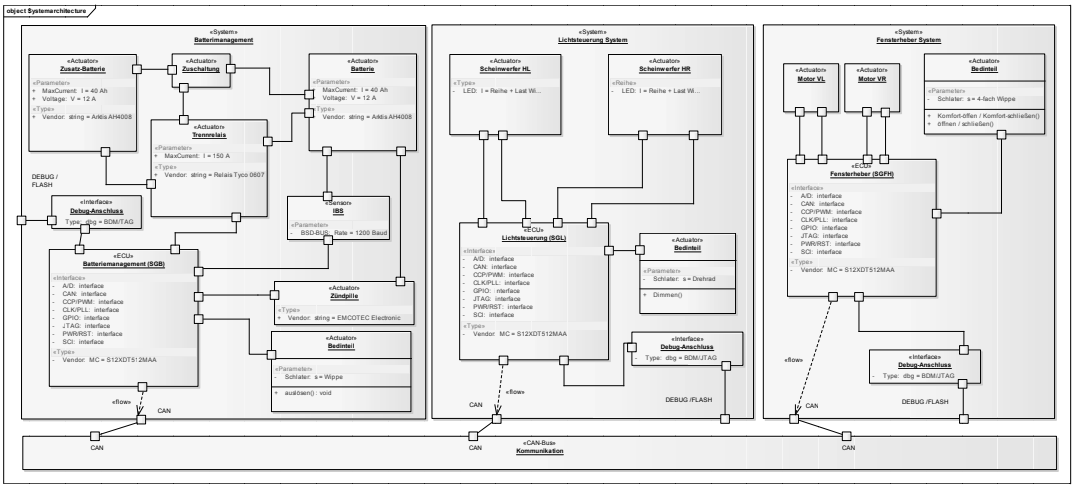


Abbildung 64: E/E-Systemarchitektur des APR im EA

Architekturmodelle sind hierarchisch strukturierte Netzwerke aus Systemobjekten, Funktionsblöcken mit Attributen und Schnittstellen (Ports), in denen die ein- und ausgehenden Daten und Operationsaufrufe spezifiziert sind. In der Abbildung 64 ist ein Ausschnitt des kollaborativen Artefakts gezeigt. Im Anhang (Kapitel 14.1) sind die jeweiligen Komponenten der Architektur noch einmal vergrößert dargestellt.

Die E/E-Systemarchitektur wird durch ein UML-Objekt-Diagramm formal strukturiert und bildet das kollaborative Artefakt. Objektdiagramme sind Teil der UML-Klassen-Diagramme, werden jedoch wie eine eigenständige Notation verwendet. Von Klassen können beliebig viele Objekte als Instanzen gebildet werden. Links stellen Verbindungen zwischen Objekten über Ports dar, die entsprechend der Assoziationen und objektwertigen Attribute ihrer Klassen gebildet werden. Eine Subklassenstruktur der Teile-Hierarchie wird im Objektdiagramm nicht erkennbar, denn die abstrakte Klasse Teil wird nicht instanziiert. Die einzelnen Subsysteme des APR sind durch `<<System>>` stereotypisiert.

Die E/E-Systemarchitektur für das APR wurde im Modellierungswerkzeug Enterprise Architect [SPAX09] (EA) modelliert und anschließend durch Modelltransformation in ein XMI-basiertes (Version 1.1) logisches Artefakt (durch den EA-Export) überführt.

8.2.3 Funktionale Spezifikation

Nachdem die E/E-Systemarchitektur des APR durch die kollaborativen Artefakte (textuelle Anforderungen und Architekturmodell) hinreichend beschrieben ist, müssen in der nächsten Prozessphase des modellbasierten Entwicklungsprozesses die Funktionalität der Hardware- und Softwarekomponenten der einzelnen Steuergeräte durch ein Funktionsmodell abgebildet werden (funktionale Spezifikation).

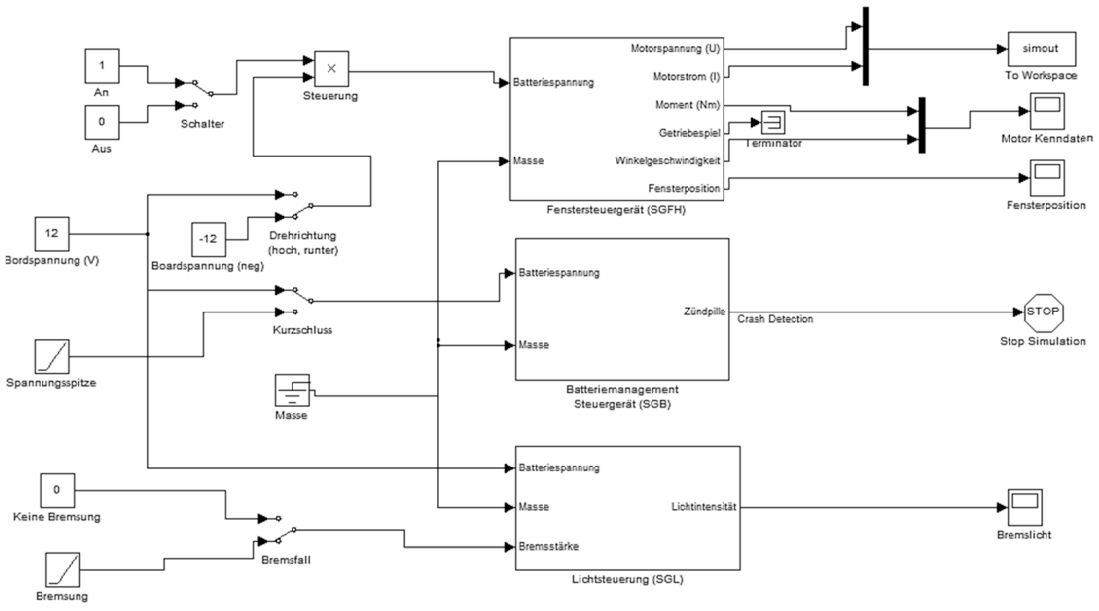


Abbildung 65: APR-Funktionsmodell in ML/SL/SF

Die Abbildung 65 zeigt die funktionale Spezifikation der physikalischen Eigenschaften mittels regelungstechnischen Systemverhaltens als Simulink-Modell in ML/SL/SF [MLSL09]. Im Simulink-Modell sind die jeweiligen APR-Komponenten als die Subsysteme Lichtsteuerung (SGL), Batteriemanagement Steuergerät (SGB) und Fenstersteuergerät

(SGFH) (Funktionsblöcke) aufgeführt. Das interne Verhalten ist im Subsystem-Block Fenstersteuergerät (SGFH) modelliert und in der Abbildung 66 dargestellt.

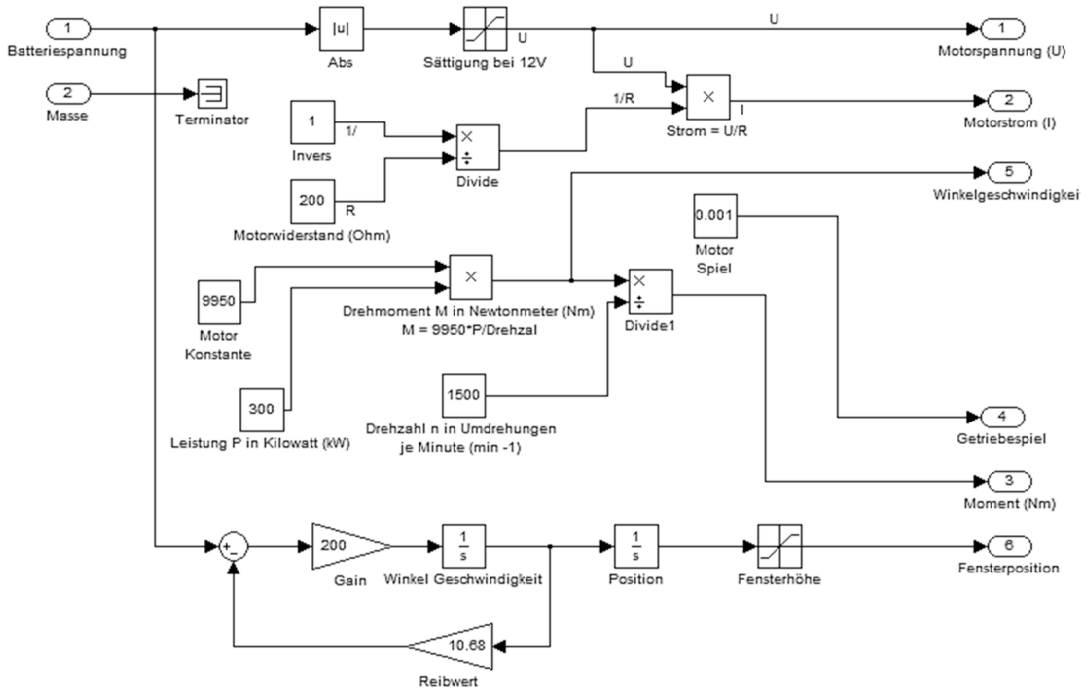


Abbildung 66: Verhaltensmodell (APR-Fensterhebermotor) in ML/SL/SF

Relevantes Verhalten zur mathematischen Beschreibung der Fenster-Position sowie typische Kenngrößen des Motors wie Motorspannung ***U***, Motorstrom ***I***, Winkelgeschwindigkeit ***w***, Getriebespiel ***s*** und das auftretende Moment ***M*** sind modelliert. Als Beschreibungsmittel für die Systemzustände werden Zustandsdiagramme verwendet.

Das Zustandsdiagramm in der Abbildung 67 beschreibt Zustände (als abgerundete Rechtecke dargestellt) der Systeme des APR im Modellierungswerkzeug Enterprise Architect.

Neben den Systemzuständen sind Transitionen (als Pfeile dargestellt) im Allgemeinen mit einem Ereignis assoziiert. Zusätzlich kann eine Bedingung (Condition) angegeben werden, unter der die Transition erfolgen kann. Über die gezeigten Elemente hinaus kann jede Transition mit einer Aktion verbunden sein, die bei der Transition ausgeführt wird.

Zusätzlich können Zustände mit Entry-, Exit- und Do-Aktionen versehen werden, die beim Betreten und Verlassen eines Zustands bzw. während des Verweilens in einem Zustand zusätzliche Funktionalität erbringen. Zum Zeitpunkt des modellbasierten Entwurfs ist noch nicht festgelegt, in welcher Form das Zustandsdiagramm (State-Chart) von den einzelnen Systemen des APR umgesetzt wird.

Das angegebene Zustandsdiagramm abstrahiert außerdem von vielen Einzelschritten, indem es diese im Einzelzustand ‚Fenster öffnen' bündelt. Einzelne Module werden auf einer feineren Granularitätsstufe in der folgenden Modulspezifikation näher spezifiziert.

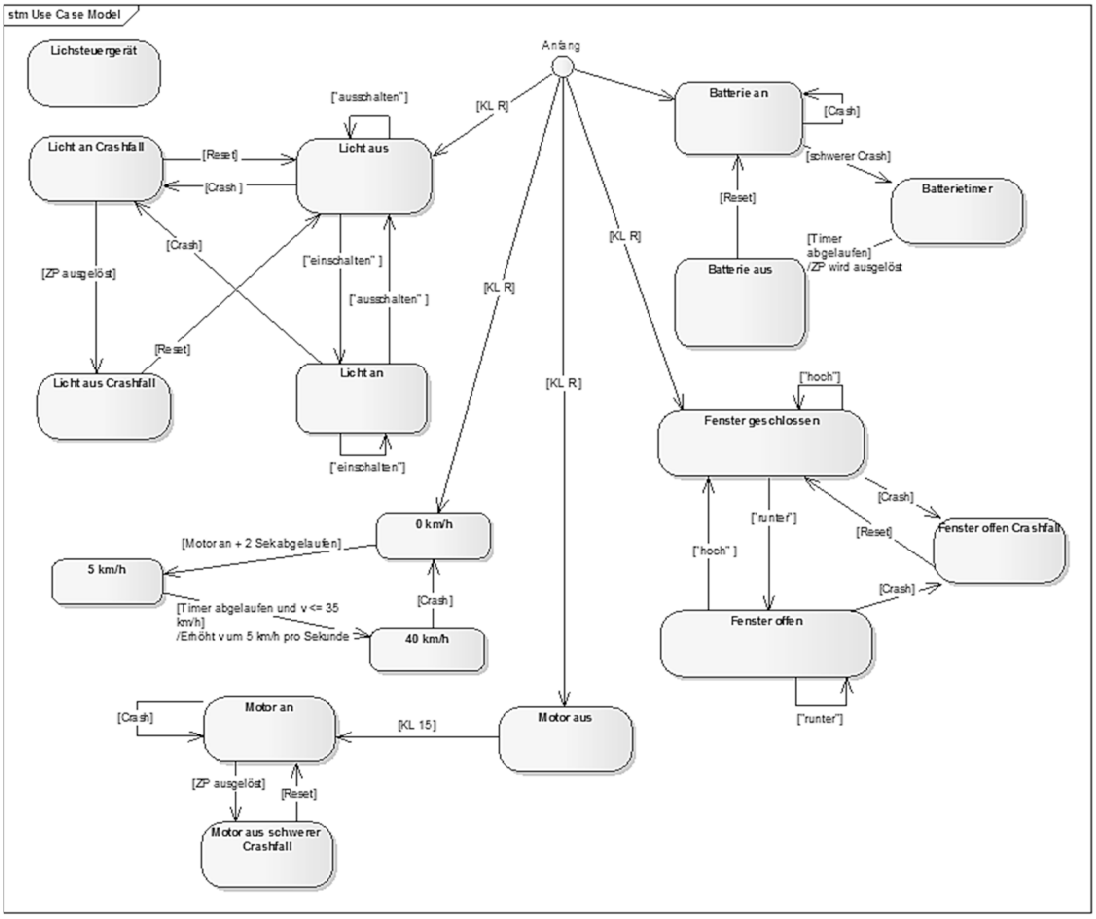


Abbildung 67: APR-Systemzustände (UML State-Chart) im EA

In der Phase der funktionalen Spezifikation entstehen somit kollaborative Artefakte (z. B. das Funktionsmodell und das Verhaltensmodell) der jeweiligen Werkzeuge, welche systemlogisch und prozesslogisch miteinander in Beziehung stehen. Beide Modelle werden als XML-basierte logische Artefakte durch Modellkonverter gespeichert.

8.2.4 Modulspezifikation

Auf der Ebene der Modulspezifikation werden die speziell auf den im APR verwendeten Freescale-Mikroprozessor und die entsprechende Speicherausstattung hinoptimierte Modellierung durchgeführt. Codegröße und Laufzeit stehen hierbei im Vordergrund. In der Modul-Spezifikation beim Entwurf des APR sind daher viele Details zu beachten. Meist müssen sehr viele Signale in die Regelung einbezogen werden. Darüber hinaus sind die rein physikalischen Zusammenhänge zwischen den Signalen äußerst komplex.

Ein Resonanztransformator, wie in der Abbildung 68 dargestellt, ist eine schwingkreisähnliche Schaltung aus Kondensator ***C*** und Spule ***L***, um auf einer vorgegebenen Frequenz Leistungsanpassung zwischen Bauelementen oder Baugruppen zu erreichen. Bei Hochfrequenzanwendungen entfällt im Regelfall der Eisenkern der Spule, bei niederfrequenten Anwendungen können bei Spule auch Ferritkerne zur Anwendung kommen. In der Hochfrequenztechnik betreibt man vorzugsweise Leistungstransistoren und Elektronenröhren als Schalter, um durch hohen Wirkungsgrad unnötige Verlustwärme zu vermeiden.

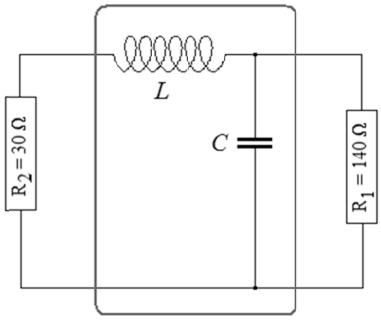


Abbildung 68: Prinzip Resonanztransformator

Gemäß den Gesetzen der Fourieranalyse entstehen durch abruptes Ein- und Ausschalten einer Spannung viele Oberwellen, die abgestrahlt werden und die Funktion anderer Geräte stören. Um das zu verhindern, müssen Tiefpassfilter oder Schwingkreise oder Resonanztransformatoren ausreichend hoher Güte eingebaut werden.

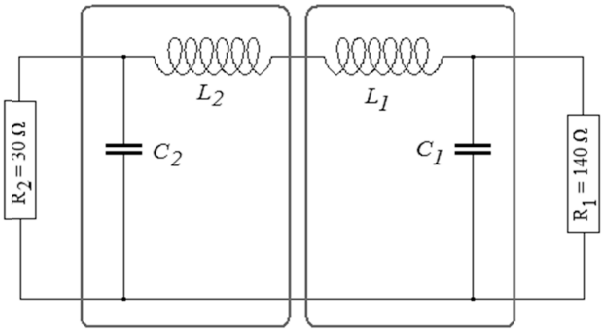


Abbildung 69: Prinzip PI-Filter

Bei den eben beschriebenen einfachen Resonanztransformatoren hängt der Gütefaktor ausschließlich vom Verhältnis der Widerstände an Ein- und Ausgang ab. Wenn die Widerstände etwa gleichen Wert haben, ist der Gütefaktor zu gering, um nennenswerte Filterwirkung sicherzustellen. Das lässt sich durch Kombination zweier Resonanz-transformatoren ändern.

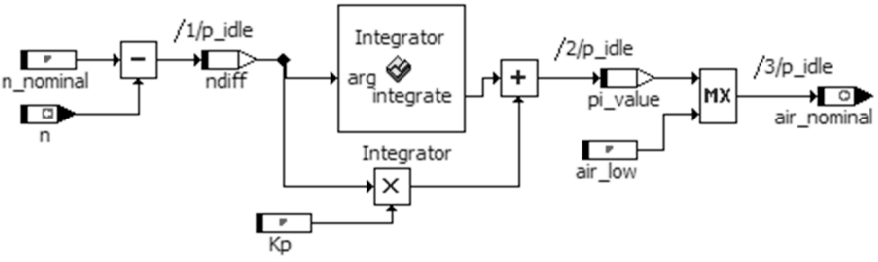


Abbildung 70: PI-Regler Drehzahl (Fensterhebermotor), nach ASCET-MD

Als Beispielmodell wird für das APR ein Regler-Entwurf mit ASCET-MD [ASCT09] modelliert, der auf einem standardmäßigen PI-Filter basiert, wie in der Abbildung 70 gezeigt. Der Regler wird zum Konstanthalten der Drehzahl des Fensterhebermotors (Betrachtung: Leerlauf) eingesetzt. Beim Regeln der Leerlaufdrehzahl eines Motors muss beachtet werden, dass die Ist-Drehzahl ***n*** möglichst dicht bei der Leerlaufnenndrehzahl ***n_nominal*** bleibt. Zum Ermitteln der zu regelnden Abweichung wird der Wert ***n*** von ***n_nominal*** subtrahiert.

In der Phase der Modulspezifikation des APR entsteht somit ein kollaboratives Artefakt (Regler-Modell) des Werkzeugs ASCET-MD, welches systemlogisch und prozesslogisch mit den anderen Artefakten in Beziehung steht. Das ASCET-MD-Modell kann als XML-basiertes logisches Artefakt exportiert werden.

8.2.5 Testspezifikation

Die systematische Ermittlung von Testszenarien unter funktionalen wie unter strukturellen Gesichtspunkten wurde bereits im Kapitel 3.8.4 ausführlich eingeführt. Testfälle eines Klassifikationsbaums ermöglichen eine effektive Aufdeckung systembezogener Fehlerfälle und prüfen die fehlerfreie Funktionalität des Systemverhaltens und dessen Einbettung in die E/E-Systemarchitektur. Die Abbildung 71 stellt grob eine Übersicht über den APR-Klassifikationsbaum modelliert im Werkzeug CTE/XL [CTEX09] dar.

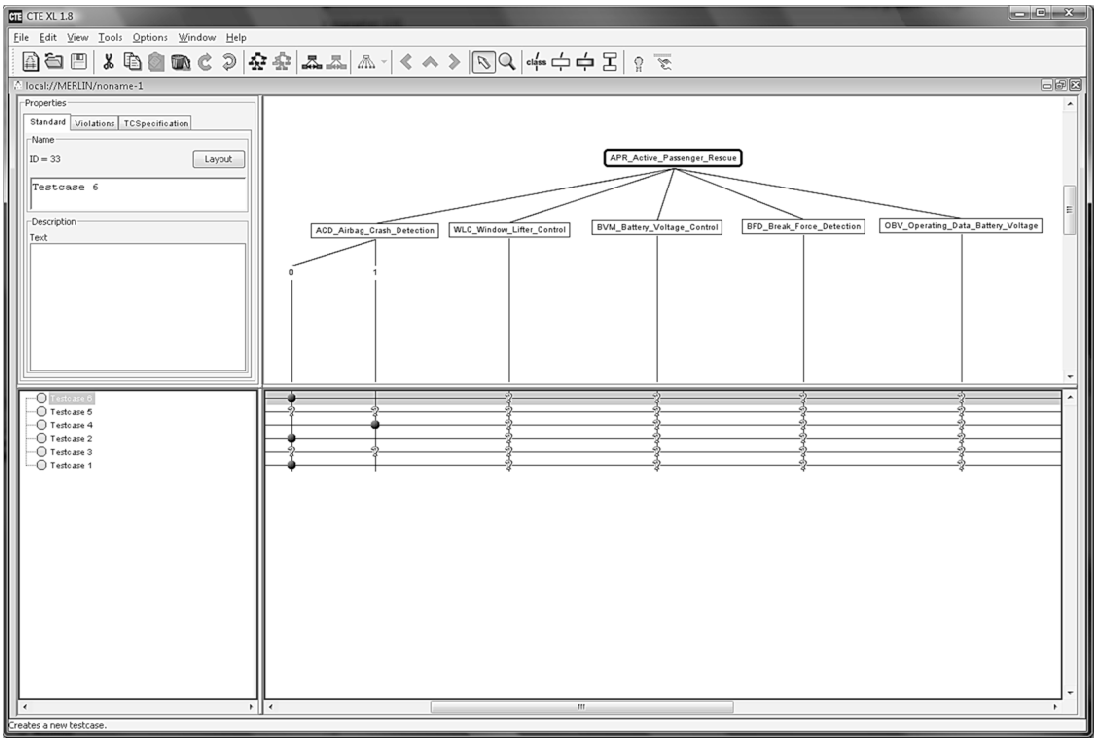


Abbildung 71: APR-Klassifikationsbaum in CTE XL

In der Phase der Testspezifikation des APR entsteht somit ein kollaboratives Artefakt (Klassifikationsbaum) des Werkzeugs CTE/XL, welches systemlogisch und prozesslogisch mit den anderen Artefakten (Modelle der funktionalen Spezifikation) in Beziehung steht. Der Klassifikationsbaum ist in einer XML-basierten Datei gespeichert und kann direkt als logisches Artefakt dienen.

8.2.6 Glossar

Im modellbasierten Entwicklungsprozess hilft der Einsatz von Glossaren bei der Benennung von Funktionen, Schnittstellen und Zustandsvariablen für die Wahl eindeutiger Bezeichnungen bei der Modellbildung. Mehrdeutigkeit in der Namensgebung kann unerwünschte Nebeneffekte hervorrufen, die durch die Modellierung gerade vermieden werden sollen. Jedoch sind Konventionen bezüglich der Namensgebung weder ein Bestandteil von Modellierungs- oder Entwicklungswerkzeugen, noch sind sie teils übergreifender Natur und obliegen somit dem Systemarchitekten.

An dieser Stelle sei daher angemerkt, dass vor dem Beginn der Entwicklung eines Systems ein Regelwerk für die Namensgebung durchaus eine verbesserte Struktur schaffen kann, sodass dies im APR-Entwurf auch eine Anwendung findet.

Die Abbildung 72 zeigt einen Auszug des tabellarischen Glossars für die APR-Entwicklung, begrifflich angelehnt an [BAU03], mit den Akronymen der Systemfunktionen, der Zugehörigkeit zum Oberbegriff (‚Catalog'), der Systemfunktionsnamen (‚Systembegriff') sowie einer kurzen Beschreibung in Microsoft Excel.

	Akronym	Systembegriff	Beschreibung
Catalog	**SYC**	**SYC_System_Control**	**Systemsteuerung**
	SGD	SGD_System_Control_Gasoline_Direct_Injection_Mod	Benzindirekteinspritzmodus
	ACD	ACD_Airbag_Crash_Detection	Sensorik Auswertung Crash-Fall
	APR	APR_Active_Passenger_Rescue	Insassenschutz beim Crash-Fall
	WLC	WLC_Window_Lifter_Control	Fensterheberfunktion, Basis und Comfort
	BVC	BVM_Battery_Voltage_Control	Batteriemanagement, Überspannungsschutz
	BFD	BFD_Break_Force_Detection	Steuergerät für Lichtsteuerung
Catalog	**TD**	**TDA_Torque_Demand**	**Drehmomentanforderung**
	TDA	TDA_Torque_Demand_Auxiliary_Functions	Systemzustandssteuerung, Steuergerätenachlauf
	TDC	TDC_Torque_Demand_Cruise_Control	Fahrgeschwindigkeitsregler
	TDD	TDD_Torque_Demand_Driver	Fahrerwunsch
	TDI	TDI_Torque_Demand_Idle_Speed_Control	Leerlaufdrehzahlregelung
	TDS	TDS_Torque_Demand_Signal_Conditioning	Momentenanforderung Signalaufbereitung
Catalog	**TS**	**TS_Torque_Structure**	**Drehmomentstruktur**
	TCA	TCA_Torque_Conversion_Air	Berechnung Momentenumsetzung Luft
	TCC	TCC_Torque_Conversion_Combustion	Berechnung Momentenumsetzung Verbrennung
	TCD	TCD_Torque_Coordination	Koordination Drehzahl
	TMO	TMO_Torque_Modelling	Berechnung Momentenmodell Drehmoment
Catalog	**OPO**	**OP_Operating_Data**	**Betriebsdatenerfassung**
	OBV	OBV_Operating_Data_Battery_Voltage	Bereitstellung Versorgungsspannungssignale, Diagnose
	OEP	OEP_Operating_Data_Engine_Position_Management	Berechnung Motordrehzahl, Position Kurbelwelle & Nockenwelle
	OMI	OMI_Misfire_Detection_Irregular_Running	Überwachung Segmentzeit Drehzahl Kurbelwelle
	OTM	OTM_Operating_Data_Temperature_Measurement	Plausiblisierung der Temperatursignale
Catalog	**VL**	**VL_Vehicle_Level**	**Fahrzeugdomäne**
	VLpt	VL_Vehicle_Level_Power_Train	Antriebsstrang
	VLps	VL_Vehicle_Level_Power_System	Spannungsversorgungsbeschaltung
	VLbd	VL_Vehicle_Level_Body	Fahrgestell
	VLcs	VL_Vehicle_Level_Chassis	Fahrwerk
	VLcm	VL_Vehicle_Level_Comfort	Komfortfunktionen, Fahrerassistenz
	Vlte	VL_Vehicle_Level_Telematics	Infotainment
Catalog	**SD**	**SD_System_Diagnostics**	**Systemdokumentation**
	SDF	SD_System_Diagnostics_Vehicle_Data	Fahrzeugdaten
	SDK	SD_System_Diagnostics_Configuration_Data	Konfigurationsbeschreibungen
	SDM	SD_System_Diagnostics_Engine_Data	Motordaten
	SDB	SD_System_Diagnostics_ECU_Data	Steuergerätebeschreibung

Abbildung 72: Glossar der Systemfunktionen (Auszug) in MS Excel

Vereinheitlichte Namenskonventionen sind weit verbreitet und oft immer dann in einem Prozess vorzufinden, sobald Software aus Artefakten herausgeneriert wird. Ein Beispiel einer Namenskonvention aus der modellbasierten Entwicklung eingebetteter Systeme soll den Nutzen eines Glossars für den modellbasierten Entwicklungsprozess verdeutlichen.

Beispiel:

Sei ***Winkelgeschwindigkeit Berechnung*** der Name einer Funktion im Modell, so ist schwierig erkennbar, wie diese Funktion in der logischen Systemarchitektur einzuordnen ist. Ein strukturierter Name, wie `VLcm_APR_WLC_AVC_SDF` hingegen erlaubt eine konkrete Zuordnung allein durch den Aufbau des Namens: Es handelt sich hierbei um die Fahrzeugfunktion APR-Funktion (`APR`) aus dem Bereich der Komfortfunktionen (`VLcm`). Die Funktion zur Berechnung der Winkelgeschwindigkeit ist eine Verfeinerung der Funktion des Fensterheber-Steuergerätes (`WLC`) Fensterheberfunktion und bietet die Funktionalität ***Angular Velocity Calculation*** (Berechnung der Winkelgeschwindigkeit). Schließlich werden für Diagnosezwecke Fahrzeugdaten (`SDF`) erfasst.

Das kollaborative Artefakt Glossar des Werkzeugs Microsoft Excel steht mit allen Artefakten (insb. Modellen der funktionalen Spezifikation) prozesslogisch in Beziehung. Als logisches Artefakt kann es XML-basiert (nach OOXML) gespeichert werden.

8.3 Evaluierung der regelbasierten Prüfung an APR-Artefakten

Nachdem die im modellbasierten Entwicklungsprozess der APR-Funktion auftretenden, prozesslogisch in Beziehung stehenden, kollaborativen Artefakte im Kapitel 8.2 vorgestellt wurden, werden in diesem Kapitel nun Prozessphasen bezogene Konformitätsrichtlinien angewandt. Im Evaluierungszeitraum wurden viele Richtlinien aus unterschiedlichen Richtlinienkatalogen herangezogen, die hier nicht alle beschrieben werden können. Daher soll exemplarisch eine Auswahl von 15 repräsentativen Richtlinien zur Demonstration der Machbarkeit des in dieser Arbeit vorgestellten Ansatzes genügen. Für die Evaluierung der regelbasierten Konformitätsprüfung bilden die kollaborativen Artefakte der APR-Systemfunktion dabei den möglichen Prüfraum. Für jede Prozessphase wird aus dem Prüfraum der Prüfaufbau vor einer Prüfung gebildet. Die Evaluierung erfolgt, indem die natürlich sprachlichen Richtlinien mittels der aus Kapitel 6 erarbeiteten Methodik formalisiert und danach in unterschiedlichen Programmiersprachen programmiert werden.

Evaluierung:

Die Richtlinien werden durch folgende Aspekte klassifiziert: die Prozessphase, in der sie gelten, der Art der Richtlinie, der Kategorie, der Priorität, welches Artefakt sie bei der Ausführung betreffen, der Implementierung sowie des Evaluierungsergebnisses. Die entstandenen Regeln müssen sodann mit dem in Kapitel 7 vorgestellten Prüfsystemen am Prüfaufbau automatisiert ausführbar sein, sodass eine regelbasierte Konformitätsprüfung an den kollaborativen Artefakten der APR-Systemfunktion möglich wird.

Die in OCL implementierten Richtlinien wurden mit dem Prüfwerkzeug *ASD-Regel-Checker* ausgeführt und das Ergebnis ausgewertet. Die in LINQ und in M-Skript implementierten Richtlinien wurden mit dem Prüfwerkzeug *Assessment Studio* ausgeführt und das Ergebnis ausgewertet. Die Richtlinien und die Ergebnisse der Evaluierung sind in der Tabelle 24 zusammengefasst. Es ließen sich alle Richtlinienprüfungen durchführen. Daraus resultierend ist der Nachweis erbracht, dass der in diese Arbeit vorgestellte Ansatz auch eine werkzeuggestützte Automatisierung zulässt.

Ergebnis: „*Am Fallbeispiel der Richtlinien und Artefakte des modellbasierten APR-Entwicklungsprozesses konnte nachgewiesen werden, dass die regelbasierte Konformitäts-prüfung kollaborativer Artefakte machbar ist. Die artefakt- bzw. werkzeugübergreifende Prüfung erfolgte werkzeuggestützt und erlaubt eine Automatisierung im Prozess*.“

Tabelle 24: Evaluierungsmatrix

	Prozessphase	Richtlinie	Kategorie	Priorität	Artefakt	Implementierung	Evaluierung
1	Anforderungs-management	Nachvollziehbarkeit	Verständlichkeit	Empfohlen	DOORS, UML	LINQ	übergreifend ausführbar
2	Anforderungs-management	Ausdrucksklarheit	Verständlichkeit	Empfohlen	DOORS	LINQ	ausführbar
3	Anforderungs-management	Nebensätze sind im Anforderungstitel nicht erlaubt	Übertragbarkeit	Erforderlich	DOORS	LINQ	ausführbar
4	Anforderungs-management	Überprüfung von eineindeutigen Anforderungs-Bezeichnern	Funktions-absicherung und Zuverlässigkeit	Erforderlich	DOORS	LINQ	ausführbar
5	Modellbasierter Systementwurf	Maximale Namenslänge von Bezeichnern in Modellen	Benutzbarkeit und Verständlichkeit	Erforderlich	UML, ML/SL/SF	OCL, LINQ	übergreifend ausführbar
6	Modellbasierter Systementwurf	Erlaubter Zeichensatz	Verbesserung der Übertragbarkeit	Erforderlich	UML, ML/SL/SF	OCL, LINQ, M-Skript	übergreifend ausführbar
7	Modellbasierter Systementwurf	Erlaubte Akronyme in Bezeichnern	Erhöhung der Änderbarkeit und Wiederverwendbkt.	Empfohlen	ASCET-MD, MS Excel	LINQ	übergreifend ausführbar
8	Modellbasierter Systementwurf	Gültigkeitsbereich für Variable	Funktions-absicherung und Zuverlässigkeit	Erforderlich	ASCET-MD	LINQ	ausführbar
9	Modellbasierter Systementwurf	Vermeidung von Division durch Null	Funktions-absicherung und Zuverlässigkeit	Erforderlich	ASCET-MD	LINQ	ausführbar
10	Modellbasierter Systementwurf	Vermeidung von Mehrfach-berechnungen	Verbesserung der Effizienz	Empfohlen	ASCET-MD	LINQ	ausführbar
11	Modellbasierter Systementwurf	Vermeidung von infiniten Wertebereichsgrenzen	Funktions-absicherung und Zuverlässigkeit	Empfohlen	ASCET-MD	LINQ	ausführbar
12	Modul-spezifikation	Referenzauflösung und Zugehörigkeit	Konsistenz	Erforderlich	AUTOSAR	LINQ	ausführbar
13	Testspezifikation	Setzen globaler Parameter	Funktions-absicherung und Zuverlässigkeit	Empfohlen	CTE/XL	LINQ	ausführbar
14	Testspezifikation	Strukturierung der Top-Level Kompositionen	Richtlinien zur Verbesserung der Benutzbarkeit und Verständlichkeit	Empfohlen	CTE/XL	LINQ	ausführbar
15	Testspezifikation	Strukturierung der Parameter-Komposition	Richtlinien zur Verbesserung der Übertragbarkeit	Empfohlen	CTE/XL	LINQ	ausführbar
16	Testspezifikation	Konformität der Signalspezifikation	Richtlinien zur Verbesserung der Benutzbarkeit und Verständlichkeit	Empfohlen	CTE/XL, MS Excel	LINQ	übergreifend ausführbar

Die in der Tabelle 24 aufgeführten Richtlinien 1 - 16 sind im Anhang − B der Arbeit ausführlich beschrieben. Zusätzlich sind auch die jeweiligen Implementierungen in den verschiedenen Programmiersprachen dokumentiert.

9 Zusammenfassung

Die Einhaltung regulatorischer Vorschriften, von Prozessrichtlinien und von Arbeitsanweisungen im Engineering-Prozess gehört gerade in sicherheitskritischen Anwendungsdomänen wie der internationalen Automobiltechnik, der Bahntechnik sowie im Flugzeugbau zu den wichtigsten Kernkompetenzen eines qualitätsorientierten Produktherstellers eingebetteter Systeme. Dabei müssen beim Hersteller und dessen Zulieferern immer komplexere Informationen aus verschiedenen Softwaresystemen unter Einhaltung der strengen Vorschriften in transparenter und nachvollziehbarer Weise, unter engen zeitlichen Vorgaben, von den Mitarbeitern im Entwicklungsprozess korrekt verstanden, konform bearbeitet und schließlich konsistent zusammengetragen werden.

Aufgrund der engen Verzahnung der Entwurfsprozesse mit der Informationstechnologie (IT) erlangen zunehmend immer mehr Konformitätsanforderungen auch Gültigkeit für die IT-Infrastruktur und die IT-gestützten, modellbasierten Entwicklungsprozesse. Dies sind Vorschriften zur konformen Verwendung der eingesetzten Programme (Software-Werkzeuge) und den damit erstellten Spezifikationen, modellbasierten Designs, den Dokumenten sowie generell allen gespeicherten Daten. Die Einhaltung von Vorschriften ist heute in jeder Entwicklungsabteilung vorgeschrieben und oftmals in großen Unternehmen durch den Oberbegriff *Compliance* institutioniert. In der Praxis kämpfen jedoch viele Entwicklungsabteilungen (Hersteller sowie Zulieferer) mit fehlerhaften Formularen, inkonsistenten Datenbeständen, kreativ designten Architektur- und Simulationsmodellen und lückenhaften Tabellenkalkulationen. Abteilungen und Geschäftsbereiche arbeiten aufgrund technischer Inkompatibilität der IT-Werkzeuge nicht immer effizient zusammen. Hohe Review-Aufwände und Abstimmungsprozesse entstehen tagtäglich. So werden konsolidierte Budgets und Projektplanungen erschwert, komplexe Produktabsicherungen für sicherheitskritische Steuerungssoftware zukünftig sogar unmöglich. Risiken werden zu oft in Kauf genommen und gravierende Design-Fehler spät bei Produkttests oder erst bei der Produktzulassung gefunden. Im sicherheitskritischen Umfeld haben jedoch Produkte, speziell die eingebetteten Systeme, bei Fehlfunktion eine unmittelbare Auswirkung auf Menschenleben. Produkthersteller unterliegen daher gesetzlichen Bestimmungen zur Produktentwicklung und müssen zunehmend notwendige Prozess-Zertifizierungen für wettbewerbsdifferenzierende Ausschreibungen nachweisen können.

Die vorliegende Arbeit wurde durch diese industrielle Herausforderung nachhaltig motiviert. Die Zielvorstellung war zu Beginn, dass Unternehmen für ingenieursmäßige Entwicklungsprozesse automatisiert nachweisen können, dass die IT-gestützten Prozesse bzgl. der geltenden Anforderungen konform ablaufen, alle Datenbestände konsistent gehalten werden können und innerhalb der Entwicklungsprozesse ‚vorschriftsgemäß' gearbeitet wurde, dass zur Weiterverarbeitung die Dokumente konsistent vom Hersteller zum Zulieferer ausgetauscht werden können, dass am Ende eines Entwicklungszyklus automatisch eine Prozessbewertung generiert würde, die Richtlinienkonformität aller elektronischen Arbeitserzeugnisse (Artefakte) ausweist sowie mögliche Verbesserungspotenziale, auch zwischen den Abteilungen, aufdeckt.

In der Arbeit wurde daher zunächst der modellbasierte Entwicklungsprozess eingebetteter Systeme vorgestellt und hinsichtlich industriell geltender Normen aus den Standardisierungsgremien (wie ASAM, DIN, HIS oder die ISO/IEC), den geltenden Vorschriften für die Handhabung von modellbasierten Software-Werkzeugen (wie AUTOSAR, MAAB oder MISRA), den Vorgaben etablierter Prozessrichtlinien aus Reifegradmodellen (wie CMMI oder SPiCE) und derer Wirkung auf elektronische Arbeitsergebnisse genauer betrachtet und erläutert. Hierbei wurde festgestellt, dass aus Normungen und Konventionen für Produkthersteller und Zulieferer unterschiedliche Entwicklungsrichtlinien für die jeweiligen Phasen eines Produktentstehungsprozesses gelten, diese teilweise überlappen und durch die natürlich sprachliche Dokumentation unterschiedlich interpretiert werden können. Eine genauere Betrachtung führte zu der Erkenntnis, dass von Entwicklungsrichtlinien implizite, prozesslogische Anforderungen an die elektronischen Arbeitsergebnisse − die sogenannten *Artefakte* − gestellt werden. Dies führte zur Schlussfolgerung, dass vielfältige Entwicklungsrichtlinien sowohl prozess- als auch werkzeugübergreifende Anforderungen an die Artefakte aufstellen. Konkret bedeutet dies die konforme Erstellung von modellbasierter Spezifikation, konsistenten Dokumenten und korrekter Daten in einer IT-Umgebung.

Eine prozessübergreifende und eine technische Analyse sowie eine stichprobenartige Untersuchung gängiger Werkzeuge ergab schließlich, dass viele Artefakte im Produktentstehungsprozess entweder durch mehrere Personen bearbeitet werden oder durch technische Automatisierungen logisch verbunden sind (Werkzeugkopplung, Code- oder Testfallgenerierung) − somit in vielen Fällen prozesslogisch miteinander in Beziehung stehen. An solchen, sogenannten *kollaborativen Artefakten* einer IT-Umgebung, wurden werkzeugübergreifende und prozesslogische Abhängigkeiten aus der Kombination von Eigenschaften einzelner Artefakte oder durch Anwendung gekoppelter Werkzeuge genau identifiziert. Des Weiteren wurde erkannt, dass *Entwicklungsrichtlinien* für solche kollaborativen Artefakte natürlich sprachlich dokumentieren, wie Produktspezifikationen einheitlich formuliert, Software-Architekturen konform modelliert, Dokumente konsistent bearbeitet, Simulationen fehlerfrei entworfen, eingebettete Software implementiert oder Daten in spezieller Formatierung für die Testautomatisierung prozessübergreifend abgelegt werden sollen. Daher erfordert der ingenieurmäßige Entwurf eines eingebetteten Systems gerade in solchen kollaborativ durchgeführten Arbeitsschritten die *Konformität* (Erfüllung der Anforderungen) zu prozessspezifischen Entwicklungsrichtlinien. Dies hat eine Auswirkung bis hin zur Produktqualität eines eingebetteten Systems, da arbeitsteilige Entwurfsartefakte in einer IT-Umgebung die frühe Systemauslegung des eingebetteten Systems bestimmen und somit die Systemspezifikation für nachgelagerte Fertigungsprozesse bilden. Durch den Vergleich zu Qualitätskriterien des ISO-Standards für Software-Produktqualität (ISO/IEC 9126) wurde herausgefunden, dass die Konformität zu den Entwicklungsrichtlinien im Prozess sich entscheidend auf die Qualitätseigenschaften des fertigen Produkts auswirkt. Daher kommt der statischen Konformitätsanalyse logisch-abhängiger, nicht-funktionaler Eigenschaften von kollaborativen Artefakten zur Entwurfszeit eine besondere Bedeutung zu.

Im Hauptteil der Arbeit wurde die Erforschung eines ingenieurmäßigen Verfahrens zur werkzeugübergreifend einsetzbaren und automatisierten Überprüfung auf Konformität von Entwicklungsrichtlinien an kollaborativen Artefakten durchgeführt. Ausgehend von regelbasierten Ansätzen zur statischen Analyse einzelner Artefakte wurde die Artefakt-Prüfung hin zu einer werkzeugübergreifenden Lösung konzipiert, welche mehrere prozesslogisch verbundene Artefakte automatisiert auf Erfüllung der Anforderungen prüft.

Hierfür wurde das das Konzept des virtuellen, plattformunabhängigen Datenmodells, des so genannten *logischen Artefakts* eingeführt. Durch die Abstraktion und Transformation kollaborativer Artefakte in ein abstraktes Datenmodell sowie durch die Formalisierung natürlich sprachlicher Richtlinien in digitale Regeln (computerausführbare Algorithmen) konnten interdisziplinäre Entwicklungsrichtlinien und ihre semantischen Abhängigkeiten auf dem logischen Artefakt (virtuelle Datenmodelle) beschrieben, ausgeführt und schließlich Entwicklungsrichtlinien hinsichtlich Konformität durch eine entwickelte Methodik und ein Prüfsystem automatisiert prüfbar werden.

Im Lösungsweg wurden zunächst umgangssprachliche Prozessrichtlinien analysiert und durch syntaktische Dekomposition und semantische Attributierung mit kollaborativen Artefakten in Beziehung gesetzt. Hierfür wurde durch ein entwickeltes Vorgehensmodell – das *VR-Modell* – eine deduktive Methodik eingeführt, welche als Abstraktionskonzept die Metamodellierung verwendet. Durch das Abstraktions-Paradigma der Model-Driven Architecture wurden logische Artefakte mittels direkter und indirekter Transformation gebildet. So konnte demonstriert werden, wie sich auch komplexere Konformitätsprüfungen auf einer abstrakteren Ebene durchführen sowie abgeleitete Aussagen auf spezifische Artefakte bestimmen lassen.

Mathematische Beschreibungsmittel wie die Mengenlehre und die Aussagen- und Prädikatenlogik verhalfen zur Formalisierung der natürlich sprachlichen Richtlinien und zur Ermittlung einer Konformitätsbewertung. Aus diesen wurden computerausführbare Ausdrücke (*Regeln*) mittels Algorithmen zur statischen Analyse auf dem logischen Artefakt konzipiert.

Zur Strukturierung von Regelalgorithmen wurde die Auszeichnungssprache *Query-based Rule Description Language (QRDL)* entworfen, um hieraus die nachfolgenden Regelimplementierungen in diversen Programmiersprachen zu unterstützen. Exemplarisch wurden Beispielrichtlinien an typischen, realitätsnahen Artefakten (Spezifikationen, Modellen sowie Testbeschreibungen) in den ausgewählten Programmiersprachen LINQ, OCL und M-Skript jeweils als Regeln implementiert.

Die Validierung der Ergebnisse erfolgte in einer Machbarkeitsstudie durch die probenhafte Umsetzung von Richtlinien an kollaborativen Artefakten des modellbasierten Entwicklungsprozesses „*Active Passenger Rescue*“ – einer Fahrzeugfunktion für den Insassenschutz. Hierfür wurde die Implementierung von technisch unterschiedlichen Prüfsystemen (*ASD-Regel-Checker* und *Assessment Studio*) vorgenommen, welche an existenten Entwicklungsrichtlinien aus Industrienormen angewendet wurden und schließlich die Konformität von kollaborativen Artefakten pro Prozessphase automatisiert untersuchen und feststellen konnten.

Eine durchgeführte Skalierbarkeitsuntersuchung der Prüfsysteme ergab, dass die regelbasierte Konformitätsprüfung kollaborativer Artefakte für die beiden umgesetzten Prüfsysteme gut skaliert.

Zusammenfassend konnte durch die Arbeit eine werkzeuggestützte Methodik für den ingenieursmäßigen Entwurf eingebetteter Systeme zur werkzeugübergreifenden Konformitätsprüfung erfolgreich entwickelt werden, welche eine wirksame Reduktion des heutzutage von Ingenieuren manuell durchgeführten Review-Aufwandes in kollaborativen Prozessen ermöglicht. Außerdem werden Schwachstellen der heutigen Verifikation und Validation in der Prozesskette früher aufgedeckt, da menschliche Fehler im prozessübergreifenden Planungs- und Designprozess verringert werden. Des Weiteren minimiert die Formalisierung der Prozessrichtlinien die falsche Anwendung von Richtlinien

aufgrund einer Fehlinterpretation durch den Menschen im Arbeitsalltag. Schließlich dient eine werkzeugübergreifend durchgeführte Analyse der Reduzierung des IT-Risikos gegeben durch inkonsistente Dokumente und Daten, des Entwicklungsrisikos durch falsche Anwendung von IT-Werkzeugen, des Produkthaftungsrisikos durch Nichteinhaltung von Entwicklungsstandards und der Betriebskostensenkung durch Reduktion der Review-Aufwände sowie der Erreichung interner und externer Konformitäts-Ziele durch Erfüllung der Nachweispflicht.

9.1 Reflektion

Während diese Arbeit entstand, sind viele Vorträge und technische Demonstrationen der regelbasierten Konformitätsprüfung kollaborativer Artefakte, insbesondere Vorführungen des Prüfsystems *Assessment Studio*, vor fachlich versiertem Publikum − national und international − abgehalten worden. Interessant war die Beobachtung, dass nahezu alle Entwicklungsprozesse durch kollaborative Artefakte geprägt sind und sich Prozessrichtlinien fügen müssen. Dies wird vermutlich noch zunehmen, da gerade *agile Prozesse* derzeit ein Unternehmenstrend sind und viele Konsistenzprobleme verantworten. Das Feedback der Vorführungen war größtenteils positiv, wobei natürlich auch Kritik geäußert wurde. Die kritisierten Aspekte sollen daher auch Platz in dieser Arbeit finden und nachfolgend diskutiert werden.

Eine oft gestellte Frage war, wo man die Konformitätsprüfung im Unternehmen sinnvollerweise etablieren sollte. Dies ist sicherlich unternehmensspezifisch und nicht generell zu beantworten. Die Verantwortung für ein *Qualitätsmanagementsystem* (QMS) jedenfalls liegt bei allen (angewandten) Qualitätsmanagementsystemen bei der Unternehmensleitung. Betrachtet man die Konformitätsprüfung als Teil des QM, muss diese ebenso für die Umsetzung der Konformitätsprüfung in jeder Hierarchieebene (abteilungsbezogen oder projektbezogen) sorgen. Die von dem Management gelebte Qualitätspolitik legt demnach die Konformitäts-Ziele und Konformitäts-Absichten sowie die Verantwortungen im Prozess fest. Ergebnis aller Erfahrungen und auch von Kundenanforderungen ist ein auf das jeweilige Unternehmen zugeschnittene QMS, dessen Teilkomponente wiederum unser vorgestelltes Prüfsystem ist. Dies beinhaltet ein oder mehrere digitale Regelwerke, welche in ausführbarer Form ein sogenanntes *Quality Gate* darstellen. Quality Gates sind vor Projektbeginn festgelegte Meilensteine im Projektablauf, bei denen anhand vorher definierter Konformitätskriterien über die Freigabe des nächsten Projektprozesses entschieden wird. Das Ergebnis ist ein System von *Quality Gates* (QGS) im gesamten Unternehmen, mit deren Hilfe ein effektives Qualitätscontrolling – also die Konformitätsbewertung pro Zeitabschnitt − innerhalb der einzelnen Prozessschritte und prozessschrittübergreifend sichergestellt werden kann.

Oftmals angefragt wurden die Voraussetzungen für die Konformitätsprüfung. Eine Vorbedingung des Ansatzes ist es, dass natürlich sprachliche Richtlinien für einen Prozess gelten und dass kollaborative Artefakte in einem Diskursbereich existieren. Sind keine Richtlinien vorgegeben oder die Richtlinien zu ungenau (zu abstrakt) verfasst, ist eine regelbasierte Konformitätsprüfung nicht möglich. Das Gleiche gilt für die Prüfung von Informationen, die nicht durch ein elektronisches System erfasst sind. Dies kann beispielsweise bei (von einem Management erteilten) Handlungsanweisungen der Fall sein. In solchen Fällen ist zu prüfen, ob der Ansatz tatsächlich eine Automatisierung erlaubt.

Weitere Fragestellungen adressierten die Korrektheit einer Konformitätsprüfung. Wann ist beispielsweise eine Konformitätsaussage verlässlich? Der Entwurf von Prüfregeln

(Algorithmen) unterliegt einem kreativen Denkprozess, durchgeführt durch eine fachlich geschulte Person. Wie bei kreativen Prozessen üblich, gibt es mehrere Lösungswege und leider auch Fallen. Der hier vorgestellte Ansatz zur regelbasierten Konformitätsprüfung kollaborativer Artefakte ist daher abhängig von der Güte der Prüf-Regeln. Erfasst ein Prüfalgorithmus beispielsweise nicht alle Daten eines Artefakts, so könnte eine falsche Schlussfolgerung die Konformität bestätigen, obwohl dies in der Realität nicht der Fall ist. In der Pragmatik wurden zur Vermeidung dieser Situation daher immer jeweils Positiv- und Negativ-Beispiele eines kollaborativen Artefakts pro Regelalgorithmus erzeugt, um nachzuweisen, dass der Prüfalgorithmus korrekt arbeitet.

Schließlich wurden Fragen hinsichtlich möglicher Risiken in der Automatisierung betreffend von Falschaussagen gestellt. Auch die Abstraktion und der Entwurf der indirekten Transformationsvorschriften in diesem Ansatz ist ein kreativer Prozess, durchgeführt durch eine fachlich geschulte Person. Hierfür gelten die gleichen Schwachpunkte wie für den Entwurf der Prüf-Regeln. Wird beispielsweise eine zu starke Abstraktion vom ursprünglichen Datenmodell des kollaborativen Artefakts durchgeführt, kann mitunter keine hinreichend verwertbare Konformitätsaussage getroffen werden. Wird durch Transformation die ursprüngliche Semantik verfälscht, ist auch eine resultierende Konformitätsaussage falsch.

9.2 Ausblick

Ein erklärtes Ziel der modellbasierten Entwicklung ist der durchgängige Einsatz im Produktentstehungsprozess. Ihr Anspruch ist die Optimierung des Entwicklungsprozesses durch Reduktion der Komplexität und des Zeitaufwandes bei gleichzeitiger Erhöhung der Qualität. Auch zukünftig wird hinsichtlich der Qualitätsaspekte die methodisch und werkzeugtechnisch durchgängige Gestaltung der Entwicklungsprozesse für eingebettete Systeme in der frühen Phase der Systementwicklung noch weitere Herausforderungen mit sich bringen. Insbesondere die im Rahmen der Forschungsarbeiten gefundenen Potenziale einer „noch intelligenteren“ Konformitätsprüfung sollten zahlreiche Ausgangspunkte für weiterführende Arbeiten bieten.

Durch die von den Prozessanforderungen auf die Konformitätsprüfung übertragenen Merkmale ergeben sich weiterführende wissenschaftliche Fragestellungen. Dies sind z. B. Untersuchungen bzgl. der Verbesserung von Effektivität (Wirksamkeit), der Erhöhung von Effizienz (Wirtschaftlichkeit), der Kontrollierbarkeit und Steuerbarkeit (Workflows) sowie der Anpassungsfähigkeit (Agilität) einer automatisierten Konformitätsprüfung.

- *Verbesserung von Effektivität (Wirksamkeit):* Zur Verbesserung der Prozesse im Unternehmen ist es eine spannende Fragestellung, wie sich die Konvergenz regelbasierter Konformitätsprüfung und betriebswirtschaftlicher Bewertungsgrößen herstellen lässt und welche Optimierungspotenziale die Synergie bietet. Betriebswirtschaftslehre Kennzahlen (Key Performance Indicators, KPI) sind ein Mittel der Betriebswirtschaft, anhand derer der Fortschritt oder der Erfüllungsgrad hinsichtlich wichtiger Zielsetzungen oder kritischer Erfolgsfaktoren innerhalb einer Organisation gemessen oder ermittelt werden kann. Da angestrebte Zielsetzungen auch immer mithilfe von aufgestellten Kennzahlen (Metriken) mit verknüpften Berechnungsformeln definiert werden, verbessert die Erweiterung von Entwicklungsrichtlinien um kennzahlengestützte Konformitätsprüfungen die Wirksamkeitsbewertung von Prozessen, um schließlich die Zielerreichung bereits in frühen Phasen evaluieren zu können. Bei automatisierter Konformitätsprüfung

können somit nicht nur die erreichten Fortschritte, sondern auch die Rückschritte anhand der Erreichung der Kennzahlen (Erfüllung einer Anforderung) rechtzeitig erkannt werden. Eventuelle Schwachstellen oder Fehlentscheidungen können somit schneller behoben bzw. frühzeitig rückgängig gemacht werden.

- *Erhöhung von Effizienz (Wirtschaftlichkeit):* Die Effizienz steht immer in Relation zu einem Ressourcenaufwand. Dieser wird maßgeblich durch den Zeitfaktor bestimmt. Die regelbasierte Konformitätsprüfung kollaborativer Artefakte nimmt in der Praxis eine gewisse Zeit in Anspruch, da das kollaborative Artefakt nach der Fertigstellung zu einem bestimmten Zeitpunkt durch die Anwendung eines Prüfsystems statisch analysiert und bewertet wird. Nach der Auswertung erfolgt dann meist die Korrektur durch den Mitarbeiter. Die Verkürzung dieser Zeitspanne würde bedeuten, dass bereits während der Bearbeitung eines Artefakts – also beispielsweise während der Eingabe von Daten im Programm oder bei Modellierung eines Systems – die Konformität durch ein im Hintergrund ausgeführtes Regelwerk geprüft würde. Dies entspräche einer *reaktiven Konformitätsprüfung* und stellt eine erweiterte Möglichkeit der statischen Analyse für weiterführende Forschungsarbeiten in Aussicht. Hierbei müssten Regeln direkt nach Eingabe von Daten oder während der Modellierung algorithmisch im Hintergrund geprüft und das Ergebnis in Form eines Ereignisses zurück an das Werkzeug gegeben werden. Somit könnte dieser Mechanismus aktiv während der Dateneingabe, Modellierung usw. in den Entstehungsprozess eingreifen und ereignisbasierte Aktionen ausführen.
- *Kontrollierbarkeit und Steuerbarkeit (Workflows):* Kontrollierbarkeit und Steuerbarkeit bedingen eine geordnete Anwendung regelbasierter Konformitätsprüfung in einer Organisation. Dieser koordinative Charakter fördert das synchrone Zusammenarbeiten und ist allgemein als Prozess-Workflow bekannt. Ein *Workflow* wird typischerweise durch die geordnete Abfolge mit Parallelisierung der Arbeitsschritte erreicht. Synchrone Aktivitäten laufen strikt getrennt ab und beziehen sich auf Teile eines Geschäftsprozesses oder andere organisatorische Vorgänge. Die einzelnen Aktivitäten stehen demnach in Abhängigkeit zueinander. Wird nun die regelbasierte Konformitätsprüfung zwischen den Aktivitäten im Workflow verankert, ließen sich *Freigaben* von kollaborativen Artefakten für nachgelagerte Aktivitäten automatisiert durch Konformitätsprüfungen realisieren, in denen eine Freigabeentscheidung aufgrund der Güte eines kollaborativen Artefakts getroffen wird.
- *Anpassungsfähigkeit (Agilität):* Software für technische Systeme unterliegt ständiger Innovation und stellt besondere Anforderungen an Entwickler. Nach [HRH02] sind agile Prozesse hierbei ein Trend im Produktentstehungsprozess, der den Anspruch erhebt, komplexe Herausforderungen effizient durch Anpassungsfähigkeit zu meistern. Agilität in den Prozessen ersetzt starre Prozessmodelle, wie nach [VM97] das V-Modell, mit wenigen menschlichen Freiheitsgraden durch wenige, essenziell notwendige Aktivitäten zusammen mit einem vorhandenen Erfahrungsschatz. Eine resultierende Fragestellung ist, wie sich die Dynamik auf die regelbasierte Konformitätsprüfung in agilen Prozessen auswirkt, wo sich viele Änderungen pro Zeit in den Prozessanforderungen ergeben können.

9.3 Epilog

Der in dieser Arbeit entwickelte Ansatz stellt die werkzeuggestützte Methodik zur regelbasierten Konformitätsprüfung kollaborativer Artefakte vor und evaluiert diesen speziell im Kontext des frühen Entwicklungsprozesses eingebetteter Systeme. Die in diesem Bereich existierenden Probleme bei modellbasierter Entwicklung, am Beispiel der Automobilindustrie, wurden durch diesen wissenschaftlichen Beitrag gelöst.

Wird nun vom Anwendungsgebiet abstrahiert und eine etwas globalere Sicht eingenommen, stellt man fest, dass der Handlungsbedarf bei der Umsetzung von Konformitäts-anforderungen auch für andere Unternehmensbranchen und auch bei öffentlichen Institutionen (z. B. Behörden oder Krankenhäuser) anwächst − meist wird dies unter dem generellen Begriff *Compliance* für eine Organisation verstanden. Dabei beschränkt sich Compliance keinesfalls nur auf Richtlinien zum Anti-Korruptionsschutz oder Datenschutz. Vielmehr fordert Compliance von einer Organisation die Einhaltung aller relevanten Gesetze und Regelungen, der zugehörigen internen Unternehmensrichtlinien und Produktions-verfahren sowie der Selbstverpflichtungen überall in den Abteilungen. Die konforme Einhaltung behördlicher Vorschriften und interner Unternehmensrichtlinien ist ein wichtiges Thema für jede Organisation.

Im Laufe der Arbeit wurden in den Diplom-/Masterarbeiten [COCA08; JANS09] auch noch weitere Fallbeispiele aus anderen, nicht-technischen bzw. eher betriebswirtschaftlichen Kontexten geführt, was die Annahme zulässt, dass Richtlinien für kollaborative Artefakte auch in nicht-technischen Domänen existieren – in denen z. B. betriebswirtschaftlich motivierte Regelungen innerhalb der Organisation eingehalten werden müssen. Diese Vermutung eröffnet neue Herausforderungen sowie ein weiterführendes Forschungspotenzial für die regelbasierte Konformitätsprüfung kollaborativer Artefakte in einem anderen Kontext.

Literaturverzeichnis

[ADAC09] Allgemeiner Deutscher Automobil-Club e.V. (ADAC) (2009): *ADAC Pannenstatistik 2008*. Online verfügbar unter http://www1.adac.de/images/25797_tcm8-250130.pdf, zuletzt geprüft am 31.12.2009.

[ADL07] Erl, H.-P.; Kirstan, S. (2007): *Studie zur Kosten-/Nutzenanalyse der modellbasierten Software-Entwicklung im Automobil*. Herausgegeben von Arthur D. Little GmbH. Online verfügbar unter http://www.adlittle.de, zuletzt geprüft am 31.12.2009.

[ALEX07] Egyed, A. (2007): *UML/Analyzer - A Tool for the Instant Consistency Checking of UML Models*. In: Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, USA. Online verfügbar unter http://www.alexander-egyed.com/publications/UML_Analyzer-A_Tool_for_the_Instant_Consistency_Checking_of_UML_Models.pdf, zuletzt geprüft am 31.12.2009.

[ALT02] Altheide, F.; Dörr, H.; Schürr, A. (2002): *Requirements to a Framework for sustainable Integration of System Development Tools*. In: Proc. of 3rd European Systems Engineering Conference (EuSEC'02), Toulouse: AFIS PC Chairs, pp. 53-57.

[AMB05] Ambler, Scott W. (2005): *The elements of UML 2.0 style*. Cambridge Univ. Press. ISBN 978-0521616782.

[ANGE05] Angermann, A.; Beuschel, M.; Rau, M.; Wohlfarth, U. (2005): *Matlab - Simulink - Stateflow. Grundlagen, Toolboxen, Beispiele*. 4. überarb. Aufl. München: Oldenbourg. ISBN 3486577190.

[ANGE07] Angermann, A. (2007): *MATLAB - Simulink - Stateflow. Grundlagen, Toolboxen, Beispiele*. 5. aktualisierte Aufl. München: Oldenbourg. ISBN 3486582720.

[APPA06] AUTOSAR (2006): *Applying ASCET to AUTOSAR*. Online verfügbar unter http://www.autosar.org/download/specs_aktuell/AUTOSAR_ASCET_Styleguide.pdf, zuletzt geprüft am 31.12.2009.

[APSA06] AUTOSAR (2006): *Applying Simulink to AUTOSAR*. Online verfügbar http://www.autosar.org/download/AUTOSAR_SimulinkStyleguide.pdf.

[ARTI09] Artisan Studio: *Artisan Software Tools*. Online verfügbar unter http://www.artisansoftwaretools.com/products/artisan-studio, 31.12.2009.

[ASAM09] ASAM e.V.: *Association for Standardisation of Automation and Measuring Systems*. Verein zur Standardisierung von Automatisierungs- und Messsystemen. Online verfügbar unter http://www.asam.net.

[ASCT09] ASCET-SD/ASCET-MD: *Advanced Simulation and Control Engineering*. Version 6.1: ETAS GmbH. Online verfügbar unter http://www.etas.com/ascet, zuletzt geprüft am 31.12.2009.

[ASR09] AUTOSAR Konsortium: *AUTomotive Open System Architecture*. Online verfügbar unter http://www.autosar.org.

[BACH92] Bachmann, P. (1992): *Mathematische Grundlagen der Informatik*. Akad.-Verl. Berlin (Informatik, 6). ISBN 3-05-501342-5.

[BALZ01] Balzert, H. (2001): *Lehrbuch der Software-Technik*. Software-Entwicklung. 2. Aufl. Heidelberg: Spektrum Akad. Verl. (Lehrbücher der Informatik). ISBN 3827404800.

[BAU02] Bauer, H. (2002): *Autoelektrik, Autoelektronik. Sensoren/Mikroelektronik; Systeme und Komponenten*. 4. vollst. überarb. und erw. Aufl., Robert Bosch GmbH. Braunschweig: Vieweg Verlag. ISBN 3-528-13872-6.

[BAU03] Bauer, H.; Oder, M. (2003): *Ottomotor-Management*. 2. vollst. überarb. und erw. Aufl. Robert Bosch GmbH. Braunschweig: Vieweg Verlag (Kraftfahrzeugtechnik). ISBN 3-528-13877-7.

[BEND05] Bender, K. (2005): *Embedded Systems. Qualitätsorientierte Entwicklung*. Berlin, Springer. ISBN 3-540-22995-7.

[BESS06] Besstschastnich, A. (2006): *Dokumentenaustausch nach RIF Standard in DOORS*. Bachelorarbeit. Fachbereich Elektrotechnik und Informatik. Fachhochschule Münster. Online verfügbar unter http://www.lab4inf.fh-muenster.de/lab4inf/docs/thesis/BA_Besstschastnich.pdf, zuletzt geprüft am 31.12.2009.

[BIT05] BITKOM Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e. V. (Hrsg.): *Leitfaden Matrix der Haftungsrisiken sowie Leitfaden Compliance in IT-Outsourcing-Projekten*. Online verfügbar unter http://www.bitkom.org/files/documents/BITKOM-Leitfaden_ Compliance.pdf, zuletzt geprüft am 31.12.2009.

[BOIS06] DuBois, B.; Lange, C.; Demeyer, S.; Chaudron, M.: *A Qualitative Investigation of UML Modeling Conventions*. Workshops and Symposia at MoDELS 2006, Italy, October 1-6, 2006, Reports and Revised Selected Papers. Models in Software Engineering. In: Lecture Notes in Computer Science, Volume 4364/2007. ISBN: 978-3-540-69488-5, pp. 91–100.

[BORN05] Born, M.; Holz, E.; Kath, O. (2005): *Software-Entwicklung mit UML 2. Die neuen Entwurfstechniken UML 2, MOF 2 und MDA*. Unveränd. Nachdr. der Ausg. 2004. München: Addison-Wesley. ISBN 3-8273-2086-0.

[BRON01] Bronstein, I. N.; Semendjajew, K. A. (2001): *Taschenbuch der Mathematik*. 5. überarb. und erw. Aufl., Thun: Deutsch. ISBN 3-8171-2005-2.

[BROY04] Broy, M.; Steinbrüggen, R. (2004): *Modellbildung in der Informatik*. Berlin: Springer (Xpert.press). ISBN 3-540-44292-8.

[CABO08] Cabot, J.; Clariso R.; Riera D. (2008): *Verification of UML/OCL Class Diagrams using Constraint Programming*. In: Proc. of IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 73-80. ISBN 978-0-7695-3388-9.

[CABO09] Cabot, J.; Clarisó, R.; Riera, D. (2009): *Verifying UML/OCL Operation Contracts*. In: Proceedings of Integrated Formal Methods. 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Herausgegeben von David Hutchison und et al. Berlin, Heidelberg: Springer Berlin Heidelberg (Springer-11645/Dig. Serial], 5423). ISBN 978-3-642-00254-0.

[CAFF04] Caffall, D.; Michael, J. (Hrsg.) (2004): *A New Paradigm for Requirements Specification and Analysis of System-of-Systems*. Radical innovations of software and systems engineering in the future. 9th International Workshop, RISSEF 2002, Venice, Italy, October 7 - 11. 2002. Unter Mitarbeit von Martin Wirsing. Berlin: Springer (Reihe: Lecture notes in computer science, Band 2941). ISBN 978-3-540-21179-2.

[CLAR02] Clark, T. (2002): *Object modeling with the OCL. The rationale behind the object constraint language*. Berlin: Springer (Reihe: State-of-the-art survey, Band: 2263). ISBN 3540431691.

[CMMI12] CMMI - *Capability Maturity Model Integration*, Version 1.2.

[COCA08] Cocanu, L. (2008): *Measurement of LINQ-based quality metrics for system models*. Diplomarbeit, Faculty of Automatic Control and Computer Science, University Politehnica of Bucharest.

[CODE09] *CODECHECK - C/C++ Source Code Analysis* sowie *Coding Rules* (*Automating Corporate Source Code Compliance*): Abraxas Software. Online verfügbar unter http://www.abxsoft.com, zuletzt geprüft am 31.12.2009.

[CON04] Conrad, M. (2004): *Modell-basierter Test eingebetteter Software im Automobil. Auswahl und Beschreibung von Testszenarien*. Techn. Univ. Berlin, Dissertation, 1. Aufl. Wiesbaden: Dt. Univ.-Verl. (Informatik). ISBN 3-8244-2188-7.

[CON05] Conrad, M.; Dörr, H.; Fey, I.; Stürmer, I. (2005): *Guidelines und Modellreviews in der Modell-basierten Entwicklung von Steuergeräte-software*. In: Proc. of 2. Tagung Simulation und Test in der Funktions- und Software-Entwicklung für die Automobilelektronik, Berlin, 14.-15.03.2005.

[CON05+] Conrad, M.; Dörr, H.; Fey, I.; Pohlheim, H.; Stürmer, I. (2005): Guidelines und Reviews in der modell-basierten Entwicklung von Steuergeräte-Software. Simulation und Test in der Funktions- und Software-Entwicklung für die Automobilelektronik. Berlin: Expert-Verlag. Online verfügbar unter http://www.ichmaschine.de/Papers/SuT2005/Conrad_Doerr_Fey_Pohlheim_Stuermer_SuT2005.pdf, zuletzt geprüft am 31.12.2009.

[CON06] Conrad, M.; Dörr, H.; Fey, I.; Stürmer, I. (2006): *Using Model and Code Reviews in Model-based Development of ECU Software*. In: Proc. of SAE World Congress 2006, SAE 2006-01-1240, SAE International, 2006.

[CON06+] Mirko, C.; Heiko, D. (2006): *Einsatz von Modell-basierten Entwicklungstechniken in sicherheitsrelevanten Anwendungen. Herausforderungen und Lösungsansätze*. In: Proc. of Dagstuhl-Workshop: Model-Based Development of Embedded Systems II (MBEES), Informatik-Bericht, No. 2006-1, Technische Universität Braunschweig. Online verfügbar unter http://www.sse-tubs.de/publications/GRS_MBEES_InfoBericht_06.pdf, zuletzt geprüft am 31.12.2009.

[COSM09] *Cosmic Software MISRA CHECKER*: Cosmic Software. Online verfügbar unter http://www.cosmic-software.de, zuletzt geprüft am 31.12.2009.

[CRB30] *CORBA Component Model* (CCM) Specification. OMG Standard.

[CTES09] *Classification Tree Editor for Embedded Systems* (CTE/ES). Grafischer Editor für Klassifikationsbäume: Razorcat. Online verfügbar unter http://www.razorcat.de, zuletzt geprüft am 31.12.2009.

[CTEX09] *Classification Tree Editor* (CTE/XL). Grafischer Editor für Klassifikationsbäume. Version 1.9.x: Berner & Mattner. Online verfügbar unter http://www.berner-mattner.com, zuletzt geprüft am 31.12.2009.

[DAVI09] *DaVinci Tool Suite*: Vector Informatik GmbH. Online verfügbar unter http://www.vector.com/vi_davinci_developer_de.html, zuletzt geprüft am 31.12.2009.

[DEFS08] ASAAC Standards Part 1 (2008): *Standards for Software*: Ministry of Defence Defence, Standard 00-74.

[DIJK05] Van Dijk, H. W.; Graaf, B.; Boerman, R. (2005): *On the Systematic Conformance Check of Software Artefacts*. In: EWSA 2005, Springer-Verlag Berlin Heidelberg, LNCS 3527, pp. 203–221, 2005. Unter Mitarbeit von R. Morrison und F. Oquendo (Hrsg.).

[DIN09] Deutsches Institut für Normung (2009): *Qualitätsmanagement*. *QM-Systeme und -Verfahren*; Normen. 6. Aufl., Stand der abgedr. Normen: Februar 2009. Berlin: Beuth (DIN-Taschenbuch Qualität, Dienstleistungen, Management, 226). ISBN 9783410170075.

[DIN5007] DIN 5007 (2005): *Ordnen von Schriftzeichenfolgen*; Teil 1 und 2.

[DIN50126] DIN EN 50126 (2006): *Bahnanwendungen - Spezifikation und Nachweis der Zuverlässigkeit, Verfügbarkeit, Instandhaltbarkeit und Sicherheit* (RAMS).

[DIN50129] DIN EN 50129: *Bahnanwendungen - Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme, Sicherheitsrelevante elektronische Systeme für Signaltechnik.*

[DIN9000] Deutsches Institut für Normung: *Qualitätsmanagement DIN EN ISO 9000 ff. Dokumentensammlung*. Ausgabe 2009. (2009). Berlin: Beuth.

[DINE08] Dinesh, N.; Joshi, A. K.; Lee, I.; Sokolsky, O. (2008): *Logic-Based Regulatory Conformance Checking*. In: Monterey Workshop 2007, LNCS 5320, pp. 147–160, 2008. B. Paech and C. Martell (Hrsg.).

[DO178B] DO-178B (1992): *Software Considerations in Airborne Systems and Equipment Certification.* Standard zur Software-Entwicklung im sicherheitskritischen Bereich der Luftfahrt. RTCA Inc. Online verfügbar unter http://www.rtca.org, zuletzt geprüft am 31.12.2009.

[DOOR09] *Rational DOORS*. Software für das Anforderungsmanagement. Version 8.x: IBM. Online verfügbar unter http://www.ibm.com, zuletzt geprüft am 31.12.2009.

[DSPA05] dSpace GmbH (2005): *MTest. Systematic and automatic testing for Simulink and TargetLink models*. Online verfügbar unter http://www.dspace.de/shared/data/pdf/flyer2005/dspace-flyer2005 _mtest.pdf, zuletzt geprüft am 31.12.2009.

[DUFF07] Duffy, J.; Essey, E. (2007): *Running Queries On Multi-Core Processors*. In: msdn magazine. October 2007. Online verfügbar unter http://msdn.microsoft.com/en-us/magazine/cc163329.aspx, zuletzt geprüft am 31.12.2009.

[DYMO09] *Dymola. Multi-Engineering Modeling and Simulation*: DASSAULT SYSTEMES. Online verfügbar unter http://www.3ds.com/products/catia /portfolio/dymola, zuletzt geprüft am 31.12.2009.

[EAST08] EAST (2008): *ADL 2.0 Specification*. ATESST2 EU-Forschungsprojekt Konsortium, 2008. Online verfügbar unter http://www.atesst.org, zuletzt geprüft am 31.12.2005.

[ECKE09] Eckert, K. -P; Ziesing, J. H.; Ishionwu, U. (2009): *Interoperabilität von Dokumentenformaten: Open Document Format und Office Open XML*. FOKUS White Paper. Fraunhofer-Institut für Offene Kommunikations-systeme. Stuttgart: Fraunhofer Verlag. ISBN 978-3-8396-0030-6.

[ECKS04] Eckstein, R.; Eckstein, S. (2004): *XML und Datenmodellierung. XML-Schema und RDF zur Modellierung von Daten und Metadaten einsetzen*. 1. Aufl. Heidelberg: dpunkt-Verlag. ISBN 3-89864-222-4.

[EGUID09] *e-Guidelines. Guidelines for Model Based Development*: MES - Model Engineering Solutions e.K., Online verfügbar unter https://www.e-guidelines.de, zuletzt geprüft am 31.12.2009.

[EISE06] Eisemann, U. (2006): *Modellierungsrichtlinien für Funktionsentwicklung und Seriencode-Generierung*. Embedded World Conference 2006, 14. - 16. Februar, 2006, Nürnberg. Online verfügbar unter http://www.dspace.de/ ftp/papers/dspace_embWorld_0602_d_p345.pdf, geprüft am 31.12.2009.

[EKKA06] Kleinod, E. (2006): *Modellbasierte Systementwicklung in der Automobilindustrie. Das MOSES Projekt*. Fraunhofer Berichte. ISST Bericht 77/06. ISSN 09431624. Online verfügbar unter http://www.isst.fraunhofer.de/Images/isst-bericht_77-06_online_tcm81-17204.pdf, zuletzt geprüft am 31.12.2009.

[EMBV09] *Embedded Validator und Sate Mate Model Checker*: BTC Embedded Systems AG. Online verfügbar unter http://www.btc-es.de/ index.php?idcatside=5, zuletzt geprüft am 31.12.2009.

[ENGE07] Engert, S.; Philippow, I. (2007): *Test von generierten AUTOSAR Basis-Software Komponenten*. Techn. Univ. Ilmenau, Diplomarbeit.

[ETAS08] ETAS Competence Exchange Symposium (2008): *Focus ASCET & LABCAR*. Vorträge und persönliche Gespräche: ETAS GmbH. Online verfügbar unter www.etas.de.

[ETSB02] Etschberger, K. (2002): *Controller-Area-Network. Grundlagen, Protokolle, Bausteine, Anwendungen*. 3. aktualisierte Aufl. München: Hanser. ISBN 3-446-21776-2.

[FAGA02] Fagan, M. (2002): *A History of Software Inspections*. In: M. Broy, E. Denert (Hrsg.) Proc. Software Pioneers - Contributions to Software Engineering 2001, Bonn, Springer Verlag, Berlin.

[FKFS07] Forschungsinstitut für Kraftfahrwesen (FKFS): *Deutsche Automobilindustrie startet Innovationsallianz Automobilelektronik*. Pressemitteilung vom 12.12.2007. Online verfügbar unter http://www.bmbf.de/press/index.php, zuletzt geprüft am 31.12.2009.

[FLAW09] *Flawfinder*: David A. Wheeler. Online verfügbar unter http://www.dwheeler.com/flawfinder, zuletzt geprüft am 31.12.2009.

[FLEI08] Fleischmann, A. (2008): *Modellbasierte Formalisierung von Anforderungen für eingebettete Systeme im Automotive-Bereich*. Techn. Univ. München, Dissertation, 1. Aufl. München: GRIN-Verl. ISBN 978-3-640-17496-6.

[FS08] Frost & Sullivan (2008): *Strategic Analysis of the European Automotive Advanced Electronic Controller Markets*. Nr. M1C6-18. Online verfügbar unter http://www.frost.com, zuletzt geprüft am 31.12.2009.

[GAR07] Gartner Inc. (2007): *'Dirty Data' is a Business Problem, Not an IT Problem, Says Gartner*. Press Release. Online verfügbar unter http://www.gartner.com/it/page.jsp?id=501733, zuletzt geprüft am 31.12.2009.

[GAR10] Gartner Inc. (2010): *Magic Quadrant for Data Quality Tools*. Gartner RAS Core Research Note G00200603. Online verfügbar unter http://www.gartner.com/technology/media-products/reprints/trillium/200603.html, zuletzt geprüft am 31.12.2009.

[GEEN09] *AUTOSAR Builder*: GEENSYS. Online unter http://www.geensys.com/?Outils/AutosarBuilder, zuletzt geprüft am 31.12.2009.

[GIES06] Giese, H.; R. Wagner (2006): *Incremental Model Synchronization with Triple Graph Grammars*. In: Proc. of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genoa, Italy, LNCS 4199, Springer Verlag, pp. 543-557.

[GNUO09] *GNU Octave*: University of Wisconsin. Online verfügbar unter http://www.gnu.org/software/octave, zuletzt geprüft am 31.12.2009.

[GOTT06] Gottschalk, B. (2006): Mastering the Automotive Challenges. München: SV Corporate Media. ISBN 3-937889-20-5.

[GRAM08] Grammatikopoulou, K. (2008): *Compliance, Compliance-Manager, Compliance-Programme: Eine geeignete Reaktion auf gestiegene Haftungsrisiken für Unternehmen und Management*. Herausgegeben von Grin Verlag. München. ISBN: 978-3640141159.

[GRO93] Grochtmann, M.; Grimm, K. (1993): *Classification Trees for Partition Testing*. Software Testing, Verification & Reliability, Volume 3, No. 2, pp. 63-82.

[GROS00] Grosof, B. N.; Labro, Y. (2000): *An Approach to Using XML and a Rule-Based Content Language with an Agent Communication Language*. Agent Communication, Springer-Verlag Berlin Heidelberg, LNAI 1916, pp. 96-117, 2000. Herausgegeben von F. Dignum and M. Greaves (Eds.).

[GRÜN09] Grünendahl, R.-T.; Steinbacher, A. F.; Will, P. H. L. (2009): *Das IT-Gesetz: Compliance in der IT-Sicherheit. Leitfaden für ein Regelwerk zur IT-Sicherheit im Unternehmen*. 1. Aufl. Wiesbaden: Vieweg+Teubner Verlag / GWV Fachverlage GmbH Wiesbaden (Springer-11774/Dig. Serial). ISBN 383480598X.

[HAGE06] Hagelstein, J.; Roelants, D.; Wodon, P. (Hg.) (1993): *Formal requirements made practical*. In: Proc. of 4th European Software Engineering Conference Garmisch-Partenkirchen, Germany September 13-17, 1993. Software Engineering ESEC '93. Unter Mitarbeit von Ian Sommerville. Berlin, Heidelberg, Springer (Lecture notes in computer science, 717). ISBN 978-3-540-57209-1.

[HIND09] Hindawi, M.; Régis, L. M.; Sourrouille, A. J. -L (2009): *Description and Implementation of a UML Style Guide*. In: Models in software engineering. Workshops and symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008; reports and revised selected papers. Herausgegeben von Michel R. V. Chaudron. International Conference on Model Driven Engineering Languages and Systems. Berlin: Springer (Springer-11645 /Dig. Serial, 5421). ISBN 978-3-642-01647-9.

[HIS09] *Herstellerinitiative Software* (HIS). Online verfügbar unter http://www.automotive-his.de, zuletzt geprüft am 31.12.2009.

[HNAT07] Hnatkowska, B. (2007): *Verification of Good Design Style of UML Models*. In: Proc. of the 10th International Conference on Information System Implementation and Modeling. (ISIM '07) 10th International Conference on Information System Implementation and Modeling. Herausgegeben von Jaroslav Zendulka. Online verfügbar unter http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-252/paper10.pdf, zuletzt geprüft am 31.12.2009.

[HOER06] Hörmann, K. (2006): *SPICE in der Praxis. Interpretationshilfe für Anwender und Assessoren basierend auf ISO/IEC 15504*. 1. Aufl. Heidelberg: dpunkt-Verlag. ISBN 3-89864-341-7.

[HOOD09] *DESIRe* (*Dynamic Expert System for Improving Requirements*): HOOD GmbH. Online verfügbar unter http://www.desire.hood-group.com, zuletzt geprüft am 31.12.2009.

[HRH02] Hruschka, P.; Rupp, C. (2002): *Agile Software-Entwicklung für embedded real-time systems mit der UML*. München: Hanser. ISBN 3-446-21997-8.

[IEC615] IEC 61508 Standard: *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbar elektronischer Systeme.*

[IKV09] *medini solutions*: IKV++ Technologies AG. Repository-basierte Infrastruktur. Online verfügbar unter http://www.ikv.de, zuletzt geprüft am 31.12.2009.

[IMMOS06] IMMOS (2004-2006): *Integrated Method for the Model-based Development of Automotive Control Units*. Programm IT-Forschung 2006 des BMBF. Online verfügbar unter http://www.immos-project.de, zuletzt geprüft am 31.12.2009.

[INRI09] *Scilab*: INRIA. Online verfügbar unter http://www.scilab.org, zuletzt geprüft am 31.12.2009.

[ISO9000] ISO 9000:2005 - *Quality management systems - Fundamentals and vocabulary*.

[ISO9001] ISO 9001:2005 - *Anforderungen an ein Qualitätsmanagementsystem.*

[ISO9003] ISO/IEC 9003 - *Software- und Systemtechnik - Richtlinien für die Anwendung der ISO 9001:2000 auf Computersoftware*.

[ISO9075] ISO/IEC 9075-1:2008 - *Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)*.

[ISO9126] ISO/IEC TR 9126-4:2004 - *Software engineering - Product quality*.

[ISO15489] ISO 15489-1:2001 - *Information and documentation. Records management*.

[ISO15504] ISO/IEC 15504 – SPICE - *Software Process Improvement and Capability Determination*.

[ISO17000] EN ISO/IEC 17000:2004 – *Konformitätsbewertung. Begriffe und allgemeine Grundlagen*.

[ISO26262] ISO 26262 - *Road vehicles - Functional safety.*

[ISO26300] ISO/IEC 26300:2006 - *Information technology - Open Document Format for Office Applications (OpenDocument)*, Version 1.0.

[JANS09] Jansen, H. C. (2009): *Evaluation von Language Integrated Queries am Beispiel eines Informationssystems zur einheitlichen Abbildung von verteilten Daten im Kontext eines Zulieferer-Entwicklungsprozesses*. Bachelorarbeit, Institut Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster.

[JMAAB09] Japan MATLAB Automotive Advisory Board (JMAAB): *Plant Modeling Guidelines using MATLAB/Simulink*. Version 2.1. Online verfügbar unter http://jmaab.mathworks.jp, zuletzt geprüft am 31.12.2009.

[KAPE06] Kapeller, R.; Vugt, B. (Hrsg.) (2006): Systeme wiederverwendbar in natürlicher Sprache spezifizieren. Informatik 2006: Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI); 2. bis 6. Oktober 2006 in Dresden. Bonn: Ges. für Informatik (GI-EditionProceedings, 93). ISSN 978-3-88579-187-4.

[KEMP07] Kemper, A. (2007): *Datenbanksysteme in Business, Technologie und Web* (*BTW*). Beiträge der 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 07.-09.03.07 in Aachen. Bonn: Ges. für Informatik (GI-Edition Proceedings, Vol. 103). ISBN 978-3-88579-197-4.

[KEMP99] Kemper, A.; Eickler, A. (1999): *Datenbanksysteme. Eine Einführung*. 3. korr. Aufl. München: Oldenbourg. ISBN 3-486-25053-1.

[KLEIN07] Kleinwechter, H. (2007): *Automotive Software Engineering*. Seminar am Institut für Softwaretechnik und Theoretische Informatik (SWT), SS 07, Techn. Univ. Berlin. Modellbasierte Funktionsentwicklung. Kapitel: *Methodik - Modellierungsrichtlinien*.

[KNAPP07] Knapp, A.; Merz, S. (2002): *Model Checking and Code Generation for UML State Machines and Collaborations*. In: Dominik Haneberg, Gerhard Schellhorn, and Wolfgang Reif (Eds.), Proc. Of 5th Wsh. Tools for System Design and Verification, pp. 59-64. Technical Report 2002-11, Institut für Informatik, Universität Augsburg. Online unter http://www.pst.ifi.lmu.de /veroeffentlichungen/knapp-merz:2002.pdf, zuletzt geprüft am 31.12.2009.

[KNEU06] Kneuper, R. (2006): *CMMI. Verbesserung von Softwareprozessen mit Capability Maturity Model Integration*. 2. überarb. und erw. Aufl. Heidelberg: dpunkt-Verl. ISBN 3-89864-373-5.

[KRAN09] Kranawetter, M. (2009): *Nutzenpotentiale regulatorischer Anforderungen zur Geschäftsoptimierung*. Herausgegeben von Microsoft Deutschland GmbH. Online verfügbar unter http://download.microsoft.com/download/ D/5/8/D58EEC38-FBAC-42BC-9C3C-C88C042103DE/IT_Infrastruktur _Compliance_Reifegradmodell_Microsoft_Kranawetter.pdf, zuletzt geprüft am 31.12.2009.

[KRUP05] Krupp, A.; Müller, W. (2005): *Die Klassifikationsbaummethode für eingebettete Systeme mit Testmustern für nichtkontinuierliche Regler-elemente*. In: Informatik 2005 - Informatik live!: Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 19.- 22. September 2005 in Bonn. Gesellschaft für Informatik. Bonn: Ges. für Informatik (GI-Edition Proceedings, 68). ISBN 3-88579-397-0.

[KRUP06] Conrad, M.; Krupp, A. (2006): *An Extension of the Classification-Tree Method for Embedded Systems for the Description of Events*. In: Proc. of Second Workshop on Model Based Testing, MBT 2006, Vienna, Austria, 2006.

[LANG04] Lange, C.; Chaudron, M. (2004): *Konsistenz und Vollständigkeit industrieller UML Modelle*. In: Proceedings zur Tagung Modellierung 2004, Marburg, Germany. Herausgegeben von Bernhard Rumpe. Tagung Modellierung. Bonn: Ges. für Informatik (GI-Edition Proceedings, 45). ISBN 3885793741.

[LDRA09] *LDRA tool suite. MISRA-C++:2008 Conformance Checker*: LDRA Ltd. Online verfügbar unter http://www.ldra.com/misracpp.asp, zuletzt geprüft am 31.12.2009.

[LEDE04] Lederer, D.; Heling, G.; Fetzer, J.; Beck, T. (2004): *Requirements-Management - Glücksspiel oder systematischer Engineering-Prozess?* Fachartikel Automotive Engineering Partners. Vector Informatik/ Consulting GmbH, 2004. Online verfügbar unter https://www.vector.com /vc_download_de.html?product=consulting, zuletzt geprüft am 31.12.2009.

[LHFB02] Lederer, D.; Heling, G.; Fetzer, J.; Beck, T. (2002): *Der Schlüssel zum Erfolg: Durchgängige Systems-Engineering-Prozesse - eine vordringliche Aufgabe*. (Nr. 3). Online unter http://www.vector-worldwide.com /portal/medien/cmc/press/PCO/AutomotiveSystemsEngineering_Elektronik Automotive_200206_PressArticle_DE.pdf, zuletzt geprüft am 31.12.2009.

[LIEB96] Liebig, H.; Thome, S. (1996): *Logischer Entwurf digitaler Systeme*. 3. vollst. neubearb. Aufl. Berlin: Springer. ISBN 3-540-61062-6.

[LIEBL00] Liebel, J. (2007): *EfficientDynamics - Der BMW Weg zur CO2-Reduzierung*. Elektronik im Kraftfahrzeug; Elektronik 2007 Baden-Baden; 13. internationaler Kongress; Tagung Baden-Baden, 10. und 11. Oktober 2007, CD-ROM.

[LIGG02] Liggesmeyer, P. (2002): *Software-Qualität. Testen, Analysieren und Verifizieren von Software*. Heidelberg: Spektrum Akademischer Verlag. ISBN 3-8274-1118-1.

[LIGG05] Liggesmeyer, P. (2005): *Software Engineering eingebetteter Systeme. Grundlagen - Methodik - Anwendungen*. 1. Aufl. München: Elsevier Spektrum Akademischer Verlag. ISBN 3827415330.

[LIM08] Lim, M.; Sadeghipour, S. (2008): *Werkzeugunterstützte Verknüpfung von Anforderungen und Tests – Voraussetzung für eine systematische Qualitätssicherung*. GI-Fachgruppe Requirements Engineering (RE) und Test, Analyse & Verifikation von Software (TAV), Treffen 5.-6. Juni, Bad Honnef.

[LINQ07] *Language-Integrated Query* (LINQ): Part of .NET Framework 3.5. Herausgegeben von Microsoft Corporation in 2007. Online verfügbar unter http://www.microsoft.com, zuletzt geprüft am 31.12.2009.

[LOUI09] Louis, D.; Strasser, S. (2009): *Microsoft Visual C 2008 - das Entwicklerbuch*. Unterschleißheim: Microsoft Press (Entwicklerbuch).

[LUNZ08] Lunze, J. (2008): *Regelungstechnik 1. Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen*. 7. neu bearb. Aufl. Berlin, Heidelberg: Springer-Verlag. ISBN 3540689079.

[MAAB09] *MAAB Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*, Version 2.1. Online verf. unter http://www.mathworks.de /automotive/standards/maab.html

[MACK07] Mackh, M. (2007): *MISRA-Konformität von ASCET-Autocode. Unterstützung der Automobilindustrie durch Entwicklung und Anwendung von sicherer und zuverlässiger Software*. Herausgegeben von ETAS GmbH. Online verfügbar unter http://www.etas.com/data/RealTimes_2007 /rt_2007_01_18_de.pdf, zuletzt geprüft am 31.12.2009.

[MDA00] *Model Driven Architecture*. OMG Standard. Online verfügbar unter http://www.omg.org/mda, zuletzt geprüft am 31.12.2009.

[MARG08] Marguerie, F.; Eichert, S.; Wooley, J. (2008): *Linq in action*. Greenwich, CT: Manning. ISBN 1933988169.

[MARP08] Marpons, G.; Marino, J.; Carro, M.; Herranz, A.; et al. (2008): *Automatic Coding Rule Conformance Checking Using Logic Programming*. In: PADL 2008, Springer-Verlag Berlin Heidelberg, LNCS 4902, pp. 18-34, 2008. Herausgegeben von P. Hudak and D.S. Warren (Eds.).

[MATE07] Stürmer, I.; Kreuz I.; Schäfer W.; Schürr A. (2007): *The MATE Approach: Enhanced Simulink and Statfelow Model Transformation*. In: Proc. of MathWorks Automotive Conference (MAC 2007), Dearborn (MI), USA.

[MATE07+] Stürmer, I.; Dörr, H.; Giese, H.; Kelter, U.; Schürr, A.; Zündorf, A. (2007): *Das MATE Projekt. Visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen*. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III. Informatik- Bericht 2007-01, Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik, Techn. Univ. Braunschweig, 2007. Herausgegeben von H. Giese B. Rumpe B. Schätz M. Conrad. Online verfügbar unter http://www.sse-tubs.de/publications/CGRS_MBEES_InfoBericht_07.pdf, zuletzt geprüft am 31.12.2009.

[MATR09] *MATRIXx*: National Instruments Corporation. Online verfügbar unter http://www.ni.com/matrixx, zuletzt geprüft am 31.12.2009.

[MINT09] *mint*. Simulink und Stateflow Style-Checker: Ricardo UK Limited. Online verfügbar unter http://www.ricardo.com/en-gb/Engineering-Consulting /Automotive-Expertise/Controls--Electronics/Software-Tools/Mint, zuletzt geprüft am 31.12.2009.

[MISRA09] MISRA AC GMG: *Generic modeling design and style guidelines.*

[MISRA09+] MISRA AC SLSF: *Modeling design and style guidelines for the application of Simulink and Stateflow.*

[MLSL09] *MATLAB/Simulink/Stateflow* (2009b). *Software zur Lösung mathematischer Probleme und zur grafischen Darstellung der Ergebnisse*: The MathWorks. Online verfügbar unter http://www.mathworks.com/products/matlab, zuletzt geprüft am 31.12.2009.

[MODA09] *Model Advisor*. Integriertes Prüfwerkzeug in MATLAB/Simulink: The MathWorks. Online verfügbar unter http://www.mathworks.com/company/newsletters/news_notes/jan06/tips.html, zuletzt geprüft am 31.12.2009.

[MODE09] *Model Examiner. Simulink und Targetlink Style-Checker*: MES -Model Engineering Solutions GmbH. Online verfügbar unter http://www.model-engineers.com, zuletzt geprüft am 31.12.2009.

[MODL09] *Modelica*. Objektorientierte Beschreibungssprache für physikalische Modelle von der Modelica Association. Online verfügbar unter http://www.modelica.org, zuletzt geprüft am 31.12.2009.

[MOF20] *Meta Object Facility* (MOF), Specification, Version 2.0. Online verfügbar unter http://www.omg.org/mof, zuletzt geprüft am 31.12.2009.

[MONO09] *Mono Project. The open source development platform based on the .NET framework*. Online verfügbar unter http://www.mono-project.com, zuletzt geprüft am 31.12.2009.

[MSOF07] *MS Office*. Werkzeugsammlung für Textverarbeitung (Word), Tabellenkalkulation (Excel), Präsentationstechnik (Powerpoint) und andere. Version 2003-2007: Microsoft Corporation. Redmond, USA. Online verfügbar unter http://office.microsoft.com, zuletzt geprüft am 31.12.2009.

[MRTE08] OMG - Object Management Group, Inc (2008): *Modeling and Analysis of Real-time and Embedded Systems* (MARTE). Beta 2, OMG Adopted Specification. OMG - Object Management Group, Inc, 2008. Online verfügbar unter http://www.omgmarte.org/specification.htm, zuletzt geprüft am 31.12.2009.

[MUEL07] Müller, M.; Hörmann, K.; Dittmann, L.; Zimmer, J. (2007): *Automotive SPiCE. Interpretationshilfe für Anwender und Assessoren*. 1. Aufl. Heidelberg: dpunkt-Verlag. ISBN 978-3898644693.

[MUTZ05] Mutz, Martin (2005): *Eine durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich*. Techn. Univ., Diss. Braunschweig, 2005. 1. Aufl. Göttingen: Cuvillier. ISBN 3-86537-684-3.

[MUTZ05+] Mutz, M.; Daginnus, M.; Hofmann, P. -M; Klein, T.; Kleinwechter, H. (2005): *Ein Richtlinienkatalog für die Erstellung von Simulink/ Stateflow- Modellen im Automobilbereich*. In: GI Jahrestagung (Band: 2), pp. 181-185. Online verfügbar unter http://subs.emis.de/LNI/ Proceedings/Proceedings68/GI-Proceedings.68-38.pdf, am 31.12.2009.

[NARD00] Nardelli, N. X. (2000): *Entwicklung eines Testkonzepts für parametrisierbare Simulationsmodelle mechanischer, hydraulischer und regelungstechnischer Systeme*. Univ. Stuttgart, Diplomarbeit.

[NASA07] NASA: *Procedure for Developing and Implementing Software Quality Programs*. Directive No. 303-PG-7120.2.1C. Online verfügbar unter http://sw-assurance.gsfc.nasa.gov/disciplines/quality/documents/doc/303-PG-7120_2C.doc, zuletzt geprüft am 31.12.2009.

[NEEM01] Neema, S. (2001): *Analysis of Matlab Simulink and Stateflow Data Model*. Technical Report, TR# ISIS-01-204. Institute for Software Integrated Systems, Vanderbilt University.

[OCL20] *Object Constraint Language* (OCL), Specification, Version 2.0. OMG - Object Management Group, Inc. Online verfügbar unter http://www.omg.org/spec/OCL/2.0, zuletzt geprüft am 31.12.2009.

[OEST04] Oestereich, B. (2004): *Objektorientierte Software-Entwicklung. Analyse und Design mit der UML 2.0*. 6. völlig überarb. Aufl. München: Oldenbourg. ISBN 3-486-27266-7.

[OHA09] Ohata, A.; Komori, S. (2009): *JMAAB Plant Modeling Guidelines and Vehicle Architecture*. In: Proc. of ICROS-SICE International Joint Conference 2009. Fukuoka International Congress Center, Japan.

[OOXML06] *Ecma International* (2006): Office Open XML. Part 2: Open Packaging Conventions. pp. 36-40.

[OSEK09] *OSEK-OS*. Spezifikation für Echtzeitbetriebssysteme für eingebettete Systeme. Herausgegeben von OSEK Gremium. Online verfügbar unter http://www.osek-vdx.org.

[OSLO09] *Open Source Library for OCL* (OSLO): Jetzt unter Eclipse Ganymede und EMF 2.4. Online verfügbar unter http://oslo-project.berlios.de, zuletzt geprüft am 31.12.2009.

[PAPM01] Pap, Z.; Majzik, I.; Pataricza, A. (2001): *Checking General Safety Criteria on UML Statecharts*. In: Proc. of the 20th international conference on Computer safety, reliability and security. Budapest, Hungary (Elektronische Ressource). SAFECOMP. Berlin: Springer (Lecture notes in computer science, 2187). Online verf. Unter http://eprints.kfupm.edu.sa /29653/1/29653.pdf, zuletzt geprüft am 31.12.2009.

[PCLi09] *PC-lint for C/C++*: Gimpel Software. Online verfügbar unter www.gimpel.com, zuletzt geprüft am 31.12.2009.

[PEPP00] Pepper, P. (2000): *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. 1. korrigierter Nachdr. Berlin: Springer (Springer-Lehrbuch). ISBN 3-540-64541-1.

[PEPP95] Pepper, P. (1995): *Grundlagen der Informatik*. 2. verb. Aufl. München: Oldenbourg. ISBN 3-486-23513-3.

[PETRA06] Petrasch, R.; Meimberg, O. (2006): *Model Driven Architecture. Eine praxisorientierte Einführung in die MDA*. 1. Aufl. Heidelberg: dpunkt-Verlag. ISBN 978-3-89864-343-6.

[PIAL07] Pialorsi, P.; Russo, M. (2007): *Microsoft LINQ Crashkurs*. Unterschleißheim: Microsoft Press. ISBN 978-3-86645-510-8.

[PINN09] Pinnow, K. (2009): *AUTOSAR-Software mit ASCET*. TU Darmstadt, Industrie Kolloquium. Online verfügbar unter http://www.es.tu-darmstadt.de/fileadmin/ download/lehre/ik/AUTOSAR_ETAS.Kolloquium .Darmstadt.2009.pdf, zuletzt geprüft am 31.12.2009.

[QACM09] *QA-C/MISRA*: QA Systems GmbH. Online verfügbar unter http://www.qasystems.de/html/deutsch/produkte/qacm/qacm.php, zuletzt geprüft am 31.12.2009.

[QVT10] *MOF Query / Views / Transformations* (QVT), Specification, Version 1.0. OMG - Object Management Group, Inc. Online verfügbar unter http://www.omg.org/spec/QVT/1.0, zuletzt geprüft am 31.12.2009.

[RATA09] *Rational Rhapsody*. Modellbasierte Entwicklung mit UML 2.1 und SysML. Version 7.x: IBM. Online verfügbar unter http://www.ibm.com, zuletzt geprüft am 31.12.2009.

[RATI09] *IBM Rational System Architect Suite*: IBM. Online verfügbar unter http://www.ibm.com/software/awdtools/systemarchitect, am 31.12.2009.

[RATR09] *Rational Rose Modeler*: IBM. Online verfügbar unter http://www-142 .ibm.com/software/products/de/de/rosemod, zuletzt geprüft am 31.12.2009.

[RATS09] *Rough Auditing Tool for Security*: Fortify Software Inc. Online verfügbar unter http://www.fortify.com/security-resources/rats.jsp, zuletzt geprüft am 31.12.2009.

[REIF06] Reif, K. (2006): *Automobilelektronik. Eine Einführung für Ingenieure*. 1. Aufl. Wiesbaden: Vieweg (ATZ/MTZ-Fachbuch). Online verfügbar unter http://www.gbv.de/dms/bs/toc/477080790.pdf. ISBN 3-528-03985-X.

[REISS02] Reißing, R. (2002): *Bewertung der Qualität objektorientierter Entwürfe*. Univ. Stuttgart, Diss. 1. Aufl. Göttingen: Cuvillier. ISBN 978-3898735452.

[RICH06] Richter, J. (2006): *Microsoft .NET Framework-Programmierung. Expertenwissen zur CLR und dem .NET Framework 2.0*. 2. Aufl. Unterschleißheim: Microsoft Press (Fachbibliothek). ISBN 3860639846.

[RIF05] Herstellerinitiative Software (HIS) (2007): *Requirements Interchange Format* (RIF). Standardisiertes Austauschformat für Anforderungen. RIF Version 1.1a. Online verfügbar unter http://www.automotive-his.de/rif, zuletzt geprüft am 31.12.2009.

[RUSS06] Russom, P. (2006): *Master Data Management. Consensus-Driven Data Definitions for Cross-Application Consistency*. Herausgegeben von The Data Warehousing Institute (TDWI). Online verfügbar unter http://download.101com.com/pub/tdwi/Files/TDWI_MDM_Report_Q406REVISED.pdf, zuletzt geprüft am 31.12.2009.

[SABE09] *Saber*: Synopsys, Inc. Online verfügbar unter http://www.synopsys.com/Tools/SLD/Mechatronics/Saber/Pages/default.aspx, am 31.12.2009.

[SCAD09] *SCADE Suite*. Werkzeug zur modellgetriebenen Software-Entwicklung: Esterel Technologies Ltd. Online verfügbar unter http://www.esterel-technologies.com/products/scade-suite, zuletzt geprüft am 31.12.1009.

[SCHÄ03] Schäuffele, J.; Zurawka, T.; Roger, C. (2005): *Automotive Software Engineering. Principles, processes, methods, and tools*. Warrendale, Pa.: SAE International (Reihe: SAE-R, Band: 361). ISBN 0-7680-1490-5.

[SCHI09] Schieferdecker, I. (2009): Neuer Standard „TTCN-3 embedded“ für bessere Softwarequalität im Fahrzeug. Herausgegeben von ELEKTRONIK PRAXIS. Vogel Business Media GmbH & Co. KG. Online verfügbar unter http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/testinstallation/articles/235470, zuletzt aktualisiert am 23.10.2009, zuletzt geprüft am 31.12.2009.

[SCHI09+] Schieferdecker, I. (2009); Vouffo-Feudjio, A.; Rennoch, A.; Großmann, J.; Wendland, M. F.; Hoffmann, A.: Research on Trends in Model-based Testing (RMBT-Study). Fraunhofer FOKUS, Project internal document, RMBT-Study_v1.1_2009-08-17.doc.

[SCHIN09] Schinz, I.; Toben, T.; Mrugalla, C.; Westphal, B. (2004): The Rhapsody UML Verification Environment. (VIS model checker). In: Proc. of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Peking, China, September 2004, IEEE. Online verfügbar http://www-omega.imag.fr/doc/d1000312_1/WP22-312-V1-ruve2004.pdf, zuletzt geprüft am 31.12.2009.

[SCHN06] Schneider, M. (2006): *Integration of Model-Driven Techniques in Automotive Software-Development*. Diplomarbeit, Tech. Univ. Berlin.

[SCHN91] Schneider, H. J. (1991): *Lexikon der Informatik und Datenverarbeitung*. 3. aktualisierte und wesentlich erw. Aufl. München: Oldenbourg. ISBN 3-486-21514-0.

[SIME09] *SimEx*. XML-Konverter von Simulink-Modellen: IT Power Consultants. Online verfügbar unter http://www.itpower.de/30-0-Download-MEval-und-SimEx.html, zuletzt geprüft am 31.12.2009.

[SIMX09] *SimulationX*: ITI GmbH. Online verfügbar unter http://www.iti.de/de/simulationx.html, zuletzt geprüft am 31.12.2009.

[SOMM07] Sommerhalder, M. (2007): *Design Guidelines und Erfahrungsberichte in MDD/MDA*. Institut für Informatik der Universität Zürich. Online unter http://seal.ifi.uzh.ch/fileadmin/User_Filemount/Vorlesungs_Folien/Seminar_SE/SS07/SemSE07-Michael_Sommerhalder.pdf, geprüft am 31.12.2009.

[SPAX09] *Enterprise Architect*. UML Modellierung mit Round Trip Engineering. SparxSystems Ltd. Online verfügbar unter http://www.sparxsystems.com, zuletzt geprüft am 31.12.2009.

[SQLR08] Software Quality Lab GmbH (2008): *Requirements-Spezifikation. Guideline für eine erfolgreiche Projektabwicklung*. Online verfügbar unter http://www.software-quality-lab.at/swql/uploads/media/SWQL-Guideline_-_Requirements-Spezifikation_-_2008_05_24.pdf, zuletzt geprüft am 31.12.2009.

[STAE05] Stärk, R. (2005): *Logik*. Skript im WS 04/05. Institut für Theoretische Informatik der ETH Zürich. Online verfügbar unter http://www.inf.ethz.ch/~staerk/, zuletzt geprüft am 31.12.2009.

[STAT09] *Statemate Design Checker*: Quality Park GmbH. Online verfügbar unter http://www.qualitypark.de/docs/Statemate_Flyer_0306.pdf, zuletzt geprüft am 31.12.2009.

[STEIN09] Steinberg, D.; Budinsky, F.; Paternostro, M. (2009): *EMF. Eclipse modeling framework*. 2nd ed., rev. and updated. Herausgegeben von Merks. Upper Saddle River, NJ: Addison-Wesley (The eclipse series). ISBN 0-321-33188-5.

[STRA09] Strano, M.; Molina-Jimenez, C.; Shrivastava, S. (2009): *Implementing a Rule-Based Contract Compliance Checker*. University of Newcastle upon Tyne: Computing Science, Technical Report, Series No. CS-TR-1150, April, 2009.

[STUE09] Stürmer, I.; Stamatov S.; Eisemann U. (2009): Automated Checking of MISRA TargetLink and AUTOSAR Guidelines. In Proceedings of SAE World Congress 2009, SAE Doc. #2009-01-0267. Detroit (USA).

[SYSC09] *SystemC-AMS*. Modellierungs- und Simulationssprache insbesondere für die Entwicklung von komplexen elektronischen Systemen: SystemC-AMS Working Group. Online verfügbar unter http://www.systemc-ams.org, zuletzt geprüft am 31.12.2009.

[SYSD09] *SystemDesk*. Werkzeug zur Entwicklung komplexer Systemarchitekturen: dSpace GmbH. Online verfügbar unter http://www.dspace.de/ww/de/gmb/home/products/sw/system_architecture_software/systemdesk.cfm, zuletzt geprüft am 31.12.2009.

[SYSML11] *Systems Modeling Language* (SysML), Specification, Version 1.1. OMG - Object Management Group, Inc, 2008. Online verfügbar unter http://www.omg.org/spec/SysML/1.1, zuletzt geprüft am 31.12.2009.

[SYST09] *SystemVision*: Mentor Graphics, Inc. Online verfügbar unter http://www.mentor.com/products/sm/system_integration_simulation_analysis/systemvision, zuletzt geprüft am 31.12.2009.

[TGTL09] *TargetLink*: dSpace GmbH. Online verfügbar unter http://www.dspace.de /ww/de/gmb/home/products/sw/pcgs/targetli.cfm, geprüft am 31.12.2009.

[TKL06] Klein, T. (2006): *Modellbasierte Funktionsentwicklung für Komfortsteuergeräte*. *Vorgehensweise, Ergebnisse und Potenziale*. Internationale Zuliefererbörse, Wolfsburg. Online unter http://www.carmeq.de/media/ Modellbasierte_Entwicklung_von_Komfortsteuergeraeten.pdf, 31.12.2009.

[TOGE09] *Together*. Visual Modeling for Software Architecture Design: Borland, Inc. Online verfügbar unter http://www.borland.com/us/products/together /index.html, zuletzt geprüft am 31.12.2009.

[UML22] UML - *Unified Modeling Language, Infrastructure and Superstructure specifications*, Version 2.2. OMG - Object Management Group, Inc, 2009. Online verfügbar unter http://www.omg.org/spec/UML/2.2, zuletzt geprüft am 31.12.2009.

[UMLT09] UMLtoCSP: *A tool for the formal verification of UML/OCL models using Constraint Programming* (2009). GRES-UOC research group within the Internet Interdisciplinary Institute (IN3) of the Open University of Catalonia. Online http://gres.uoc.edu/UMLtoCSP, geprüft am 31.12.2009.

[UMTQ09] UMT-QVT: *Werkzeug für Modelltransformation und Codegenerierung von UML/XMI Modellen*. Online verfügbar unter http://umt-qvt.sourceforge.net, zuletzt geprüft am 31.12.2009.

[USE09] Gogolla, M.; Richters, M. (2009): *USE - A UML-based Specification Environment*. Online verfügbar unter http://www.db.informatik.uni-bremen.de/projects/USE/#sysreq, zuletzt geprüft am 31.120.2009.

[VAMS09] *Verilog-AMS*: Verilog-AMS Committee (Hrsg.). Online verfügbar unter http://www.vhdl.org/verilog-ams, zuletzt geprüft am 31.12.2009.

[VDA09] Verband der Automobilindustrie e.V. (2009): *Jahresbericht 2009*. ISSN: 0171-4317. Online verfügbar unter http://www.vda.de/de/downloads/636, zuletzt geprüft am 31.12.2009. ISBN 0171-4317.

[VERI09] *Verilog HDL*. Hardware Beschreibungssprache: The Insititue of Electrical and Electronics Engineers (IEEE). Online verfügbar unter http://www.verilog.com, zuletzt geprüft am 31.12.2009.

[VERS04] Versteegen, G.; Heßeler, A. (2004): *Anforderungsmanagement. Formale Prozesse, Praxiserfahrungen, Einführungsstrategien und Toolauswahl*. Berlin: Springer (Xpert.press). ISBN 3540009639.

[VERS06] Versteegen, G.: *Softwarequalität im Automotive Umfeld*. Fachzeitung OBJEKTspektrum, Bericht in der OS Embedded/2006, SIGS-DATACOM Verlag, Troisdorf, 2006.

[VHDL09] *Very High Speed Integrated Circuit Hardware Description Language* (VHDL): The Electronic Design Automation (EDA) und Electronic Computer-Aided Design (ECAD) Industry Working Groups. Online verfügbar unter http://www.vhdl.org, zuletzt geprüft am 31.12.2009.

[VM97] V-Modell: *V-Modell 97 / XT*. Herausgegeben von Industrieanlagen-Betriebsgesellschaft mbH (IBAG). Online verfügbar unter http://www.v-modell.iabg.de, zuletzt geprüft am 31.12.2009.

[WAGN03] Wagner, G.; et al. (2003): *Erfolgsfaktoren für ein zuverlässiges und stabiles Fahrzeug-Bus-System*. Fachartikel ATZ 9/2003 Jahrgang 105. Online http://www.vector.com/portal/medien/cmc/press/PCO/IntegrationsReview Audi _ATZ_200309_PressArticle_DE.pdf, zuletzt geprüft am 31.12.2009.

[WANG08] Wang, Y. (2008): *Guideline checking for AUTOSAR-based system development*. Diplomarbeit, Tech. Univ. Berlin.

[WARD06] Ward, D. D. (2006): *MISRA Standards for Automotive Software*. In: Proc. Of the 2nd IEE Conference on Automotive Electronics. London. Reference /Institution of Electrical Engineers, 2006, 11400. ISBN 0-86341-609-8.

[WEST96] Westermann, T. (1996): *Mathematik für Ingenieure mit Maple*. 1. Aufl. Berlin: Springer (Springer-Lehrbuch, Band: 1). ISBN 3-540-61249-1.

[WYKE02] Wyke, R. A.; Rehman, S.; Leupen, B. (2002): *XML - Das Entwicklerbuch*. Unterschleißheim: Microsoft Press. ISBN 3-86063-636-7.

[XMI21] *XML Metadata Interchange* (XMI). MOF 2.0/XMI Mapping, v2.1.1. OMG, Inc, 2007. Online http://www.omg.org/spec/XMI/2.1.1, am 31.12.2009.

[XML09] World Wide Web Consortium (W3C): *Extensible Markup Language* (*XML*). Online verfügbar unter http://www.w3.org/standards/xml.

[ZAN08] Zander-Nowicka, Justyna (2008): *Model-based testing of real-time embedded systems in the automotive domain*. Techn. Univ. Berlin, Diss. Berlin: Fraunhofer IRB Verl. ISBN 3816779743.

Veröffentlichungen

Ideen, Methoden und werkzeugtechnische Teilaspekte der vorliegenden Arbeit sind bereits im Vorfeld in den folgenden wissenschaftlichen Veröffentlichungen, Büchern und Vorträgen auf Fachtagungen vorgestellt und wissenschaftlich diskutiert worden:

Buchkapitel, Zeitschriften und Journal Artikel

[1] Farkas, T.; Klein, T.; Röbig, H.: "*Application of Quality Standards to Multiple Artifacts with a Universal Compliance Solution",*In: "*Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)*", Lecture Notes in Computer Science (LNCS), Giese, H.; Karsai, G.; Lee, E.; Rumpe, B.; Schätz, B. (Eds.), Springer, 2010, Band 6100, ISBN: 978-3-642-16276-3, pp. 377-384

[2] Farkas, T.: "*Quality Improvement in Automotive Software Engineering using a Model-Based Approach",*In:"*Model-Driven Software Development: Integrating Quality Assurance (Premier Reference Source)"*, Rech, J., Bunse, C. (Eds.), Idea Group Publishing, 2008, ISBN-13: 978-1605660066, pp. 374-399

[3] Farkas, T.: "*Automotive Software Engineering",* In: "*Das vernetzte Automobil. Mehr Sicherheit und Effizienz durch Informations- und Kommunikations-technik*", Jörg Eberspächer (Ed.), Münchner Kreis, Hüthig Verlag, Heidelberg, 2009, ISBN: 978-3-7785-4050-3, pp. 189-203

[4] Farkas, T., Hinnerichs, A., Neumann, C.: "*A Service-oriented Approach for Interdisciplinary Simulation of Mechatronic Systems using Web Services",* In: "*PIK - Praxis der Informationsverarbeitung und Kommunikation*", Journal Volume 31, Issue 4, pp. 239 - 243, ISSN (Online): 1865-8342, ISSN (Print): 0930-5157, DOI: 10.1515/piko.2008.0041, October-December 2008

[5] Farkas, T.: "*Modelle machen Software mobil – Standardisierung von Methoden schafft mehr Geschwindigkeit in der Entwicklung"*, In: InnoVisions-Magazin, Fraunhofer-IuK-Verbund, Ausgabe Herbst 2007, S.55, ISSN (Print): 1862-7226

Konferenzpapiere und ausgewählte Vorträge

[6] Farkas, T., Hinnerichs, A., Neumann, C.: *"An Integrative Approach for Embedded Software Design with UML and Simulink*", In: Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference, (COMPSAC 2009), 2nd IEEE International Workshop on Industrial Experience in Embedded Systems Design (IEESD 2009), Seattle, USA, 20.-24. July, 2009, ISBN 978-0-7695-3726-9, pp. 516-521

[7] Clauß, C.; Schneider, A.; Schneider, P.; Stork, A.; Bruder, T.; Farkas, Tibor: "*Functional Digital Mock-Up für mechatronische Systeme*", Informations-technische Gesellschaft u.a.: 7. GI / GMM / ITG-Workshop Multi-Nature Systems: Entwicklung von Systemen mit elektronischen und nicht-elektronischen Komponenten, Ulm, 2.-3. Februar, 2009

[8] Farkas, T.; Kamiya, S.; Neumann, C.; Meiseki, E.; Hinnerichs, A.; Okano, K.: "*Integration of UML with Simulink into Embedded Software Engineering*", In: Proceedings of the ICROS-SICE International Joint Conference 2009 (ICCAS-SICE 2009) at SICE Embedded Control Systems Summit 2009, Fukuoka International Congress Center, Fukuoka , Japan, 18.-21. August, 2009

[9] Farkas, T.: "*Model Driven Engineering of Automotive Software",* Invited talk at `Fraunhofer Technologietag bei der ESG`, ESG Elektroniksystem-und Logistik-GmbH, Unternehmenszentrale, Fürstenfeldbruck, 2009

[10] Schneider, P.; Clauß, C.; Schneider, A.; Stork, A.; Bruder, T.; Farkas, T.: "*Towards more Insight with Functional Digital Mockup*", In: Proceedings of Simulation for Innovative Design. European Automotive Simulation Conference (EASC 2009), Munich, July, 2009, pp. 325-336

[11] Farkas, T.; Li, M.: "*Overall Guideline Checking along the Product Development Process",* In: Proceedings of Embedded World Conference 2009, Exhibition and Conference, Weka Fachmedien GmbH (Verl.), Nürnberg, 3.-5. March, 2009, ISBN: 978-3-7723-3798-7

[12] Farkas, T.: "*Automotive Software Engineering"*, In: Proceedings of the 1st Fraunhofer Automotive Symposium Korea at Hyundai-Kia Research & Development Center, Fraunhofer et al. (Eds.), Seoul, Korea, 2008

[13] Farkas, T.: "*Automated Quality Inspection for Automotive Software*", Invited talk at, electronica 2008 Exhibition, Automotive Forum, ZVEI - Zentralverband Elektrotechnik- und Elektronikindustrie e.V., Munich, 2008

[14] Farkas, T.: "*Examples of Use MDE in Germany within Embedded Software Engineering for Automotive*", In: Proceedings of Embedded SW Insight Conference 2008, Korea SW Industry Promotion Agency (KIPA), Seoul, Korea, 2008

[15] Farkas, T., Hinnerichs, A., Neumann, C.: "*A Distributed Approach for Multi-Domain Simulation of Mechatronic Systems",* Proceedings of the 12th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2008), Kunming, China, October 2008, IEEE Computer Society Press, ISBN: 978-0-7695-3377-3, pp. 185-191

[16] Farkas, T., Hinnerichs, A., Neumann, C.: "*An Interdisciplinary Approach to Functional Prototyping for Mechatronic Systems using Multi-Domain Simulation with Model-Based Debugging"*, In: Proceedings of International Multi-Conference on Engineering and Technological Innovation (IMETI 2008), June 29th - July 2nd, 2008, Florida, USA, Vol.2, ISBN: 1-934272-44-2, pp. 7-12

[17] Schäfer, C., Voigt, L., Bruder, T., Stork, A., Schneider, P., Clauß, C., Schneider, A., Farkas, T., Hinnerichs, A.: "*Framework for the Development of Mechatronic Systems"*, SIMVEC - Numerical analysis and simulation in vehicle engineering 2008, In: VDI-Gesellschaft Fahrzeug- und Verkehrstechnik (Hg.) (2008): Berechnung und Simulation im Fahrzeugbau 2008, 14. Internationaler Kongress und Fachausstellung;Tagung, Baden-Baden, 26.-27. November 2008, VDI (VDI-Berichte, 2031), ISBN: 978-3-18-092031-3

[18] Farkas, T.: *"Verfahren zur automatisierten Qualitätssicherung für die Automotive Steuergeräte Entwicklung",* In: Proceedings of the Automotive Workshop `High-Tech-Forschung für wettbewerbsrelevante Produkt- und Prozessinnovation im Automobilbau`, Fraunhofer-Gesellschaft e.V. in conjunction with Pannon Autoindustry Cluster (PANAC) and Széchenyi István University, Györ, Hungary, 2008

[19] Farkas, T.: „*Modellbasierte Entwicklung - Herausforderungen und Lösungen im Bereich Automotive",* invited talk at the 1st Workshop „Modellierung und Simulation"*,* Weinmann Medical Technology, Karlsruhe, Germany, 2008

[20] Farkas, T., Hinnerichs, A., Neumann, C., et. al.: *"Automotive-Lab"*, Invited talk at the Symposium on Quality Engineering for Embedded Systems in conjunction with the ECMDA 2008 (European Conference on Model Driven Architecture), Berlin, 2008

[21] Farkas, T., Neumann, C., Mauch, A., Köchlin, K.: *"Automotive Software Engineering - Integrative Migration to the Unified Modeling Language",* In Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER4), Gehrke, M., Giese, H., Stroop, J. (Eds.), Institut für Informatik, Universität of Paderborn, Paderborn, Germany, 2007

[22] Farkas, T., Neumann, C.: *"Automotive Software Engineering mit der Unified Modeling Language (UML)"*, Beitrag, Gewinner des Innovationspreises und Ausstellung im Rahmen des Würzburger Automobil Gipfel 2007 ausgeschriebenen Innovationswettbewerb 'Network of Automotive Excellence 2007', ewf institute NoAE, Würzburg , 14.-15. Juni, 2007

[23] Farkas, T., Grund, D.: "*Rule Checking in Model Based Development of Safety Critical Software and Information Technical Systems",* In: Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007), Arizona, USA, 2007, ISBN: 0-7695-2804-X, pp. 287-294

[24] Farkas, T., Röbig, H.: „*Automatisierte, werkzeugübergreifende Richtlinienprüfung zur Unterstützung des Automotive-Entwicklungsprozesses"*, in Proceedings of Model Based Engineering of Embedded Systems III (MBEES III), Editor: Conrad, M., Giese, H., Rumpe, B., Schätz,B. (Eds.), TU Braunschweig Report TUBS-SSE 2007-01, Dagstuhl, Germany, 2007

[25] Farkas, T.: "*Automotive Software Development for System Safety and Security"*, In: Proceedings of the 4th Conference Embedded Security in Cars (ESCAR), isits International School of IT Security (Eds.), Berlin, Germany, 2006

[26] Farkas, T.: "*Automotive Software Development & Automatic Evaluation of Design Guidelines for Matlab/Simulink/Stateflow Models",* Invited talk at Samsung R&D Software Laboratories, Seoul, Korea, September 2006

[27] Farkas, T.: "*Automatic Evaluation of Design Guidelines for Matlab/Simulink Models",* Invited talk at Fraunhofer und Siemens VDO Automotive Workshop, Siemens VDO, Regensburg, Germany, Oktober 2006

[28] Farkas, T., Leicher, A., Röbig, H., Born, M., Klein, T., Zander-Nowicka, J.: "*Werkzeugübergreifende Konsistenzsicherung von Artefakten bei der Entwicklung softwarebasierter Systeme im Automobil",* In: Proceedings of the 4th Workshop Automotive Software Engineering, Jahrestagung der Gesellschaft für Informatik (Informatik 2006), Gesellschaft für Informatik (Eds.), Dresden, Germany, 2006

[29] Farkas, T., Hein, C., Ritter, T.: "*Automatic Evaluation of Modelling Rules and Design Guidelines"*, 2nd Workshop 'From code centric to model centric software engineering: Practices, Implications and ROI', In: Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA 2006), Bilbao Spain, 2006

[30] Zander-Nowicka, J., Schieferdecker, I., Farkas, T.: "*Derivation of Executable Test Models From Embedded System Models using Model Driven Architecture Artefacts - Automotive Domain*", In: Proceedings of Model Based Engineering of Embedded Systems II (MBEES II), Giese, H., Rumpe, B., Schätz, B. (Eds.), TU Braunschweig Report TUBS-SSE 2006-01, Dagstuhl, Germany, 2006

Abkürzungen

A, B	
AAES	*Automotive Application Evaluation System*
ADL	*Architecture Description Language*
API	*Application Programming Interface*
APR	*Active Passenger Rescue*
ASAM	*Association for Standardization of Automation and Measuring Systems*
ASCET	*ASCET-MD Werkzeugfamilie*
ASD	*Automotive Software Development*
ASIL	*Automotive Safety Integrity Level*
AUTOSAR	*AUTomotive Open System ARchitecture*
B2B	*Business-to-Business*
BFD	*Brake Force Detection*

C, D	
CAD	*Computer Aided Design*
CAN	*Controller Area Network*
CEN	*Europäischen Komitee für Normung*
CLI	*Common Language Infrastructure*
CLR	*Common Language Runtime*
CMMI	*Capability Maturity Model Integration*
COM	*Component Object Model*
CSS	*Cascading Style Sheet*
CTE	*Classification Tree Editor*
DBMS	*Datenbankmanagementsystem*
DBS	*Relationales Datenbanksystem*
DDL	*Data Definition Language*
DIN	*Deutsche Institut für Normung*
DLINQ	*LINQ-to-SQL*
DML	*Data Manipulation Language*
DOM	*Document Object Model*
DTD	*Document Type Definition*
DXL	*DOORS Script Language*

E-K	
E/E/PE	*Elektrische, elektronische und programmierbar elektronischen Systeme*
EAST-ADL	*Electronics Architecture and Software Technology Architecture Description Language*
ECU	*Electronic Control Unit*
EMF	*Eclipse Modeling Framework*
ERD	*Entity-Relationship-Diagramm*
FIT	*Fahrzeugintensivtest*

FOL	*First-Order Logic*
GUI	*Graphical User Interface*
HIS	*Hersteller Initiative Software*
HiL	*Hardware in the Loop*
HTML	*Hypertext Markup Language*
IDL	*Interface Definition Language*
IEC	*International Electrotechnical Commission*
IL	*Intermediate Language*
ISO	*International Organisation for Standardization*
IT	*Informationstechnologie*
ITU	*Internationale Fernmeldeunion*
JRE	*Java Runtime Environment*
KFZ	*Kraftfahrzeug*
KPI	*Key Performance Indicator*

L-P

LCA	*Load-Condition-Action*
LINQ	*Language-Integrated Query*
MAAB	*MathWorks Automotive Advisory Board*
MARTE	*Modeling & Analysis of Real-time and Embedded Systems*
MDA	*Model Driven Architecture*
MDL	*MATLAB/Simulink/Stateflow Datenformat*
MISRA	*Motor Industry Software Reliability Association*
ML/SL/SF	*MATLAB/Simulink/Stateflow*
MOF	*Meta Object Facility*
MPR	*Model Persistance Rule*
MSIL	*Microsoft Intermediate Language*
.NET	*Microsoft .NET Framework Technologie*
OCL	*Object Constraint Language*
ODBC	*Open Database Connectivity*
ODF	*Open Document Format for Office Applications*
ODS	*Open Data Service*
ODX	*Open Diagnostic Data Exchange*
OMA	*Object Management Architecture*
OMG	*Object Management Group*
OOXML	*Office Open XML*
OSEK	*Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*
OSLO	*Open Source Library for OCL*
PIM	*Platform Independant Model*
P-LINQ	*Parallel Language Integrated Query*
PSM	*Platform Specific Model*

Q-U

QGS	*Quality Gate*

QM	*Qualitätsmanagementsystem*
QRDL	*Query-based Rule Description Language*
QVT	*Query View Transformation*
RIF	*Requirements Interchange Format*
RTE	*Runtime Environment*
SGB	*Batteriemanagement-Steuergerät*
SGFH	*Fenstersteuergerät*
SGL	*Steuergerät für Lichtsteuerung*
SOP	*Start of Production*
SPiCE	*Software Process Improvement and Capability Determination*
SQL	*Structured Query Language*
SUT	*System Under Test*
SWC	*AUTOSAR-Software-Komponenten*
SysML	*Systems Modeling Language*
TestML	*Test Markup Language*
UML	*Unified Modeling Language*
UUT	*Unit Under Test*

V-Z

V-Modell	*Vorgehensmodell im Produktentstehungsprozess*
VR-Modell	*Vorgehensmodell zur Regelentwicklung*
W3C	*World Wide Web Consortium*
WSC	*World Standards Cooperation*
XLINQ	*LINQ-to-XML*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*
Xpath	*XML Path Language*
Xquery	*XML Query Language*
XSLT	*Extensible Stylesheet Language Transformation*

Symbole

$\boldsymbol{A}$	*Artefakt*
$\boldsymbol{aB}$	*Bezeichner*
$\varnothing$	*Leere Menge*
$\boldsymbol{A}_H$	*Hierarchisches Artefakt*
$\boldsymbol{A}_I$	*Logisches Artefakt*
$\boldsymbol{A}_K$	*Kollaboratives Artefakt*
$\boldsymbol{aT}$	*Datentyp*
$\boldsymbol{d}$	*Datum*
$\boldsymbol{D}$	*Daten*
Δ	*Prüfaufbau (Artefakt-Menge)*
$\boldsymbol{F}$	*Ausführungsumgebung*
$\boldsymbol{f}$	*Funktion*
Φ	*Prüfsystem*
$\boldsymbol{G}$	*Graph*
Γ	*Menge von Richtlinien*
$\boldsymbol{I}$	*Instanzen*
$\boldsymbol{i, j, k}$	*Indizes (natürliche Zahlen)*
$\mathfrak{I}$	*Menge logischer Artefakte*
$\boldsymbol{k}$	*Knoten (Graph)*
$\boldsymbol{K}$	*Kanten (Relation, Graph)*
$\boldsymbol{L}$	*Laufzeitumgebung*
Λ	*Prüfergebnis*

M	*Modell*
n, m	*Natürliche Zahl*
P	*Prozess*
Π	*Menge aller Prüfergebnisse*
Θ, Σ	*Mengen*
R	*Relation*
S	*Spezifikation (Artefakt)*
V	*Knoten (Menge, Graph)*
w	*Wurzel (Graph)*
W	*Wrapper*
Ω	*Prüfraum (Menge von Artefakten*
x	*Quantifikator*
Ξ	*Regelkatalog (Menge von Regeln)*
Ψ	*Regel*

Anhang - A

Architekturmodell der APR-Funktion

Nachfolgende Abbildungen zeigen Vergrößerungen des APR-Architekturmodells, welches im Kapitel 8 bei der Evaluierung der regelbasierten Konformitätsprüfung kollaborativer Artefakte vorgestellt wurde.

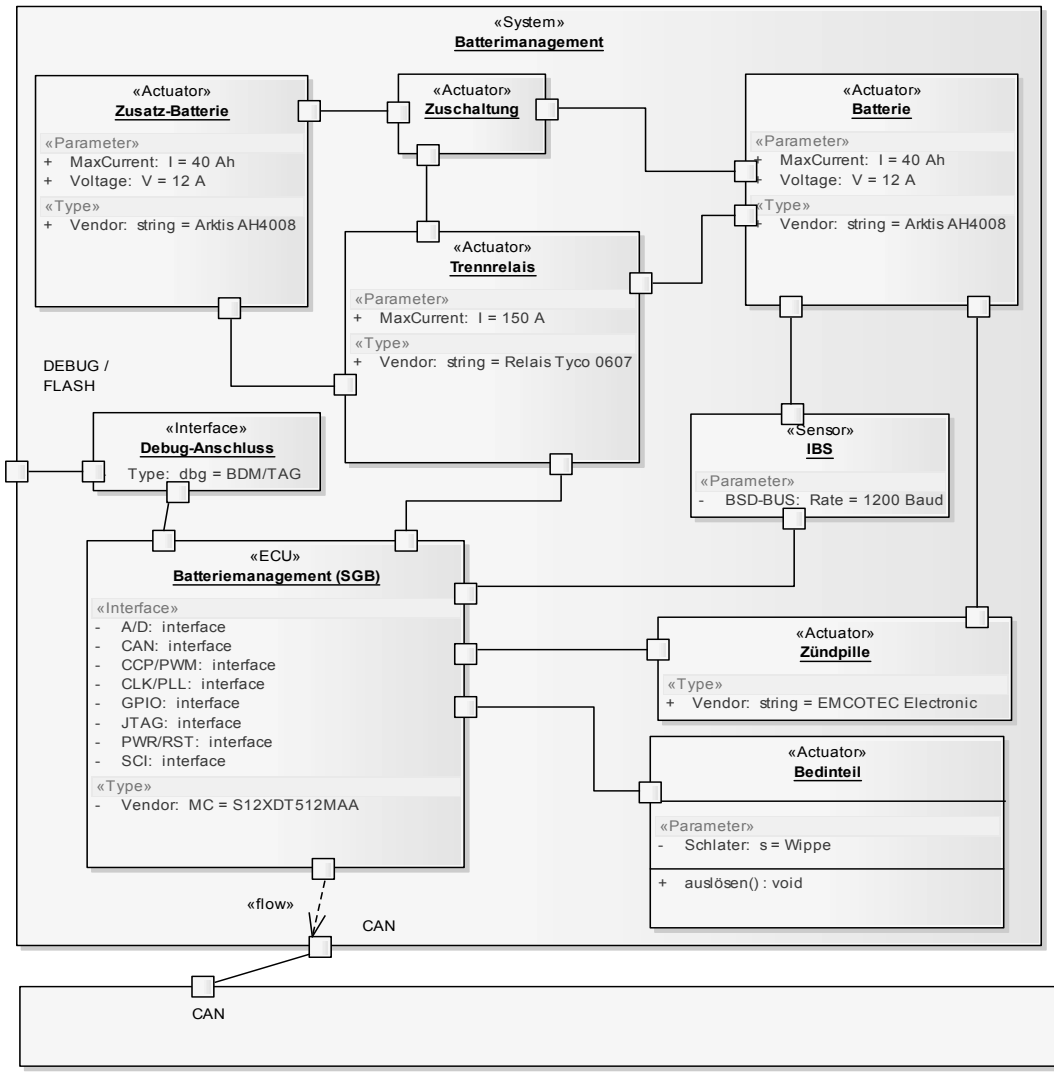


Abbildung 73: Architektur-Modell des Batteriemanagements

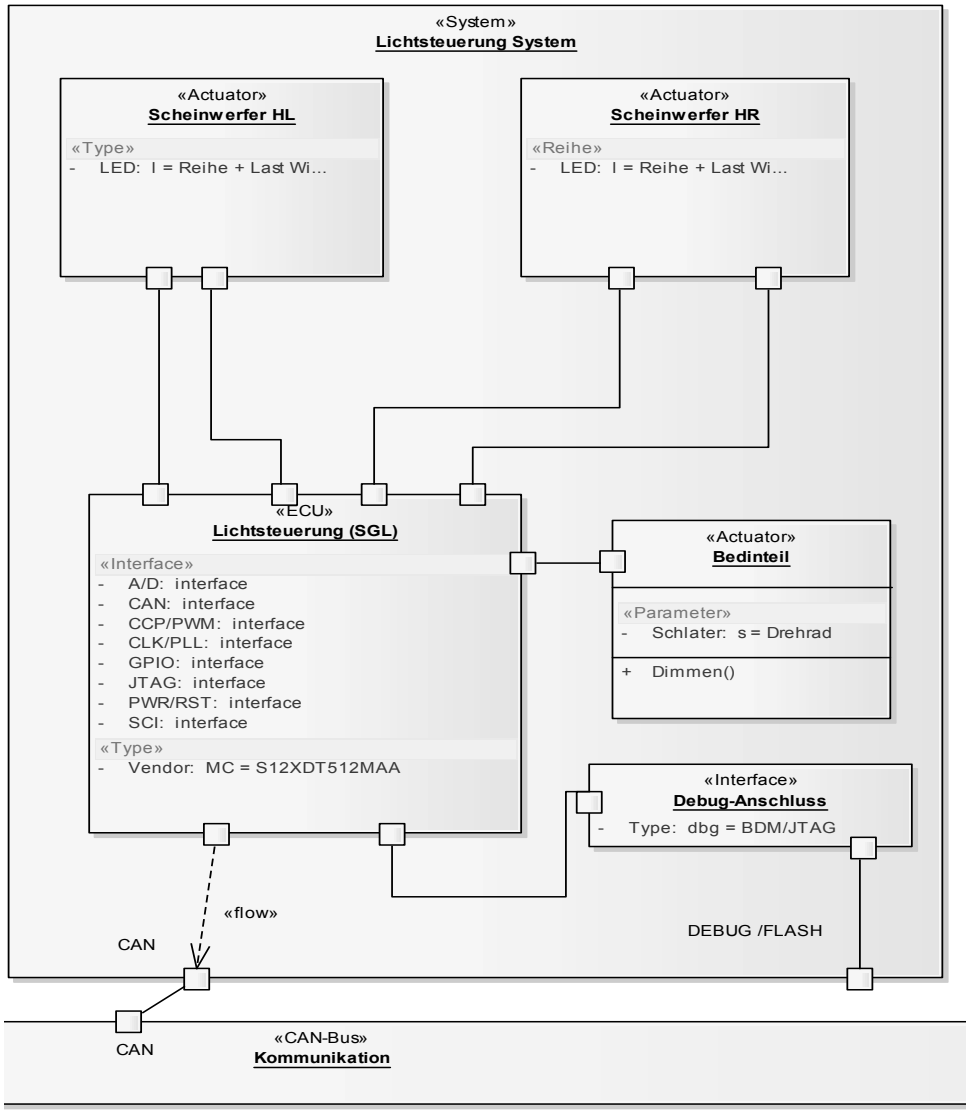


Abbildung 74: Architektur-Modell der Bremslichtsteuerung

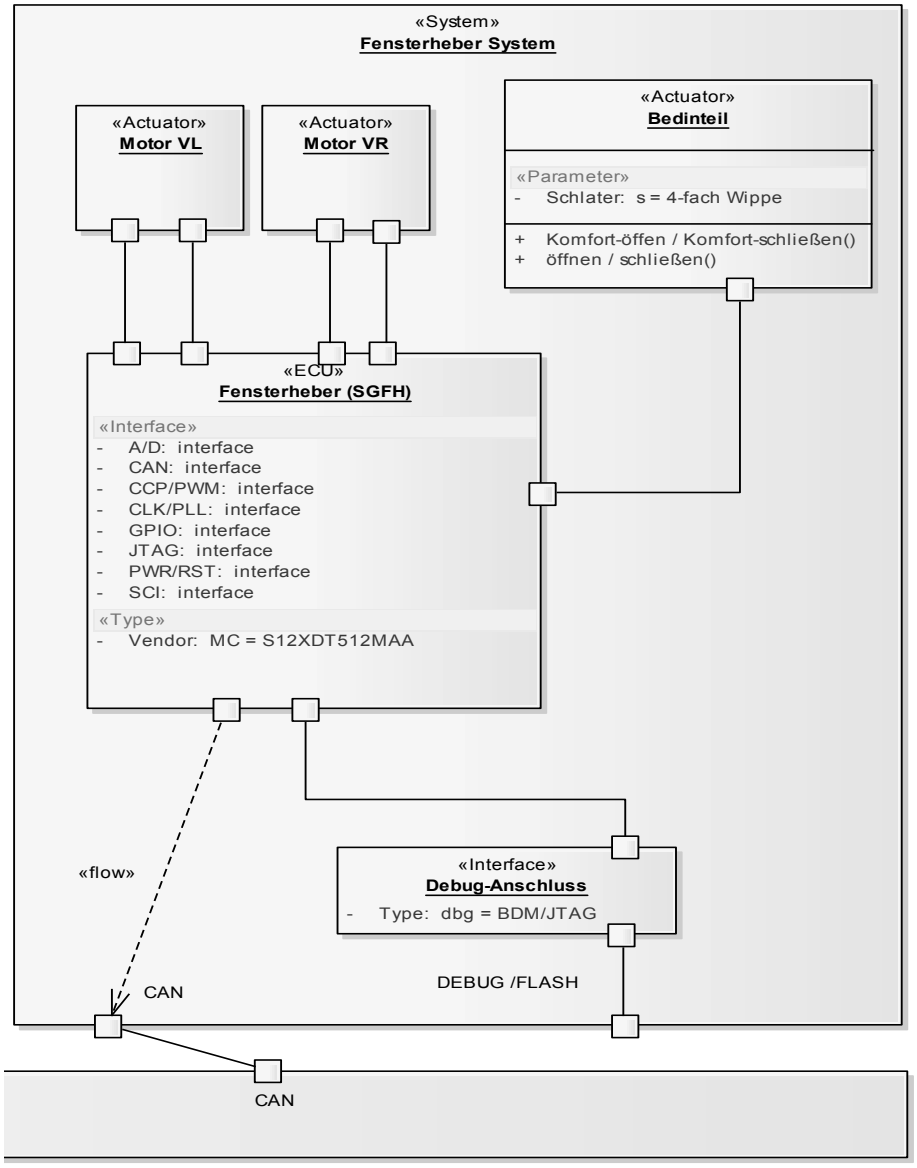


Abbildung 75: Architektur-Modell des Fensterhebers

Anhang - B

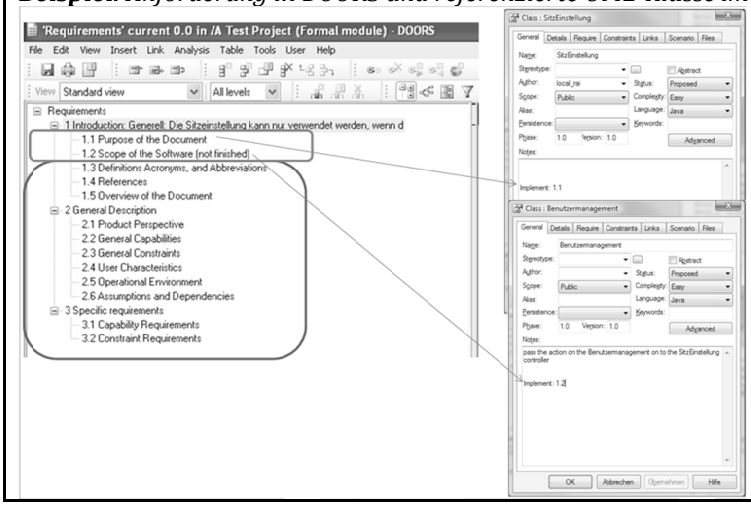
Richtlinien- und Regelbeispiele im APR-Entwicklungsprozess

Nachfolgend werden alle für die Evaluierung im Kapitel 8.3 (Evaluierungsmatrix aus Tabelle 24) aufgestellten Prozessrichtlinien für jede Phase näher vorgestellt sowie einige Implementierungsbeispiele in den Programmiersprachen LINQ, M-Skript und OCL gegeben.

Werkzeugübergreifende Konformitätsprüfung der Anforderungen und Modelle

Richtlinie: Zur Sicherstellung der Nachvollziehbarkeit jeder Systemanforderung des APR muss der Bezug von Anforderungen zu Modellen hergestellt werden. Dies geschieht durch Festlegung der Requirements-ID aus den Anforderungen in das Notizfeld jeder UML-Klasse. Aus den daraus entstehenden logischen Beziehungen lässt sich pro Artefakt jederzeit ableiten, welches Modellelement welche Anforderung abdeckt. Die Tabelle 26 zeigt die Richtlinie sowie die daraus implementierte LINQ-Regel, welche die Konformität zur aufgestellten Anforderung werkzeugübergreifend sicherstellt.

Tabelle 25: Anforderung - Richtlinie für Nachvollziehbarkeit

ID: 1	**Artefakt:** DOORS, UML
Prozessphase:	*Anforderungsmanagement*
Titel:	***Nachvollziehbarkeit***
Kategorie:	*Verständlichkeit* **Priorität:** *Empfohlen*
Vorbedingung:	$A_n \neq \emptyset$, (*Der Anforderungstext ist nicht leer.*)
Richtlinie:	*Zur Sicherstellung der Nachvollziehbarkeit jeder Systemanforderung des APR muss der Bezug von Anforderungen zu Modellen hergestellt werden.*
Beschreibung:	*Um bei der Modellierung jederzeit zu einem Modellelement die jeweilige Anforderung zuordnen zu können, sollen bei den UML-Modellen im Notizfeld (Bemerkung) die jeweilige Anforderungs-ID hinterlegt sein.* ***Beispiel:*** *Anforderung in DOORS und referenzierte UML-Klasse im EA.* 

Konformität:

$$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{Ao, \ldots, An + 1\} \end{cases}$$

Ausführbare Regel in LINQ:

```
Regex impl = new Regex("Implement: [1-9](.[0-9][0-9]*)*");

var Query1 = from c in Artefact["Doors_Database"].Descendants()
             where c.Name == "theDRSObject" && c.Attribute("identifier") != null
             select c;

var Query2 = from c in Artefact["Enterprise_Architect_Model"].Descendants()
             where c.Attribute("tag") != null && impl.IsMatch(c.Attribute("value").Value)
             let m = impl.Match(c.Attribute("value").Value)
             let s = m.ToString()
             select s.Substring(s.IndexOf(":") + 2);

List<XElement> Query = new List<XElement>();
foreach (XElement xele in Query1)
{
    string id = xele.Attribute("identifier").Value;
    if (!Query2.Contains(id))
        Query.Add(xele);

}
if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

Zunächst wird ein Muster zur Erkennung der Anforderungs-ID (Gliederung) durch das Pattern `impl` durch einen regulären Ausdruck festgelegt. Daran wird erkannt, ob es eine Entsprechung der Anforderungs-ID in dem prozesslogisch referenzierten Artefakt gibt. Die werkzeugübergreifende Regel-Implementierung ist zu Beginn in zwei wesentliche Abfragen unterteilt. Die erste Abfrage sammelt alle nicht leeren Überschriften des logischen Artefakts der Anforderungen in einer Menge `Query1` auf.

In der zweiten Abfrage werden alle Elemente des UML-Modells durchsucht, welche eine Notiz (Tag-Attribut) besitzen können. Als zusätzliche Bedingung wird in die Abfrage aufgenommen, dass der Inhalt der Notiz auch eine Anforderungs-ID aufweist. Alle gefundenen Anforderungs-IDs werden in einer Menge `Query2` gespeichert.

Eine ***foreach***-Schleife untersucht nun alle Elemente der ersten Menge nach Entsprechungen in der zweiten Menge durch das ***Contains***-Schlüsselwort. Hat ein Element keine Entsprechung, wird es in der Ergebnismenge `Query` gespeichert. Ist die Ergebnismenge leer, so hatte jedes Element seine Entsprechung und die Konformität zur Richtlinie konnte werkzeugübergreifend nachgewiesen werden. Andernfalls ist der `Query.Count() > 0` und eine Verletzung der Richtlinie kann ausgegeben werden. Zusätzlich erhält man in der Ergebnismenge alle die Elemente, die keine Entsprechung besitzen. Werden diese Elemente im Prüfbericht mit ausgegeben, so fällt die Fehlerfindung deutlich einfacher aus.

Konformitätsprüfung der Anforderungen

Richtlinie: Um die Eindeutigkeit in der Aussagekraft einer Systemanforderung des APR nicht zu beeinträchtigen, sollen ungenaue Hilfsverben zum Ausdruck der Modalität wie ‚*kann*', ‚*könnte*', ‚*können*', ‚*könnten*' und der Temporalformen ‚*konnte*' oder ‚*konnten*' vermieden werden. Die Tabelle 26 zeigt die Richtlinie sowie daraus implementierte Regel, welche die Konformität zur aufgestellten Anforderung sicherstellt.

Tabelle 26: Anforderung - Richtlinie für Benutzbarkeit und Verständlichkeit

ID: 2	**Artefakt:** DOORS
Prozessphase:	*Anforderungsmanagement*
Titel:	***Ausdrucksklarheit***
Kategorie:	*Verständlichkeit* **Priorität:** *Empfohlen*
Vorbedingung:	$A_n \neq \emptyset$, (*Der Anforderungstext ist nicht leer.*)
Richtlinie:	*Die Ausdrücke ‚kann', ‚könnte', ‚können', ‚könnten', ‚konnte' oder ‚konnten' sind in einer Spezifikation nicht erlaubt.*
Beschreibung:	*Um die Eindeutigkeit einer Systemanforderung nicht zu beeinträchtigen, sollen ungenaue Formulierungen (Weak Words) wie kann, könnte, können, könnten, konnte oder konnten vermieden werden.*
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{Ao, \ldots, An+1\} \end{cases}$
Ausführbare Regel in LINQ:	

```
string[] prohibited = new string[]

{ " kann ", " könnte ", " können ", " könnten ", " konnte ", " konnten " };

var Query = from c in Artefact["Doors"].Descendants()

            where c.Element("AttributeValue") != null
               && (c.Element("AttributeValue").Value.Contains(prohibited[0])
               || c.Element("AttributeValue").Value.Contains(prohibited[1])
               || c.Element("AttributeValue").Value.Contains(prohibited[2])
               || c.Element("AttributeValue").Value.Contains(prohibited[3])
               || c.Element("AttributeValue").Value.Contains(prohibited[4])
               || c.Element("AttributeValue").Value.Contains(prohibited[5]))
            select c.Parent;

if (Query.Count() > 0) Result = "FAIL"; else Result = "PASS";
```

Der Algorithmus definiert zunächst eine beliebige Menge `prohibited` von nicht erlaubten Wörtern und deren Wortformen. Anschließend wird für alle Attribute mit dem Bezeichner `AttributeValue` (entspricht Anforderungstext) die *Contains*-Überprüfung durchgeführt, womit ein ***String***-Vergleich mit dem Attribut und der Menge nichterlaubter Wörter durch logische ODER-Verknüpfung durchgeführt wird. Die Richtlinie gibt bei einem Pendant die fehlerhaften Elemente (Anforderungstexte) zurück.

In dieser Implementierungsform stehen die unerlaubten Wörter direkt im Regelalgorithmus. In der Praxis werden solche Wörter häufig in einem anderen Artefakt (wie z. B. MS Excel) geführt. Hier bietet sich eine werkzeugübergreifende Prüfung von zwei Artefakten an (DOORS-Anforderung ↔ Excel-Tabelle). Solch eine Implementierung wird später gezeigt.

Richtlinie: Die Erzeugung von syntaktisch und semantisch korrekten Artefakten aus den Systemanforderungen ist nur durch eine unikonforme Bezeichnung der Anforderungstitel gegeben. Durch Klammerung eingefasste Nebensätze mit den Symbolen *(* und *)* sind im Anforderungstitel nicht erlaubt, da sie im weiteren Entwicklungsprozess für abgeleitete Modellbezeichnungen verwendet werden. Die Tabelle 27 zeigt die zugehörige Richtlinie sowie die daraus implementierte Regel, welche die Konformität zur aufgestellten Anforderung sicherstellt.

Tabelle 27: Anforderung - Richtlinie für Verbesserung der Übertragbarkeit

ID: 3	**Artefakt:** DOORS
Prozessphase:	*Anforderungsmanagement*
Titel:	***Nebensätze sind im Anforderungstitel nicht erlaubt***
Kategorie:	*Übertragbarkeit* **Priorität:** *Erforderlich*
Vorbedingung:	$A_n \neq \emptyset$, (*Der Anforderungstext ist nicht leer.*)
Richtlinie:	*Klammer-Symbole sind im Anforderungstitel nicht erlaubt*
Beschreibung:	*Die Erzeugung von weiteren syntaktisch und semantisch korrekten sowie auch effizienten Artefakten aus Systemanforderungen ist nur durch eine unikonforme Bezeichnung der Anforderungstitel gegeben. Durch Klammerung eingefasste Nebensätze mittels Symbolen (Klammern) sind im Anforderungstitel nicht erlaubt.*
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\text{Ao}, \ldots, An + 1\} \end{cases}$
Ausführbare Regel in LINQ:	

```
var Query = from c in Artefact["Doors"].Descendants("theDRSValue")
            where c.Element("AttributeName").Value == "Object Heading"
            && c.Element("AttributeValue").Value.Contains("(")
            && c.Element("AttributeValue").Value.Contains(")")
            select c.Parent;

if (Query.Count() > 0) Result = "FAIL"; else Result = "PASS";
```

Der Algorithmus adressiert zunächst den Wurzelknoten `theDRSValue` im Anforderungsbaum und traversiert alle seine Kind-Elemente `Object Heading` (Anforderungstitel) mit der VERUNDETEN-Bedingung, dass das geöffnete Klammer-Symbol und das geschlossene Klammer-Symbol nicht im Wert des `AttributeValue` (***String***-Datentyp) enthalten sind. Die Richtlinie gibt bei einem Pendant die fehlerhaften Elemente (Anforderungstitel) zurück.

Richtlinie: Für die eindeutige Identifizierung jedes Anforderungstextes (Textbausteins) im APR müssen unternehmensspezifische Bezeichner, sogenannte `Object IDs` zu jeder Anforderung vergeben werden. Diese müssen vom Requirements-Engineer festgelegt werden. Außerdem ist darauf zu achten, dass diese bei Vergabe eindeutig sind und sich nicht bereits in einem anderen Projekt und dessen Anforderungstext in Verwendung befinden.

Tabelle 28: Anforderung - Richtlinie Funktionsabsicherung & Zuverlässigkeit

ID: 4	**Artefakt:** DOORS
Prozessphase:	*Anforderungsmanagement*
Titel:	***Überprüfung von eindeutigen Anforderungs-Bezeichnern***
Kategorie:	*Funktionsabsicherung und Zuverlässigkeit* **Priorität:** *Erforderlich*
Vorbedingung:	$A_n \neq \emptyset$, *Der Anforderungstext ist nicht leer.*
Richtlinie:	*Es dürfen nur eindeutige Bezeichner einmalig verwendet werden.*
Beschreibung:	*Zur Funktionsabsicherung und Zuverlässigkeit ist sicherzustellen, dass Anforderungen eindeutig referenzierbar sind und nicht doppelt vorkommen. Für die eindeutige Identifizierung eines Anforderungstextes (Bausteins) müssen daher unternehmensspezifische Bezeichner, sogenannte Object IDs zu jeder Anforderung vergeben werden. Diese müssen vom Requirements Engineer festgelegt werden. Außerdem ist darauf zu achten, dass diese bei Vergabe eineindeutig sind und sich nicht in einem anderen Projekt (Anforderungstext) bereits in Verwendung befinden.*
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\mathrm{Ao}, \ldots, An+1\} \end{cases}$
Ausführbare Regel in LINQ:	

```
int index = 1;
var Query1 = from c in Artefact["Doors"].Descendants("AttributeName")
             where c.Value == "ObjID" && c.ElementsAfterSelf().First().Value != ""
             && c.Parent.Parent.Name != "theDRSFormalModule"
             select c;

List<XElement> Query = new List<XElement>();
for (int i = 0; i < Query1.Count(); i++)
{
    for (int j = i + 1; j < Query1.Count(); j++)
        if (Query1.ElementAt(i).ElementsAfterSelf().First().Value
            == Query1.ElementAt(j).ElementsAfterSelf().First().Value)
        {
            Query.Add(Query1.ElementAt(i).Parent.Parent);
            Query.Add(Query1.ElementAt(j).Parent.Parent);
            ResultMessage = ResultMessage + "Xml Element "
                + index + " and Xml Element "
                + (index + 1).ToString()
                + " have same ObjID" + "\n";
            index += 2;
        }
}

if (Query.Count() > 0) Result = "FAIL"; else Result = "PASS";
```

Der Algorithmus adressiert zunächst alle Knoten im DOORS-Artefakt-Baum für die gilt, dass nur deren nachfolgende Elemente mit in die Prüfung einbezogen werden sollen, die sich auch in der Spalte `ObjID` befinden. Dies entspricht der Element-Menge in der Spalte, in der die Bezeichner vermerkt werden. Mit `c.ElementsAfterSelf().First().Value` wird dann sichergestellt, dass diese Elemente auch Bezeichner enthalten und demnach nicht leer sind.

Schließlich muss noch die Menge auf einen lokalen Gültigkeitsbereich (z. B. nur innerhalb eines Kapitels) eingeschränkt werden. Im DOORS-Artefakt nennen sich diese Gültigkeitsbereiche ‚*Module*'. So schränkt der letzte Ausdruck `theDRSFormalModule` die ausgewählte Menge der Bezeichner in den Gültigkeitsbereich eines Moduls ein. Die Menge `Query1` enthält nun alle relevanten Bezeichner der Anforderungen im Gültigkeitsbereich. Nun muss die Eineindeutigkeit in diesem Gültigkeitsbereich sichergestellt werden. Dies funktioniert exemplarisch mit zwei verschachtelten Schleifendurchläufen, wobei jedes Element `i` in der Menge iteriert durchlaufen wird und mit dessen Nachfolger `j (i + 1)` verglichen wird. Wird eine Gleichheit festgestellt, werden die gefundenen Elemente in eine zweite Ergebnismenge `Query` für die spätere Auffindung der Elemente kopiert und eine detaillierte Ergebnisnachricht `ResultMessage` dynamisch aufgebaut. Schließlich gibt die Richtlinie bei einer nicht leeren Ergebnismenge `Query` mit `Query.Count() > 0` das „`FAIL`" Ergebnis sowie die fehlerhaften Elemente (Bezeichner) in der Ergebnisnachricht `ResultMessage` zurück.

Konformitätsprüfung im modellbasierten Systementwurf

Richtlinie: Im modellbasierten Systementwurf des APR ist bei der Vergabe von Namen, bzw. eindeutigen Bezeichnern für Modellobjekte wie Blöcke, Signale oder Parameter grundsätzlich darauf zu achten, dass die Namen der Modellobjekte (die sie benennen) aussagekräftig beschreiben und dass Modellobjekte dadurch eindeutig identifiziert werden. Dies gilt für Klassendiagramme, Funktionsmodelle gleichwie für Zustandsautomaten. Für eine angestrebte Wiederverwendung muss diese Eindeutigkeit über Modellgrenzen hinweg sichergestellt sein, um die Integrationsfähigkeit von Teilmodellen zu gewährleisten. Beim prozessübergreifenden Austausch werden teilweise deutlich unterschiedliche Namenskonventionen praktiziert. Ein manueller Abgleich der Namensrichtlinien ist im Allgemeinen sehr aufwendig.

Einhaltung der Namenskonventionen

Richtlinie: Für die nachgelagerte Codegenerierung aus Klassendiagrammen, Funktionsmodellen oder Zustandsautomaten der APR-Systemfunktion ist zudem sicherzustellen, dass Compiler beim Kompilieren und Linken des generierten Codes keine zusätzlichen Fehler aufgrund von falschen Namensbenennungen produzieren. Dies liegt darin begründet, dass Modellierungswerkzeuge im Gegensatz zu Compilern erweiterte Zeichensätze sowie lange Zeichenketten verarbeiten können. Fassen wir die Konventionen für Namen der APR kurz zusammen:

i. *Namen haben eine maximale Länge.*

ii. *Namen sollen einem vorgegebenen Muster folgen.*

iii. *Namen sollen keine Sonderzeichen beinhalten.*

iv. *Namen sollen nur Akronyme verwenden, die in einem Glossar hinterlegt sind.*

v. *Namen sollen in einheitlicher Sprache (z. B. deutsch/englisch) beschrieben sein.*

Die Tabelle 29 zeigt die für die Richtlinie (***i***) entwickelte Regel für zwei verschiedene Artefakte, welche die Konformität zur aufgestellten Anforderung sicherstellt.

Tabelle 29: Systementwurf - Richtlinie für Benutzbarkeit & Verständlichkeit

ID: 5	**Artefakt:** UML, ML/SL/SF
Prozessphase:	*Funktionaler und technischer Systementwurf*
Titel:	***Maximale Namenslänge von Bezeichnern in Modellen***
Kategorie:	*Benutzbarkeit und Verständlichkeit* **Priorität:** *Erforderlich*
Vorbedingung:	$ID_n \neq \emptyset$, (*Der Bezeichner ist nicht leer.*)
Richtlinie:	*Es sind nur Bezeichner bis zur maximalen Länge von 31 Zeichen erlaubt.*
Beschreibung:	*Modelle verbleiben nicht isoliert im Entwicklungsprozess, sondern interagieren mit weiteren nachgelagerten Entwicklungsartefakten oder Werkzeugen: Beispiele sind Codegeneratoren für das Rapid Prototyping sowie Compiler für aus Modellen abgeleiteten Programmcode. Diese Tools sind im Allgemeinen hinsichtlich des erlaubten Zeichensatzes für Bezeichner limitiert und richten sich häufig nach dem ANSI C Standard.*
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\mathrm{IDo}, \ldots, \mathrm{ID}n+1\} \end{cases}$
Ausführbare Regel in LINQ:	a) Algorithmus für ein UML-Klassendiagramm:

```
const int length = 31;
var Query = from c in Artefact["UML_Model"].Descendants()
            where (c.Name.LocalName == "Class" || c.Name.LocalName == "Attribute"
                                               || c.Name.LocalName == "Operation")
               && c.Attribute("name") != null
               && c.Attribute("name").Value != "EARootClass"
               && c.Attribute("name").Value.Length > length
            select c;

 if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

b) Algorithmus für ein MATLAB/Simulink/Stateflow-Modell:

```
const int length = 31;
var Query = from c in Artefact["Matlab_Simulink_Model"].Descendants()
            where (c.Name == "Block" && c.Element("Name") != null
               && c.Element("Name").Value.Length > length)
               || (c.Name == "state" && c.Element("labelString") != null
               && c.Element("labelString").Value.Substring(1,
                  c.Element("labelString").Value.IndexOf("\\n") - 1).Length > length)
            select c;

 if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

Der Algorithmus in a) und b) definiert die maximale Obergrenze für die Namenslänge in der Konstanten `length` auf ein Maximum von 31 Zeichen für Bezeichner. Vom Wurzelknoten des UML- bzw. ML/SL/SF-Modells werden alle folgenden Knotenelemente (`Descendants`) im Artefakt-Baum traversiert. Der Algorithmus prüft hierbei insbesondere nur die relevanten Elemente wie in a) Klassennamen, Klassenattribute und Klassenoperationen und in b) Blöcke (Wert: Block für ***Name***-Attribut) und Zustände (Wert: State für ***Name***-Attribut). Die Voraussetzung, dass die Bezeichner nicht leer sind, wird in a) und b) zusätzlich sichergestellt. In b) wird zudem gefordert, dass auch keine Umbrüche (Line Feeds/Carriage-Returns) im Namen vorhanden sind, da das Werkzeug ML/SL/SF dies erlaubt. Schließlich

wird mittels `Value.Length > length` der Vergleich mit dem Maximum vorgenommen. Die Richtlinie gibt bei einem Pendant die fehlerhaften Elemente wie Klassenbezeichner, Blockbezeichner oder Zustandsbezeichner zurück.

Die Namenskonventionen aus (***ii***) und (***iii***) lassen sich durch einen gemeinsamen Algorithmus formulieren, der auf ‚regulären Ausdrücken' basiert. Ein regulärer Ausdruck bezeichnet ein Schema zum Erzeugen oder Filtern von Zeichenketten nach bestimmten Regeln. Mit einem regulären Ausdruck kann eine Menge von Zeichenketten erzeugt werden (die Sprache). Genauso kann überprüft werden, ob eine Zeichenkette in dieser Menge liegt. Letzteres Verfahren wird als *Mustererkennung* (Pattern Matching) bezeichnet.

Die Tabelle 30 zeigt die für (***ii***) und (***iii***) entwickelte Richtlinie zur Erkennung von Sonderzeichen (Mustererkennung wäre identisch, nur im Ausdruck verschieden) als Regel mittels regulären Ausdruckes, welche die Konformität zur aufgestellten Forderung sicherstellt.

Tabelle 30: Systementwurf - Richtlinie zur Verbesserung der Übertragbarkeit

ID: 6	**Artefakt:** UML, ML/SL/SF
Prozessphase:	*Funktionaler und technischer Systementwurf*
Titel:	***Erlaubter Zeichensatz***
Kategorie:	*Verbesserung der Übertragbarkeit* **Priorität:** *Erforderlich*
Vorbedingung:	$ID_n \neq \emptyset$, (*Der Bezeichner ist nicht leer.*)
Richtlinie:	*Es sind nur Bezeichner bis zur maximalen Länge von 31 Zeichen erlaubt.*
Beschreibung:	*Der erlaubte Zeichensatz richtet sich nach den in der ANSI C Spezifikation erlaubten Vorgaben für Zeichen. Alle Namen für Modellobjekte sowie Datei- und Parameternamen dürfen sich ausschließlich aus folgenden Zeichen zusammensetzen: Großbuchstaben A – Z, Kleinbuchstaben a – z, Nummern 0 – 9 oder Unterstrich '_'. Keinesfalls dürfen Umlaute oder Sonderzeichen verwendet werden, wie z. B. 'ä', 'ö', 'ü', 'ß', '&', '$' sowie ein Leerzeichen.*
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\mathrm{IDo}, \ldots, \mathrm{ID}n+1\} \end{cases}$
Ausführbare Regel in LINQ:	a) Algorithmus für ein UML-Klassendiagramm:

```
Regex mr56 = new Regex("[^a-zA-Z0-9_]");
var Query = from c in Artefact["UML_Model"].Descendants()
            where (c.Name.LocalName == "Class"
                || c.Name.LocalName == "Attribute"
                || c.Name.LocalName == "Operation")
                && c.Attribute("name") != null
                && c.Attribute("name").Value != "EARootClass"
                && mr56.IsMatch(c.Attribute("name").Value)
            select c;

 if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

b) Algorithmus für ein MATLAB/Simulink/Stateflow-Modell:

```
Regex mr56 = new Regex("[^a-zA-Z0-9_]");
var Query = from c in Artefact["Matlab_Simulink_Model"].Descendants()
            where (c.Name == "Block" && c.Element("Name") != null
               && mr56.IsMatch(c.Element("Name")
                               .Value.Substring(1, c.Element("Name")
                               .Value.Length - 2)))
               || (c.Name == "state" && c.Element("labelString") != null
               && c.Element("labelString").Value.Contains("\\n")
               && mr56.IsMatch(c.Element("labelString")
                      .Value.Substring(1, c.Element("labelString")
                      .Value.IndexOf("\\n") - 1)))
            select c;

 if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

Ausführbare Regel in M-Skript:

Implementierung in M-Skript:

```
function [Result, ResultDetail] = checkBlockNames(system)

    allBlocks = find_system(system,'FindAll','on',...
    'Type', 'block');
    inBlocks = find_system(system,'FindAll','on',...
    'BlockType', 'Inport');
    outBlocks = find_system(system,'FindAll','on',...
    'BlockType', 'Outport');
    subsyss = find_system(system,'FindAll','on',...
    'BlockType', 'SubSystem');

    blocks = setdiff(allBlocks, union(inBlocks, outBlocks));
    blocks = setdiff(blocks, subsyss);
    ResultDetail = {};
  Result = 'fail';

    for i = 1: length(blocks)
       bname = get_param(blocks(i),'Name');
       erg = regexp(bname, '[^a-zA-Z_0-9\n]|^[\d\s]', 'match');

    if ~isempty(erg)
      ResultDetail{end + 1} = getfullname(blocks(i));
        end
    end
    if isempty(ResultDetail)
    Result = 'pass';
    end
```

Ausführbare Regel in OCL:

Implementierung in OCL:

```
context OclVoid inv: let valid_characters = Set
{
'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p'
,'q','r','s','t','u','v','w','x','y','z','A','B','C','D','E','F
','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','
V','W','X','Y','Z','0','1','2','3','4','5','6','7','8','9','_'
}
in AutomotiveSystemDevelopment::Tools::Simulink::SLStructure::
SLBlock.allInstances()->reject(x| valid characters-
>includesAll(Set{2..x.identifier.size()-1}->iterate(
i: Integer; sum: Sequence(String)=Sequence{} |
sum->append(x.identifier.substring(i,i))))).Derivation
```

Der Algorithmus in a) und b) definiert zunächst den regulären Ausdruck mit dem Muster (*Pattern*) `"[^a-zA-Z0-9_]"` der die Menge an vordefinierten Zeichen in einem Bezeichner beschränkt. Dies wird durch die allgemeine Ausdrucksform realisiert: `[^xyz]` *Ausdruck passt auf jedes Zeichen, mit "x", "y" und "z"*. Danach wird, wie in dem vorherigen Beispiel, der Artefakt-Baum traversiert und hierbei insbesondere nur auf die relevanten Elemente geprüft wie in a) Klassennamen, Klassenattribute und Klassenoperationen und in b) Blöcke (Wert: *Block* für ***Name***-Attribut) und Zustände (Wert: *State* für ***Name***-Attribut). Die Besonderheit liegt nun in der Anwendung des Patterns auf den Bezeichner-String. Dies wird durch die Funktion `isMatch()` realisiert die `true` oder `false` als Rückgabewert liefert und somit eine Aussage zur Einhaltung der Konformität erlaubt.

Als Beispiel sind die Implementierungen auch in M-Skript und in OCL vorgenommen worden. Da die OCL 2.0 keine regulären Ausdrücke unterstützt, muss hier die Menge an Buchstaben und Zahlen angegeben werden, um jedes einzelne Zeichen eines Bezeichners auf das Vorkommen in der Menge abzuprüfen.

Werkzeugübergreifende Namensprüfung

Die Namenskonventionen aus (***iv***) und (***v***) sind durchaus komplexer, jedoch vom Prüfalgorithmus her identisch aufgebaut. Beide Richtlinien erfordern, dass die Bezeichner nur aus einer bestimmten Untermenge an erlaubten Zeichenketten aufgebaut sind. Dies wäre durch einen Abgleich von den in einem Glossar hinterlegten Akronymen, wie in der Abbildung 72 dargestellt, oder aber durch die in einem digitalen (deutschen/englischen) Wörterbuch hinterlegten Worte gegeben. Da Bezeichner aus mehreren Worten oder Akronymen aufgebaut sein könnten, muss der Algorithmus die Bezeichner-Zeichenkette zunächst in ihre einzelnen Worte zerlegen. Die Zerlegung muss durch vordefinierte Trennzeichen erfolgen wie beispielsweise ein großgeschriebenes Zeichen, Minus, Punkt oder Unterstrich. Beispielsweise würde `WLC_Window_Lifter_Control` zerlegt in die Worte `WLC`, `Windows`, `Lifter`, `Control`. In der Tabelle 31 ist die für (***iv***) und (***v***) entwickelte Richtlinie zur Sicherstellung bekannter Akronyme sowie Verwendung einheitlicher Sprache (z. B. Deutsch/Englisch) nach einem Muster der Form *Aa_Bb[[Cc .. Cc][_Bb][Cc .. Cc]]]_Dd* als Regel dargestellt, welche die Konformität zur aufgestellten Forderung sicherstellt.

Tabelle 31: Systementwurf - Richtlinie für Änderbarkeit & Wiederverwendbarkeit

ID: 7	**Artefakt:** ASCET-MD, Excel-Glossar	
Prozessphase:	*Funktionaler und technischer Systementwurf*	
Titel:	***Erlaubte Akronyme in Bezeichnern***	
Kategorie:	*Erhöhung der Änderbarkeit und Wiederverwendbarkeit*	**Priorität:** *Empfohlen*
Vorbedingung:	$ID_n \neq \emptyset$, (*Der Bezeichner ist nicht leer.*)	
Richtlinie:	*Alle Akronyme in Bezeichnern mit dem Glossar konsistent.*	
Beschreibung:	*Ausschließlich die im Glossar für Funktionsnamen definierten Akronyme sind in Bezeichnern zu verwenden. Dabei gilt folgende Namenskonvention für die Verwendung der Akronyme:* *Formatierung nach dem Muster* Aa_Bb[[Cc .. Cc][_Bb][Cc .. Cc]]]_Dd *wobei die geklammerten Ausdrücke optional und wiederholbar sind:*	

Konformität:

$$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\text{IDo}, \ldots, \text{ID}n+1\} \end{cases}$$

Ausführbare Regel in LINQ:

Regel für ein ASCET-Modell und ein Glossar in Excel:

```
// Section 1 -- get all model-elements
var Elements = from c in
Artefact["Ascet-Model"].Descendants("Element")
               where c.Attribute("name") != null
               select c;

// Section 2 -- get and group all acronyms from glossary
List<XElement> Query = new List<XElement>();
List <XElement > Querytmp = new List<XElement>();

//Get AA list from glossary
var aas = from c in Artefact["Excel-Glossary"].Descendants("Item")
          where c.Parent.Attribute("name").Value == "Aa"
          select c.Attribute("value").Value;

//Get BB list from glossary
var bbs = from c in Artefact["Excel-Glossary"].Descendants("Item")
          where c.Parent.Attribute("name").Value == "Bb"
          select c.Attribute("value").Value;

//Get CC list from glossary
var ccs = from c in Artefact["Excel-Glossary"].Descendants("Item")
          where c.Parent.Attribute("name").Value == "Cc"
          select c.Attribute("value").Value;

//Get DD list from glossary
var dds = from c in Artefact["Excel-Glossary"].Descendants("Item")
          where c.Parent.Attribute("name").Value == "Dd"
          select c.Attribute("value").Value;


// Section 3 – analyze single words in identifiers

Regex letterAndUppercase = new Regex("[0-9A-Z]");

foreach (XElement xele in Elements) // all named elements
{
    string testString = xele.Attribute("name").Value;
    string[] splitResult = testString.Split('_');

    XElement ParentEl = xele.Parent.Parent;
    XElement ParentElout = new XElement (xele.Parent.Parent);
    XElement xresult = new XElement(xele);
    xresult.SetAttributeValue("ParentDiagramOID",
                              ParentEl.Attribute("OID").Value);



    if (!Querytmp.Contains(ParentEl))
    {
       Querytmp.Add(ParentEl);
       Query.Add(ParentElout);
    }

    //check otional condition: if there are 3 or 4 parts
    if (!(splitResult.Count() == 3 || splitResult.Count() == 4))
    {
        Query.Add (xresult); continue;
    }

    //check AA part
    if (!aas.Contains(splitResult[0]))
    {
        Query.Add(xresult); continue;
    }
    //check DD part
    if (!mdds.Contains(splitResult.Last()))
    {
        Query.Add(xresult); continue;
    }
```

```
    //check BB part
    string temp = splitResult[1];  int i = 0;
    for (; i < temp.Length; i++)
    {
        if (letterAndUppercase.IsMatch(temp[i].ToString ())){break;}
    }
    string temp1 = temp.Substring(0, i);
    if (!bbs.Contains(temp1))
    {
        Query.Add(xresult); continue ;
    }
    //check CC part
    else
    {
        if (Query.Contains (xele)) continue;
        string temp2 = temp.Substring(i);
        while (temp2 != null && temp2 !="")
        {
            int ii = 1;
            for (; ii < temp2.Length; ii++)
            {
              if (letterAndUppercase.IsMatch(temp2[ii].ToString ()))
              {break;}
            }
            string temp3 = temp2.Substring(0, ii);
            if (!ccs.Contains(temp3))
            {
                Query.Add(xresult);
            }
            temp2 = temp2.Substring(ii);
        }
    }

  if (splitResult.Count() == 4)
    {
        //check DD part
        string stemp = splitResult[2]; int si = 0;
        for (; si < stemp.Length; si++)
        {
            if (letterAndUppercase.IsMatch(stemp[si].ToString()))
            {break;}
        }
        string stemp1 = stemp.Substring(0, si);
        if (!bbs.Contains(stemp1))
        {
            Query.Add(xresult); continue;
        }
        else
        {
            if (Query.Contains(xele))
                continue;
            string stemp2 = stemp.Substring(si);
            while (stemp2 != null && stemp2 != "")
            {
                int sii = 1;
                for (; sii < stemp2.Length; sii++)
                {
                    if (letterAndUppercase.IsMatch(
                                          stemp2[sii].ToString()))
                    { break; }
                }
                string stemp3 = stemp2.Substring(0, sii);
                if (!ccs.Contains(stemp3))
                {
                    Query.Add(xresult);
                }
                stemp2 = stemp2.Substring(sii);
            }
        }
    }
}

if(Query.Count() >0)  Result = "FAIL"; else Result = "PASS";
```

Der implementierte Algorithmus ist grob in drei Sektionen (*Sections*) untergliedert. Zunächst werden in *Section 1* alle betreffenden, benannten Bezeichner aus dem ASCET-Modell

abgefragt und als Menge in `Elements` gespeichert. In *Section 2* werden pro Wortgruppe *Aa, Bb, Cc* und *Dd* die einzelnen Akronyme aus dem Glossar (Abbildung 72) gelesen und den jeweiligen Mengen `aas`, `bbs`, `ccs` und `dds` zugeordnet. Schließlich wird innerhalb der *Section 3* die hauptsächliche Untersuchung durchgeführt. Zunächst wird hier ein regulärer Ausdruck `[0-9A-Z]` definiert anhand dessen Großbuchstaben und Zahlen innerhalb eines Bezeichners gefunden werden können. Die Auffindung dient zur Silbentrennung bei Großbuchstaben, Zahlen und dem Unterstrich. Im restlichen Verlauf erfolgt sodann die Einzeluntersuchung des Bezeichners bzgl. der Wortgruppen. Eine besondere Beachtung findet auch das optionale Vorkommen von Akronymen. Die Richtlinie gibt die Menge der nicht im Glossar enthaltenden Elemente zurück.

Funktionale Sicherheit

Die funktionale Sicherheit im Sinne der Sicherstellung der Funktionsausführung in einem Modell ist durch sehr umfangreiche Artefakt-Abfragen bestimmt. Da die Fahrzeugfunktion APR im sicherheitskritischen Einsatzbereich liegt, ist diese Betrachtung notwendig. Einige sicherheitskritische Aspekte wurden mit dem Hinweis auf den MISRA-Standard bereits erwähnt. Die Abbildung aller MISRA-Regeln wurde während der Arbeit auf Modell-Basis untersucht. Die Ergebnisse würden in dieser Arbeit zu weit führen, jedoch soll die Machbarkeit einiger grundlegender Absicherungen für den modellbasierten Systementwurf des APR an zwei Beispielen exemplarisch durchgeführt werden.

Hierzu gehört die folgende Sicherstellung im Implementierungsmodell des APR:

i. *Absicherung der korrekten Einstellungen für den Code-Generator*

ii. *Ausschließliche Verwendung erlaubter Datentypen*

iii. *Absicherung von maximalen Wertebereichen für Parameter und Variablen*

iv. *Vermeidung von Divisionen durch Null*

Richtlinie: Grundsätzlich sollten in Implementierungsmodellen nur lokale Variable benutzt werden. Für die globale Kommunikation müssen sogenannte Nachrichten (*Messages*) eingesetzt werden, da diese die Datenkonsistenz durch das Prinzip des Kopiermechanismus sicherstellen.

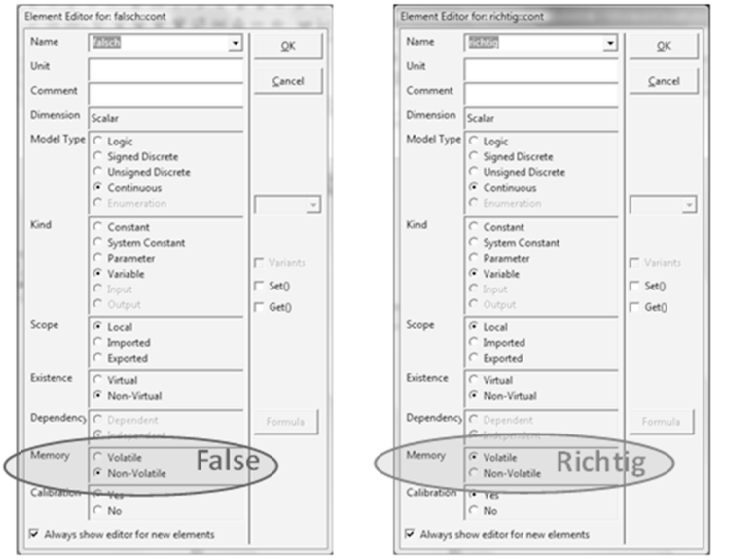
Die Tabelle 32 zeigt die für (***i***) entwickelte Richtlinie zur Code-Generator-Einstellung für jedes Element in ASCET-MD als ausführbare Regel, welche eine Forderung aus (***iii***) zur Einstellung des Gültigkeitsbereichs von Variablen sicherstellt.

Tabelle 32: Systementwurf - Funktionsabsicherung und Zuverlässigkeit

ID: 8	**Artefakt:** ASCET-MD		
Prozessphase:	*Funktionaler und technischer Systementwurf*		
Titel:	***Gültigkeitsbereich für Variable***		
Kategorie:	*Funktionsabsicherung und Zuverlässigkeit*	**Priorität:**	*Erforderlich*
Vorbedingung:	*Keine. Einstellungswert ist immer vom Programm vorgegeben.*		
Richtlinie:	*Die Memory-Einstellung ist immer auf Volatile einzustellen.*		

Beschreibung:

Variablen liegen in der Regel im flüchtigen Speicher, sodass in der Einstellungsoption „Memory“ die standardmäßige Einstellung auf „Volatile“ eingestellt werden muss.



Anmerkung: Die Option „Non-Volatile“ dürfte nur gesetzt sein, wenn die Variablen nach dem Abschalten des Steuergerätes nicht aus dem flüchtigen Speicher gelöscht werden sollen.

Konformität:

$$\Sigma \begin{cases} true = \emptyset, \\ false, sonst \end{cases}$$

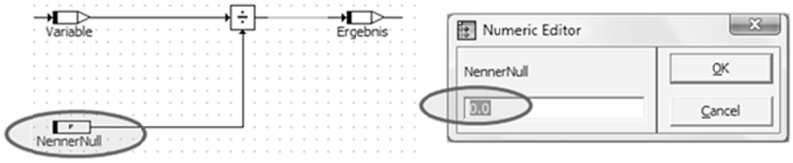
Ausführbare Regel in LINQ:

```
var Query = from c in Artefact["Ascet-Model"]
                    .Descendants("Elements").Descendants("Element")

            where c.Descendants("PrimitiveAttributes").Count()>0
            && c.Descendants("PrimitiveAttributes").First()
                .Attribute("kind").Value == "variable"
            && c.Descendants("PrimitiveAttributes").First()
                .Attribute("volatile").Value == "false"
          select c;

if(Query.Count() >0) return false; else return true;
```

Der hierbei implementierte Algorithmus lässt sich sehr kompakt formulieren. Das Implementierungsmodell wird im Wurzelknoten `Elements` traversiert, wobei zwei Bedingungen geprüft werden: Einerseits muss es sich um Typ-Variable handeln (*Filterung*), was mittels Attribut-Prüfung des `kind`-Attributs gelingt, und andererseits ist die Programmeinstellung für die Memory-Einstellung auf `volatile == false` gesetzt. Ergibt die Prüfung eine leere Menge, wurden keine falschen Einstellungen entdeckt und die Rückgabe kann mit dem Wahrheitswert erfolgen.

Analog zu dieser Prüfung funktioniert auch die Abfrage für (***ii***) und (***iii***) zur Sicherstellung der Wertebereiche oder Datentypen, mit dem Unterschied, dass andere Attribute geprüft und verglichen würden. Etwas komplexer ist hingegen die Überprüfung von Richtlinie (***iv***). Die Richtlinie in Tabelle 33 zeigt die entwickelte Regel zur Prüfung auf Division durch Null.

Tabelle 33: Systementwurf - Richtlinie für Benutzbarkeit & Verständlichkeit

ID: 9	**Artefakt:** ASCET-MD
Prozessphase:	*Funktionaler und technischer Systementwurf*
Titel:	***Vermeidung von Division durch Null***
Kategorie:	*Funktionsabsicherung und Zuverlässigkeit* **Priorität:** *Erforderlich*
Vorbedingung:	∃ d ∈ Divisoren
Richtlinie:	*Der Nenner darf in Divisionen nicht Null annehmen.*
Beschreibung:	*Bei der Verwendung von Divisionen innerhalb von Berechnungen ist darauf zu achten, dass der Nenner ungleich Null ist.* 
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\mathrm{ID}o, \ldots, \mathrm{ID}n+1\} \end{cases}$
Ausführbare Regel in LINQ:	

```
// Section 1 - Find all division elements in model
var divisors = from c in Artefact["Ascet-Model"]
                    .Descendants("DiagramElements")
                    .Descendants("DiagramElement")
                    .Descendants("Operator")
               where (c.Attribute("type").Value == "/")
      select c;

// Search for divisions by zero
List <XElement > Query = new List<XElement> ();
foreach(XElement xVariable in divisors)
{
// Section 2 - Get denominator conection line from each division

var denominators = from c in xVariable.Descendants("Interfaces")
                                      .Descendants("ReturnPort")
                                      .Descendants("SelectorPort")
                   where c.Attribute("index").Value == "1"
     select c;

foreach(XElement xVariable2 in denominators)
{
string outElConn= "End"; string inElConn= "Start";

// Section 3 - Get connection lines elementc for selected division

var connections = from c in xVariable.Parent.Parent.Descendants()
                  where c.Name == "Connection"
select c;

foreach(XElement xVariable3 in connections)
{ //Get denominator simple element from each division

if (xVariable2.Attribute("graphicOID").Value
== xVariable3.Element(outElConn).Attribute("graphicOID").Value)
{
var qtemp02  = from c in Artefact["Ascet-Model"]
                         .Descendants("DiagramElements")
                         .Descendants("DiagramElement")
                         .Descendants("SimpleElement")

               where c.Descendants("Interfaces")
                      .Descendants("ReturnPort").Count() > 0

               && ( c.Descendants("Interfaces")
                     .Descendants("ReturnPort")
```

```
                        .First().Attribute("graphicOID").Value
                        == xVariable3.Element(inElConn)
                        .Attribute("graphicOID").Value)
select c;

foreach(XElement xout  in qtemp02)
//Get value of the found denominator
{
var elements = from c in Artefact["Ascet-Model"]
                          .Descendants("ComponentData")
                          .Descendants("DataSet")
                          .Descendants("DataEntry")

              // Check for divison by zero

               where c.Attribute("elementOID")
                      .Value == xout.Attribute("elementOID").Value
                && c.Descendants("Numeric").Count()>0
                && c.Descendants("Numeric")
                     .First().Attribute("value").Value == "0.0"

         select c;

if (elements.Count() > 0) //Add denominator with value = 0 is found
{
  XElement ParentEl = xout.Parent.Parent.Parent.
                     Parent.Parent.ElementsAfterSelf("ComponentMain")
                     .First().Element("Component");

  XElement xresult = new XElement(xout);
  xresult.SetAttributeValue("ParentDiagramOID",
  ParentEl.Attribute("OID").Value);
  Query.Add(xresult);
 }}
break;
}}}}
  if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

In diesem Algorithmus wird aus dem Modell zunächst die Menge `divisors` aller Operatoren gebildet die Divisionen sind. Ausgehend von diesen Divisionen werden die „Eingänge“ der Divisionsblöcke untersucht – d. h., es muss im Modell zugeordnet werden welcher Eingang eines Divisionsblocks der Zähler und welcher der Nenner ist. Hierfür werden die Mengen `denominators` und `connections` gebildet. Schließlich lässt sich die Untermenge aller Nenner mittels `Attribute("value").Value == "0.0"` (Attribut-Werteprüfung) auf eine Nulldivision hin abprüfen. Im letzten Teil wird eine Menge `elements` fehlerhafter Divisoren aufgebaut. Ergibt die Prüfung eine leere Menge, wurde keine Nulldivision entdeckt.

Effiziente Modellierung

Die Effizienzsteigerung innerhalb der Modellierung der APR adressiert ein umfangreiches Spektrum an Möglichkeiten. Es soll daher hier nur auf einen kleinen Ausschnitt, der Betrachtung von Rechenoperationen der APR-Software, eingegangen werden und exemplarisch eine Regel entwickelt werden. Innerhalb der Machbarkeitsanalyse wurden auch weitere Regeln zur Effizienzerhöhung implementiert, die nachfolgend beschrieben sind.

Richtlinie: Zur Effizienzbetrachtung zählt bei der Modellierung von Implementierungsmodellen meist der generierte C/C++-Code. Dieser sollte nicht zu umfangreich sein und effizient auf dem Target (Mikrocontroller) ausgeführt werden können. Eine wesentliche Betrachtung liegt somit auf den modellierten Berechnungs-Operationen.

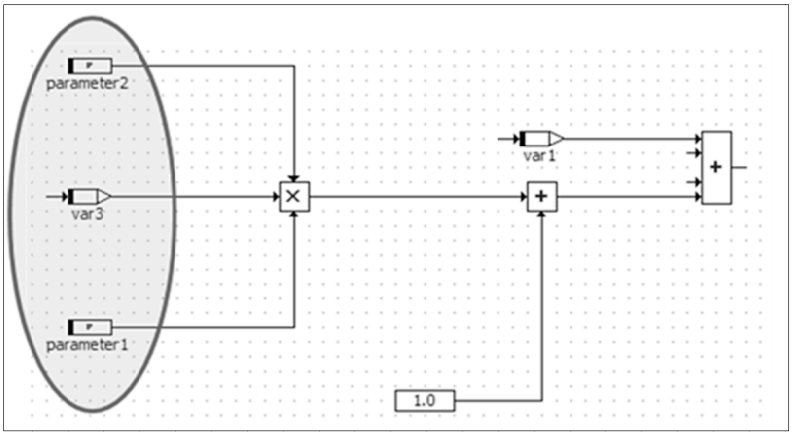


Abbildung 76: Multiple Eingänge am Operator im Modell

Für Berechnungsfunktionen sind daher Regeln zur Verbesserung der Effizienz anzuwenden. Arithmetische Operationen, wie in der Abbildung 77 gezeigt, mit den Basiselementen Addition, Subtraktion, Multiplikation oder Division, werden aus einem Implementierungsmodell nicht effizient generiert sollten mehr als zwei Eingänge verwendet werden.

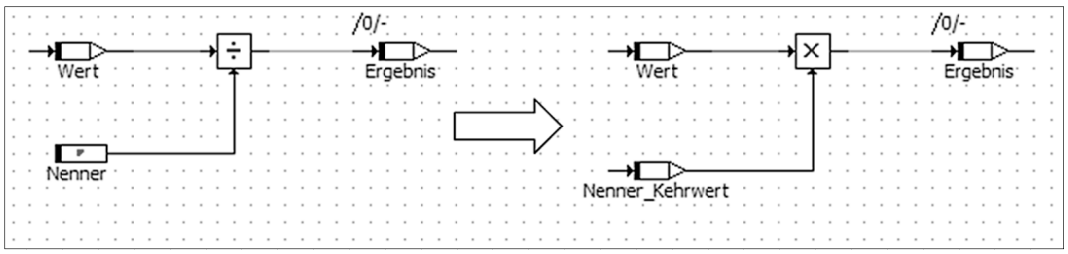


Abbildung 77: Auflösung von Divisions-Operatoren

Divisionen sind beispielsweise meist sehr rechenintensiv in ihrer Ausführung und damit nicht effizient. Teilweise benötigen sie doppelt so viele Taktzyklen wie eine Multiplikation. Eine Auflösung einer Divisionsberechnung zeigt die Abbildung 77. Hier wird die Umwandlung eines Parameters in seinen Kehrwert vorgenommen. Die Umwandlung fester Parameterwerte für Reibung der Fensterscheibe $\boldsymbol{x = 20}$ wird somit zu $\boldsymbol{1 / x = 0.5}$ um eine Divisionsberechnung zu vermeiden.

Richtlinie: Des Weiteren kann es bei Divisionen aufgrund interner Rundungen bei Berechnung zu Genauigkeitsverlusten kommen. Divisions-Operationen sollten im Implementierungsmodell sparsam eingesetzt werden. Zur Vermeidung von Rundungsfehlern sollte der Zähler bedeutend größer als der Nenner sein. Der Nenner wiederum sollte nicht zu klein werden. In arithmetischen Rechenketten sollen die Divisionen möglichst weit hinten ausgeführt werden, um unnötige ‚Shift'-Operationen zu vermeiden. Dabei muss immer auch auf die benötigten Wertebereiche für Zwischengrößen (temporäre Variable) geachtet werden. Besonders bei aufwendigen Berechnungen sind mehrere Operationen im Ausdruck enthalten.

Zur Vermeidung von erneuten Berechnungen in einer komplexen Berechnungskette können

a) temporäre Variablen,

b) prozesslokale oder methodenlokale Variablen

c) *sowie lokale Variablen*

zur Zwischenspeicherung von Werten eine enorme Effizienzsteigerung hervorrufen. Zudem sind sie kompakt im Modell darstellbar.

Der zusätzliche Speicherbedarf steigt hierbei, jedoch nicht bei der expliziten Angabe, dass diese Variable temporär gehalten wird.

Bevor diese Richtlinie implementiert wird, seien daher Vor- und Nachteile in der folgenden Tabelle aufgeführt.

Tabelle 34: Systementwurf - effiziente Verwendung von Variablen

a) Temporäre Variablen	
Vorteil: • Schnelle Zugriffszeit (Registergröße) • Grafisch kompakt darstellbar	Nachteil: • Nicht implementierbar und damit nicht an einer anderen Stelle verwendbar
b) Prozesslokale oder methodenlokale Variablen	
Vorteil: • Schnelle Zugriffszeit (Registergröße) • Implementierbar • Mehrfach verwendbar	Nachteil: • Nicht applizierbar • Stack-Verwaltung
c) Lokale Variablen	
Vorteil: • Implementierbar • Applizierbar • Adressierbar • Prozessübergreifend einsetzbar	Nachteil: • Speicher-Bedarf permanent

Das Modell in der Abbildung 78 zeigt einen solchen Fall der mehrfachen Berechnung, die sich durch die Abzweigung der Konnektoren nach der Addition des Offsets durch die Multiplikation der Parameter *Param1*, *Param2* und *Param3* ergibt. Darunter ist das um eine Variable zur Zwischenwertspeicherung erweiterte Modell angegeben – zur Berücksichtigung der Richtlinie zur Vermeidung von Mehrfachberechnungen.

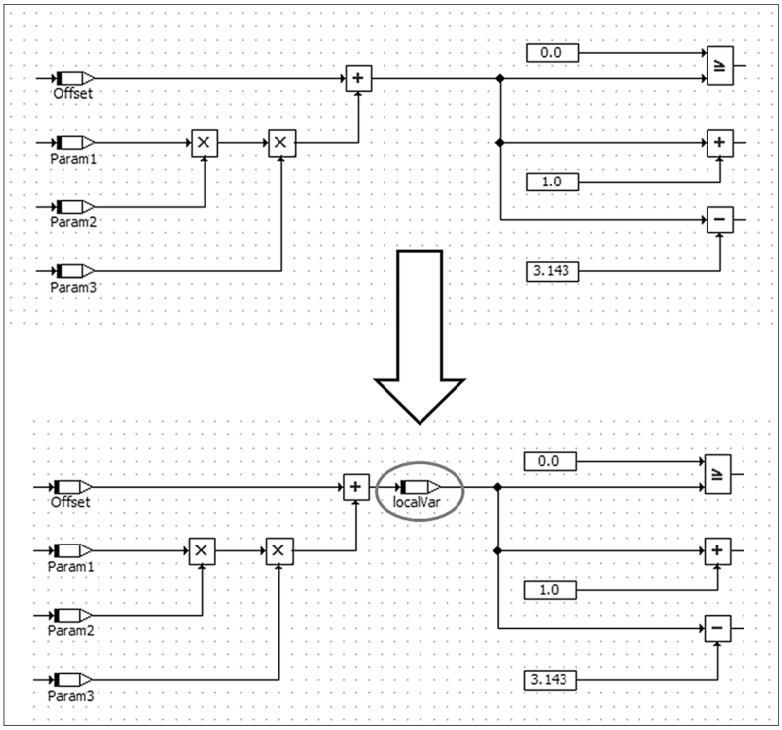


Abbildung 78: Vermeidung von Mehrfachberechnung

Die Tabelle 35 zeigt die aus der Richtlinie exemplarisch entwickelte Regel zur Vermeidung von Mehrfachberechnungen in Modellen, welche die Konformität zur aufgestellten Forderung sicherstellt.

Tabelle 35: Systementwurf - Richtlinie für Verbesserung der Effizienz

ID: 10	**Artefakt:** ASCET-MD	
Prozessphase:	*Funktionaler und technischer Systementwurf*	
Titel:	***Vermeidung von Mehrfachberechnungen***	
Kategorie:	*Verbesserung der Effizienz*	**Priorität:** *Empfohlen*
Vorbedingung:	–	
Richtlinie:	*Mehrfache Berechnungen sollen mittels zusätzlicher Speicherung von Zwischenergebnissen in Variablen aufgelöst werden.*	
Beschreibung:	*Besonders bei aufwendigen Berechnungen sind mehrere Operationen im Ausdruck enthalten. Zur Vermeidung von erneuten Berechnungen in einer komplexen Berechnungskette können temporäre Variablen, prozesslokale oder methodenlokale Variablen sowie lokale Variablen zur Zwischenspeicherung von Werten eine enorme Effizienz-steigerung hervorrufen. Zudem sind sie kompakt darstellbar.*	

Konformität:

$$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\mathrm{IDo}, \ldots, \mathrm{ID}n+1\} \end{cases}$$

Ausführbare Regel in LINQ:

```
//Get all connections
var connects = from c in Artefact["Ascet-Modell"]
                        .Descendants("DiagramElement")
               where c.Descendants("Connection").Count ()>0
               select c;

List<XElement > Query = new List<XElement> ();
List<XElement > Querytmp = new List<XElement> ();

foreach (XElement x1 in connects)
{ string s1 = x1.Element("Connection").Element("Start")
.Attribute("graphicOID").Value;

foreach (XElement x2 in x1.Parent.Elements())
{
if (x2 != x1 && x2.Elements("Connection").Count() > 0)
{
    string s2 = x2.Element("Connection").Element("Start")
                  .Attribute("graphicOID").Value;
    if (s1 == s2)
    {
        var op = from c in x1.Parent.Descendants("Operator")
                 where c.Descendants ("ReturnPort").Count() > 0
                 && c.Descendants ("ReturnPort").First()
                 .Attribute("graphicOID").Value == s1
                 && c.Attribute("tempVar").Value == "false"
                 select c;

if (op.Count() > 0)
{
foreach (XElement xout in op)
 {
  XElement ParentEl = xout.Parent.Parent.Parent.Parent.Parent
  .ElementsAfterSelf("ComponentMain").First().Element("Component");
  XElement xresult = new XElement(xout);
  xresult.SetAttributeValue("ParentDiagramOID",
  ParentEl.Attribute("OID").Value);
  xresult.SetAttributeValue("name", "Operator " +
  xout.Attribute("type").Value);

  if (!Query.Contains(ParentEl)){Query.Add(ParentEl);}
  if (!Querytmp.Contains(xout)){  Querytmp.Add(xout);

  Query.Add(xresult);
} }} }} }}

if(Query.Count() >0) Result = "WARNING"; else Result = "PASS";
```

Der entwickelte Algorithmus besteht aus zwei wesentlichen Teilen. Zunächst werden im Modell alle Konnektoren (`DiagrammElement` mit der Eigenschaft, dass sie eine `Connection` besitzen) in einer Abfrage in die Menge `connects` aufgenommen. Danach wird jedes einzelne Element `x1` und nachfolgende Element `x2` in der Menge `connects` untersucht. Haben `x1` und `x2` gleiche Startwerte (linke Wertbelegung des Konnektors), so kann eine Abzweigung vorliegen. Eine genauere Untersuchung erfolgt in diesem Fall in der Abfragemenge `op`. Hier wird auch die rechte Wertbelegung auf Operatoren untersucht, ob der Anfangswert für eine Neuberechnung im Operator-Eingang Verwendung findet. Ist dies der Fall – und keine temporäre Variable wurde verwendet – wird das fehlerhafte Operator-Element in die Ergebnismenge `Query` gespeichert. Ist die Ergebnismenge nicht leer, so existieren Mehrfachberechnungen im Modell.

Konformitätsprüfung der Modulspezifikation

Jeder Algorithmus in einem Modul arbeitet mit sogenannten Elementen. Ein Element enthält eine Datenangabe und stellt eine Schnittstelle für den Zugriff auf seine Daten oder die

Rückgabe des Wertes einer Berechnung (z. B. Interpolation einer Kennlinie) zur Verfügung. Jedes Element ist von einem festen Typ (Datentyp), d. h., Elemente sind immer streng typisiert. Da mehr als nur ein einziges Element von einem gegebenen Typ existieren kann, wird ein Element als eine Instanz eines gegebenen Typs bezeichnet.

Richtlinie: Im Modellierungswerkzeug kann die Variablenbelegung für Elemente und Konstante erfolgen. Zudem können neben Datentypen auch Wertebereiche definiert werden. Die Angabe von Maxima und Minima muss dabei konform gemäß den Vorgaben erfolgen.

Die Tabelle 36 zeigt die exemplarisch entwickelte Regel der Richtlinie zur Vermeidung von unendlichen Wertebereichsgrenzen in Modellen, welche die Konformität zur aufgestellten Forderung sicherstellt.

Tabelle 36: Modulspezifikation - Funktionsabsicherung & Zuverlässigkeit

ID: 11	**Artefakt:** ASCET-MD
Prozessphase:	*Modulspezifikation und Codegenerierung*
Titel:	***Vermeidung von infiniten Wertebereichsgrenzen***
Kategorie:	*Funktionsabsicherung und Zuverlässigkeit* **Priorität:** *Empfohlen*
Vorbedingung:	$\exists$ e $\in$ *ElementImplementation*
Richtlinie:	*Die Angabe von Maxima und Minima für Variablen sind in der Beachtung des angegebenen Datentyps einzuhalten.*
Beschreibung:	*Nachfolgende Abbildung zeigt die Einstellungen für Implementierungsvariable, die fälschlicherweise für Minima- und Maxima-Einstellungen infinite Grenzen verwenden.*

Elements | Data | Implementation | Layout

Impl

Name	Type	Impl. Type	Impl. Min	Impl. Max	Q	Formula	Limit to ma bit length	Limit Assignmer	Zero not incl.	Min	Max	Memory Location	Cache Locking
BlnkTime	cont	real64	-oo	+oo	0	ident	No		No	-oo	+oo	Default	Automatic
error	log	int8	0	1	1	ident		Yes	No	0	1	Default	Automatic
geen	log	int8	0	1	1	ident		Yes	No	0	1	Default	Automatic
GeenTime	cont	real64	-oo	+oo	0	ident	No		No	-oo	+oo	Default	Automatic
red	log	int8	0	1	1	ident		Yes	No	0	1	Default	Automatic
RedTime	cont	real64	-oo	+oo	0	ident	No		No	-oo	+oo	Default	Automatic
sn	udisc	uint8	0	10	1	ident	Auto		No	0	10	Default	Automatic
Timer	Timer	Impl							-			Default	Automatic
yellow	log	int8	0	1	1	ident		Yes	No	0	1	Default	Automatic
YellowTime	cont	real64	-oo	+oo	0	ident	No		No	-oo	+oo	Default	Automatic

Konformität:	$\Sigma \begin{cases} true = \emptyset, \\ false, sonst \end{cases}$
Ausführbare Regel in LINQ:	

```
var Query = from c in Artefact["Ascet-Modell"]
                      .Descendants("ElementImplementation")

            where c.Descendants("ImplementationInterval").Count() > 0
                  && c.Descendants("ImplementationInterval")
                      .First().Attribute("min").Value == "INF" ||
                  && c.Descendants("ImplementationInterval")
                      .First().Attribute("max").Value == "INF"
            select c;

if(Query.Count() >0) return false; else return true;
```

Aus dem Artefakt werden alle diejenigen Elemente herausgesucht, die zur Menge `ElementImplementation` zählen. Minima- (`min`) und Maxima- (`max`) Wertbelegungen werden durch einen logischen Wahrheitsausdruck auf das Vorkommen von infiniten Angaben vergleichen. Wurden Elemente gefunden, ist die Ergebnismenge `Query` nicht leer, wird als Ergebnis `false` zurückgegeben, was einen Fehlschlag nach Prüfung bedeutet.

Richtlinie: Eine andere Prüfung auf Gültigkeitsbereiche sind oftmals verwaiste oder fehlerhafte Referenzierungen. Entnommen aus der ARTOSAR Software Component Template V3.0.1 Beschreibung lässt sich die dort formulierte Richtlinie wie folgt als Regel umsetzen. Die Tabelle 37 zeigt die exemplarisch entwickelte Regel zur Vermeidung von falscher Referenzauflösung, welche die Konformität zur aufgestellten Forderung sicherstellt.

Tabelle 37: Modulspezifikation - Sicherstellung der Konsistenz

ID: 12	**Artefakt:** AUTOSAR
Prozessphase:	*Modulspezifikation und Codegenerierung*
Titel:	***Referenzauflösung und Zugehörigkeit***
Kategorie:	*Konsistenz* **Priorität:** *Erforderlich*
Vorbedingung:	–
Richtlinie:	*Jedes RTEEvent muss eine Referenz zu einem RunnableEntity implementieren, das zur gleichen internen Verhaltensbeschreibung InternalBehavior gehört, wie das RTEEvent selbst.*
Beschreibung:	*Eine interne Verhaltensbeschreibung InternalBehavior kann verschiedene Ereignisse sowie RunnableEntitys enthalten. Die Ereignisse werden für die ausführbaren Entitäten definiert. Jedoch muss die Elternbeziehung zum InternalBehavior korrekt zu den einzelnen ausführbaren Entitäten referenziert sein, wie in der nachfolgenden Abbildung als Beispiel gezeigt.*
Konformität:	$\Sigma \begin{cases} true = \emptyset, \\ false, sonst \end{cases}$

Ausführbare Regel in LINQ:

```
//AUTOSAR Guideline
var rteevents = from c in Artefact["SGL-SWC"].Descendants()
                where c.Name.LocalName == "EVENTS"
                select c;

int index = 0;
List<XElement> Query = new List<XElement>();
string nameSpace = "{http://autosar.org/2.1.2}";
foreach (XElement events in rteevents)
{
foreach (XElement rteevent in events.Elements())
{
if (rteevent.Element(nameSpace + "START-ON-EVENT-REF") != null)
{
    string refrence = rteevent.Element(nameSpace
        + "START-ON-EVENT-REF").Value;
    //get path of this event
    string path = string.Empty;
    XElement temp = events;
    while (temp.Parent.Name.LocalName != "AUTOSAR")
    {
        if (temp.ElementsBeforeSelf(nameSpace
                                     + "SHORT-NAME").Count() > 0)
        {
            XElement tempName
                = temp.ElementsBeforeSelf(nameSpace
                                  + "SHORT-NAME").First();
            path = "/" + tempName.Value + path;
        }
        temp = temp.Parent;
    }
    if (!refrence.Contains(path))
    {
        Query.Add(rteevent);
        index++;
        ResultMessage = ResultMessage + "Xml Element " +
            index + ":path of this event (" + refrence + ")
        could not be found, this event is under path "
        + path + ".\n";
    }
    else
    {
        string runableEntity = refrence.
                          Substring(refrence.LastIndexOf("/") + 1);

        bool foundEntity = false;

        foreach (XElement entity in
                    events.ElementsAfterSelf(nameSpace
                                                + "RUNNABLES")
                      .First().Elements(nameSpace
                                        + "RUNNABLE-ENTITY"))
        {
            if (entity.Element(nameSpace + "SHORT-NAME")
                       .Value == runableEntity)
            {
                foundEntity = true;

                break;
            }
        }

        if (!foundEntity)
        {
            Query.Add(rteevent);
            index++;
            ResultMessage = ResultMessage + "Xml Element "
            + index + ":the runnable-entity with the short name "
            + runableEntity
            + " which this event references could not be found unter
            this INTERNAL-BEHAVIOR.\n";
        }
    }}} }

if(Query.Count() >0) return false; else return true;
```

Der implementierte Algorithmus erzeugt zunächst eine Menge `rteevents` in der sämtliche RTE-Ereignisse (`EVENTS`) des Artefakts enthalten sind. Danach wird für jedes Ereignis innerhalb der internen Verhaltensbeschreibung mit einer *Contains*-Abfrage binnen der Menge geprüft, ob die Referenz gefunden wird. Zudem wird in diesem Beispiel eine Ergebnisnachricht `ResultMessage` dynamisch aufgebaut, um bei Fehlschlagen der Prüfung die entsprechenden Elemente im Prüfbericht anzeigen zu können.

Konformitätsprüfung der Testspezifikation

Bei der Erstellung von Testfallspezifikationen mit dem Werkzeug CTE/XL mittels der Klassifikationsbaummethode sind mehrere Regeln für die APR-Funktion einzuhalten. Die Regelprüfung auf Klassifikationsbäumen wird nachfolgend exemplarisch vorgestellt.

Setzen globaler Parameter

Richtlinie: Viele Fehler beim Modellieren von Klassifikationsbäumen beruhen auf Parameter, die nicht angegeben werden. Das Weglassen eines *Teststep*-Werts führt beispielsweise beim automatischen Ausführen einer Testfallspezifikation (z. B. mit MTest) zu einem Fehler, wie die Abbildung 79 zeigt.

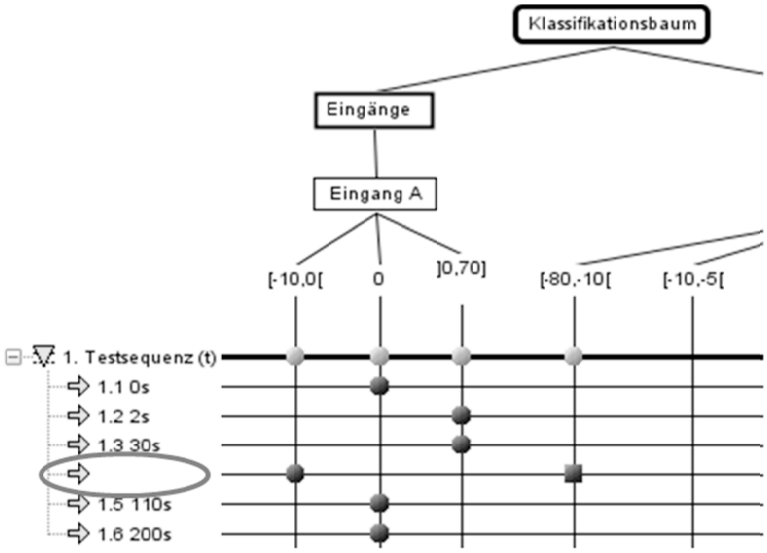


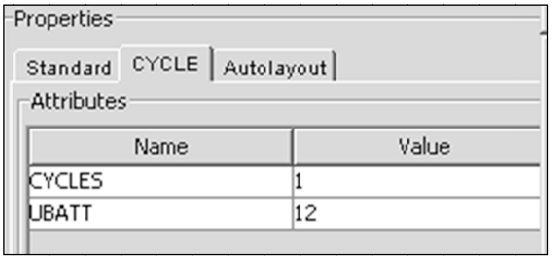
Abbildung 79: Fehlende Wertebelegungen führen zu Fehlern

Zunächst muss zur Feststellung der Konformität eines Klassifikationsbaums eine Überprüfung der Wurzelknoten erfolgen. Unter dem Wurzelknoten werden alle Daten spezifiziert, die für die gesamte Testspezifikation gültig sind. Insbesondere sind hier also globale Parameter abzulegen, die für eine Testdurchführung relevant sind aber nicht mehr explizit im CTE-Baum auftreten. Im APR hat beispielsweise die Batteriespannung für alle beteiligten Steuergeräte die konstante $\boldsymbol{U_{batt} = 12Volt}$. Dieser Wert ist für die Testspezifikation und ggf. Testausführung an einem HiL-Prüfstand immer konstant einzustellen.

Die Tabelle 38 zeigt die exemplarisch entwickelte Regel zur Richtlinie für die konforme Einstellung globaler Parameter im Wurzelbaum, welche die Konformität zu der aufgestellten Forderung sicherstellt.

Tabelle 38: Testspezifikation - Funktionsabsicherung & Zuverlässigkeit

ID:	13	**Artefakt:**	Klassifikationsbaum/CTE XL

Prozessphase:	*Testspezifikation*
Titel:	***Setzen globaler Parameter***
Kategorie:	*Funktionsabsicherung und Zuverlässigkeit* **Priorität:** *Empfohlen*
Vorbedingung:	∃ root ∈ *Klassifikationsbaum*
Richtlinie:	*Die Batteriespannung ist für die Testausführung als konstanter Wert global vorzugeben.*
Beschreibung:	*Zur Testdurchführung sind in der Registerkarte CYCLE alle globalen Parameter wie z. B. die Batteriespannung vorzugeben. Nachfolgende Abbildung zeigt ein Beispiel.* 
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\text{ID}o, \ldots, \text{ID}n + 1\} \end{cases}$

Ausführbare Regel in LINQ:	`var Query = from c in Artefact["cte"]`
	`.Descendants("tree")`
	`.Descendants("activetag")`
	
	`where c.Parent.Attribute("type").Value == "root" &&`
	`c.Attribute("tagtype").Value == "CYCLE" &&`
	`(`
	`c.Element("VARIABLE") == null ||`
	`c.Element("VARIABLE").Attribute("name") == null ||`
	`c.Element("VARIABLE").Attribute("value") == null`
	`)`
	`select c;`
	
	`if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";`

Die Richtlinie fordert einerseits die Überprüfung im Wurzelknoten (`root`), ob das Attribut (`tagtype`) CYCLE existiert, welches nur vorhanden ist, wenn andererseits auch ein Wert dafür gesetzt wurde. Ist dies gesetzt, wird in der Abfrage die Überprüfung durchgeführt, ob die Eigenschaften (`name`) und deren Werte (`value`) angelegt sind. Wird eine nicht leere Ergebnismenge `Query` zurückgegeben, so sind ihre Elemente fehlerhaft – bzw. hierbei nicht wertbelegte Parameter.

Strukturierung der Top-Level Kompositionen

Innerhalb der Klassifikationsbäume werden verschiedene Äste modelliert, welche beispielsweise Signaleingänge und Signalausgänge definieren. Signaleingänge und -ausgänge sind von ihrer Semantik voneinander zu differenzieren. Oftmals trifft dies auch für bestimmte Signalgruppen zu, da sie semantisch unterschiedlich sind – z. B. verschiedene Steuergeräte betreffen: wie im APR-Beispiel zum *SGFH*, zum *SGB* oder zum *SGL* gehören oder bereits von der Art des Datentyps her verschieden sind. Differenzierungen durch Vermeidung doppelter Elemente in verschiedenen Ästen des Klassifikationsbaums sind daher typische Strukturprüfungen auf einem Klassifikationsbaum. In der Abbildung 80 ist ein solcher Fall dargestellt, wo sichergestellt werden soll, dass die Eingangssignale der Fahrzeugfunktion nicht auch in den Ausgängen auftreten, wie dies für Signaleingang `Eingang A` der Fall ist.

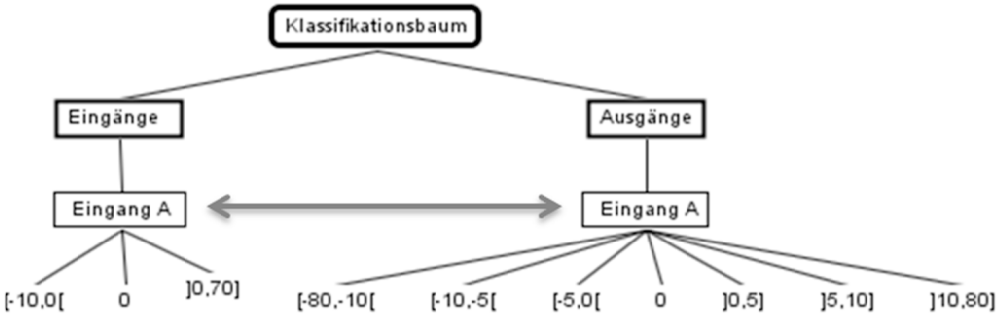


Abbildung 80: Vermeidung von Strukturfehlern (Wiederholung)

Eine weitere Konformitätsprüfung ist im Falle der vereinheitlichten Testdatengenerierung sinnvoll.

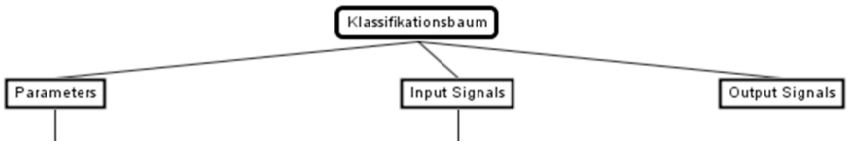
Richtlinie: Auf der obersten Ebene des Klassifikationsbaums sollen zumindest die drei wichtigsten Kompositionen wie Signaleingänge (*Input Signals*), Signalausgänge (*Output Signals*) und die Parameterwerte (*Parameters*) modelliert werden. Unter dem Knoten *Input Signals* stehen somit alle möglichen Eingangswerte und unter dem Knoten *Output Signals* die dazu erwarteten Ausgaben des Systems. Der Knoten *Parameters* schließlich, definiert die Setzungen der Konstanten des Testobjektes.

Die Tabelle 39 zeigt die exemplarisch entwickelte Richtlinie zur Strukturierung des Klassifikationsbaums auf oberster Ebene, welche die Konformität zu der aufgestellten Forderung sicherstellt.

Tabelle 39: Testspezifikation - Benutzbarkeit & Verständlichkeit

ID:	14	**Artefakt:**	Klassifikationsbaum, CTE XL

Prozessphase:	*Testspezifikation*
Titel:	***Strukturierung der Top-Level-Kompositionen***
Kategorie:	*Richtlinien zur Verbesserung der Benutzbarkeit und Verständlichkeit* — **Priorität:** *Empfohlen*
Vorbedingung:	$\exists$ root $\in$ *Klassifikationsbaum*
Richtlinie:	*Überprüfen, ob die obersten Kompositionen nach Root-Knoten Parameters, Input Signals und Output Signals existieren.*
Beschreibung:	*Allgemeine Strukturierung der Kompositionsknoten unterhalb des Root-Nodes für die vereinheitlichte Testdatengenerierung. Nachfolgende Abbildung zeigt das Beispiel eines Klassifikationsbaums in erster und zweiter Hierarchie.*



Konformität:

$$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\text{ID}o, \ldots, \text{ID}n+1\} \end{cases}$$

Ausführbare Regel in LINQ:

```
bool hasPara, hasInput, hasOutput; hasPara = hasInput = hasOutput = false;
List<XElement> Query = new List<XElement>();

//Select all compositions in the root node
var comps = from c in Artefact["cte"]
                      .Descendants("tree")
                      .Elements("tree")

    where c.Parent.Attribute("type").Value == "root"
    && c.Attribute("type").Value == "composition"
    group c by c.Parent into rootGroup
    select rootGroup;

// Check each element in the composition
foreach (var rootGroup in comps)
{ foreach (XElement x in rootGroup)
{    if (String.Compare(x.Attribute("name")
        .Value, "Parameters") == 0)
        hasPara = true;
    else
        if (String.Compare(x.Attribute("name")
            .Value, "Output Signals") == 0)
            hasOutput = true;
        else if (String.Compare(x.Attribute("name")
            .Value, "Input Signals") == 0)
            hasInput = true;
}
if (!hasInput || !hasOutput || !hasPara)
Query.Add(rootGroup.Key);
}
if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

Der hier entwickelte Regel-Algorithmus bildet zunächst eine Obermenge (`comps`) aller Kompositionsmengen (`rootGroup`) unterhalb des Wurzelknotens (`root`). In einer doppelten Schleife werden dann alle Kompositionsmengen unterhalb des Wurzelknotens iteriert und die jeweiligen Wurzelknoten `x` der Kompositionsmengen `rootGroup` mittels Namensprüfung

durch Attributvergleich `String.Compare(x.Attribute("name"))` auf Konformität bzw. Gleichheit hin untersucht. Auch hierbei ist die Regel so entwickelt, dass die Ergebnismenge `Query` leer sein muss, um ein positives Ergebnis zu erhalten.

Strukturierung der Parameter-Komposition

Richtlinie: Neben syntaktischen Prüfungen sind verschiedene semantische Beziehungen in der Testspezifikation zwischen den einzelnen Testfällen vom Ingenieur festzulegen. Sind beispielsweise im APR zwei Signaleingänge des Systems (Steuergerät) festgelegt wie Batteriespannung U_{batt} und die Zündung I_{gn} an Klemme 15, so ist es evident, dass ohne Batteriespannung auch die Zündung nicht angeschaltet sein kann. In der Abbildung 81 wäre eine solche Situation möglich. Der Signaleingang `bFFBAnfOeffnen` kann nur dann aktiv sein, wenn auch die Zündung I_{gn} an Klemme 15 geschaltet ist. In der Abbildung dargestellt ist ein Sonderfall der Voraussetzung, dass überhaupt ein Wert für beide Signale gesetzt wurde.

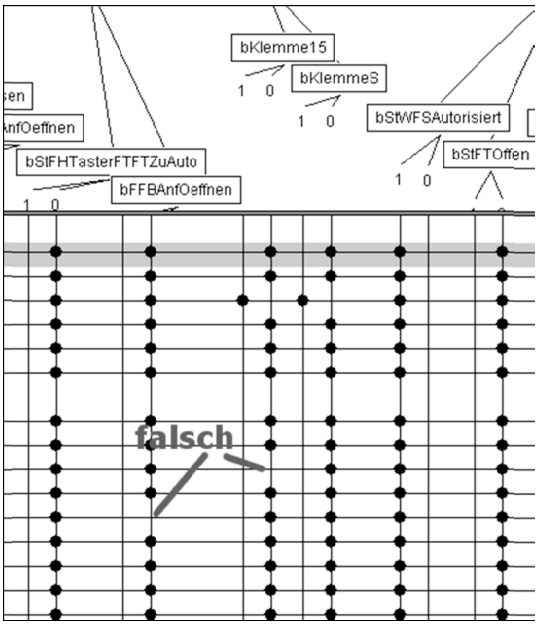


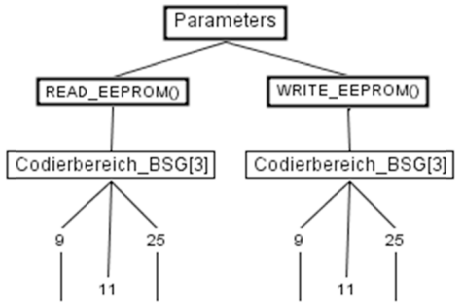
Abbildung 81: Semantische Beziehungen müssen konsistent sein

Eine weitere APR-Richtlinie stellt sicher, dass im Falle der Testdatengenerierung auf der Sub-Ebene der Komposition *Parameters* zumindest die zwei bedeutenden Kompositionen WRITE_EEPROM() und READ_EEPROM() stehen. Unter WRITE_EEPROM() werden die zu setzenden Parameter und unter READ_EEPROM() werden die zu lesenden Parameter spezifiziert.

Die Tabelle 40 zeigt die exemplarisch entwickelte Richtlinie zur Strukturierung des Klassifikationsbaums in den Ästen auf tieferen Ebenen, welche die Konformität zu der aufgestellten Forderung sicherstellt.

Tabelle 40: Testspezifikation - Verbesserung der Übertragbarkeit

ID:	15	**Artefakt:**	Klassifikationsbaum, CTE XL

Prozessphase:	*Testspezifikation*
Titel:	***Strukturierung der Parameter-Komposition***
Kategorie:	*Richtlinien zur Verbesserung der Übertragbarkeit* **Priorität:** *Empfohlen*
Vorbedingung:	∃ parameters ∈ *Komposition*
Richtlinie:	*Die oberste Ebene unter Parameter-Komposition ist so strukturiert, dass zumindest die zwei wichtigsten Sub-Kompositionen: WRITE_EEPROM() und READ_EEPROM() modelliert sind.*
Beschreibung:	*Allgemeine Strukturierung des Kompositionsknoten **Parameters** für die vereinheitlichte Testdatengenerierung mit WRITE_EEPROM() und READ_EEPROM(). Nachfolgende Abbildung zeigt das Beispiel des allgemeinen Aufbaus der Parameter-Komposition für die Testdatengenerierung.*  • *Es muss sichergestellt sein, dass die Kompositionen WRITE_EEPROM() und READ_EEPROM() existieren.* • *Es muss sichergestellt sein, dass die Parameternamen gültig sind. (Die Kompositionen müssen Elemente enthalten).*
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\mathrm{ID}o, \ldots, \mathrm{ID}n+1\} \end{cases}$

Ausführbare Regel in LINQ:	`List<XElement> Query = new List<XElement>();`
	`// Select composition "Parameters"`
	`var params = from c in Artefact["cte"].Descendants("tree")`
	`where c.Attribute("type").Value == "composition"`
	`&& c.Attribute("name").Value == "Parameters"`
	`select c;`
	
	`foreach (XElement x in params)`
	`{`
	`// Get all READ_EEPROM() sub-parameters`
	`var read = from c in x.Elements("tree")`
	`where c.Attribute("type").Value == "composition"`
	`&& c.Attribute("name").Value == "READ_EEPROM()"`
	`select c;`
	
	`//Get all WRITE_EEPROM() sub-parameters`
	`var write = from c in x.Elements("tree")`
	`where c.Attribute("type").Value == "composition"`

```
                && c.Attribute("name").Value == "WRITE_EEPROM()"
                select c;

    if (read.Count() == 0 || write.Count() == 0)
        Query.Add(x);

    if (read.Count() != write.Count())
        Query.Add(x);

}
if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

Der hier entwickelte Algorithmus bildet zunächst die Obermenge (`params`) aller Kompositionsmengen mit der Wertbelegung *Parameters* im Wurzelknoten (`tree`). In einer doppelten Abfrage wird dann nach den Sub-Kompositionsmengen a) WRITE_EEPROM() und b) READ_EEPROM() im Ast des Klassifikationsbaums gesucht. Alle Elemente der der Äste a) und b) werden in den Sub-Mengen `read` und `write` gehalten. Des Weiteren wird in zwei separaten Plausibilitätsprüfungen die Konformität sichergestellt. Einerseits wird gefordert, dass beide Sub-Mengen nicht leer sind und andererseits, dass sie in der Anzahl gleich sind. Eine Erweiterung (hier nicht gezeigt) ist die Verschärfung der Regel, in dem nun auch noch alle Sub-Elemente der beiden Mengen miteinander verglichen werden, um sicherzustellen, dass auch alle lesbaren Register auch den schreibbaren Registern entsprechen. Die Regel ist so entwickelt, dass die Ergebnismenge `Query` aufgebaut wird, welche die fehlerhaften Elemente beinhaltet.

Werkzeugübergreifender Abgleich von Testfällen mit Anforderungen

Im Falle des modellbasierten Systementwurfs sollen für die Testdatengenerierung auf der obersten Ebene alle notwendigen Eingangssignale im Klassifikationsbaum modelliert werden.

Signaltabelle					
Signal	**Alias**	**Init-Status**	**Exit-Status**	**E/A/AM**	**Bemerkungen**
Klemmen zur Stromversorgung:					
Batterie	Batt	U0	U0	E	Spannungsversorgung Bordnetz
Klemmen Eingang					
Klemme 15	KL_15	OL	OL	E	Zündschlosskontakt Klemme 15
Klemme S	KL_S	OL	OL	E	Zündschlosskontakt Klemme S
Klemme X	KL_X	OL	OL	E	Zündschlosskontakt Klemme X
Klemme 50	KL_50	OL	OL	E	Zündschlosskontakt Klemme 50
Fensterheber POTI	SGFH_Poti	OL	OL	E	Interenes Poti im Crashfall Fensterheber. 2 Stufen, woraus sich von der Geschwindigkeit abhaengige Stufen ergeben.
Fensterheber END	SGFH_END	OL	OL	E	Endlagenkontakt Fensterheber
Fensterheber Stufe 1	SGFH_St_1	OL	OL	E	Crashfall Stufe 1 Fensterheber
Fensterheber Stufe 2	SGFH_St_2	OL	OL	E	Crashfall Stufe 2 Fensterheber
Fensterheber Airbag	SGFH_Airbag	OL	OL	E	Crashfall Ansteuerung Airbag
Klemme SRA aktiv	KL_SRA	OL	OL	E	Freigabe bei Crash
Motorhaube		OL	OL	E	Motorhaubenkontakt
Klemmen Ausgang					
Fensterheber Ansteuerung Stufe 1	SGFH_An_St_1	OL	OL	A	Ansteuerung Fensterhebermotor in Stufe 1
Fensterheber Ansteuerung Stufe 2	SGFH_An_St_2	OL	OL	A	Ansteuerung Fensterhebermotor in Stufe 2
SGLen Ansteuerung SRA	SGL_SRA	OL	OL	A	Ansteuerung Scheinwerferreinigungsanlage
CAN-Signale Eingang:					
Geschwindigkeit Fahrzeug	v_Fzg	0i	0i	E	
CAN-Signale Ausgang:					
Fensterheber blockiert	SGFH_block	0o	0o	A	Fensterheber Blockiererkennung
Fensterheber Stufe 1 Staus	SGFH_ST_1_Status	0o	0o	A	Statusmeldung der Fensterheberstufe 1
Fensterheber Stufe 2 Staus	SGFH_ST_2_Status	0o	0o	A	Statusmeldung der Fensterheberstufe 2
Fensterheber Geschwindigkeit norm	SGFH_normal	0o	0o	A	Statusmeldung der Fensterhebergeschwindigkeit = normal
Fensterheber Geschwindigkeit schn	SGFH_schnell	0o	0o	A	Statusmeldung der Fensterhebergeschwindigkeit = schnell
Crash Status	SGL_Status	0o	0o	A	Statusmeldung

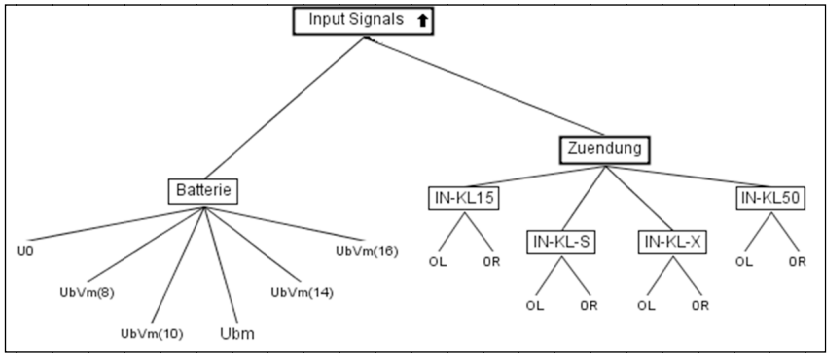
Abbildung 82: Werkzeugübergreifende Prüfung aus der Signal-Tabelle

Die Eingangssignale können durch Kompositionen auch zusammengefasst sein. Die Klassifikationen stellen dann die Signale dar. Hierbei ist eine Überprüfung wichtig, dass die Klassifikationsnamen mit den Signalnamen bzw. mit den Aliasnamen aus der Anforderung (z. B. aus dem Lastenheft) übereinstimmen. Diese Informationen sind einem weiteren Artefakt (der *Signalliste*) zu entnehmen (Abbildung 82). In unserem APR-Beispiel ist dies eine MS Excel-Tabelle. Die Namen der Eingangssignale sind mit den Spalten Signal bzw. Alias zu vergleichen.

Die Tabelle 41 zeigt die Regelimplementierung einer artefaktübergreifenden Richtlinie zur Prüfung des CTE/XL-Klassifikationsbaums auf die Konformität gegen die Signalliste in der MS Excel-Tabelle aus der Anforderungsdokumentation (Spezifikation).

Tabelle 41: Testspezifikation - Verbesserung der Benutzbarkeit & Verständlichkeit

ID: 16	**Artefakt:** Klassifikationsbaum, CTE XL, Excel Tabelle

Prozessphase:	*Testspezifikation*
Titel:	***Konformität der Signalspezifikation***
Kategorie:	*Richtlinien zur Verbesserung der Benutzbarkeit und Verständlichkeit* **Priorität:** *Empfohlen*
Vorbedingung:	∃ root ∈ *Klassifikationsbaum*
Richtlinie:	
Beschreibung:	*Die Bezeichner der Signalnamen unterhalb des Root-Nodes im Klassifikationsbaum müssen für die Testdatengenerierung identisch mit den Vorgaben aus der Signaltabelle sein. Nachfolgende Abbildung zeigt das Beispiel eines Klassifikationsbaums in erster und zweiter Hierarchie.*  *Überprüfen der Klassifikation mit der Signaltabelle:* *Des Weiteren dürfen Signale nur als Eingangssignale (Input Signals) verwendet werden, wenn in der Spalte E/A/AM ein „E“ ausgewählt ist. Diese Informationen sind aus der Signaltabelle zu entnehmen.*
Konformität:	$\Sigma \begin{cases} pass = \emptyset, \\ fail = \{\mathrm{IDo}, \ldots, \mathrm{ID}n+1\} \end{cases}$

Ausführbare Regel in LINQ:

```
//Select the composition with the name "Input Signals"
var Query1 = from c in Artefact["cte"].Descendants("tree")
             where c.Attribute("type").Value == "composition" &&
                   c.Attribute("name").Value == "Input Signals"
             select c;

//Select all the classifications in "Input Singals"
var Query2 = from cc in Query1.First().Descendants("tree")
             where cc.Attribute("type").Value == "classification"
             select cc;

//Select all the singal elements in Signal-Table
var Query3 = from c in Artefact["excel"].Descendants()
             where c.Name.LocalName == "Row"
             //filter the commentar row
             && (!c.Elements().ElementAt(0).Value.StartsWith("#"))
             //filter the title row
             && c.Elements().ElementAt(0).Value != "Singal"
             select c;

List<XElement> Query = new List<XElement>();
List<string> Eingabe = new List<string>();
//get the name and alias of those whose values
//in E/A/AM column are E
foreach (XElement row in Query3)
{
    if (row.Elements().ElementAt(5).Value == "E")
    {
        Eingabe.Add(row.Elements().ElementAt(0).Value);
        Eingabe.Add(row.Elements().ElementAt(1).Value);
    }
}

//put the classification, whose name exists neither
//in Name nor in Alias, in the result
foreach (XElement classi in Query2)
{
    if (!Eingabe.Contains(classi.Attribute("name").Value))
        Query.Add(classi);
}

if(Query.Count() >0) Result = "FAIL"; else Result = "PASS";
```

Die umfangreiche Regel-Implementierung ist zu Beginn in drei wesentliche Abfragen unterteilt. Zunächst wird eine Untermenge `Query1` der Eingangssignale (eine Komposition) des Klassifikationsbaums `Artefact["cte"].Descendants("tree")` gebildet. Danach folgt die Bildung der Menge `Query2` der konkreten Klassifikationen für Eingangssignale im Baum. Schließlich wird das zweite prozesslogisch verbundene Artefakt `Artefact["excel"].Descendants()` mit der Abfragemenge `Query3` eingebunden. Zwei Filter-Bedingungen beschränken das Artefakt auf die wesentlichen Zeilen innerhalb der Signaltabelle. Nur die Signalspezifikationen bleiben übrig. Unnötige Kommentare oder Überschriften werden gefiltert. Gemäß der zweiten Richtlinien-Bedingung, dass in der Spalte E/A/AM ein „E“ ausgewählt ist, wird eine neue Menge `Eingabe` aufgebaut, die nur noch die Signale mit Namen und Alias (Feldpositionen der Spalte 0 und Spalte 1) enthalten. Nun ist es soweit: die korrespondierenden Elemente aus beiden Artefakten können miteinander auf Konformität (Gleichheit) untersucht werden. Hierzu wird mittels *Contains*-Beziehung `Eingabe.Contains(classi.Attribute("name").Value))` geprüft, ob alle Elemente der Menge `Eingabe` mit den Namen aus der Signaltabelle (`Query2`) übereinstimmen. Die Negation macht es möglich: Bei nicht erfolgreichem Auffinden des Namens im Klassifikationsbaum wird das fehlerhaft spezifizierte Element in die Ergebnismenge `Query` für eine spätere Modellanalyse aufgenommen.

Im modellbasierten Entwicklungsprozess eingebetteter Systeme stellen industrielle Normen und daraus resultierende Prozessrichtlinien hohe Anforderungen an die Erstellung von elektronischen Arbeitserzeugnissen (Artefakte). Heutzutage erfordert der ingenieurmäßige Entwurf eines eingebetteten Systems die nachgewiesene Konformität (Erfüllung der Anforderungen) zu prozessspezifischen Entwicklungsrichtlinien, gerade in kollaborativ durchgeführten Arbeitsschritten. Richtlinien legen prozessübergreifend fest, wie Spezifikationen einheitlich formuliert, Architekturen konform modelliert, Dokumente konsistent bearbeitet, Simulationen fehlerfrei entworfen, eingebettete Software generiert oder Daten in spezieller Formatierung abgelegt werden. Die nachweisliche Erfüllung der Anforderungen aus vielfältigen Entwicklungsrichtlinien erweist sich in einer heterogenen IT-Landschaft als überaus zeitaufwändig und fehleranfällig. In der Arbeit wird ein automatisierbarer Ansatz zur regelbasierten Konformitätsprüfung kollaborativer Artefakte von interdisziplinär geltenden Entwicklungsrichtlinien im Automotive-Kontext vorgestellt, eine Methodik sowie die werkzeugtechnische Umsetzung beschrieben.

ISBN 978-3-8396-0266-9