Ein modellbasiertes, graphisch notiertes, integriertes Verfahren zur Bewertung und zum Vergleich von Elektrik/Elektronik-Architekturen

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

an der Fakultät für Elektrotechnik und Informationstechnik

am Karlsruher Institut für Technologie (KIT)

genehmigte

DISSERTATION

von

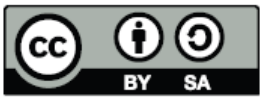
Dipl.-Ing. Daniel Josef Gebauer

geb. in: Krappitz

Tag der mündlichen Prüfung: 22.7.2016

Hauptreferent: Prof. Dr.-Ing. Klaus-D. Müller-Glaser

Korreferent: Prof. Dr. Albert Zündorf



Danksagung

Ich möchte mich ganz besonders bedanken bei

Herrn Prof. Dr.-Ing. Klaus D. Müller-Glaser für die wissenschaftliche Betreuung dieser Arbeit, für seine Ratschläge und nicht zuletzt für seine Geduld. Seine Erfahrung und sein überragendes und disziplinübergreifendes Fachwissen haben ganz wesentlich zum Gelingen dieser Arbeit beigetragen.

Herrn Prof. Dr. Albert Zündorf für die Übernahme des Korreferats dieser Arbeit.

allen Kolleginnen und Kollegen bei der aquintos Gmbh und später Vector Informatik Gmbh. Insbesondere möchte ich mich bei Herrn Dr.-Ing. Clemens Reichmann und Dr.-Ing. Johannes Matheis die reibungslose Zusammenarbeit und Hilfsbereitschaft sowie für zahllose wissenschaftliche Diskussionen im weiten Umfeld der E/E-Architekturen bedanken.

meiner Frau Manuela (sie ist auch eine ganz hervorragende Korrekturleserin) und den Kindern Nils und Simon für das Verständnis und die zeitlichen Freiräume, die sie mir an zahlreichen Wochenenden schufen.

den zahlreich beteiligten studentischen Mitarbeitern, vor Allem Holger Bremer, Bertram Fesseler, Bernhard Hommel, Michael Kohlbecker, Prudence Kouam, Johannes Quast und Thibault Schneider, (in alphabetischer Reihenfolge) die im Rahmen von Abschlussarbeiten wichtige Ergebnisse für die praktische Umsetzung und Evaluierung erzielten.

meine Eltern Regina und Peter Gebauer, die stets eine Stütze waren und auch die richtigen motivierenden Worte fanden. Insbesondere möchte ich mich bei meinem Vater bedanken, der mir bereits in jugendlichen Jahren die Gelegenheit gab, selbst an Fahrzeugen zu schrauben, Leitungssätze unter seiner fachmännischer Aufsicht zu verlegen und die Parametrisierung von Steuergeräten anzupassen.

und Allen anderen, mit denen ich in den letzten Jahren im Rahmen meiner Arbeit Kontakt hatte, insbesondere Herrn Dr.-Ing. Julian Broy für die gute Kooperation bei der Parametrisierung von FlexRay-Bussystemen.

Kurzfassung

Die digitale Revolution hat große Auswirkung auf alle Lebensbereiche. Dies zeigt sich sehr eindrucksvoll an der Geschichte der Automobilentwicklung in den letzten Jahrzehnten. Angefangen mit der elektrischen Beleuchtung in der Frühzeit hielten elektrische und elektronische Systeme Einzug in den Bereichen Sicherheit, Komfort und Ökonomie. In den letzten Jahren ist eine zunehmende Vernetzung der zuvor isoliert wirkenden Systeme zu beobachten. Damit ergeben sich Synergieeffekte und neue Funktionalitäten. In Zukunft wird es, neben immer effizienteren und umweltschonenden Antriebssystemen, teil- oder vollautomatisierte Fahrzeuge geben.

Diese Entwicklungen haben, zusätzlich getrieben durch den großen Konkurrenz- und Kostendruck, massive Auswirkung auf die Entwicklung und den Entwicklungsprozess neuer Elektrik/Elektronik-Architekturen (E/E-Architekturen). Es werden ganze Fahrzeugfamilien, basierend auf Plattformen und Modulen, parallel entwickelt. Der Einfluss von architektonischen Entscheidungen und -änderungen muss schnell und umfassend bewertet, analysiert und mit Referenzsystemen verglichen werden.

Die vorliegende Arbeit befasst sich mit einem computergestützten, modellbasierten Verfahren zur Bewertung, Analyse und Vergleich von E/E-Architekturen im Fahrzeug. Es wird eine domänenspezifische Sprache (DSL) zur modellbasierten und effizienten Beschreibung von Bewertungen, Analysen und Vergleichen (Metriken) vorgestellt. Es wird unterschieden zwischen Blöcken und Datenflüssen. Blöcke repräsentieren unterschiedliche Aktivitäten, z.B. für Datenaquise, Berechnung und Ergebnisaufbereitung. Es stehen eine Vielzahl von unterschiedlichen Blöcken zur Bewältigung der anfallenden Aufgaben zur Verfügung. Die Blöcke werden über Ports durch Datenflüsse miteinander verbunden. Ports sind Schnittstellen eines Blocks als Datenausgang bzw. -eingang. Mit diesen Beschreibungsmitteln werden komplexe Berechnungsabläufe strukturiert und damit verständlich und nachvollziehbar. Über eine graphische Notation, die an Blockschaltdiagrammen orientiert ist, lassen sich diese Berechnungsvorschriften visualisieren und über einen entsprechenden Editor graphisch modellieren. Das in dieser Arbeit vorgestellte Framework zur Bewertung, Analyse und Vergleich von E/E-Architekturen im Fahrzeug ist integriert in die Beschreibungssprache für E/E-Architekturen EEA und verfügt über vollen lesenden und schreibenden Datenzugriff. Zusammen mit den Ergebnisblöcken, tabellarischen Ergebnisansichten und der Debugfähigkeit der Berechnungsvorschriften wird eine hohe Transparenz der Abläufe und Ergebnisse erreicht.

Über spezifische Blöcke werden Varianten und Realisierungsalternativen direkt hinsichtlich vorgegebener Bewertungskriterien miteinander vergleichbar. Damit wird dem Plattform- und Modularisierungsgedanken bei Automobilherstellern Rechnung getragen. Zusätzlich unterstützt das Framework Sensitivitätsanalysen zur Untersuchung von Was-Wäre-Wenn-Szenarien.

Abstract

The digital revolution has a big influence on every part of our daily life. We can observe this through the history of automobile development over the last few decades.

Electrical and Electronical systems started at electric lighting and found their way into the areas of safety, comfort and economy. We can observe an increasing cross linking of systems that previously seemed to be isolated prior to the last few years. The effects of synergy and new functionalities are therefore increasing. In the near future there will be as well as more efficient and environmentally driven systems, particularly or fully automated automobiles.

This evolution has a huge impact on the development and the development process of new electrical and Electronical Architecture (E/E architectures), additionally driven by the huge pressure of competition and costs. Whole automobile families are developed in parallel based on platforms and modules. The influence of architectural decisions and changes has to be evaluated quickly and completely also analysed and compared to reference systems.

This study deals with a computational model based method for evaluation, analysis and comparison of E/E architecture in automobiles. There will be an introduction of a domain specific language (DSL) for model based and efficient characterisation of evaluations, analyses and comparisons (Metrics). We distinguish between blocks and data flows. Blocks represent different activities, e.g. for data acquisition, the calculation and presentation of results. Many different blocks are available to handle the occurring tasks. The blocks are combined via ports through data flows. Ports are interfaces of a block as data output or input. Complex calculation processes are structured by this DSL and are thereby conceptional and comprehensible. These metrics are made visible by a graphic notation which is geared to a block diagram and are geographically modelled by a corresponding editor. The framework for evaluation, analysis and comparison of EE architectures in automobiles presented in this study is integrated in the description language for EE architecture EEA and has complete reading and writing data access. A high transparency of processes and results is reached, together with the result blocks, tabular result views and the debug capability of the evaluation rules.

Variants and realisation alternatives are directly comparable throughout specific blocks according to given evaluation criteria. So the thought of platform and modularization coming from the automobile producer is accommodated. Additionally the framework supports sensitivity analyses for investigations of what-if scenarios.

Inhaltsverzeichnis

1 Einleitung

1.1 Einführung

In den letzten Jahrzehnten sind im Bereich der Elektrik/Elektronik enorme Innovationssprünge zu verzeichnen. Diese Entwicklung startete, wie in Abbildung 1.1 dargestellt, in den 1970/80er Jahren mit der Einführung von Motorsteuergeräten und Tempomat (Cruise Control).

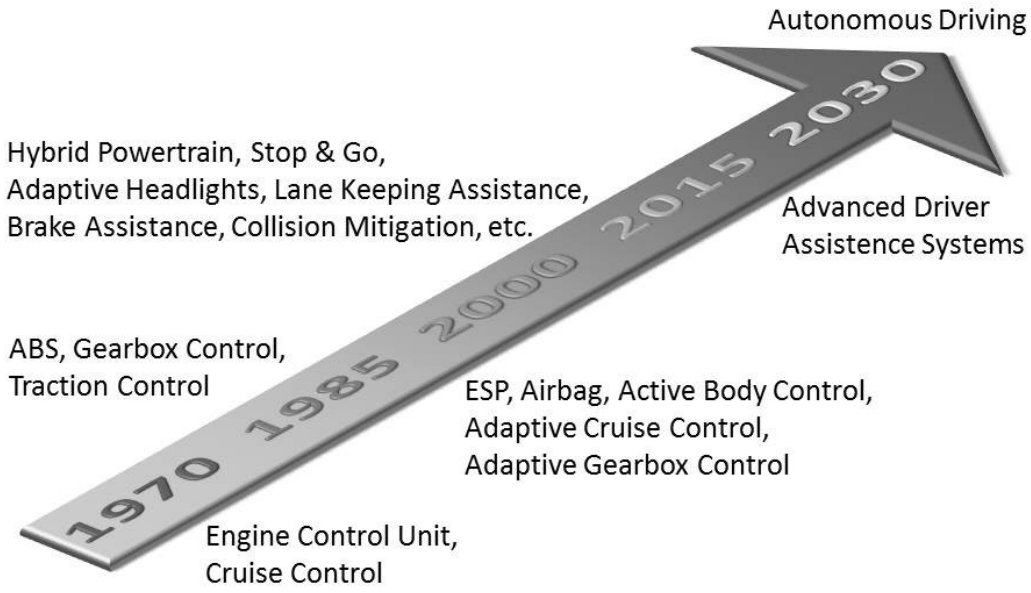


Abbildung 1.1: E/E-relevante Innovationen in Anlehnung an [SCH06]

Es folgten das Anti-Blockier-System ABS, die Getriebesteuerungen (Gearbox Control) und die Traktionskontrolle. Ab den 1990er Jahren konzentrierte sich die Entwicklung auf das Elektronische Stabilitätsprogramm ESP, Airbags sowie auf die Weiterentwicklung bestehender Systeme (Adaptive Cruise Control, Adaptive Gearbox Control). Um die Jahrtausendwende kamen die ersten Assistenzsysteme auf den Markt, welche kontinuierlich weiterentwickelt wurden.

Neben dem Ausbau der Assistenzsysteme, welcher im Prinzip durch neue Sensorik/Aktuatorik und den Ausbau der Rechenkapazität zur Datenauswertung getragen wird, geht in den letzten Jahren der Trend zur Vernetzung und zur Integration der vorhandenen Systeme.

Diese Entwicklungstendenzen lassen sich, wie in Abbildung 1.2 dargestellt, nach [KOH12] in drei Phasen einteilen.

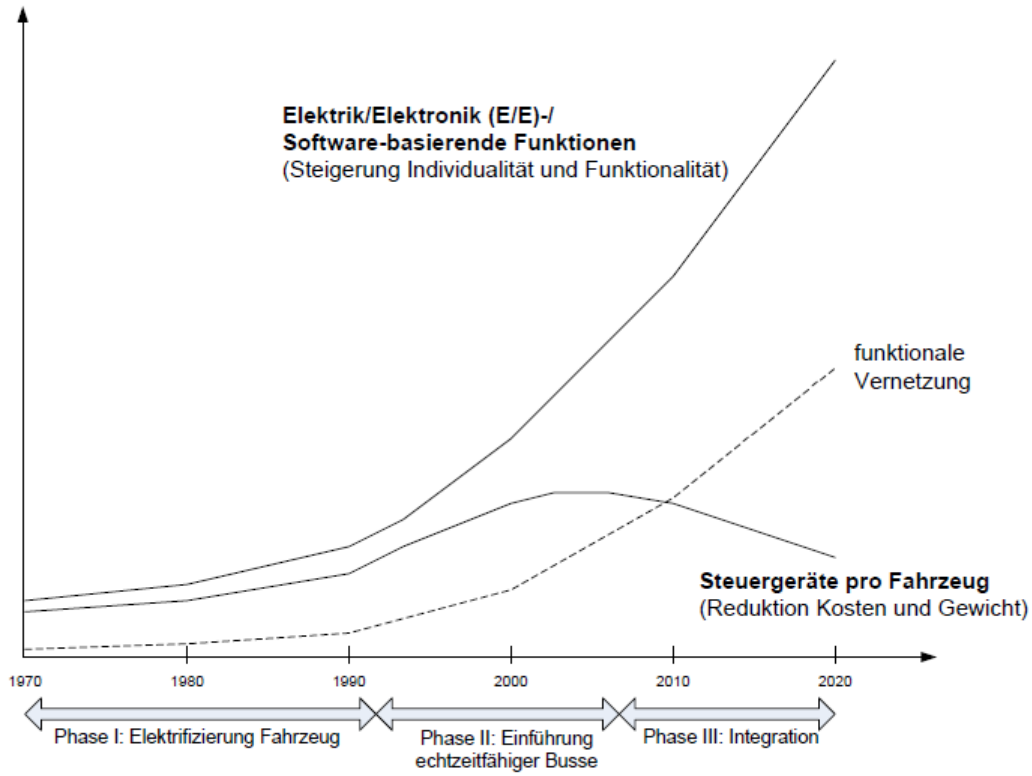


Abbildung 1.2: Phasenmodell der Elektrifizierung von Fahrzeugen nach [KOH12]

Während Phase 1 wurden Basisfunktionalitäten für mehr Komfort, Sicherheit und Ökonomie durch elektrische und elektronische Systeme zur Verfügung gestellt. Phase 2 war geprägt von der Vernetzung der Systeme durch leistungsfähige Bussysteme, die während dieser Zeit eingeführt wurden. Die Phase 3 umfasst die Integration der vorhandenen und neuer Systeme, um daraus Mehrwerte zu erzielen (funktionale Vernetzung).

Während vor allem in Phase 1 und anfangs in Phase 2 neue Funktionen hauptsächlich mit Hilfe zusätzlicher Steuergeräte realisiert wurden, finden in den Phasen 2 und 3 eine Konsolidierung und eine Reduktion der Anzahl statt.

Die Vernetzung und Integration der vorhandenen und künftigen Systeme wird in Zukunft autonom fahrende Fahrzeuge ermöglichen. Der Weg dorthin lässt sich in folgende fünf Stufen unterteilen [BAS12]:

- Driver Only:
  „Fahrer führt dauerhaft (während der gesamten Fahrt) die Längsführung (Beschleunigen / Verzögern) und die Querführung (Lenken) aus.“ [BAS12]
- Assistiert:
  „Fahrer führt dauerhaft entweder die Quer- oder die Längsführung aus. Die jeweils anderen

Fahraufgaben wird in gewissen Grenzen vom System ausgeführt.
- Der Fahrer muss das System dauerhaft überwachen...“ [BAS12]

- Teilautomatisiert:
  „Das System übernimmt die Quer- und Längsführung (für einen gewissen Zeitraum und/oder in spezifischen Situationen).
  - Der Fahrer muss das System dauerhaft überwachen...“ [BAS12]

- Hochautomatisiert:
  „Das System übernimmt die Quer- und Längsführung für einen gewissen Zeitraum in spezifischen Situationen.
  - Der Fahrer muss das System dabei nicht dauerhaft überwachen...“ [BAS12]

- Vollautomatisiert (Autonom):
  „Das System übernimmt Quer- und Längsführung vollständig in einem definierten Anwendungsfall.
  - Der Fahrer muss das System dabei nicht überwachen...“ [BAS12]

Diese Trends zum automatisierten und autonomen Fahren werden von der Automobilbranche vorangetrieben ([AUD14],[DAI14],[HER14]), regelmäßig werden die neusten Ergebnisse vorgestellt (z.B. [BMW14],[DAI13],[DFT25],[GOO14]). Als Beispiel sei hier die Fahrt eines Mercedes Benz S500 Prototypen im August 2013 genannt. Dieses Fahrzeug fuhr ca. 100 km im Überland- und Stadtverkehr von Mannheim nach Pforzheim autonom [DAI13].

Diese Entwicklungen haben große Auswirkungen auf die komplette Elektrik/Elektronik-Architektur (Def. 22, E/E-Architektur) eines Fahrzeugs. Hinzu kommen weitere, parallele Aktivitäten, beispielsweise im Bereich der Elektrifizierung der Antriebstechnik. Ca. 90% aller Innovationen haben einen Einfluss auf die E/E-Architektur [HAU04]. Die Entwicklung dieser Systeme mit der geforderten Qualität, Sicherheit, Ökonomie, Handhabbarkeit und Wartbarkeit unter Berücksichtigung des Kosten- und Konkurrenzdrucks stellt eine große Herausforderung dar.

1.2 Motivation

Die E/E-Architektur moderner Fahrzeuge zeichnet sich durch eine hohe Komplexität und Variabilität zur Unterstützung von Fahrzeugplattformen und -varianten aus. Abbildung 1.3 zeigt die wesentlichen Komponenten und deren Vernetzung am Beispiel einer Mercedes Benz S-Klasse aus dem Jahr 2013.

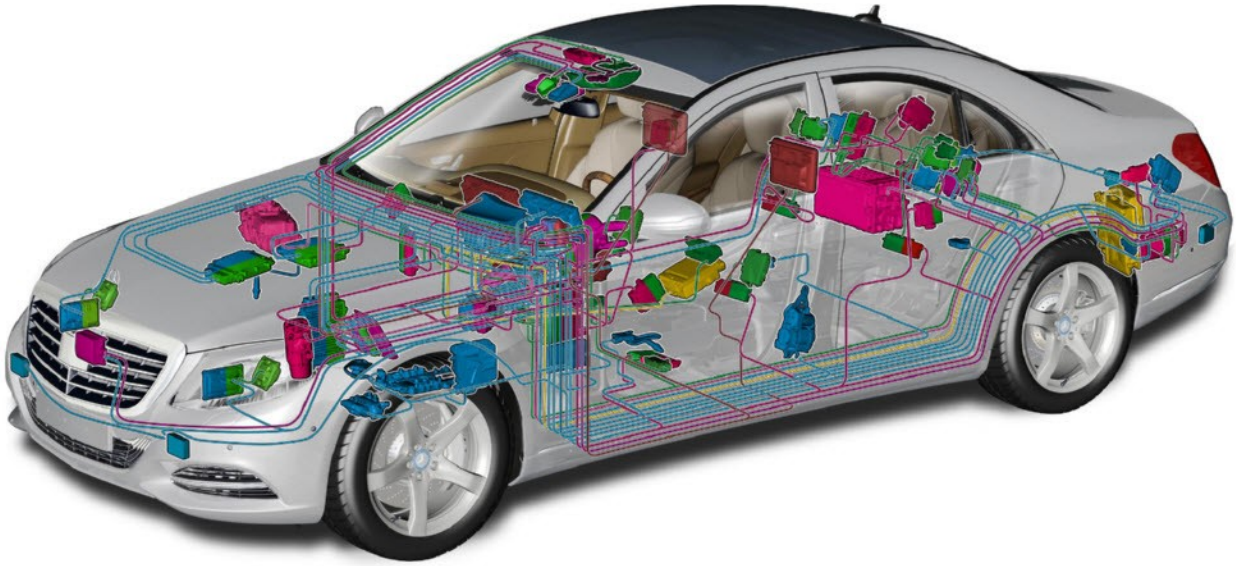


Abbildung 1.3: E/E-Architektur einer Mercedes Benz S-Klasse 2013 [DAIS13]

Die E/E-Architektur-relevanten Eckdaten[1] sind:

Bis zu 144 Steuergeräte	10 FlexRay-Steuergeräte
	73 CAN-Steuergeräte
	61 LIN Steuergeräte
	1 Ethernet-Steuergerät (Diagnose und Hauptuntersuchung)
Bis zu 30 Bussysteme	20 LIN
	7 CAN
	je 1 FlexRay, MOST, Ethernet
Leitungssatz	bis zu 734 Module
	max. 2385 Leitungen (4293 m, 45,8 kg)
	Einsatz von Aluminium für >10 mm²

Tabelle 1: E/E-Architektur-relevante Eckdaten [DAIS13]

Allein die in Tabelle 1 genannten Hardware-seitigen Zahlen dieses Beispiels zeigen die Komplexität einer E/E-Architektur.

Zusätzlich sind die Software-seitigen Merkmale einer E/E-Architektur nicht zu vernachlässigen. Die Software einer modernen E/E-Architektur realisiert über 50.000 funktionale Anforderungen mit über 15 Millionen Zeilen Quellcode (>10 MB Kontrollsoftware) und >5.000 Software Parametern [DAV13]. Es ergeben sich über 10.000 baubare Serienfahrzeug-Varianten (bezogen auf Komponen-

1 Diese Daten und das zugehörige Schaubild [DAIS13] wurden freundlicherweise von der Daimler AG zur Verfügung gestellt.

ten-basierte Permutationen einer Produktlinie) [DAV13].

Daraus ergeben sich unzählige Einspar- und Optimierungspotentiale, die es zu aufzudecken und auszunutzen gilt. Im Folgenden werden einige ausgewählte, relevante Fragestellungen aufgezählt:

- Anzahl bestimmter Elemente (z.B. Komponenten, Bussysteme und Anbindungsbausteine, Leitungen, Stecker, Splices, Trennstellen, etc.) für ausgewählte Varianten. Welche Gewichte, Kosten ergeben sich daraus? Können in bestimmten Varianten Komponenten oder andere Elemente der Architektur entfallen (Wenn ja, welche?)?
- Bewertung der Verlegewege von Leitungen für ausgewählte Varianten. Können durch unterschiedliche Leitungsführungen Einsparungen (kürzere Leitungen, weniger oder weniger große Steckverbinder) erzielt werden? Wie hoch sind diese Einsparungen?
- Befüllungsgrad von Segmenten für ausgewählte Varianten. Wie viel Platz benötigen die verlegten Leitungen innerhalb der Karosserie? Reicht dieser Platz aus?
- Softwarepartitionierung auf Steuergeräte für ausgewählte Varianten. Welcher Ressourcenbedarf ergibt sich durch eine bestimmte Partitionierung für einzelne Komponenten? Welcher Kommunikationsaufwand entsteht dadurch zwischen den Komponenten? Ergeben sich durch Änderungen der Partitionierung Einsparpotentiale (z.B. Wegfall von Komponenten für bestimmte Varianten)?
- Buslasten für ausgewählte Varianten. Welche Lasten entstehen auf welchen Bussystemen? Welche Auswirkungen haben Änderungen der Softwarepartitionierungen oder Änderungen des Signalroutings oder der Bustechnologie (Einsatz schnellerer/modernerer Bussysteme)?
- Digitalisierung von bisher analoger Sensorik/Aktuatorik. Welche Änderungen ergeben sich, wenn bisher analog und konventionell angeschlossene Sensoren und Aktuatoren digitalisiert und an (evtl. neue) Bussysteme angeschlossen werden (in Bezug auf Verlegewege der Leitungen, Kosten und Gewichte, Buslasten, etc.)?

Diese Aufzählung ließe sich immer weiter fortsetzen. Des weiteren sind neben den beschriebenen Bewertungen ebenfalls die Varianten-übergreifenden Vergleiche relevant. Zudem haben die unterschiedlichen Hersteller und Zulieferer eigene Vorstellungen über Ziele, Prioritäten und Vorgehensweisen.

1.3 Abstraktionsebenen einer E/E-Architektur

Um die in Abschnitt 1.2 beschriebenen Fragestellungen beantworten zu können, müssen die Grundlagen zur Beschreibung von E/E-Architekturen gelegt sein. Hierzu kommt die formale Beschreibungssprache EEA (siehe Kapitel 3) zur Modellierung von E/E-Architekturen zum Einsatz. Die

Strukturierung der EEA orientiert sich an den Vorschlag des VDA [PRO01], indem einzelne Ebenen abstrahiert werden [BFM05].

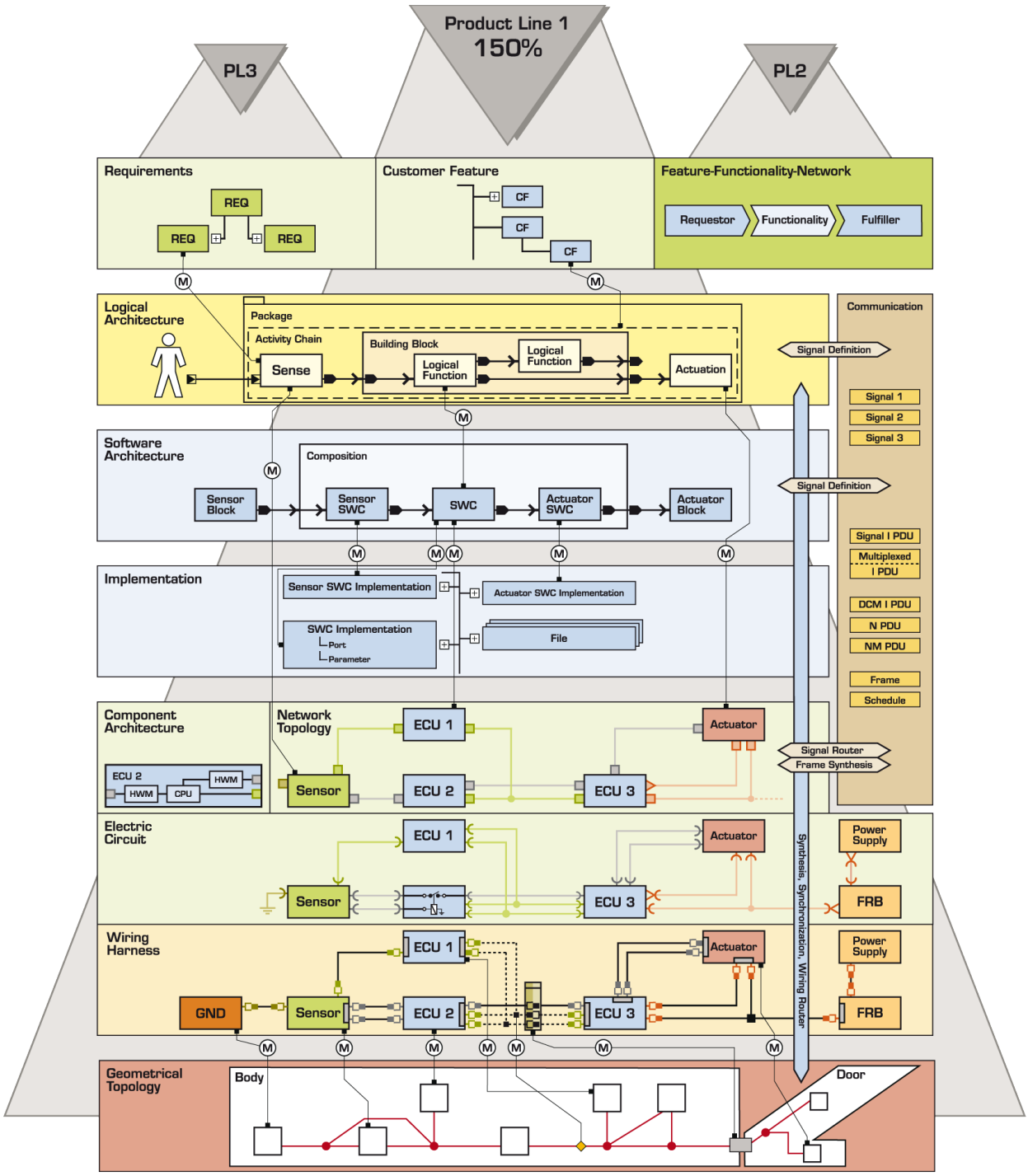


Abbildung 1.4: Ebenen einer E/E-Architektur [PREE]

Abbildung 1.4 zeigt die Abstraktionsebenen der EEA-Beschreibungssprache. Die abstrakteste Ebene stellen die an die E/E-Architektur gestellten Anforderungen (vgl. Abschnitt 3.3.1) dar. Diese können aufeinander referenzieren und Bezug zu anderen Artefakten der Architektur nehmen. In der

logischen Architektur (vgl. Abschnitt 3.3.2) bietet eine einheitliche, abstrahierte Sicht der Hardware- und Software-Architektur (vgl. Abschnitt 3.3.3), der nächst darunter liegenden Ebene. Dort werden die in Software umgesetzten funktionalen Zusammenhänge modelliert.

Die Hardware-Architektur (vgl. Abschnitt 3.3.5) wird drei Subebenen unterteilt, in denen die logische, die elektrologische und die Leitungssatz-orientierten Sichten der Hardware-Architektur behandelt werden. Parallel zur logischen-, Software- und Hardware-Architektur ist das Kommunikationsmodell (vgl. Abschnitt 3.3.4) angesiedelt. Dort werden die zur Kommunikation notwendigen Signale und deren Verfeinerung in die Protokollschichten der Übertragungsmedien beschrieben.

Die geometrische Topologie (vgl. Abschnitt 3.3.6) spezifiziert das mechanische Modell des Fahrzeugs mit Einbauräumen für Komponenten und Segmenten für verbindende Leitungen. Des weiteren werden physikalische Umwelteinflüsse, die an diversen Stellen zu erwarten sind, modelliert.

Ebenenübergreifende Zusammenhänge [MAT09] werden transparent durch (vgl. Abschnitt 3.3.7) sogenannte Mappings beschrieben. Die grauen Dreiecke im Hintergrund des Ebenenmodells visualisieren die Varianten- und Plattformaspekte einer E/E-Architektur. Dies ist insofern wichtig, dass es nicht nur darauf ankommt, eine einzelne Architektur zu bewerten, sondern Alternativen miteinander zu vergleichen und Auswirkungen von Änderungen auf gesamte Plattformen und Varianten im Auge zu behalten.

Mit diesem Beschreibungsmodell ist die Grundlage für eine formale Beschreibung von E/E-Architekturdaten gegeben. Mit dem CASE-Werkzeug PREEvision [PREE] (siehe Abschnitt 2.9) steht eine Software zur Verfügung, welche dieses Beschreibungsmodell implementiert. Sämtliche Aspekte des in dieser Arbeit vorgestellten Bewertungs-, Analyse- und Vergleichsframework, dessen genauen Ziele im folgenden Abschnitt detailliert beschrieben sind, beziehen sich auf dieses Beschreibungsmodell.

1.4 Ziele und eigener Beitrag

Die Beherrschung der Komplexität bei stark steigendem Funktionsumfang unter Berücksichtigung der geforderten Qualität und enormen Kostendruck ist die zentrale Herausforderung bei der Entwicklung neuer Kraftfahrzeuge. Hierfür ist es für die Ingenieure wichtig, wesentliche Kennzahlen im Blick zu haben, Analysen durchzuführen und zu vergleichen, um daraus wichtige Schlüsse für die weitere Entwicklung zu ziehen. Durch die Vielzahl von zu beachtender Varianten und zu unter-

stützenden Plattformen wird diese Arbeit zusätzlich erschwert.

Mit der vorliegenden Arbeit wird ein Verfahren zur Bewertung, Analyse und zum Vergleich von Elektrik/Elektronik-Architekturen, die als EEA-konformes Modell vorliegen vorgestellt. Damit sollen Fragestellungen, wie sie in Abschnitt 1.2 skizziert sind und in Kapitel 4.2 detaillierter beschrieben sind, effizient beantwortet werden können.

Im Einzelnen ist die Arbeit gekennzeichnet durch:

- Modellbasierte Beschreibung von Bewertungs-, Analyse- und Vergleichsvorschriften:
  Durch die modellbasierte Beschreibung, gestützt auf einem formalen Metamodell (siehe Abschnitte 2.1, 3.2.1 und Kapitel 6) wird eine Domänenspezifische Sprache (siehe Abschnitt 2.2.3) zur Formulierung von Bewertungs-, Analyse- und Vergleichsvorschriften geschaffen. Damit wird erreicht, dass Konzepte und Notation domänenspezifischen Anwendern bekannt sind. Die Modellierung von Lösungen wird durch Fokussierung auf den eingeschränkten Problembereich vereinfacht [BIS09].

- Graphische Notation:
  Die Definition einer graphischen Notation zur Darstellung von Bewertungs-, Analyse- und Vergleichsvorschriften ermöglicht einerseits eine schnelle und komfortable Editierung der Vorschriften. Andererseits können Abhängigkeiten und Abläufe der Berechnungen schnell nachvollzogen werden. Dies ist ein wichtiger Aspekt zur Sicherstellung der Nachhaltigkeit der erstellen Berechnungsvorschriften.

- Transparentes Bewertungs-, Analyse- und Vergleichsframework:
  Die Bewertungs-, Analyse- und Vergleichsvorschriften sollen direkt mit Artefakten und Daten aus dem E/E-Architektur-Modell operieren. Dazu benötigt das Framework lesenden Zugriff. Für erweiterte Anwendungsfälle, wie beispielsweise die automatisierte Optimierung von E/E-Architekturen, wird ein schreibender Zugriff benötigt.
  Durch den transparenten Umgang mit den Architektur-Artefakten soll eine Nachvollziehbarkeit für die Berechnungsergebnisse erreicht werden.

- Effiziente Unterstützung von Architekturvarianten und Realisierungsalternativen:
  Aufgrund der zahlreichen Varianten von Fahrzeugen und deren Ableitung aus Plattformen muss es für Bewertungen, Analysen und Vergleiche die Möglichkeit geben, den Kontext von Architekturvarianten oder Realisierungsalternativen bei der Berechnung zu beachten. Dabei soll die Transparenz des Bewertungs-, Analyse- und Vergleichsframework erhalten bleiben.

- Sensitivität und Was-Wäre-Wenn-Szenarien:
  Bei der Analyse von Berechnungsergebnissen können Fragen nach dem Einfluss spezifischer Daten aus dem Architekturmodell auftreten. Das heißt es soll ermittelt werden, welchen Anteil

ein Parameter oder ein Merkmal der betrachteten E/E-Architektur an einem Zwischen- oder Endergebnis hat. Durch eine Sensitivitätsanalyse und dir Durchführbarkeit von Was-Wäre-Wenn-Szenarien sollen für diese Anwendungsfälle Lösungen bereitgestellt werden.

Die Implementierung des Bewertungs-, Analyse- und Vergleichsframeworks erfolgt als Erweiterung zu PREEvision [PREE]. Dieses Framework steht damit bereits für produktive Anwendungsfälle zur Verfügung. Die Umsetzung erfolgte unabhängig von der EEA-Beschreibungssprache, dass die Mechanismen auch auf andere Domänen übertragbar sind.

1.5 Aufbau der Arbeit

Im folgenden Kapitel 2 werden die für diese Arbeit relevanten technischen Grundlagen vorgestellt. Dies sind vor allem Themen aus der Metamodellierung sowie der modellbasierten Entwicklung.

In Kapitel 3 wird auf die modellbasierte Beschreibung und Visualisierung von E/E-Architekturen eingegangen. Das in Abschnitt 3.2.1 vorgestellte Verfahren zur Aufstellung formaler Metamodelle wurde im Rahmen dieser Dissertation entwickelt [GSKMG05] und fand sowohl bei der Entwicklung des EEA-Metamodells (Abschnitt 3.3) als auch bei der Entwicklung des Metamodells zur Beschreibung von Bewertungs-, Analyse- und Vergleichsvorschriften (Abschnitt 6.4) Anwendung.

Kapitel 4 geht auf Bewertungen, Bewertungsverfahren und den Vergleich von E/E-Architekturen ein. Dort werden, neben den relevanten Begriffsdefinitionen, E/E-relevante Bewertungskriterien diskutiert sowie auf Verfahren zur Multi-Kriterien-Optimierung eingegangen, die wichtig für den Vergleich von E/E-Architekturen sind.

In Kapitel 5 werden die Anforderungen an ein Bewertungs-, Analyse- und Vergleichsframework diskutiert, welches den Zielen dieser Arbeit im vorangegangenen Abschnitt gerecht wird. Diese Anforderungen werden mit dem aktuellen Stand der Technik verglichen und am Ende des Kapitels zusammengefasst.

Das Kapitel 6 beschreibt die Umsetzung des Bewertungs-, Analyse- und Vergleichsframeworks. Dort werden Metamodell, graphische Notation, die unterschiedlichen Kernartefakte, Ausführungsschicht sowie weitere Konzepte beschrieben.

Ergebnisse von Bewertungen, Analysen und Vergleichen werden in Kapitel 7 behandelt. Dort werden auch die Ergebnisse einer Kooperation mit der Porsche AG vorgestellt, bei der es darum geht, komplexe FlexRay-Parametersätze anhand des Frameworks zu berechnen und zu bewerten (siehe Abschnitt 7.2). Die Arbeit schließt in Kapitel 8 mit einer Zusammenfassung und einem Ausblick.

2 Technische Grundlagen

2.1 Metamodellierung

Metamodelle (Def. 38) bilden die standardisierte Grundlage, um ein Modell (Def. 45) mit Hilfe eines Modells (dem Metamodell) zu beschreiben. Entsprechend setzt sich der Begriff „Metamodell“ aus „meta“ (altgriechisch μετά für „hinter“, „danach“, „jenseits“) und „Modell“ zusammen, und beschreibt ein Modell, das *hinter* einem Modell steht.

Unter Metamodell versteht man *„die Konstruktionsregeln [...] für die Erstellung des Modellsystems; Metamodelle spezifizieren die verfügbaren Arten von Bausteinen (Meta-Objekte), die Beziehungen zwischen den Bausteinen (Meta-Beziehungen) sowie Konsistenzbedingungen für die Verwendung von Bausteinen und Beziehungen“* [POM97]. Danach spezifiziert ein Metamodell die Grammatik einer Modellierungssprache, indem Modellartefakte und deren mögliche Verknüpfungen beschrieben werden. Jedoch bemerkt [JECK00, S.1] dazu, dass *„Trotz des einheitlich verwandten Begriffs Metamodell, um ein Modell, das ein Modell beschreibt zu bezeichnen, [...] die zugrundeliegenden Interpretationen autorenabhängig sehr stark (variieren). […] Konkret ist zwar immer eine modellhafte Beschreibung eines Modells gemeint, jedoch im Allgemeinen keine Aussage darüber getroffen welcher Aspekt des Modells beschrieben wird.“* Dies hängt vom meist nicht explizit angegebenen, angewandten Metaisierungsprinzip [JECK00] ab. Dieses bezeichnet die gewählte Abstraktionsebene über der tatsächlichen Modellebene, durch welche der Modellierungsprozess beschrieben wird. Dies kann zum Beispiel eine dynamische Prozessschicht oder eine statische, struktur-semantische Sicht auf das Modell sein. Die struktur-semantische Sicht stellt das am weitesten verbreitete Metaisierungsprinzip dar und findet auch für das in dieser Arbeit entwickelte Metamodell, beschrieben in Kapitel 6.4, Anwendung. Zudem wird sie durch das in der Praxis oft verwendete Vier-Schichten-Modell, das im folgenden Abschnitt 2.1.1 beschrieben wird, in das Zentrum der Betrachtung gerückt.

2.1.1 Das Vier-Schichten-Modell der OMG

Das Vier-Schichten-Modell der Object Management Group [OMG] spezifiziert, wie komplexe Datenstrukturen metamodellbasiert formal beschrieben werden können. In der Praxis kommt meist ein impliziertes statisches, struktur-semantisches Metaisierungsprinzip zum Einsatz (siehe Abschnitt 2.1).

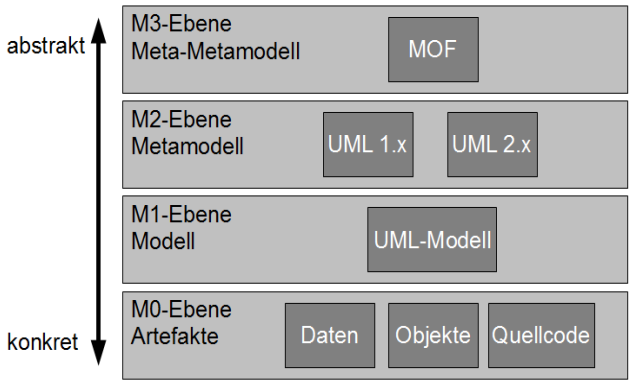


Abbildung 2.1: Das Vier-Schichten-Modell der OMG

Das Vier-Schichten-Modell besteht, dem Namen entsprechend, aus den vier Abstraktionsschichten oder -ebenen M3 (abstrahierte Schicht) bis M0 (konkrete Schicht). Eine obere Ebene abstrahiert die darunter liegende Ebene, was im Umkehrschluss heißt, dass eine untere Ebene die darüber liegende Ebene instanziiert. Die einzelnen Ebenen sind:

- M3-Ebene: Diese Ebene beinhaltet Meta-Metamodelle, die Konstrukte zur Beschreibung von Metamodellen zur Verfügung stellen. Meta-Metamodelle können sich selbst beschreiben, das heißt mit den vorhandenen Beschreibungsmitteln kann das Meta-Metamodell beschrieben werden. So wird ein Rekursionsschluss erreicht und keine weiteren abstrakteren Ebenen benötigt. Vertreter dieser Ebene sind die Meta Object Facility MOF [MOF14, MOF241], spezifiziert von der OMG, oder eCore vom Eclipse Modeling Framework EMF [EMF].
- M2-Ebene: Diese Ebene beinhaltet Metamodelle, welche die abstrakte Syntax und Semantik von Modellen spezifizieren. Ein Metamodell ist eine Instanz eines Meta-Metamodells und wird mit dessen Beschreibungsmittel formal beschrieben. Bekannter Vertreter der M2-Schicht ist das Metamodell der Unified Modeling Language UML [UML241].
- M1-Ebene: Diese Ebene beinhaltet konkrete Modelle. Ein Modell ist eine Instanz eines Metamodells aus der M2-Ebene und verwendet ausschließlich die dort spezifizierten Beschreibungsmittel. Modelle beschreiben für einen Aspekt wesentliche Zusammenhänge aus der Realität in abstrahierter Form. In dieser Schicht sind beispielsweise konkrete UML-Modelle vertreten, welche Struktur und Verhalten von Softwaresystemen beschreiben (siehe Abschnitt 2.3).
- M0-Ebene: Konkrete Daten, Quelltexte oder reale Zusammenhänge, welche durch Modelle der M1-Ebene beschrieben werden, befinden sich in der M0-Ebene. Artefakte dieser Ebene sind Instanzen der Modelle aus der M1-Ebene.

In der Spezifikation für die MOF 2.0 lockert die OMG das starre Vier-Schichten-Modell auf, indem beliebig viele Ebenen, mindestens aber zwei, zum Einsatz kommen können. Dabei müssen folgende Prinzipien erfüllt sein:

- Eine Schicht spezifiziert die direkt darunter liegende Schicht
- Eine Schicht ist eine Instanz der direkt darüber liegenden Schicht

Das in dieser Arbeit entwickelte Metamodell zur Beschreibung von Berechnungsmetriken (siehe Kapitel 6.4) sowie das Metamodell zur Beschreibung von E/E-Architekturen EEA-ADL im Fahrzeug (siehe Kapitel 3.3) basieren auf dem Vier-Schichten-Modell der OMG.

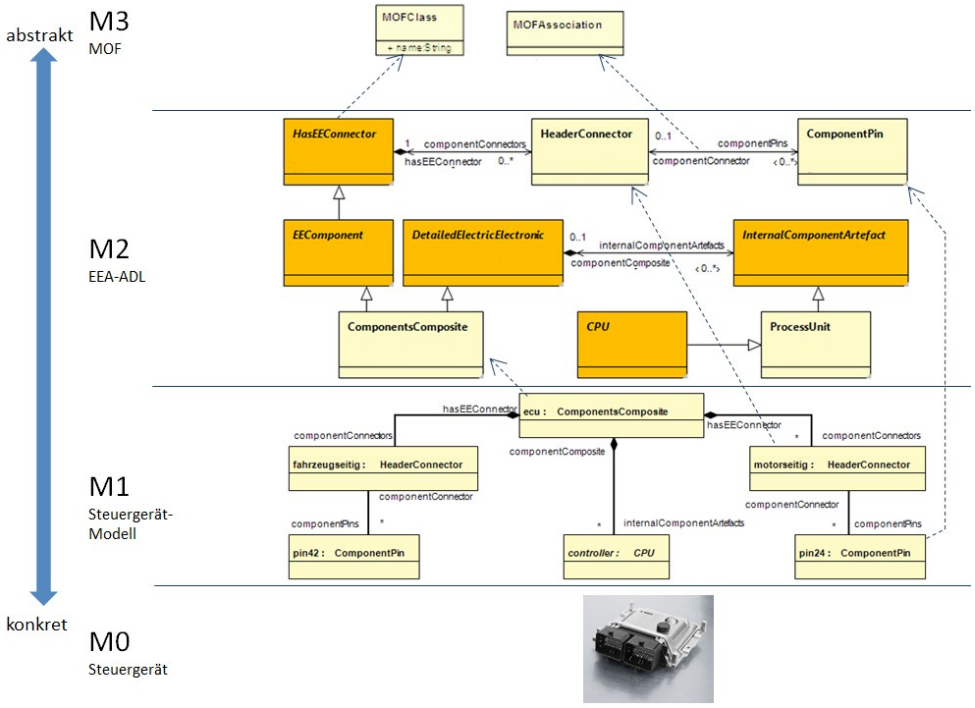


Abbildung 2.2: Beispiel-Modellierung eines Steuergeräts gemäß Vier-Schichten-Modell

Abbildung 2.2 zeigt eine mögliche Modellierung eines Steuergerätes gemäß Vier-Schichten-Modell der OMG. In der M0-Ebene ist eine Abbildung des Steuergerätes dargestellt. Dessen modellbasierte Beschreibung ist Bestandteil der M1-Ebene und wird dort als Objektdiagramm (Notation und Details sind in Abschnitt 2.3.4.2 erklärt) gezeigt. Darin besteht das Steuergerät, repräsentiert durch das Objekt *ecu:ComponentsComposite*, aus einem *controller:CPU*, einem fahrzeugseitigem- (*fahrzeugseitig:HeaderConnector*) und einem motorseitigem (*motorseitig:HeaderConnector*) Stecker. Jeder der Stecker verfügt in dieser Abbildung über einen Pin (*pin42:ComponentPin* bzw. *pin24:ComponentPin*).

In der M2-Ebene wird der zugehörige Teil des EEA-ADL Metamodells dargestellt. Dort werden die Zusammenhänge modelliert, welche dann in der M1-Ebene verwendet werden können. Im konkreten Beispiel ist das Objekt der M1-Ebene *ecu:ComponentsComposite* eine Instanz der Metaklasse *ComponentsComposite*. Über die vererbten Beziehungen kann es über Stecker (*HeaderConnector*) und interne Komponenten (z.B. *CPU*) verfügen.

Die Repräsentation der Metaartefakte der M2-Ebene werden in der M3-Ebene durch MOF

beschrieben. Eine Metaklasse der M2-Ebene ist eine Instanz der MOF-Klasse (MOFClass), eine Metaassoziation ist eine Instanz der MOF-Assoziation (*MOFAssociation*). Im folgenden Abschnitt wird die MOF detailliert erklärt.

2.1.2 Meta Object Facility MOF

Die Meta Object Facility MOF [MOF241] ist ein Standard der OMG [OMG] und spezifiziert ein Meta-Metamodell gemäß des 4-Schichten-Modells (beschrieben in 2.1.1). Zurzeit liegt der Standard in der Version 2.4.1 vor. MOF stellt Konstrukte zur Verfügung, um domänenspezifische Metamodelle wie das UML-Metamodell oder die EEA-ADL zu modellieren. Die konkrete Syntax der MOF stellt eine Teilmenge des UML-Kerns dar (siehe 2.3). Zur Modellierung von MOF-konformen Metamodellen können somit auch Klassendiagramme der UML (näher beschrieben in Kapitel 2.3.4.1) verwendet werden.

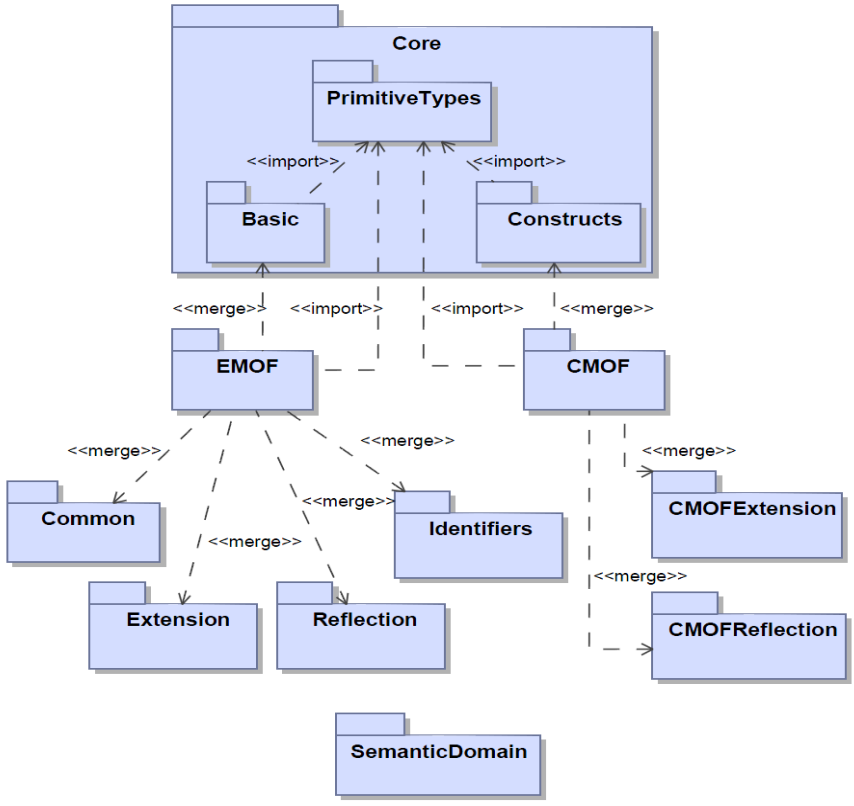


Abbildung 2.3: Paketstruktur des MOF-Metamodells

In Abbildung 2.3 ist die Paketstruktur des MOF 2.0-Standards dargestellt. Die mit einer Lasche versehenen Rechtecke repräsentieren die verschiedenen Pakete des Standards. Die Pakete sind mit annotierten, gerichteten Abhängigkeiten miteinander verbunden, dargestellt als gestrichelte Pfeile. Die

<<*import*>>-Annotation bedeutet, dass ein Paket die Inhalte des am Pfeilende referenzierten Pakets importiert, d.h. für seine eigenen Inhalte nutzbar macht. Die <<merge>>-Annotation bedeutet, dass die Inhalte eines Pakets mit den Inhalten des am Pfeilende referenzierten Pakets verschmolzen (engl. Merge) werden.

Ausgangspunkt ist das *Core*-Paket, welches der MOF 2.0-Standard mit dem UML 2.0-Standard teilt. Darin enthalten sind die Pakete *Primitive Types*, *Basic* und *Constructs*. Das Paket *Primitive Types* spezifiziert einfache Datentypen wie ganzzahlig Zahlen, Zeichenketten oder boolesche Datentypen. Im Paket *Basic* findet sich die Spezifikation wesentlicher Kernkonzepte. Dazu gehören das Paketkonzept, komplexe Datentypen, Vererbung, Attribute und Operationen. Das Paket *Constructs* behandelt Namensräume, Sichtbarkeiten, Assoziationen inklusive Rollennamen und Multiplizitäten sowie die Abhängigkeiten *merge* und *import*, mit denen Paketinhalte verschmolzen (*merge*) bzw. anderen Paketen zur Verfügung gestellt (*import*) werden können und wiederum selbst in der in Abbildung 2.3 dargestellten Paketstruktur zum Einsatz kommen.

Die Pakete *EMOF* (Essential MOF) und *CMOF* (Complete MOF) spezifizieren den MOF-spezifische Konzepte. Die *EMOF* umfasst alle benötigten Konstrukte zum Aufbau einfacher, klassenbasierter Metamodelle und entspricht dem MOF-Standard der Version 1.4 [MOF14]. *CMOF* baut auf *EMOF* auf und spezifiziert die MOF ab Version 2.0 [MOF241] vollständig und umfasst komplexe Sprachmuster. Alle Metamodelle der OMG basieren auf diesem Standard.

2.2 Modellbasierte Ansätze

2.2.1 Model Driven Architecture MDA

Die Object Management Group OMG [OMG] spezifiziert die Model Driven Architecture MDA [MDA] als Rahmen zur modellgetriebenen Entwicklung von Software. Die MDA selbst unterliegt Standardisierungsbemühungen und integriert unterschiedliche andere Standards der OMG, wie zum Beispiel die UML. Begriffe aus den Standardisierungsdokumenten der OMG haben sich im Umfeld der modellgetriebenen Entwicklung durchgesetzt und etabliert [STA05].

Die MDA geht von einem mehrstufigen Ansatz aus, wobei jede Stufe einen Abstraktionsgrad des Modells beinhaltet. Die Stufenübergänge werden durch Transformationen[2] realisiert. Dabei kann es

2 Eine Transformation stellt eine Übergangsvorschrift dar, mit deren Hilfe ein Modell in ein anderes Modell überführt (transformiert) werden kann. Man unterscheidet zwischen Modell-zu-Modell- und Modell-zu-Text-Transformationen [REI05].

sich um automatische / automatisierte oder manuell durchgeführte Transformationen handeln. Die Abstraktion erfolgt durch Trennung von Anforderungen, Funktionalität und Technik.

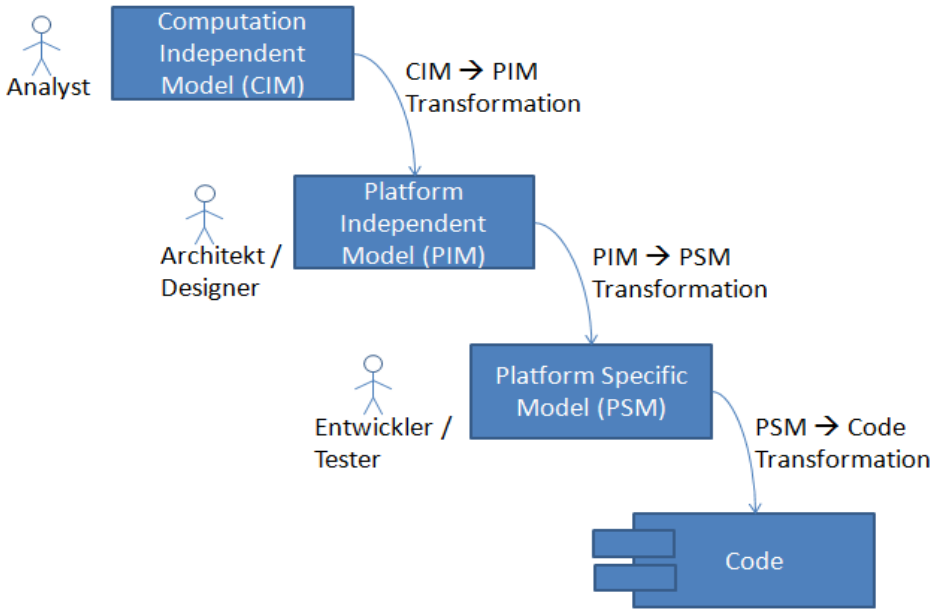


Abbildung 2.4: Übersicht Model Driven Architecture

Ausgangspunkt ist, wie in Abbildung 2.4 gezeigt, ein Computation-Independent-Model (CIM). Es beinhaltet die Nutzeranforderungen des Systems (Requirements). Das CIM repräsentiert domänenspezifisches Wissen und bildet keine funktionale Strukturen ab. In einem Transformationsschritt wird daraus ein sogenanntes Platform-Independent-Model (PIM) erstellt. Dieses Modell beschreibt die grundlegende Funktionalität, wobei Aspekte, welche die konkreten Kenntnisse der Zielplattform benötigen, ausgelassen werden. Durch Verfeinerung und Spezialisierung auf die konkrete Zielplattform wird daraus das Platform-Specific-Model (PSM). Dieses Modell beinhaltet alle Realisierungsdetails und ist auf die jeweilige Zielplattform applizierbar. Die Anzahl der Zwischenstufen und -schritte zwischen PIM und PSM wird durch MDA nicht begrenzt.

Die MDA bindet unterschiedliche, existierende Standards der OMG wie MOF oder UML ein.

2.2.2 Model Driven Development MDD

Die Modellgetriebene Entwicklung (Model Driven Development, MDD) umfasst einen Entwicklungsansatz, bei dem Modelle nicht nur zur Dokumentation eingesetzt wird, sondern bei dem computerlesbare Modelle zum Einsatz kommen, *„um die Absichten präzise zu erfassen und ihre Implementierung durch automatische Verfeinerung partiell oder komplett zu automatisieren [...]“* [GSCK06, S.117]. Unter der Automatisierung ist gemeint, dass diese Modelle *„[...] kompiliert wer-*

den, um ausführbare Programme zu erzeugen, oder um mit ihrer Hilfe die manuelle Entwicklung ausführbarer Programme zu erleichtern […].“ [GSCK06, S.117]

Abbildung 2.5 zeigt eine Gegenüberstellung zwischen einem modellgetriebenen Entwicklungsansatz zur herkömmlichen, manuellen Implementierung. Auf der Ordinate sind der Informationsgehalt, auf der Abszisse die Detaillierung aufgetragen. Die blaue Kurve zeigt den Verlauf der manuellen Implementierung an, die orange Kurve zeigt den Effekt durch Modellentwicklung im Sinne der MDD.

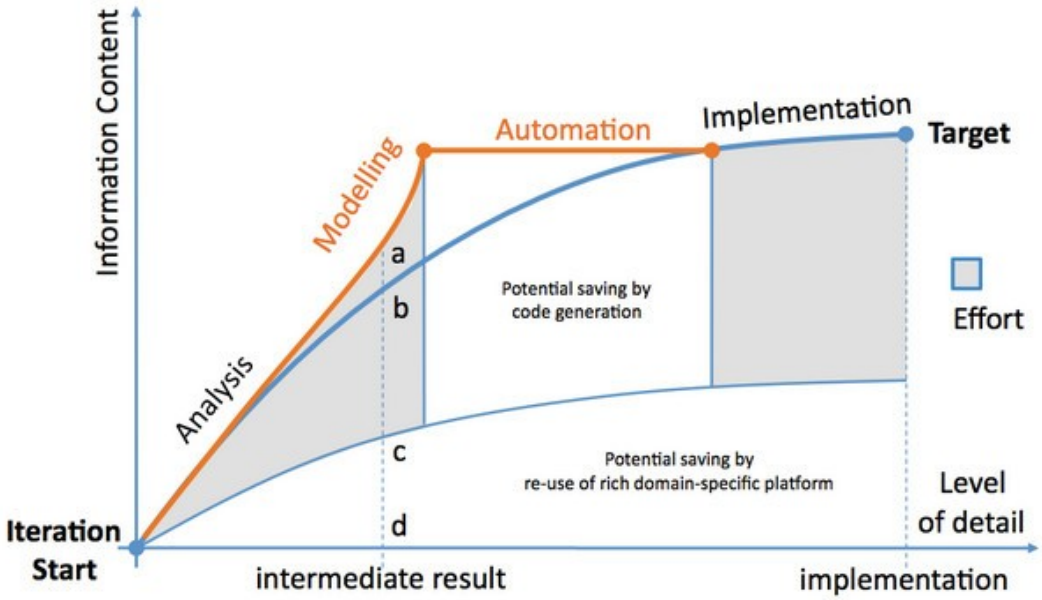


Abbildung 2.5: MDD im Vergleich zur manuellen Implementierung [STA06]

Der initiale Aufwand bei der Analyse ist bei MDD zunächst höher, um den Informationsgehalt des Modells auf das geforderte Niveau zu bringen. Durch einen folgenden Automatisierungsschritt (in diesem Beispiel wird die Generierung von Quellcode angenommen) macht der Implementierungsfortschritt einen Sprung, so dass sich nach dieser Theorie Einsparpotentiale beim Aufwand ergeben. Diese sind im Wesentlichen davon abhängig, wie viel Informationsgehalt (*Information Content*) durch die Modellierung eingebracht werden kann und wie viel Implementierungsarbeiten nach dem Automatisierungsschritt noch notwendig sind.

2.2.3 Domain Specific Language DSL

Eine domänenspezifische Sprache (DSL) modelliert die in einer bestimmten Fachdomäne verwendeten Begriffe und sind für Domänen-affine Benutzer leicht verständlich. DSL ist eine Technologie, die aus dem Umfeld der Model Driven Architecture MDA (siehe Abschnitt 2.2.1) und Model Dri-

ven Development MDD (siehe Abschnitt 2.2.2) entstammt. Martin Fowler definiert eine DSL als *„a computer programming language of limited expressiveness focused on a particular domain“* [FOW10, S.27] (Def. 20). Anders ausgedrückt: *„Eine wohldefinierte DSL ist eine leistungsstarke Implementierungssprache, die viel präzisere Ausdrucksmöglichkeiten als eine Allzwecksmodellierungssprache wie etwa die UML bietet. Eine DSL kann entweder eine textliche oder grafische Notation haben.“* [GSCK06, S.120] Eine domänenspezifische Sprache muss nach [GSCK06], [BIS09] folgende Kriterien erfüllen:

- *„Die Domäne muss präzise definiert sein.“* [BIS09, S.94]
- *„Die Konzepte in der Modellierungssprache müssen für Anwender aus der Domäne klar verständlich sein. Ebenso sollten die Konzepte so benannt sein, dass Domänenexperten sie sofort zuordnen können.“* [BIS09, S.94]
- *„Die grafische oder textliche Notation für die Modellierungssprache sollte für den fraglichen Zweck leicht zu verwenden sein.“* [GSCK06, S.119]
- *„Die Modellierungsprache sollte eine klare Grammatik zur zweifelsfreien Definition von Ausdrücken besitzen.“* [BIS09, S.94]
- *„Die Bedeutung aller wohlgeformten Ausdrücke sollte wohldefiniert sein, so dass Benutzer Modelle erstellen können, die von anderen Benutzern verstanden werden...“* [GSCK06, S.119].

Bei der EEA-ADL (siehe Abschnitt 2.9.1) handelt es sich beispielsweise um eine domänenspezifische Sprache zur Modellierung von Elektrik/Elektronik-Architekturen im Fahrzeug. Mit PREEvision ([PREE], siehe Abschnitt 2.9) steht ein Werkzeug zur Verfügung, welches für die EEA-ADL eine graphische Notation zur Verfügung stellt.

2.3 Unified Modeling Language UML

Die Unified Modeling Language UML [UML241] ist ein von der OMG [OMG] veröffentlichter Standard zur Beschreibung von Software- und anderen Systemen. In der Version 2.4.1 wurde die UML, unterteilt in Infra- und Superstructure, als ISO-Standard verabschiedet [ISO19505-1] [ISO19505-2]. Diese standardisierte Sprache definiert Bezeichner für die wichtigen Modellierungsbegriffe sowie deren Beziehungen untereinander. Ferner spezifiziert die eine graphische Notation, mit deren Hilfe durch unterschiedliche Diagrammtypen sowohl statische, als auch dynamische Zusammenhänge modelliert werden können.

2.3.1 Geschichte der UML

Die erste Version der UML wurde von Booch, Jacobson und Rumbaugh an die OMG übergeben, welche am 19. November 1997 die UML als Standard akzeptierte [RUM98]. Die ersten Vorläufer der UML entstanden bereits Anfang der 1990er Jahre, als sich ein Paradigmenwechsel von strukturierter Softwareentwicklung hin zu objektorientierten Ansätzen abzeichnete.

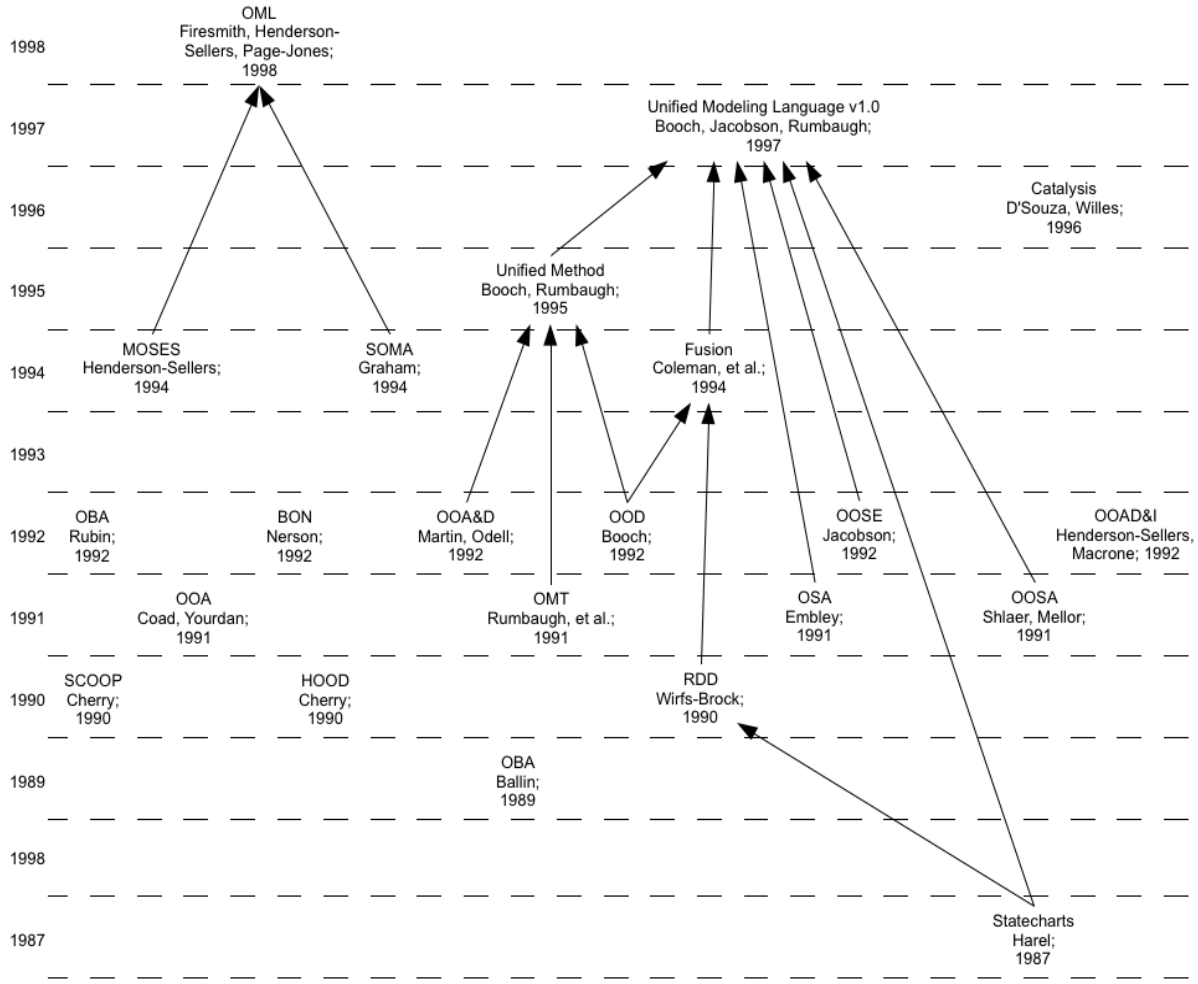


Abbildung 2.6: Entstehungsgeschichte der UML bis 1998 in Anlehnung an [JECK03]

Abbildung 2.6 zeigt eine Übersicht der unterschiedlichen Ansätze, die schließlich in die Entstehung der UML im Jahre 1997 und der OML [OML] im Jahre 1998 mündeten. Neben konzeptionellen Ähnlichkeiten und Überschneidungen gab es jedoch auch eine Vielzahl unterschiedlicher Zielrichtungen, aber auch Inkompatibilitäten untereinander. Die außergewöhnliche Vielfalt der Ansätze beruhte nicht nur auf unterschiedlichen wissenschaftlichen, sondern auch wirtschaftlichen Interessen. Man spricht dabei von den „Methodenkriegen der Softwareentwicklung“ [JECK04, Folie 9], aus denen sich die UML durchgesetzt hat.

Wesentlich für die Entstehung waren die Unified Method von Booch und Rumbaugh [BOO95] aus dem Jahre 1995 und der OOSE-Ansatz (Object-Oriented Software Engineering) [JAC92] von Jacobson (mit Booch und Rumbaugh auch „die drei Amigos“ genannt) aus dem Jahre 1992. Die weitere Entwicklung und Standardisierung wurde an die OMG übergeben, wo die UML seitdem kontinuierlich weiterentwickelt wird. Die aktuelle Version ist UML 2.4.1, der aktuelle Stand in der Entwicklung ist Version 2.5.

2.3.2 Aufbau der UML

Während die Spezifikation für 1.x-Versionen der UML aus einem Dokument bestand, setzt sich die Spezifikation der UML ab Version 2.0 aus mehreren Teilspezifikationen zusammen. Die *UML Infrastructure Specification* [UML241Inf] bildet das Fundament, in dem die wichtigsten Elemente und Konstrukte definiert werden, wie zum Beispiel Klassen, Assoziationen oder Attribute, die von den anderen Teilspezifikationen spezialisiert werden. Darauf aufbauend beschreibt die *UML Superstructure Specification* [UML241Sup] UML-spezifische Modellelemente wie Anwendungsfälle, Aktivitäten und Zustandsautomaten. Der dritte Teil der Spezifikation *UML Object Constraint Language* [OMG231OCL] beschreibt eine Sprache, mit deren Hilfe Bedingungen für Modellierungsartefakte der anderen Spezifikationsteile definiert werden können.

Mit der Spezifikation *Diagram Definition* [OMG10DD] wird ein Format für den Austausch von graphischen Layout-Informationen der modellierten Diagramme spezifiziert. Diese sind zum Verständnis und Austausch von UML-Modellen essentiell. Die Diagrammtypen der UML werden in Abschnitt 2.3.4 vorgestellt, wobei die für diese Arbeit relevanten Diagramme detaillierter beschrieben werden. Die zum Verständnis notwendige, wesentliche Modellierungsartefakte werden im folgenden Abschnitt 2.3.3 erläutert.

2.3.3 Wesentliche Modellierungsartefakte

In diesem Anschnitt werden die wesentlichen Modellierungsartefakte der UML beschrieben. Für weiterführende Informationen sei an dieser Stelle auf die UML-Dokumentation [UML241] verwiesen.

Die konkrete Syntax der MOF (siehe 2.1.2) ist als Teilmenge der UML spezifiziert. Mit Hilfe dieser Konstrukte ist es möglich, MOF-konforme Metamodelle zu beschreiben. Die hier vorgestellten Modellierungsartefakte beziehen sich alle auf die Modell-Ebene. Folglich verfügen ihre Metamodell-Pendants in der MOF über das Präfix „Meta“ (z.B. Metapaket, Metaklasse oder Metaattribut). Un-

terschiede zwischen UML und MOF werden bei den jeweiligen Modellierungsartefakten beschrieben.

Die Hierarchisierung erfolgt durch das so genannte Paket-Konzept (engl. Package), welches eine Ausprägung des Kompositum-Entwurfsmusters (siehe 2.5.1) ist. Dazu werden Pakete definiert, die wiederum Pakete und andere Modellierungsartefakte, enthalten. Die Hierarchisierung dient vor allem der semantischen Ordnung des Modells. Jede Hierarchiestufe erweitert den entsprechenden Namensraum. Gleichnamige Modellierungsartefakte in unterschiedlichen Paketen sind von daher eindeutig unterscheidbar.

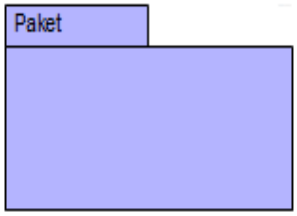


Abbildung 2.7: Darstellung eines Pakets

In Abbildung 2.7 ist ein Paket dargestellt. Der Name des Pakets darf laut Spezifikation entweder in der Lasche oder im oberen Teil des Rechtecks unterhalb der Lasche stehen. Zwischen Paketen dürfen Abhängigkeiten (sie werden gemäß MOF als gestrichelte Pfeile visualisiert) modelliert werden. Diese sagen aus, dass die Inhalte des untergeordneten Pakets von Inhalten des übergeordneten Pakets abhängig sind. Die Art der Abhängigkeit wird durch eine Annotation am Pfeil bestimmt (siehe auch Abschnitt 2.1.2).

Klassen sind bei Modellen wesentliches Modellierungsartefakt. In der UML spezifiziert eine Klasse eine mögliche Typisierung ihrer Instanzen (Objekte) (siehe 2.1.1). Innerhalb eines Metamodells spezifiziert eine Metaklasse zusätzlich eine semantische Einheit eines Modellierungskonzeptes ab. Klassen können konkret oder abstrakt sein. Konkrete Klassen können auf die spezifizierende Typisierung der Klasse instanziiert werden. Abstrakte Klassen können nicht direkt instanziiert werden. Sie werden eingesetzt, um gemeinsame Eigenschaften oder Konzepte abzubilden, die jedoch einer weiteren Spezialisierung, modelliert über Generalisierungen (s.u.), bedürfen.

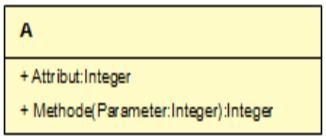


Abbildung 2.8: Beispiel einer Klasse mit Attribut und Methode

Die Eigenschaften einer Klasse werden durch Attribute modelliert. Diese sind einem Datentyp (z.B. *Integer*) zugeordnet. In Abbildung 2.8 ist eine Klasse *A* mit einem Attribut und einer Methode mit einem Parameter abgebildet. Im Gegensatz zur UML werden in Metaklassen üblicherweise keine Methoden zur Verhaltensmodellierung modelliert, dies ist abhängig vom verwendeten Metaisierungsprinzip (siehe 2.1; [JECK00]). Die in dieser Arbeit vorgestellten Metamodelle beschreiben ausschließlich statische, struktur-semantische Zusammenhänge, so dass auf die Modellierung von Methoden verzichtet wird.

Klassen können miteinander in Beziehung stehen. Es gibt drei Arten von Beziehungen, die als Linien visualisiert werden (Abbildung 2.9). Die Pfeilenden der Linien zeigt in Richtung der Sichtbarkeit der beteiligten Klassen. Beidseitige Pfeilenden drücken gegenseitige Sichtbarkeit aus. Die *Assoziation* (dargestellt als einfache Linie ohne Rauten) drückt die schwächste der Beziehung aus und sagt aus, dass sich die zwei beteiligten Klassen „kennen“. Die *Aggregation* (dargestellt als nicht ausgefüllte Raute) drückt eine stärkere Bindung aus, derart, dass die aggregierende Klasse A die aggregierte Klasse B temporär besitzt. Das heißt, wird eine Instanz von A gelöscht, so bedeutet dies nicht, dass aggregierte Instanzen von B mit gelöscht werden.

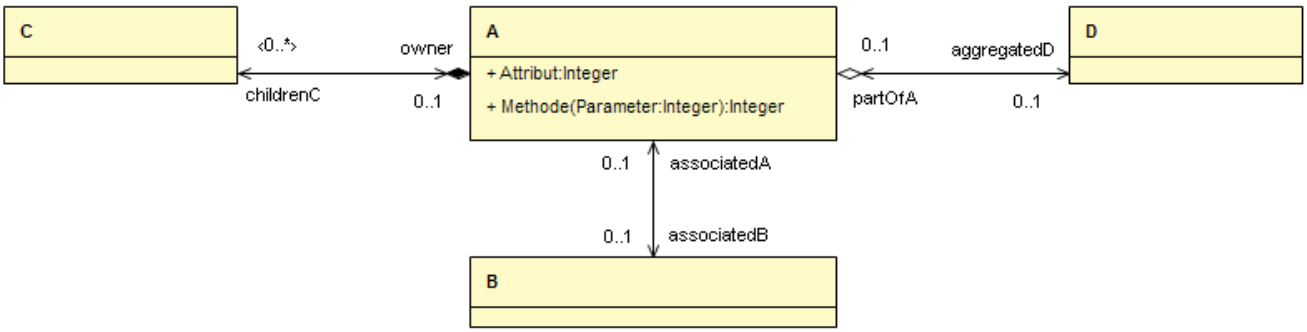


Abbildung 2.9: Beispiel für Komposition, Aggregation und Assoziation

Die dritte Art von Beziehungen ist die *Komposition* (dargestellt als ausgefüllte Raute). Wie die Aggregation drückt die Komposition eine *besitzt*-Relation aus, mit dem Unterschied, dass diese Beziehung stärker ist, sodass die untergeordneten Instanzen gelöscht werden, sobald das Vaterobjekt gelöscht wird.

In der Metamodellierung mit statischem, struktur-semantischem Metaisierungsprinzip spielt der semantische Unterschied zwischen einer Aggregation und einer Komposition praktisch keine Rolle [GSKMG05]. Dieser Unterschied kann in Form von statischen Klassendiagrammen, welche bei der Metamodellierung zum Einsatz kommen, nicht vollständig aufgelöst werden. Allein durch unter-

schiedliche Argumentation kann für denselben Zusammenhang eine Aggregation oder eine Komposition verwendet werden. Aus diesem Grund beinhalten alle in dieser Arbeit vorgestellten Metamodelle ausschließlich Assoziationen und Kompositionen. Auf den Einsatz von Aggregationen wird verzichtet.

Vererbungshierarchien von Klassen werden mit Generalisierungen modelliert. Dabei erbt die Unter- oder Subklasse alle Attribute, Methoden und Beziehungen zu anderen Klassen von der Ober- oder Superklasse.

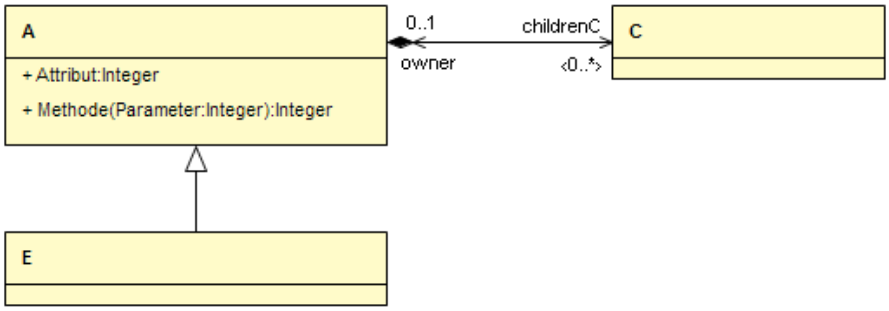


Abbildung 2.10: Beispiel für eine Generalisierung

In Abbildung 2.10 wird eine Generalisierung als Pfeil von der Klasse E (Unterklasse) zu A (Oberklasse) dargestellt.

2.3.4 Diagramme

Die UML spezifiziert insgesamt 13 Diagrammtypen (siehe Tabelle 2). Darunter sind sechs Diagrammtypen, die statische Inhalte wiedergeben und sieben Diagrammtypen, die dynamische Inhalte darstellen.

Diagrammtyp	
statisch	**dynamisch**
Klassendiagramm	Anwendungsfalldiagramm
Paketdiagramm	Aktivitätsdiagramm
Objektdiagramm	Zustandsdiagramm
Kompositionsstrukturdiagramm	Sequenzdiagramm
Komponentendiagramm	Kommunikationsdiagramm
Verteilungsdiagramm	Zeitverlaufsdiagramm
	Interaktionsübersichtsdiagramm

Tabelle 2: Übersicht der UML Diagrammtypen

Statische Diagramme visualisieren strukturelle Zusammenhänge des modellierten Systems mit den jeweiligen Eigenschaften und Beziehungen untereinander. Dynamische Diagramme visualisieren zeitliche oder ereignisgesteuerte Abläufe. Interaktionsdiagramme bilden eine Untergruppe der dynamischen Diagramme und visualisieren Kommunikationsaspekte des modellierten Systems.

In den folgenden Abschnitten wird detaillierter auf die für diese Arbeit relevanten Diagrammarten eingegangen. Für die übrigen Diagrammtypen sei auf die UML-Spezifikation [UML241] verwiesen.

2.3.4.1 Klassendiagramme

Statische Elemente objektorientierter Programmiersprachen lassen sich auf Klassendiagrammen abbilden. In einem solchen Diagramm werden Pakete, Klassen, Schnittstellen, Attribute, Methoden, Beziehungen (Assoziationen, Aggregationen und Kompositionen), Vererbungen und Abhängigkeiten dargestellt (Abbildung 2.11).

Das Symbol für Klassen ist ein Rechteck, welches in bis zu drei übereinander liegende Teile partitioniert sein kann. Im ersten, obersten Teil steht der Klassenname sowie bei Bedarf annotierte Stereotypen oder der Paketpfad. Bei abstrakten Klassen wird der Klassenname kursiv geschrieben.

In der zweiten Sektion werden die Attribute der Klasse untereinander aufgelistet. Vor dem Namen des Attributs steht dessen Sichtbarkeit. Diese sagt aus, ob das Attribut nur innerhalb der Klasse bekannt ist („-“ für *private*), auch für Subklassen sichtbar ist („#“ für *protected*), oder öffentlich ist („+“ für *public*). Hinter dem Attributnamen wird der Typ des Attributs (z.B. *int* für ganze Zahlen, Integer). Der Bereich zur Darstellung von Attributen innerhalb der Klasse ist optional und kann bei Bedarf weggelassen werden. Zudem schreibt die UML auch nicht vor, dass alle vorhandenen Attribute der Klasse im Klassensymbol dargestellt werden müssen.

Im dritten Teil werden die Methoden der Klasse untereinander aufgestellt. Es wird die komplette Methodensignatur dargestellt, bestehend aus Methodenname, gefolgt von Parametern und Parametertypen in Klammern sowie der Rückgabetyp nach einem Doppelpunkt. Analog zu den Attributen wird den Methoden das Zeichen für die Sichtbarkeit der Methode vorangestellt. Ebenfalls müssen nicht alle Methoden der Klasse im Klassensymbol dargestellt werden.

Die Darstellung von Schnittstellen (*Interface*) ist analog zur Darstellung von Klassen, mit dem Unterschied, dass in der oberen rechten Ecke ein Kreis dargestellt ist und es keine Sektion für Attribute gibt.

Klassen werden durch Assoziationen in Beziehung zueinander gesetzt. Diese Assoziationen werden

als Linie zwischen den Klassen dargestellt. An den Enden werden Rollennamen und Multiplizität eingeblendet. Ein optional pfeilförmiges Linienende zeigt die Navigierbarkeit der Assoziation in diese Richtung. Eines der Linienenden kann zusätzlich eine leere oder ausgefüllte Raute zeigen. Die Form des Endes visualisiert, ob es sich um eine Assoziations- (ohne Raute), Aggregations- (leere Raute) oder Kompositionsbeziehungen (ausgefüllte Raute) handelt.

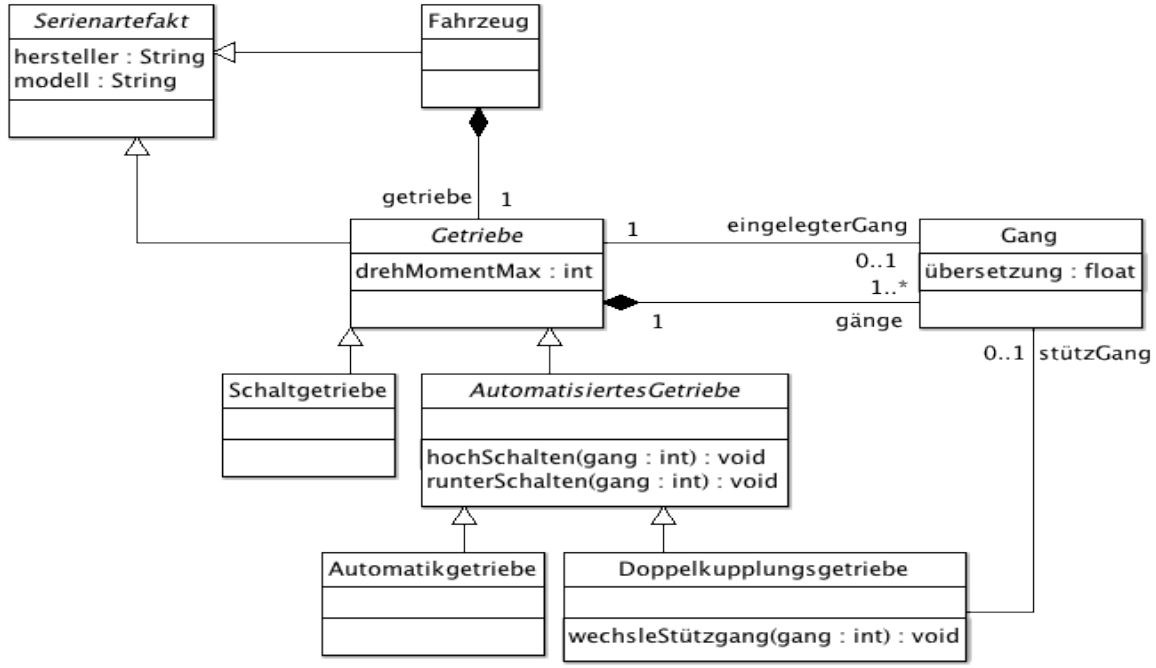


Abbildung 2.11: Beispiel Klassendiagramm

In Abbildung 2.11 ist ein Beispiel für ein Klassendiagramm dargestellt. Es zeigt eine einfache Modellierung eines *Fahrzeugs* mit einem *Getriebe*. Beide stammen aus einer Serienproduktion und erben von der abstrakten Klasse *Serienartefakt*. Damit verfügen sie über die Attribute *hersteller* und *modell*. Das *Getriebe* besitzt mindestens einen *Gang*, von denen genau einer eingelegt sein kann. Als Spezialisierung des Getriebes gibt es *Schaltgetriebe*, *Automatikgetriebe* und *Doppelkupplungsgetriebe*. Die Letzteren beiden erben von der abstrakten Klasse *AutomatisiertesGetriebe* und können somit automatisiert geschaltet werden. Dazu stehen die beiden Methoden *hochSchalten(gang : int)* und *runterSchalten(gang : int)* zur Verfügung.

2.3.4.2 Objektdiagramme

Objektdiagramme bilden den Zustand des modellierten Systems zur Laufzeit zu einem bestimmten Zeitpunkt ab, indem Instanzen von Klassen (Objekte) sowie die instanziierten Relationen (Links) zum relevanten Zeitpunkt dargestellt werden. Attribute von Objekten tragen den konkreten Wert zu diesem Zeitpunkt.

Die graphische Notation von Objektdiagrammen basiert auf der von Klassendiagrammen. Objekte werden als Rechteck symbolisiert. Im oberen Teil steht unterstrichen der Bezeichner des Objektes sowie der Name der Klasse, welche durch das Objekt instanziiert ist. Im unteren Teil stehen die Attribute der Klasse mit den Werten, welche sie in dem Objekt tragen.

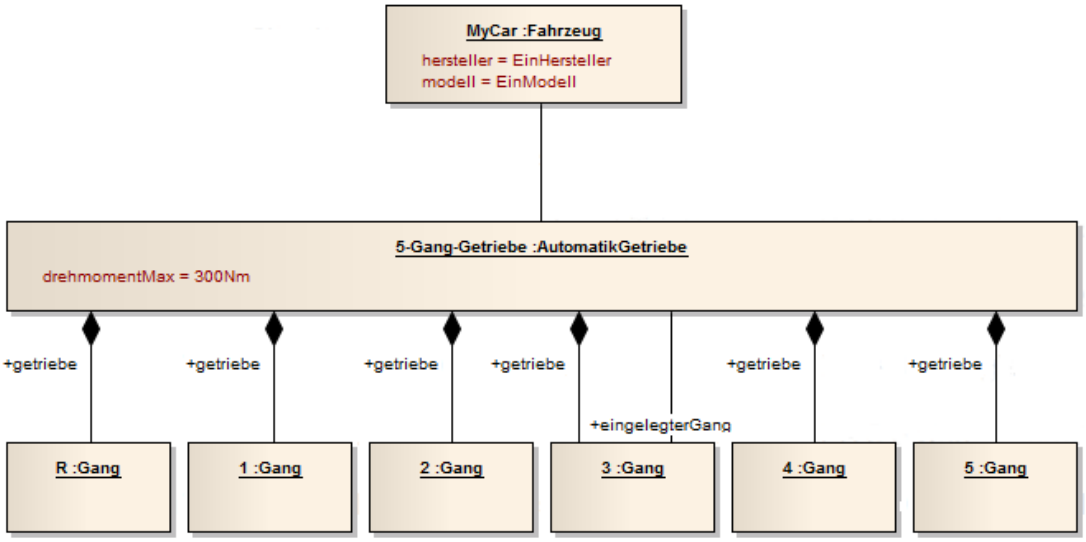


Abbildung 2.12: Beispiel Objektdiagramm

Abbildung 2.12 zeigt ein Objektdiagramm, welches Instanzen der im Klassendiagramm aus Abbildung 2.11 gezeigten Klassen beinhaltet. Die Instanz der Klasse Fahrzeug ist *MyCar* benannt und trägt die Werte der Attribute *hersteller* und *modell*.

Das Fahrzeug ist assoziiert mit dem Objekt *5-Gang-Getriebe* des Typs *Automatikgetriebe*. Das Getriebe-Objekt besteht über Kompositionen aus einem Rückwärtsgang und 5 Vorwärtsgängen. Die Assoziation zwischen dem Getriebe und dem dritten Gang zeigt den aktuell eingelegten Gang.

2.3.4.3 Zustandsdiagramme

Zustandsdiagramme visualisieren endliche Zustandsautomaten nach David Harel [HAR87]. Die UML erweitert das Konzept der endlichen Zustandsautomaten um Hierarchie und Nebenläufigkeit. Außerdem sind Aspekte von Moore-Automaten [VOS11] (Aktionen beim Betreten eines Zustandes) und Mealy-Automaten [VOS11] (Aktionen bei Transitionsübergängen) integriert [RUM98].

Zustandsautomaten eignen sich einerseits zur Verhaltensmodellierung von Klassen. Andererseits kann ein Zustandsautomat zur Modellierung von Protokollen verwendet werden.

Zustandsdiagramme bestehen aus Zuständen und Transitionen. Ein Zustand repräsentiert ein Set von internen Werten eines Objektes. Zustände werden im Diagramm als Boxen mit abgerundeten Ecken gezeichnet. Im oberen Teil der Box steht der Name des Zustands. Der Übergang von einem Zustand zu einem anderen erfolgt durch Transitionen, die als Pfeile zwischen zwei Zuständen dargestellt werden. Die Aktivierung einer Transition geschieht durch Ereignisse. Ferner werden diese noch an Bedingungen geknüpft. Ereignis und Bedingung werden an der Transition visualisiert.

Zustandsautomaten haben definierte Start- und Endzustände, die als ausgefüllte Kreise bzw. als

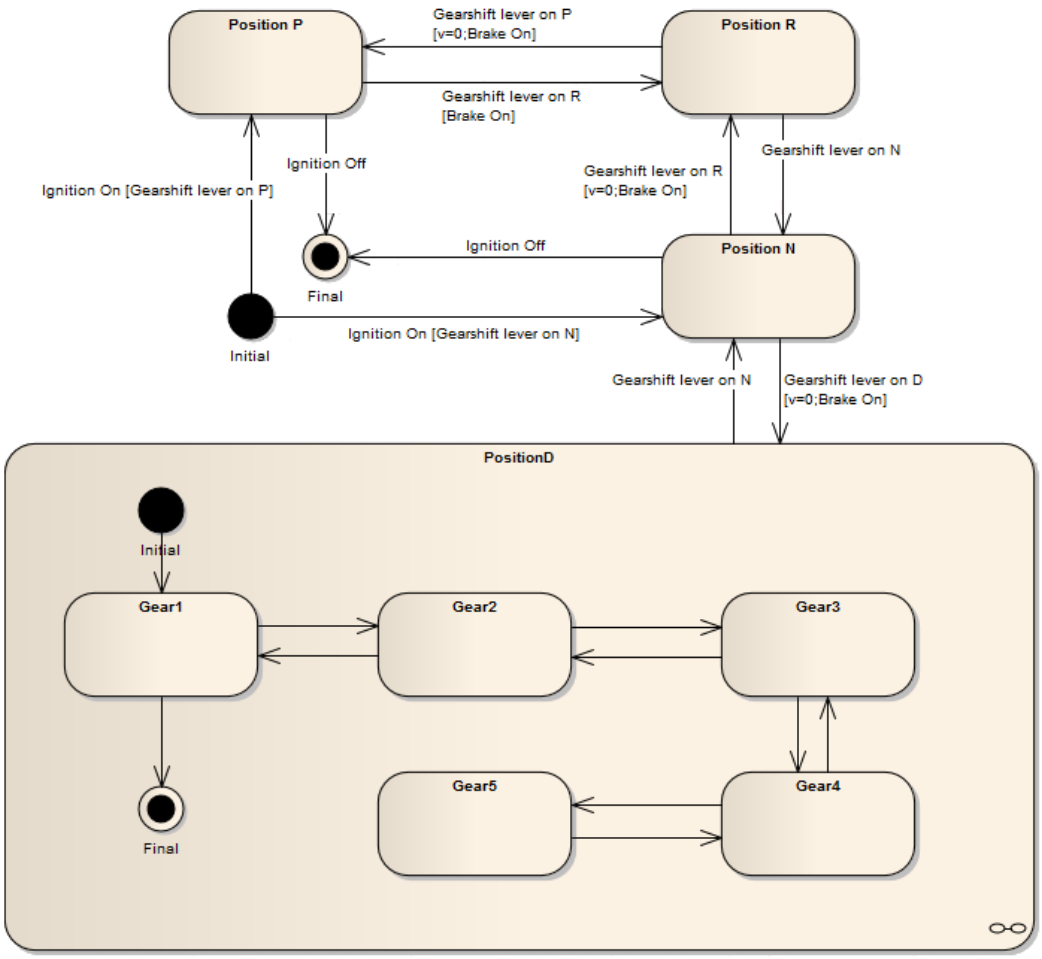


Abbildung 2.13: Beispiel Zustandsdiagramm

kleine ausgefüllte Kreise mit Ring visualisiert werden.

In Abbildung 2.13 wird ein Beispiel für einen Zustandsautomaten gezeigt. Das Beispiel modelliert einen einfachen Automaten für eine Getriebesteuerung. Der Anfangszustand wird mit dem Einschalten der Zündung verlassen und es wird der Zustand für die Park- oder neutrale Position betreten, je nachdem, in welcher Position der Wählhebel des Getriebes steht.

Jede Position wird durch einen eigenen Zustand repräsentiert, die Transitionen dazwischen geben gültige Übergänge an, die zum Teil mit Bedingungen verknüpft sind. So darf beispielsweise die Position N vom Rückwärtsgang (Position R) erst dann betreten werden, wenn das Fahrzeug still steht (v=0) und die Bremse getreten wird (Brake On).

Position D ist einen hierarchischer Zustand, der einen Subautomaten zur Modellierung der aktiven Fahrstufen. Einfachheit halber wurden hier die Übergangsereignisse und Bedingungen an den Transitionen weggelassen.

2.3.4.4 Aktivitätsdiagramme

Aktivitätsdiagramme stellen eine spezielle Art von Zustandsautomaten dar [RUM98]. Daher weisen die graphischen Notationen von Zustandsdiagrammen und Aktivitätsdiagrammen Gemeinsamkeiten auf. Während bei Zustandsautomaten durch Ereignisse Änderungen in Form von Zustandsübergängen hervorgerufen werden, steht bei Aktivitätsdiagrammen die Modellierung von Berechnungen und Workflows im Vordergrund.

Ein Aktivitätsdiagramm besteht hauptsächlich aus Aktivitäten, Transitionen, Entscheidungspunkten und Gabelungen bzw. Zusammenführungen. Aktivitäten zeigen die Ausführung von Berechnungen oder Workflows. Diese werden in der Regel erst nach Beendung der Aktivität verlassen. Transitionen schalten zur nächsten Aktivität weiter. Über Entscheidungspunkte (Branch) können alternative Folgeaktivitäten modelliert werden. Nebenläufigkeiten werden durch Gabelungen (Fork) und Zusammenführungen (Join) beschrieben.

2.4 Die Modell-zu-Modell-Transformationssprache M²ToS

Die Modell-zu-Modell-Transformationssprache M²ToS wurde im Jahre 2005 in der Arbeit von Reichmann [REI05] entwickelt. Sie ist eine graphisch notierte Sprache, um MOF-Metamodellbasierte Modelle zu transformieren. Wichtige Aspekte dieser Arbeit stützen sich auf M²ToS, so dass diese Sprache im Folgenden näher beschrieben wird.

2.4.1 Aufbau der M²ToS

Die M²ToS basiert auf graphisch notierten Regeln. Alle Regeln einer Transformationsvorschrift bilden ein Regelwerk oder auch Regelmodell genannt. Regeln können aufeinander aufbauen und in Abhängigkeit zueinander stehen. Zur Ausführung der Regelwerke steht eine Regelausführungssemantik mit einer entsprechenden Ausführungsstrategie zur Verfügung. Insgesamt spezifiziert die

M²ToS 69 Regelnotationsdefinitionen in vier Bereichen, die im Folgenden näher erläutert werden.

2.4.1.1 Regelmakrostruktur

Die Regelmakrostruktur beschreibt grundsätzliche Zusammenhänge des Regelmodells, wie sie in Abbildung 2.14 gezeigt ist. Dazu gehören die zu verwendenden Metamodelle und die Transformationsbeschreibung. Die Transformationsbeschreibung besteht aus einem Logikpaket, welches für die

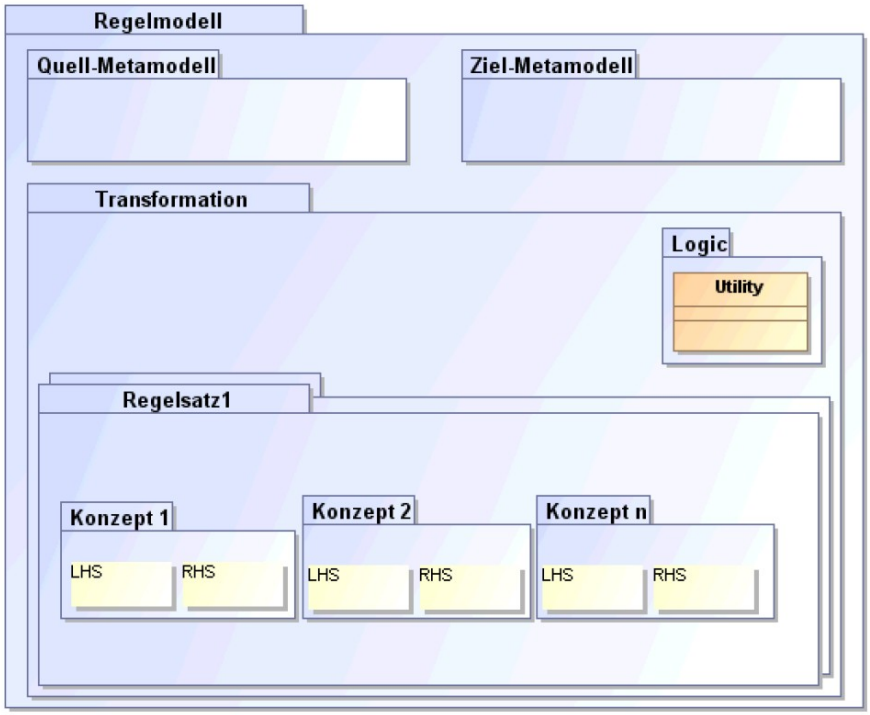


Abbildung 2.14: Regelmakrostruktur der M²ToS

Transformation benötigte Hilfsklassen enthält, und mindestens einem Regelsatz, in denen die Transformationsregeln (Konzepte), durch Regelgruppen hierarchisiert, abgelegt sind.

2.4.1.2 Regelmikrostruktur

Die Regelmikrostruktur spezifiziert den Aufbau der Transformationsregeln. Eine Regel verfügt, wie in Abbildung 2.15 dargestellt, über eine linke (LHS – Left Hand Side) und eine rechte Seite (RHS – Right Hand Side). In der LHS werden Objektmuster als Instanzen des Quellmetamodells definiert, die im Quellmodell der Transformation gesucht werden sollen. Die RHS beschreibt Objektmuster als Instanzen des Zielmetamodells, die ausgehend vom Objektmuster der LHS als Teilmodell des Zielmodells erstellt werden sollen. Transformationsregeln verfügen über eine graphische Notation. Diese lehnt sich an die aus der UML bekannten Objektdiagramme (siehe Abschnitt 2.3.4.2) an.

Der Einstiegspunkt in der LHS bildet das sogenannte *source*-Objekt. Es wird durch ein abgerunde-

tes Rechteck im Regeldiagramm symbolisiert. Die anderen Objekte der LHS spezifizieren das Umfeld des *source*-Objektes und stehen durch Assoziationen mit diesem direkt oder indirekt in Verbindung. Die verwendeten Assoziationen sind durch ihre Rollennamen eindeutig identifiziert. Gesetzte Attributwerte in Objekten der LHS werden als Bedingung interpretiert. Damit wird die Suche auf Objekte in bestimmten Zuständen eingeschränkt. Die Menge bestehend aus Typen der Objekte, der Assoziationen inklusive ihrer Rollen und der Attribute ist durch das dazugehörige Quellmetamodell wohldefiniert.

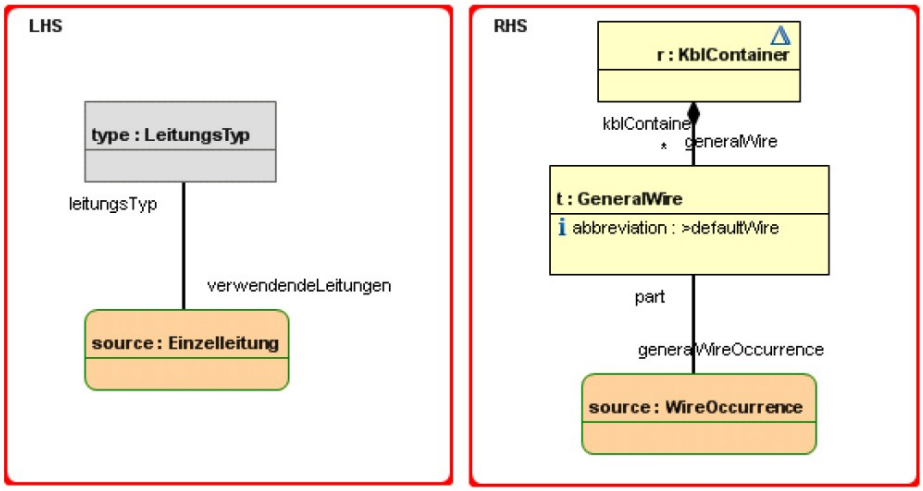


Abbildung 2.15: Beispiel für eine M²ToS Regel

Auf der rechten Seite einer Regel werden Objektmuster definiert, die erstellt werden, sofern das Objektmuster der LHS gefunden wurde. Für jeden Treffer der LHS wird die entsprechende Objektstruktur der RHS aufgebaut und als Teilmodell und Instanz des Zielmetamodells gemäß der später beschriebenen Regelausführungssemantik weiter verwendet.

Objekte der RHS können direkt aus Objekten der LHS heraus entstehen. In diesem Falle handelt es sich um eine sogenannte Quell-Ziel-Beziehung und beide Objekte verfügen über denselben Objektnamen. Im Regeldiagramm werden solche Objekte orange dargestellt. Objekte der RHS können auch additiv durch eine semantische oder syntaktische Notwendigkeit erstellt werden. Diese Objekte werden im Regeldiagramm mit gelbem Hintergrund dargestellt. Attributwerte von Objekten der LHS können innerhalb von Objekten der RHS verwendet werden, indem anhand eines Ausdrucks im Attribut des RHS-Objektes auf das Objektattribut der LHS verwiesen wird. In Sonderfällen sind hier komplexe Rechenoperationen notwendig. Diese können durch eine Implementierung im Logikpaket ausgelagert und vom Attribut des RHS-Objektes aus aufgerufen werden.

2.4.1.3 Regelausführungsstrategie

Regeln können durch Abhängigkeiten miteinander verknüpft sein. Die *HasHit*-Abhängigkeit spezifiziert, dass eine von Regel A abhängige Regel B nur dann angewendet wird, wenn A mindestens einmal einen Treffer im Quellmodell hatte.

Im Gegensatz dazu beschreibt die *NoHit*-Abhängigkeit, dass die von Regel A abhängige Regel B nur dann angewendet wird, wenn die Regel A auf dem Quellmodell keinen Treffer erzielt.

Mit Hilfe dieser Abhängigkeiten können Anwendungsreihenfolgen für Regeln festgelegt und die Transformation spezifischer Strukturen durchgeführt werden.

2.4.1.4 Regelausführungssemantik

Die Regelausführungssemantik unterteilt insgesamt vier Ausführungsschritte, der Mustersuche, Erzeugung, Objektzuordnung und Verschmelzung. Die Mustersuche verwendet die LHS der Regeln und sucht entsprechende Strukturen im Quellmodell. Jede gefundene Struktur wird als Treffer markiert und den nachfolgenden Ausführungsschritten zur Verfügung gestellt. Die Erzeugung baut für jeden Treffer die in der RHS spezifizierten Objektstrukturen zu einem Teilmodell auf, diese werden durch die Objektzuordnung mit den entsprechenden Quellobjekten der LHS verknüpft. Im Ausführungsschritt Verschmelzung werden die erstellten Teilmodelle zum Zielmodell zusammengefasst. Für die Verschmelzung gelten insgesamt vier Kriterien:

- Quell-Ziel-Beziehung: Zwischen einem aus der RHS erzeugten Objekt, welches direkt aus einem Objekt der LHS hervorgegangen ist, besteht eine Quell-Ziel-Beziehung. Innerhalb einer Regel besitzen derartige Objekte auf der LHS und der RHS denselben Objektnamen.
- Wurzelknoten: Innerhalb eines Modells gibt es einen Wurzelknoten, über den alle Artefakte des Modells direkt oder indirekt über die Kompositionen erreicht werden können. Alle Artefakte, die aus einem in der RHS als „root“-markierten Objekts (blaues Dreieck im Objekt *r* in Abbildung 2.15) erzeugt wurden, werden zu einem Artefakt verschmolzen. Dieses ist der Wurzelknoten des Zielmodells.
- * zu 1-Beziehungen: Im Metamodell werden Beziehungen zwischen Metaklassen durch Assoziationen oder Kompositionen modelliert. Diese werden mit Multiplizitäten versehen. Handelt es sich bei einer Relation um eine * zu 1-Beziehung, so werden die erzeugten Objekte am 1-wertigen Ende der Assoziation zu einem Objekt verschmolzen, da sie identisch sein müssen. Durch dieses Kriterium wird die syntaktische Korrektheit des Zielmodell sichergestellt.

- Explizite Identität: Sollen aus einer RHS konstruierte Objekte aus semantischen Gründen zu einem Artefakt verschmolzen werden, so wird auf dem entsprechenden Objekt der RHS ein ID-Tag gesetzt. Erzeugte Artefakte mit derselben Identität werden miteinander verschmolzen.

Werden Artefakte durch eines der vier Kriterien verschmolzen, so werden auch deren Links miteinander verschmolzen haben, sofern sie zu den identischen benachbarten Objekten laufen und gleiche Rollennamen haben.

2.4.2 Aufbau des Modell-zu-Modell Transformators

Die Transformation eines Quellmodells mit Hilfe eines in M²ToS-notierten Regelwerks in ein Zielmodell erfolgt durch einen Transformator (siehe Abbildung 2.16). Dieser besteht im Wesentlichen aus drei Teilen, dem *Matcher*, dem *Mapper* und dem *Merger*. Diese drei Teile spiegeln sich in zum Teil entsprechenden Konzepten der M²ToS-Ausführungsschritte wieder (siehe Abschnitt 2.4.1.4).

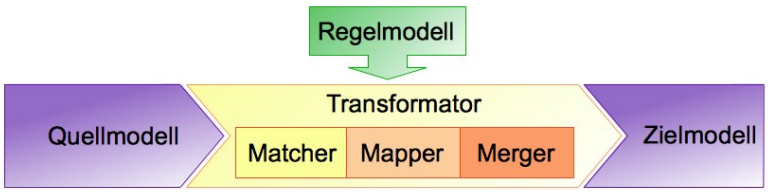


Abbildung 2.16: Aufbau Modell-zu-Modell Transformator

Der Matcher korrespondiert mit dem Ausführungsschritt Mustersuche. Der Mapper vereinigt die Ausführungsschritte Erzeugung und Objektzuordnung. Der Merger ist eine Implementierung des Ausführungsschrittes Verschmelzung.

2.4.2.1 Spezialfälle einer M²ToS-basierten Transformation

Die Modell-zu-Modell Transformation durch M²ToS unterstützt einige Spezialfälle, die in diesem Abschnitt näher beschrieben werden.

- Modellabfrage:

  Wird in einer Regel lediglich die LHS spezifiziert, so handelt es sich um eine Modellabfrage. In diesem Fall wird im Transformator nur der Matcher zur Suche der in der LHS spezifizierten Objektstrukur ausgeführt. Das Ergebnis ist in diesem Fall kein Zielmodell sondern die Menge der durch die LHS gefundenen Artefakte. Modellabfragen können entweder auf das komplette Zielmodell angewendet werden oder auf vorgegebene Teile davon.
  Modellabfragen werden verwendet, um komplexe Objektstrukturen in einem Modell zu finden.

Die Interpretation der Ergebnisse der Modellabfrage hängt von dem jeweiligen Anwendungsfall ab.

- Lokale Transformation:

  Ist das Quellmodell identisch Mit dem Zielmodell, handelt es sich um eine lokale Transformation. Dies impliziert, dass die Quell- und Zielmetamodelle ebenfalls identisch sein müssen. In diesem Fall werden die durch die RHS spezifizierten Objektstrukturen auf Quellmodell angewendet und damit das Quellmodell modifiziert. Eine lokale Transformation kann entweder auf das gesamte Quellmodell angewendet werden oder nur auf bestimmte, vorgegebene Modellteile. Das Ergebnis einer lokalen Transformation ist das modifizierte Quellmodell.
  Ein typischer Anwendungsfall für lokale Transformationen sind Modellrefactorings, bei denen Teile eines Modells den Vorgaben entsprechend modifiziert werden.

2.5 Entwurfsmuster

Entwurfsmuster (Def. 26) sind Lösungsansätze für wiederkehrende Problemstellungen in der Softwarearchitektur. Sie werden als Vorlage zur Lösung des Problems in Abhängigkeit des jeweiligen spezifischen Kontextes verwendet.

Entwurfsmuster stammen ursprünglich aus der Architektur im Bauwesen und wurden von Christopher Alexander [ALE77] in den 1970er Jahren vorgestellt. Es folgten verschiedene Ansätze, dieses Thema unter anderem auf die Softwaretechnik zu übertragen. Der Durchbruch gelang schließlich Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides („Gang of Four“, GoF) mit dem Buch „Design Patterns – Elements of Reusable Object-Oriented Software“ aus dem Jahr 1994 [GOF95]. Im Laufe der Zeit wurden diverse Entwurfsmuster in unterschiedlichen Granularitäten und Problemfeldern vorgestellt.

Im Wesentlichen lassen sich drei unterschiedliche Klassen von Entwurfsmustern bilden. Die *erzeugenden Muster* befassen sich mit der effizienten Erzeugung von Objekten und Objektstrukturen. Die *strukturellen Muster* bieten Lösungsvorschläge zur Abbildung und Verwaltung von komplexen Datenstrukturen. Die *Verhaltensmuster* beinhalten Schablonen zur Lösung komplexer Verhaltensweisen. Zudem gibt es eine Fülle weiterer Muster, die sich nicht direkt in diese Klassifizierung einteilen lassen. Dazu gehört zum Beispiel das in Abschnitt 2.5.6 beschriebene *Model-View-Presenter*-Muster. Dieses Muster ist nicht als Lösungsvorlage für eine konkrete Datenstruktur konzipiert, sondern als prinzipielle Architekturstrategie.

Im Folgenden werden die für diese Arbeit relevanten Entwurfsmuster vorgestellt. Vor allem die

strukturellen Muster [GOF95] sind von großer Bedeutung für die Entwicklung komplexer Datenstrukturen in Form von Metamodellen.

2.5.1 Kompositum

Das Kompositum-Entwurfsmuster [GOF95] spezifiziert, wie streng hierarchische Teil-Ganzes-Strukturen aufgebaut werden. Mit diesem Entwurfsmuster lassen sich streng hierarchische Objektstrukturen, wie sie zum Beispiel beim UML-Paketkonzept [UML241Inf] vorkommen, abbilden.

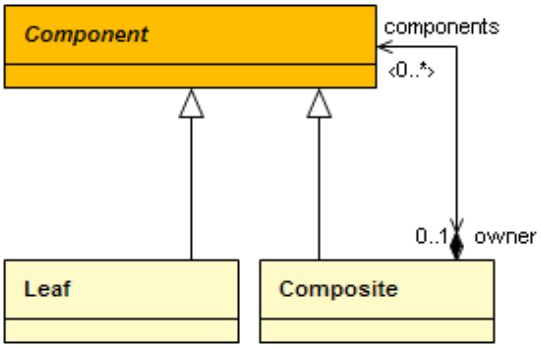


Abbildung 2.17: Das Kompositum-Entwurfsmuster

Wie in Abbildung 2.17 dargestellt, ist das Kernstück dieses Entwurfsmusters ist eine gemeinsame Superklasse *Component*, die über eine erbende *Composite*-Klasse aggregiert werden kann. Jede Instanz dieser Klasse definiert eine eigene Schicht in der Baumhierarchie. Ist die Assoziation zu einem aggregierenden *Composite* frei, so handelt es sich um das Wurzelelement des Baums. Beinhaltete *Component*-Instanzen bilden den Inhalt dieser Hierarchieebene ab. Blätter des Baums werden durch die Klasse *Leaf* abgebildet, die wiederum von *Component* erbt und somit Bestandteil eines *Composite* sein kann.

2.5.2 Typ-Instanz

Die Bildung von Typen und Instanzen ist ein in der Softwaretechnik gängiger Mechanismus, welcher die explizite Trennung von Typ und Instanz erlaubt. Sie wird häufig in der Metamodellierung verwendet und gehört daher nicht zu den klassischen Entwurfsmustern, die sich hauptsächlich mit Problemstellungen bei der Softwareentwicklung befassen. Wegen seiner Bedeutung für die Metamodellierung wird dieses Konzept innerhalb der Arbeit als Entwurfsmuster aufgefasst.

Durch die Deklaration von Typen, die durch mehrfache Instanziierung wiederverwendet werden, können instanzübergreifende Eigenschaften im Typ abgelegt werden. Jeder Instanz ist dabei ein

Typ zugeordnet, von dem es beliebig viele Instanzen geben kann. Werden Eigenschaften des Typ-Artefakts verändert, überträgt sich dies indirekt (d.h. semantisch) auf alle Instanzen des Typs.

Abbildung 2.18: Das Typ-Instanz-Entwurfsmuster

In Abbildung 2.18 wird das Typ-Instanz-Entwurfsmuster dargestellt. Die Instanz-übergreifenden Eigenschaften *typeProperty1* und *typeProperty2* werden der Typ-Klasse *Type* zugeordnet. Die Instanz-spezifischen Eigenschaften *instanceProperty1* und *instanceProperty2* sind der Instanz-Klasse *Instance* zugeordnet.

2.5.3 Prototypen

Dieses Entwurfsmuster gehört zur Klasse der *erzeugenden Muster* [GOF95]. Eine prototypische Instanz dient als Vorlage für daraus erstellte Instanzen. Diese können bei Bedarf an den jeweiligen Kontext angepasst werden. Dazu verfügt der Prototyp über eine Methode, die den Prototypen kopieren bzw. klonen kann.

Grundsätzlich wird bei der Kopie unterschieden in eine „flache Kopie“ (engl. s*hallow copy*) und eine „tiefe Kopie“ (engl. *deep copy*). Im ersten Fall wird nur eine Kopie des Objektes und seiner Eigenschaften angefertigt. Im zweiten Fall wird die Umgebung des Objektes, die sich aus Assoziationen zu anderen Objekten ergibt, beachtet und gegebenenfalls ebenfalls kopiert. Die Entscheidung, welche Assoziationen mit kopiert werden, muss im jeweiligen Anwendungsfall vorgenommen werden.

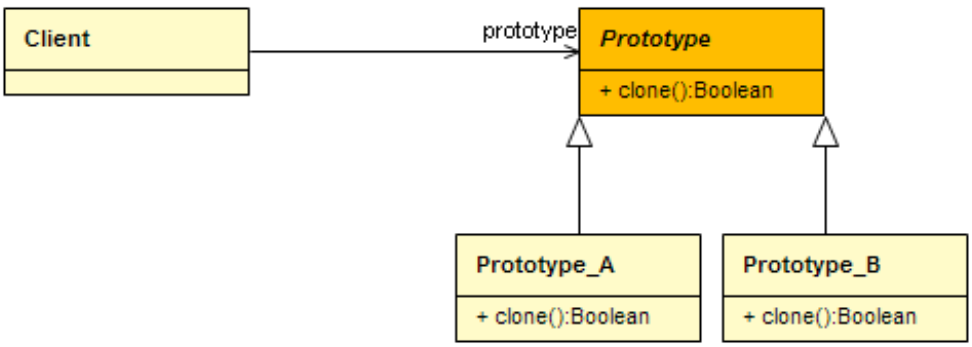


Abbildung 2.19: Das Prototyp-Entwurfsmuster

In Abbildung 2.19 ist das Prototyp-Entwurfsmuster abgebildet. Ein *Client* kann über die *clone()*-Methode Kopien des Prototypen anfordern. Je nach Implementierung des Prototypen in den Subklassen *Prototype_A* oder *Prototype_B* wird eine entsprechende Kopie angefertigt.

2.5.4 Mapping

Mithilfe des Mapping-Entwurfsmusters können Artefakte einer Ebene explizit durch Einsatz des Mapping-Artefakts auf Artefakte einer anderen Ebene abgebildet werden.

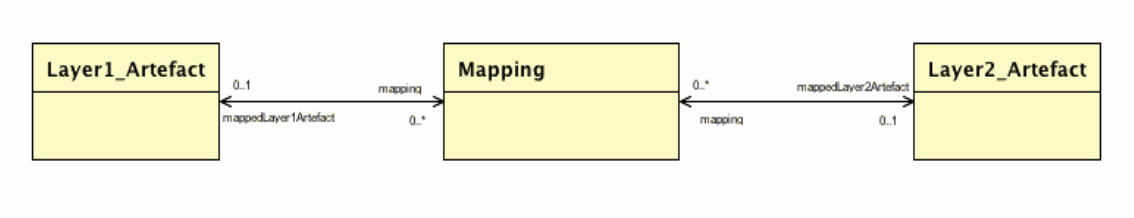


Abbildung 2.20: Allgemeines Mapping-Konzept

Wie in Abbildung 2.20 dargestellt, wird ein Artefakt einer Ebene 1 (*Layer1_Artefact*) einem Artefakt einer Ebene 2 (*Layer2_Artefact*) zugeordnet („gemappt“). Dafür kommt ein eigenständiges Artefakt vom Typ *Mapping* zum Zuge. Über die dargestellten Multiplizitäten zur Mapping-Klasse können viele dieser Beziehungen realisiert werden, wobei jedes Mapping nur auf ein Artefakt einer Ebene referenziert. Im Einzelfall können die hier vorgestellten Multiplizitäten abweichend spezifiziert sein, ferner kann ein Mapping-Artefakt durch eine entsprechende Attributierung in seiner Semantik erweitert werden.

Die resultierende Semantik eines Mappings bezieht sich auf den konkreten Anwendungsfall. Deshalb werden die konkreten ebenenübergreifenden Zusammenhänge in den jeweiligen Abschnitten (siehe 3.3.7) für die verschiedenen Abstraktionsebenen detailliert vorgestellt.

2.5.5 Port-Konzept

Objekte werden durch Relationen miteinander verknüpft. Damit wird ausgedrückt, dass die beteiligten Objekte in einer bestimmten Form, die durch die Art der Assoziation (Komposition, Aggregation, Assoziation) spezifiziert ist, in Beziehung stehen. Es gibt jedoch keine formale Beschreibung darüber, welche Inhalte durch diese Beziehung ausgetauscht werden.

Das Port-Konzept, wie in Abbildung 2.21 dargestellt, ermöglicht durch Einsatz von dedizierten Port-Objekten an den Relationsenden die konkrete Modellierung der Schnittstellen. Der Port fungiert als Vermittler zwischen den an der Relation beteiligten Elementen (*Component*) und der Rela-

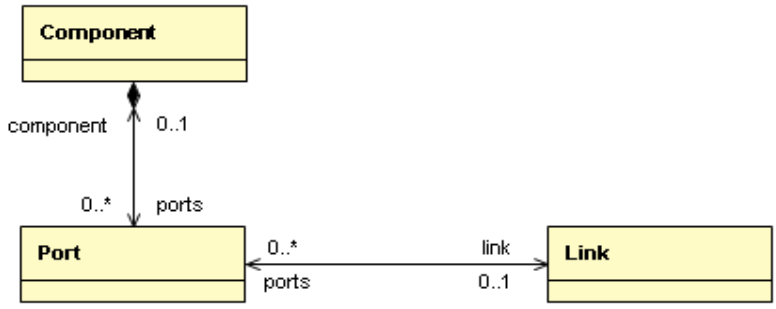


Abbildung 2.21: Port-Konzept

tion selbst, die selbst wieder durch ein eigenständiges Objekt gekapselt sein kann (*Link*). Das Portobjekt kann Teil anderer Konzepte sein, zum Beispiel kann es durch das in Abschnitt 2.5.2 beschriebenen Typ-Instanz-Konzeptes weiter typisiert werden.

Das Port-Konzept eignet sich zum Beispiel vor allem zur Modellierung von Kommunikationsstrecken wie Steuergeräte, die durch Bussysteme vernetzt sind. Die Ports sind in diesem Fall die Anbindungen, über welche die Datenübertragung zwischen Steuergerät und Bussystem realisiert ist.

2.5.6 Model-View-Controller (MVC)

Das Model-View-Controller-Muster MVC [GOF95] ist ein Architekturmuster, bei dem das Zusammenspiel von komplexen Daten (*Model*) und seiner Darstellung (*View*) durch einen Präsentator (*Presenter*) koordiniert wird.

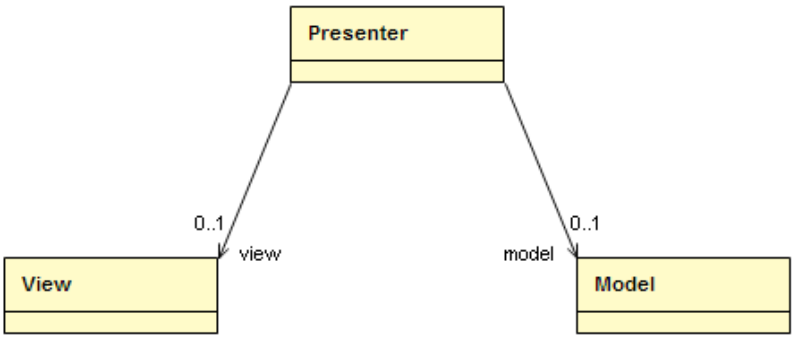


Abbildung 2.22: Das Model-View-Controller-Architekturmuster

In Abbildung 2.22 wird das Model-View-Controller-Architekturmuster dargestellt. Eine Komponente wird durch eine Klasse repräsentiert. Weder Modell, noch Ansicht kennen den Präsentator oder die jeweilige andere Komponente. Dreh- und Angelpunkt ist der Präsentator, welcher das Zusammenspiel der verschiedenen Komponenten koordiniert.

Eine Weiterentwicklung ist das Model-View-Presenter-Entwurfsmuster [POT96], schreibt jedoch eine deutlichere Trennung zwischen den Komponenten vor. Dadurch wird eine bessere Testbarkeit erreicht.

2.6 Petri-Netze

Carl Adam Petri veröffentlichte 1962 die Grundlagen der nach ihm benannten Petri-Netze [PET62]. Dabei handelt es sich um ein mathematisches Modell zur Berechnung und Darstellung asynchroner Systeme. Es eignet sich zum Beispiel zur Modellierung von Produktionsstraßen, Firmenorganisationen, Kommunikationsprotokollen oder Gerätesteuerungen.

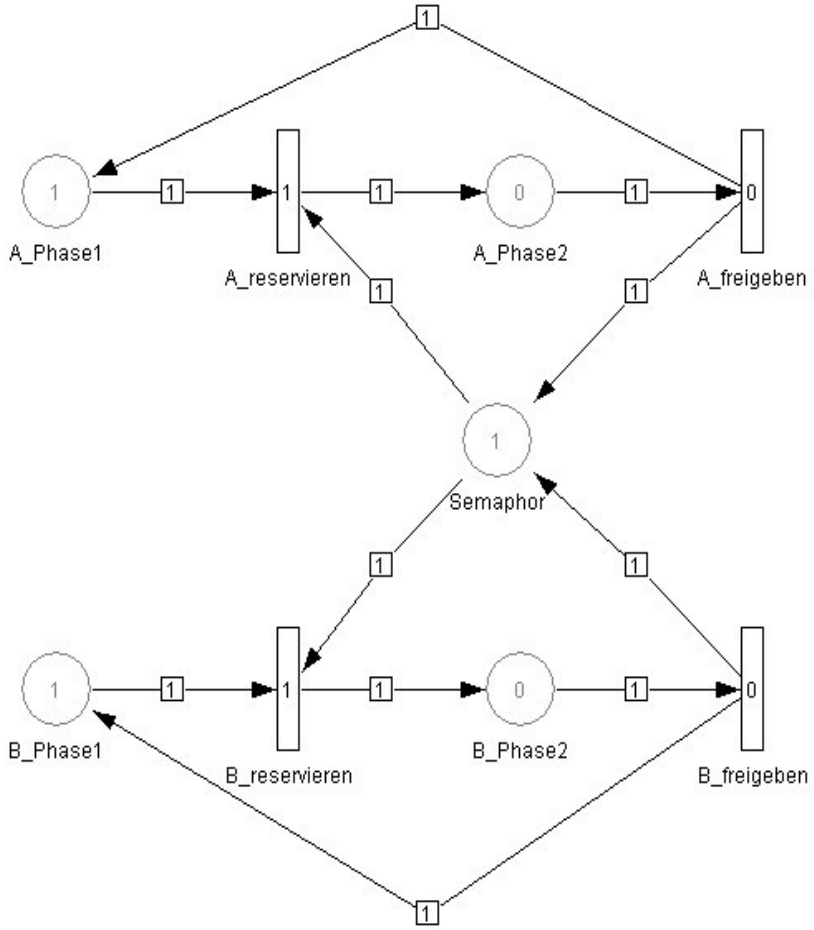


Abbildung 2.23: Petri-Netz für exklusive Zugriffssteuerung über eine Semaphore [MAR99]

Ein Petri-Netz ist ein bipartiter, gerichteter Graph. Er besteht aus Stellen und Transitionen. Stellen und Transitionen wechseln sich im Graphen immer ab, das heißt es kann keine zwei aufeinanderfolgende Stellen oder Übergänge geben. Abbildung 2.23 zeigt die graphische Notation von Petri-Netzen. Darin werden Stellen als Kreise und Transitionen als Rechtecke dargestellt. Deren Verbindun-

gen werden durch Pfeile mit einer Wertigkeit (meist „1“, in Abbildung 2.23 als kleines Rechteck auf einem Pfeil dargestellt) repräsentiert.

Stellen haben eine Kapazität, um Marken aufzunehmen. Sind keine Kapazitäten angegeben, beträgt die Kapazität im Allgemeinen 1. In Abbildung 2.23 wird die Kapazität der Stellen nicht dargestellt, sie beträgt für alle Stellen „1“. Die Zahl in den Stellen repräsentiert die Anzahl der aufgenommenen Marken. Transitionen werden mit Gewichten versehen. Diese repräsentieren die Kosten der Kante im Graphen. Die Belegung der Stellen im Netz bildet den Zustand des Petri-Netzes ab. Befinden sich an den Eingangsstellen einer Transition mindestens so viele Markierungen wie das Gewicht der Transition ist, so ist diese aktiviert, beziehungsweise schaltbereit. Diese können zu beliebigen Zeitpunkten schalten, sofern die Kapazität der nachfolgenden Stellen ausreicht, um die Anzahl der Marken aufzunehmen.

Abbildung 2.23 zeigt ein Petri-Netz für eine exklusive Zugriffssteuerung über eine Semaphore. Dazu gibt es zwei parallel laufende Prozesse mit jeweils zwei Arbeitsphasen, von denen sich jedoch die beiden Phasen eine Ressource teilen müssen. Die Stellen *A_Phase1*, *B_Phase1* und *Semaphor* verfügen über einen Marker. In dieser Konstellation können die Transitionen *A_reservieren* beziehungsweise *B_reservieren* schalten. Je nachdem welcher der beiden laufenden, unabhängigen Arbeitsphasen von *A* und *B* schneller terminiert wird für diesen Prozess die *Semaphor* in Anspruch genommen, das heißt *A_Phase2* beziehungsweise *B_Phase2* erhält einen Marker, damit ist für den anderen Prozess die zweite Phase gesperrt, da die Semaphor nicht reserviert werden kann.

2.7 Java

Java [JAVA] ist eine objektorientierte Programmiersprache und ist Bestandteil der Java-Technologie, welche neben der Programmiersprache auch die Laufzeitumgebung für Computer spezifiziert. Die Firma Sun Microsystems, Ende Januar 2010 von Oracle übernommen, veröffentlichte 1995 die erste Alphaversion von Java.

2.7.1 Technologie

Die Java-Technologie spezifiziert zum einen die Programmiersprache Java sowie verschiedene Laufzeitumgebungen, auf denen in Java geschriebene Computerprogramme ausgeführt werden können. Eine Laufzeitumgebung ist abhängig vom Betriebssystem, Java-Programme hingegen nicht. Dadurch ist eine Plattformunabhängigkeit gegeben.

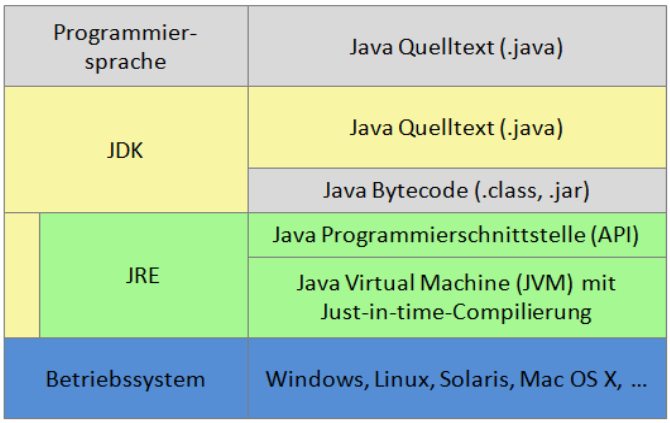


Abbildung 2.24: Java-Technologie (in Anlehnung an [WIJT11])

In Abbildung 2.24 zeigt eine Übersicht der Java-Technologie. Basierend auf dem Betriebssystem gibt es passende Laufzeitumgebungen (JRE – Java Runtime Environment), welche hauptsächlich eine virtuelle Maschine (JVM – Java Virtual Machine) sowie eine passende Programmierschnittstelle zur Ausführung von Java-Programmen zur Verfügung stellt.

Auf der Laufzeitumgebung setzt das Java Entwicklungskit (JDK – Java Development Kit) auf. Diese wird nur für die Entwicklung von Java-Programmen benötigt.

2.7.2 Debugging

Java-Programme können während der Ausführungszeit zu Fehlerlokalisation und Analysezwecken einem Debugging unterzogen werden. Dazu stehen entsprechende Debug-Schnittstellen, die in Abbildung 2.25 dargestellt sind, zur Verfügung.

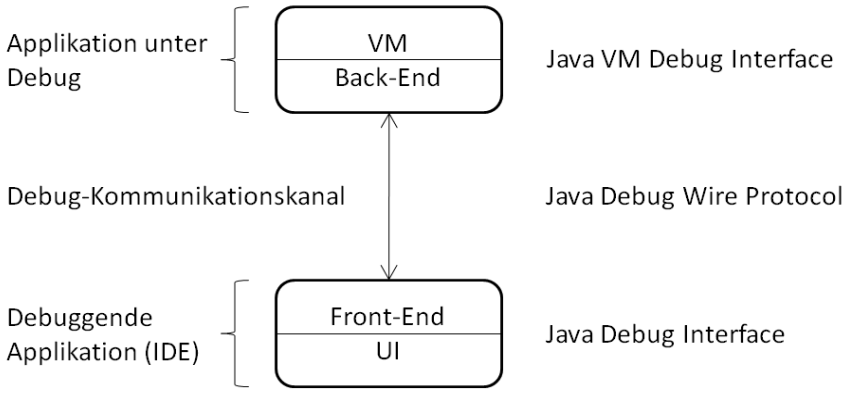


Abbildung 2.25: Debugging von Java-Programmen

Über einen Debug-Kommunikationskanal wird zu debuggende Applikation mit der debuggenden Applikation (meist eine Java-Entwicklungsumgebung) verbunden. Beide Applikationen können auf

derselben Hardware oder verteilt über ein Netzwerk verbunden laufen.

Über den Debug-Kommunikationskanal werden die Debug-Steuerbefehle von der debuggenden Applikation an das zu debuggende Programm gesendet und der Programmablauf dort von der virtuellen Maschine entsprechend umgesetzt.

2.8 Eclipse

Im November 2001 gründete die Firma IBM zusammen mit sieben anderen Firmen, darunter zum Beispiel Borland, RedHat und Suse, die Eclipse Foundation [ECLI]. Eclipse, zuvor von IBM als kommerzielles Tool entwickelt, wurde damit ein Open-Source-Projekt. In den folgenden Jahren wurde Eclipse stark weiterentwickelt und mittlerweile ist es eine Plattform für viele Werkzeuge und Anwendungen. Die Eclipse-Plattform kann durch Erweiterungen an die jeweiligen Einsatzgebiete angepasst werden. So existieren neben der weitverbreiteten Java IDE noch Erweiterungen für C/C++, Cobol, Python und andere Programmiersprachen beziehungsweise Einsatzgebiete. Dieser Erfolg ist vor Allem folgenden Entwicklungs-Paradigmen der Eclipse-Plattform zu verdanken:

- Einfache Erweiterbarkeit auf Basis eines stabilen Kerns
- Weitgehende Plattformunabhängigkeit durch Einsatz von Java
- Natives Look & Feel der entsprechenden graphischen Oberfläche durch die direkte Verwendung der jeweiligen plattformspezifischen Graphik-API

2.8.1 Plugin-Konzept

Eines der wichtigsten Konzepte der Eclipse-Plattform ist die Verwendung von wiederverwendbaren Plugins, welche seit der Eclipse-Version 3.0 OSGi-Bundles [OSGi] sind. Ein Plugin dient zur Kapselung einer bestimmten Funktionalität. Über Erweiterungspunkte (Extension Point respektive OSGi-Service) spezifiziert ein Plugin Schnittstellen für erweiterbare Funktionalitäten anderer Plugins.

In Abbildung 2.26 werden die wesentlichen Komponenten des Standard-Eclipse gezeigt. Im Inneren sind die Kernkomponenten Equinox (Eclipse-Implementierung des OSGi-Standards) und Update (Framework für Softwareupdate-Funktionalität). Diese Kernkomponenten bilden die Basisplattform, die zunächst ohne graphische Benutzerschnittstelle (GUI) auskommt, welche durch die Bundles Runtime, UI, SWT und JFace zur Verfügung gestellt wird. Die IDE-Funktionalität (IDE – Integrated Development Environment) wird durch die nächst äußeren Komponenten hinzugefügt. Die

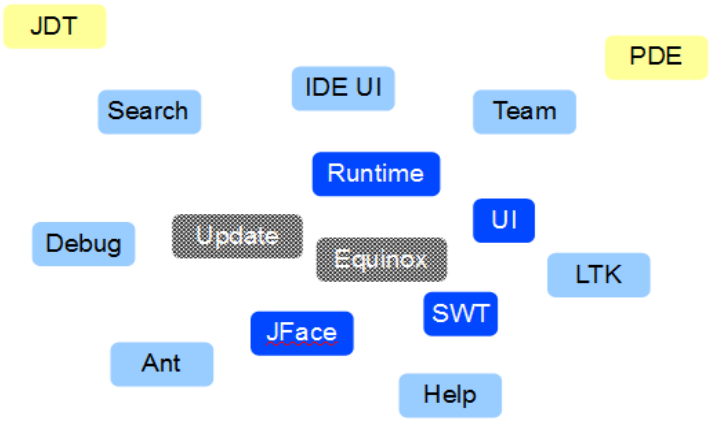


Abbildung 2.26: Wesentliche Eclipse-Komponenten

Spezialisierung für die Java-Entwicklungsumgebung erfolgen durch JDT (Java Development Tools) und PDE (Plug-in Development Environment), welche standardmäßig in Eclipse integriert sind.

2.8.2 Das Graphical Editing Framework

Außer der eigentlichen Plattform existieren weitere Projekte unter dem Dach der Eclipse Foundation. Zur Gruppe der Eclipse Tools-Projects gehört unter anderem das Graphical Editor Framework, oder kurz GEF [GEF]. Dieses ermöglicht dem Entwickler, relativ einfach graphische Diagrammeditoren zu entwickeln. GEF ist dabei eine Umsetzung des in Kapitel 2.5.6 vorgestellten Model-View-Controller Entwurfsmusters, seine Architektur ist in Abbildung 2.27 dargestellt.

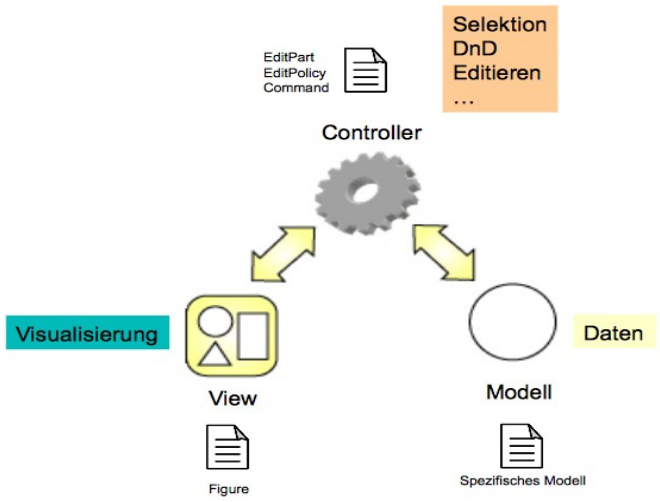


Abbildung 2.27: Architektur von GEF

Die Modellschicht rechts unten, welche die darzustellenden und zu manipulierenden Objekte beinhaltet, wird vom Entwickler spezifiziert. GEF bietet dort keine Vorlagen, macht allerdings auch keine Vorgaben. Somit können bereits existierende Modelle als Basis des Editors verwendet werden.

Die Modellschicht beinhaltet alle relevanten Daten.

Die Controller im oberen Teil der Abbildung können in GEF aus mehreren Objekten bestehen. Die Hauptkomponente der Controller-Funktionalität bilden sogenannte *EditParts*, sie sind für die Synchronisation zwischen Modell und Visualisierung verantwortlich. Die durch Benutzerinteraktion hervorgerufenen Änderungswünsche werden durch *EditPolicy*-Objekte aufgefangen und mit Hilfe von *Commands* auf das Modell appliziert. Für unterschiedliche Arten von Änderungen wie Verschieben oder Löschen sind separate *Policies* zuständig. Durch diese Modularisierung wird die Wiederverwendbarkeit deutlich erhöht. Für diese Objekte der Controller-Ebene existieren Interfaces und Standardimplementierungen. Durch Spezialisierung können Sie auf eigene Bedürfnisse angepasst werden.

Auf der View-Ebene (im rechten unteren Teil der Abbildung 2.27)verwendet GEF das mitgelieferte Draw2D-Plugin. Dieses basiert auf SWT [SWT] und stellt ein graphisches Framework für die Visualisierung zur Verfügung. Von der abstrakten Klasse *Figure* ist für jede Darstellung eine Subklasse zu erstellen und das entsprechende Erscheinungsbild in der entsprechenden Methode zu implementieren. Zusätzlich existieren Layouter, welche die Positionierung von Kindobjekten im Elternobjekt vornehmen.

Als weitere Möglichkeit bietet GEF die Definition einer Werkzeugpalette für Editoren. Durch die enge Verzahnung mit Eclipse stehen zusätzlich Drag & Drop auf der Oberfläche, der werkzeugweite Selektionsmechanismus sowie andere zentrale Funktionalitäten zur Verfügung.

2.9 PREEvision

Die EEA-ADL (Electric Electronic Analysis Architecture Description Language) ist ein MOF-basiertes Metamodell zur Beschreibung von Elektrik/Elektronik-Architekturen im Fahrzeug. Dieses Metamodell bildet die Grundlage für das Bewertungs- und Modellierungswerkzeug PREEvision [PREE], es wurde parallel zu dieser Anwendung entwickelt.

PREEvision geht auf ein gemeinsames Forschungsprojekt zwischen DaimerChrysler (heute Daimler) [DAI] und dem Forschungszentrum Informatik FZI [FZI] mit Unterstützung des Instituts für Technik der Informationsverarbeitung ITIV an der Universität Karlsruhe (TH) [ITIV] im Jahre 2003 zurück. Darin ging es um die Konzeption von Elektrik/Elektronik-Architekturen in der frühen Entwurfsphase. Als prototypisches Projektergebnis entstand das EEKonzeptTool (vgl. Abschnitt 4.4.3), einem frühen Vorläufer von PREEvision. Im Jahre 2005 wurde das Spin-Off Unternehmen

aquintos GmbH gegründet, mit dem Ziel, die Projektergebnisse weiterzuentwickeln und zur Marktreife zu bringen. 2010 wurde aquintos Teil der Vector-Gruppe, PREEvision wird seitdem gemeinsam weiterentwickelt. PREEvision bildet den programmatischen Rahmen, in welchem die Implementierung des in dieser Dissertation behandelten Metrik-Frameworks erfolgte.

2.9.1 EEA-ADL

Die EEA-ADL (Electric Electronic Analysis Architecture Description Language) [PREE] ist ein MOF-basiertes Metamodell (siehe Abschnitt 2.1.2) zur Beschreibung von Elektrik/Elektronik-Architekturen im Fahrzeug. Dieses Metamodell bildet die Grundlage für PREEvision.

Die EEA-ADL implementiert das in Kapitel 3.1 beschriebene Abstraktionsebenen-Modell für E/E-Architekturen. Orthogonal zu diesen semantischen Abstraktionsebenen ist das MOF-konforme Metamodell hierarchisch durch Verwendung von MOF-Paketen (vgl. Abschnitt 3.2.2.3) unterteilt. Diese Hierarchisierung orientiert sich, soweit möglich, an das Abstraktionsebenen-Modell.

Aufgrund ihres Umfangs und detaillierten Beschreibungstiefe wird die EEA-ADL als zentrale Datenstruktur für Bewertungen und Vergleiche in dieser Arbeit verwendet. Daher wird sie in Kapitel 3.3 ausführlich beschrieben.

2.9.2 Aufbau von PREEvision

Bei PREEvision handelt es sich um eine Eclipse-basierte RCP-Anwendung [RCP11] (siehe auch Kapitel 2.8) und ist daher, wie Eclipse auch, in Java implementiert. Die EEA-ADL bildet als Datenstruktur den Kern von PREEvision. Darüber hinaus stehen unterschiedliche Im- und Exporter zu anderen Formaten und Beschreibungssprachen zur Verfügung.

Es stehen eine Vielzahl von Ansichten, Dialogen, Formularen und graphische Editoren zur Verfügung, um E/E-Architekturmodelle darzustellen, zu analysieren und weiterzuentwickeln. Die Kollaborationsplattform von PREEvision ermöglicht die Zusammenarbeit ganzer Teams und Beteiligten in unterschiedlichen Rollen.

2.9.3 Das Generic Diagram Framework (GDF)

Das Generic Diagram Framework basiert auf dem in Abschnitt 2.8.2 vorgestellten Graphical Editing Framework von Eclipse sowie auf einigen Konzepten für modellbasierte Symboldefinitionen, die in der Diplomarbeit [QUA05], welche im Rahmen dieser Dissertation betreut wurde, näher

beschrieben sind. Das GDF realisiert Diagrammeditoren für Modelle mit DI-konformer Repräsentation der graphischen Modellteile. Diese Spezialisierung auf das graphische Modell ermöglicht

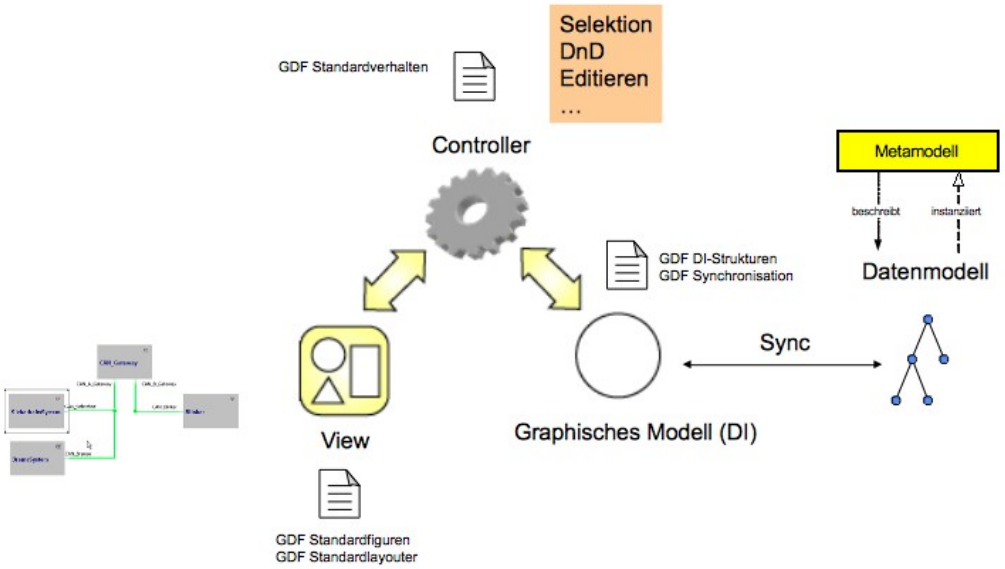


Abbildung 2.28: Aufbau des Generic Diagram Framework GDF

eine weitgehende generische, d.h. Domänen-unabhängige Programmierung weiter Teile eines Editors. Domänen-abhängige, konkrete Editoren werden durch Spezialisierung sowie Konfiguration des GDF erreicht.

Wie in Abbildung 2.28 dargestellt, basiert GDF auf in Abbildung 2.27 gezeigten GEF und somit auf dem MVC-Entwurfsmuster (siehe 2.5.6). Während GEF keine Vorgaben bezüglich des zugrunde liegende Datenmodells macht, setzt GDF genaugenommen zwei Modelle voraus. Das DI-konforme graphische Modell für alle graphischen Informationen wie Koordinaten und Aussehen von Figuren, Beschriftungen und Linien. Das zweite Modell ist das Datenmodell und beinhaltet alle domänenspezifischen Informationen, die letztlich durch den GDF-basierten Editor visualisiert werden. Um die Datenkonsistenz zu gewährleisten sind DI-Modell und domänenspezifisches Datenmodell in einem Modell zusammengefasst.

Die Verknüpfung der graphischen Informationen mit den fachlichen Artefakten erfolgt über eine Assoziation zwischen *AbstractDIRepresentation*, von welcher alle Artefakte des DI-Modells erben,und *DIRepresentableArtefact*, von der alle Artefakte des domänenspezifischen Modells erben. Dieser Zusammenhang ist in Abbildung 2.29 gezeigt.

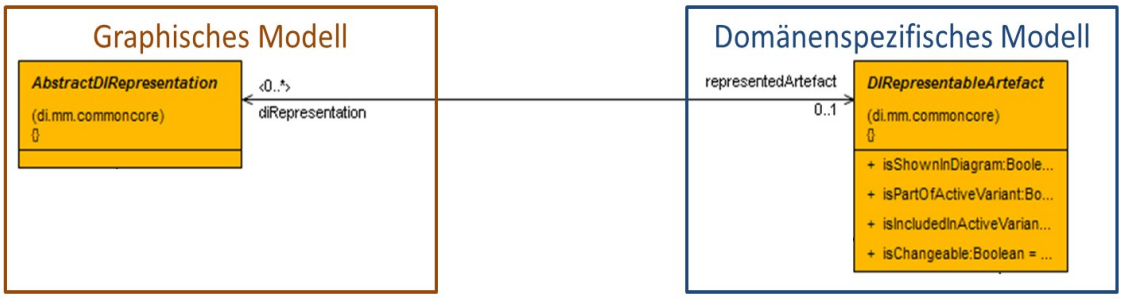


Abbildung 2.29: Verknüpfung zwischen graphischem und domänenspezifischen Modell

Diese klare Trennung zwischen graphischem und Datenmodell entkoppelt die Information von seiner Repräsentation bereits durch die Organisation des Modells. Dieselbe Information kann in unterschiedlichen semantischen Kontexten mit Hilfe von entsprechenden Diagrammeditoren unterschiedlich dargestellt werden. Das GDF bildet die programmatische Grundlage für den in dieser Arbeit entstandenen Metrikeditor (siehe Kapitel 6).

3 Modellbasierte Beschreibung und Visualisierung von E/E-Architekturen

3.1 E/E-Architekturen als verteilte eingebettete Systeme

Als eingebettetes System versteht man Komponenten bestehend aus Software- und Hardwareanteilen, die aufeinander abgestimmt spezifische Aufgaben übernehmen. Zutreffend erweist sich folgende Definition [BRO98] (Def. 21):

> *Ein eingebettetes System [...] ist eine Software-/Hardware-Einheit, die über Sensoren und Aktuatoren mit einem Gesamtsystem verbunden ist und darin Überwachungs-, Steuerungs- beziehungsweise Regelungsaufgaben übernimmt. In der Regel handelt es sich bei eingebetteten Systemen um reaktive, häufig auch um hybride verteilte Systeme mit Echtzeitanforderungen. Typischerweise sind solche Systeme dem menschlichen Benutzer nicht direkt sichtbar, er interagiert unbewusst mit dem eingebetteten System.*

Beispiele für eingebettete Systeme finden sich in vielen Gütern des alltäglichen Lebens, wie zum, Beispiel Unterhaltungselektronik, Waschmaschinen, Mikrowellen, Heizungsanlagen, etc.

In Kraftfahrzeugen sind eingebettete Systeme sind zum Beispiel ein Automatikgetriebe, bei dem die Steuerungselektronik im Getriebe integriert ist oder das Anti-Blockier-System, bei welchem die elektronische Steuerung der einzelnen Bremskanäle und deren mechanische Umsetzung in eine Komponente integriert sind. Eine Besonderheit in diesem Bereich ist der Umstand, dass diese Systeme miteinander vernetzt sind und miteinander kommunizieren. Es handelt sich um verteilte eingebettete Systeme, die neben der Automobiltechnik auch in anderen Bereichen wie Avionik, Telekommunikation, Medizintechnik, Anlagenbau und Schienenverkehr Anwendung finden [BRO98].

Für den Begriff der Architektur im Allgemeinen gilt im Kontext dieser Arbeit folgende Definition des Institute of Electrical and Electronics Engineers (IEEE) [IEEE1471, S.3] (Def. 5):

> *„The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.“*

Die Elektrik/Elektronik-Architektur (kurz: E/E-Architektur) eines Kraftfahrzeugs ist ein verteiltes eingebettetes System. Sie umfasst neben den Hardware- und Softwareeinheiten, die Vernetzung, den Leitungssatz, die Lage aller verbauten Teile im Fahrzeug, die funktionale Beschreibungen, An-

forderungen und Randbedingungen, die den Architekturentwurf beeinflussen. Ein weiterer wichtiger Aspekt von E/E-Architekturen ist die Betrachtung von Ausstattungsvarianten und Realisierungsalternativen [FREE07].

Abbildung 3.1 zeigt die Ebenen einer E/E-Architektur nach der EEA-ADL [PREE]. Die Ebenen sind horizontal angeordnet. Artefakte der einzelnen Ebenen werden durch sogenannte Mappings (vgl. Abschnitte 2.5.4 und 3.3.7) miteinander verknüpft.

In der Anforderungsebene (detailliert beschrieben in Abschnitt 3.3.1) werden die Anforderungen an die E/E-Architektur erfasst. Dazu gehören funktionale und nicht-funktionale Anforderungen sowie technische und gesetzliche Randbedingungen.

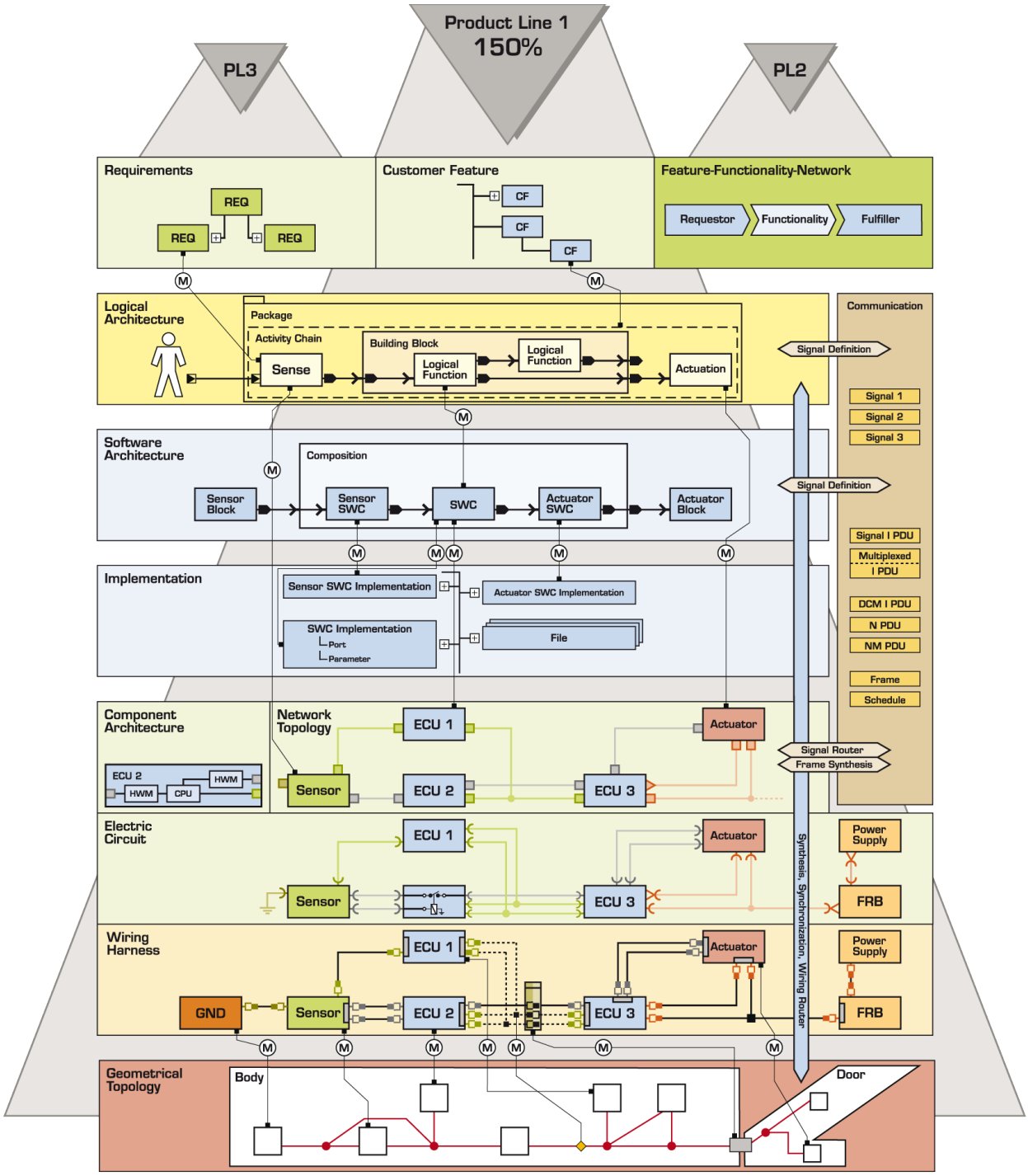


Abbildung 3.1: Ebenen einer E/E-Architektur [PREE]

In der logischen Architektur (detailliert beschrieben in Abschnitt 3.3.2) wird das abstrakte Funktionsnetzwerk sowie die Schnittstellen modelliert. Diese Ebene bietet die Modellierung und Darstellung funktionaler Zusammenhänge unabhängig davon, ob die konkrete Funktion direkt in Hardware oder in Software, welche dann auf einem Steuergerät ausgeführt wird, umgesetzt wird.

Die Softwarearchitektur stellt die strukturelle Sicht auf die Software, die in den einzelnen Steuerge-

räten läuft, dar und zeigt detaillierte Abhängigkeiten, benötigte und zur Verfügung gestellte Schnittstellen. Diese Ebene ist eine Umsetzung von Teilen des AUTOSAR-Standards [AUT]. Die darunter angeordnete Implementierungsebene beinhaltet die zur Software gehörenden Quelltexte, Ausführungsmodelle oder andere Implementierungsdateien.

Das Signalmodell (detailliert beschrieben in Abschnitt 3.3.4) listet alle innerhalb der Architektur benötigten Signale sowie deren Typisierung. Die Signale sind verknüpft mit Schnittstellendefinitionen in der logischen- und Softwarearchitektur, welche die Notwendigkeit des Signals erklären. Des weiteren sind die Signale mit Hardware und Kommunikationsmedien assoziiert, auf denen sie übertragen werden.

Die Hardwareebene (detailliert beschrieben in Abschnitt 3.3.5) unterteilt sich in drei weitere Subebenen. Die logische Vernetzung (detailliert beschrieben in Abschnitt 3.3.5.1) zeigt die in der Architektur vorhandenen Komponenten sowie deren Vernetzung untereinander. Der Stromlaufplan (detailliert beschrieben in Abschnitt 3.3.5.2) zeigt die Vernetzung aus elektrologischer Sicht. Im Leitungssatz (detailliert beschrieben in Abschnitt 3.3.5.3) werden detailliert die Leitungen, deren Kontakte, Stecker, Trennstellen usw. modelliert.

Die physikalische Geometrie (detailliert beschrieben in Abschnitt 3.3.6) zeigt die mechanischen Begebenheiten im konkreten Fahrzeug in Form von Platzierungsmöglichkeiten für Komponenten und Leitungen unter Berücksichtigung von zu erwartenden äußeren Einflüssen und Umwelteigenschaften.

Wie bereits erwähnt, werden Artefakte der Ebenen durch Mappings (detailliert beschrieben in Abschnitt 3.3.7) miteinander verbunden. Dadurch werden ebenenübergreifende Zusammenhänge modelliert, welche zum gesamtheitlichen Verständnis einer E/E-Architektur eminent wichtig sind (vgl. [MAT09]).

In vertikaler Ausrichtung in Abbildung 3.1 wird dem Produktlinien- und Variantengedanken Rechnung getragen (detailliert beschrieben in Abschnitt 3.3.8). Das E/E-Architekturmodell beinhaltet nicht nur die Beschreibung für ein konkretes Fahrzeug, sondern auch Ausstattungs- und Realisierungsalternativen, Ländervarianten, Fahrzeugderivate und ganze Plattformen.

3.2 Beschreibungssprache für E/E-Architekturen

Zur Beschreibung komplexer Datenstrukturen, wie dies bei E/E-Architekturen der Fall ist, eignet sich ein modellbasierter Ansatz. In diesem Kapitel wird dargestellt, wie sich E/E-Architekturen (sie-

he Kapitel 3.3) modellbasiert mit Hilfe des in Kapitel 2.1.1 vorgestellten Vier-Schichten-Modells der OMG als MOF-basiertes (siehe Kapitel 2.1.2) Metamodell beschreiben lassen. Dazu kam ein im Rahmen dieser Arbeit entwickeltes Verfahren zur Beschreibung formaler Metamodelle zum Einsatz. Dieses Verfahren ist in Kapitel 3.2.1 detailliert beschrieben.

Wesentlicher Bestandteil für Austauschbarkeit und Verständlichkeit der Architekturmodelle sind graphische Visualisierungen, z.B. in Form von Diagrammen oder Tabellen- bzw. Baumansichten. Während in der ersten Version des UML Standards den Diagrammaustausch noch nicht berücksichtigte, wurde in der UML 2.0 [UML20] das Diagram Interchange DI, inzwischen abgelöst durch die Diagram Definition [OMG10DD], spezifiziert.

Wie in Abschnitt 2.1.1 beschrieben, sind Metamodelle in der M2-Schicht des Vier-Schichten-Modells der OMG eingeordnet. Mit Hilfe von Metamodellen können Domänen-spezifische Informationen und deren Zusammenhänge formal spezifiziert werden. Dazu stehen die in Kapitel 2.3.3 beschriebenen Sprachkonstrukte zur Verfügung.

3.2.1 Verfahren zur Aufstellung formaler Metamodelle

In [GSKMG05] wird ein Verfahren zur Aufstellung formaler Metamodelle beschrieben. Es bezieht sich auf das struktur-semantische Metaisierungsprinzip und fand sowohl bei der Entwicklung des EEA-ADL (siehe Kapitel 3.2) als auch bei der Entwicklung des Metrik-Metamodells (siehe Kapitel 6.4) Anwendung.

Es geht davon aus, dass das zur Beschreibung der relevanten Datenstrukturen notwendige Wissen und das Wissen zur Metamodellierung dieser Zusammenhänge auf mehrere Rollen verteilt ist, im folgenden „Kunde“ und „Modellierer“ genannt.

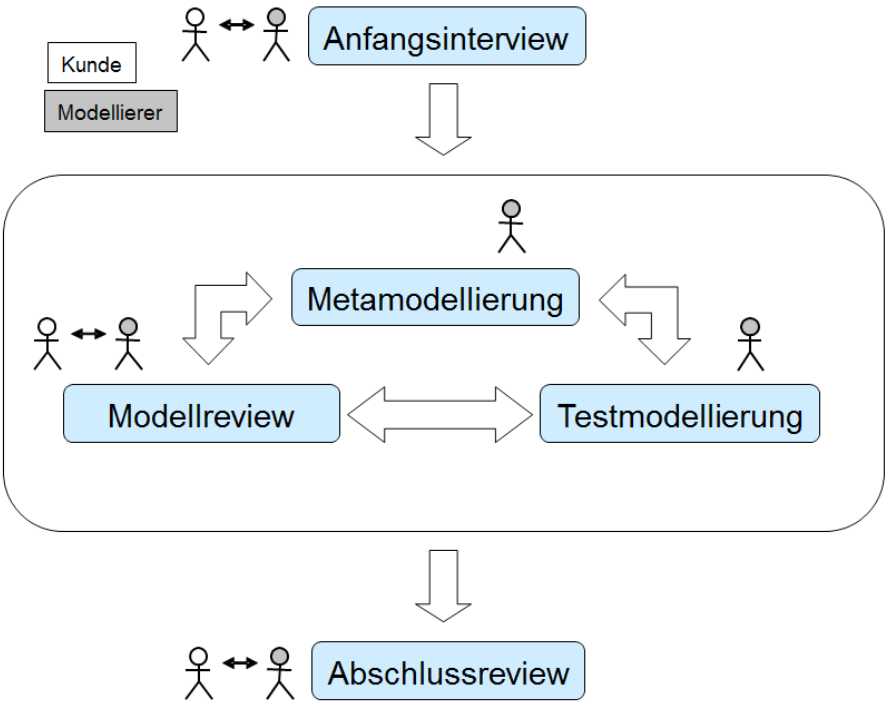


Abbildung 3.2: Verfahren zur Aufstellung formaler Metamodelle [GSKMG05]

Das Verfahren wird in Abbildung 3.2 gezeigt. In einem Anfangsinterview wird ein einheitliches Verständnis über Begrifflichkeiten, grundlegende Zusammenhänge und Umfang des zu entwickelnden Metamodells geschaffen. Die nachfolgenden Schritte werden iterativ bis zur Fertigstellung des Metamodells durchlaufen. In einer Metamodellierungsphase wird das Metamodell durch den Modellierer weiterentwickelt. In sinnvollen Abständen wird das Metamodell in Form einer Testmodellierung instanziiert und überprüft, ob die Strukturen und Zusammenhänge den Vorstellungen des Kunden entsprechen. Dies wird ebenfalls vom Modellierer durchgeführt. In regelmäßigen Abständen kommen Modellierer und Kunde zu einem Modellreview zusammen, um den Entwicklungsstand des Metamodells und das weitere Vorgehen zu besprechen. Sind die Arbeiten am Metamodell abgeschlossen, wird es durch ein Abschlussreview finalisiert.

Für spätere Weiterentwicklung wird das Verfahren von Anfang an durchlaufen. Das Ergebnis ist das Metamodell in einer neuen Version. Die Migration der vorhandenen Modelle in die neue Metamodellversion ist in diesem Fall bereits vom Anfangsinterview an zu beachten. Die technische Umsetzung der Migration ist eine Modell-zu-Modelltransformation (siehe Kapitel 2.4) naheliegend, kann aber auch anders implementiert werden [REI05].

3.2.2 Entwurfsmuster für Metamodelle

In diesem Abschnitt werden allgemeine, nicht domänenspezifische Konzepte des EEA-Metamodells vorgestellt. Diese Konzepte tragen vor allem zur einheitlichen Strukturierung eines Modells bei.

3.2.2.1 Modellwurzel

Alle in dieser Arbeit behandelten Modelle erfüllen ein wesentliches Ordnungskriterium: Sie sind bezüglich ihrer Kompositionen (siehe Kapitel 2.5.1) streng hierarchisch aufgebaut. Unter Modellwurzel versteht man den obersten Knoten im Modell, in den (direkt oder indirekt) alle anderen Modellartefakte eingefügt sind.

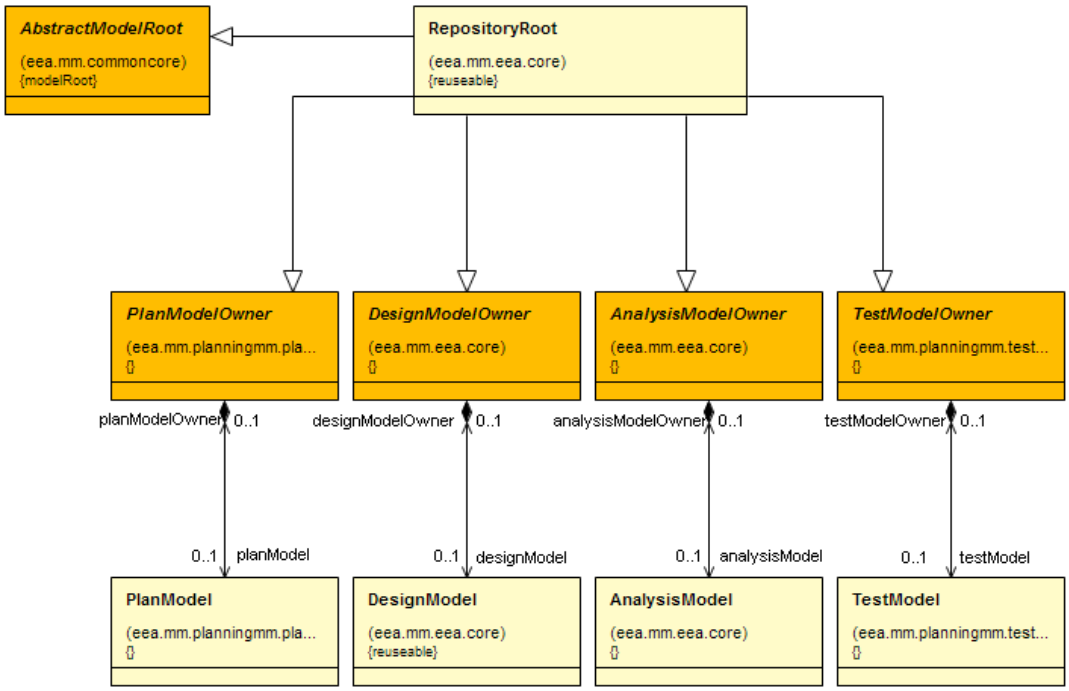


Abbildung 3.3: Konzept für Modellwurzel

Wie in Abbildung 3.3 dargestellt, repräsentiert die abstrakte Metaklasse *AbstractModelRoot* eine allgemeine Modellwurzel. Die erbende Klasse RepositoryRoot repräsentiert die Wurzelklasse im domänenspezifischen Modell und spannt dort die ersten Hierarchieebenen auf. In diesem Diagramm sind das die Submodelle *PlanModel* zur Projektplanung und Veränderungsmanagement, *DesignModel* für die E/E-spezifischen Inhalte, *AnalysisModel* zur Abbildung von sicherheitsrelevanten Daten und *TestModel* für Testdaten.

3.2.2.2 Container

Allgemein gesehen ist ein Container ein Behälter für Artefakte. Jedoch unterstützt das Container-Konzept keine hierarchische Komposition, wie zum Beispiel die im nachfolgenden Abschnitt 3.2.2.3 beschriebene paketorientierte Kompositionshierarchie. Außerdem haben diese Kompositionen zunächst keine semantische Bedeutung. Das heißt, dass die Semantik einer solchen Komposition aus Sicht des Metamodells allein ordnungshierarchische Gründe hat. In der Regel folgt einem Container im Kompositionsbaum eine paketorientierte Kompositionshierarchie.

Die Semantik, dass ein Artefakt ein Container ist, wird durch die Vererbung zu *AbstractContainer* ausgedrückt. Jedoch verfügt diese abstrakte Metaklasse über keine Kompositionen, welche spezifizieren, welche Elemente enthalten sein dürfen. Dies obliegt den erbenden Subklassen, deren Kompositionsbeziehungen zu anderen Metaklassen den spezifischen Containerinhalt beschreiben.

Auf der einen Seite birgt dieses Konzept eine Schwäche, da die abstrakte Metaklasse das repräsentierte Konzept nicht selbst realisiert, sondern nur die Semantik vorgibt. Demnach ergibt sich hier eine Anforderung an die Konsistenz eines Metamodells, dass nur Metaklassen von *AbstractContainer* erben dürfen, die auch tatsächlich Container sind. Auf der anderen Seite trägt dieses Konzept dazu bei, dass alle im Modell enthaltenen Containerartefakte über diese Metaklasse behandeln kann, obwohl die spezifischen erlaubten Inhalte nur durch Kenntnis der konkreten Subklasse zu erschließen ist.

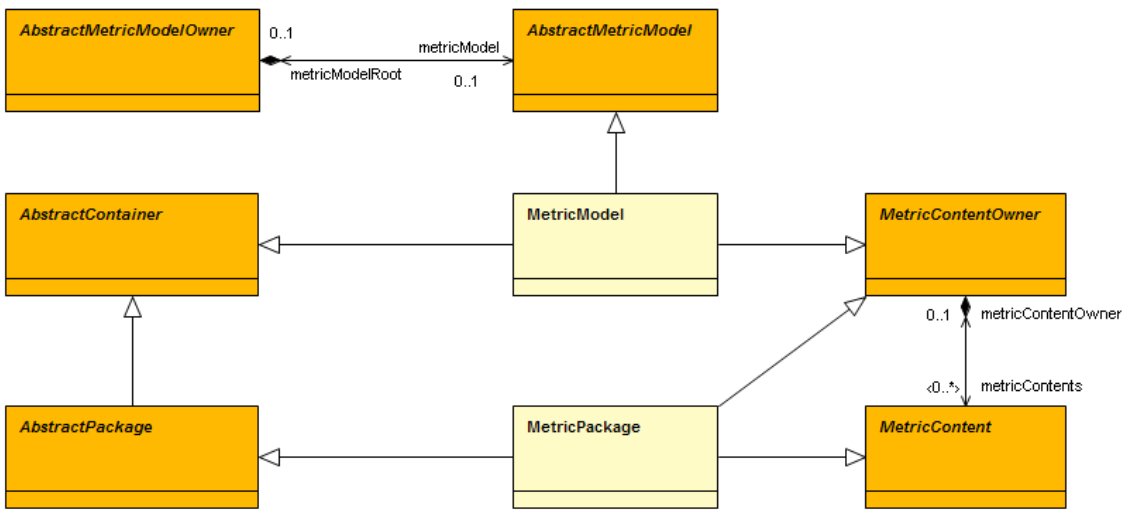


Abbildung 3.4: Beispielkonzepte für Container- und Pakethierarchisierung

In Abbildung 3.4 werden Beispiele für Container- und Pakethierarchisierung gezeigt. Die Klasse *MetricModel* erbt von *AbstractContainer* und kann über die Vererbung zu *MetricContentOwner* Ar-

tefakte des Typs *MetricContent*, wie zum Beispiel das *MetricPackage*, beinhalten. Das *MetricModel* kann keine weiteren Metrikmodelle aufnehmen, das ist der wesentliche Unterschied zur paketorientierten Kompositionshierarchie.

3.2.2.3 Paketorientierte Kompositionshierarchie

Die paketorientierte Kompositionshierarchie basiert auf dem Paketkonzept der UML. Kerngedanke ist, dass ein Paket Elemente oder andere Pakete durch Komposition beinhalten darf. Wie in Abbildung 3.4 dargestellt, erbt die Metaklasse *AbstractPackage* von der Metaklasse *AbstractContainer*, da ein Paket andere Elemente enthält, um diese dadurch in einer bestimmten Ordnung zu verwalten. Zusätzlich zum Containerkonzept spezifiziert das Paketkonzept eine strenge Kompositionshierarchie der Pakete bezüglich der Kompositionen. Dadurch können streng hierarchische Bäume aufgespannt werden.

Analog zum Containerkonzept spezifiziert das abstrakte Paketkonzept nicht die erlaubten Kindelemente. Dies obliegt den erbenden Subklassen.

Das in Abbildung 3.4 zu sehende Beispiel zeigt ebenfalls das Paketkonzept. Die Klasse *MetricPackage* erbt von *AbstractPackage* zur semantischen Bindung an das Paketkonzept. Über die Vererbungen zu *MetricContentOwner* und *MetricContent* kann ein Metrikpaket Inhalte aufnehmen, die wiederum andere Metrikpakete sein können.

3.2.2.4 Deskriptoren

Deskriptoren erlauben die Beschreibung detaillierter Informationen. Der Kontext, auf den sich diese Informationen beziehen, wird durch eine übergeordnete Komposition angegeben.

Ein wichtiger Deskriptor ist beispielsweise das Konzept zur Beschreibung von Materialkosten nach Abschnitt 4.2.2, wie in Abbildung 3.5 dargestellt.

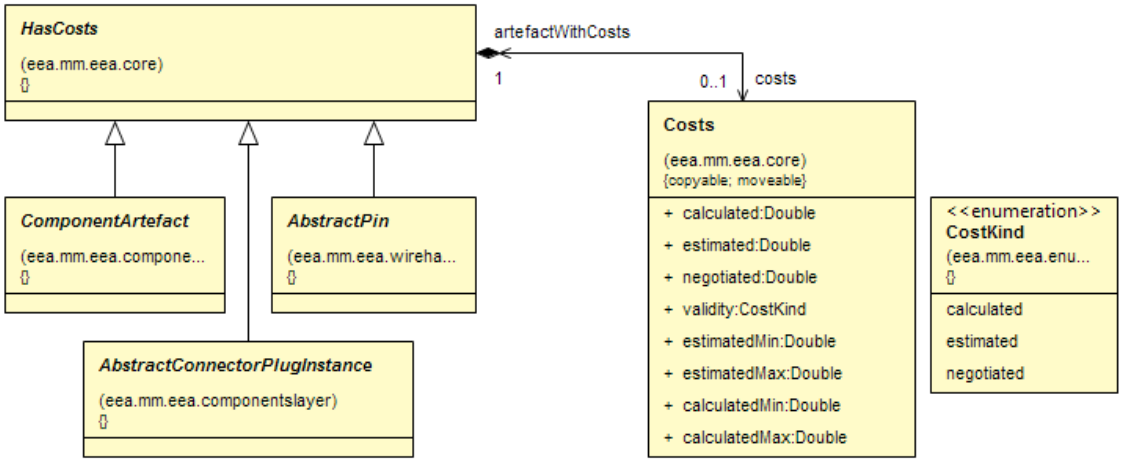


Abbildung 3.5: Konzept für Materialkosten

Artefakte, welche über detaillierte Kosteninformationen verfügen, erben von der abstrakten Klasse *HasCosts*. Die gezeigten Vererbungen in Abbildung 3.5 sind der besseren Übersichtlichkeit wegen nicht vollständig und dienen lediglich dem besseren Verständnis. Über eine Komposition kann maximal ein Kostenobjekt der Metaklasse *Costs* zugeordnet werden. Diese Klasse ist Kern des Deskriptor-Konzeptes, sie beinhaltet die Kostenattribute für geschätzte, berechnete und verhandelte Kosten, jeweils mit zusätzlichen Minimum- und Maximumangaben. Das Attribut *validity* spezifiziert die aktuelle Gültigkeit. Es ist durch die Enumeration *CostKind* typisiert und kann daher nur die dort enthaltenen diskreten Werte annehmen.

Analog dazu gibt es viele weitere Deskriptor-Konzepte, z.B. zur Beschreibung von Gewichten, Kommunikationsattributen etc.

3.2.2.5 Typ-Prototyp-Instanz

Beim Typ-Prototyp-Instanz[3]-Entwurfsmuster handelt sich um eine kaskadierte Anwendung des Typ-Instanz-Entwurfsmusters (vgl. Abschnitt 2.5.2). Obwohl der Begriff *Prototyp* eine Analogie zum Prototyp-Entwurfsmuster (vgl. Abschnitt 2.5.3) suggeriert, die sich auch in der statischen Struktur der Konzepte zeigt, ist die Semantik dieser Muster durchaus verschieden. Das Prototyp-Entwurfsmuster gehört zu den objektbasierten Erzeugungsmustern [GOF95]. Es findet seine Stärken in der Delegation von Aufrufen von einem Client zu einem einheitlich typisierten Prototyp, dessen konkrete Implementierung schnell und einfach ausgetauscht werden kann. Das Typ-Prototyp-

3 Eine „Instanz“ ist in diesem Abschnitt, sofern nicht anders angegeben, als Konzept innerhalb des Metamodells zu verstehen. An dieser Stelle ist kein Ebenenwechsel gemäß des Vier-Schichten-Modells (vgl. Abschnitt 2.1.1) z.B. aus der Metamodell-Ebene M2 in die Modell-Ebene M1 gemeint.

Instanz-Entwurfsmuster dient formalen, mehrstufigen Typisierung komplexer Objektstrukturen und findet daher vor allem in der Metamodellierung Anwendung.

Zwischen Typ und Instanz wird ein expliziter Prototyp definiert. Dieser Prototyp wird typisiert durch die Assoziation zum Typ und ist aus als Instanz zu sehen[4]. Der Prototyp dient wiederum als Typisierung für die darauffolgende Instanz[5]. Die Instanz ist somit als direkte Instanz des Prototyps und als in direkte Instanz des Typs anzusehen.

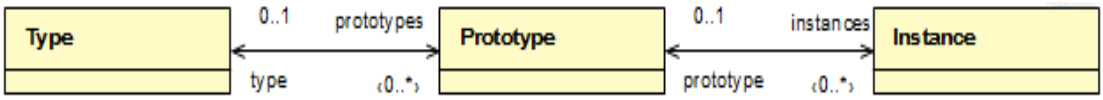


Abbildung 3.6: Das Typ-Prototyp-Instanz-Entwurfsmuster

Abbildung 3.6 zeigt das Typ-Prototyp-Instanz-Entwurfsmuster mit den wesentlichen Klassen, ihrer Semantik folgend benannt, *Type*, *Prototype* und *Instance*. Das linke Ende beider Assoziationen zeigt auf den jeweiligen direkten Typen, das rechte Ende auf die Instanz.

Die Semantik dieses Entwurfsmusters ist, dass der Prototyp durch den Typ und die Instanz durch den Prototypen typisiert werden. Durch Einbettung dieser Klassen in weiterführende Konzepte entfaltet dieses Entwurfsmuster erst eine tiefere Semantik.

3.3 Modellbasierte Beschreibung und Visualisierung von E/E-Architekturen

In diesem Abschnitt werden die zur Beschreibung von E/E-Architekturen notwendigen Abstraktionsebenen vorgestellt. Zusätzlich werden für jede Abstraktionsebene die wesentlichen, strukturellen Konzepte im Metamodell vorgestellt (Kapitel 3.3.1 bis 3.3.6). Auf die Darstellung der Attribute wurde dabei aus Gründen der besseren Übersichtlichkeit verzichtet. Im Anschluss werden in Abschnitt 3.3.7 die ebenenübergreifenden Zusammenhänge und deren Modellierung im Metamodell beschrieben. In Kapitel 3.3.8 wird das Variantenmanagement mit den entsprechenden Konzepten im Metamodell vorgestellt.

3.3.1 Anforderungsbeschreibung

In der Anforderungsbeschreibung werden Anforderungen an die Elektrik/Elektronik-Architektur erfasst und modelliert. Diese Ebene wird auch Featureliste oder Funktionskatalog genannt. Es werden

4 Hierbei handelt es sich um die erste Anwendung des Typ-Instanz-Entwurfsmusters

5 Hierbei handelt es sich um die zweite Anwendung des Typ-Instanz-Entwurfsmusters.

Serien- und Sonderausstattungen sowie Fahrzeugmerkmale beschrieben. Hauptsächlich dient diese Ebene zur Dokumentation und zur Spezifikation.

Es handelt sich im wesentlichen um hierarchisch aufgebaute Textbausteine, die jeweils eine Anforderung oder eine Teilanforderung beschreiben. Die detaillierte Beschreibung von Anforderungen erfolgt durch Attribute, welche bestimmte Eigenschaften der Anforderung näher spezifizieren. Semantisch zusammengehörende Anforderungen verfügen über eine gemeinsame Menge von Attributen. Zusätzlich können Anforderungen untereinander referenziert sein, um Querbeziehungen auszudrücken. Diese Verweise werden dabei mit einer Semantik versehen, um die Art der Referenz wiederzugeben. Jedoch ist der Anteil an freier textueller Beschreibung hoch, so dass eine umfangreiche Anforderungsbeschreibung Inkonsistenzen, Redundanzen oder Widersprüche aufweisen kann.

3.3.1.1 Metamodell

Die zentrale Metaklasse der Anforderungsbeschreibung ist *Requirement* (siehe Abbildung 3.7). Über ihre Superklassen *RequirementOwner* und *RequirementChild* können Anforderungen hierarchisiert werden. Eigenschaften von Anforderungen werden durch *RequirementAttributeDefinition* typisiert, der konkrete Inhalt des Attributs ergibt sich aus der Assoziation zu *RequirementAttributeValue*. Die Zusammenstellung von Attributdefinitionen erfolgt über die Metaklasse *RequirementAttributeDefinitionSet*, welche über ein entsprechendes Kompositionsmuster hierarchisiert wird.

Die Metaklasse *RequirementLink* realisiert Verweise zwischen Anforderungen. Dessen Metaattribut *type* stattet diesen Verweis mit einer Semantik aus. Die erlaubten Werte stammen aus der Enumeration *RequirementLinkKind* mit den folgenden Bedeutungen:

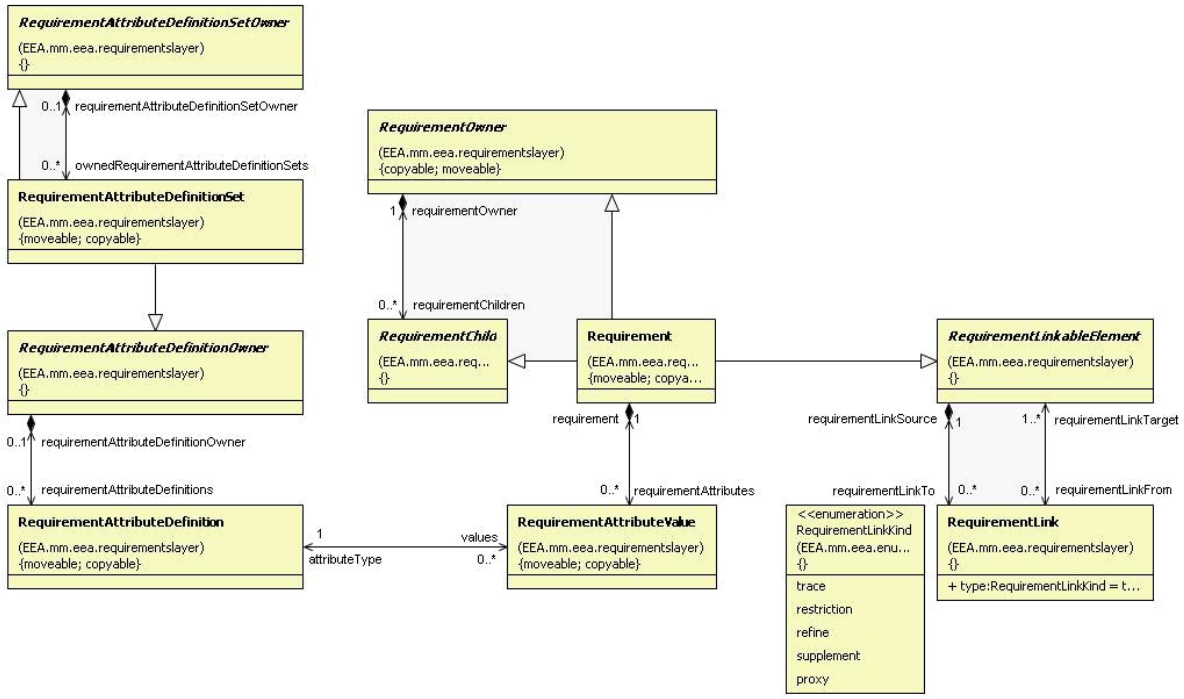


Abbildung 3.7: Wesentliche Semantik des Anforderungsmodells im EEA-Metamodell

- *trace*: Der Verweis führt zu einer Anforderung, die zum besseren Verständnis der Ursprungsanforderung dient.
- *restriction*: Der Verweis führt zu einer Anforderung, die als Bedingung für die Ausgangsanforderung zu interpretieren ist.
- *refine*: Der Verweis führt zu einer Anforderung, welche die Ausgangsanforderung näher beschreibt.
- *supplement*: Der Verweis führt zu einer erweiternden Anforderung, die zusätzlich umgesetzt werden kann.
- *proxy*: Der Verweis führt zu einer Kopie der ausgehenden Anforderung, die jedoch noch spezifische Informationen enthält.

Der Verweis wird über eine Komposition der Quellanforderung zugeordnet, die referenzierten Anforderungen werden über eine Assoziation adressiert.

3.3.1.2 Graphische Notation

Da es sich bei Anforderungsmodellen um textlastige Datenstrukturen handelt, werden Anforderungen oft tabellarisch dargestellt.

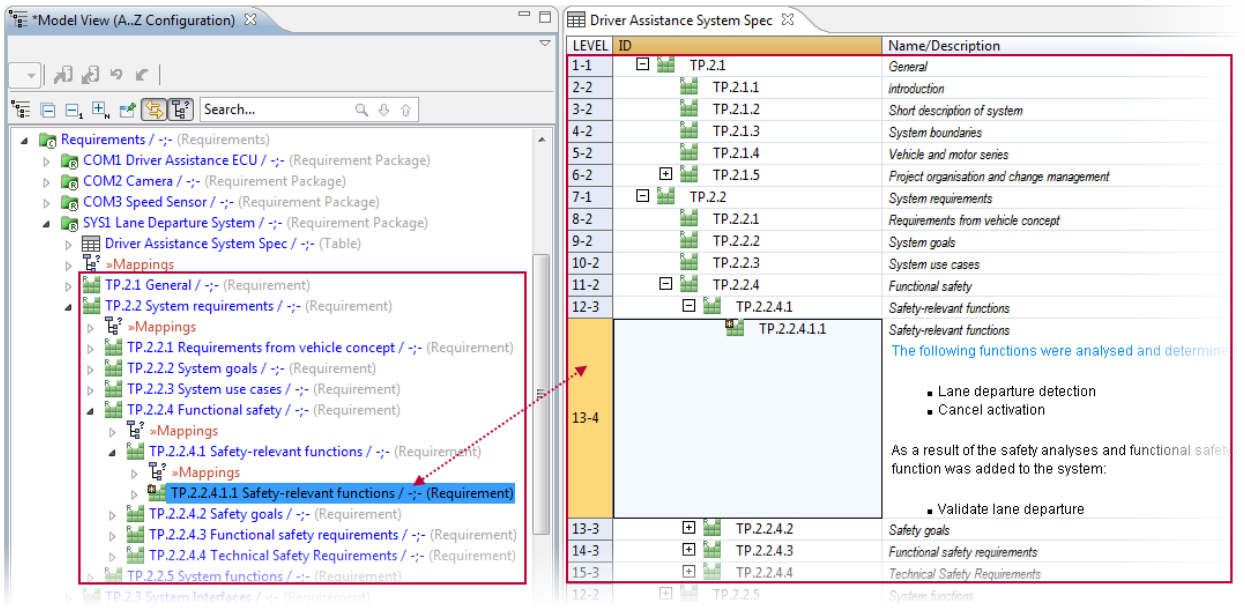


Abbildung 3.8: Darstellung von Anforderungen in PREEvision [PREE]

In Abbildung 3.8 ist eine solche Anforderungstabelle dargestellt. In ihr werden werden hierarchisch aufgebaute Anforderungen (auf der linken Seite in der *Model View*) tabellarisch aufbereitet. In den weiteren Spalten der Tabelle können Name, Beschreibung und weitere Attribute der Anforderungen visualisiert werden.

3.3.2 Logische Architektur

Die logische Architektur beschreibt die logisch funktionalen Zusammenhänge einer E/E-Architektur, unabhängig davon, ob die einzelnen Funktionen in Software oder in Hardware implementiert werden.

Über Domänen werden Subsysteme mit klar definierten Schnittstellen modelliert, über welche Interaktionen mit der Umgebung des Subsystems stattfinden. Logische Domänen können keine weiteren logischen Domänen beinhalten.

Die weiteren Artefakte der logischen Architektur unterteilen sich in atomare und zusammengesetzte Funktionen. Zu den atomaren Funktionen gehören:

- Aktuatorfunktion: Repräsentiert eine Funktionalität, die Informationen von Vorgängerblöcken entgegennimmt und ihrer Implementierung entsprechend verarbeitet bzw. ausführt. Aktuatorfunktionen werden durch Hardware oder Software in der E/E-Architektur realisiert.
- Sensorfunktion: Repräsentiert eine Funktionalität, die Informationen erfasst und nachfolgenden Blöcken zur Verfügung stellt. Sensorfunktionen werden durch Hardware oder Software in der

E/E-Architektur realisiert.

- Logische Funktion: Repräsentiert eine allgemeine logische Funktionalität, die Informationen von Vorgängerblöcken entgegennimmt, verarbeitet und an nachfolgende Blöcke weitergibt. Logische Funktionen werden durch Hardware oder Software in der E/E-Architektur realisiert.
- Umgebungsfunktion: Repräsentiert eine in der Umgebung stattfindende Funktionalität und kann daher nicht auf Hardware- oder Softwareartefakte gemappt werden.
- Servicefunktion: Repräsentiert eine externe Funktionalität für Servicezwecke[6] und kann daher nicht auf Hardware- oder Softwareartefakte gemappt werden.

Zu den zusammengesetzten Funktionen gehört der Building Block. Dieser kann neben atomaren Funktionen weitere Building Blöcke beinhalten und ermöglicht einer Hierarchisierung. Analog zu Compositions aus der System Software Architektur (vgl. Abschnitt 3.3.3) dienen Building Blocks ausschließlich zur Hierarchisierung bei. Ihre Funktionalität ergibt sich aus der Summe der beinhalteten atomaren Funktionen.

Allen Funktionen der logischen Architektur ist, dass sie durch Funktionstypen explizit typisiert werden (vgl. Typ-Instanz-Entwurfsmuster in Abschnitt 2.5.2). Die Schnittstellen werden durch typisierte Ports modelliert (vgl. Port-Entwurfsmuster in Abschnitt 2.5.5). Die Definition von Funktions- und Porttypen erfolgt analog zur System Software Architektur. Hierbei kommen dieselben Konzepte aus dem Metamodell und graphischen Diagrammeditoren zum Einsatz, deren detaillierte Beschreibung im Abschnitt 3.3.3 zu finden ist.

3.3.2.1 Metamodell

Wesentliches Merkmal der logischen Architektur ist die Umsetzung des Typ-Instanz-Entwurfsmusters (siehe Abschnitt 2.5.2) für atomare / zusammengesetzte Funktionen und Ports. Abbildung 3.9 zeigt die wesentlichen Zusammenhänge der Instanzseite der logischen Architektur.

Wesentlich für die Hierarchisierung der Funktionen sind die Klassen *AbstractLogicalBlockOwner* und *AbstractLogicalBlock*, einer Superklasse für alle Funktionsblöcke in der logischen Architektur. Sowohl *LogicalDomain* als auch *LogicalBuildingBlock* erben von *AbstractLogicalBlockOwner* und können somit weitere Blöcke beinhalten. Durch die zusätzliche Vererbung von *LogicalBuildingBlock* zu *AbstractLogicalBlock* wird die Hierarchisierung über Building Blocks realisiert. Über die typisierende Assoziation von *AbstractLogicalBlock* zu *LogicalBlockType* erfolgt der Wechsel für

6 Ein Beispiel für eine Servicefunktion ist zum Beispiel eine Funktionalität, die Diagnoseinformationen erfasst, sammelt und ausgibt.

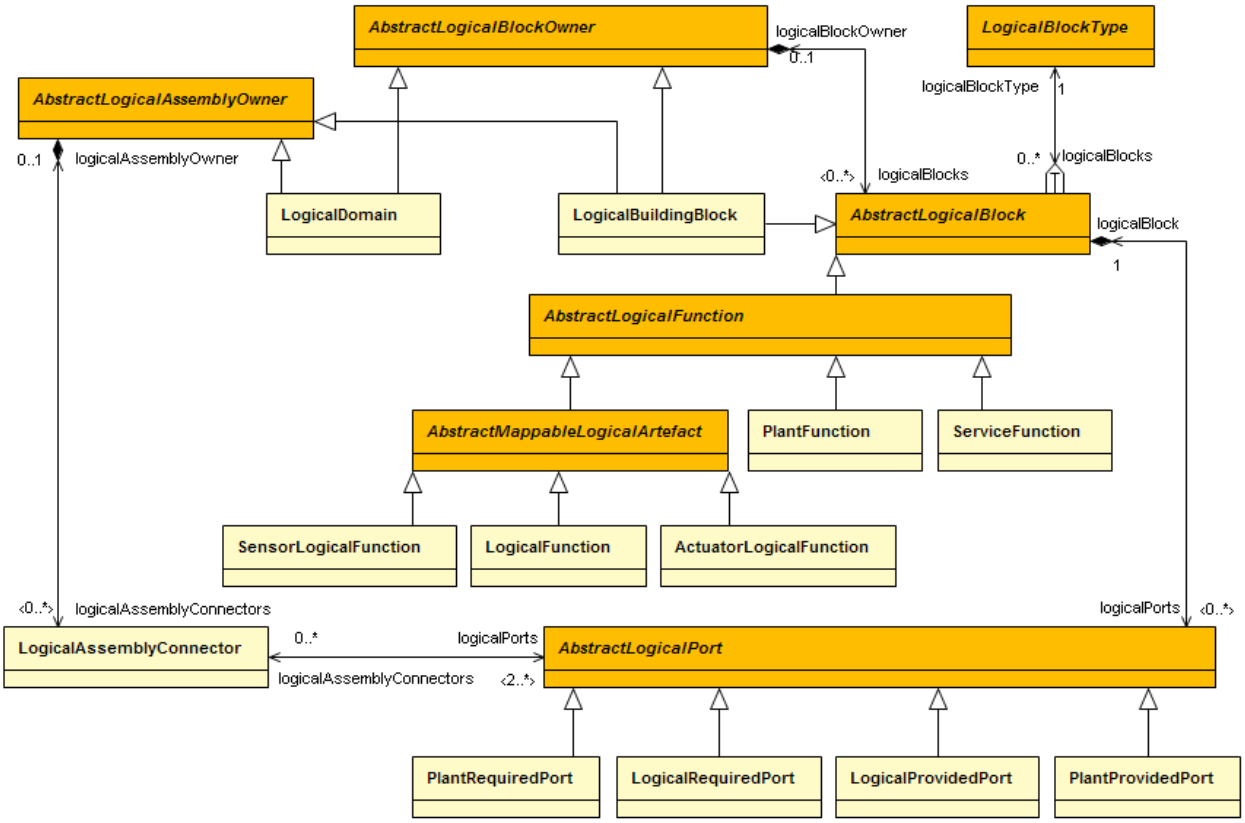


Abbildung 3.9: Instanzartefakte der logischen Architektur im EEA-Metamodell

Blöcke in die Typseite der logischen Architektur (siehe auch Abbildung 3.10).

Die Klasse *AbstractLogicalFunction* ist eine Superklasse der atomaren Funktionen. Durch weitere Vererbung wird in die verfügbaren atomaren Funktionen spezialisiert, wobei *PlantFunction* und *ServiceFunction* nicht von *AbstractMappableLogicalArtefact* erben und somit nicht auf Hardware- oder Softwareartefakte gemappt werden können (siehe Abschnitt 3.3.7.4).

Die Ports (und damit indirekt die Funktionen) werden über *LogicalAssemblyConnector* miteinander verbunden.

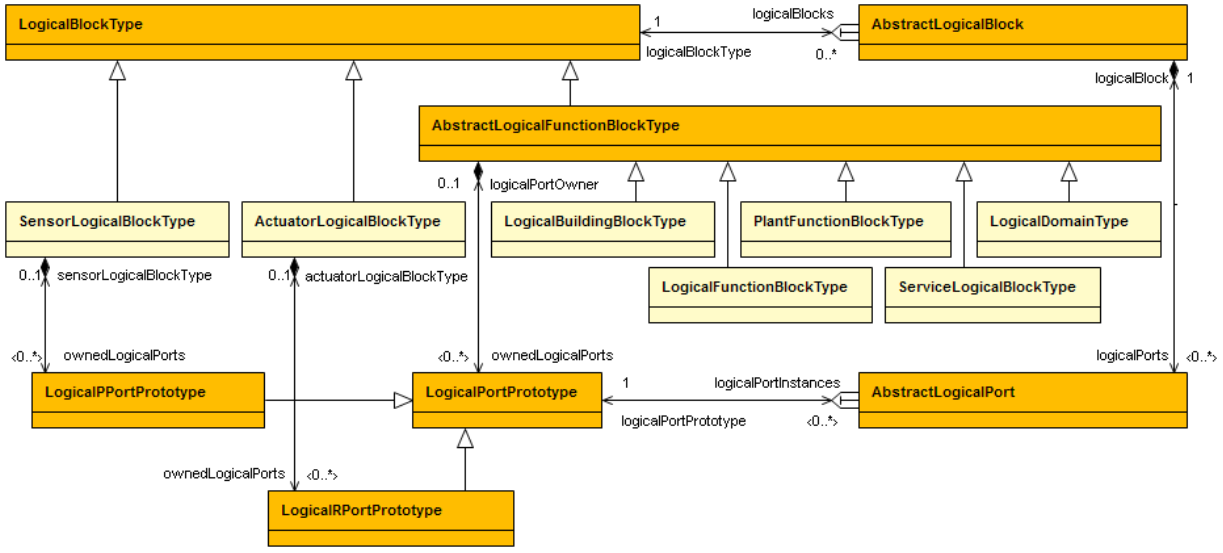


Abbildung 3.10: Typartefakte der logischen Architektur im EEA-Metamodell

Abbildung 3.10 zeigt die Modellierung der Typseite der logischen Architektur. Sämtliche Blocktypen erben von der Klasse *LogicalBlockType*, wobei die Typen der atomaren Funktionen mit Ausnahme der Aktuator- und Sensorfunktion von *AbstractLogicalFunctionBlockType* erben und damit die Komposition zu *LogicalPortPrototype* erben. Da Aktuator- und Sensorfunktionen entweder über Provided bzw. Required Ports verfügen, wird dies auf der Typseite durch dedizierte Kompositionen zu den logischen Portprototypen modelliert.

Die beiden typisierende Assoziationen zwischen *LogicalBlockType* und *AbstractLogicalBlock* bzw. *LogicalPortPrototype* und *AbstractLogicalPort* stellen die Verbindung zur Instanzseite dar. Wesentlich an dieser Stelle ist die synchrone Modellierung zwischen Blöcken und Ports sowohl auf Instanz- (Komposition zwischen *AbstractLogicalBlock* und *AbstractLogicalPort*) als auch auf der Typseite (alle anderen Komposition in Abbildung 3.10). Diesem Umstand muss bei der E/E-Architekturmodellierung in der M1-Ebene des Vier-Schichten-Modells (siehe Abschnitt 2.1.1) Rechnung getragen werden[7].

3.3.2.2 Graphische Notation

In der graphischen Notation der logischen Architektur kommen Blockdiagramme zum Einsatz. Atomare (rechtwinklige Ecken) und zusammengesetzte (abgerundete Ecken) Funktionen werden als Blöcke dargestellt. Die Ports sind an den Rändern der Blöcke platziert. Linien, welche die Ports miteinander verbinden, repräsentieren logische Assembly Connectors.

7 Das Architekturwerkzeug PREEvision verfügt hier über entsprechende Automatismen, welche die Konsistenz des Modells sicher stellen.

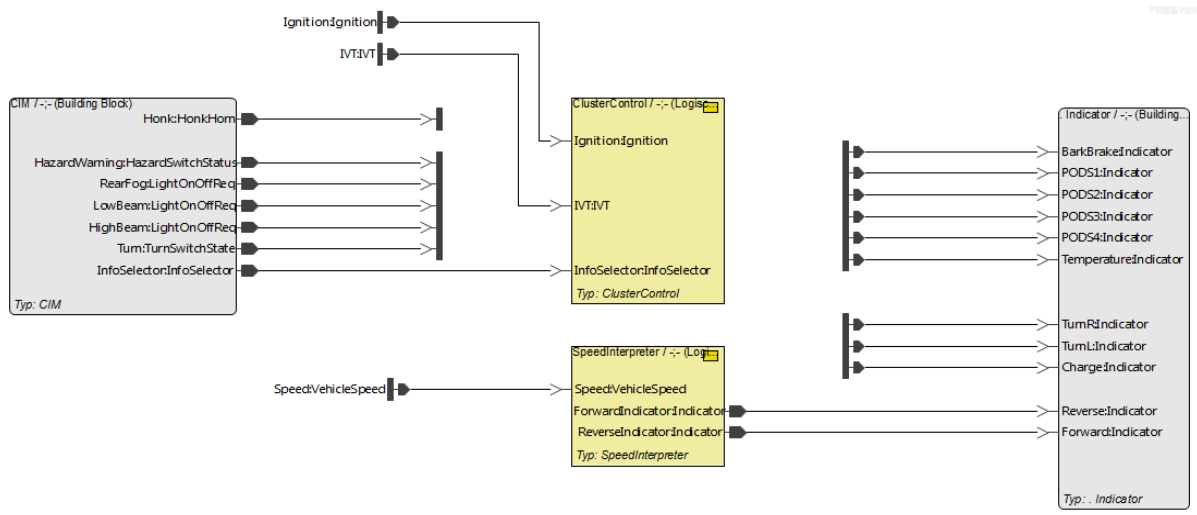


Abbildung 3.11: Graphische Notation logische Architektur (Instanzseite) [ELY11]

Abbildung 3.11 zeigt ein instanzseitiges logisches Architekturdiagramm. Die grauen Blöcke mit abgerundeten Ecken sind Building Blocks. Die Blöcke in der Mitte mit rechtwinkligen Ecken sind logische Funktionen. Die Ports an den Blockrändern verbinden über Assembly Connectors die Blöcke miteinander. Ports, die in der freien Diagrammfläche stehen gehören dem übergeordneten Building Block, dessen Innenleben in diesem Diagramm dargestellt wird, und dienen zur Kommunikation in anderen Hierarchieebenen.

3.3.3 System Software Architektur und Implementierung

Die System Software Architektur basiert im Wesentlichen auf dem Software Component Template des AUTOSAR-Standards [AUT]. Die wichtigsten Artefakte dieser Ebene sind hierarchisierbare (*Composition*) und atomare (*Atomic Software Component*) Softwarekomponenten. Zusätzlich zum AUTOSAR-Standard können Aktuator- und Sensorblöcke modelliert werden, die Datensenken und -quellen repräsentieren. Die Schnittstellen der Blöcke werden durch sogenannte Ports spezifiziert und werden miteinander durch Assemblies verbunden. Diese Artefakte bilden in ihrer Gesamtheit die System Software Architektur. Aufbau der Informationsfluss der Software wird dadurch transparent modelliert. Dabei wird in folgende, unterschiedliche Kommunikationsformen unterschieden:

- Sender – Receiver: Die Sender – Receiver Schnittstelle definiert die von entsprechenden Portprototypen gesendeten bzw. empfangenen Informationen. Bei einer Sender- / Empfängerarchitektur werden Informationen über einen direkten Übertragungskanal von einem Sender zu einem oder mehreren Empfängern gesendet. Eine Architektur nach diesem Schema ist z.B. das CAN-Netzwerk.

- Client – Server: Die Client – Server Schnittstelle definiert die von entsprechenden Portprototypen gesendeten bzw. empfangenen Informationen. Bei der Client- / Serverarchitektur besteht die Möglichkeit, Dienste in einem Netzwerk zu verteilen. Clients stellen bei Bedarf Anfragen an Dienste, die auf Servern laufen. Diese beantworten die Anfragen, indem die gewünschten Informationen zurückgeliefert werden.
- Slave – Controller: Die Slave – Controller Schnittstelle definiert die von entsprechenden Portprototypen gesendeten bzw. empfangenen Informationen. Bei einer Slave- /Controllerarchitektur handelt es sich um ein serielles Bussystem, welches physikalisch betrachtet eine Ringtopologie aufweist und auf synchroner Datenübertagung basiert. Eine Architektur hierfür ist zum Beispiel ein MOST-Bis in Ring-Struktur. Diese Schnittstellenart geht über den AUTOSAR-Standard hinaus.

Die Blöcke werden durch Kompositionsblöcke, die wiederum über eine eigene Schnittstellenbeschreibung verfügen, hierarchisiert. Diese Hierarchisierung hat keine funktionalen Auswirkungen. Es dient zur Modularisierung von semantisch zusammenhängenden Softwarekomponenten sowie zur Reduzierung der Komplexität innerhalb einer Hierarchieebene. Ports an Kompositionen haben delegierenden Charakter[8] und benötigen im Gegensatz zu den Ports der atomaren Softwarekomponenten keine weitere Definition ihrer Kommunikation.

3.3.3.1 Metamodell

Ein wesentliches Merkmal im Metamodell der System Software Architektur ist das Typ-Prototyp-Instanz[9]-Konzept (TPI) (vgl. Abschnitt 3.2.2.5) für Softwarekomponenten und Ports.

Diese Modellierung dient zur Umsetzung der folgenden Konzepte:

- Aufbau einer wiederverwendbaren Software-Komponentenbibliotheken für Typen und Prototypen
- Trennung zwischen der im Typisierung von Softwarekomponenten und deren Instanziierung und Hardwarezuweisung

In Abbildung 3.12 wird der Metamodellausschnitt für das Typ-Prototyp-Instanz-Konzept in der System Software Architektur gezeigt. Auf der linken Seite ist die Typ-Modellierung, in der Mitte

8 Ein solcher Port heißt *DelegationPort*. Über ihn wird Kommunikation modelliert, welche die Kompositionsgrenze überschreitet.

9 Eine „Instanz“ ist in diesem Abschnitt, sofern nicht anders angegeben, als Konzept innerhalb des Metamodells zu verstehen. An dieser Stelle ist kein Ebenenwechsel gemäß des Vier-Schichten-Modells (vgl. Abschnitt 2.1.1) z.B. aus der Metamodell-Ebene M2 in die Modell-Ebene M1 gemeint.

die Prototyp-Modellierung und auf der rechten Seite die Modellierung der Instanzen.

Es wird grundsätzlich zwischen zusammengesetzten („*Composition*“ kommt im Namen der Klasse vor) und atomaren Softwarekomponenten („*AtomicSWComponent*“ kommt im Namen der Klasse vor) unterschieden. Diese Unterscheidung zieht sich durch alle drei Instanziierungsebenen des TPI durch.

Auf der Typ-Ebene gibt es *CompositionType* für zusammengesetzte und *AtomicSWComponentType* für atomare Softwarekomponenten. Die Schnittstellen von atomaren Softwarekomponenten-Typen werden durch *PortType* spezifiziert, deren detaillierte Modellierung später in diesem Abschnitt näher beschrieben wird. Die KompositionsTypen verfügen über *DelegationPortType*, die nicht weiter verfeinert werden.

Die Prototypen führen die Struktur der Typen fort. Es *CompositionPrototype*, *AtomicSWComponentPrototype* und *SWPortPrototype*, die jeweils über eine typisierende Assoziation zu den Typklassen verfügen. Analog zu den *PortTypes* werden *SWPortPrototypes* über eine Kompositionsbeziehung Softwareprototypen untergeordnet. Zur Modellierung zusammengesetzter Prototypen ist die Komposition zwischen *AbstractSWPrototype* zu *CompositionType* von entscheidender Bedeutung. Darüber werden Prototypen einem *CompositionType* untergeordnet und spezifizieren ihren inneren Aufbau. Über die typisierende Assoziation der Prototypen zu anderen Typen werden hierarchische Kompositionen aufgebaut.

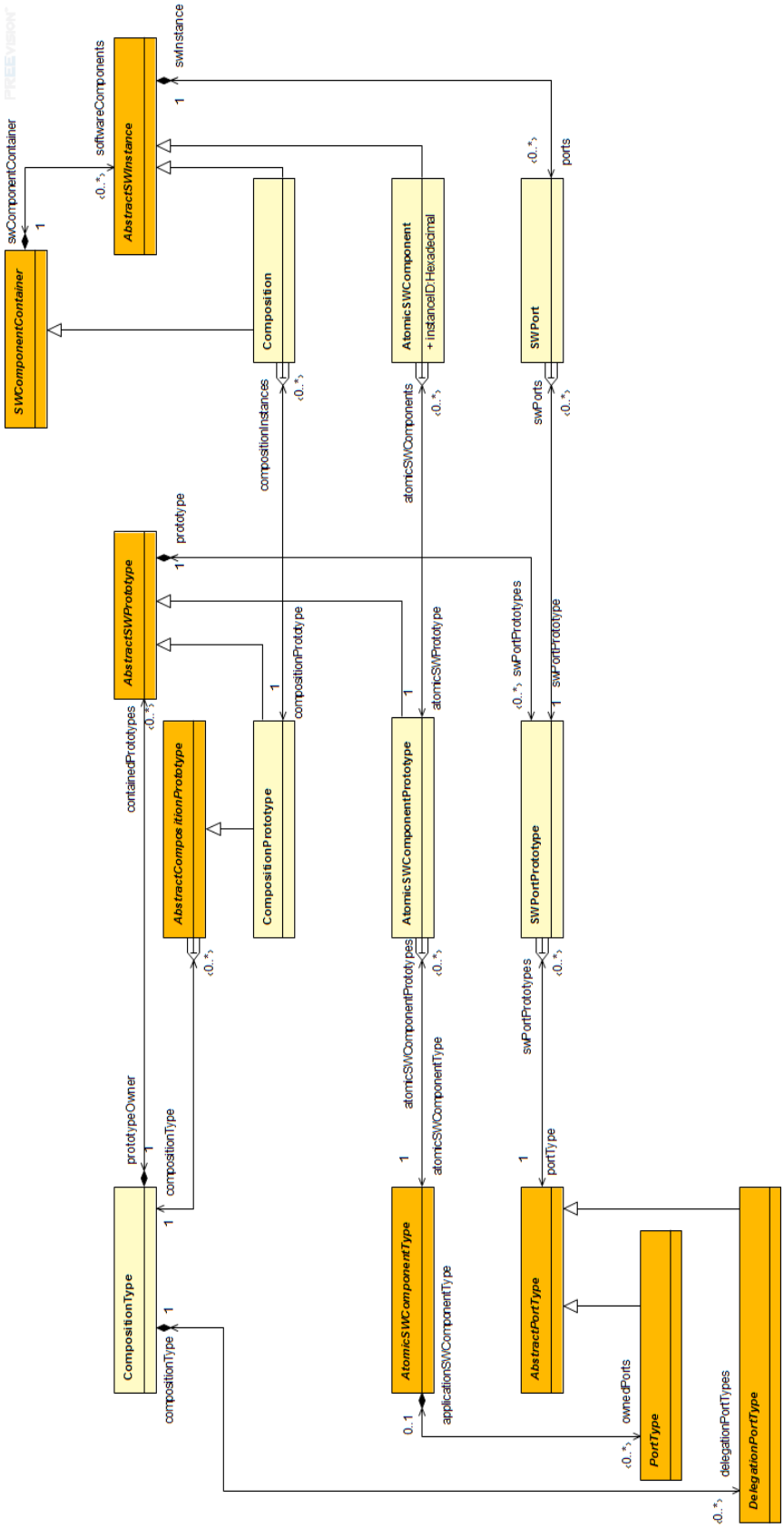


Abbildung 3.12: System Software Architektur

Analog zu den Typen und den Prototypen wird die Struktur durch die Instanzklassen *Composition*, *AtomicSWComponent* und *SWPort* fortgeführt, die durch die jeweiligen Prototypen typisiert werden. Über die Kompositionsbeziehung zu *AbstractSWInstance* sind *SWPort* den Softwareblöcken untergeordnet. Über die Kompositionsbeziehungvon *AbstractSWInstance* zu *SWComponentContainer* und den Vererbungen in diesem Umfeld, findet eine Variante des Kompositum-Entwurfsmusters Anwendung und erlaubt die Modellierung hierarchischer *Composition*, die wiederum aus *Composition* oder *AtomicSWComponent* bestehen.

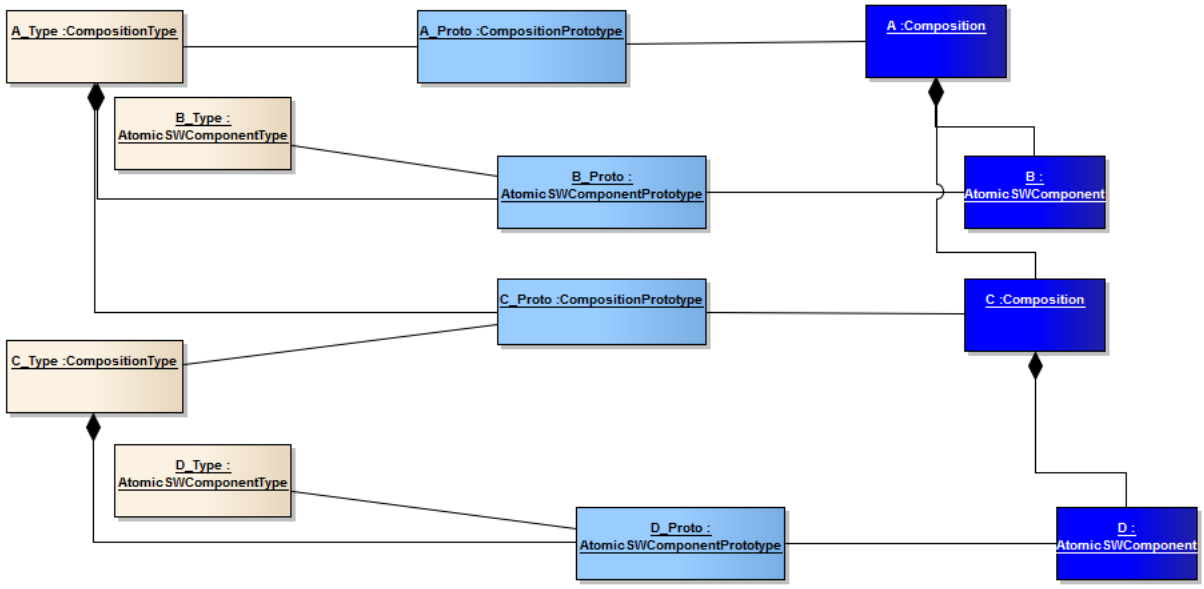


Abbildung 3.13: Beispiel für TPI in der System Software Architektur

Zur Veranschaulichung ist im Objektdiagramm in Abbildung 3.13 ist ein Beispiel zum TPI-Konzept in der System Software Architektur dargestellt. Der besseren Übersichtlichkeit wegen wird hier auf die Darstellung von Ports und Assemblies verzichtet. Ausgehend von den Kompositionstypen *A_Type*, *C_Type* und den atomaren Softwarekomponententypen *B_Type* und *D_Type* gibt es entsprechende Prototypen *A_Proto* bis *D_Proto*. Der innere Aufbau des Kompositionstyps *A_Type* wird durch *B_Proto* und *C_Proto* beschrieben. *C_Proto* wird durch *C_Type* typisiert und beinhaltet *D_Proto*. Durch die Einrückung der Prototypen wird die Struktur des hierarchischen Composition-Prototyps *A_Proto* angedeutet. Diese Hierarchie wird in der Instanzebene durch die Kompositionen explizit modelliert.

Port-Typen werden mit an die UML angelehnten Interfaces [UML241] verbunden. Sie spezifizieren die Schnittstellenkonfiguration der Softwarekomponenten. Ein Interface beinhaltet die Datenelemente, die für den Informationsaustausch verwendet werden. Zudem werden Interfaces über ein

Vererbungskonzept, dessen Semantik ebenfalls an die UML-Notation angelehnt ist, hierarchisiert. Dieser Sachverhalt wird in Abbildung 3.14 dargestellt.

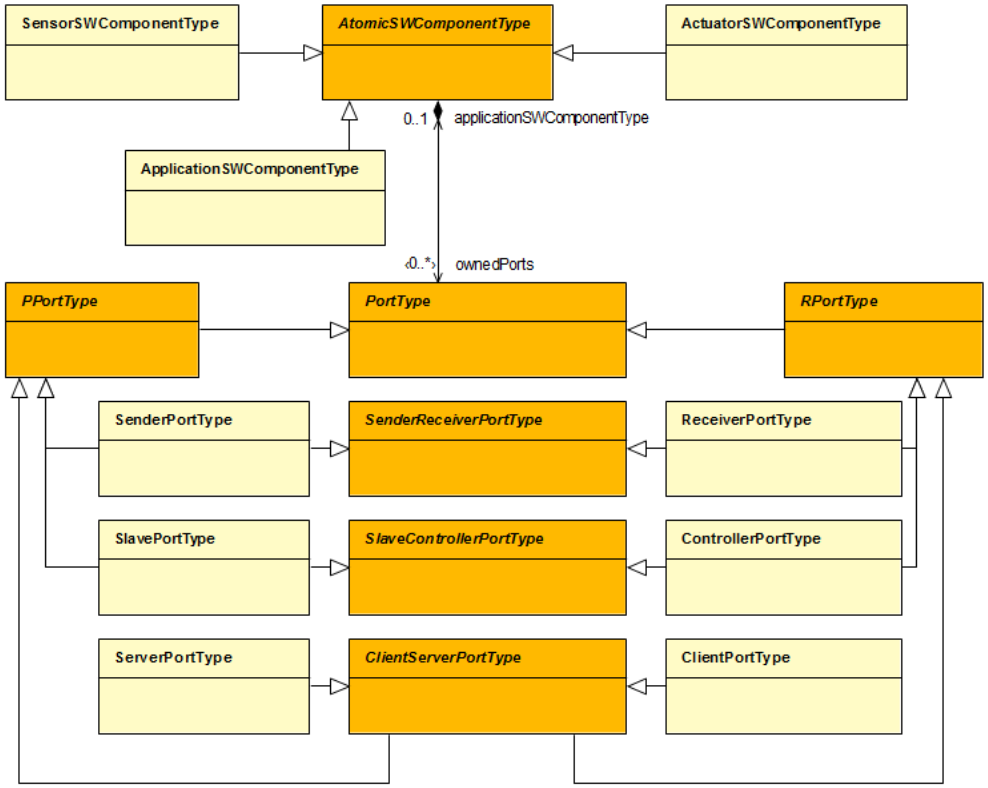


Abbildung 3.14: Portprototypen der Funktionsbibliothek

Über die Metaklasse *PortType* wird die Schnittstellensignatur von atomaren Softwarekomponententypen spezifiziert. Die spezifischen Subklassen *RPortType*[10] und *PPortType*[11] unterteilen die Portprototypen weiter in Ports, die Daten empfangen und Ports, die Daten zur Verfügung stellen. Die grundsätzliche Unterscheidung in *PPortType* und *RPortType* wird weiter verfeinert durch die verschiedenen unterstützen Kommunikationsarten, wie sie bereits im vorangegangenen Abschnitt 3.3.3 beschrieben sind. Diese werden hauptsächlich durch die Metaklassen *SenderReceiverPortType*, *ClientServerPortType* und *SlaveControllerPortType* repräsentiert. Die davon erbenden konkreten Klassen stellen die jeweiligen sendenden bzw. empfangenden Porttypen dar. Über die entsprechenden Kompositionen werden die Porttypen an die entsprechenden atomaren Softwarekomponententypen angehängt.

10 Der Anfangsbuchstabe *R* steht für Receiver. *RPortType* ist eine Superklasse für alle Porttypen, die Informationen erwarten.

11 Der Anfangsbuchstabe *P* steht für Provider. *PPortType* ist eine Superklasse für alle Porttypen, die Informationen zur Verfügung stellen.

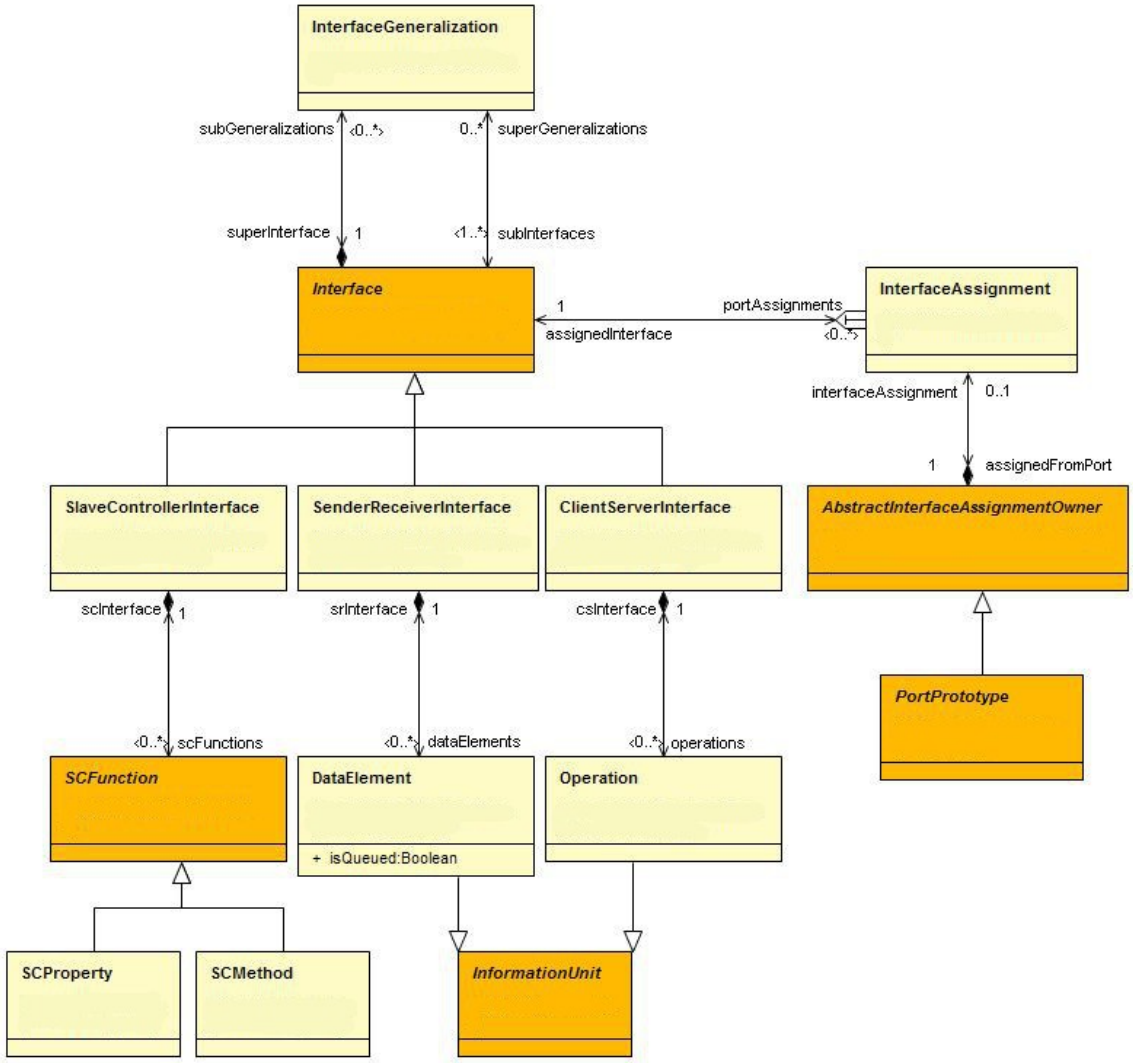


Abbildung 3.15: Interfaces der Funktionstypen-Ebene

Die detailliertere Beschreibung der Porttypen erfolgt über eine Zuweisung *InterfaceAssignment* zu einer Schnittstelle (Metaklasse *Interface*), wie in Abbildung 3.15 gezeigt. Um die bereits beschriebenen Kommunikationsarten bei der Schnittstellenspezifikation zu unterstützen, gibt es die entsprechenden Spezialisierungen *SlaveControllerInterface*, *ClientServerInterface* und *SenderReceiverInterface*. Über sie verfügen Schnittstellen durch entsprechende Kompositionen über Informationseinheiten (Metaklasse *InformationUnit*). Bedingt durch die Unterschiede in den Kommunikationsarten gibt es entsprechend verschiedene Spezialisierungen einer Informationseinheit.

Neben der Schnittstellenzuweisung und den verschiedenen Schnittstellenarten zeigt Abbildung 3.15 das Vererbungskonzept für Schnittstellen. Über die Metaklasse *InterfaceGeneralization* können Schnittstellen miteinander verbunden werden. Dabei gehört die Schnittstellengeneralisierung über

die Komposition dem vererbenden Interface. Die erbende Schnittstelle wird über die Assoziation an die Generalisierung verbunden. Die Semantik dieses Vererbungskonzepts ist an die UML angelehnt. Die zu den Schnittstellen gehörenden Informationseinheiten werden über die Generalisierung vererbt und sind mit Attributen einer Klasse in der UML vergleichbar. Die Schnittstellenzuweisungen werden hingegen nicht vererbt, das nahe liegende Analogon zu vererbten Assoziationen aus der UML gilt hier nicht.

3.3.3.2 Graphische Notation

Analog zum Metamodell orientiert sich die graphische Notation der System Software Architektur am AUTOSAR-Standard [AUT]. Dabei handelt es sich um Blockdiagramme (Repräsentation von atomaren Softwarekomponenten- und Kompositionsblöcken), die über Ports und Assembly Connectoren miteinander verbunden sind und die System Software Architektur darstellen. In dieser Diagrammart (siehe Abbildung 3.16) steht die Vernetzung von Funktionen und deren hierarchischer Aufbau im Vordergrund.

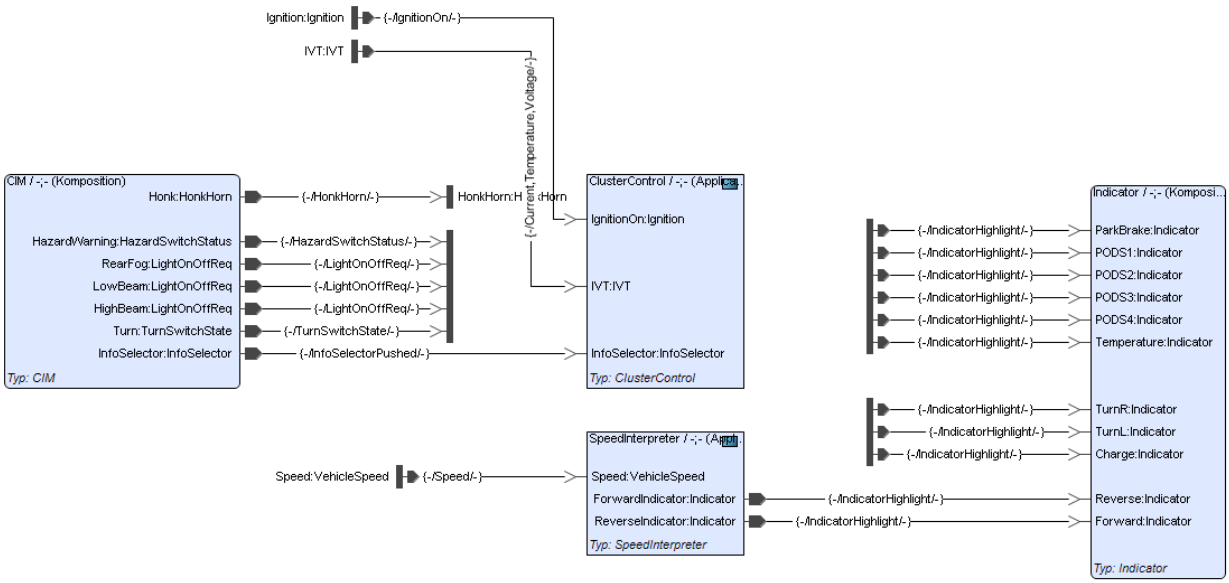


Abbildung 3.16: Graphische Notation System Software Architektur [ELY11]

Die in dieser Sicht dargestellten Blöcke und Ports sind Instanzen von entsprechenden Prototypen. Atomare Softwarekomponenten werden als Block mit rechtwinkligen Ecken dargestellt, Kompositionen werden als Block mit abgerundeten Ecken gezeichnet. In der oberen linken Ecke ist der Name des Blocks, in der unteren rechten Ecke der zugeordnete Prototyp. An den Blockrändern werden die vorhandenen Ports angeordnet, daneben deren Beschriftung. Ports in der freien Diagrammfläche repräsentieren die Delegationsports der übergeordneten Komposition. Assembly Connec-

toren werden als beschriftete Linien dargestellt.

Die graphische Repräsentation für Block- und Porttypen basiert ebenfalls auf einem Blockdiagramm, in welchem der strukturelle Aufbau der Typen im Vordergrund steht (Abbildung 3.17).

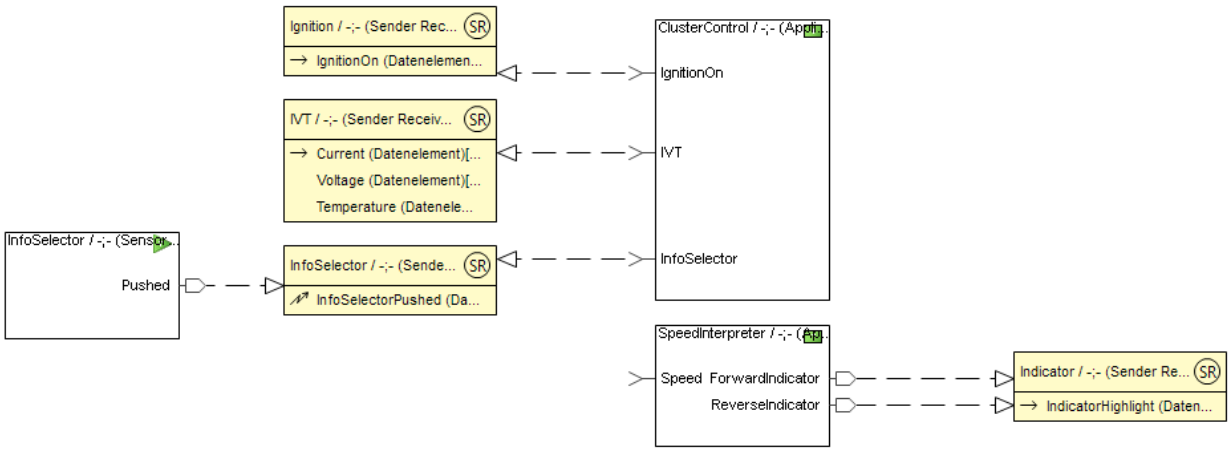


Abbildung 3.17: Graphische Repräsentation Funktionstypen [ELY11]

Analog zur graphischen Repräsentation der Blockinstanzen werden atomare Softwarekomponenten- und Kompositionstypen als Block dargestellt, an den Rändern der Blöcke werden die Porttypen platziert. Die Portprototypen werden über InterfaceAssignments (dargestellt als gestrichelter Pfeil) einem Interface zugeordnet, welches wiederum Datenelemente und Operationen beinhalten kann. Wegen ihrer Analogie zur UML (siehe auch Kapitel 2.3) lehnt sich die graphische Notation an die der UML an.

3.3.4 Signalmodell

Signale spielen in einer vernetzten E/E-Architektur eine wichtige Rolle. Ein Signal wird dabei als Träger einer Information betrachtet und von einem Sender zu einem Empfänger verschickt. Die Information wird über die Zuordnung einer Informationseinheit aus der Funktionsbibliothek (siehe 3.3.3) zu dem Signal spezifiziert. Dieser Mechanismus wird in Kapitel 3.3.7.3 näher beschrieben. Signale können in sogenannten Frames zusammengefasst werden. Diese implementieren Teile des Protokolls des Übertragungsmediums und sind somit realisierungsabhängig. Signale, die strukturell unabhängig, jedoch semantisch abhängig voneinander sind, werden in PDU's (PDU-Protocoll Data Unit) zusammengefasst.

Auf dem Weg vom Sender zum Empfänger kann ein Signal mehrere Zwischenstationen, wie zum Beispiel Gateways, A/D-Wandler oder D/A-Wandler, durchlaufen. Dabei können unterschiedliche Übertragungsmedien zum Einsatz kommen, auf denen das Signal in verschiedenen Repräsentatio-

nen verschickt wird. Dieser Sachverhalt wird durch ein Typ-Instanz-Konzept (siehe 2.5.2) beschrieben. Dabei bildet das Signal den Typ, die hardwareabhängige Instanz wird durch eine sogenannte Signaltransmission modelliert.

Bei der protokollbasierten Kommunikation über Bussysteme werden Signaltransmissionen in sogenannten Frametransmissionen zusammengefasst und versendet, die wiederum als Instanz eines Frame zu verstehen sind.

3.3.4.1 Metamodell

Die wesentlichen Metaklassen im Signalmodell (siehe Abbildung 3.18) sind *Signal* und *Frame*, welche als Typen von *SignalTransmission* und *FrameTransmission* definiert sind. Die Komposition zur *SignalConnection*, welches konkret eine konventionelle Verbindung oder ein Bussystem sein kann, repräsentiert die Referenz zum Träger der jeweiligen Transmission.

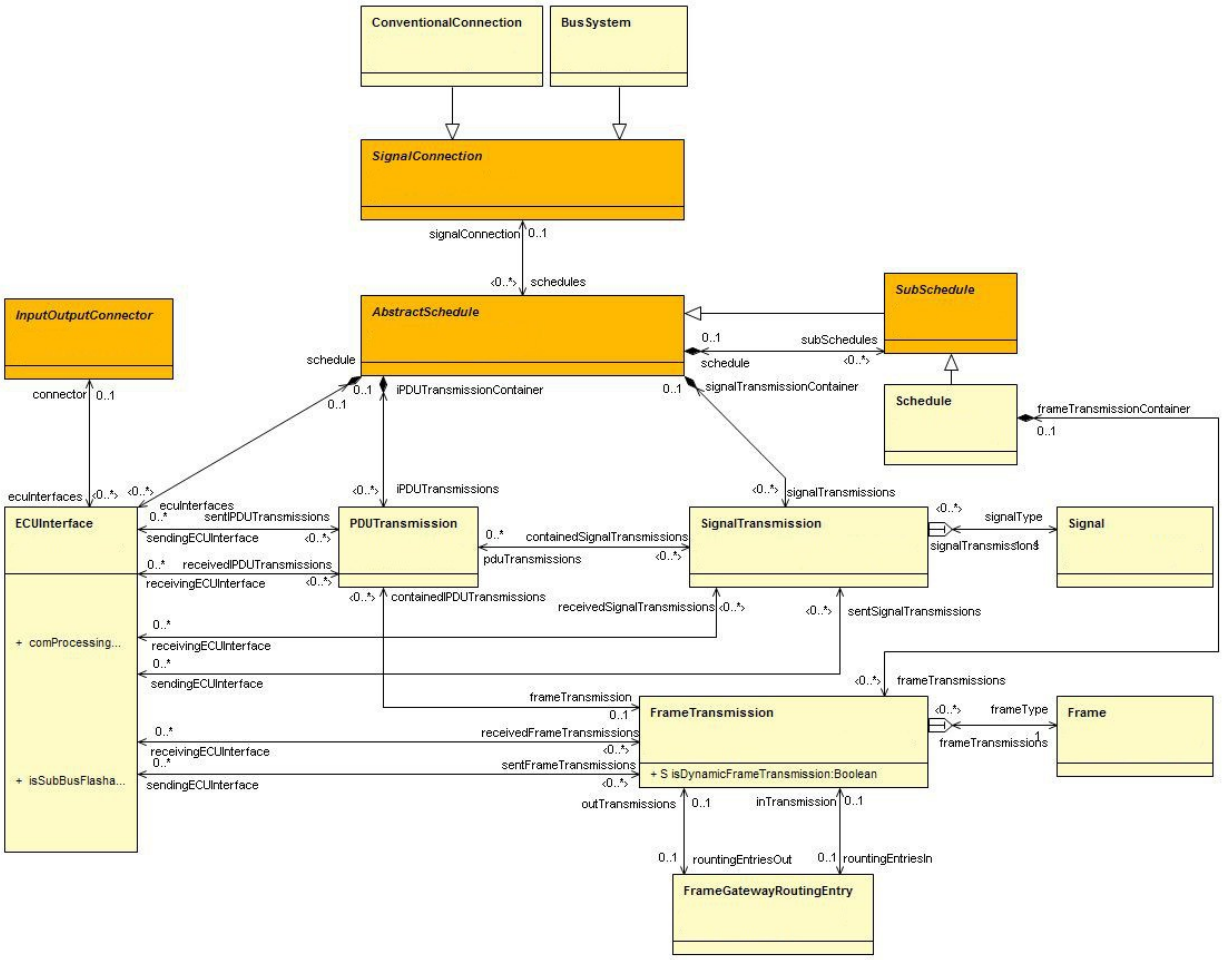


Abbildung 3.18: Signal- und Frameübertragung

SignalTransmissionen können durch die entsprechende Assoziation in *FrameTransmissionen*

zusammengefasst werden.

Semantisch nicht unabhängige Informationen werden durch das in Abbildung 3.19 gezeigte Konzept für Protocol Data Units (PDU) zusammengefasst. Dazu werden *Signale* durch *SignalIPDUAssignments* einer *SignalIPDU* zugewiesen. Die Assoziation zwischen einer *PDU* (von der *SignalIPDU* erbt) und *PDUFrameAssignment*, modelliert die Zuweisung der *PDU* zu einem *Frame*.

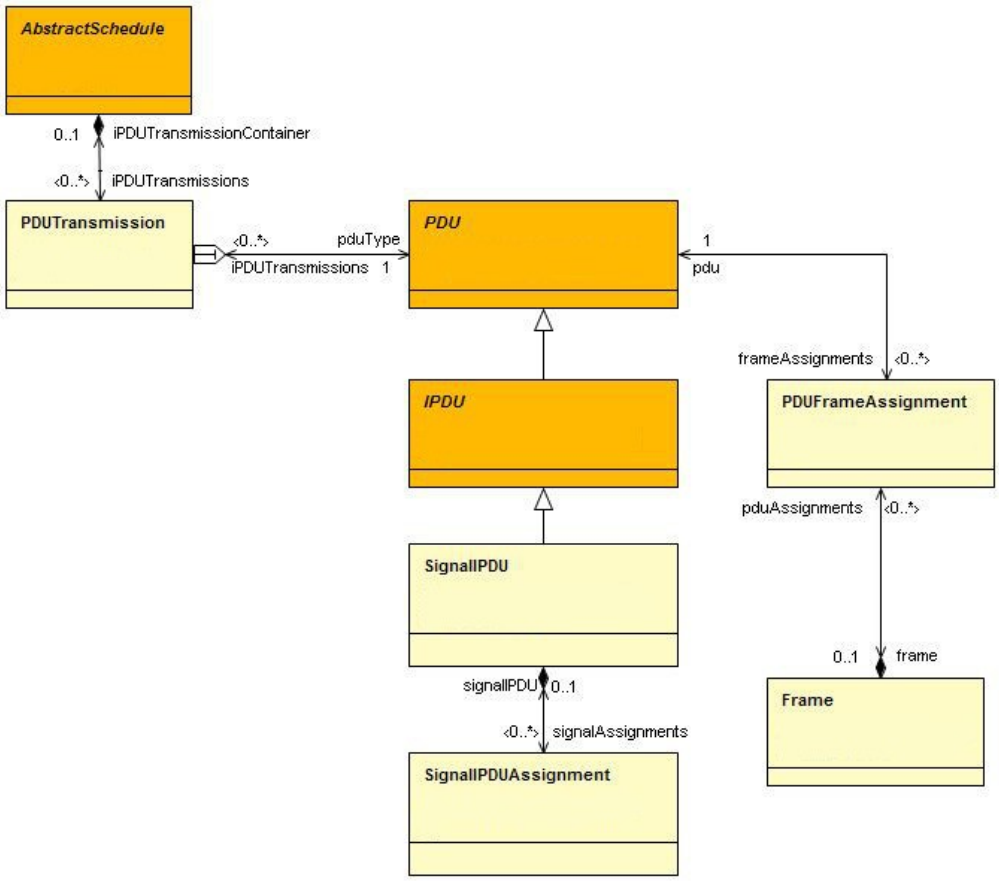


Abbildung 3.19: Protocol Data Unit (PDU)

3.3.5 Hardwareebene

Komponenten repräsentieren Aktoren, Sensoren, Steuergeräte und Sicherungsdosen in der E/E-Architektur. Deren innerer Aufbau wird ebenso beschrieben, wie die logischen Anbindungen, an die Verbindungen der logischen Vernetzung (siehe 3.3.5.1), des Stromlaufplans (siehe 3.3.5.2) und aus dem Leitungssatz (siehe 3.3.5.3) angeschlossen werden. Diese verschiedenen Verbindungsarten beschreiben die Vernetzung der Komponenten in unterschiedlichen Ansichtsabstraktionen, die in den nachfolgenden Abschnitten näher beschrieben sind.

Die innere Beschreibung von Komponenten dient vor allem zur Erfassung der wichtigsten Eck-

daten. Dazu gehören physikalische, elektrische und monetäre Größen. Zu den Artefakten für die innere Beschreibung zählen Recheneinheiten (CPU, FPGA, ASIC, etc.) zur Umsetzung von Softwarefunktionen. Hardwaremodule dienen zur Beschreibung von verbauten Hardwareelementen, die zur analogen Umsetzung einer Funktion benötigt werden. Speicherbausteine bilden den zur Verfügung stehenden Speicher ab. Anbindungen stellen Konnektivität zu Bussystemen und anderen logischen Verbindungen (siehe 3.3.5.1) her und repräsentieren entsprechende Treiber- und Transceiverbausteine. Komponenten können aus anderen komplexen Komponenten zusammengesetzt sein, so dass sich komplexe Einheiten beschreiben lassen.

Zur Beschreibung kleiner, aber trotzdem wesentlicher Bauteile wie Widerstände, Sicherungen, Relais oder Dioden stehen generische Hardwareelemente mit konfigurierbaren grafischen Repräsentationen zur Verfügung. Die Verschaltung der internen Bauteile erfolgt über entsprechende Verbindungen.

Die folgenden Abschnitte gehen auf die unterschiedlichen Ansichtsabstraktionen zur Beschreibung der Vernetzung von Komponenten ein. Diese sind im Einzelnen die logische Vernetzung (3.3.5.1), die schematischen Stromlaufbeschreibungen (3.3.5.2) und der Leitungssatz (3.3.5.3).

3.3.5.1 Logische Vernetzung

Die logische Vernetzung beschreibt das Netzwerk einer E/E-Architektur, welches Komponenten über logische Verbindungen miteinander verbindet. Bussysteme sind logische Verbindungen, die zur Übertragung großer Datenmengen, gesteuert durch ein Protokoll, dienen. Konventionelle Verbindungen sind logische Verbindungen zur Übertragung von analogen Daten wie Spannungspegel oder Ströme. Logische Verbindungen zur Leistungsversorgung werden zur Beschreibung der elektrischen Leistungsverteilung verwendet. Masseverbindungen bilden die vierte Gruppe der logischen Verbindungen und repräsentieren den Anschluss der Komponenten an die elektrische Fahrzeugmasse.

3.3.5.2 Schematische Stromlaufbeschreibungen

Stromlaufpläne detaillieren den logischen Vernetzungsentwurf aus der Komponenten- und Netzwerkebene (siehe Abschnitt 3.3.5) derart, dass die modellierten logischen Verbindungen physikalisch durch eine oder mehrere elektrische Verbindungen repräsentiert werden. Diese elektrischen Verbindungen werden über schematische Pins an die jeweiligen Anbindungen aus der Komponentenebene angeschlossen. Sie verbinden Komponenten unterbrechungsfrei, die Start- und Zielpunkte

befinden sich auf identischem, elektrischem Potential.

3.3.5.3 Leitungssatz

Im Leitungssatz werden Leitungen und Kabel, die wiederum aus Adern und Kabeln bestehen können, beschrieben. Eine Leitung oder ein Kabel stellt eine Punkt zu Punkt Verbindung zwischen Komponenten, Trennstellen oder Splices dar. Leitungen werden über Leitungssatz-seitige Pins, welche Bestandteil eines schematischen Pins sind, mit Komponenten verbunden.

Leitungen, die über Trennstellen oder Splices zusammenhängen und elektrisch auf einem Potential liegen, werden durch elektrische Verbindungen in Stromlaufplänen (siehe 3.3.5.2) abstrahiert.

3.3.5.4 Metamodell

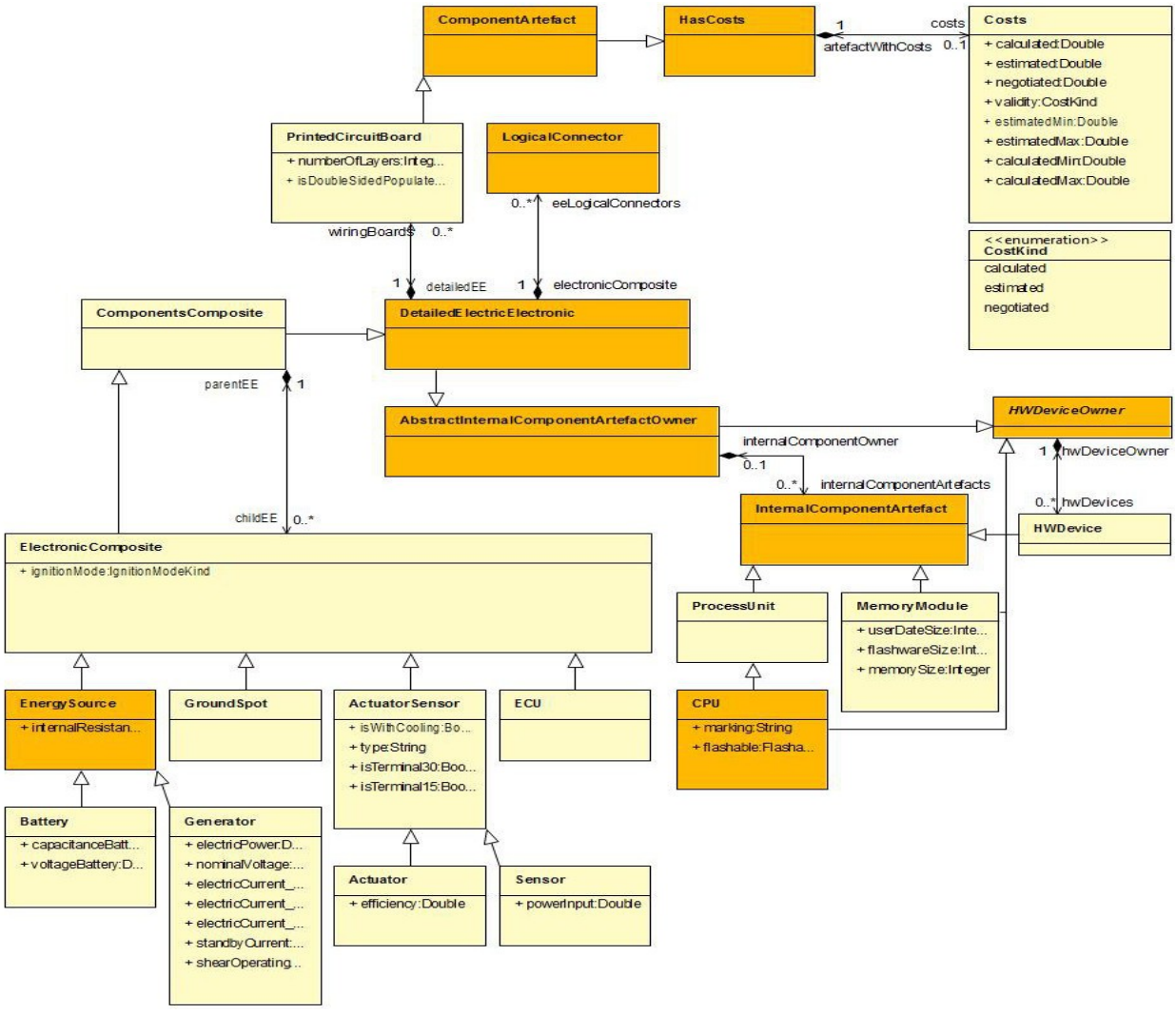


Abbildung 3.20: Detaillierter Aufbau von EE-Komponenten

Wichtigster Bestandteil des Metamodells der Hardwareebene sind die Hardware-Komponenten.

Diese verfügen über einen internen und einen externen Aufbau. Es gibt zahlreiche unterschiedliche Arten von E/E-Komponenten, im Metamodell erben sie alle von der abstrakten Metaklasse *DetailedElectricElectronic* (siehe Abbildung 3.20). Über das Kompositum-Entwurfsmuster (Kapitel 2.5.1) zwischen *ComponentsComposite* und *ElectronicComposite* werden zusammengesetzte E/E-Komponenten modelliert. Von ElectronicComposite erben die spezifischen Komponentenarten, wie z.B. ECU, Sensoren und Aktoren.

Der interne Aufbau von Komponenten besteht aus Leiterplatten (*PrintedCircuitBoard*), auf denen Bestandteile wie CPU (Subklasse von *ProcessUnit*), Speichermodule (*MemoryModule*) und andere, spezifische Hardwareelemente wie Sicherungen, Relais, Spulen, etc. (*HWDevice*). Der interne Aufbau wird über die Komposition zwischen *AbstractInternalComponentArtefactOwner* (Superklasse von *DetailedElectricElectronic*) und *InternalComponentArtefact* realisiert.

Der Anschluss von Komponenten an Verbindungssysteme wie Bussysteme, konventionelle Verbindungen oder Energieversorgung erfolgt durch logische Anbindungen (*LogicalConnector*). Diese Zusammenhänge werden in Abbildung 3.21 detailliert gezeigt.

Im oberen Teil sind die externen Verbindungssysteme (als Subklassen von LogicalConnection) dargestellt. Dies können Bussysteme (*Bussystem*) und konventionelle Verbindungen (*ConventionalConnection*) als Signalverbindungen (*SignalConnection*) sowie Energie- und Masseverbindungen (*PowerConnection* bzw. *GroundConnection*) als Versorgungsverbindungen (*SupplyConnection*) sein. Bei Bussystemen wird die konkrete Art des Busses über ein Typ-Instanz-Konzept zu einem dedizierten Bustyp-Artefakt (z.B. *CANType*, *LINType*, etc.) mit Medium-spezifischen Attributen modelliert (nicht im Klassendiagramm dargestellt).

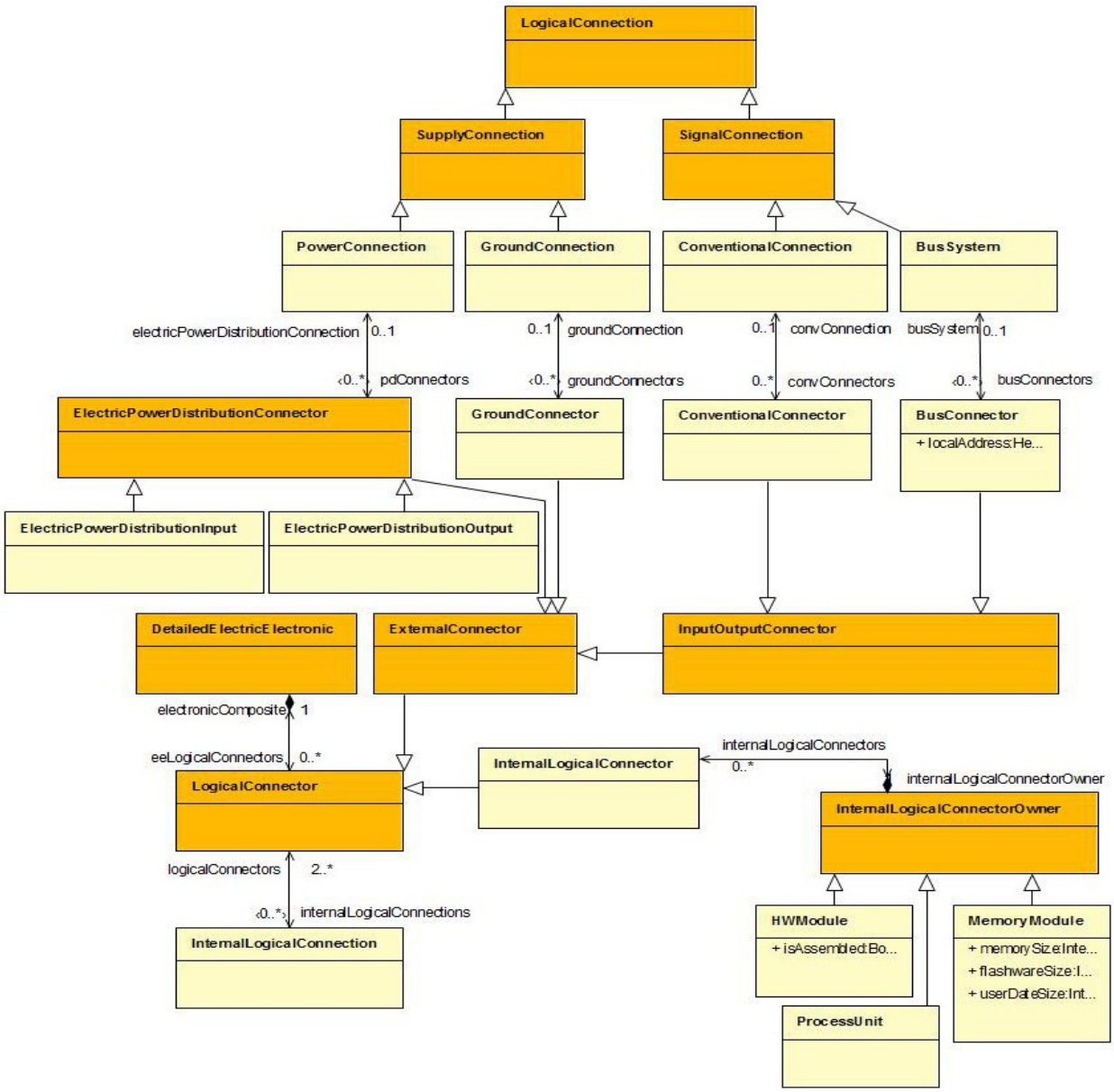


Abbildung 3.21: Logische Verbindungssysteme der Hardware-Ebene

Für jede dieser Verbindungssysteme gibt es dedizierte Anbindungen, z.B. *BusConnector* für Bussysteme. Diese sind in der Betrachtung der logischen Vernetzung bidirektional. Als Ausnahme werden Anbindungen der Energieversorgung in Ein- und Ausgang verfeinert (*ElectricPowerDistributionInput* und *ElectricPowerDistributionOutput*).

Die Weiterführung von logischen Verbindungen zu dem internen Aufbau der Komponenten erfolgt über interne logische Verbindungen (*InternalLogicalConnection*), welche über interne logische Anbindungen (*InternalLogicalConnector*) angeschlossen werden. Dieser Zusammenhang ist im unteren Teil der Abbildung 3.21 dargestellt.

Während die Darstellung der Komponenten in den drei Ansichtsabstraktionen weitgehend identisch bleibt (z.T. wird der interne Aufbau nicht dargestellt), ändert sich die Modellierung der Verbindungen und der passenden Anschlusskonzepte. Während die logische Sicht die vorhandenen Komponentenverbindungen auf Bussysteme, konventionelle Verbindungen, Massekonzept und Stromversorgung abstrahiert darstellt, werden im Stromlaufplan die Verbindungen durch ihre einzelnen, elektrischen Verbindungen detaillierter dargestellt.

Im Metamodell wird dieser Zusammenhang durch die schematischen Verbindungen und Pins beschrieben. Die schematische Verbindung (*SchematicConnection*) repräsentiert eine logische Verbindung, in Abbildung 3.22 dargestellt als Assoziation zwischen, *SchematicConnection* und *RepresentigLogicalLayerArtefact* (einer abstrakten Superklasse aller logischen Verbindungen).

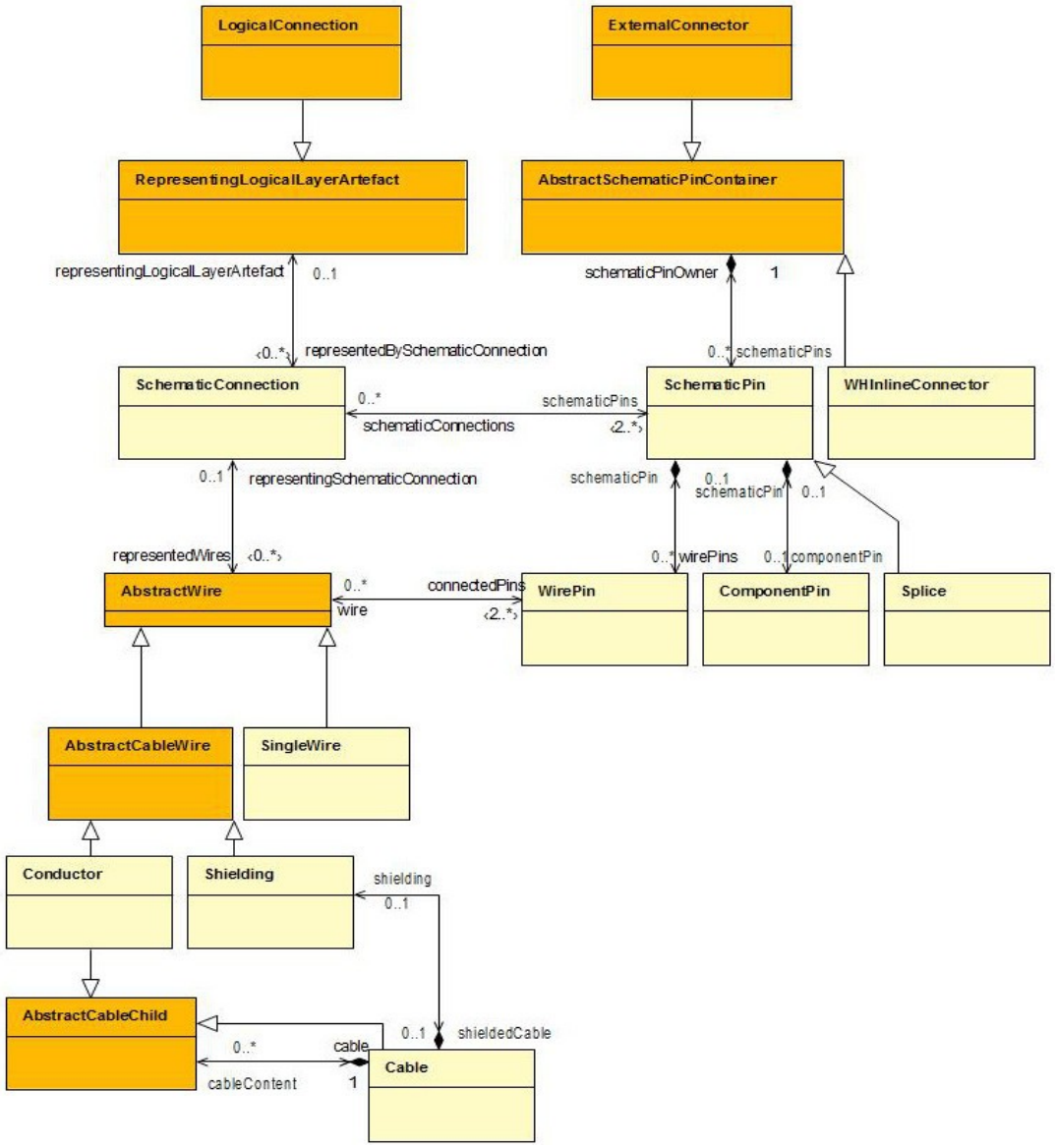


Abbildung 3.22: Schematische Verbindungen und Leitungssatz

Die schematische Verbindung schließt Komponenten elektrisch zusammen. Dies wird über schematische Pins (*SchematicPin*) modelliert. Der Bezug zur Komponente oder zugehörigem Bauteil erschließt sich über die Kompositionsbeziehung zu *AbstractSchematicPinContainer*. Dies ist eine abstrakte Superklasse, von der die Trennstelle im Leitungssatz (*WHInlineConnector*) und die externe logische Anbindung (*ExternalConnector*) erben.

Eine weitere Detaillierung erfolgt über die Beschreibung des Leitungssatzes. In dieser Abstraktionsansicht der Hardwareebene werden die elektrischen Verbindungen aufgeschlüsselt in einzelne Leitungen, Stützpunkten im Leitungssatz (Splice) und Trennstellen. Dazu ist die schematische Verbindung im Metamodell mit der Klasse *AbstractWire* assoziiert. Diese abstrakte Darstellung einer Lei-

tung wird spezialisiert als Einzelleitung (*SingleWire*) oder als Kabel, welches wiederum über Adern (*Conductor*) und optionalen Schirmungen (*Shielding*) zusammengesetzt wird.

Die schematischen Pins werden weiter verfeinert durch Leitungssatz-seitige Pins zum Anschluss von Leitungen (*WirePin* mit Assoziation zu *AbstractWire*) und durch Komponenten-seitige Pins. Darüber hinaus werden Leitungssatz-seitige und Komponenten-seitige Pins Steckern zugeordnet (nicht in Abbildung dargestellt).

Vor allem im Leitungssatz werden die wesentlichen Artefakte typisiert (siehe Abschnitt 2.5.2), um gemeinsame Eigenschaften zu beschreiben. Neben einfachen Eigenschaften wie zum Beispiel Farbgebungen und Kosten pro Meter sind es auch komplexe Eigenschaften wie Befestigungsclips und Dichtungsarretierungen. Diese Informationen sind einerseits wichtige Kostentreiber im Leitungssatz und andererseits relevant zur Erstellung von Stücklisten.

3.3.5.5 Graphische Repräsentation

Die graphische Repräsentation der Hardware-Ebene erfolgt prinzipiell durch Blockschaltbilder. In der logischen Vernetzung werden die Komponenten (durch blaue, rechteckige Blöcke dargestellt), wie in Abbildung 3.23 dargestellt, durch eine einzelne graphische Verbindung, welche das Verbindungssystem (*BodyCAN:LowSpeedCAN*) visualisiert, miteinander verknüpft.

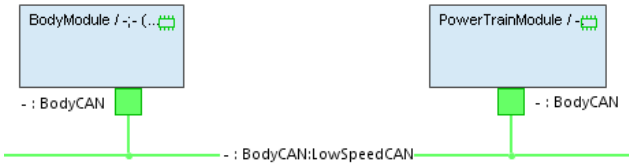


Abbildung 3.23: Graphische Repräsentation logische Vernetzung

Wie bereits beschrieben, erfolgt die Verknüpfung an das Übertragungsmedium indirekt durch eine Anbindung (*BodyCAN*), welche graphisch durch kleine Quadrate visualisiert werden.

Mit dieser abstrahierten Sicht auf die Hardwarebausteine einer komplexen E/E-Architektur lassen sich verständliche Darstellungen und Übersichten verwirklichen. Zusätzliche Informationen, z.B. zu Ausstattungsvarianten, lassen sich optional durch graphische Effekte wie Blockschatten oder Farblegenden konfigurieren.

Im Stromlaufplan bleibt die Darstellung der Komponenten erhalten, wie in Abbildung 3.24 gezeigt. Gegebenenfalls werden hier zusätzlich wesentliche beinhalteten Bausteine, wie zum Beispiel Sicherungen oder Relais, und ihre interne Verschaltung angezeigt (nicht dargestellt). Die CAN-Busver-

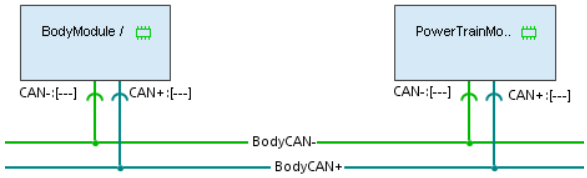


Abbildung 3.24: Graphische Repräsentation Stromlaufplan

bindung aus der logischen Vernetzung wird in diesem Beispiel durch zwei elektrische Verbindungen konkretisiert. Die Busanbindung zur Verknüpfung der Komponenten an das Übertragungsmedium wird verfeinert durch schematische Pins für die jeweiligen elektrischen Verbindungen.

Mit dieser graphischen Repräsentation lassen sich komplexe Stromlaufpläne der E/E-Architektur visualisieren. Vor allem die Anschlüsse von Komponenten an die unterschiedlichen Versorgungsverbindungen (die bekanntesten hier sind Klemme 15 für Zündung ein und Klemme 30 für Dauerversorgung über Batterie) und dich Absicherungen der elektrischen Verbindungen in Sicherungs- und Relaisboxen sind hier von großer Bedeutung.

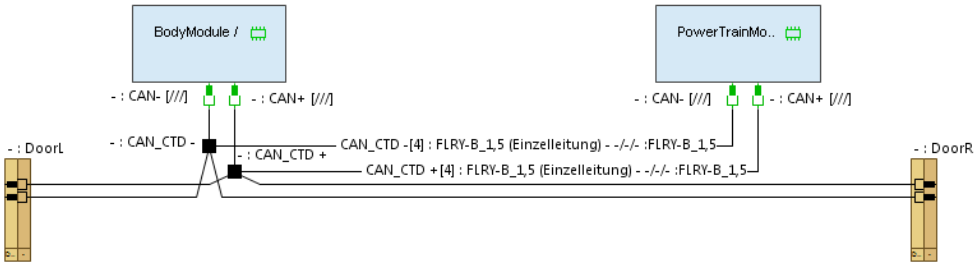


Abbildung 3.25: Graphische Repräsentation Leitungssatz

Im Leitungssatzdiagramm bleibt die Darstellung der Komponenten unverändert. Die elektrischen Verbindungen aus dem Stromlaufplan werden konkretisiert durch die einzelnen Leitungen (dies können Ein), welche detaillierte Informationen über Beschaffenheit, Geometrie, Kosten, Aussehen etc. tragen. Stützpunkte (Splices, in Abbildung 3.25 dargestellt durch schwarz ausgefüllte Quadrate) zum Anschluss von mehreren Leitungen an dasselbe Potential werden ebenso visualisiert wie Trennstellen (in Abbildung 3.25 dargestellt durch zweiteilige braune Rechtecke), an denen der Leitungssatz durch ein Steckerpaar aufgetrennt wird.

Die schematischen Pins aus dem Stromlaufplan werden durch ein Pinpaar, bestehend aus einem leitungssatz- und komponentenseitigen Pin, visualisiert. Brücken (direkte Verbindung mehrerer leitungssatzseitiger Pins) und Mehrfachanschläge (ein leitungssatzseitiger Pin ist mit mehreren Leitun-

gen verbunden) werden ebenfalls im Leitungssatzdiagramm visualisiert (nicht in Abbildung 3.25 dargestellt).

Durch die vielen Einzelleitungen und Bauteile eines Leitungssatzes werden die graphischen Repräsentationen sehr schnell sehr komplex. Leitungssatzdiagramme eignen sich daher vor allem zur Visualisierung von Ausschnitten oder Subsystemen einer E/E-Architektur.

3.3.6 Physikalische Topologie

Die physikalische Topologie beschreibt mögliche Einbauorte für Komponenten, mögliche Ausbindungen für Leitungen sowie deren Verlegemöglichkeiten in Form von Topologiesegmenten. Über Topologietrennstellen werden Punkte in der Fahrzeugtopologie beschrieben, an denen durchgehende Leitungen aufgetrennt werden müssen. Dies ist zum Beispiel an den Verbindungsstellen für Türen der Fall. Bauräume beschreiben Räume im Fahrzeug mit gleichen oder ähnlichen Bedingungen wie Temperatur oder Feuchtigkeit für die darin enthaltenen Einbauorte, Ausbindungen und Trennstellen.

3.3.6.1 Metamodell

Im wesentlichen besteht die geometrische Topologie einer E/E-Architektur aus Bauräumen (*InstallationSpace*), Einbauorten (*InstallationLocation*), Ausbindungen (*BranchOff*), Trennstellen (*InlineConnector*) und verbindenden Segmenten (*TopologySegment*). Der passende Metamodellausschnitt ist in Abbildung 3.26 dargestellt.

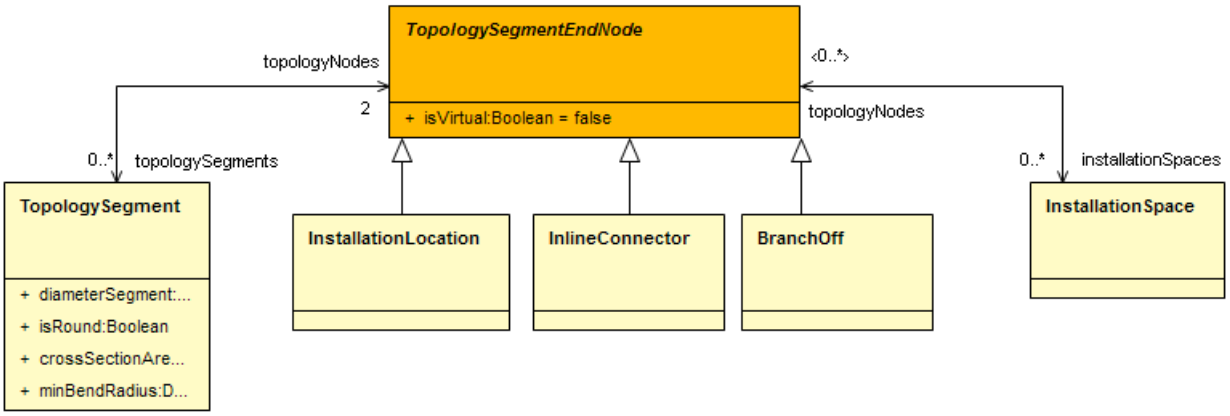


Abbildung 3.26: Geometrische Topologie

Einbauorte repräsentieren im Fahrzeug einen Bereich, in welchem Komponenten (z.B. Steuergeräte, Aktoren oder Sensoren) platziert werden können. Einbauorte werden durch Segmente miteinander

verbunden. Ein Segment ist ein Pfad im Fahrzeug, in welchem Leitungen oder Kabel verlegt werden können. An Stellen, an welchen der Leitungssatz durch ein Steckerpaar unterbrochen werden muss, werden Trennstellen platziert. Dies ist zum Beispiel an den Übergängen der Karosserie zu den Türen der Fall. Ausbindungen sind Stellen im Fahrzeug, an denen der Leitungssatz aufgetrennt werden kann.

Im Metamodell wird erben die Metaklassen für Einbauort (*InstallationLocation*), Trennstelle (*InlineConnector*) und Ausbindung (*BranchOff*) von der abstrakten Metaklasse *TopologySegmentEndNode* welche einen Endknoten für Segmente repräsentiert. Zwei Endknoten können über die Assoziation zum Segment (*TopologySegment*) miteinander verbunden werden.

Zudem können Endknoten über die Assoziation zum Bauraum (*InstallationSpace*) zu größeren Einheiten mit gemeinsamen physikalischen Randbedingungen gruppiert werden. Ein Beispiel hierfür ist der Motorraum.

Mit dieser einfachen Struktur lassen sich die komplexen Netze der geometrischen Topologie einer E/E-Architektur abbilden. Über die Attribute der Artefakte, die wichtigsten sind in Abbildung 3.27 dargestellt, lassen sich die geometrischen Maße und Umgebungsbedingungen modellieren.

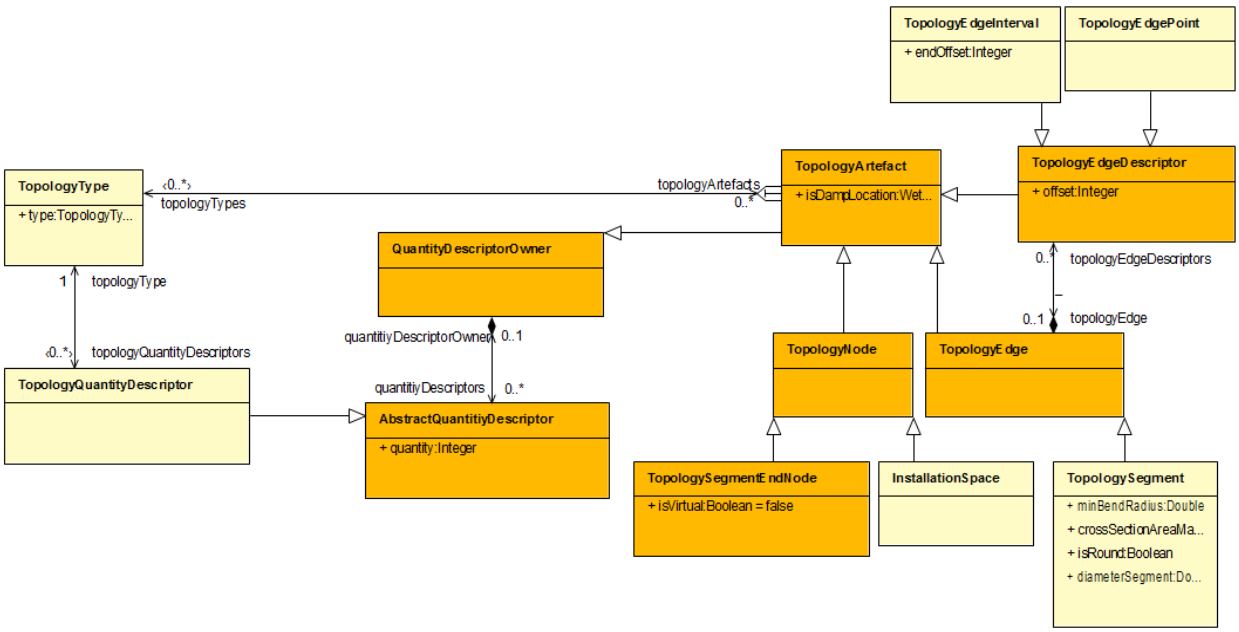


Abbildung 3.27: Attribute und Anbauteile der geometrischen Topologie

Ein wesentlicher Bestandteil für geometrische Topologien sind die Anbauteile der Topologie, da sie einen nicht unerheblichen Kostenfaktor darstellen. Diese Anbauteile sind zum Beispiel Kabelkanä-

le, Klipse, Tapes und Tüllen zur Fixierung und Absicherung von Leitungen und Kabeln, oder auch Halterungen für Komponenten in Einbauorten etc. Diese Anbauteile werden durch Typartefakte (*TopologyType*) definiert, indem das type-Attribut die Art des Anbauteils beschreibt und die anderen Informationen wie Sachnummer und zulässige Einsatztemperaturen in den jeweiligen Attributen abgelegt werden. Über die typeOf-Assoziation wird der Typ auf die Instanzen (*TopologyArtefact* mit den indirekten Subklassen wie *InstallationLocation*, *TopologySegment* etc.) appliziert.

Einige Arten der Anbauteile, wie zum Beispiel Klipse und Tüllen, befestigen Leitungen punktuell. Deren Position auf dem Topologiesegment wird über das Deskriptor-Artefakt *TopologyEdgePoint* angegeben. Andere Arten der Anbauteile, wie zum Beispiel Kabelkanäle und Tapes, beziehen sich auf ein Intervall von Segmenten, welches über das Deskriptor-Artefakt *TopologyEdgeInterval* spezifiziert wird.

3.3.6.2 Graphische Repräsentation

Die geometrische Topologie einer E/E-Architektur lässt sich grundsätzlich in zwei Arten darstellen. Die naheliegende ist die dreidimensionale Darstellung, wie sie in gängigen 3-D-Konstruktionswerkzeugen bekannt ist. Diese Darstellungsform hebt die räumlichen Zusammenhänge der Elemente einer geometrischen Topologie hervor und fördert das Verständnis komplexer Strukturen und Details. Allerdings werden im Allgemeinen immer nur zweidimensionale Projektionen des dreidimensionalen Modells (von verschiedenen Standorten aus gesehen) dargestellt, da die meisten Visualisierungsmedien (Bildschirm, Papier, etc.) zweidimensionale Darstellungsmöglichkeiten besitzen.

Die zweidimensionale Darstellung der geometrischen Topologie müssen einerseits Abstraktionen der Darstellung in Kauf (Verlust einer Dimension) genommen werden, dafür können aber die geometrischen Netze sehr übersichtlich dargestellt werden. In Abbildung 3.28 ist eine zweidimensionale Darstellung einer geometrischen Topologie zu sehen. Die Ansicht repräsentiert im Prinzip eine Draufsicht auf das Fahrzeug. Das heißt, dass die Längen- und Breiteninformationen dargestellt werden, während unterschiedliche Höhen von Artefakten nicht zu erkennen sind. Die Koordinaten der einzelnen Artefakte spiegeln in diesem Fall aber nicht die tatsächlichen Koordinaten im Fahrzeug wider, sondern wurden manuell so angeordnet, um die prinzipielle Struktur der Topologie zu visualisieren.

Bauräume (*InstallationSpace*) werden als rote Vielecke repräsentiert und visualisieren Bereiche im Fahrzeug mit gemeinsamen Umgebungswerten, wie zum Beispiel den Motorraum, den Innenraum (dort nochmals in die Sitze unterteilt), den Kofferraum, etc. Einbauorte werden als schwarze Viere-

cke dargestellt. In ihnen können Komponenten platziert werden. Ausbindungen werden als rote, ausgefüllte Kreise gezeichnet. Die roten Linien im Diagramm repräsentieren die Segmente, welche Pfade im Fahrzeug repräsentieren, in denen Leitungen und Kabel verlegt werden können. In Abbildung 3.28 sind die Einbauorte beschriftet (für die anderen Artefakte sind die Beschriftungen ausgeblendet). Auf den Segmenten sind Zahlen abgebildet. Diese gibt die Länge des Segmentes in Millimetern wieder, diese Angabe ist sehr wichtig, da sich im Allgemeinen die Kosten der Leitungen und Kabeln durch einen Fixanteil und einem längenabhängigen Teil zusammensetzen.

Trennstellen werden als grau ausgefüllte Rechtecke visualisiert. Sie zeigen Stellen in der Topologie, an denen ein im Fahrzeug verbauter Leitungssatz aus topologischer Sicht aufgetrennt werden muss. Dies ist beispielsweise an Türen und Sitzen der Fall, damit diese Teile bei Bedarf abmontiert werden können.

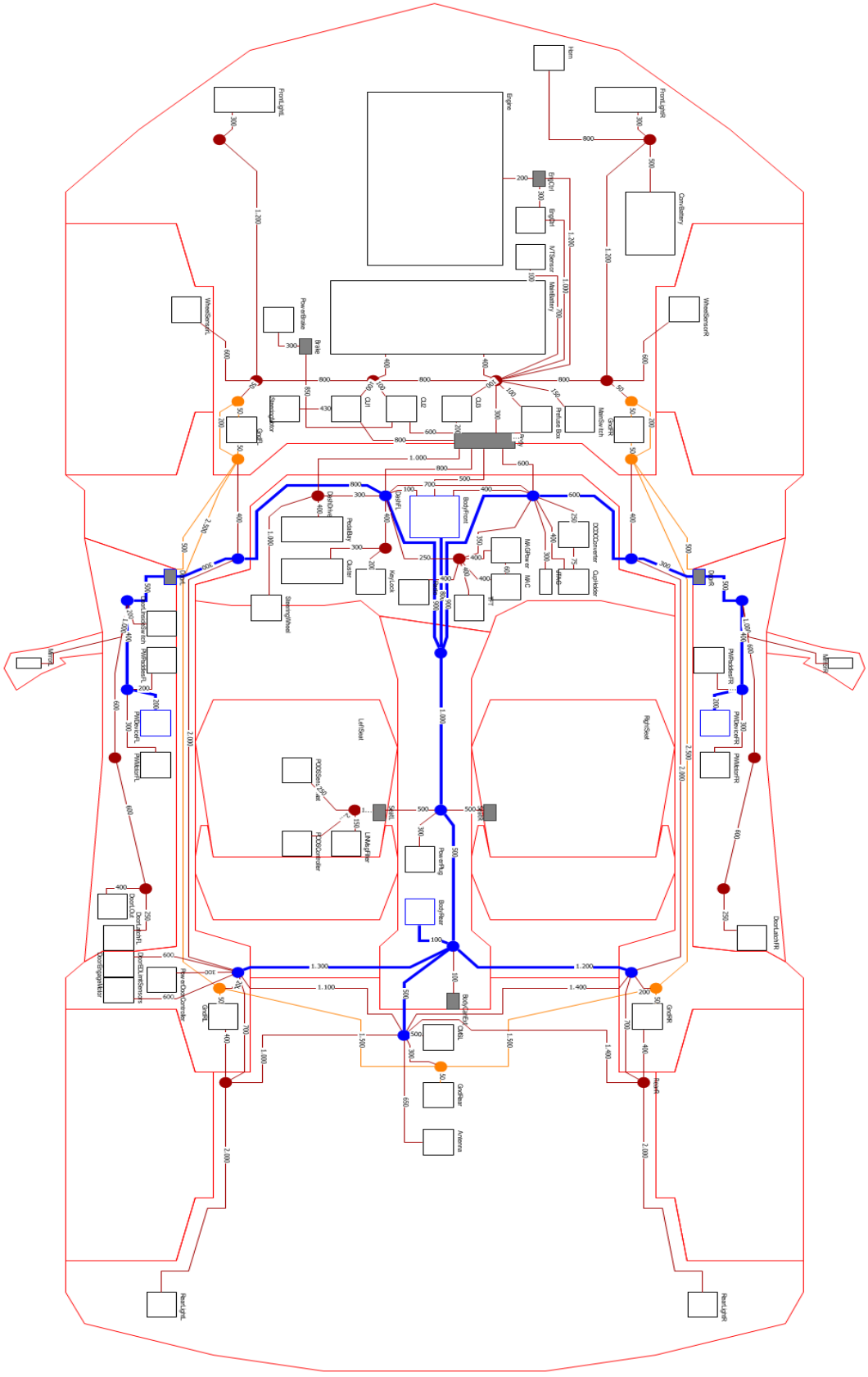


Abbildung 3.28: Graphische Repräsentation geometrische Topologie [ELY11]

Einige der in Abbildung 3.28 gezeigten Segmente sind dunkelblau eingefärbt und breiter gezeichnet. Diese Segmente visualisieren den Verlegepfad aus dem Vorgängerkapitel 3.3.5 gezeigten Bus BodyCAN. Dessen Leitungen sind über Mappingartefakte mit den Segmenten verknüpft. Mappingartefakte stellen ebenenübergreifende Zusammenhänge dar und werden im Folgekapitel genauer vorgestellt.

3.3.7 Ebenenübergreifende Zusammenhänge

Die in den Abschnitten 3.3.1 bis 3.3.6 vorgestellten Abstraktionsebenen werden über ebenenübergreifende Zusammenhänge, sogenannte Mappings, miteinander verknüpft. Das entsprechende Entwurfsmuster wird in 2.5.4 detailliert beschrieben. Die resultierende Semantik eines Mappings bezieht sich immer auf den konkreten Anwendungsfall. Im Folgenden werden die konkreten ebenenübergreifenden Zusammenhänge vorgestellt.

3.3.7.1 Allgemeines Mapping der Anforderungsebene

Das allgemeine Mapping (siehe Kapitel 2.5.4) der Anforderungsebene ist generisch formuliert und erlaubt die Verknüpfung einer Anforderung mit Artefakten der E/E-Architektur.

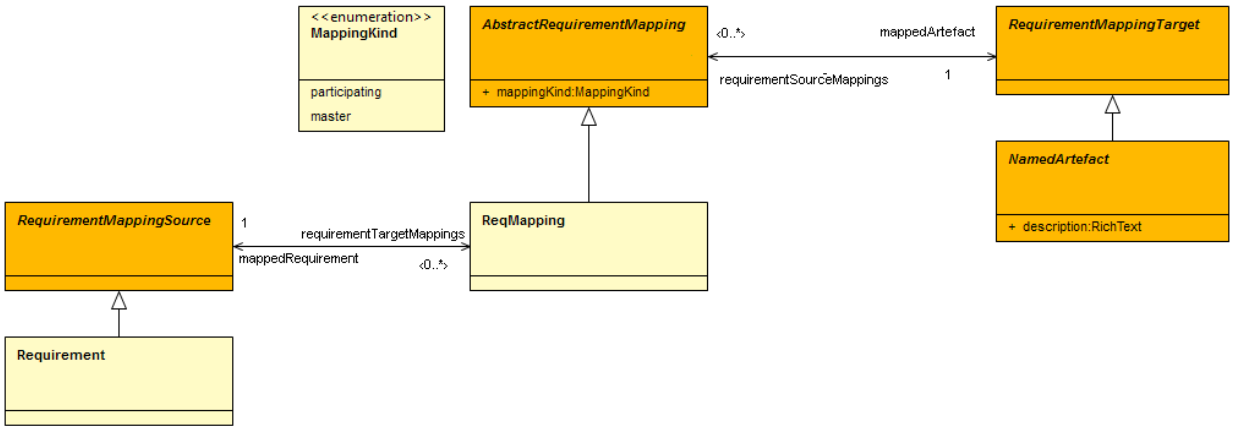


Abbildung 3.29: Allgemeines Mapping der Anforderungsebene

Wie in Abbildung 3.29 dargestellt, wird das Mapping-Artefakt durch die Klasse *ReqMapping* repräsentiert. Sie ist assoziiert mit der Klasse *RequirementMappingSource*, was eine Superklasse von *Requirement* ist. Über die abstrakte Superklasse *AbstractRequirementMapping* ist das Mapping-Artefakt mit der Klasse *RequirementMappingTarget* assoziiert, welche das Ziel des Mappings darstellt. Eine Subklasse von diesem Mappingziel ist *NamedArtefact*, was wiederum eine (indirekte)

Superklasse fast[12] aller Artefakte ist.

Das allgemeine Mapping der Anforderungsebene wird verwendet, um Anforderungen (dies können zum Beispiel funktionale oder Nicht-funktionale Anforderungen, technische Randbedingungen oder gesetzliche Vorgaben sein) direkt mit den betroffenen Artefakten zu verknüpfen.

3.3.7.2 Anforderungsebene – System Software Architektur

Ein Mapping von einer Anforderung in die Software-Ebene bedeutet, dass die Anforderung durch die referenzierte Artefakte der Software-Ebene umgesetzt wird.

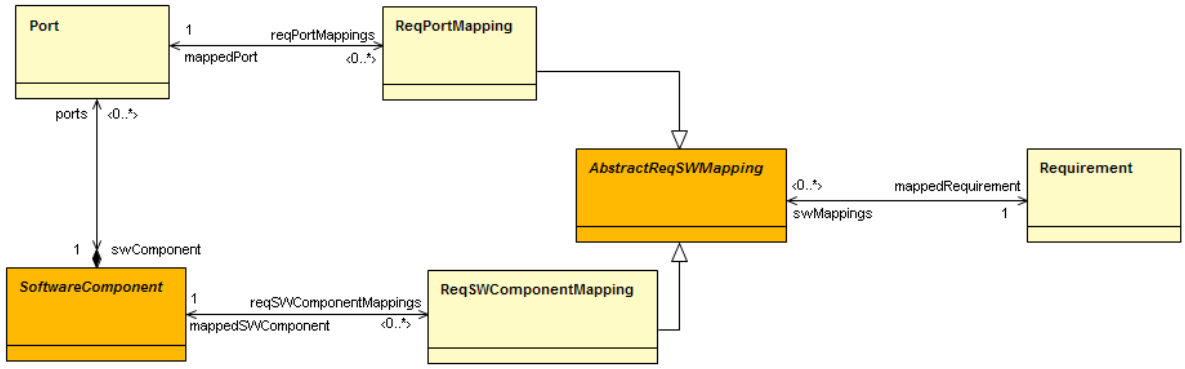


Abbildung 3.30: Mapping zwischen Anforderungs- und Funktionsnetzebene

In Abbildung 3.30 ist der entsprechende Ausschnitt im Metamodell abgebildet. Eine Anforderung (*Requirement*) wird über das Mapping-Artefakt *ReqSWComponentMapping* auf eine Software-Komponente (*SoftwareComponent*) gemappt. Dieses Mapping sagt aus, dass die referenzierte Software-Komponente für die Umsetzung der Anforderung verantwortlich ist. Alternativ dazu kann eine Anforderung an einen bestimmten Port einer Software-Komponente gemappt werden (über die Mapping-Klasse *ReqPortMapping*). In diesem Fall ist die Semantik, dass die Anforderung durch diesen Port der Software-Komponente umgesetzt wird. Die über die Komposition zur Software-Komponente ist diese indirekt mit der Anforderung verknüpft.

3.3.7.3 Funktionsnetzwerk – Signalmodell

Die zu übertragende Information eines Signals stammt aus der Funktionsbibliothek des Funktionsnetzwerks. Die dort spezifizierten Schnittstellenbeschreibungen für Kommunikationsports beinhalten Informationseinheiten (Datenelemente, Operationen und Operationsparameter), welche auf die übertragenden Signale gemappt werden. Dadurch werden dem Signal neben dem zu übertragenden

12 Gemeint sind alle fachlich relevanten Artefakte. Elemente für Modellinterne Konzepte und die Speicherung graphischer Diagramminformationen erben nicht von *NamedArtefact* und sind hiervon nicht betroffen.

Datum selbst noch weitere Informationen wie Datentyp und Einheit zugeordnet.

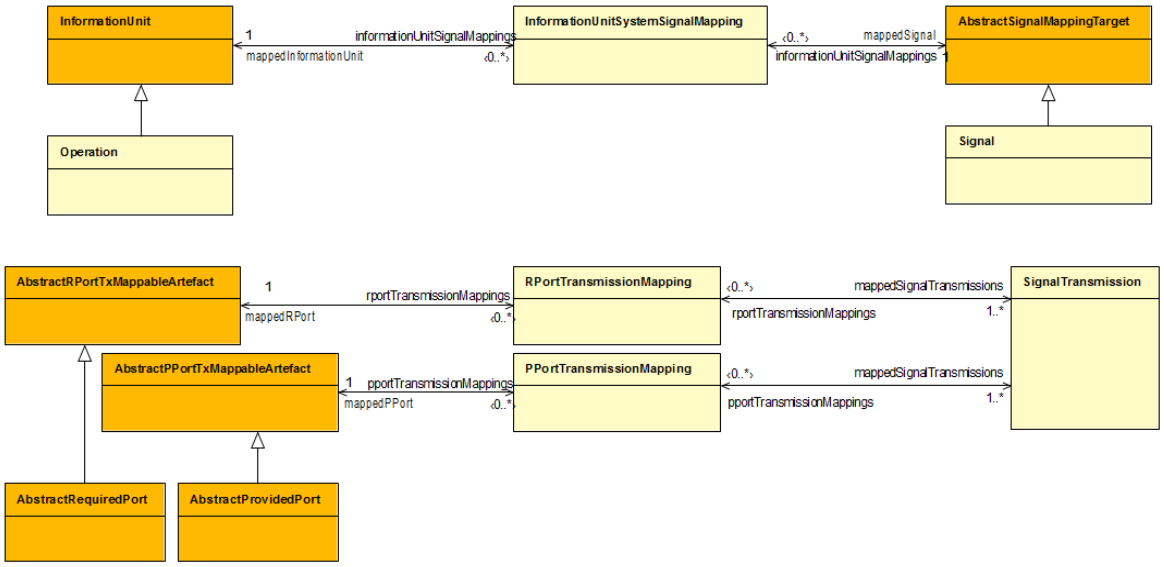


Abbildung 3.31: Mapping: Funktionsnetzwerk - Signalmodell

Abbildung 3.31 zeigt diesen Sachverhalt im Metamodell: Eine *InformationUnit* wird über eine *InformationUnitSystemSignalMapping* (über die abstrakte Klasse *AbstractSignalMapping*) einem *Signal* zugeordnet. *SignalTransmissionen* werden über entsprechende Mappings Portinstanzen (*RequiredPort*, *ProvidedPort*, wiederum über abstrakte Oberklassen) aus dem Funktionsnetzwerk zugeordnet, um Sender- und Empfängerports einer Signalübertragung eindeutig identifizieren zu können. Dieses explizite Mapping auf die Ports ist notwendig, da es Blocktypen geben kann, die über mehrere Ports verfügen, die denselben Interfaces zugeordnet sind. Dadurch wäre Ursprung oder Ziel einer *SignalTransmission* im Funktionsnetzwerk nicht mehr eindeutig identifizierbar.

3.3.7.4 Logische Architektur – System Software Architektur / Hardwareebene

Logische Blöcke, die auf andere Ebenen gemappt werden können, erben, wie in Abbildung 3.32 dargestellt, von der abstrakten Superklasse *AbstractMappableLogicalArtefact*. Die Funktionalität dieser Blöcke wird entweder in Software oder in Hardware realisiert, diese Möglichkeiten werden im Metamodell durch die folgenden Klassen abgebildet:

- *LogSWMapping*: In diesem Fall wird die Funktionalität auf Artefakte der System Software Architektur (siehe Abschnitt 3.3.3) gemappt, wo die detaillierte Modellierung erfolgt. Artefakte dieser Ebene sind auf die ausführenden Hardware gemappt (siehe Abschnitt 3.3.7.5).
- *LogicalSWNetMapping*: In diesem Fall wird die Funktionalität auf eine Recheneinheit (*ProcessUnit*) gemappt. Dies bedeutet, dass es für die Funktionalität des logischen Blocks keine Modellierung in der System Software Architektur gibt, z.B. wenn es sich um eine Legacy-Bestandsim-

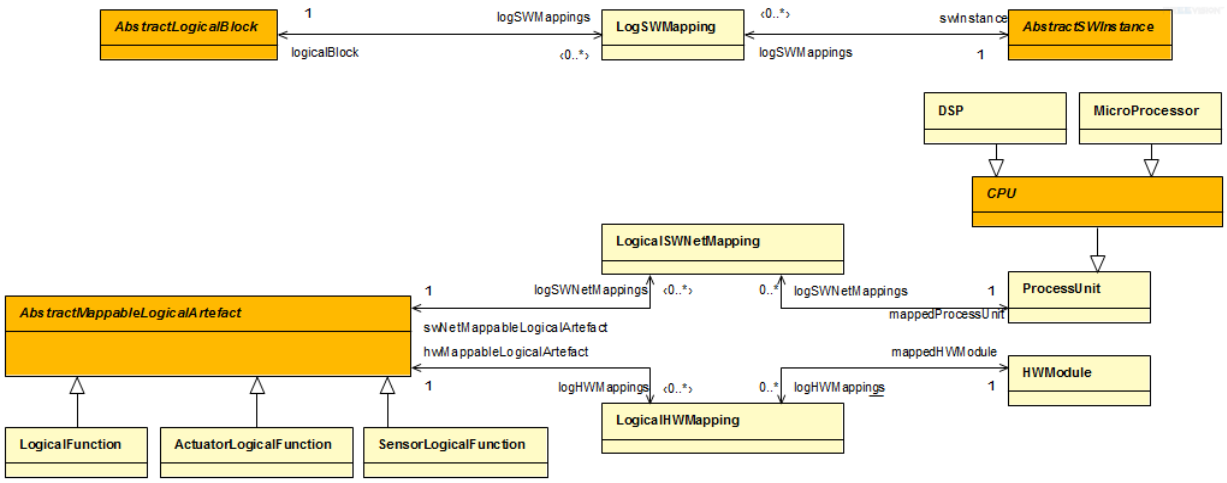


Abbildung 3.32: Mapping zwischen logischer Architektur und System Software Architektur / Hardwareebene

plementiertung handelt.

- *LogicalHWMapping*: In diesem Fall wird die Funktionalität auf Hardware-Module (*HWModule*) gemappt. Dies bedeutet, dass die Funktionalität durch einen (meist analogen) Schaltkreis realisiert wird. Ein Beispiel hierfür ist z.B. eine temperaturerfassende Sensorfunktion, die auf einen temperaturabhängigen Widerstand gemappt ist, der in einem Temperaturfühler verbaut ist.

3.3.7.5 System Software Architektur – Hardwareebene

Aktor-, Sensor- und und Softwarekomponentenblöcke können auf bestimmte Hardwareelemente aus der Komponentenebene gemappt werden (siehe Abbildung 3.33). Dazu gehören Aktor, Sensor sowie die Prozesseinheit und seine Derivate. Semantisch gesehen drückt das Mapping aus, dass der Block aus dem Funktionsnetzwerk auf der referenzierten Komponente ausgeführt wird. Erst durch diese Zuordnung wird entschieden, ob ein Block als Hardware (z.B. durch Mapping auf ein Hardwaremodul) oder als Software (z.B. durch Mapping auf eine CPU) ausgeführt wird. Ist diese Entscheidung zum Zeitpunkt der Zuordnung noch nicht endgültig getroffen, so wird der Block auf eine Prozesseinheit gemappt, die später konkretisiert wird.

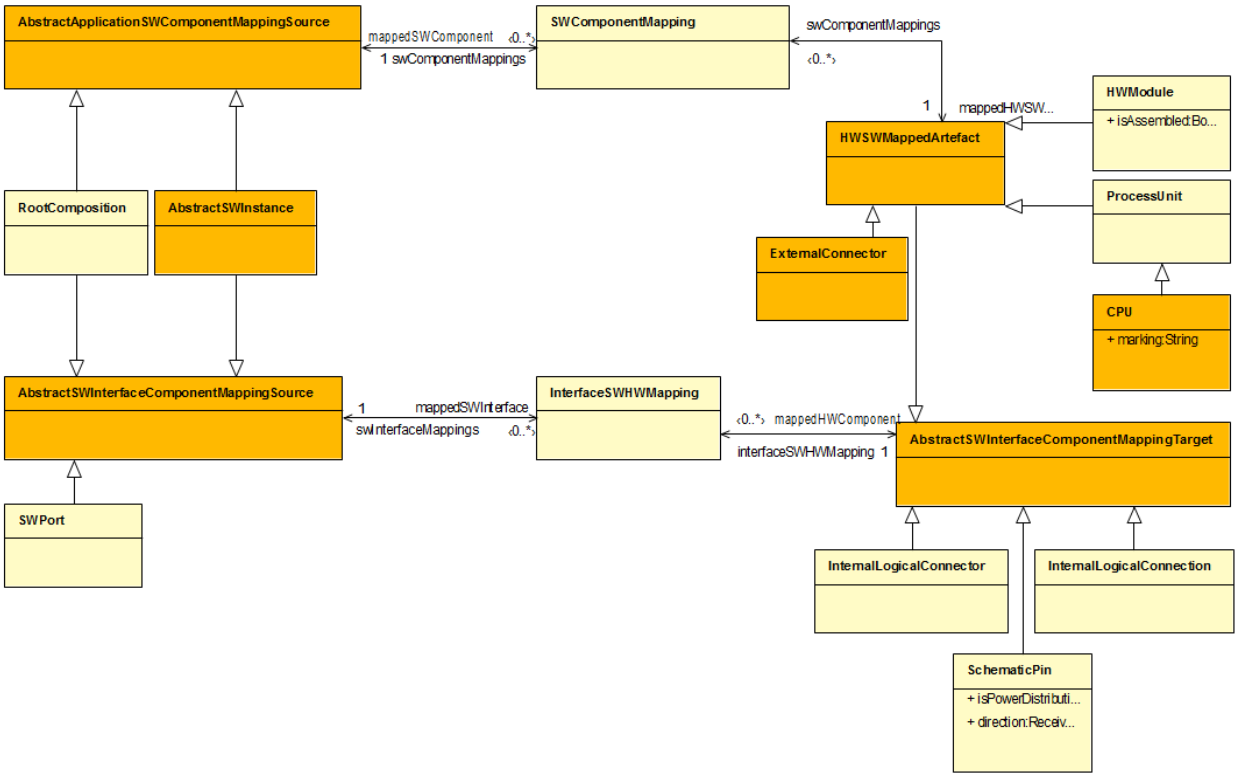


Abbildung 3.33: Mapping zwischen System Software Architektur und Hardwareebene

Durch das Mapping von Blöcken des Funktionsnetzwerks auf Hardwarebausteine der Komponente-nebene werden die ausführenden Elemente von Funktionen festgelegt. Dadurch ergeben sich Realisierungsvorgaben für Funktionen. Funktionsblöcke bezüglich ihrer Implementierungsform in Software oder Hardware. Im Umkehrschluss ergibt sich aus diesem Sachverhalt die Randbedingung, dass fertig implementierte Funktionsblöcke nur auf kompatible Hardwareelemente gemappt werden dürfen.

3.3.7.6 Hardwareebene – Physikalische Topologie

Das Mapping von Komponenten (z.B. Steuergeräte, Aktuatoren, Sensoren) auf Einbauorte der physikalischen Topologie beschreibt die Platzierung der jeweiligen Komponente im Fahrzeug. Dies wird im Metamodell in Abbildung 3.34 durch die Mappingklasse *PlacementComponentMapping* und den Assoziationen zu *EEComponent* und *InstallationLocation* gezeigt. *Splices* werden über die Mappingklasse *PlacementSpliceMapping* zu Topologieausbindungen (*BranchOff*) oder Einbauorten (*InstallationLocation*) zugeordnet. Trennstellen im Leitungssatz (*WHInlineConnector*) werden über die Mappingklasse *PlacementInlineConnectorMapping* auf Trennstellen in der Topologie (*InlineConnector*) gemappt.

Leitungen und Kabel (*SingleWire* und *Cable*) werden über die Mappingklasse *PathMapping* auf To-

pologiesegmente (*TopologySegment*) gemappt, in denen sie verlegt werden. Erst durch dieses Mapping werden die Leitungslängen bekannt.

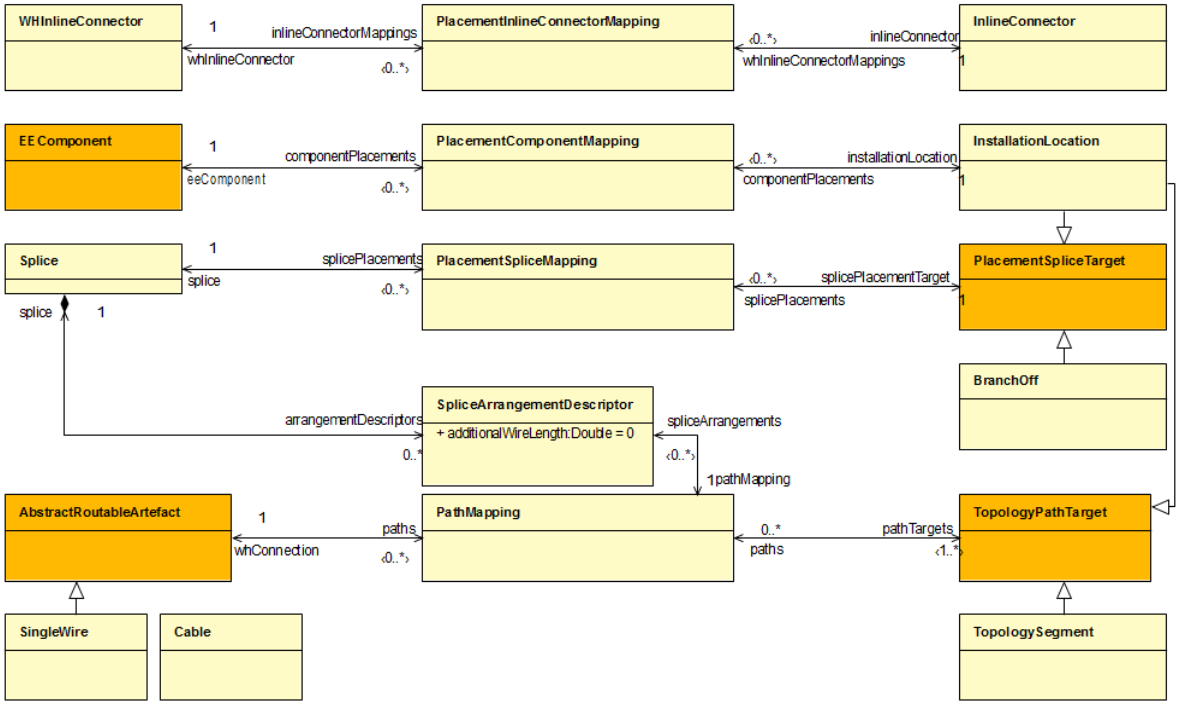


Abbildung 3.34: Mapping zwischen Hardware und Topologie

Beim *PathMapping* gibt es eine Besonderheit zur Modellierung von Stützpunkten (*Splices*). Je nach verwendeter Technologie (z.B. Quetschen oder Löten) oder Durchmesserunterschieden der eingehenden Leitungen können bestimmte Splicearten nur einseitig angeschlossen werden. Damit ergibt sich, wie in Abbildung 3.35 dargestellt, eine zusätzliche Leitungslänge für die Leitungen, die um den Splice herum geführt werden müssen.

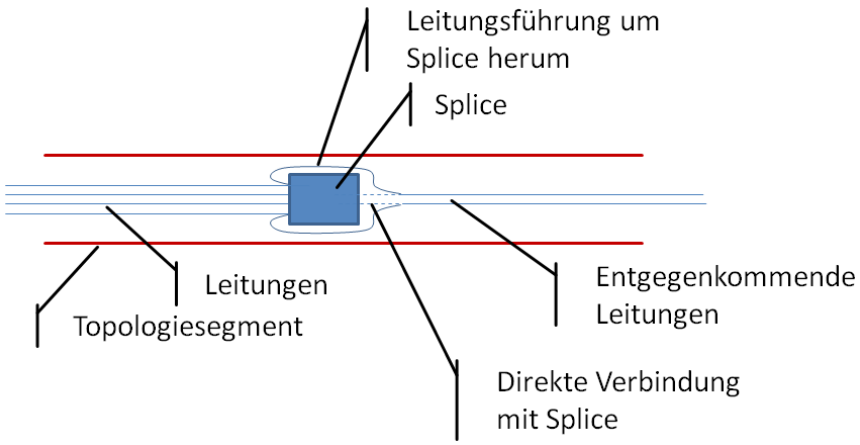


Abbildung 3.35: Einseitiger Spliceanschluss

Im Metamodell wird dieser Sachverhalt durch die Klasse *SpliceArrangementDescriptor* abgebildet.

Dieses Artefakt ist dem *Splice* untergeordnet und besitzt eine Assoziation zu dem *PathMapping*, welches die Platzierung der zu verbindenden entgegenkommenden Leitung in das Topologiesegment beschreibt. Das Attribut additionalWireLength gibt die zusätzliche Leitungslänge an.

3.3.8 Beschreibung von Architekturvarianten und Realisierungsalternativen

Bei der modellbasierten Beschreibung von Elektrik/Elektronik-Architekturen ist die durchgehende Berücksichtigung von Varianten und Architekturalternativen wichtig, um zentralen Schwerpunkten wie Plattform- und Baureihenentwicklungen Rechnung zu tragen. Varianten stellen unterschiedliche Fahrzeugkonfigurationen dar. Dies können beispielsweise unterschiedliche Ausstattungspakete, Karosserieformen oder Ländervarianten (Stichwort Rechts- und Linkslenker) sein. Architekturalternativen sind verschiedene technische Realisierungsmöglichkeiten, welche eine gestellte Anforderung umsetzen und so bezüglich verschiedener Metriken (siehe Kapitel 7) miteinander verglichen werden können. Architekturalternativen können sich auch aus Optimierungsmaßnahmen ergeben.

Aufgrund der hohen Anzahl von Möglichkeiten, die sich über unterschiedliche Ausstattungs- und Architekturalternativen ergeben, können nicht alle Variationen berechnet und miteinander verglichen werden. Jedoch können sinnvolle, sogenannte Eck- und Basistypen definiert werden, welche im weiteren Verlauf der Variantenbetrachtung genauer untersucht werden.

In Bezug auf die modellbasierte Beschreibung von Architekturvarianten umfasst das E/E-Architekturmodell ein sogenanntes 150% Modell. Dies beschreibt eine Architektur, die alle Möglichkeiten von Ausstattungs- und Architekturalternativen umfasst. Dies bedeutet, dass zum Beispiel alle für das modellierte Fahrzeug, Baureihe oder Plattform angebotenen Scheinwerfersysteme (Halogen, Xenon oder LED) im 150%-Modell enthalten sind. Durch die Variantenmodellierung ergeben sich dann daraus gültige E/E-Architekturen oder -Teilarchitekturen.

3.3.8.1 Metamodell

Die modellbasierte Beschreibung von Varianten in E/E-Architekturen stützt sich im Wesentlichen auf die Mengenbildung von Artefakten. Dazu gibt es eine Metaklasse zur Bildung von Modellgruppen (*Set*), gezeigt in Abbildung 3.36. Modellgruppen können über die Vererbung zu *AbstractSet* und dessen Assoziation zu *SubSet* andere Modellgruppen einschließen bzw. Teil anderer Modellgruppen sein. Über die weiteren Superklassen *AbstractModelInclusionAssignment* und *ModelExclusionAssignment* können Artefakte (*AbstractAssignableArtefact*) der Modellgruppe zugeordnet (*in-*

cludedArtefacts) oder explizit nicht zugeordnet (*excludedArtefact*) werden.

Im Metamodell gibt es Klassen zur Abbildung von Varianten (Abbildung 3.36). Zentraler Punkt ist die Klasse Konzeptraum (*ConceptSpace*). Sie spannt den Raum für eine Variantenbetrachtung des E/E-Architekturmodells auf.

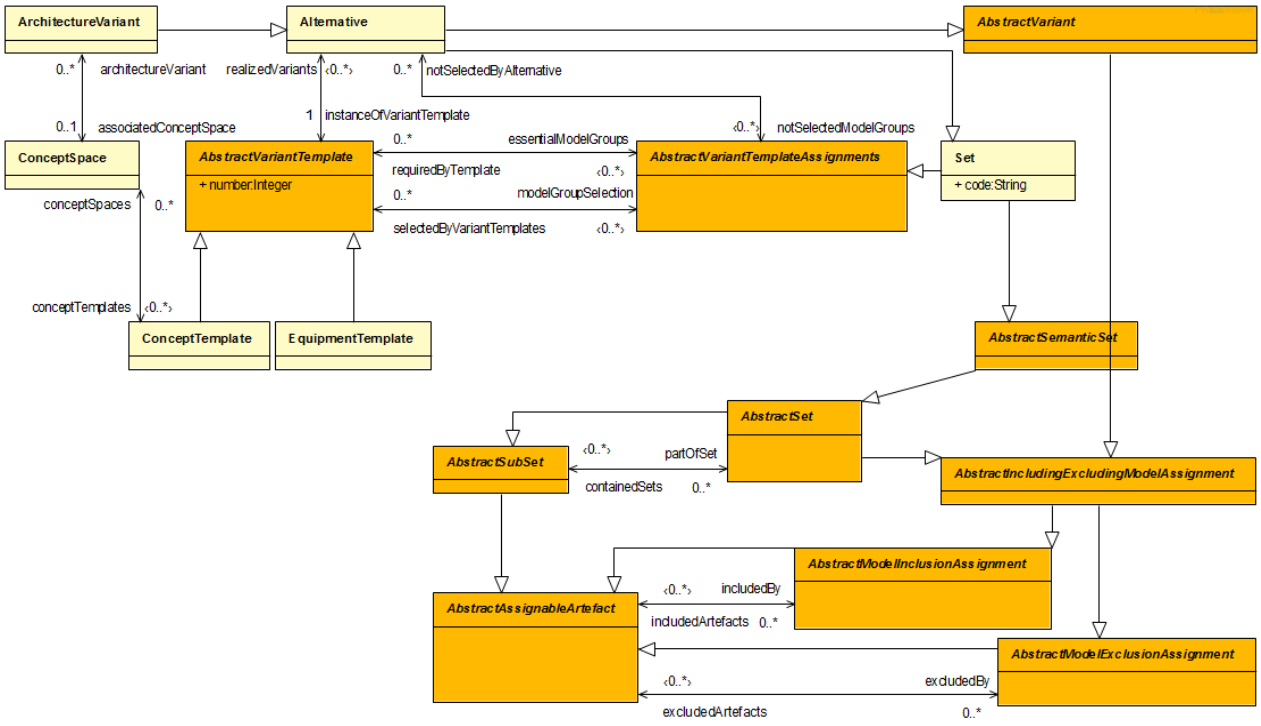


Abbildung 3.36: Variantenmodellierung

Dem Konzeptraum zugeordnet sind Konzeptvorlagen (*ConceptTemplate*). Sie bilden zusammen mit der Ausstattungsvorlage (*EquipmentTemplate*) die grundsätzlichen Unterscheidungskriterien der Variantenbetrachtung. Im übertragenen Sinne können sie auch als Dimensionen verstanden werden, die im Konzeptraum aufgespannt werden. Den Konzept- und Ausstattungsvorlagen werden Architekturalternativen über die Assoziation zwischen *AbstractVariantTemplate* und *Alternative* zugeordnet. Sie referenzieren auf die bereits beschriebenen Modellgruppen und damit direkt oder indirekt auf alle Artefakte der E/E-Architektur, die zur Umsetzung dieser Architekturalternative benötigt werden. Darüber hinaus verfügen die Vorlagen (über die gemeinsame Superklasse *AbstractVariantTemplate*) über zwei Assoziationen zu *AbstractVariantTemplateAssignment*, einer abstrakten Oberklasse der Modellgruppen. Über diese Assoziationen wird gesteuert, ob die referenzierte Modellgruppe stets zur Vorlage gehört (*essentialModelGroups*) oder ob die referenzierte Modellgruppe optional zur Vorlage gehören kann (*modelGroupSelection*). Über diesen Mechanismus können über Modellgruppen optionale und Basiskonfigurationen bereitgestellt werden. Die Alternativen werden

daraus durch Selektion und Deselektion bestückt.

Architekturvarianten (*ArchitectureVariant*) selektieren aus jeder Ausstattungs- und Konzeptvorlage des Konzeptraums genau eine Architekturalternative. Dies wird dadurch erreicht, dass *ArchitectureVariant* von *Alternative*, und damit dessen Assoziation zur abstrakten Oberklasse der Vorlagen, *AbstractVariantTemplate*, erbt.

3.3.8.2 Graphische Repräsentation

Neben zahlreichen Tabellen- und Formularbasierten Visualisierungen eignet sich vor allem das Netzdiagramm, um die Beziehungen zwischen den für Variantenbeschreibungen relevanten Artefakten darzustellen.

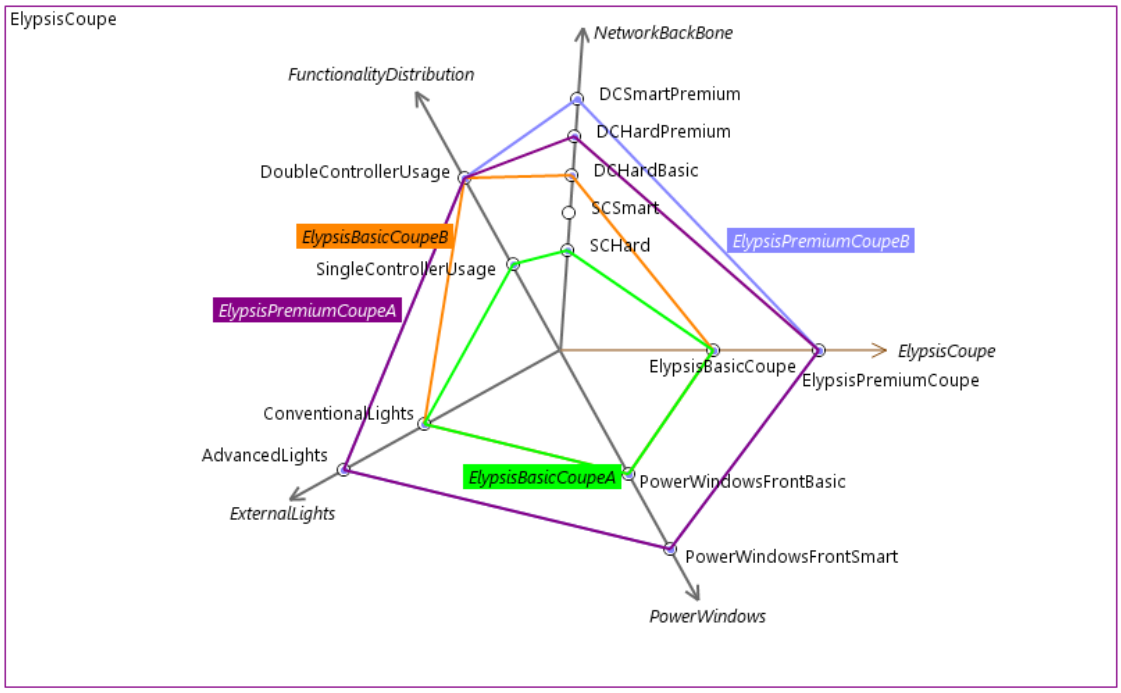


Abbildung 3.37: Darstellung von Architekturvarianten im Netzdiagramm [ELY11]

In Abbildung 3.37 ist ein Netzdiagramm dargestellt [ELY11]. Es zeigt für den Konzeptraum *ElypsisCoupe* die betrachteten Konzeptvorlagen *NetworkBackBone*, *FunctionalityDistribution*, *ExternalLights* und *PowerWindows*. Diese unterscheiden verschiedene Realisierungsalternativen für das Klappdach, Licht und Fensterheber mit variierendem Funktionsumfang und Vernetzungskonzept, ohne an dieser Stelle tiefer in die technischen Details zu gehen. Die Ausstattungsvorlage differenziert zwischen einem *ElypsisBasicCoupe* und *ElypsisPremiumCoupe*.

Ferner sind im Netzdiagramm die Architekturvarianten *ElypsisBasicCoupeA*, *ElypsisBasicCoupeB*,

ElypsisPremiumCoupeA und *ElypsisPremiumCoupeB* enthalten. Diese selektieren aus jeder Vorlage eine Ausstattungs- oder Realisierungsalternative und beschreiben somit unterschiedliche Architekturvarianten für das Elypsis-Fahrzeug.

4 Bewertung, Bewertungsverfahren und Vergleich von E/E-Architekturen

In diesem Kapitel werden die für diese Arbeit zentralen Begriffe Bewertung und Optimierung im Kontext dieser Arbeit beschrieben. Im Anschluss an diese Begriffsdefinitionen werden im Kontext der Modellbewertung und -optimierung relevante Arbeiten vorgestellt und gegen diese Arbeit abgegrenzt.

4.1 Definitionen

In diesem Abschnitt werden relevante Begriffe wie „Metrik“, „Bewertung“ und „Optimierung“ für deren Verwendung innerhalb dieser Arbeit definiert.

4.1.1 Metrik

Der Begriff Metrik stammt aus dem griechischen μετρική und bedeutet soviel wie Zählung oder Messung. Im technischen Sinne ist mit einer Metrik eine Berechnungsvorschrift verknüpft, mit deren Hilfe sich komplexe Zusammenhänge eines Systems in Kennzahlen formulieren lassen.

Ein Beispiel dazu sind Softwaremetriken, die Aussagen über Größe und Komplexität einer Software machen:

Def. 63: Softwarequalitätsmetrik

> *„A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.“ [IEEE1061, S.3]*

In Anlehnung dazu wird im Fokus dieser Arbeit der Begriff Metrik verstanden als Verfahren zur Erfassung von Kennzahlen basierend auf der Analyse der Daten des E/E-Architekturmodells. Im Rahmen dieser Arbeit wird der Begriff Metrik im Kontext von E/E-Architekturmodellen wie folgt definiert:

Def. 24: E/E-Architekturmetrik, Metrik (angelehnt an Softwarequalitätsmetrik)

> *Eine E/E-Architekturmetrik ist eine Berechnungsvorschrift, mit deren Hilfe die Bewertung einer E/E-Architektur bezüglich vorgegebener Bewertungskriterien in eine oder mehrere Kennzahlen durchgeführt wird.*

Während die oben genannte Definition für Softwaremetriken als Ausgabe einen einzelnen numerischen Wert vorsieht, lockert die Definition für E/E-Architekturmetriken diese Einschränkung auf, indem sie als Ausgabe einer Metrik eine oder mehrere Kennzahlen erlaubt. Oft ist es bei der Erhebung einer E/E-spezifischen Kennzahl sinnvoll, weitere, verwandte Kennzahlen zu erheben. Bei der Berechnung der Anzahl der im Fahrzeug verlegten Einzelleitungen können parallel die entsprechenden Längen und Gewichte mit bestimmt werden (vgl. Abschnitte 7.1.3.1 und 7.1.3.2).

4.1.2 Bewertung

Def. 10: Bewertung

> *„Bewertung heißt die Verknüpfung der zugänglichen Information eines Sachverhaltes mit dem persönlichen Wertesystem zu einem Urteil über den entsprechenden Sachverhalt. Eine Bewertung ist ein Hilfsmittel der Entscheidungsfindung, stellt jedoch noch nicht die Entscheidung selbst dar.“ [GIEG91, S.19]*

Auf technische Anwendungen übertragen handelt es sich bei dem Sachverhalt um ein technisches System (Def. 66). Die dort zugänglichen und zu betrachtenden Informationen beruhen im allgemeinen auf sachliche Fakten, sie müssen in einem funktionalen Zusammenhang mit dem zu entscheidenden Sachverhalt stehen. Das persönliche Wertesystem repräsentiert die Wertevorstellungen der verantwortlichen Personen / des verantwortlichen Teams innerhalb eines Unternehmens oder einer Organisation. Darin fließen auch früher gewonnene Einschätzungen, gewonnene Erfahrungen und Einschätzungen der Umgebung ein [GIEG91]. Sind mehrere Personen an der Bewertung beteiligt, müssen sie sich auf ein gemeinsames Wertesystem einigen. An dieser Stelle wird der Bewertungsvorgang durch subjektive Meinungen und Wertvorstellungen der beteiligten Personen beeinflusst.

Die Verknüpfung der sachlichen Informationen mit dem Wertesystem führt über die Bewertungslogik zu einem Bewertungsurteil.

Def. 11: Bewertungslogik

> *„Eine Methode der Bewertungslogik ist ein formalisiertes Regelwerk, das vorschreibt, wie zugängliche Informationen und Werthaltungen zu einem Bewertungsurteil verknüpft werden müssen. Eine Bewertungslogik muss unabhängig von den jeweiligen Sachinhalten anwendbar sein.“ [GIEG91, S.20]*

Werthaltungen werden als Bewertungskriterium formuliert.

Def. 33: Kriterium, Bewertungskriterium

> *„Ein Kriterium … ist ein Merkmal, das bei einer Auswahl zwischen … Objekten (Gegenständen, Eigenschaften, .. usw.) relevant für die Entscheidung ist.“ [WIBK]*

Damit diese allgemein definierten Bewertungskriterien im Sinne von Def. 24 in E/E-Architekturmetriken verwendet werden können, gilt im Kontext dieser Arbeit folgende Konkretisierung:

Def. 23: E/E-Architektur-Bewertungskriterium

> *Ein E/E-Architektur-Bewertungskriterium ist ein Bewertungskriterium. Es definiert als Eingangsgröße eine Menge von Artefakten einer E/E-Architektur. Als Ausgangsgröße definiert es eine oder mehrere Kennzahlen, die für eine Entscheidung relevant sind. Ein Sonderfall von E/E-Architektur-Bewertungskriterien sind Ausschlusskriterien.*

Das Bewertungsurteil (Def. 13) steht nach der Bewertung zur Verfügung und trägt zur Erreichung des Bewertungsziels (Def. 15) bei. Dieses dient zur Vorbereitung einer planerischen Entscheidung. Von der Reichweite der planerischen Entscheidung hängt ab, welche sachlichen Informationen und Wertehaltungen zur Bewertung herangezogen werden müssen [GIEG91].

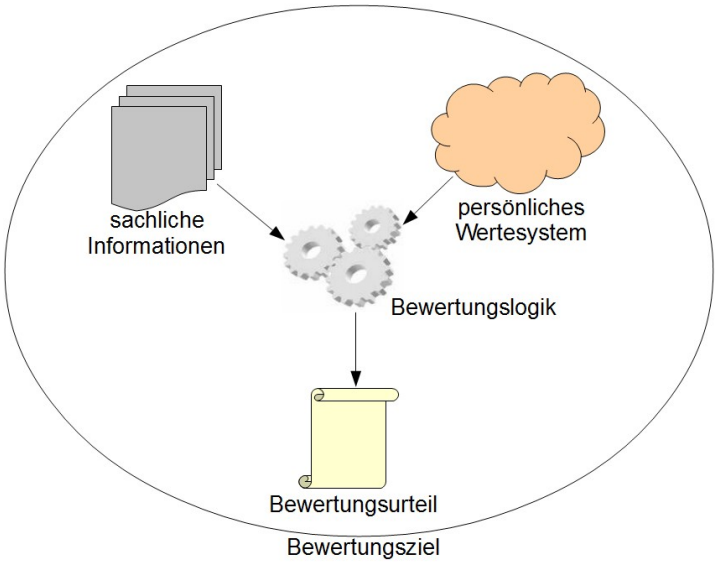


Abbildung 4.1: Definition einer Bewertung

Sind die folgen der planerischen Entscheidung sehr weitreichend, so müssen zum Beginn der Bewertung fundierte Informationen zur Verfügung stehen. Der Kreis der Entscheidungsträger muss entsprechend besetzt sein, woraus sich das in die Bewertung einfließende Wertesystem ableitet.

Der Zusammenhang zwischen sachlichen Informationen, persönlichem Wertesystem, Bewertungs-

verfahren, Bewertungsurteil und Bewertungsziel wird in Abbildung 4.1 veranschaulicht.

Diese Definition des Begriffs Bewertung ist noch sehr allgemein gehalten und bedarf einer Konkretisierung für modellbasierte Bewertungen von Elektrik/Elektronik-Architekturen, um in dieser Arbeit Anwendung zu finden.

Die zur Bewertung heranzuziehenden sachlichen Informationen sind Teil eines E/E-Architekturmodells, wie es in Kapitel 3.3 spezifiziert ist. Das Wertesystem bzw. die Wertehaltung werden von den beteiligten Entwicklern, E/E-Architekten, Entscheidern und anderen relevanten Personen/Rollen (relevante Akteure) in Form von mindestens einer Bewertungsmetrik formuliert. Jede Bewertungsmetrik spiegelt ein Bewertungskriterium (Def. 33) wieder. In der Summe aller Bewertungskriterien sind alle (von einem Computer ausführbaren) Berechnungsvorschriften abgelegt, die zu einer Bewertung benötigt werden. Die Bewertungslogik wird rechnergestützt durchgeführt, indem die bewertungsrelevanten Daten des Architekturmodells mit den Bewertungsmetriken verknüpft werden. Die Bewertungslogik ist unabhängig von den verarbeiteten Sachdaten. Interpretation des Bewertungsurteils und Einordnung in das Bewertungsziel erfolgen wiederum durch die relevanten Akteure.

4.1.3 Optimale Architektur

Laut [WIOPTU14] ist der allgemeine Begriff „Optimum“ wie folgt definiert:

Def. 52: Optimum

> *„Unter einem Optimum (lateinisch optimum, Neutrum von optimus ‚Bester, Hervorragendster‘, Superlativ von bonus ‚gut‘) versteht man das beste erreichbare Resultat im Sinne eines Kompromisses zwischen verschiedenen Parametern oder Eigenschaften unter dem Aspekt einer Anwendung, einer Nutzung oder eines Zieles.“ [WIOPTU14]*

Im Kontext dieser Arbeit kann diese Definition auf E/E-Architekturen angewandt werden. Das beste erreichbare Resultat ist die optimale E/E-Architektur. Die erwähnten Parameter und Eigenschaften sind die Ergebnisse von Bewertungsmetriken, welche auf Basis der Architekturdaten berechnet werden. Diese liefern zum Teil Ergebnisse, die gegenläufige Entwicklungsrichtungen der Architektur bedeuten können. Die Verknüpfung der Ergebnisse stellt den Kompromiss dar, der zur Findung des besten Resultats benötigt wird. Die Auswahl und Spezifikation der beteiligten Metriken spiegeln den Aspekt der Anwendung, die Nutzung oder das Ziel wider.

4.1.4 Optimierung

In [WIOPT10] findet sich eine Definition zum Begriff „Optimierung“:

Def. 51: Optimierung

> *„Das Gebiet der Optimierung in der angewandten Mathematik beschäftigt sich damit, optimale Parameter eines - meist komplexen - Systems zu finden.“ [WIOPT10]*

Diese Definition des Begriffs bezieht sich zwar explizit auf das Gebiet der angewandten Mathematik, ist jedoch mit kleinen Änderungen auf die Optimierung komplexer E/E-Architekturmodelle übertragbar. Die zu optimierenden Parameters ergeben sich aus den Bewertungskriterien, nach denen eine Architektur bewertet wird (siehe 4.1.2). Die Optimierung der Parameter erfolgt durch Modifikation und Neubewertung der Architektur. Das „*komplexe System*“ ist das E/E-Architekturmodell selbst.

4.2 Bewertungs- und Optimierungsziele

Bei E/E-Architekturen handelt es sich um komplexe, verteilte, eingebettete Systeme. Beim Entwurf bieten sich viele Bewertungs- und Optimierungspotentiale. In diesem Abschnitt werden verfügbare

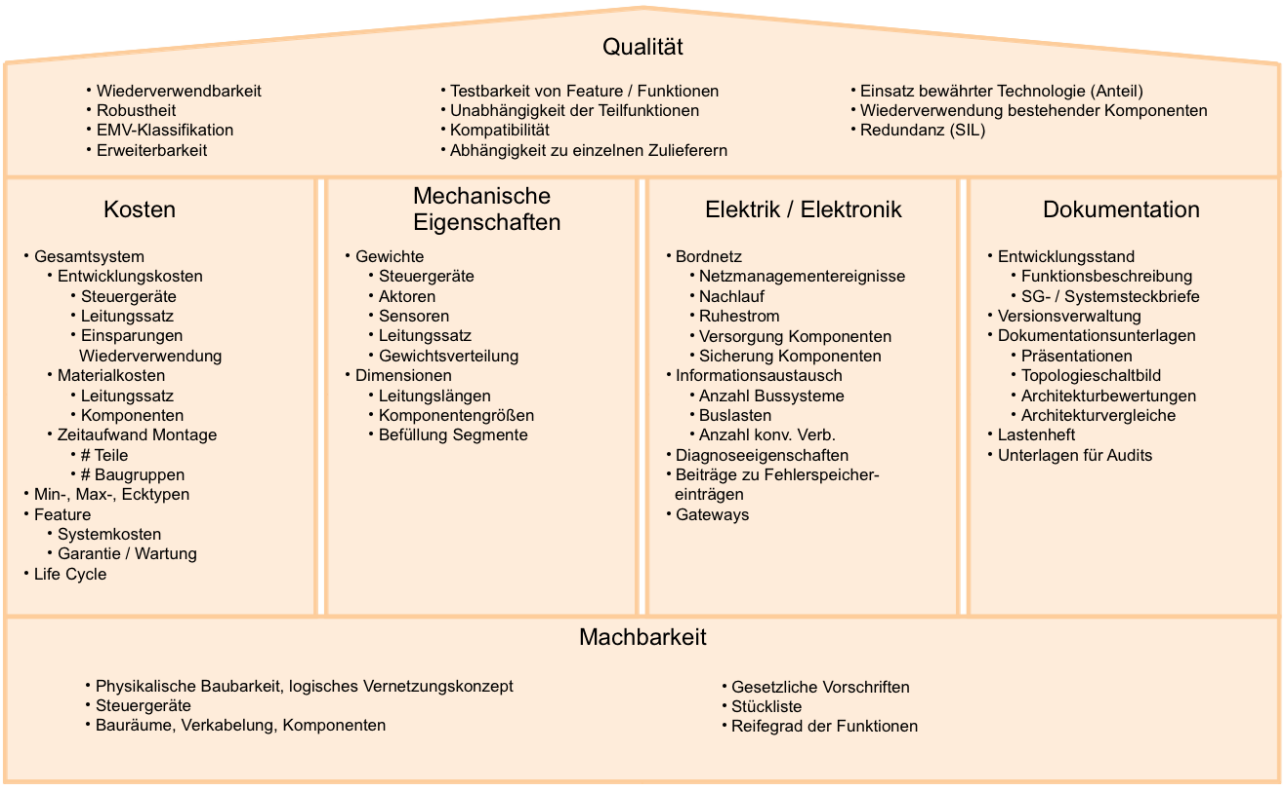


Abbildung 4.2: Kategorisierung von Bewertungskriterien

Bewertungskriterien aufgezeigt und kategorisiert.

Abbildung 4.2 zeigt eine Kategorisierung möglicher Bewertungskriterien. Das Fundament jeder Architekturentwicklung ist die Sicherstellung der Machbarkeit. Darauf bauen unterschiedliche Bewertungskriterien auf, die sich wiederum in Kosten, mechanische Eigenschaften, Elektrik / Elektronik und Dokumentation aufteilen lassen. Die letzte Kategorie Qualität beinhaltet Bewertungskriterien, für die qualitative Bewertung von E/E-Architekturen. Im Folgenden werden die einzelnen Kategorien detailliert vorgestellt und die darin enthaltenen Bewertungskriterien diskutiert. Die Kategorisierung der Bewertungskriterien ist nicht als absolut und auch nicht als vollständig anzusehen. Vielmehr kann ein Bewertungskriterium mehrere Aspekte beinhalten und so Teil mehrerer Kategorien sein. Auch kann hier kein Anspruch auf Vollständigkeit erhoben werden, da die zum Einsatz kommenden Bewertungskriterien herstellerabhängig sind.

Die hohe Variantenvielfalt der unterschiedlichen Fahrzeuge hat auf die E/E-Architektur und deren Bewertung Auswirkungen. So gibt es neben zahlreichen Varianten durch unterschiedliche Ausstattungskombinationen auch verschiedene Ländervarianten. Durch den Plattformgedanken entstehen zusätzliche Varianten einer E/E-Architektur, die in vielen Baureihen verbaut wird. Durch die Gesetzte der Kombinatorik ergibt sich eine sehr hohe Anzahl möglicher Varianten, die unmöglich alle bewertet werden können. Durch geschickte Bildung sogenannter Minimal-, Maximal- und Ecktypen wird versucht, bei den Käufern besonders beliebte oder für die Gesamtarchitektur wichtige Varianten zu definieren, die repräsentativ zur Bewertung herangezogen werden.

Hinzu kommt, dass während der Entwicklung einer Architektur unterschiedliche Realisierungsalternativen entstehen, aus denen nur ein Teil in das fertige Produkt übernommen werden. Dadurch wird die Anzahl der für die Bewertung relevanten Varianten während der Entwicklung nochmals erhöht.

4.2.1 Machbarkeit

Die Machbarkeit einer E/E-Architektur wird durch Anwendung unterschiedlicher Bewertungs- oder Ausschlusskriterien überprüft und ist im Rahmen dieser Arbeit wie folgt definiert:

Def. 34: Machbarkeit (E/E-Architektur)

> *Die Machbarkeit einer E/E-Architektur wird durch eine Reihe von Bewertungs- oder Ausschlusskriterien beschrieben, die deren Ergebnis die Entscheidungsgrundlage bilden, ob die bewertete E/E-Architektur realisiert werden kann.*

Ein Ausschlusskriterium die Überprüfung der **physikalischen Baubarkeit**. Darin wird geprüft, ob die verbauten Steckverbinder passend zueinander sind, ob die Leitungsquerschnitte eingehalten werden und ob die verbauten Leitungen in Bezug auf Schirmung und Bauart passend zu den Steckern gewählt sind. Befestigungskonzepte für Leitungssatz und Komponenten gehören ebenso dazu wie die Überprüfung, ob Komponenten in die vorgesehenen Einbauorte räumlich passen.

Das **logische Vernetzungskonzept** beinhaltet Fragestellungen, ob die vernetzten Komponenten auf diese Art und Weise miteinander verbunden werden können. Dies betrifft beispielsweise Aspekte der geplanten Topologie (Stern, Ring, Bus), die daraus resultierenden Leitungslängen und die Anzahl der miteinander verbundenen Steuergeräte. Auch die zu erwartende Datentransferrate beeinflusst die Entscheidung über die Machbarkeit einer Architektur.

Ein weiteres Bewertungskriterium betrifft die **Komponenten** der Architektur selbst. Hier sind Fragestellungen nach Verfügbarkeit, Ausfallwahrscheinlichkeiten, physikalische Größe und Gewicht zu beantworten. Aus der Platzierung in **Einbauorte** und die daraus resultierende **Verkabelung** der Komponenten können weitere Schlüsse über die Machbarkeit gezogen werden.

Des Weiteren müssen **gesetzliche Vorschriften** beachtet werden, die relevant für die geplante E/E-Architektur sind. So müssen beispielsweise Fahrzeuge mit Bi-Xenonlicht mit einer automatischen Leuchtweitenregulierung ausgestattet sein, um in Deutschland eine Zulassung zu erhalten.

Für eine detaillierte Betrachtung der Machbarkeit einer Architektur müssen auch Aspekte wie **Stücklisten** und **Reifegrad der Funktionen** in die Bewertung einbezogen werden. Die Überprüfung der Stücklisten ergibt, ob genügend Bauteile für eine erwartete Anzahl gebauter Architekturen am Markt verfügbar sind. Mit Reifegrad der Funktionen ist gemeint, ob die geplanten Funktionen einer Architektur vom Entwicklungsstand her soweit sind, dass sie in den produktiven Einsatz gehen können oder nicht.

4.2.2 Kosten

Bewertungskriterien, welche die Kosten einer Architektur untersuchen, spielen häufig eine große Rolle. Dies ist vor allem dem hohen Kostendruck innerhalb der Branche geschuldet. Interessant für eine Bewertung sind unterschiedliche Kostenarten.

Def. 32: Kosten (E/E-Architektur)

Die Kosten einer E/E-Architektur setzen sich aus unterschiedlichen Kostenteilen zusammen.

Neben den reinen Materialkosten gibt es Kosten für Entwicklung, Montage, Test, Wartung, Aftermarket, etc, die z.T. nicht unerheblich sind. Meist werden Kosten monetär betrachtet, es kann jedoch auch andere Sichtweisen, z.B. physikalische Eigenschaften, geben. Des weiteren gibt es, auf die einzelnen Bauteile der E/E-Architektur, geschätzte, berechnete und verhandelte Kosten.

Zunächst sind die **Materialkosten** einer konkreten Architektur von großem Interesse. Hier unterscheidet man zwischen geschätzten, berechneten und verhandelten Kosten, je nach Entwicklungsstand der zu bewertenden Architektur. In einem frühen Entwurfsstadium einer Architektur sind noch nicht alle Kosten bekannt, so dass an diesen Stellen mit geschätzten Kosten gearbeitet wird. Die berechneten Kosten ergeben sich aus bekannten Daten, wie zum Beispiel die Vorgängerbaureihe. Die verhandelten Kosten stehen an einem relativ späten Zeitpunkt zur Verfügung und repräsentieren die Kosten, welche mit den Zulieferern ausgehandelt wurden. Es ist durchaus möglich, dass ein Architekturmodell zu einem bestimmten Zeitpunkt mehrere Reifegrade der Materialkosten vereint. Die detaillierte Aufschlüsselung der unterschiedlichen Kostentypen kann so durch einen Unsicherheitsfaktor in eine Bewertung einbezogen werden.

Einen weiteren Aspekt der Kostenbewertung stellen **Montagekosten** dar. Dies sind besondere Zeitaufwände während der Montage, komplexe Baugruppen, die spezielle Arbeitsschritte oder Werkzeuge erfordern oder eine Vielzahl unterschiedlicher Teile. Ein anschauliches Beispiel stellt der Leitungssatz dar. So kann es durchaus sein, dass im Leitungssatz Trennstellen eingeplant werden, die weder durch die geometrische Struktur des Fahrzeugs, noch durch entsprechende Bedürfnisse der E/E-Architektur begründet sind. Sondern allein deshalb verbaut werden, um den Leitungssatz überhaupt bei der Montage verbauen zu können.

Die Bewertung der Kosten einer Architektur beinhaltet noch andere Facetten. So ist es üblich, dass durch Lernkurven, Produktivitätssteigerungen und Massenproduktion die Kosten für Komponenten oder Leitungssatz mit der Zeit fallen. Diese und andere Aspekte, wie zum Beispiel **Garantie- und Wartungskosten** sind bei der Bewertung des **Lebenszyklus** zu beachten.

4.2.3 Mechanische Eigenschaften

Def. 36: Mechanische Eigenschaften (E/E-Architektur)

Die mechanischen Eigenschaften einer E/E-Architektur beschreiben physikalische Eigenschaften von Bauteilen der Elektrik/Elektronik selbst als auch deren direkte relevante Umge-

bung im Fahrzeug.

Die Bewertung der mechanischen Eigenschaften umfasst vor allem **Gewichte** von Komponenten und Leitungssatz eine wesentliche Rolle. Ähnlich wie bei der Bewertung der Kosten (siehe Abschnitt 4.2.2) werden Gewichte während der Entwicklung in unterschiedlichen Reifegraden angegeben. Da gibt es zunächst geschätzte Gewichte für Komponenten, die sich noch in der Entwicklung befinden. Berechnete Gewichte werden angegeben, wenn das Gewicht eines Bauteils durch die Summierung der Gewichte seiner Einzelteile bestimmt wird. Sind die zu verbauenden Komponenten verfügbar, so können die tatsächlich gemessenen Gewichte angegeben werden.

Andere zu bewertende Größen sind **Längen** und **Querschnitte** vor allem im Leitungssatzbereich und die Abmessungen der Komponenten. Aus diesen Angaben können mitunter weitere Kennwerte der Architektur bestimmt werden. Während für einige, einfache Leitungsarten Festkosten unabhängig von der Leitungslänge entstehen, werden andere Leitungen über die jeweiligen Längen abgerechnet.

Weitere, wichtige mechanische Eigenschaften ist die Bestimmung der **Umwelteinflüsse** in den Einbauorten und Topologiesegmenten. Dazu gehören Temperaturbereiche, Feuchtraum- und Vibrationseigenschaften und Crashzonen, in denen die verbauten Bauteile und Leitungen bei Unfällen in ihrer Funktion besonders gefährdet sind.

4.2.4 Elektrik / Elektronik

Def. 25: Elektrik/Elektronik (E/E-Architektur)

Die Elektrik/Elektronik umfasst alle elektrischen und elektronischen Bauteile eines Systems zur Erfüllung von Funktionen und Funktionalitäten im Fahrzeug. Unter den elektrische Bauteilen versteht man die analogen Komponenten des Systems, unter elektronischen Bauteilen versteht man die digital arbeitenden Bauteile des Systems.

Weitere, wichtige Bewertungskriterien setzen sich mit den elektrischen und elektronischen Eigenschaften der Architektur auseinander. Im Bereich der Elektrik spielt das **Bordnetz** eine wichtige Rolle. Hierzu zählen vor allem Aspekte der Leistungsversorgung sowie das Massekonzept. Für eine Bewertung dieser Kriterien ist eine Berechnung der **Ströme** auf den Leitungen notwendig. Über diese Informationen können Versorgung der Komponenten und Energiebedarf bei unterschiedlichen Betriebsarten des Fahrzeugs bestimmt werden. Darüber hinaus lässt sich überprüfen, ob das **Ab-**

sicherungskonzept richtig dimensioniert ist. Werden beispielsweise für eine Leitung Ströme berechnet, die über die zulässige Absicherungsebene liegen, muss an dieser Stelle nachgearbeitet werden. Auf der anderen Seite ergeben sich Mehrkosten für überdimensionierte Leitungen, falls die berechneten Ströme auf Leitungen weit unterhalb der zulässigen Ströme für die Absicherungsebene liegen.

Weitere Bewertungskriterien sind die Bestimmung der **Ruhe**- und **Nachlaufströme** in den unterschiedlichen Betriebsarten. Daraus lassen sich maximale Standzeiten berechnen, bei denen das Fahrzeug ohne Aufladezyklen betriebsbereit bleibt bzw. **Abschaltungsszenarien** aktiviert werden müssen.

Weitere Bewertungskriterien setzen sich mit informationstechnischen Aspekten der Architektur auseinander. Dazu gehört die Anzahl der verwendeten Bussysteme, die Anzahl der vernetzten Komponenten und wie viele Gateways es gibt. Über den Kommunikationsbedarf der einzelnen auf den Steuergeräten verteilten Funktionen und die Vernetzungstopologie der Steuergeräte lässt sich ein Kommunikationsaufwand und damit die **Buslast** auf den einzelnen Bussystemen bestimmen. Anhand der Funktionen lassen sich auch Prozessor- und Speicherauslastungen auf den Steuergeräten ermitteln.

Eine weitere wichtige Größe sind die **Flashzeiten** für die einzelnen Steuergeräte. Diese sind nicht nur beim initialen Flashen der Steuergeräte am Ende der Fahrzeugproduktion wichtig, sondern spielen auch bei Werkstattbesuchen eine Rolle, wenn Softwareupdates vorgenommen werden müssen. Die ermittelten Flashzeiten spielen dann eine Rolle bei der Ermittlung der Garantie- und Lebenszykluskosten.

4.2.5 Dokumentation

Def. 19: Dokumentation (E/E-Architektur)

> *Die Dokumentation einer E/E-Architektur umfasst versionierte, archivierbare, auf Basis einer oder mehrerer E/E-Architekturen zusammengetragene Informationen in einem geeigneten Format. Dazu gehören das Lastenheft, Unterlagen zu Audits, Bewertungs- und Vergleichsergebnisse von E/E-Architekturen, etc.*

Eine durchgehende, konsistente Dokumentation einer E/E-Architektur während der Entwicklungsphase und darüber hinaus spielt eine wichtige Rolle. Bewertungskriterien in dieser Kategorie sind

unter anderem die Überprüfung, ob zu bestimmten Terminen wie **Projektmeilensteinen** oder **Audits** die relevanten Dokumente aus dem E/E-Architekturmodell generiert wurden. Diese umfassen beispielsweise das **Lastenheft**, den aktuellen **Entwicklungsstand** oder die **Versions-** und **Änderungsdokumentation** zu einem Referenzstand. Der Entwicklungsstand wird in Form von Funktionsbeschreibungen, System- und Komponentensteckbriefen dokumentiert. Bestandteil dieser Dokumente können wiederum Ergebnisse von Bewertungskriterien oder Architekturvergleiche sein.

Bei den Dokumenten selbst handelt es sich neben herkömmlichen textuellen Beschreibungen um Präsentationen, Topologieschaltbilder, aufbereitete Architekturbewertungen und -vergleiche oder tabellarische Ausgaben.

4.2.6 Qualität

Der Begriff der Qualität ist sehr weitreichend. Die für Qualitätsmanagement gültige Norm ISO 9000:2005 [ISO9000] definiert Qualität als:

Def. 55: Qualität

> *„Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt“ [ISO9000]*

„‚Inhärent‘ bedeutet im Gegensatz zu ‚zugeordnet‘ ‚einer Einheit innewohnend‘, insbesondere als ständiges Merkmal.“ [ISO9000]

Damit lässt sich die Qualität einer E/E-Architektur als solche nicht durch eine oder einen Satz von Bewertungskriterien abbilden. Jedoch können Teilbereiche des Qualitätsverständnisses abgebildet werden und Bewertungs- und Entscheidungsgrundlagen liefern. Dies können konkret zum Teil sehr einfache oder auch komplexe Berechnungsvorschriften für Bewertungskriterien sein.

Die **Wiederverwendbarkeit / Wiederverwendbarkeit bestehender Komponenten** gibt Auskunft darüber, inwieweit sich Komponenten in anderen Varianten, Produktlinien oder Baureihen einsetzen lassen. Komponenten können in diesem Zusammenhang Steuergeräte, Bauteile oder -gruppen, oder auch (Software-)Funktionen sein. In diesem Zusammenhang können vor allem Schnittstelleninkompatibilitäten (**Kompatibilität**) aufgedeckt werden. Als Schnittstellen kommen digitale (z.B. Busprotokoll), elektrische (z.B. Spannungen und auftretende Ströme) oder mechanische (z.B. Temperaturbereiche oder Baugrößen) Aspekte in Frage. Ein Umkehrschluss ist ebenfalls möglich: Welche Kompatibilitäten muss eine neu zu entwickelnde Komponente aufweisen, um in vorgegebenen E/E-Architekturen eingesetzt werden zu können.

Eng verknüpft mit der Wiederverwendbarkeit sind auch Fragestellungen der **Erweiterbarkeit**. In diesem Fall fokussieren sich die Bewertungskriterien nicht auf (In-)Kompatibilitäten, sondern inwieweit die konkrete Verwendung einer Komponente die vorhandenen Schnittstellen ausnutzt und somit Auskunft darüber gibt, welche Erweiterungen durchgeführt werden können.

Weitere Bewertungskriterien im Bereich der Qualität ist die **Testbarkeit von Funktionen / Feature**. Wichtige Kenngrößen zur Ermittlung sind die Anzahlen der Ein- und Ausgangsgrößen der Funktionalitäten oder auch die Abhängigkeit zu anderen Funktionalitäten. Dieses Bewertungskriterium geht damit mit der **Unabhängigkeit von Teilfunktionen** einher).

Bewertungskriterien zur Bestimmung der **Abhängigkeit von einzelnen Zulieferern** und der **Anteil** von **bewährten und erprobten Technologien** liefern Aussagen zu bestimmten Risikoszenarien, z.B. den Ausfall eines Zulieferers oder das Risiko zu unentdeckten Fehlers, die zu teuren Rückrufaktionen führen können.

4.3 Verfahren für Multi-Kriterien-Optimierungen

Bei E/E-Architekturen handelt es sich um komplexe Gebilde mit einer Vielzahl von Bewertungs- und Optimierungsmöglichkeiten. Die in Kapitel 4.2 beschriebenen Bewertungs- und Optimierungsziele beleuchten mögliche Bewertungskriterien aus unterschiedlichen Blickwinkeln, sind jedoch allgemein gehalten und nicht vollständig. Optimierungen einer E/E-Architektur lassen sich auf Basis einzelner Kriterien durchführen, interessanter und weitaus schwieriger sind jedoch Optimierungen auf einer breiten Basis von Bewertungskriterien, deren Optimierungsziele zum Teil gegenläufig sind und sich widersprechen können. In diesem Fall spricht man von Multi-Kriterien-Optimierungen, die in den folgenden Abschnitten beschrieben werden.

4.3.1 Vektorbasierte Optimierung

Für die Multi-Kriterien-Optimierung werden als Designvariablen q_i gebildet, die auf den zu verwendenden Bewertungskriterien basieren. Eine Designvariable kann aus einem Bewertungskriterium hervorgehen oder Teilaspekte mehrerer Bewertungskriterien abbilden. Die Interpretation von Bewertungskriterien zu Designvariablen werden durch eine Abbildungsfunktion realisiert [BUR08].

Die Designvariablen q_i werden als Vektor $\vec{q}$ definiert:

$$\vec{q} = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix}$$

Zum Ausschluss von nicht sinnvollen Lösungen werden Randbedingungen [BUR08]. Diese werden weiter unterteilt in Ungleichheitsrandbedingungen

$$\vec{g}(\vec{q}) \geqslant \vec{0}$$

und Gleichheitsrandbedingungen

$$\vec{h}(\vec{q}) = \vec{0} \quad .$$

Die Optimierung erfolgt über eine sogenannte Zielfunktion $\vec{f}(\vec{q})$. Meist werden diese Funktionen so formuliert, dass sie zur Optimierung minimiert werden müssen. Damit ergibt sich unter Berücksichtigung der Randbedingungen für die Optimierung [BUR08]:

$$min\{\vec{f}(\vec{q}) : \vec{h}(\vec{q}) = \vec{0}, \vec{g}(\vec{q}) \geq \vec{0}\}; \vec{q} \in \mathbb{R}^n$$

Als mögliche Zielfunktionen eignen sich die in den Abschnitten 4.3.3 und 4.3.4 Bewertungsverfahren über gewichtete Summen und beschränktes Referenzpunkt-Verfahren. Es sind jedoch auch weitere Zielfunktionen denkbar (z.B. DD96).

4.3.2 Pareto-Optimalität

Über die Anwendung der im vorangegangenen Abschnitt 4.3.1 beschriebenen vektorbasierten Optimierung lassen sich Ergebnisse bewerten und zueinander in Relation setzen. Dabei kann es zu einer großen Anzahl von Ergebnissen kommen, von denen nicht alle zu einer sinnvollen Lösung führen [BUR08]. Über die Pareto-Optimalität (Def. 53) kann die Menge der weiter zu untersuchenden Ergebnisse eingeschränkt werden.

In der Menge aller Lösungen $\mathbb{L}$ gibt es eine Lösung $\vec{a}$ bestehend aus den Designvariablen $(a_1, \cdots, a_n)$. Diese Lösung gilt dann als pareto-optimal, wenn es keine Lösung $\vec{b}$ gibt, die bezüglich aller Elemente $(b_1, \cdots, b_n)$ besser ist als $\vec{a}$. D.h. es gilt (im Fall der Minimierung):

$$für\ alle\ i = 1 \ldots n : a_i \leq b_i \wedge a_k < b_k (\ für\ min\ ein\ k\ aus\ 1 \ldots n)$$

Im Umkehrschluss heißt dies, dass man ein Element a_k einer pareto-optimale Lösung $\vec{a}$ nicht weiter verbessern kann, ohne ein anderes Element a_l damit zu verschlechtern.

In Abbildung 4.3 werden für den zweidimensionalen Fall mit den Designvariablen q_1 und q_2

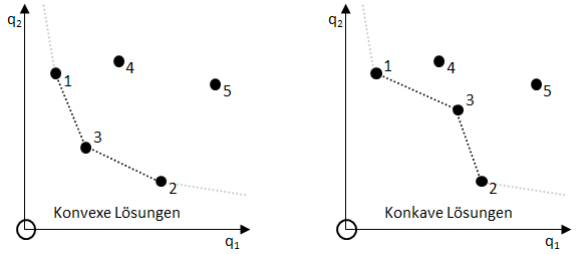


Abbildung 4.3: Beispiel für konvexe (links) und konkave (rechts) pareto-optimale Lösungen (in Anlehnung an [BUR08])

mögliche Lösungen graphisch dargestellt. Für die hier gezeigte Optimierung durch Minimierung liegt die beste, theoretische Lösung im Ursprung. Die Lösungspunkte mit den besten Ergebnissen sind durch gestrichelte Linien verbunden, sie bilden die sogenannte Pareto-Front. Dabei kann es zu konvexen und konkaven Lösungsverläufen kommen. Bei den konkaven Lösungsverläufen kann es zu minderwertigen Lösungen (in Abbildung 4.3 ist das Lösung 3 im rechten Bild) kommen, die teilweise schwer zu finden sind [DD97].

Lösungen, die nicht Teil der Pareto-Front sind, werden nicht weiter betrachtet.

4.3.3 Gewichtete Summen

Gewichtete Summen stellen eine gängige Zielfunktion dar und gehören zu den sogenannten non-pareto-Techniken, da über sie nicht alle (z.B. konkave) pareto-optimale Lösungen nicht gefunden werden [DD97]. Die Zielfunktion gewichtet die Elemente des Designvariablen-Vektors, anschließend erfolgt eine Summenbildung:

$$\vec{f}(\vec{q}) = \sum_{i=1}^{n} w_i f_i(q_i) \quad \text{mit} \quad \sum_{i=1}^{n} w_i = 1$$

Die Wertebereiche der einzelnen $f_i(q_i)$ müssen gleich sein, um den Effekt der Gewichtung nicht zu beeinflussen. Wichtige Kriterien werden mit einem höheren Gewichtskoeffizienten w_i versehen, nicht so wichtige mit einem niedrigeren. Diese Priorisierung fällt in der Praxis nicht immer leicht [BUR08].

4.3.4 Beschränktes Referenzpunkt-Verfahren

Das beschränkte Referenzpunkt-Verfahren (engl. Bounded Reference Point Method, BRPM) [BUR06] ist eine Erweiterung des Referenzpunkt-Verfahrens [WIE00] und wurde entwickelt, um insbesondere Kommunikationsnetzwerke mit Hilfe von mehreren unterschiedlichen Bewertungskri-

terien zu optimieren. Die Besonderheit hier ist, dass die Messdaten um zum Teil mehrere Größenordnungen auseinander liegen können. Im Gegensatz zur Referenzpunkt-Methode (RPM) können Bewertungskriterien in diesem Fall besser miteinander vergleichbar gemacht werden. Eine detaillierte Gegenüberstellung dieser beiden Verfahren findet sich in [BUR06].

4.3.4.1 Funktionsweise

Die Bewertungskriterien werden durch Designvariablen q_i mit einem gemeinsamen und damit vergleichbaren Wertebereich $(0,q_{max})$ repräsentiert. Die notwendige Transformation bildet die Werte so ab, dass der beste zu erreichende Wert einer Designvariable Q^u (u für *Utopia*, soll idealerweise den Wert 0 annehmen) und der schlechteste Wert einer Designvariable den Wert 1 = Q^n (n für *Nadir*, soll idealerweise den Wert 1 annehmen) entspricht. Für jedes Bewertungskriterium ist eine eigene Transformationsvorschrift erforderlich, da die Rohwerte unterschiedliche Wertebereiche haben können und gegenläufig bezüglich ihrer Wertinterpretation sein können.

Weitere Werte sind Q^a (a für *Aspiration*) für den Wert, welcher bezüglich der Designvariable q_i noch ein zufriedenstellendes Ergebnis darstellt und Q^r (r für *Reservation*) für nicht mehr akzeptable Werte der Designvariablen. Damit gilt folgender Zusammenhang:

$$Q_j^u < Q_j^a < Q_j^r < Q_j^n$$

Über diese Schwellenwerte können Leistungsfunktionen $l_i(q_i)$ definiert werden, welche die Designvariablen q_i auf einen einheitlichen, und damit vergleichbaren Wertebereich abbilden, in denen die Leistungswerte liegen. Im folgenden Detail unterscheidet sich das Referenzpunkt-Verfahren zum beschränkten Referenzpunkt-Verfahren: Während bei der RPM eine Beschränkung des Wertebereichs für Leistungswerte nicht vorgesehen ist, findet bei der BRPM eine Beschränkung des Maximalwerts auf 1,5 und des Minimalwerts auf -2 statt [BUR06]. Damit wird verhindert, dass die Leistungswerte zwischen mehreren Bewertungskriterien sich zu stark unterscheiden und so zu Verzerrungen führen.

Es ergibt sich damit folgende Definition der Leistungsfunktion für die BRPM:

$$l_i(q_i)=\begin{cases} 1+0{,}5\dfrac{Q_i^a-q_i}{Q_i^a-Q_i^u} & \text{für } Q_i^u<q_i<Q_i^a \\ \dfrac{Q_i^r-q_i}{Q_i^r-Q_i^a} & \text{für } Q_i^a<q_i<Q_i^r \\ 2\dfrac{Q_i^r-q_i}{Q_i^n-Q_i^r} & \text{für } Q_i^r<q_i<Q_i^n \end{cases}$$

Mit dieser abschnittsweise linearen Leistungsfunktion sinkt die Bewertung im Optimalbereich zwischen Q_i^u und Q_i^a weniger stark als im Zielbereich zwischen Q_i^a und Q_i^r und dort wiederum nicht so stark wie im nicht akzeptablen Leistungswertbereich zwischen Q_i^r und Q_i^n.

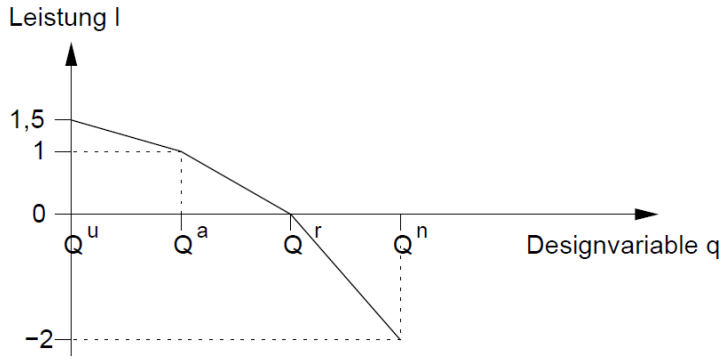


Abbildung 4.4: Abbildung von Disignvariablen q auf Leistungswerte l [BUR08]

Dieser Zusammenhang wird in Abbildung 4.4 veranschaulicht.

Abschließend wird über eine Zielfunktion $f(\vec{q})$ unter Ausnutzung der Leistungsfunktion eine Gesamtbewertung vorgenommen.

$$f(\vec{q})=\min_i(l_i(q_i,Q_i^u,Q_i^a,Q_i^r,Q_i^n))+\epsilon\sum_i l_i(q_i,Q_i^u,Q_i^a,Q_i^r,Q_i^n)$$

Der erste Term der Zielfunktion sucht den schlechtesten aller Leistungswerte heraus. Der zweite Term integriert eine Gesamtbetrachtung aller Leistungsfunktionsfunktionen. Damit dieser Term den ersten Term nicht überschattet, wird ein Faktor ϵ eingeführt. Hier werden Werte von 0,01 bis 0,1 vorgeschlagen [BUR06].

4.4 Stand der Technik – Bewertung von E/E-Architekturmodellen

4.4.1 Architekturevaluation eingebetteter Systeme im Automobil (ArchiVal)

Die Universität Braunschweig und die Volkswagen AG arbeiteten im Rahmen des Kooperationsprojektes „Architekturevaluation eingebetteter Systeme im Automobil“ (ArchiVal) [ARVL] zusammen. Ziel des Projektes ist, Sensitivitäten der Architekturanforderungen basierend auf Architekturentscheidungen zu analysieren und zu identifizieren. Dazu werden extra-funktionale Anforderungen an die E/E-Architektur modelliert. Dadurch können aussagekräftige Evaluationen von Architekturvarianten durchgeführt werden. Die identifizierten Sensitivitäten können genutzt werden, um Verbesserungen bei der Integration von Entwicklungsprozessen, der Architekturbetrachtung zu erreichen. Ferner weisen die identifizierten Sensitivitäten auf ungenutztes Potential bei Änderungen von Architekturvarianten hin.

4.4.1.1 Quality Attribute Directed Acyclic Graph (QADAG)

Kern dieses Projektes ist der Quality Attribute Directed Acyclic Graph (QADAG). Dabei handelt es sich um eine Evaluationsstruktur, welche die Darstellung extra-funktionaler Anforderungen erlaubt und bildet sowohl für die Evaluation als auch die Architekturanalyse die Grundlage [FLO08].

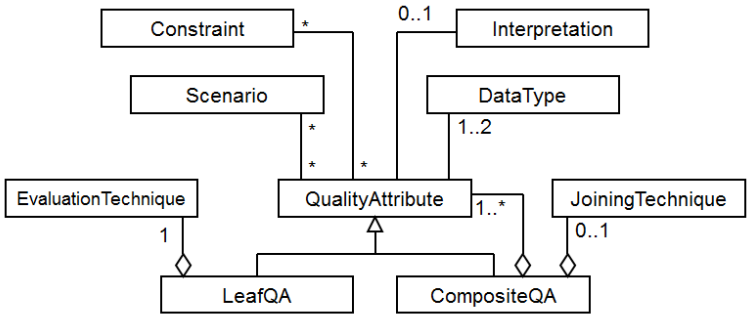


Abbildung 4.5: Das QADAG-Metamodell [FLO08]

Im Zentrum des QADAG steht das namensgebende Qualitätsattribut (*QualityAttribute*, Abbildung 4.5). Zur Bildung eines Kompositionbaums (unterer Teil der Abbildung 4.5) gibt es zwei Subklassen, das Blatt-Qualitätsattribut (*LeafQA*) und das zusammengesetzte Qualitätsattribut (*CompositeQA*). Jedem Blatt wird eine Evaluierungstechnik (*EvaluationTechnique*) hinterlegt, die Informationen darüber enthält, wie das Qualitätsattribut zu bewerten ist. Einem zusammengesetzten Qualitätsattribut kann eine Vereinigungstechnik (*JoiningTechnique*) gehören, die aussagt, wie die Ergebnisse

der untergeordneten Qualtätsattribute miteinander zu vereinigen sind. Hier können auch Gewichtungen angegeben werden.

Im oberen Teil der Abbildung 4.5 sind die weiteren Eigenschaften von Qualitätsattributen modelliert. Wichtigster Aspekt ist die Interpretation der Evaluierungsergebnisse (*Interpretation*). Die Evaluierungstechniken geben ihre Ergebnisse als Rohdaten wieder. Diese müssen interpretiert und auf eine Skala von 0% bis 100% (sogenannte Qualitätsrate) normiert werden (siehe Abschnitt 4.4.1.2). Somit können Evaluierungsergebnisse einerseits allgemein verständlich präsentiert werden, andererseits können zusammengesetzte Ergebnisse besser miteinander verknüpft werden. Die Datentypen (*DataType*) geben Auskunft über das Format der Rohdaten und der Qualitätsrate.

Bewertungskontexte, z.B. in Form von Anwendungsfällen und Benutzerinteraktionen, werden als Szenarien (*Scenario*) mit dem Qualitätsattribut verknüpft. Szenarien können mit mehreren Qualitätsattributen verknüpft werden, ebenso können Qualitätsattribute durch mehrere Szenarien näher beschrieben werden.

Bewertungen werden ferner durch Bedingungen (*Constraint*) eingeschränkt, die nicht durch die Betrachtung der Rohdaten der verwendeten Evaluierungstechnik abgedeckt werden. Dazu zählen beispielsweise Einschränkungen für erlaubte Zulieferer oder Performance-Eckdaten.

4.4.1.2 Wertinterpretation

Wichtiger Bestandteil für QADAG-basierte Architekturevaluationen ist die Wertinterpretation. Bei der Interpretation wird festgelegt, wie die Rohwerte in Qualitätsraten zwischen 0% und 100% übersetzt werden. Diese Festlegungen sind notwendig, um die Rohdaten in einheitlich normiertes Wertesystem zu übersetzen und damit vergleichbar und weiterverarbeitbar zu machen.

Die Interpretation der Werte lässt sich als Kurve in einem Koordinatensystem darstellen. In Abbildung 4.6 ist ein entsprechender Graph dargestellt. Hier wird die Auslastung des Festspeichers (ROM) eines Body Comfort System (BCS) den jeweiligen Qualitätsraten gegenüber gestellt [FLO08, S.111ff., Kapitel 6.1.4].

Die Entscheidung, welcher Rohwert (in diesem Fall Auslastung des Speichers in Prozent) welcher Qualitätsrate zugeordnet wird, muss vorgegeben werden. D.h. hier ist Entwicklungsarbeit notwendig. Die detaillierte Ausgestaltung der Interpretation kann technisch motiviert sein, oder auch Entwicklungsvorgaben folgen.

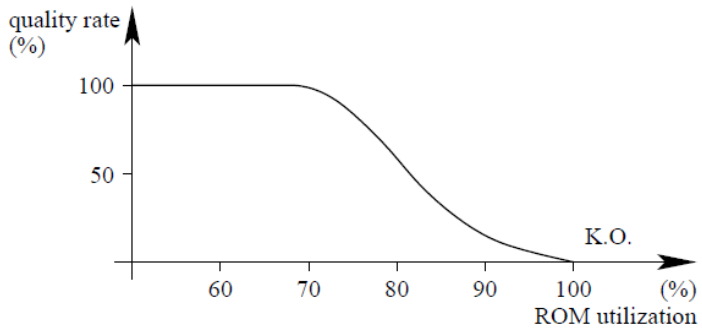


Abbildung 4.6: Beispiel: Interpretation ROM-Auslastung [FLO08]

4.4.1.3 QADAG-Evaluierungsergebnisse

QADAG-Evaluierungsergebnisse lassen sich über eine Tabellenkalkulation berechnen und darstellen. Das in Abbildung 4.7 dargestellte Beispiel zeigt die Evaluierungsergebnisse für eine in [FLO08, S.45ff., Kapitel 3.1] untersuchte Variante[13].

Variant SLC on PWD, CTC on MiniCTD, 61 Ah										
70.6 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
98.2 %			38 %			89.2 %			85.0 %	
-			€ 132			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	98.3 %		-	-	-	94 %	86.0 %		100 %	70 %
50 %	-		€ 92	€ 17	€ 23	19.6 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	100 %	100 %	95 %				92 %	80 %		
	60 %	68 %	73 %				48 d	3.89 a		

(weights)
QA name
quality rate
result

Legende

Abbildung 4.7: Beispiel: QADAG-Evaluierungsergebnis für eine Variante [FLO08]

Betrachtet werden in dem Beispiel die Aspekte Performance, Kosten, physikalische Eigenschaften und Modifizierbarkeit. Im Qualitätsattribut ROM führt die Auslastung des Festspeichers von 73% zu einer Qualitätsrate von 95% (siehe Abbildung 4.6). Diese Bewertung geht mit einer Gewichtungszahl von 100 in das zusammengesetzte Qualitätsattribut für Steuergeräte (ECU) ein, so dass sich eine resultierende Qualitätsrate von 98,3% ergibt. Dieser Wert wiederum geht mit einer

13 *SLC on PWD*: Die Funktion Short Lift Control (SLC) (Funktion, um die Fenster bei Betätigung des aufklappbaren Verdecks herunter zu lassen) wird im Steuergerät des elektrischen Fensterhebers (Power Window Device) integriert
CTC on MiniCTD: Die Steuerung des aufklappbaren Verdecks (Convertible Top Control CTC) wird auf einem eigenen Steuergerät MiniCTD (Mini Convertible Top Device) realisiert
61 Ah: Kapazität der verwendeten Batterie

Gewichtungszahl von 100 in das zusammengesetzte Qualitätsattribut für die Performance ein. Am Ende ergibt sich (das Qualitätsattribut Performance geht mit einer Gewichtung von 80 in die Gesamtbetrachtung ein) eine Gesamtbewertung für die betrachtete Variante von 70,6%.

4.4.1.4 Abgrenzung

Mit QADAG-basierten Evaluierungen des ArchiVal-Projektes werden hierarchisch strukturierte Qualitätsattribute erstellt. Diese werden, angereichert mit Informationen über Interpretation, Datentypen, Szenarien und Bedingungen über vorgegebene Techniken evaluiert bzw. vereint. Am Ende einer Evaluierung steht ein Ergebnis zwischen 0 und 100 Prozent. Diese Methodik setzt damit Multi-Kriterien-Bewertungen auf transparente und verständliche Weise um. Bisherige Implementierungen basieren auf Tabellenkalkulationen.

Das QADAG-Modell ist ein eigenständiges Modell zur Abbildung von Qualitätsattributen. Die Rohdaten dieser Attribute sind zum Teil isolierte Eigenschaften von Artefakten einer E/E-Architektur (z.B. die Kosten eines Bauteils) oder sie abstrahieren Eigenschaften einer Architektur (z.B. die Auslastung des ROM-Speichers). Im QADAG-Modell werden keine E/E-spezifischen Inhalte oder Strukturen abgebildet, es ist auch keine explizite Möglichkeit vorgesehen, direkt auf solche Inhalte zuzugreifen. Bei Änderungen der E/E-Architektur müssen die relevanten Rohdaten auf das QADAG-Modell (bzw. die Berechnungsvorlage in Form einer Tabellenkalkulation) übertragen werden.

Für die Editierung und Visualisierung von QADAG-basierten Metriken wurde ein übersichtliches Format basierend auf Tabellenkalkulation vorgestellt. Die Anpassung von Berechnungsvorschriften oder die Einführung neuer Qualitätsattribute erfordert den Eingriff in die Tabellenkalkulationsvorlage.

Die Untersuchung von mehreren Architekturvarianten oder Realisierungsalternativen erfolgt durch mehrere QADAG-Modelle. Vereinfachende Mechanismen sind nicht vorhanden. Sensitivitätsanalysen oder Was-Wäre-Wenn-Szenarien werden insofern unterstützt, dass über die angegebenen Qualitätsraten Abschätzungen über den Einfluss von Qualitätsattributen getroffen werden können. Über manuelle Änderung der Qualitätsraten können einzelne Analysen durchgeführt werden.

4.4.2 VEIA

Die BMW Group, das Fraunhofer Institut Software und Systemtechnik, die PROSTEP IMP GmbH sowie die Technische Universität München haben im Rahmen der Forschungsoffensive „Software

Engineering 2006“ im Verbundprojekt VEIA (Verteilte Entwicklung und Integration von Automotive Produktlinien) zusammengearbeitet. Die Projektlaufzeit war von Mai 2006 bis Oktober 2008.

Es wurden auf der Grundlage der Konzepte der Produktlinientechnik eine Methode für die verteilte Entwicklung und Integration von Automotive-Systemen erarbeitet. Das Projekt orientierte sich den konkreten Anforderungen industrieller Entwicklungsprozesse und widmete sich auch der praktischen Anwendbarkeit [VEIA].

4.4.2.1 VEIA-Referenzprozess

Im VEIA-Projekt wurde ein Referenzprozess definiert. Die Artefakte dieses Prozesses lassen sich in drei Bereiche aufteilen, wie auch in Abbildung 4.8 dargestellt [GR07RP]:

- Produkte: Hier wird beschrieben, welche Systeme mit welchen Eigenschaften zu entwickeln sind. Konkret werden Artefakte der Produktlinienbeschreibung entwickelt, welche Informationen über Produkte und deren Merkmale tragen.
- Funktionen: Hier wird beschrieben, welche Funktionalitäten zu implementieren sind. Dieser Bereich ist unterteilt in drei Hierarchieebenen. In der oberen (abstrakten) Ebene wird die sogenannte Dienstlandkarte spezifiziert. Dort werden die umzusetzenden Dienste beschrieben und hierarchisiert. Im Funktionsnetz werden Funktionen miteinander vernetzt und modelliert. In der SW-Architektur wird die zu implementierende Softwarearchitektur und die eingesetzten Softwarekomponenten beschrieben.

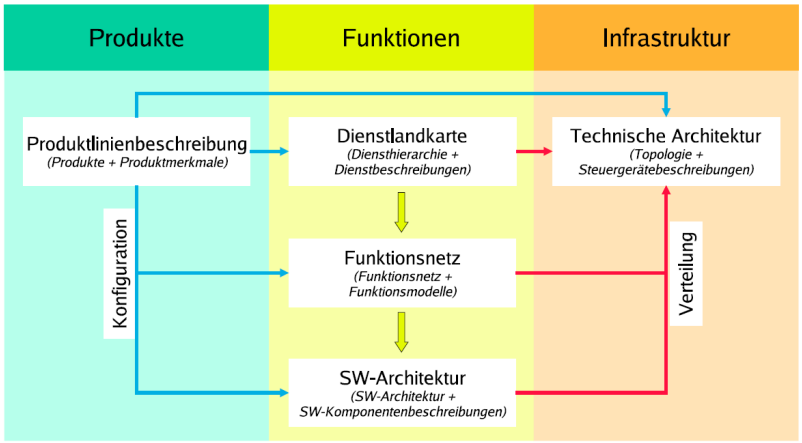


Abbildung 4.8: Der VEIA-Referenzprozess [GR07RP]

- Infrastruktur: Hier werden die technischen Ressourcen beschrieben, die zur Umsetzung der beschriebenen Funktionalitäten erforderlich sind. Dies umfasst zum einen die Topologie und zum anderen die Steuergerätebeschreibungen.

Der VEIA-Referenzprozess umfasst mehrere Einstiegspunkte, da Architekturen oder Architekturartefake oft projekt- oder produktlinienübergreifend und parallelisiert entwickelt werden.

4.4.2.2 VEIA-Vorgehensweise zur Architekturbewertung

Im VEIA-Projekt wurde eine Vorgehensweise zur Architekturbewertung entwickelt [GR07AB]. Die Fragestellungen der Architekturbewertung werden Szenarien zugeordnet (Abbildung 4.9). Ein Szenario werden die zur Beantwortung der Frage notwendigen Kriterien ermittelt. Damit werden die relevanten Architekturartefakte erfasst.

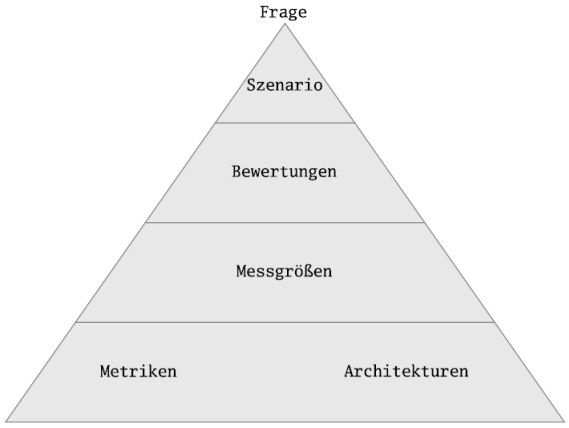


Abbildung 4.9: Elemente einer Architekturbewertung nach VEIA [GR07AB]

Über die Bewertung werden die Mess- oder Berechnungsergebnisse interpretiert und somit vergleich- und bewertbar. Die Mess- und Berechnungsergebnisse werden durch Metriken ermittelt, welche die Informationen aus dem E/E-Architekturmodell beziehen.

Die Vorgehensweise selbst orientiert sich an das V-Modell (Abbildung 4.10). Im linken Zweig des V beschreibt die Vorgehensweise die Entwicklung von Architekturbewertungen. Dazu werden im ersten Schritt Szenarien für die Bewertung aufgestellt. Daran anschließend werden die Bewertungen und die zu bewertenden Architekturen bestimmt. Im nächsten Schritt wird nach den erforderlichen Messgrößen gesucht. Für eine Bewertung können eine oder mehrere Messgrößen benötigt werden. Im unteren Teil des V werden die Metriken bestimmt, welche die geforderten Messgrößen ermitteln.

Der rechte Zweig des V deckt die Anwendungsseite ab. Nachdem die erforderlichen Artefakte einer Architekturbewertung zur Verfügung stehen, werden sie nacheinander ausgeführt, so dass am Ende das Ergebnis bekannt ist.

Abbildung 4.10: VEIA-Vorgehen zur Architekturbewertung [GR07AB]

4.4.2.3 Prototypische Implementierung

Für das VEIA-Projekt wurde ein prototypisches CASE-Tool implementiert [MAN08]. Damit lassen sich Artefakte der Produktlinienbeschreibung, des Funktionsnetzes, und der Softwarearchitektur des VEIA Referenzprozesses modellieren. Ferner werden einige der im Projekt erarbeiteten Metriken für Architekturbewertungen in einer Werkzeugerweiterung unterstützt. Ein anderer Teil der Metrikberechnungen basiert noch auf Tabellenkalkulation mit manueller Datenübernahme.

4.4.2.4 Abgrenzung

Im VEIA-Projekt wird ein Referenzprozess vorgestellt, welcher die Entwicklung und Bewertung von E/E-Architekturen erlaubt. Darin enthalten ist eine Vorgehensweise, wie von abstrakten Fragestellung an eine Architektur konkrete Bewertungen durch Berechnungen durchgeführt werden. Begleitend dazu steht ein prototypisches CASE-Werkzeug zur Verfügung, über welches wesentliche Artefakte einer Architektur modelliert werden und einige der entwickelten Metriken ausgeführt werden. Die Berechnung der restlichen Metriken beruht auf Tabellenkalkulationen.

Im VEIA-Projekt wird strikt zwischen Metrik und Architektur unterschieden. Für Architekturdaten wurde ein Metamodell-basierter Ansatz entwickelt, über welches die wesentlichen Artefakte einer Architektur abgebildet werden. Die Metriken stehen zum Teil als ausführbare Funktion mit Visualisierungsmöglichkeit im Demonstrator zur Verfügung. Der Datenaustausch erfolgt über ein im Projekt spezifiziertes Format. Die Implementierung der restlichen Metriken erfolgt über Tabellenkalkulationsvorlagen, wobei der Datenaustausch manuell erfolgt. Metrikanpassungen oder die Entwicklung neuer Metriken können nur durch Erweiterung des CASE-Tools oder der Tabellenkalkulationsvorlagen umgesetzt werden. Der Ansatz sieht keine graphische Notation und graphische Editiermöglichkeit von Metriken vor.

Die Bildung von Architekturvarianten und Realisierungsalternativen werden durch die Modellbeschreibung durch Bildung von Konfigurationen unterstützt. Für deren automatisierte Auswertung gibt es keine Unterstützung. Dasselbe gilt für Sensitivitätsanalysen und Was-Wäre-Wenn-Szenarien.

4.4.3 Skriptbasierte E/E-Architekturbewertung

In der Dissertation von Freess [FREE07] wird eine skriptbasierte Bewertung von E/E-Architekturen beschrieben. Als Referenzimplementierung dieser Arbeit diente das E/E-KonzeptTool (EEKT), einem frühen Prototypen von PREEvision, welcher im Forschungszentrum Informatik in Karlsruhe [FZI] entwickelt worden ist.

4.4.3.1 Python-Skripte zur Architekturbewertung

Das EEKT basiert auf einer frühen Version des in Kapitel 3.3 vorgestellten Metamodells zur Beschreibung von E/E-Architekturen. Als Programmiersprache für die Skripte zur Architekturbewertung kommt Python [PYT] zum Einsatz, konkret in Form einer Java-basierten Implementierung namens Jython [JYT]. Ein Adapter [GOF95] ermöglicht der Python-Schnittstelle lesenden Modellzugriff. Die Skripte werden als Datei persistiert und vom EEKT aus aufgerufen.

Die Parametrisierung der Evaluierungsskripte erfolgt über weitere Dateien im XML-Format. Die Struktur dieser Dateien wird in Abbildung 4.11 gezeigt. Es handelt sich im Wesentlichen um eine baumartige Struktur bestehend aus Parameterblöcken mit Datensätzen, die Skalare und tabellarischen Werten.

Die Parameter werden über ein eigenes Jythonmodul den Evaluierungsskripten zur Verfügung gestellt.

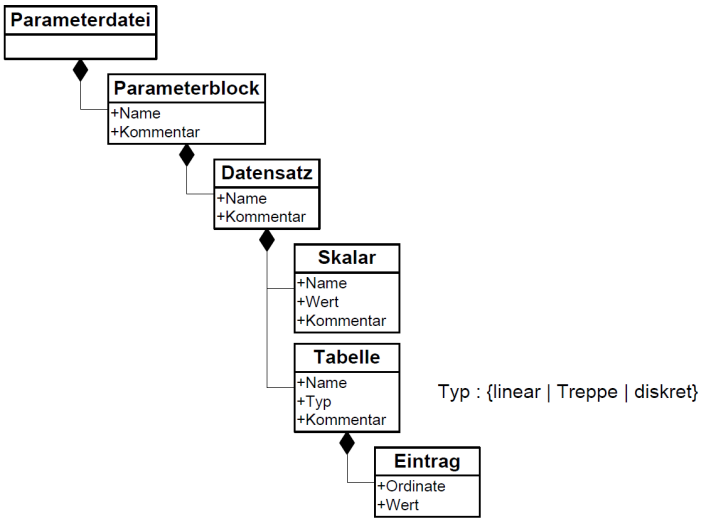


Abbildung 4.11: Struktur einer Parameterdatei [FREE07]

4.4.3.2 Darstellung von Evaluierungsergebnissen

Zur Ausgabe von Evaluierungsergebnissen kommen entweder print-Ausgaben in die Konsole oder Ausgaben in Dateien zum Einsatz. Eine spezifische Ausgabe erfolgt in XML-Dateien. Über eine ID-basierte Referenzierung zu den Modellierungsartefakten wird eine lose Kopplung zwischen Ergebnis und Modellierungsartefakt erreicht.

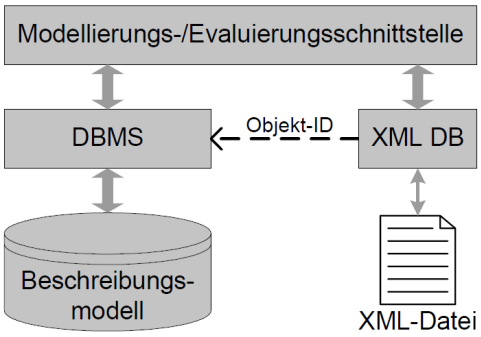


Abbildung 4.12: Ablage von Evaluierungsergebnissen [FREE07]

Dieser Sachverhalt wird in Abbildung 4.12 gezeigt. Auf diese Weise referenzierte Evaluierungsergebnisse werden vom EEKT ausgelesen und konfigurierbar in Editoren als zusätzliche Beschriftung der Modellierungsartefakte dargestellt. Für weiterführende Analysen können die textbasierten Ausgaben in weitere Formate, die mit anderen Werkzeugen kompatibel sind, ausgegeben werden.

4.4.3.3 Abgrenzung

Bei der skriptbasierten Architekturevaluierung mit dem E/E-KonzeptTool handelt es sich um einen hybriden Ansatz. Einerseits werden die E/E-spezifischen Inhalte als Modell (eines MOF-basierten

Metamodells) verwaltet, andererseits werden die Evaluierungsmetriken, die Parametrisierungsdaten und ggf. die Ergebnisse als Dateien abgelegt. Die Verwaltung dieser Dateien obliegt dem EEKT oder den Pythonskripten. Bedingt durch diese Architektur können die Evaluierungsmetriken als partiell in das EEKT integriert gewertet werden.

Die Eingabe der Evaluierungsmetriken erfolgt ausschließlich über Python-Quelltexte. Es gibt keine graphische Notation zur Darstellung der Metriken. Die Ausgabe der Ergebnisse wird ebenfalls über die Pythonskripte organisiert und erfolgt in der Regel textbasiert.

Da das EEKT die Bildung von Architekturvarianten und Realisierungsalternativen unterstützt, können diese durch die pythonbasierten Evaluierungsmetriken ausgewertet werden. Allerdings erfolgt die Auswahl der zu einer Variante zugehörigen Artefakte im Python-Quellcode selbst, was die Anzahl der Codezeilen deutlich erhöht. Ein integriertes Verfahren zum durchschalten der zu untersuchenden Varianten und vereinfachende Mechanismen zur Artefaktauswahl sind nicht vorhanden.

Sensitivitätsanalysen oder Was-Wäre-Wenn-Szenarien werden durch diese Arbeit nicht adressiert. Hier sind weitergehende Analysen in externen Werkzeugen mit einer entsprechenden Ergebnisdatenkopplung denkbar.

4.4.4 SPEEDS

SPEEDS (SPEculative and Exploratory Design in System Engineering) [SPEE] ist ein von der Information Society Technologies gefördertes Projekt, um eine neue Generation von Methodiken, Prozessen und Hilfswerkzeugen für das Design sicherheitsrelevanter, eingebetteter Systeme zu definieren.

Kernpartner des Projektes kommen aus der Wissenschaft (OFFIS, PARADES, Verimag, INRIA), der Luftfahrt- (Airbus, SAAB, IAI), Automobil- (Daimler, Bosch, Knorr Bremse, Magna Powertrain) und Softwareindustrie (Esterel Technologies, Extessy, Telelogic, TNI).

Ziel des Projektes ist die Entwicklung eines Modellierungsformalismus (Heterogeneous Rich Component formalism HRC) skalierbarer, modularer Analysemethoden für den komponentenbasierten Entwurf. Zudem wird ein spekulativer Designprozess, bei dem mehrere Teams parallel an einem Projekt arbeiten und Annahmen über Subsysteme getroffen werden, die von anderen Teams entwickelt werden.

Eine Komponente innerhalb einer HRS definiert sich als Satz angenommener und garantierter Zu-

sicherungen. Diese Zusicherungen können in Kategorien eingeteilt werden, bei denen funktionale und nicht-funktionale (Zeitverhalten, Zuverlässigkeit, Ressourcenverbrauch, etc.) Eigenschaften des Komponentenverhaltens beschrieben werden.

Der HRC-Formalismus setzt auf existierende Standards auf (UML und SysML der OMG). Die Implementierung erfolgt als Eclipse-Plugins unter Berücksichtigung moderner Metamodellierungs-Technologien [SPE10]. Es sind Schnittstellen zu existierenden IDE-Werkzeugen wie SCADE (Esterel Technologies), Rhapsody (Telelogic / IBM), RT-Builder (TNI) und MatLab/Simulink geplant.

4.4.4.1 Das SPEEDS-Metamodell

Das SPEEDS-Metamodell definiert die Sprache zur Modellierung von Systemen gemäß des Rich Component Models. Dieses Metamodell muss einerseits mächtig genug sein, um alle Modellierungsaspekte des SPEEDS-Prozesses abzudecken. Andererseits soll es so einfach wie möglich gestaltet werden, um leicht verständlich für Benutzer und umsetzbar für Analysewerkzeuge zu sein.

Das SPEEDS-Metamodell ist in drei Ebenen aufgeteilt, die zum Teil durch eigenständige Metamodelle repräsentiert werden:

- L-0-Metamodell: Kernebene (*core level*): Dieses Metamodell definiert Kernkonzepte der Modelle. Die Semantik basiert auf ESM's (Extended State Machine). Das Metamodell ist unabhängig von Analyse- und COTS-Werkzeugen. Die Kernebene bestimmt die Ausdrucksstärke des kompletten SPEEDS-Metamodells. Die Metamodelle der übergeordneten Schichten müssen eindeutig auf dieses Metamodell abbildbar sein.
- L-1 Metamodell: Analysewerkzeug-Ebene (*analysis tool level*): Dieses Metamodell erweitert die Kernebene durch abgeleitete Konzepte wie Ansichten, Templates und Koordinationsmechanismen. Durch direkte Unterstützung der abgeleiteten Konzepte dieses Metamodells können Analysewerkzeuge effizienter arbeiten. Aus einem L-1 Modell kann ein äquivalentes L-0 Modell transformiert werden.
- L-2 Bibliothek: COTS-Werkzeug-Ebene (*COTS tool level*): Diese Ebene fügt dem L-1 Metamodell Standard-Bibliothekselemente aus COTS-Werkzeugen hinzu. Diese spezifischen Elemente werden mit Hilfe von Konzepten der L-1-Ebene beschrieben.

Die Zusammenhänge der unterschiedlichen Ebenen werden in Abbildung 4.13 dargestellt. Während die L-1-Ebene durch eine Transformation in ein L-0-Modell überführt werden kann, besteht die Verbindung zwischen der COTS-Werkzeug-spezifischen L-2- und der allgemeinen L-1- Ebene

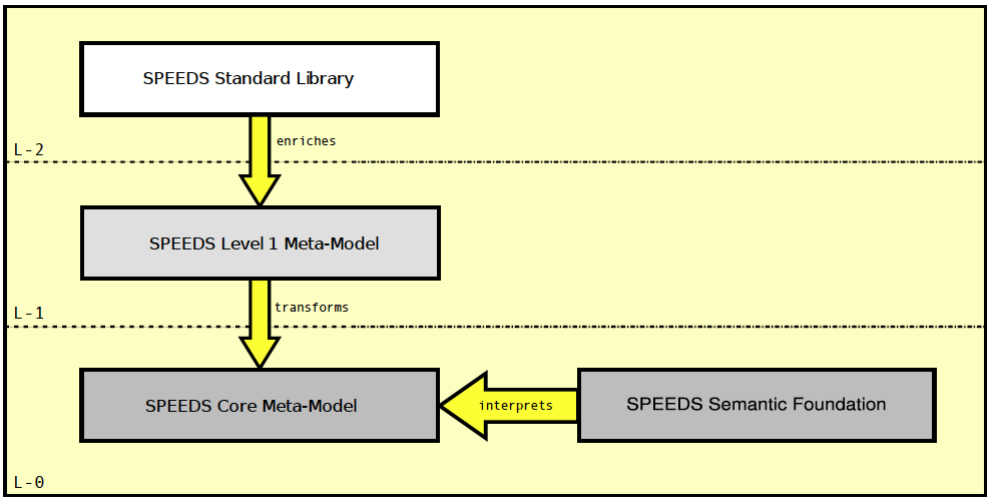


Abbildung 4.13: Die Schichten des SPEEDS-Metamodells [LINK SPEEDS MM]

durch eine Erweiterung innerhalb der Syntax der L-1-Ebene.

4.4.4.2 Die L-1 Ebene

Das Paketdiagramm in Abbildung 4.14 zeigt die wichtigsten Pakete der L-1-Ebene des SPEEDS-Metamodells. Im wesentlich ist das Metamodell die Vereinigung der Pakete *kernel*, *Rich connectors* und *Priorities*.

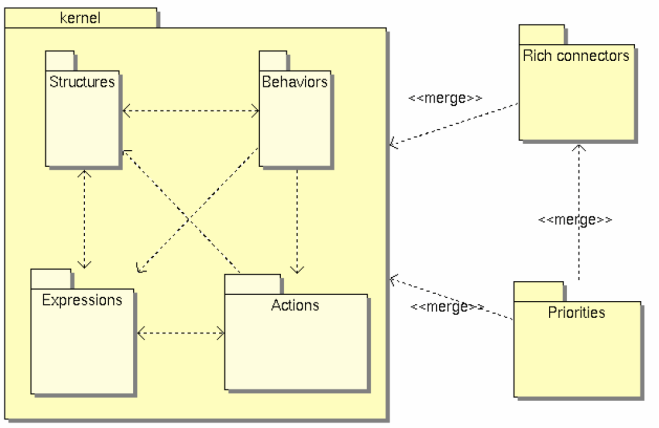


Abbildung 4.14: Paketstruktur des SPEEDS L-1-Metamodells [SPE10]

Das *kernel*-Paket spezifiziert notwendige Grundlagen des Metamodells. Im Subpaket *Structures* werden strukturelle Basiskonzepte und die wichtigsten (meist abstrakten) Superklassen definiert. Datentypen und Datenstrukturen werden im *Expressions*-Subpaket beschrieben. Das *Behaviors*-Subpaket spezifiziert Verhaltensaspekte wie Initialisierungen, Service-Implementierungen, Contracts, Zusicherungen (Assertions), usw. Im *Actions*-Subpaket wird spezifiziert, wie unterschiedliche Aktionen, deren Parameter und deren Rückgabewerte zu modellieren sind. Die Pakete *Rich*

Connectors und *Priorities* werden für das L-1-Metamodell mit dem *kernel* verschmolzen. *Rich connectors* beinhaltet Konzepte, um die Koordination und Kommunikation von Komponenten auszudrücken. Das Subpaket *Priorities* definiert Filtermechanismen und Ablaufsteuerungen.

Im Folgenden werden Teile des L-1-Metamodells detaillierter vorgestellt, in denen es partielle semantische Überschneidungen zum in dieser Arbeit entwickelten Metrik-Metamodell (siehe 6.4) gibt. Daher wird dieser Teil ausführlicher vorgestellt. Eine Abgrenzung dieser Arbeit zu SPEEDS erfolgt in Abschnitt 4.4.4.3. Die hell eingefärbten Klassen sind die innerhalb des jeweiligen Diagramms relevanten Klassen. Die dunkel hinterlegten Klassen bilden den Kontext zum Verständnis der relevanten Klassen.

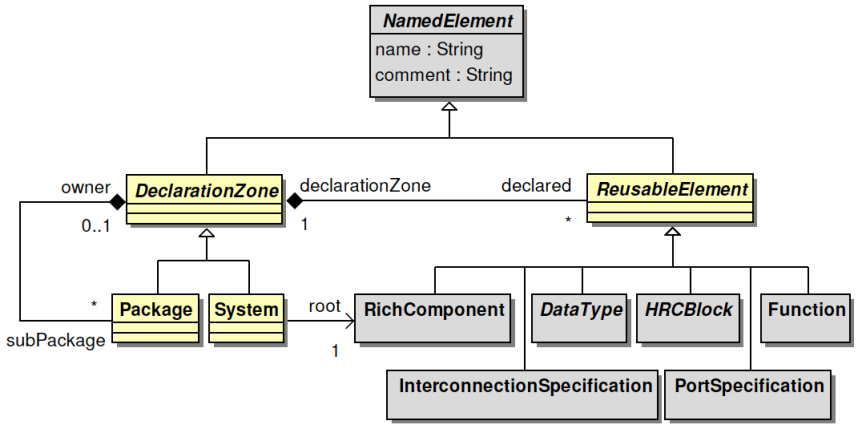


Abbildung 4.15: Pakete im SPEEDS-Metamodell

In Abbildung 4.15 wird das Paketkonzept dargestellt. Ein Paket (*Package*) ist eine *DeclarationZone*, welche weitere Pakete oder Systeme enthalten kann. Ein Paket dient lediglich einer hierarchischen Ordnungsstruktur und nimmt keine Partitionierungsentscheidungen vorweg. Die *DeclarationZone* bildet ein Modul, welches wiederverwendbare Elemente (*ReusableElement*), zum Beispiel ein *HRCBlock*, kapselt.

Ein *System* ist eine *DeclarationZone*, welche zusätzlich eine *RichComponent* als Wurzel (*root*) kennt. Diese Wurzel repräsentiert die Top-Level-Komponente des Systems. Das bedeutet, dass das System instanziiert werden kann, indem die Wurzelkomponente und all seine Subkomponenten (rekursiv) instanziiert werden.

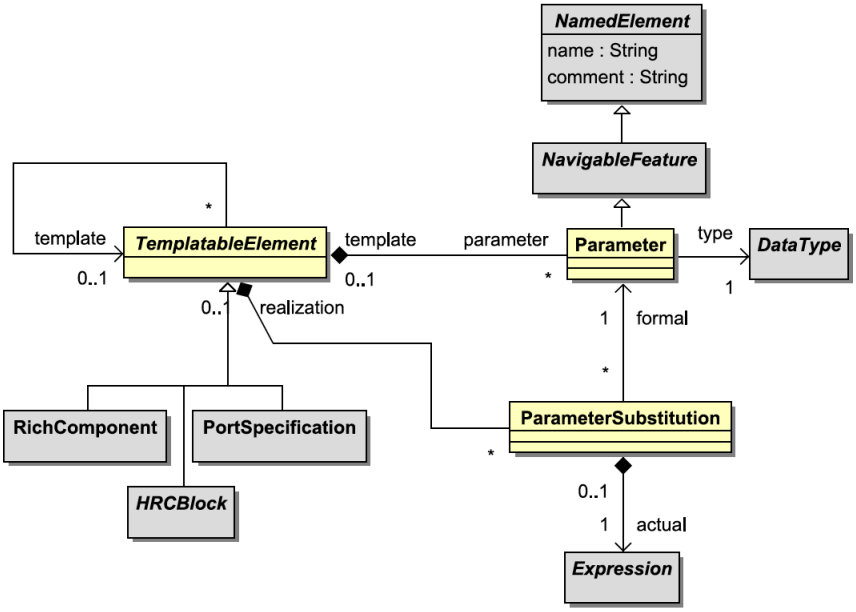


Abbildung 4.16: Templates im SPEEDS-Metamodell

In SPEEDS können bestimmte Elemente, wie zum Beispiel *RichComponent*, *HRCBlock* oder *PortSpecification* als Template definiert werden. Abbildung 4.16 zeigt dieses Konzept. Ein Template (*TemplateableElement*) definiert eine wiederverwendbare, parametrierbare (*Parameter*) Familie von Modellelementen. Mitglieder der Familie unterscheiden sich nur in der Parametrisierung (*ParameterSubstitution*), welche die Werte für die vorgegebenen *Parameter* liefert.

Eine *RichComponent* (siehe Abbildung 4.17) beschreibt eine strukturelle Einheit im Modell, welche die gleiche Zusammensetzung aus Attributen (*Variable*), Ports (*Port*), Teilen (*RichComponent-*

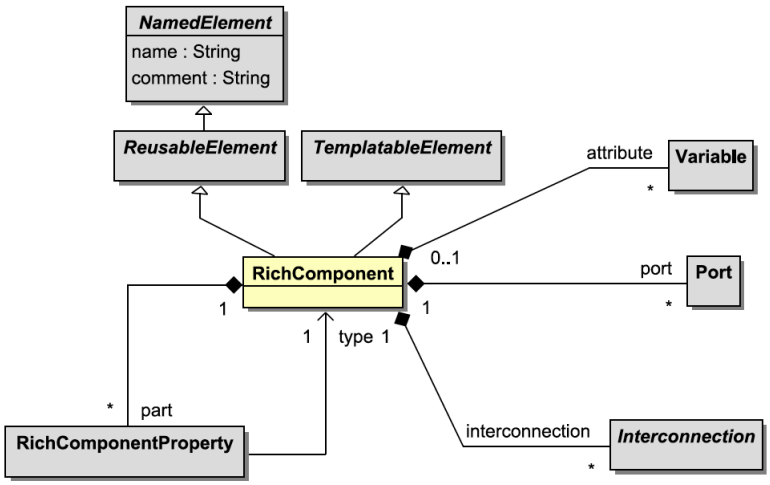


Abbildung 4.17: Struktur einer RichComponent im SPEEDS-Metamodell

Property) und Verbindungen (*Interconnection*) zu anderen Komponenten haben. In den *Variable*

werden die Attribute der Komponente abgelegt. Diese verfügen über jeweils eine Assoziation zu Datentypen (nicht im Diagramm dargestellt), damit sind die abgelegten Werte eindeutig typisiert. Die Metaklasse *Port* realisiert das Port-Konzept (siehe Abschnitt 2.5.5) in SPEEDS.

Die *RichComponent* ist sowohl ein wiederverwendbares (*ReusableElement*), als auch ein parametrisierbares (*TemplatableElement*) Element.

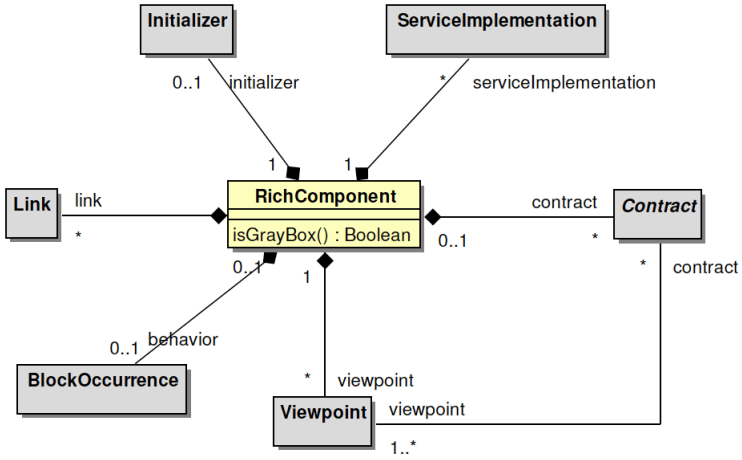


Abbildung 4.18: Verhaltensbeschreibung von RichComponents im SPEEDS-Metamodell

Zusätzlich zu seiner strukturellen Beschreibung verfügt eine *RichComponent* auch über eine Beschreibung der Verhaltensaspekte, wie in Abbildung 4.18 dargestellt. Die Metaklasse *Link* realisiert ein allgemeines Konzept, um die *RichComponent* mit anderen Artefakten zu verbinden. Diese können Attribute, Pins, Datenflüsse oder Services sein. Über den *Initializer* werden die Anfangswerte der *RichComponent* gesetzt. Die *ServiceImplementation* stellt die Implementierung eines Service der *RichComponent* zur Verfügung. Zusicherungen (*Contract*) und *Viewpoints* erlauben unterschiedliche Sichtweisen auf die *RichComponent*, wobei jede Sichtweise über ein Set von Zusicherungen verfügt, die für die Sichtweise relevant sind. Zusicherungen werden unterschieden in atomare Zusicherungen und zusammengesetzte Zusicherungen, so dass komplexe Strukturen abgebildet werden können. Das Verhalten der *RichComponent* wird über das Konzept *BlockOccurrence* beschrieben, die im folgenden näher beschrieben wird.

Dies erfolgt durch Modellierung von Zustandsautomaten (StateMachine) oder Funktionsaufrufen (FunctionCall, sind einer nicht dargestellten Funktion zugeordnet, welche eine seiteneffekt-freie berechenbare Einheit repräsentiert). Bei der Initialisierung können optionale Anfangswerte (*Initializer*) für Eigenschaften und Subkomponenten der *RichComponent* gesetzt werden. Eine *ServiceImplementation* bildet die Implementierung einer Funktion auf der Komponente. Über eine (hier nicht

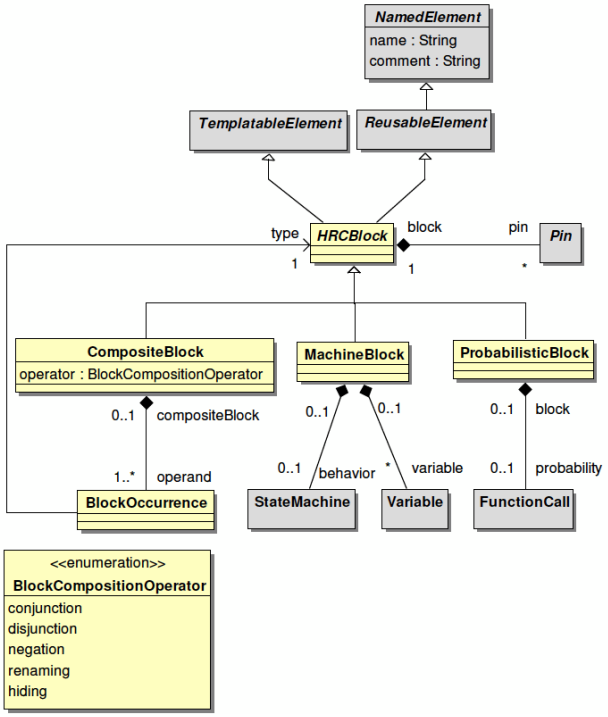


Abbildung 4.19: HRC-Blöcke

dargestellte) Beziehung zu einem *Service* wird die Deklaration der implementierten Funktion beschrieben.

4.4.4.3 Abgrenzung

SPEEDS ist ein Projekt, um eine neue Generation von Methodiken, Prozessen und Hilfswerkzeugen für das Design sicherheitsrelevanter, eingebetteter Systeme zu definieren. Ziel ist die Entwicklung eines Modellierungsformalismus skalierbarer, modularer Analysemethoden für den komponentenbasierten Entwurf.

Ein Bestandteil von SPEEDS ist ein spekulativer Designprozess, bei dem mehrere Teams parallel an einem Projekt arbeiten. Annahmen und Zusicherungen für Subsysteme werden an den Schnittstellen von Komponenten definiert, die von anderen Teams entwickelt werden. Über diesen Mechanismus lassen sich Bewertungen ableiten, wenn auch nicht dafür vorgesehen. Hierbei handelt es sich um eingebettete Berechnungsvorschriften zur Prüfung von vereinbarten Schnittstellen und Übergabeparametern. Die Berechnungsvorschriften sind fest mit den Komponenten verwoben und kommen nur dort zur Anwendung. Bei diesem Ansatz handelt es sich nicht um eine explizite Methodik zur Bewertung und Optimierung von Komponenten.

Die Bildung von Architekturvarianten und Realisierungsalternativen wird von SPEEDS nicht unter-

stützt, ebensowenig wie Sensitivitätsanalysen und Was-Wäre-Wenn-Szenarien.

4.4.5 UML MARTE

UML MARTE (Modeling and Analysis of Real-Time and Embedded systems) [MARTE] ist ein Standard der Object Management Group [OMG] zur Modellierung und Analyse von Echtzeit- und eingebetteten Systemen in Hard- und Software.

Die Projektteilnehmer von MARTE stammen aus Wissenschaft (Carleton University, Commissariat à l'Energie Atomique, ESEO, ENSIETA, INRIA, INSA aus Lyon, Software Engineering Institute (Carnegie Mellon University), Universidad de Cantabria), aus der Industrie (Alcatel, France Telecom, Lockheed Martin, Thales) und aus dem Bereich der Softwarehersteller (ARTiSAN Software Tools, International Business Machines, Mentor Graphics Corporation, Softeam, Telelogic AB (I-Logix), Tri-Pacific Software, No Magic, The Mathworks) [MARTUT07].

Die Ziele des Projektes sind:

- Bessere Kommunikation zwischen den Entwicklern durch Verwendung eines gemeinsamen Standards
- Sicherstellung der Interoperabilität der beteiligten Werkzeuge für Spezifikation, Design, Verifikation, Codegenerierung, etc.
- Quantitative Vorhersagen der Echtzeit- und eingebettete Eigenschaften der modellierten Systeme unter Berücksichtigung der spezifischen Kriterien für Hard- und Software

4.4.5.1 Die MARTE-Architektur

MARTE ist als Profil auf Basis der UML spezifiziert. Das Profil besteht, wie in Abbildung 4.20 dargestellt, aus einem allgemeinen Teil (*MARTE foundations*), Erweiterungen für Design- und Analysemodelle (*MARTE design model* / *MARTE analysis model*), sowie diversen Zusätzen (*MARTE annexes*).

Im allgemeinen Teil werden neben einigen Kernelementen Erweiterungen für nicht-funktionale Attribute (Profil NFP – Non Functional Property, diese Attribute beschreiben die „Fitness" des Systemverhaltens wie zum Beispiel Performanz, Speicherbedarf oder Energieaufnahme [MARTUT07]), zur Zeitbeschreibung (Profil Time), für ein generische Ressourcenmodellierung (Profil GRM – Generic Resource Modeling) und zur Allokation (Profil Alloc).

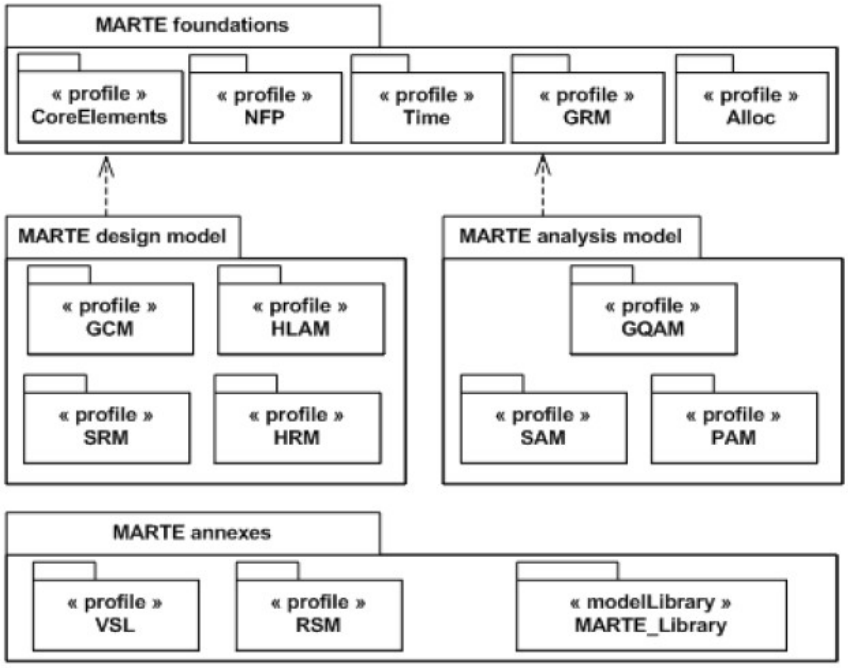


Abbildung 4.20: Architektur des MARTE-Profils [MART11]

Das Designmodell beinhaltet Erweiterungen für ein generisches Komponentenmodel (Profil GCM – Generic Component Model), für eine High-Level Applikationsmodellierung (HLAM – High-Level Application Modeling) sowie für eine detaillierte Ressourcenmodellierung, bestehend aus einem Software- (Profil SRM – Software Resource Modeling) und einem Hardwareanteil (HRM – Hardware Resource Modeling). Für die konkreten Details des allgemeinen Teils und des Designmodells sei hier auf die MARTE-Spezifikation verwiesen [MART11].

Das Analysemodell besteht aus den Erweiterungen für die generische, quantitative Analysemodellierung (Profil GQAM - Generic Quantitative Analysis Modeling), der Scheduling-Analysemodellierung (Profil SAM – Schedulability Analysis Modeling) und die Performanz-Analysemodellierung (PAM - Performance Analysis Modeling). Diese Erweiterungen werden im folgenden Abschnitt näher betrachtet.

4.4.5.2 Systemanalysen mit MARTE

Mit den MARTE-Erweiterungen lassen sich, auf Basis der UML, eingebettete und Echtzeitsysteme beschreiben. Diese werden mit um Datenwerte erweiterte Stereotypen versehen, welche dann zur Analyse herangezogen werden. Zum Einsatz kommen die aus der UML gängigen Diagrammarten.

In Abbildung 4.21 ist ein Sequenzdiagramm mit MARTE-Erweiterungen dargestellt. Es zeigt eine einfache Web-Applikation als Beispiel. Die MARTE-Erweiterungen sind als Stereotypen, die mit

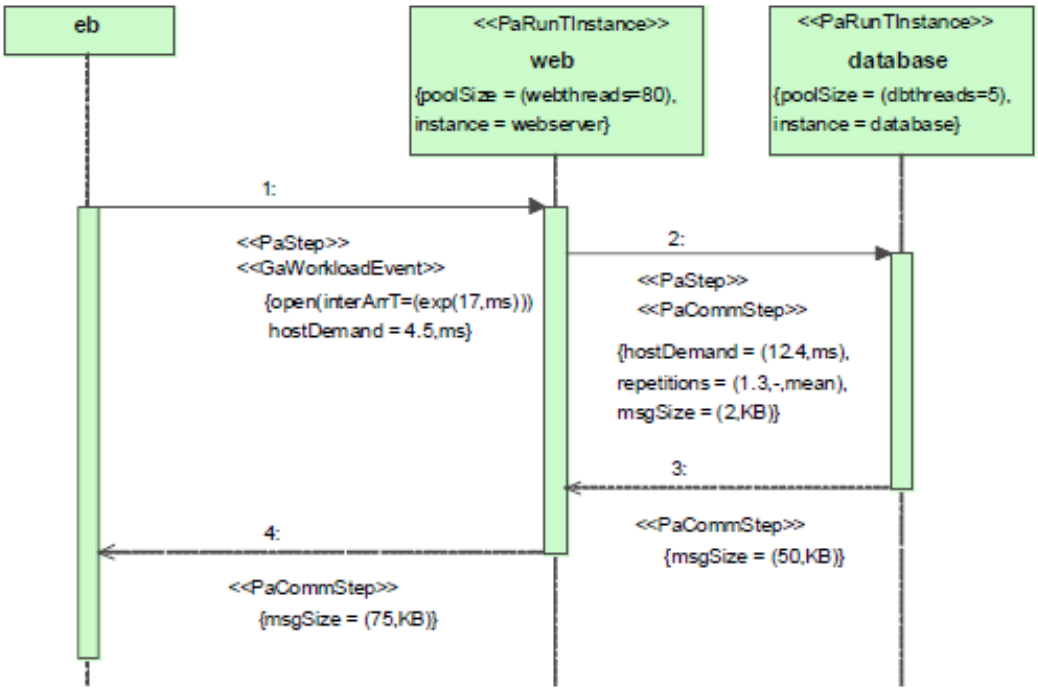


Abbildung 4.21: Beispiel einer einfachen Web-Applikation als Sequenzdiagramm mit MARTE-Erweiterungen [MART11]

zusätzlichen Werten (NFP – Non Functional Property), z.B. über benötigte Antwortzeiten, hinterlegt sind.

Mit Hilfe dieser Informationen lassen sich spezifische Analysemodelle erstellen. Für Performanzanalysen eignet sich beispielsweise ein Warteschlangenmodell wie das Layered Queuing Network [LQN], welches aus der Warteschlangentheorie [WAR94] entstammt.

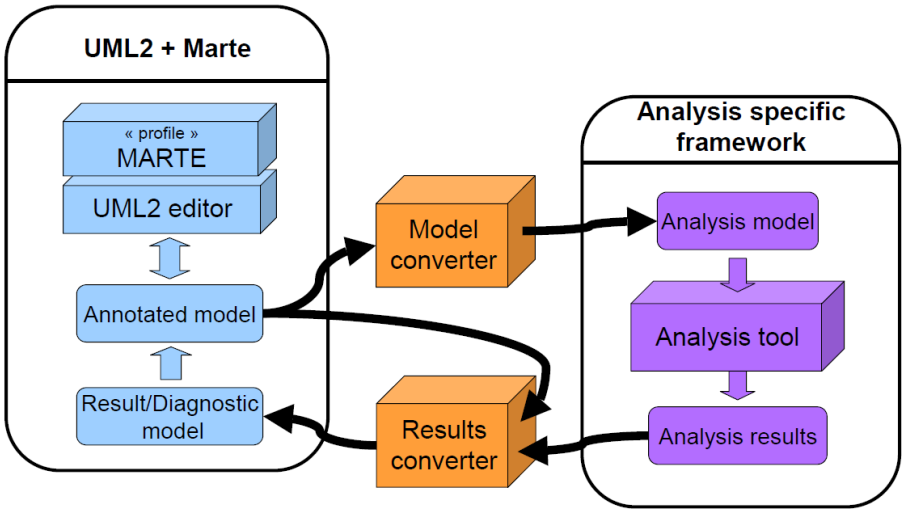


Abbildung 4.22: Modellanalysen mit MARTE

Abbildung 4.22 visualisiert das MARTE-Vorgehen für Modellanalysen. Die Kernidee besteht darin,

die mit UML und MARTE beschriebenen Modelle durch Modellkonverter in Analyse-spezifische Formate zu übertragen und mit spezialisierten Frameworks auswerten zu lassen. Die Ergebnisse werden durch Ergebniskonverter zurück in das UML/MARTE-Modell gespiegelt.

4.4.5.3 Abgrenzung

MARTE ist ein Projekt der OMG, um die Unified Modeling Language UML mit Hilfe von Profilen für den Einsatz von Echtzeit- und eingebetteten Systemen zu erweitern. Die Modellierung erfolgt im Wesentlichen mit Hilfe der UML-Beschreibungsmittel und erweiterte Stereotypen, welche mit zusätzlichen Werten angereichert sind. Die Analyse dieser Modelle wird mit Hilfe von Konvertern an spezifische Analysewerkzeuge übergeben, deren Ergebnisse zurück in das Modell geschrieben werden.

Der Vorteil des Ansatzes ist die weitgehende Kompatibilität mit bestehenden UML-Werkzeugen, da MARTE als formales UML-Profil vorliegt. Da dies gleichzeitig bedeutet, dass im Wesentlichen die modellierten Konstrukte in vorhandene UML-Strukturen abgebildet werden muss, ist dies auch ein Nachteil. Bewertungs- und Analyserelevante Daten werden im Prinzip lediglich als Key-Value-Paare abgelegt. Für die graphische Notation bedeutet dies, dass vor allem die MARTE-Erweiterungen als Stereotypen sehr textlastig dargestellt werden. Größere Diagramme oder viele NFP führen schnell zu Unübersichtlichkeit.

Die Ergebnisaufbereitung und Nachvollziehbarkeit von MARTE-Analysen hängt stark vom eingesetzten, spezifischen Analysewerkzeug ab. Durch die Formulierung von Bedingungen, Parametern und zurückgespielte Ergebnisse als Annotationen sind sehr textlastige Diagramme zu erwarten. Durch die strikte Trennung zwischen Modell und Analyse ist eine direkte Zuordnung von Ergebnissen zu Modellierungsartefakten nur bedingt möglich. Varianten und Realisierungsalternativen werden nicht unterstützt, durch geschickte Modellierungsrichtlinien und -strukturen könnten grundlegende Aspekte realisiert werden. Sensitivitätsanalysen und Was-Wäre-Wenn-Szenarien werden nicht von MARTE adressiert. Je nach verwendetem Analysewerkzeug könnten Teile davon dort durchgeführt werden.

4.4.6 ATESST / ATESST2

ATESST (Advancing Traffic Efficiency and Safety through Software Technology) und ATESST2 [ATE11] sind Folgeprojekte des bekannten EAST-EEA-Projektes [EAS04]. Sie trieben die Weiterentwicklung des EAST-ADL zu EAST-ADL2 [EAS07] voran. Insbesondere wurde hier die Integ-

reation des AUTOSAR-Standards [AUT] erreicht. Die Projektpartner für ATTEST waren Daimler-Chrysler AG (heute Daimler AG), Volvo Cars, Volvo Technology, VW / Carmeq GmbH, ETAS, Mecel, Mentor Graphics, Continental, CEA-LIST, The Royal Institute of Technology sowie die Technische Universität Berlin.

Für ATESST2 kamen das Centro Ricerche Fiat und die University of Hull hinzu. Daimler und ETAS verließen das Konsortium.

4.4.6.1 EAST-ADL

Die EAST-ADL unterscheidet fünf Phasen der Fahrzeugentwicklung, welche in dem MOF-konformen Metamodell durch Abstraktionsebenen abgebildet werden. Eingebettete Systeme werden ebenenübergreifend in den relevanten Abstraktionsebenen beschrieben. Abbildung 4.23 zeigt die EAST-ADL Abstraktionsebenen.

In der Fahrzeugebene (Vehicle Level) wird das technische Feature-Modell (Technical Feature Model, TFM) beschrieben. In der Analyseebene (Analysis Level) werden das Verhalten und die Algorithmen der Funktionen aus der TFM in der funktionalen Analysearchitektur (Functional Analysis Architecture, FAA) modelliert. Aus diesen beiden Ebenen ergibt sich eine genaue Spezifikation des Leistungsumfangs des späteren Fahrzeugs.

Die Entwurfsebene (Design Level) ist unterteilt in eine funktionale (Functional Design Architecture, FDA) und eine hardware-seitige Entwurfsebene (Hardware Design Architecture, HDA). Die FDA ist eine Realisierung der in der FAA beschriebenen Funktionen. Die Modellierung erfolgt mit Hilfe von hierarchisierten Funktionsblöcken. Diese Funktionsblöcke werden als ausführbare, hardwareunabhängige Einheiten auf ausführende Hardwarekomponenten, die in der HDA modelliert werden, verteilt. Dabei werden Hardwareanforderungen aus der FAA und FDA berücksichtigt.

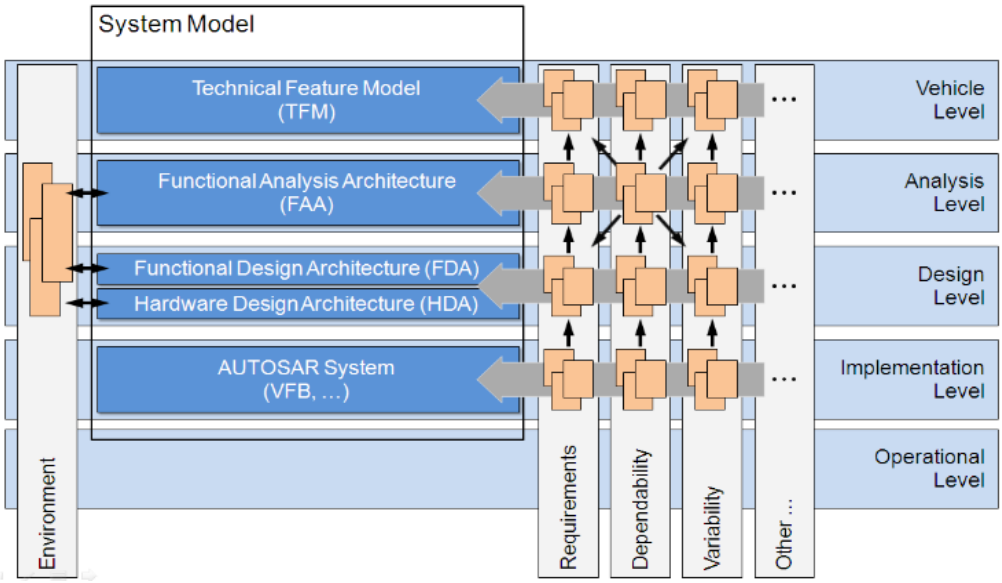


Abbildung 4.23: Abstraktionsebenen EAST-ADL [EAA11]

Die Implementierungsebene (Implementation Level) beinhaltet das Instanziierungsmodell der Funktionen. Durch die weitere Entwicklung der EAST-ADL in den Folgeprojekten ATESST und ATESST2, sowie der Vereinheitlichung mit dem AUTOSAR-Standard, ist in dieser Ebene beispielsweise der AUTOSAR Virtual Function Bus eingeordnet.

Die Verteilung von Funktionen aus der FDA wird im Allokationsmodell, welches in der operationalen Ebene (Operational Level) enthalten ist, indirekt über die Funktionsinstanzen auf Hardwarekomponenten beschrieben.

4.4.6.2 Analyse und Optimierung

Das ATESST2-Projekt setzt sich unter anderem mit einer Analyse-getriebenen Architekturentwicklung und -optimierung auf Basis der EAST-ADL auseinander [ATEANA11]. Dieser Ansatz sieht vor, über externe Analysewerkzeuge Architekturbewertungen und -optimierungen durchzuführen. Hierbei ist einerseits eine automatische Modellkonvertierung in ein für das eingesetzte Analysewerkzeug verständliches Format. Andererseits ist eine Rück-Konvertierung der Ergebnisse in das EAST-ADL-Modell notwendig (siehe Abbildung 4.24).

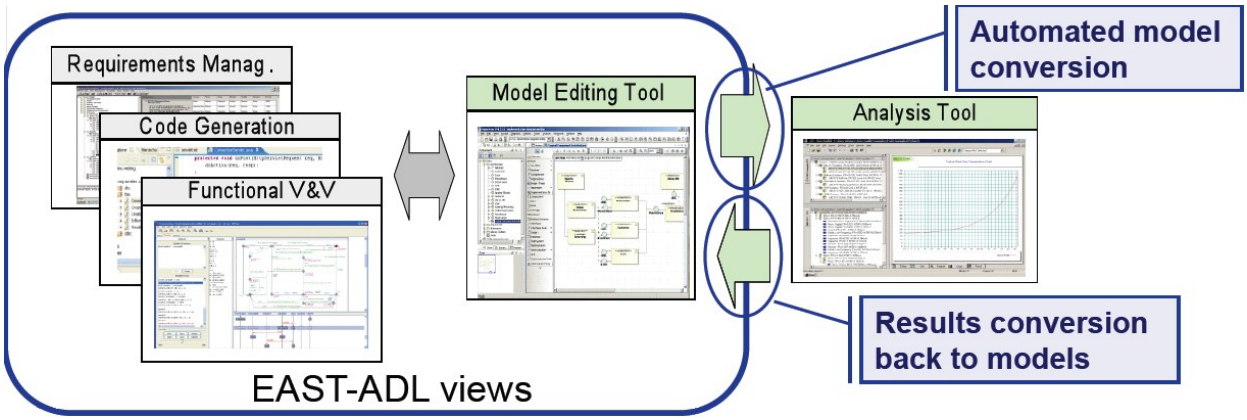


Abbildung 4.24: Analyse und Optimierung von EAST-ADL Modellen nach ATESST2 [ATEANA11]

Passend zu diesem Ansatz wurde eine Methodik [ATEANA10] entwickelt. Sie benötigt den kompletten Anforderungssatz und eine spezifische Architekturbeschreibung. Anschließend werden Anforderungen und Systemkomponenten identifiziert, die für den Evaluierungskontext relevant sind. Für die Methodik sind drei Aktivitäten, wie in Abbildung 4.25 dargestellt, von großem Interesse:

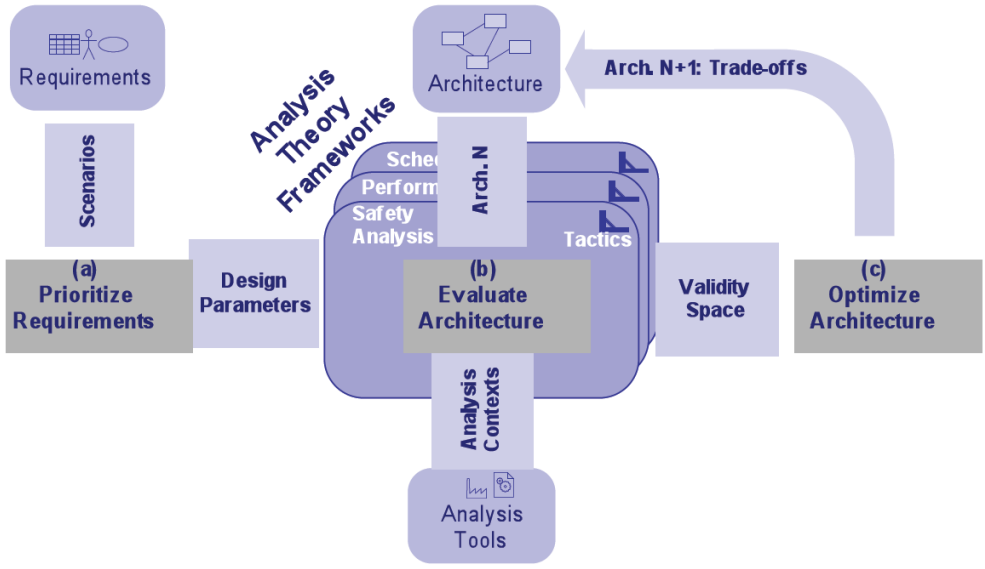


Abbildung 4.25: Methodik für Analyse-getriebene Architekturevaluierung und -optimierung [ATEANA10]

(a) Anforderungen priorisieren: In dieser Aktivität werden für die Analyse relevante Qualitätsparameter bestimmt. Sie repräsentieren Designbedingungen der Evaluierung und beinhalten abhängige und unabhängige Parameter sowie deren Beziehungen [ATEANA10].

(b) Architektur evaluieren: Mit dieser Aktivität wird sichergestellt, dass die Qualitätsattribute der Architektur den Systemanforderungen genügen und zur Analyse herangezogen werden können. Die Evaluierung wird auf den notwendigen Modellsichten ausgeführt, ggf. werden neue Sichten hinzugefügt. Die Ergebnisse der Evaluierung werden dem Modell annotiert und stehen für Optimierun-

gen zur Verfügung [ATEANA10].

(c) Architektur optimieren: Diese Aktivität findet die beste anwendbare Lösung aus dem Suchraum heraus. Diese Lösung berücksichtigt die vorgegebenen Optimierungsziele aus Aktivität (a) und die zu beachtenden Randbedingungen [ATEANA10]. Ergebnisse der Optimierung werden auf die Architektur angewendet. Somit entsteht eine Nachfolgearchitektur, welche wiederum für Analyse, Evaluierung und Optimierung zur Verfügung steht. Als potentielle Optimierungsalgorithmen wurden Multi-Kriterien-Optimierer auf Basis genetischer Algorithmen untersucht [PAR10].

4.4.6.3 Abgrenzung

Die EAST-ADL dient zur Modellierung von Elektrik/Elektronik-Architekturen im Fahrzeug. Im Vordergrund steht vor Allem die Trennung zwischen vorwiegend in Software umgesetzte Funktionen und Hardware. In diesem Kontext wurde in den ATESST / ATESST2-Projekten eine Methodik für eine Analysegetriebene Architekturevaluierung und -optimierung vorgestellt. Darin sollen über entsprechende Konverter die Architekturdaten von externen Analyse- und Optimierungswerkzeuge weiterverarbeitet und zurückgeschrieben werden. Die zu optimierenden Daten werden als Qualitätsattribute dem Architekturmodell annotiert.

In der EAST-ADL werden keine spezifischen Werkzeuge zur Modellierung, Analyse und Optimierung gefordert. Über Konverter wird eine lose Kopplung der Werkzeuge erreicht. Dadurch sind die zu analysierenden / optimierenden Architekturdaten lose mit den Ergebnissen gekoppelt. Die Nachvollziehbarkeit der Ergebnisse hängt von der Transparenz von den Transformationsregeln ab. Für die Analyse- und Optimierungswerkzeuge stehen die Architekturdaten zur Verfügung, die von der Transformation erfasst / unterstützt werden. Erweiterungen haben demnach Anpassungen an den Transformationsregeln zur Folge.

Die zur Analyse und Optimierung eingesetzten Metriken sind stark abhängig von den angebundenen Werkzeugen.

4.4.7 MAENAD

Das Akronym MAENAD [MAE11] steht für Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles und ist ein von der Europäischen Union gefördertes Projekt im siebten Forschungsrahmenprogramm [FRP7]. Ziel des Projektes ist die EAST-ADL beson-

ders für die Bedürfnisse zur Entwicklung vollelektrischer Fahrzeuge weiterzuentwickeln. Darüber hinaus engagiert sich das Projekt bei der Harmonisierung und Erweiterung der EAST-ADL mit AUTOSAR sowie anderen Projekten, welche die EAST-ADL verwenden. Ein weiterer Schwerpunkt ist die Analyse und Optimierung von EAST-ADL-Modellen. Dazu greift MAENAD auf Ergebnisse anderer Projekte aus dem EAST-Umfeld (z.B. ATESST / ATESST2, beschrieben in Abschnitt 4.4.6) zurück und erweitert sie. Zudem wird der MARTE-Standard der OMG (siehe Abschnitt 4.4.5) eingebunden.

Am MAENAD-Projekt sind folgende Organisationen und Unternehmen beteiligt: Volvo Technology, Centro Ricerche Fiat, Delphi/Mecel, 4S S.r.l., MetaCase, Pulse-AR, Systemite, Commissariat a l'Energie Atomique, Kungliga Tekniska Högskolan, Technische Universität Berlin, University of Hull. Das Projekt startete im September 2010 mit einer Laufzeit von 42 Monaten.

4.4.7.1 EAST-ADL-Analysen mit MAENAD

Zur Analyse von EAST-ADL-Modellen sieht MAENAD die Einbindung externer Werkzeuge, sogenannten Gateways, vor. Abbildung 4.26 zeigt das dazugehörige Konzeptschaubild.

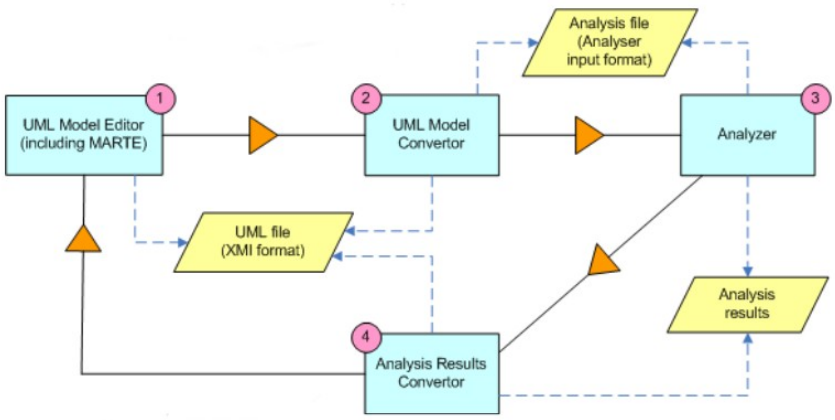


Abbildung 4.26: Architekturanalysen mit MAENAD [MAEAW]

Der Ausgangspunkt ist ein EAST-ADL-Modellierungswerkzeug mit MARTE-Unterstützung (1). Konkret wird an dieser Stelle im MAENAD-Projekt das UML-Werkzeug Papyrus [PAPY] mit MAENAD-spezifischen und anderen Erweiterungen zum Einsatz.

Über einen Modellkonverter (2) wird das EAST-ADL-Modell in ein Analysewerkzeugspezifisches Format überführt. Im MAENAD-Projekt wurden Gateways für unterschiedliche Werkzeuge entwickelt [MAED5], zum Beispiel:

- AUTOSAR-Gateway: Anbindung zu einer AUTOSAR-konformen Softwarearchitektur [AUT].

- Timing Analysis-Gateway: Timing-Analysen EAST-ADL-Modellen in frühen Entwicklungsphasen
- Simulink-Gateway: Anbindung von EAST-ADL-Modellen zur Simulation mit Simulink [SIM].
- HiP-HOPS-Gateway: Anbindung an das Sicherheitsanalyse- und Optimierungswerkzeug Hip-HOPS [HIP].
- UPPAAL&SPIN-Gateway: Analyse von Verhaltensbedingungen mit den Analysewerkzeugen UPPAAL [UPP] und spin [SPIN].

Die eigentlichen Analysen und Optimierungen bleiben dem Analysetool (3) vorbehalten. Deren Ergebnisse werden über den Rückkanal der Gateways (4) an das EAST-ADL-Modellierungswerkzeug (1) zurück geschrieben.

4.4.7.2 Abgrenzung

MAENAD vereint mehrere Projekte und Standards wie EAST-ADL, ATESST/ATESST2 und UML MARTE bzw. erweitert diese. Die grundsätzliche Architektur bleibt, so dass Bewertungen, Analysen und Optimierungen durch externe Werkzeuge angebunden werden. Im MAENAD-Projekt wurden dazu Gateway-Schnittstellen implementiert. Ergebnisse werden zurück in das Ausgangsmodell geschrieben. Erweiterungen für Bewertung, Analyse und Optimierung haben Anpassungen an den Gateways zur Folge. Die Nachverfolgbarkeit der Ergebnisse hängt stark von der Transparenz der bidirektionalen Kopplung zu den externen Werkzeugen ab, ebenso die Auswahl und Anpassbarkeit der eingesetzten Metriken.

Die graphische Repräsentation der Architekturdaten, der eingesetzten Metriken und deren Ergebnisse werden nicht von MAENAD adressiert sondern durch den Einsatz von UML MARTE (vgl. Abschnitt 4.4.5) behandelt. Die automatisierte Bewertung und Analyse von Varianten und Realisierungsalternativen wird ebensowenig unterstützt wie Sensitivitätsanalysen und Was-Wäre-Wenn-Szenarien.

5 Anforderungen an ein Frameworks zur Bewertung, Analyse und zum Vergleich von E/E-Architekturen

5.1 Konzeption

Der Schwerpunkt dieser Arbeit ist das Framework zur Bewertung, Analyse und zum Vergleich von Elektrik/Elektronik-Architekturen. Dazu ermöglicht das Framework die Entwicklung, Ausführung, Dokumentation und Wartung von Logiken, über welche die Bewertung, Analyse und der Vergleich von E/E-Architekturen durchgeführt werden. Diese Logiken werden im Kontext dieser Arbeit auch verkürzend „Metrik“ oder „Bewertungsmetrik“ genannt[14] und sind in Abbildung 5.1 dargestellt. Das dieser Arbeit zugrundeliegende Konzept konzentriert sich vor allem auf die Verständlichkeit und Nachvollziehbarkeit der Berechnungsvorschriften selbst, deren Entwicklung, deren Ausführung und vor allem deren Ergebnisse und Ergebnisaufbereitung.

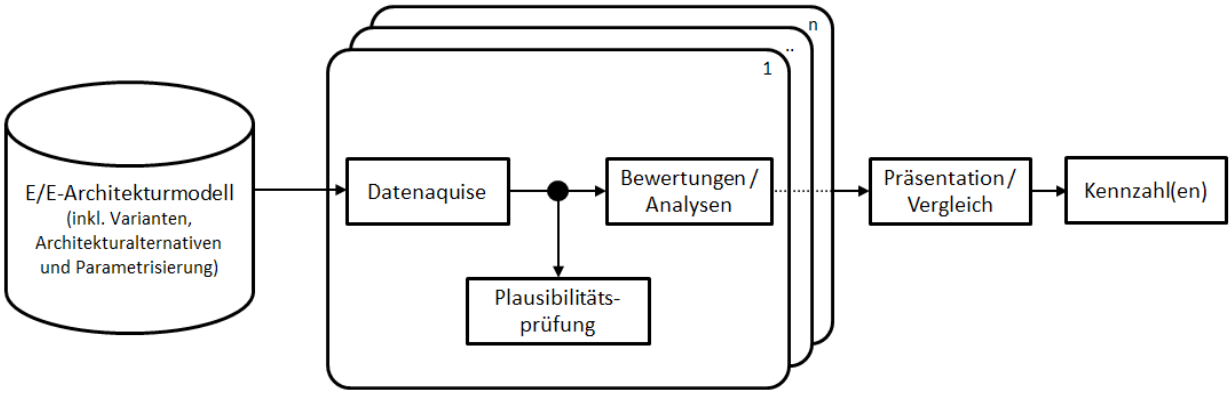


Abbildung 5.1: Konzept zum Ablauf von Bewertungen, Analysen und Vergleichen

Ausgehend von einer einheitlichen Datenbasis des E/E-Architekturmodells inklusive Variantenbeschreibungen, Realisierungsalternativen und Parametrisierungen werden Datenaquisen angewendet. Deren Ergebnisse werden einer Plausibilitätsprüfung unterzogen und den Bewertungs- und Analyseverfahren zugeführt. Diese Schritte können mit unterschiedlichen Kontexten mehrfach ausgeführt werden, beispielsweise um mehrere relevante Architekturvarianten zu bewerten oder zu vergleichen. Nach der Ausführung der Bewertungen und Analysen kommt es zur Präsentation der Ergeb-

14 Ursprünglich wurde das im Rahmen dieser Arbeit zur reinen Bewertung von E/E-Architekturmodellen über Metriken konzipiert. Es zeigte sich jedoch früh, dass mit diesem Konzept neben Bewertungen auch Analysen, Vergleiche und Optimierungen von E/E-Architekturen durchführbar sind. Um den Rahmen dieser Arbeit nicht zu sprengen, wurde das Themengebiet der E/E-Architekturoptimierungen ausgespart.

nisse oder des Vergleichs sowie zur Berechnung einer oder mehrerer Kennzahlen (vgl. Abschnitt 4.3.1). In diesem Kapitel werden die Anforderungen an ein solches Framework diskutiert und mit dem aktuellen Stand der Technik gespiegelt.

Die wichtigen Punkte sind die modellbasierte Beschreibung (Abschnitt 5.2) und graphische Notation (Abschnitt 5.3) von Metriken, eingebettet in ein integriertes Framework zur Darstellung, Bedienung und Ausführung (Abschnitt 5.4) sowie zur Aufbereitung der Metrikergebnisse (Abschnitt 5.5). Ein weiterer wichtiger Aspekt ist die Sensitivitätsanalyse und die Durchführung von Was-wäre-Wenn-Szenarien (Abschnitt 5.6).

Das Kapitel schließt mit der Spiegelung des aktuellen Standes der Technik (Abschnitt 5.8), der systematischen Zusammenfassung der Anforderungen (Abschnitt 5.9) und die weitere Vorgehensweise, die sich aus den Anforderungen ergibt.

5.2 Modellbasierte Beschreibung

Die Beschreibung von Bewertungsmetriken soll modellbasiert durch die Spezifikation eines Metamodells erfolgen. Damit wird eine domänenspezifische Sprache (DSL, Domain Specific Language, vgl. Abschnitt 2.2.3) zur modellbasierten Beschreibung von Bewertungen, Analysen und Vergleichen geschaffen. Der DSL-Ansatz bildet zusammen mit der in Abschnitt 5.3 beschriebenen graphischen Notation den Grundstein für Verständlichkeit und Nachvollziehbarkeit der Berechnungsvorschriften.

Die Entwicklung dieser DSL erfolgt gemäß der Spezifikation des Vier-Schichten-Modells der OMG (Abschnitt 2.1.1) mit Hilfe eines formalen, MOF-basierten Metamodells (Abschnitt 6.4). Instanzen davon repräsentieren Bewertungsmetriken, welche durch eine entsprechende Ausführungsschicht berechnet werden können. Durch den modellbasierten Ansatz kommen zwei wesentliche Prinzipien zur Reduktion der Komplexität zum Einsatz: Abstraktion und Dekomposition. Durch die formale Metamodellierung erfolgt eine Abstraktion der notwendigen Zusammenhänge bei der Modellierung von Metriken. Im Metrik-Metamodell beinhalten Konzepte zur Dekomposition (Kompositum-Pattern in Abschnitt 2.5.1 im Allgemeinen und z.B. Package-Pattern in Abschnitt 3.2.2.3 im Speziellen) zur Beherrschung von Komplexität nach den divide et impera-Ansatz (*lat. teile und herrsche*).

Durch die formale, MOF-basierte Spezifikation des Metrik-Metamodells wird mit der automatisierten Generierung des ausführbaren Quellcodes eine passende Persistenzschicht mitgeneriert. Damit lässt sich ein modellübergreifender Austausch von Metriken durch Ex- und Importe realisieren.

Das Metamodell für Berechnungsmetriken unterscheidet zwischen unterschiedlichen Artefaktarten. Die Metrikblöcke sollen ausführbare Einheiten abbilden. Es soll für die unterschiedlichen Anwendungsfälle wie Datenaquise, Berechnung und Ergebnisaufbereitung unterschiedliche, konkrete Submetaklassen geben. So soll es beispielsweise einen Berechnungsblock geben, der in der Lage ist, Java-Quelltexte auszuführen. Ferner soll es generische Metrikblöcke geben, deren Implementierung von optionalen, installierbaren Komponenten bereitgestellt wird. Diese generischen Blöcke (Abschnitt 6.2.13) erlauben es, den Metrikeditor vor Ort bei Anwendern um internes Wissen und Verfahren anzureichern.

Der Datenaustausch der Metrikblöcke soll über ein Port-Konzept (Abschnitt 2.5.5) realisiert werden, die den Metrikblöcken zugeordnet sind, wobei explizit zwischen Eingangs- und Ausgangsport unterschieden werden soll. Die Ports sollen im Metamodell mit Hilfe von Datenflüssen verbunden werden. Ein Datenfluss soll mit genau einem Eingangs- und beliebig vielen Ausgangsports verbunden werden können.

5.3 Graphische Notation

Für die Beschreibung von Bewertungsmetriken soll eine graphische Notation, umgesetzt durch einen Diagrammeditor, entworfen werden. Der Ablauf von Metriken ist damit leichter verständlich und einzugebende Quelltexte werden minimiert. Die umgesetzte graphische Notation ist Datenflussorientiert und lehnt sich an Blockschaltdiagramme an. Ein Metrikdiagramm besteht demnach aus Knoten und Kanten, wobei die Knoten Metrikblöcke (siehe 6.4.2) und die Kanten Datenflüsse (siehe 6.4.3) repräsentieren. Auf den Einsatz von Kontrollflüsse wird verzichtet, um die Anzahl der verwendeten Beschreibungsmittel zu reduzieren und gleichzeitig Lesbarkeit und Verständlichkeit von Metrikdiagrammen zu fördern. Die Berechnungsreihenfolge ergibt sich somit allein implizit durch die Modellierungsstruktur, indem die Vorgänger- / Nachfolger-Beziehungen der Datenflussmodellierung ausgewertet werden. Eingangs- und Ausgangsports sollen, als kleine Dreiecke gezeichnet, am Rand ihrer zugeordneten Blöcke platziert werden, von dort aus werden sie mit den Datenflüssen verbunden.

Für die unterschiedlichen Arten von Metrikblöcken, die im Metrik-Metamodell spezifiziert werden, sollen unterschiedliche graphische Repräsentationen entworfen werden (Abschnitt 6.2), so dass sie innerhalb einer Metrik gut voneinander zu unterscheiden sind. In der linken oberen Ecke des Blocks soll der Name des Metrikblocks erscheinen. Hier sollen bei der Metrikentwicklung sprechende Namen verwendet werden, welche die Verständlichkeit der Metrik fördern. In der rechten oberen Ecke

soll der aktuelle Berechnungszustand des Blocks in einer Farbcodierung eingeblendet werden (Abschnitt 6.5.2.1). In der restlichen Fläche eines Metrikblocks sollen Block-spezifische Informationen, wie zum Beispiel Ergebnisse der Berechnung oder Eingabefelder dargestellt werden.

Neben der reinen graphischen Repräsentation ist die graphisch gestützte Modellierung ein wichtiges Merkmal für den Einsatz der graphischen Notation.

5.4 Integriertes Framework

Integriert steht an dieser Stelle für mehrere Merkmale. Zum Einen heißt es an dieser Stelle, dass Modelle von Bewertungsmetriken als Teil des zu bewertenden Datenmodells (E/E-Architekturmodell) integriert sind. Das Metrik-Metamodell ist als eigenständige Erweiterung konzipiert, so dass sich dieser Ansatz für andere formale MOF-kompatible Metamodelle (z.B. M^{2}TOS, [REI05]) eignet (Siehe Kapitel 2.4). Darüber hinaus bedeutet *integriert* auch, dass Modellierung von E/E-Architekturen und Ausführung von Metriken innerhalb einer Applikation stattfinden. Bei der Ausführung der Metriken steht voller Modellzugriff zur Verfügung. Damit können alle im Modell verfügbaren Daten für Bewertungen herangezogen[15] werden. Darüber hinaus besteht auch die Möglichkeit, während der Ausführung einer Metrik Modellmanipulationen durchzuführen, um so zum Beispiel Architekturoptimierungen vorzunehmen. *Integriert* umfasst ferner eine funktionale Integration in die umgebende Applikation. Dies soll erreicht werden, indem das Metrikframework durch eine API gekapselt ist, so dass Metriken durch Events oder Kontextmenüaufrufe ausgeführt werden. Darüber hinaus können Metrikergebnisse in Oberflächenkomponenten (wie zum Beispiel Tabellen) eingeblendet oder in generierte Report-Dokumentationen verwendet werden.

Die Hauptaufgabe des integrierten Metrik-Framework ist, modellierte Metriken auszuführen. Hierzu soll die Berechnungsreihenfolge durch die Analyse der Modellierungsstruktur der Metrik ermittelt werden. Sind einem Metrikblock ein oder mehrere andere Metrikblöcke durch Datenflüsse vorgeschaltet, so müssen diese zuerst berechnet werden. Sollte es Zyklen geben, so soll dies dem Anwender angezeigt werden. Die Berechnung der einzelnen Metrikblöcke soll an eine, durch eine öffentliche API gekapselte Implementierungsschicht delegiert werden. Für die im Metrikmetamodell beschriebenen Metrikblöcke soll das Framework die passenden Implementierungen vorhalten. Die Berechnung generischer Blöcke (Abschnitt 6.2.13) wird an die bereitstellenden Komponenten dele-

15 Es besteht prinzipieller Zugriff auf alle Daten im Modell. Dieser wird unter Umständen eingeschränkt durch ein Rechte- und Rollenmanagement, welches es ermöglicht, Schreib- und Sichtbarkeitsrechte an bestimmte Benutzergruppen bzw. Rollen zu vergeben.

giert.

Eine weitere Eigenschaft des integrierten Metrikframeworks ist die Debugfähigkeit von Metriken (Abschnitt 6.5.4). Dies ist für eine effiziente Entwicklung und Fehlerbeseitigung von Metriken wichtig. Es sollen zwei wesentliche Aspekte unterstützt werden: Modell- und Quellcodebasiertes Debugging. Beim Modellbasierten Debug werden Haltepunkte auf Metrikblöcken oder Ports (je nach Art des Haltepunktes, siehe Abschnitt 6.5.4.2) gesetzt, die dann den Ablauf einer Metrik an den definierten Stellen anhalten und detaillierte Informationen in begleitenden Ansichten darstellen. Der Quelltextbasierten Debug ermöglicht, analog zu modernen integrierten Entwicklungsumgebungen (IDE), das Debugging während der Laufzeit direkt auf dem Quelltext von Berechnungsblöcken (Abschnitt 6.5.4.3).

5.5 Ergebnisaufbereitung und Nachvollziehbarkeit von Ergebnissen

Ein wesentlicher Aspekt bei der Bewertung ist die Ergebnisaufbereitung und die Nachvollziehbarkeit, wie es zu den Ergebnissen gekommen ist.

Bei der Ergebnisaufbereitung gibt es, je nach Anwendungsfall, unterschiedliche Anforderungen, die sich widersprechen können. Einerseits sollen die Ergebnisse eine möglichst einfaches Format, zum Beispiel als Gütezahl, haben. Dies ist vor Allem im Zusammenhang mit automatisierten nachgelagerten Prozessen oder Vergleichen der Fall. Andererseits gibt es Bedarf an möglichst detaillierte Ergebnisse, meist in tabellarischer Form, bei denen nicht nur die Ergebnisse oder Teilergebnisse aufbereitet sind, sondern wo sie ebenfalls mit einem oder mehreren Modellierungsartefakten verknüpft sind. Dies ist vor Allem dann der Fall, wenn Analysen oder detaillierte Vergleiche durchgeführt werden. Ein Beispiel hierfür ist eine Leitungssatzanalyse, bei der die errechneten Leitungslängen direkt mit den Leitungen verknüpft sind.

Um diesen Anforderungen gerecht zu werden, sollen mehrere Mechanismen genutzt werden. Zur Visualisierung von einfach strukturierten Ergebnissen wie Skalare oder Vektoren eignen sich Ampeln oder Skalenanzeigen. Für weiterführende Verarbeitungen in Form von tabellarischen Vergleichen oder Reportdokumentationen sollen die Ergebnisse weitergegeben werden können. Komplexe und detaillierte Ergebnisse sollen in Form von Tabellen, Bäumen, Reportdokumentationen oder spezifische Exporte zur weiteren Verarbeitung ausgegeben werden können. Bei einer Ausgabe innerhalb der Applikation ist die Navigierbarkeit zu den Modellierungsartefakten, die mit den Ergebnissen verknüpft sind besonders wichtig.

Bei der Nachvollziehbarkeit von Metrikergebnissen ist nicht nur das Endergebnis an sich wichtig, sondern auch die Teilergebnisse, die zu diesem Ergebnis führen. Dazu soll es die Möglichkeit geben, nach einer Berechnung eines Metrikblocks seine Ausgaben auf den Ausgangsports darzustellen. Ferner soll es möglich sein, bei Bedarf detailliertere Informationen zu erhalten, indem Teilergebnisse an weitere, zusätzliche Blöcke weiterzugeben.

Ein wichtiger Aspekt bei der Nachvollziehbarkeit von Metrikergebnissen ist die Abdeckung über die Historie eines Architekturmodells. Dies bezieht sich auf das Architekturmodell eines bestimmten Standes selbst, als auch auf die Metrik, welche zu der Berechnung der Ergebnisse verwendet wurde. Denn nachträgliche Änderungen an Architekturmodell oder Berechnungsmetrik können zu abweichenden Ergebnissen führen. Dazu soll es möglich sein, freigegebene Modell- und Metrikstände zu revisionieren und so festzuschreiben. Änderungen sind dann nur noch möglich, indem neue Revisionen gebildet werden, die dann zu einer automatischen Unterscheidbarkeit führen.

5.6 Sensitivität und Was-Wäre-Wenn-Szenarien

Bei der Analyse von Bewertungsergebnissen kommt es immer wieder zu Fragestellungen, inwiefern sich Änderungen an den Eingangsgrößen auf die Ergebnisse auswirken. Darüber lassen sich Aussagen über Robustheit und Stabilität treffen. Ebenso von Interesse ist, dass die Ergebnisse einer Sensitivitätsanalyse transparent sind und sich nahtlos mit den in Abschnitt 5.5 beschriebenen Anforderungen vertragen.

Es soll die Möglichkeit geben, für skalare Werte (Sensitivitätsparameter) Anfangs- und Endwerte vorzugeben, die in einer einstellbaren Schrittweite durchlaufen werden und den skalaren Wert mit dem jeweiligen Wert überschreiben. Ab diesem Punkt in der Berechnungsreihenfolge der Metrik sollen die nachfolgenden Blöcke mit dem jeweils aktuellen Wert einmal berechnet werden. Die Ergebnisse der Blöcke sollen zwischengespeichert werden. Gibt es mehrere Sensitivitätsparameter, für die eine Sensitivitätsanalyse innerhalb einer Metrik durchgeführt werden soll, so werden die nachfolgenden Metrikblöcke für jede mögliche Kombination der Sensitivitätswerte einmal berechnet werden.

Hierbei ist zu beachten, dass die Anzahl der zu berechnenden Blöcke mit dem Produkt der Schrittanzahlen der Sensitivitätsparameter steigt. In diesem Umfang nehmen auch Berechnungszeit und Speicherbedarf zu.

Nachdem die Berechnung der Metrik abgeschlossen ist, soll der Benutzer die Möglichkeit haben,

die einzelnen Wertkombinationen nachzustellen und die Ergebnisse zu analysieren. Des Weiteren soll eine tabellarische Ausgabe aller berechneten Kombinationen für weiterführende Analysen vorhanden sein.

Eine weitere, nicht ganz so rechenintensive Art zur Durchführung von Was-Wäre-Wenn-Szenarien ist die einfache Überschreibung von Ausgangsports auf Berechnungsblöcken. In diesem Fall werden die Ergebnisse der Ports zwar berechnet, auf die Ausgangsports wird aber der vorgegebene Wert gelegt und zur weiteren Berechnung den nachfolgenden Metrikblöcken übergeben.

5.7 Bewertung und Vergleich von Architekturvarianten und -Realisierungsalternativen

Wichtig bei der Bewertung von Elektrik/Elektronik-Architekturen ist die integrierte Unterstützung von Architekturvarianten und Realisierungsalternativen (Abschnitt 3.3.8), um diese schnell und effizient miteinander vergleichen zu können.

Bei der Umsetzung gibt es hier Abhängigkeiten zur umgebenden Applikation (PREEvision), bei der es möglich ist, eine Architekturvariante oder Realisierungsalternative als aktiv zu kennzeichnen. In diesem Modus greifen spezielle Filter, so dass die Ermittlung, ob ein Modellierungsartefakt zur aktiven Variante gehört oder nicht sehr schnell und einfach ist.

Das Framework für Bewertungsmetriken soll sich dieses Detail zunutze machen, indem es einen Metrikblock gibt, der in der Lage ist, eine Menge von Architekturvariante und Realalternativen als Eingabe derart nacheinander durchzuschalten, so dass bei jeder erneuten Berechnung des Blocks die nächste Variante als aktiv gekennzeichnet wird. Da in einer Metrik jeder Block nur einmal ausgeführt wird, soll es einen Schleifenblock geben, in welchen der Block zur Variantenumschaltung eingebettet wird, so dass für jede zu untersuchende Variante die Bewertungsmetriken einmal ausgeführt werden.

5.8 Vergleich mit dem aktuellen Stand der Technik

In diesem Abschnitt werden die in Kapitel 4.4 vorgestellten Projekte, die den aktuellen Stand der Technik widerspiegeln, mit den Inhalten dieser Arbeit verglichen. Als Vergleichskriterien kommen die in den vorangegangenen Abschnitten 5.2 bis 5.7 diskutierten Merkmale zum Einsatz.

	ArchiVal	VEIA	EEKT	SPEEDS	MARTE	ATESST2	MAENAD
Modellbasierter Ansatz zur Metrikbeschreibung	☐	☐	☐	☑	☑	☑	☑
Graphische Notation für Metriken	☐	☐	☐	☐	☐	☐	☐
Integriertes Framework	☐	☐	☑	☐	☐	☐	☐
Ergebnisaufbereitung und Nachvollziehbarkeit	☑	☑	☑	☐	☑	☑	☑
Sensitivität und Was-Wäre-Wenn-Szenarien	☐	☐	☐	☐	☐	☐	☐
Automatisierte Variantenbetrachtung	☐	☑	☑	☐	☐	☐	☐

Tabelle 3: Vergleich mit dem aktuellen Stand der Technik

Bis auf SPEEDS, MARTE, ATESST2 und MAENAD, die teilweise eine modellbasierte Metrikbeschreibung erlauben, werden die Metriken in den anderen Projekten nicht modellbasiert beschrieben. Keines der Projekte spezifiziert oder beinhaltet eine graphische Notation zur Beschreibung von Metriken. Bis auf das EEKT-Projekt unterstützt auch kein anderes Projekt ein integriertes Framework zur Verwaltung, Ausführung und integrierte Ergebnisdarstellung von Metriken. Die Ergebnisaufbereitung und Nachvollziehbarkeit von Ergebnissen ist im ArchiVal-Projekt durch den eingesetzte, tabellenorientierte Darstellung eines QADAG-Modells gut möglich. In den anderen Projekten (bis auf VEIA) ist dies mit Einschränkungen möglich. Sensitivitätsanalysen und Was-Wäre-Wenn-Szenarien werden von keinem der betrachteten Ansätze unterstützt. Eine automatisierte Variantenbetrachtung im Zusammenhang mit der Metrikausführung ist in VEIA und EEKT mit Abstrichen möglich.

Zusammenfassend erfüllt, wie in Tabelle 3 zusammengefasst, keines der untersuchten Projekten die Anforderungen zur effizienten, nachvollziehbaren Bewertung von E/E-Architekturen mit der Unterstützung für Sensitivitätsanalysen und automatisierte Variantenbetrachtungen. Zugleich schreiben die Projekte fest definierte Metriken vor oder delegieren die Bewertung an spezialisierte, aber domänenfremde Werkzeuge.

5.9 Systematische Zusammenfassung der Anforderungen

Die in diesem Kapitel beschriebenen Anforderungen lassen sich folgendermaßen systematisch zusammenfassen:

Modellbasierte Beschreibung von Metriken

- Entwurf eines MOF-basierten Metamodells zur Beschreibung von Bewertungsmetriken als do-

mänenspezifische Sprache (DSL)

- Definition spezifischer Metrikblöcke mit spezifischen Eigenschaften im Metrik-Metamodell
- Abbildung eines expliziten Portkonzepts und Datenflüsse zum transparenten Aufbau von Metriken und der Nachvollziehbarkeit von Datenströmen
- Integration des Metrik-Metamodells in den Kontext der EEA-ADL zur Bewertung von Elektrik/Elektronik-Architekturen im Fahrzeug
- Integration des Metrik-Metamodells in den Kontext des RM3-Metamodells zum Nachweis der Unabhängigkeit zur EEA-ADL

Graphische Notation von Metriken

- Entwurf einer übersichtlichen, intuitiv verständlichen graphischen Repräsentation der Metrikblöcke, Ports und Datenflüsse
- Darstellung der aktuellen Berechnungszustände von Metrikblöcken
- Darstellung von Metriken in mehreren Diagrammen mit unterschiedlichen Detaillierungsgraden für spezifische Anwendungsfälle
- Komfortable und effiziente graphisch gestützte Editier- und Modellierungsfunktionen für Metriken

Integriertes Metrik-Framework

- Ausführung von Metriken innerhalb von domänenabhängigen Applikationen inklusive Ermittlung der Berechnungsreihenfolge und Zyklenerkennung
- Voller Datenzugriff auf alle Artefakte im Modell
- Weitreichende Integration in die umgebende Applikation
- Modell- und Quelltextbasiertes Debugging von Metriken

Ergebnisaufbereitung und Nachvollziehbarkeit von Metrikergebnissen

- Übersichtliche Darstellung von Ergebnissen direkt im Metrikdiagramm in Form von Ampeln, Skalen oder anderen Darstellungsmöglichkeiten
- Detaillierte Darstellung von Teil-, Zwischen- und Endergebnissen auf Ausgangsports von Metrikblöcken in begleitenden, kontextsensitiven Ansichten
- Export von Ergebnissen zur Analyse mit externen Applikationen
- Nachvollziehbarkeit von Ergebnissen auf festgeschriebenen Modell- und Metrikständen, auch auf historischen Datenständen

Sensitivität und Was-Wäre-Wenn-Szenarien

- Möglichkeit zur Variation von Parameter- und Berechnungswerten
- Möglichkeit zur interaktiven Analyse von Sensitivitätsberechnungen
- Tabellarische Ausgabe aller berechneten Kombinationen für weiterführende Analysen
- Überschreibung von einzelnen Berechnungswerten, die anstelle der kalkulierten Werte an nachfolgende Blöcke übergeben werden

Vergleich von Architekturvarianten und Realisierungsalternativen

- Ausführung von Bewertungsmetriken auf mehrere vorgegebene Architekturvarianten und Realisierungsalternativen
- Transparente zusammengefasste Ergebnisse für die einzelnen Berechnungsdurchläufe, verfügbar für nachfolgende Metrikblöcke oder zur manuellen Auswertung

6 Integriertes, modellbasiertes, graphisch notiertes Bewertungs-, Analyse- und Vergleichsframework

6.1 Kernaufgaben

Ausgehend von den in Kapitel 5 beschriebenen Anforderungen an ein Bewertungs-, Analyse- und Vergleichsframework für E/E-Architekturen werden hier zunächst die daraus abgeleiteten Kernaufgaben sowie die Konzeption dieses Frameworks beschrieben. In den folgenden Abschnitten dieses Kapitels wird auf die technische Realisierung dieser Aufgaben eingegangen.

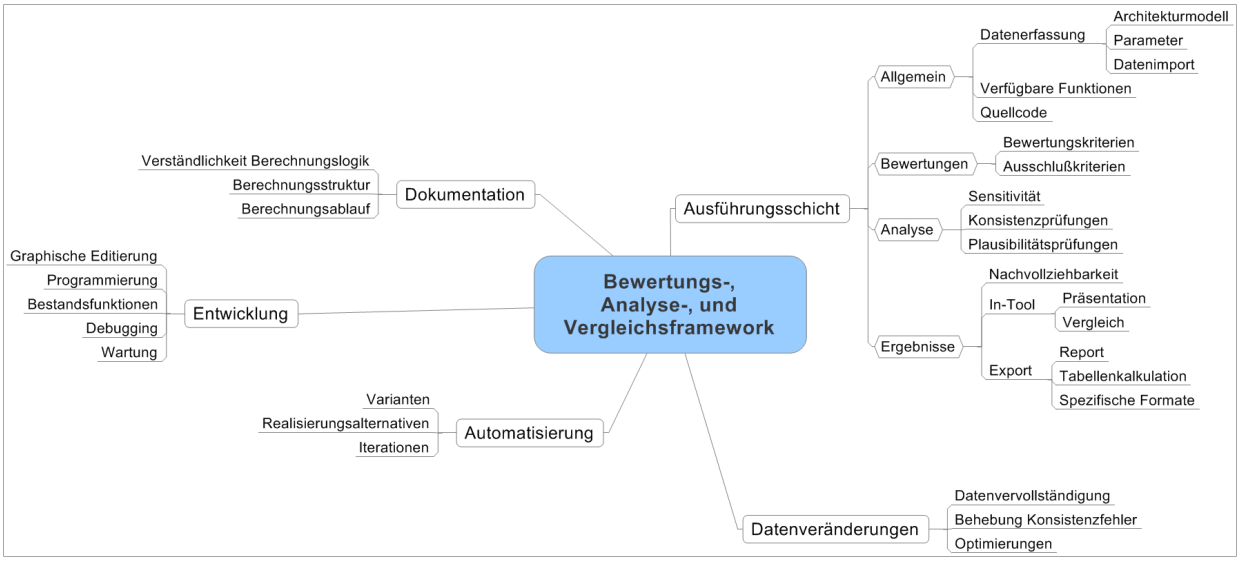


Abbildung 6.1: Aufgaben Bewertungs-, Analyse- und Vergleichsframework

Die Kernaufgaben werden in 6.1 dargestellt, sie lassen sich in fünf Kategorien einteilen:

- Ausführungsschicht
  - Allgemein
    Die Ausführungsschicht beinhaltet zunächst allgemeine Aufgaben, um vorhandene Berechnungsvorschriften auszuführen. Die Datenerfassung dient der Beschaffung relevanter Informationen, die für die spätere Berechnung benötigt werden. Platzhalter, die einen dynamischen Kontext für die Berechnung darstellen, der erst zum Zeitpunkt der Berechnung ermittelt wird, gehören ebenfalls zur Datenerfassung.
    Die Ausführungsschicht selbst soll eine Reihe von Funktionen anbieten und die Möglichkeit geben, vorhandenen Quellcode oder Bibliotheken einzubinden und auszuführen.
  - Bewertungen
    Berechnung von Bewertungs- (z.B. Gewicht Leitungssatz) und Ausschlusskriterien (z.B.

Energieversorgung einer Komponente nicht vorhanden).

- Analyse
  Durchführung von Plausibilitätsprüfungen (z.B. ist das berechnete Gewicht plausibel?), Konsistenzprüfungen (z.B. Bedatung von Gewichtsattributen vollständig?), Sensitivitätsanalysen (siehe Abschnitt 6.4.12).

- Ergebnisse
  Zu den wesentlichen Ergebnissen gehört die Sicherstellung der Nachvollziehbarkeit der Ergebnisse. Das heißt, dass die Ergebnisse angefangen von der Datenerfassung bis hin zur Berechnung von Zwischenergebnissen und des Endergebnisses transparent und Schritt für Schritt nachvollziehbar sein sollen.
  Des weiteren werden Ergebnisse sowohl In-Tool als auch optional als Export ausgegeben.
  Zur Aufbereitung der In-Tool-Ergebnisse gehört die Ausgabe in geeignete Formate (z.B. Listen, Tabellen, Charts, Ampeln). Sofern Modellierungsartefakte teil des Ergebnisses sind, so werden diese navigierbar dargestellt (Teil der Nachvollziehbarkeit). Hier ist auch der Vergleich von Bewertungsergebnissen.
  Zu den Exportmöglichkeiten gehört die Ausgabe in Report-Dokumenten, Tabellenkalkulationen oder andere, spezifische Formate.

- Datenveränderungen
  Änderungen von Daten beziehen sich hier auf Änderung von Inhalten des Architekturmodells. Dies geschieht während des Ablaufs von Berechnungsvorschriften und gehört somit zur Ausführungsschicht. Da es sich hier um schreibenden Zugriff auf Architekturdaten handelt (während Bewertungen, Analysen und Vergleiche lesenden Zugriff haben), erfolgt eine Einordnung ein eine eigene Kategorie.
  Diese Kategorie lässt sich in drei weitere Bereiche einteilen. Die Datenvervollständigung repräsentiert Berechnungsvorschriften, welche die manuelle Eingabe oder Bearbeitung von Architekurmodellen vereinfachen, indem sie Attribute oder Modellstrukturen anlegen, die dann weiter bearbeitet werden. Auf diese Weise lassen sich anwendungsspezifische Lösungen nachrüsten. Einen ähnlichen Charakter hat die automatisierte Beseitigung von Konsistenzfehlern.
  Ein weiterer Bereich stellen automatisierte Optimierungen dar. Diese werden im Kontext dieser Arbeit nicht weiter betrachtet.

- Automatisierung
  Zweck der Automatisierung ist, Berechnungsvorschriften automatisiert durchzuführen. Dies kann zum Beispiel die iterative Bewertung mehrerer Varianten oder Realisierungsalternativen sein, die als Kontext der Berechnungsvorschrift übergeben werden und nach der Berechnungsdurchführung miteinander verglichen werden.
  Eine andere Art der Automatisierung ist die ereignisgesteuerte Aktivierung von Berechnungsvorschriften, deren Berechnungskontext zum Aktivierungszeitpunkt ermittelt wird.

- Entwicklung
  In dieser Kategorie sind Aufgaben aufgelistet, die mit der Entwicklung der Berechnungsvorschriften selbst zusammenhängen. Wichtiger Punkt hierbei sind die graphische Repräsentation und Editierfunktionen. Sowohl vorhandene Bestandsfunktionen (z.B. eine Funktion zur Sortierung und Filterung von Daten) als auch programmierte (oder über eine Bibliothek eingebundene) Funktionen werden durch eigene Symbole graphisch repräsentiert.
  Bei der Fehleranalyse und -beseitigung sind Debug-Funktionalitäten von entscheidender Bedeutung. Das Debugging für das Bewertungs-, Analyse-, und Vergleichsframework bezieht sich sowohl auf Debugfähigkeiten auf struktureller Ebene, indem die Berechnung vor oder nach einzelner Schritte zu Analysezwecken angehalten oder weitergeschaltet werden kann, als auch das klassische Debugging von Quellcode (vgl. Abschnitt 6.5.4).
  Ein weiterer wichtiger Punkt ist die Wartung bestehender Berechnungsvorschriften. Hier kommt es vor allem darauf an, dass sich Entwickler schnell und effizient in die Logik der Berechnungsvorschrift einarbeiten und die Wartungsarbeiten durchführen können.

- Dokumentation
  Zur Dokumentation der Berechnungsvorschriften ist die graphische diagrammbasierte Darstellung wesentlich. Diese wird ergänzt durch textuelle Dokumentationsmöglichkeiten der einzelnen Berechnungsschritte. Damit ist die Verständlichkeit der Berechnungslogik, der Ablauf und die Struktur der Berechnungsvorschrift gegeben.

6.2 Kernartefakte

Kernartefakte für Architekturbewertung und -analysen sind Blöcke in denen spezifizierte Aufgaben durchgeführt werden und Kanten, die Datenflüsse repräsentieren und Daten von den Quell- zu den Zielblöcken transportieren. Blöcke und Datenflüsse werden über ein Port-Konzept (siehe Abschnitt 2.5.5) miteinander verknüpft.

Def. 40: Metrikblock

> *Ein Metrikblock kapselt einen Ausführungsschritt einer Metrik. Ein Metrikblock verfügt über Metrikblockports (Def. 41), um benötigte Daten zu empfangen und seine Ergebnisse weiterzugeben.*

Def. 41: Metrikblockport (Port)

> *Ein Metrikblockport stellt die Verbindung zwischen einem Metrikblock (Def. 40) und einem Metrikdatenfluss (Def. 42) her. Man unterscheidet zwischen Eingangports, über welche benötigte Daten dem Metrikblock zur Verfügung gestellt werden und Ausgangports, über welche*

der Metrikblock seine Ergebnisse an folgende Metrikblöcke weitergibt.

Def. 42: Metrikdatenfluss (Datenfluss)

Ein Metrikdatenfluss verbindet einen ausgehenden Metrikblockport (Def. 41) mit vielen eingehenden Metrikblockports. Der Metrikdatenfluss übermittelt das Ergebnis des Metrikblocks (Def. 40) am ausgehenden Metrikblockport zum Zeitpunkt der Ergebnisbereitstellung an die eingehenden Metrikblockports.

Durch die sinnvolle Verschaltung von Blöcken mit Hilfe der Datenflüsse ergeben sich zusammenhängende Strukturen, die eine Logik zur Bewertung und Analyse von E/E-Architekturmodellen abbilden, im folgenden auch vereinfachend Metrik genannt. Datenflüsse transportieren alle Arten von Daten von einem vorausgehenden Block zu einem oder mehreren nachfolgenden Blöcken. Am Anfang der Metrik stehen Blöcke, die Daten zur Verfügung stellen. Dies können beispielsweise Modellabfragen sein, die bestimme Modellartefakte im Datenmodell aufspüren und zur Verfügung stellen. Verarbeitende Blöcke nehmen Eingangsdaten entgegen und berechnen daraus Ausgangsdaten, die wiederum als Eingangsdaten für folgende Blöcke dienen. Abschließend gibt es Ergebnisblöcke, die nur Daten einlesen und diese als visuelles Ergebnis, z.B. als Ampel aufbereiten.

6.2.1 Allgemeine Darstellungen

Blöcke werden in der Regel innerhalb einer Metrik, wie in Abbildung 6.2 gezeigt, als Viereck dargestellt. An den Außenkanten können in Abhängigkeit vom entsprechenden Blocktyp Eingangs- und/oder Ausgangsports in der Form von eingehenden bzw. ausgehenden Pfeilspitzen angebracht sein.

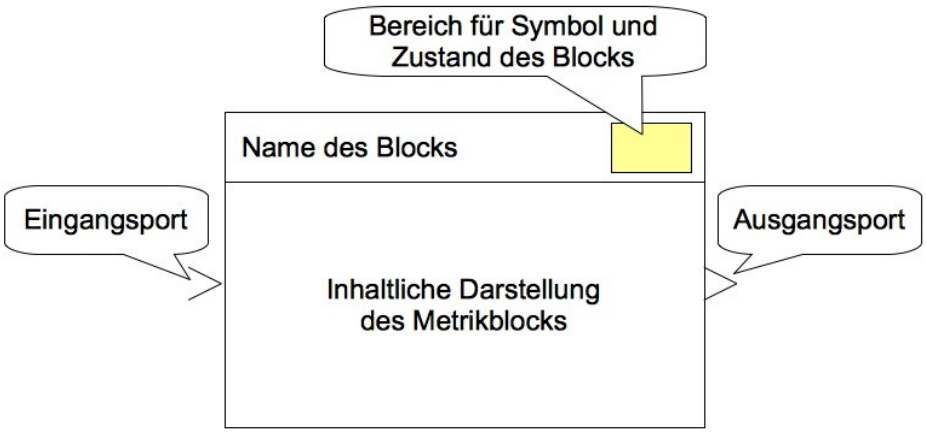


Abbildung 6.2: Schematische Darstellung eines Metrikblocks

Oben links ist der Name des Blocks eingeblendet. Auf der oberen rechten Seite werden ein Symbol zur besseren Identifikation des Blocktyps sowie der aktuelle Berechnungszustand des Blocks gezeigt. Die jeweiligen Symbole der einzelnen Metrikblöcke werden in den Abschnitten 6.2.2 bis 6.2.14 gezeigt. Die Berechnungszustände werden durch einen in Abschnitt 6.5.2.1 behandelten Zustandsautomaten definiert. Dort werden auch für jeden Zustand die im Block anzuzeigende Farbe festgelegt. 6.5.2.1

Metrikblöcke werden, wie in Abbildung 6.3 dargestellt, über Datenflüsse miteinander verbunden. Sie werden im Metrikdiagramm als schwarze Linie dargestellt. Ein Datenfluss ist immer an genau

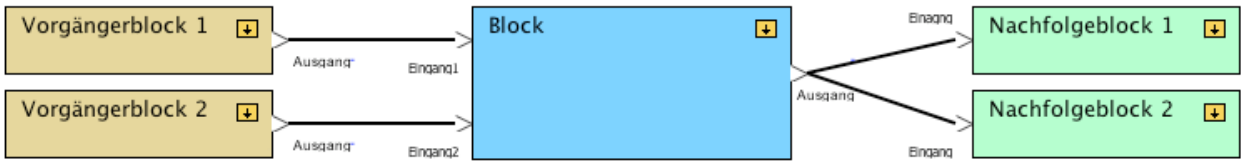


Abbildung 6.3: Datenflüsse zur Verbindung von Blöcken

einen Quellport (von dem weitere Datenflüsse ausgehen können) angeschlossen und kann mit vielen Zielports verbunden werden. Jeder Block kann prinzipiell über beliebig viele Ports verfügen, die Anzahl der unterstützten Port ist abhängig von der Blockimplementierung. Über die Datenflüsse ergeben sich Abhängigkeiten zwischen den Blöcken. Ein Block ist von seinen Vorgängerblöcken abhängig. Nachfolgende Blöcke sind von dem betrachteten Block abhängig.

Generell gilt, dass ein Datenfluss alle Arten von Daten transportiert, ohne diese in irgendeiner Form zu analysieren, zu interpretieren oder zu manipulieren.

6.2.2 Metrikkontextblock

Def. 43: Metrikkontextblock

> *Ein Metrikkontextblock referenziert auf beliebige Artefakte des Modells und übergibt diese über einen ausgehenden Metrikblockport (Def. 41) an nachfolgende Metrikblöcke (Def. 40). Metrikkontextblöcke können zur Laufzeit dynamisch, mit einem erst zur Ausführungszeit bekannten Artefakt überschrieben werden. In diesem Fall werden die referenzierten Artefakte des Modells ignoriert.*

Der Metrikkontextblock dient dazu, selektiv Modellartefakte in eine Metrik einzubringen, die von nachfolgenden Blöcken als Eingabe verwendet werden sollen. Diese Artefakte werden vom Benut-

zer ausgewählt und in den Metrik-Kontextblock integriert. Ein Anwendungsbeispiel ist eine Buslastmetrik, die für ausgewählte Bussysteme die jeweilige Last berechnen soll.

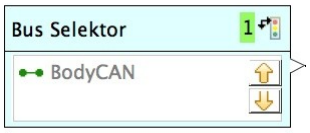


Abbildung 6.4: Metrikkontextblock

In Abbildung 6.4 ist ein Metrikkontextblock zu sehen. Ihm zugeordnet ist das Bussystem *BodyCAN*. Dieses wird über den Ausgangsport auf der rechten Seite des Blocks ausgegeben.

Metrikkontextblöcke können zur Laufzeit dynamisch mit anderen Artefakten überschrieben werden. In diesem Fall werden die referenzierten Artefakte ignoriert. Diese Funktion wird vom Metrikausführer, der in Abschnitt 6.2.14 beschrieben ist, verwendet.

6.2.3 Modellabfrageblock

Def. 47: Modellabfrageblock

> *Ein Modellabfrageblock stellt die Funktionalität einer Modellabfrage (Def. 46) in einer E/E-Architekturmetrik (Def. 24) zur Verfügung.*

Die Modellabfrage dient dazu, zur Durchführung der Metrik benötigte Artefakte im Datenmodell zu finden und zur Verfügung zu stellen. Die Spezifikation des Suchmusters erfolgt über den LHS-Teil einer M^2TOS-Regel (siehe Kapitel 2.4) oder als programmierter Quellcode.

Der Suchbereich von LHS-basierten Modellabfragen kann durch Angabe zusätzlicher Artefakte eingeschränkt werden. Diese werden über eine Eingangsport an die Modellabfrage übergeben und als Anker-Artefakt der LHS-Regel interpretiert.

Das Ankersymbol des Objektes CANBus:BusSystem sagt aus, dass dieses Artefakt der Regel als Referenzartefakt vorgegeben werden muss. Der Modellabfrageblock, der diese Regel in einem Metrikdiagramm repräsentiert, besitzt einen zusätzlichen Eingangsport, der das vorgegebene Bussystem-Artefakt für das Ankerobjekt erwartet.

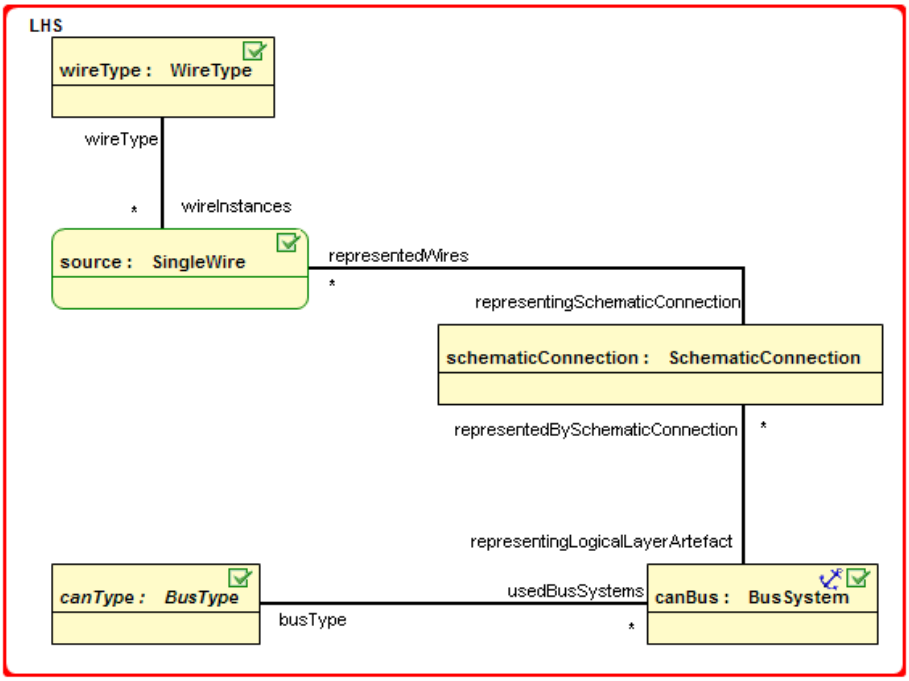


Abbildung 6.5: Beispiel einer Suchregel

Abbildung 6.5 zeigt die LHS einer M²TOS-Regel. In diesem Beispiel wird nach allen Leitungen gesucht, die an der Realisierung eines CAN-Busses im Leitungssatz beteiligt sind. Alle mit einem Häkchen gekennzeichneten Objekte sind Teil des Ergebnisses, welches die Modellabfrage zurück gibt.

Das Ergebnis einer Modellabfrage ist als Tabelle darstellbar, wobei für jedes ergebnisrelevante Objekt eine Spalte mit dem Objektnamen angelegt wird. Die Spaltenreihenfolge ergibt sich aus aus dem Attribut *returnOrder* vom Typ *Integer*, das auf den Objekten der Regel gesetzt wird.

Standardmäßig wird das Ergebnis der Modellabfrageblocks als Liste, die alle gefundenen Artefakte beinhaltet, über einen Ausgangsport zurück gegeben. Durch Setzen des Portnamens können andere Ausgabeformate eingestellt werden. Mit dem Wert <*Table*> (Dies ist der Standardwert und wird daher nicht im Diagramm visualisiert) wird das Ergebnis über diesen Ausgangsport als Tabelle ausgegeben. Wird für den Portnamen ein Spaltennamen der Tabelle gesetzt, so werden alle Artefakte dieser Spalte als Liste ausgegeben.

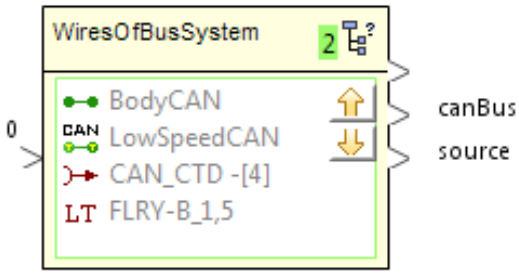


Abbildung 6.6: Modellabfrageblock

In Abbildung 6.6 ist ein Modellabfrageblock zu sehen. Die diesem Block zugeordnete Suchregel ist in Abbildung 6.5 dargestellt. Als Ankerartefakt wurde das Bussystem BodyCAN aus Abbildung 6.4 verwendet.

6.2.4 Joinblock

Def. 28: Joinblock

> *Ein Joinblock ist ein Metrikblock (Def. 40), der zwei Ausgangstabellen (rechte und linke Tabelle) miteinander bezüglich eines gemeinsamen Spaltennamens als Verknüpfungskriterium zu einer Ergebnistabelle verknüpft.*

Mit dem Joinblock (*to join* – engl. verbinden) können zwei Tabellen von vorhergehenden Metrikblöcken miteinander zu einer Ergebnistabelle verschmolzen werden. Dies ist vor allem dann sinnvoll, wenn mehrere Modellabfragen Teile einer komplexen Fragestellung abbilden. Die Ergebnistabellen können über Joinblöcke zusammengeführt werden. Über Kaskaden von Joinblöcken und anderen Metrikblöcken werden viele Tabellen miteinander Tabelle verschmolzen. Zwischenresultate können für andere Berechnungen herangezogen werden.

Der Joinblock verfügt, wie in Abbildung 6.7 dargestellt, über zwei Eingänge, an denen die beiden

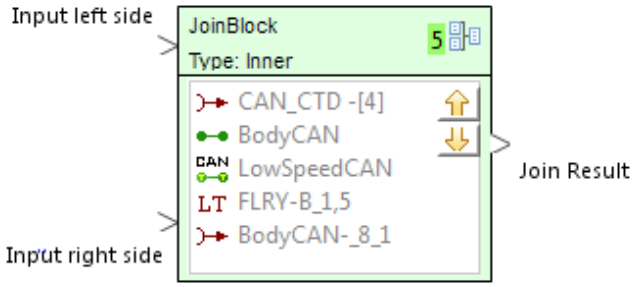


Abbildung 6.7: Joinblock

Eingangstabellen erwartet werden. Die beiden Eingänge werden mit *Input left side* und *Input right*

side bezeichnet. Die Anzahl der Spalten und Zeilen beider Tabellen ist für die Verschmelzung unerheblich, jedoch muss jeweils eine Spalte als Joinkriterium ausgewählt werden. Bei der Verschmelzung der Tabellen werden die Zelleninhalte, im Folgenden linkes und rechtes Joinobjekt genannt, der beiden ausgewählten Spalten zeilenweise herangezogen. Je nach Ergebnis und eingestellter Verschmelzungsart wird die rechte, die linke oder die Kombination aus beiden Zeilen in die Ergebnistabelle übernommen. Der Joinblock unterstützt insgesamt vier unterschiedliche Verschmelzungsarten, die im Folgenden näher erklärt werden.

- *Full Join*: Sind linkes und rechtes Joinobjekt unterschiedlich, werden beide Zeilen in die Ergebnistabelle übernommen. Sind die Joinobjekte identisch, werden linke und rechte Zeile zu einer Zeile verschmolzen und in die Ergebnistabelle übernommen.
- *Inner Join*: Linke und rechte Zeile werden bei Gleichheit des linken und des rechten Joinobjekts verschmolzen und in die Ergebnistabelle übernommen. Andernfalls wird der Ergebnistabelle nichts hinzugefügt.
- *Right Join*: Die rechte Zeile wird auf jeden Fall der Ergebnistabelle hinzugefügt. Weisen linkes und rechtes Joinobjekt Gleichheit auf, so wird die rechte Zeile mit der linken verschmolzen, bevor sie in die Ergebnistabelle geschrieben wird.
- *Left Join*: Verhält sich analog zum *Right Join*, mit dem Unterschied, dass hier die Rollen der beiden Eingangszeilen vertauscht sind.

Die Anzahl der Spalten n_{result} der Ergebnistabelle berechnet sich aus $n_{result}=n_l+n_r-1$, mit n_l für die Anzahl der Spalten der linken Eingangstabelle und n_r für die Anzahl der Spalten der rechten Eingangstabelle. Die beiden Spalten für die Joinkriterien werden zu einer Spalte zusammengefasst, welche die erste Spalte der Ergebnistabelle bildet. Die Reihenfolge der anderen Spalten ergibt sich standardmäßig aus der Reihenfolge der Spalten der linken Eingangstabelle gefolgt von den Spalten der rechten Eingangstabelle. Der Metrikentwickler hat jedoch die Möglichkeit, sowohl die Ordnung der Spalten als auch der Zeilen über Konfigurationsparameter zu ändern. Zusätzlich wird die Ergebnistabelle im inneren Bereich des Joinblocks als Liste reduziert dargestellt.

6.2.5 Berechnungsblock

Def. 9: Berechnungsblock

Ein Berechnungsblock ist ein Metrikblock (Def. 40), der vorgegebenen Java-Quellcode aus-

führen kann. Benötigte Daten werden über eingehende Metrikblockports (Def. 41) erwartet, Berechnungsergebnisse werden an ausgehende Metrikblockports übergeben.

Der Berechnungsblock ist in der Lage, Quellcode auszuführen. Zur Zeit wird die Programmiersprache Java unterstützt. Der Berechnungsblock kann über beliebig viele Eingangs- und Ausgangsports verfügen, die über den Namen des Ports identifiziert und vom Quellcode aus angesprochen werden. Über einen zentralen Schalter können beim Berechnungsblock die berechneten Ergebnisse für die ausgehenden Ports durch benutzerdefinierte Ergebnisse überschrieben werden. Auf diese Weise können „Was wäre wenn“-Szenarien durchgeführt werden, ohne an der Metrikstruktur etwas zu ändern.

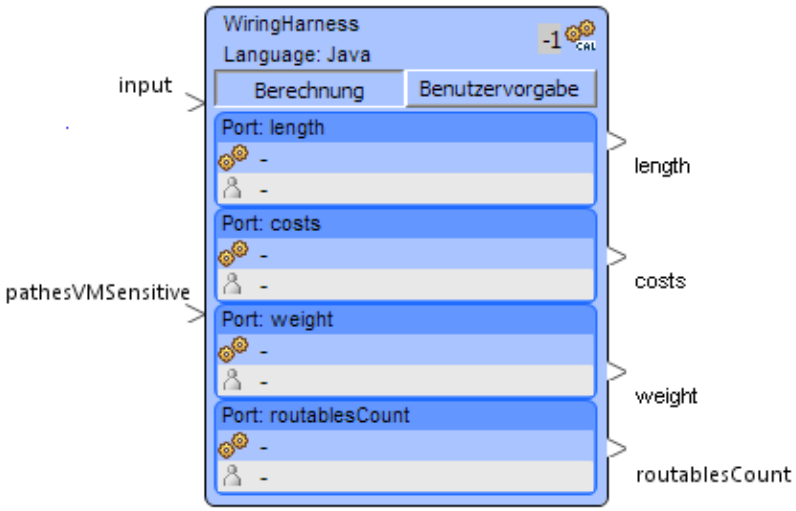


Abbildung 6.8: Berechnungsblock

In Abbildung 6.8 ist ein Berechnungsblock mit vier Ausgängen zu sehen. Für jeden Ausgang ist im Innern des Blocks ein Ergebnisbereich eingeblendet, in dem sowohl das berechnete als auch das benutzerdefinierte Ergebnis eingeblendet sind. Eingehende und ausgehende Ports sind an den Außenkanten des Blocks frei verschiebbar, um den Block innerhalb eines Metrikdiagramms mit möglichst wenigen graphisch überschneidenden Datenflüssen platzieren zu können. Die Zuordnung zwischen ausgehendem Port an einer Außenkante und seiner Repräsentation im Innern des Berechnungsblocks erfolgt daher über eine farbliche Hervorhebung des jeweiligen selektierten Ports.

Der Berechnungsblock verfügt über ein Operation-Flag. Ist dieses aktiviert, so wird der Quellcode als Undo- / Redo-fähige Modelloperation ausgeführt. In diesem Modus dürfen im Berechnungsblock Modelländerungen durchführen.

6.2.6 Sortier- und Filterblock

Def. 64: Sortier- und Filterblock

Ein Sortier- und Filterblock ist ein Metrikblock (Def. 40), der Eingangsdaten bezüglich einer vorgegebenen Konfiguration sortiert und/oder filtert und das Ergebnis weitergibt.

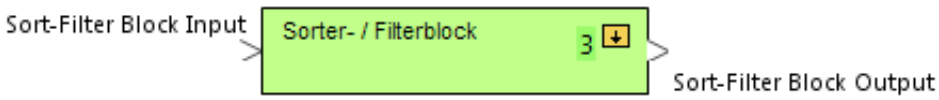


Abbildung 6.9: Sortier- und Filterblock

Der Sortier- und Filterblock ermöglicht, Sortier- und Filteroperationen auf Daten auszuführen, die auf dem eingehenden Port ankommen. Da der eingehende Datenfluss auch andere Blöcke versorgen kann, werden die Sortier- und Filteroperationen auf einer Kopie der Eingangsdaten durchgeführt und das Ergebnis auf den Ausgangsport gelegt.Als Sortierkriterien steht die Ordnung der Spaltenreihenfolge und die Sortierung der beinhalteten Elemente nach einstellbaren Attributwerten zur Verfügung. Als Filterkriterium werden Duplikate innerhalb einer Spalte herausgefiltert.

6.2.7 Ampelblock

Def. 4: Ampelblock

Ein Ampelblock ist ein Metrikblock (Def. 40), der eingehende Daten mit vorgegebenen Schwellwerten vergleicht und das Ergebnis als Ampel visualisiert.

Der Ampelblock ist ein Ergebnisblock und visualisiert eingehende Daten als Ampel. Dazu verfügt er über drei Eingangs- und keinen Ausgangsport. Der erste Eingangsport liefert den Wert für den gelb/rot-Übergang $\vec{k}_{gelb/rot}$ und heißt *Upper bound*. Der zweite Eingangsport liefert den Wert für den grün/gelb-Übergang $\vec{k}_{grün/gelb}$ und heißt *Lower bound*. Mit diesen beiden Werten ist der Ampelblock vollständig konfiguriert. Im Normalfall ist $\vec{k}_{gelb/rot} > \vec{k}_{grün/gelb}$ und die Ampel schaltet für kleine Eingangswerte in Richtung grün und für hohe Eingangswerte im Richtung rot. Bei $k_{gelb/rot} < k_{grün/gelb}$ arbeitet die Ampel invertiert. Der Eingangswert $\vec{x}$ wird am dritten Eingangsport mit dem Namen *input* erwartet.

In Abbildung 6.10 sind drei Ampelblöcke in den möglichen Zuständen rot, gelb und grün zu sehen.

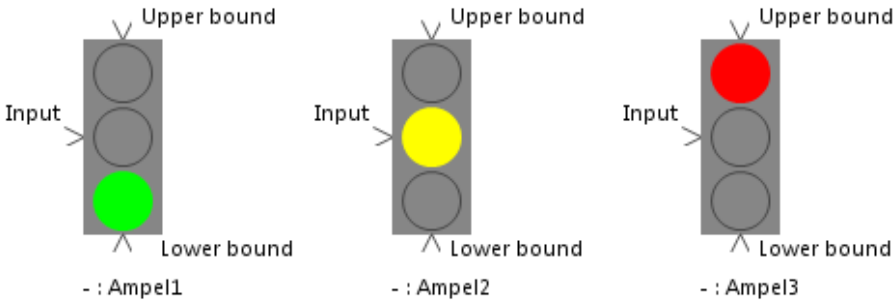


Abbildung 6.10: Ampelblöcke

Im Gegensatz zur in 6.2.1 beschriebenen allgemeinen Darstellung von Metrikblöcken verfügt der Ampelblock über keine Titelleiste mit Blocknamen, Typsymbol und Berechnungszustand. Der Name wird als bewegliche Beschriftung eingeblendet und ist in der Abbildung jeweils unter dem Block dargestellt.

Der Ampelblock ist in der Lage, die gängigen nummerischen Datentypen *Integer*, *Double*, *Float*, *Long*, *Short* und *Byte* [JAPI7] zu verarbeiten. Diese können als Skalar oder als Vektor vorliegen.

Liegt der Eingangswert $\vec{x}$ als n-dimensionalem Vektor, werden skalare Grenzwerte $k_{gelb/rot}$ und $k_{grün/gelb}$ als einheitliche Grenze über alle Dimensionen interpretiert. Liegen die Grenzwerte ebenfalls als n-dimensionaler Vektor vor, gelten die Grenzwerte für die jeweilige Dimension. Bei unterschiedlicher Anzahl von Dimensionen wird ein entsprechender Fehler ausgegeben.

6.2.8 Skalenblock

Def. 62: Skalenblock

> *Ein Skalenblock ist ein Metrikblock (Def. 40), der eingehende Daten relativ und maßstäblich zu konfigurierbaren Bezugsgrößen in Form eines vertikalen Bandes visualisiert.*

Analog zum oben im Abschnitt 6.2.7 beschriebenen Ampelblock, gehört der Skalenblock zu den Ergebnisblöcken einer Metrik (siehe Abbildung 6.11). Mit Hilfe dieses Blocks können Berechnungsergebnisse relativ und maßstäblich zu konfigurierbaren Bezugsgrößen dargestellt werden. Wie beim Ampelblock kommen dazu die Farben rot, gelb und grün zum Einsatz, welche aber verschieden große Intervalle der Skala belegen können und doppelt gegenläufig angeordnet sind, so dass das obere und untere Ende die roten Bereiche markieren und der grüne Zielbereich in der Mitte angeordnet ist.

Der Skalenblock verfügt über fünf Eingangsports und keinen Ausgangsport. Vier der Eingangsports

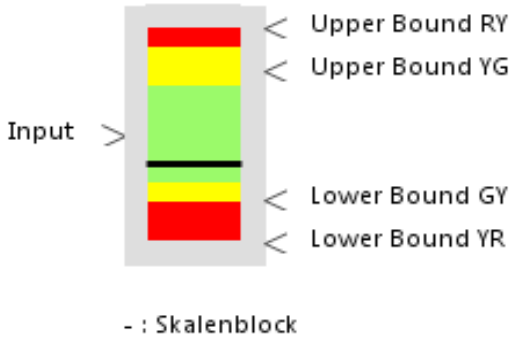


Abbildung 6.11: Skalenblock

dienen zur Konfiguration der Bezugsgrößen. Die *Ports Upper bound YR* und *Upper bound YG* spezifizieren den rot/gelb-Übergang bzw. gelb/grün-Übergang am oberen Ende des Skalenblocks. Die Ports *Lower bound YR* und *Lower bound YG* spezifizieren den rot/gelb-Übergang bzw. gelb/grün-Übergang am unteren Ende des Skalenblocks.

Der zu visualisierende Eingangswert wird am Port *Input* erwartet. Analog zum Ampelblock unterstützt der Skalenblock für alle Eingänge sowohl skalare Eingangswerte als auch Vektor-Eingangswerte. Analog zum Ampelblock in Abschnitt 6.2.7 gelten dieselben Restriktionen bezüglich der Anzahl der Dimensionen bei Vektoren und das Zusammenspiel zwischen Skalaren und Vektoren.

6.2.9 Diskreter Ergebnisblock

Def. 18: Diskreter Ergebnisblock

> *Ein diskreter Ergebnisblock ist ein Metrikblock (Def. 40), der eingehende Daten bezüglich vorgegebener, diskreter Werte vergleicht und das Ergebnis im Fall einer Übereinstimmung grün und andernfalls rot darstellt.*

Ein weiterer Block der Klasse der Ergebnisblöcke ist der diskrete Ergebnisblock. Dieser dient vor allem zur Überprüfung eines Ergebnisses gegen definierte, bekannte Referenzwerte. Gleicht der Eingangswert einem der Referenzwerte, so wird der Block mit einem grünen Rand eingefärbt, wie in Abbildung 6.12 dargestellt. Ist keiner der gegebenen Referenzwerte zum Eingangswert gleich, so wird der Block rot eingefärbt.

Wie Ampel- und Skalenblock, kann auch der diskrete Ergebnisblock sowohl mit skalaren Eingangswerten als auch mit Vektor-Eingangswerten umgehen, indem der Wert jeder Dimension mit den Referenzwerten verglichen wird. Führen alle Vergleiche zu einem positiven Ergebnis, so wird der Block mit grünem Rand gezeichnet, andernfalls wird der Block rot eingefärbt.

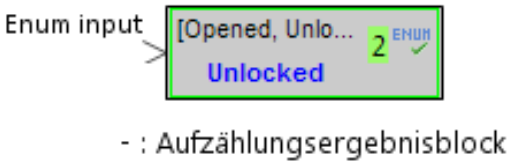


Abbildung 6.12: Aufzählungsergebnisblock

6.2.10 Report- und Dokumentationsblock

Def. 58: Report- und Dokumentationsblock

> *Ein Report- und Dokumentationsblock ist ein Metrikblock (Def. 40), der eingehende Daten einem Dokumentengenerator übergibt.*

Der Report- und Dokumentationsblock erlaubt die Anbindung von Metrikergebnissen an einen Dokumentengenerator. Dazu kann er mit beliebig vielen Eingangsports bestückt werden. Die an diesen Ports anliegenden Daten werden zu einer Tabelle zusammengefasst, wobei die Portnamen als Spaltennamen dienen, und die Werte in diese Spalte eingetragen werden. Liegt ein einem Port bereits eine Tabelle an, so wird die Ergebnistabelle um die Eingangstabelle erweitert.

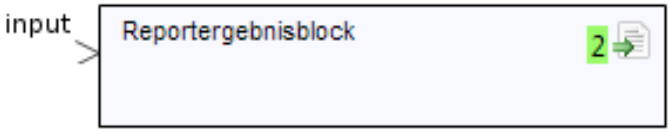


Abbildung 6.13: Report- und Dokumentationsblock

Die Ergebnistabelle kann bei Bedarf sortiert und gefiltert werden und wird dann bei der Dokumentengenerierung abgerufen und dessen Informationen in das Dokument eingetragen. Abbildung 6.13 Zeigt einen Report-Dokumentationsblock mit einem Eingangsport.

6.2.11 Schleifenblock

Def. 60: Schleifenblock

> *Ein Schleifenblock ist ein Metrikblock (Def. 40), der über beinhaltete Metrikblöcke bezüglich einer vorgegebenen Anzahl von Elementen iteriert. Die Teilergebnisse der Iterationsläufe werden als Gesamtergebnis zusammengefasst und weitergegeben.*

Der Schleifenblock ermöglicht die iterative Berechnung von Metrikblöcken, die innerhalb des Schleifenblocks liegen. Als Eingabe erwartet der Schleifenblock eine Liste von Elementen, über die

iteriert wird. Nach Ablauf aller Iterationsschritte wird eine Tabelle zurückgegeben, welche zu jedem Iterationsdurchgang die jeweiligen Einzelergebnisse zurück liefert.

Der Schleifenblock verfügt über einen Eingangsport, der durch Namen *Input* gekennzeichnet ist. Über diesen Port wird eine Liste von Elementen erwartet, über die iteriert wird. Dieser Port verfügt über zwei graphische Repräsentationen. Die externe Repräsentation fungiert als Datenzielport, um den Schleifenblock mit einem äußeren Block verbinden zu können. Die interne Repräsentation orientiert sich an den Startzustand von UML-Zustandsdiagrammen. Sie dient als Datenquellport und liefert bei jeder Iteration einen Wert der Eingangsmenge an die internen Blöcke. Weitere Eingangsports müssen andere Namen tragen und können zusätzliche Daten in die Schleife liefern. Diese sind Invarianten der Schleife und werden vor Betreten des Blocks berechnet und bleiben über alle Iterationen konstant.

Der Schleifenblock kann beliebig viele Teilergebnisse zurückliefern. Für jedes der Teilergebnisse wird ein Ausgangsport modelliert. Jedes Teilergebnis wird sowohl durch einen internen als auch einen externen Port visualisiert. Die Repräsentation des internen Ports orientiert sich an Endzustände von UML-Zustandsdiagrammen. Dieser Port dient als Zielport für die innerhalb der Schleife durchgeführten Metrikblöcke. Er gibt das Teilergebnis an die externe Portrepräsentation weiter, die als Datenquellport das Ergebnis an die nachfolgenden Blöcke weiterreicht.

Zudem gibt es einen weiteren Ausgangsport mit dem Namen <*table*>, der über keine interne Repräsentation verfügt. Dieser Port fasst alle Teilergebnisse in einer Tabelle zusammen. Die erste Spalte beinhaltet die Eingangswerte und jedes Teilergebnis belegt eine weitere Spalte.

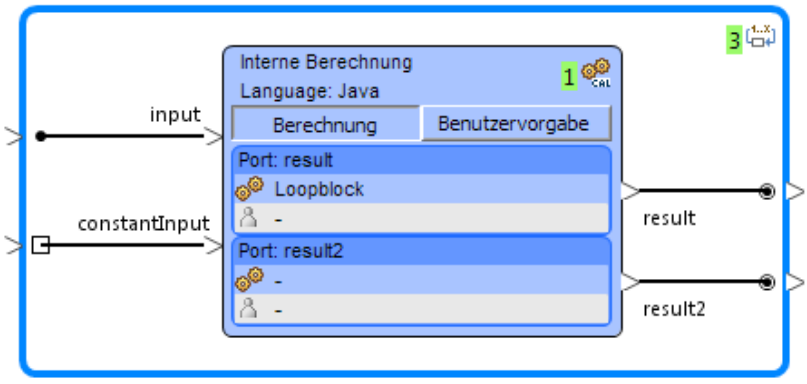


Abbildung 6.14: Schleifenblock

In Abbildung 6.14 wird ein Schleifenblock mit einem schleifen-varianten Eingang (*input*), einem schleifen-invarianten Eingang (*constantInput*) sowie zwei Ausgängen dargestellt. Im inneren des

Schleifenblocks ist beispielsweise ein Berechnungsblock (*Interne Berechnung*) gezeigt. In diesem Bereich können auch wesentlich komplexere Metriken, bestehend aus einer Vielzahl von Metrikblöcken, enthalten sein. Schleifenblöcke dürfen ebenso ineinander geschachtelt sein. Hier ist jedoch zu beachten, dass mit solchen Strukturen der Berechnungsaufwand und damit die Berechnungszeit stark zunehmen können.

6.2.12 Interne Schleifen

In einigen Fällen ist es sinnvoll, das Blöcke während ihrer eigenen Berechnung benötigte Teilberechnungen an andere Blöcke delegieren. In diesem Fall wird die eigene Berechnung solange unterbrochen, bis die Teilergebnisse vorliegen und dann weiter fortgeführt. Zur Berechnung des Blockes kann eine interne Schleife beliebig oft (mit unterschiedlichen Übergabeparametern) ausgeführt werden. Ein Block kann über mehrere interne Schleifen verfügen, jede verfügt über mehrere interne Eingangs- und Ausgangsports, um mehrere Eingabeparameter übergeben zu können und mehrere Ergebnisse entgegennehmen zu können.

Die internen Eingangs- und Ausgansports haben eine eigenständige, pfeilförmige graphische Repräsentation, die von den normalen Ports abweicht.

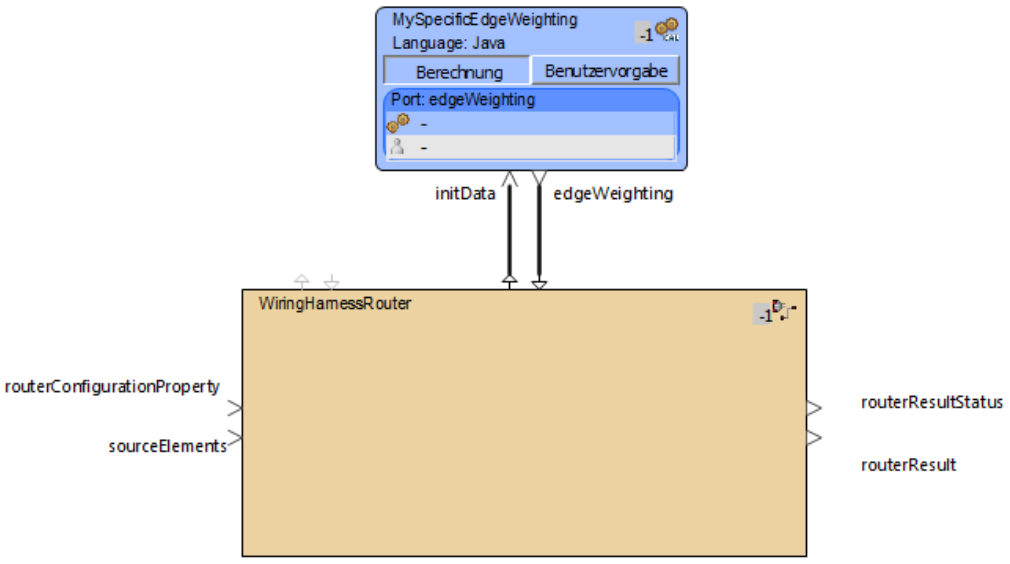


Abbildung 6.15: Interne Schleifen

In Abbildung 6.15 ist ein Beispiel für interne Schleifen dargestellt. Der benutzerspezifische Block (siehe Abschnitt 6.2.13) *WiringHarnessRouter* (vgl. Abschnitt 6.2.13.3) verfügt über zwei interne Schleifen mit je einem internen Datenquellport und einem internen Datenzielport.

Die interne Schleife mit den grauen Ports dient zur Ankopplung eines eigenen Routingalgorithmus. Dieser ist hier nicht belegt, in diesem Fall wird der Standardrouter verwendet (das Verhalten nicht angeschlossener interner Schleifen ist implementierungsabhängig, damit gilt eine optionale Belegung nicht für alle interne Schleifen).

Die interne Schleife mit den schwarzen Ports dient zum Anschluss eines eigenen Kantengewichters. Damit können die Routingergebnisse an die individuellen Bedürfnisse, wie zum Beispiel die Beachtung spezieller Bedingungen bei der Verlegung von Leitungen, angepasst werden.

6.2.13 Benutzerspezifische Blöcke

Benutzerspezifische Blöcke werden im Metrikeditor als *CustomizedBlock* (siehe 6.4.11) repräsentiert. Sie werden in Aussehen, Ein- und Ausgabesignatur und Funktion durch zusätzlich installierbare Module konfiguriert, die in den Metrikeditor integriert werden. Diese Module werden als Eclipse-Plugins ausgeführt, die den vorgegebenen Erweiterungspunkt *customizedBlock* nutzen. Über diese Erweiterung werden dem Metrikeditor Implementierungen zur Verfügung gestellt, die Aussehen, Aufbau und Funktion des benutzerdefinierten Blocks beschreiben.

In Abbildung 6.15 ist ein benutzerspezifischer Block zu sehen, welcher den in PREEvision integrierten Leitungssatzrouter konfiguriert, auf das E/E-Architekturmodell ausführt und die Ergebnisse zur nachfolgenden Auswertung auf Datenquellports zur Verfügung stellt. Im Folgenden werden einige der wichtigsten benutzerspezifischen Blöcke vorgestellt.

6.2.13.1 Excel-Exportblock

Der Excel-Exportblock ist eine Erweiterung, um Daten in Exceldateien zu exportieren. Der Block setzt dazu zwei Eingangsports voraus, über welche Dateiname und Name der Tabelle erwartet werden. Die zu exportierenden Daten werden über weitere Eingangsports an den Exportblock übergeben. Je nach Eingangsdatenformat (skalare Werte, Wertelisten und Wertetabellen) werden die Informationen in der Zieltabelle abgelegt.

6.2.13.2 Generischer Block für ausführbare Einheiten

Wie in Kapitel 5.4 beschrieben, ist das Metrikframework als in bestehende Werkzeuge integrierbare Lösung konzipiert. Diese bestehenden Werkzeuge (z.B. PREEvision) sind bereits mit einer Vielzahl von Programmfunktionen ausgestattet. Viele dieser Programmfunktionen sind auch für den Einsatz in Metriken sinnvoll, obwohl sie die hierfür notwendigen Schnittstellen und Adapterimplementie-

rungen nicht erfüllen. Da es sich hierbei um sehr viele Programmfunktionen handeln kann, deren Anpassung die Stabilität der Werkzeuge beeinträchtigen können und mit hohem Implementierungs- und Testaufwand verbunden sind, kommt eine minimal-invasive Methode zum Einsatz.

Diese Methode wurde im Rahmen einer Diplomarbeit [BRE10] behandelt, um das MetrikFramework entsprechend zu erweitern. Die erarbeitete Lösung basiert auf Java-Annotationen [LOU07], welche die schnittstellenrelevanten Stellen im Quellcode der Programmfunktionen markieren. Hierbei kommen Annotationen für Konstruktoren, Felder und Methoden zum Einsatz. Eine allgemeine Adapterimplementierung kann die annotierten Programmfunktionen instanziieren und als Block im Metrikeditor zur Verfügung stellen. Die zur Ausführung benötigten Daten erwartet der Block über angelegte Eingangsports. Die Ausgaben der Programmfunktion werden der Metrik über Ausgangsports zur Verfügung gestellt. Die Ausführung der Metrik wird von der Metrik-Ausführungsschicht (siehe 6.5) an die Adapterimplementierung delegiert.

Im Rahmen der genannten Diplomarbeit wurde eine Vielzahl von PREEvision-Programmfunktionen (ca. 150) für den Metrikeditor annotiert. Darunter sind auch sehr E/E-spezifische Funktionen wie eine Stromlaufplansynthese oder Komfortfunktionen für dem Umgang mit Varianten.

6.2.13.3 Routerblöcke

Das Routing von Signalen in der Netzwerktopologie und des Leitungssatzes in die geometrische Topologie sind in PREEvision integrierte, laufzeitoptimierte und konfigurierbare Funktionen. Diese Funktionen können zusätzlich als benutzerspezifische Blöcke in Metriken verwendet werden. Dazu werden alle benötigten Einstellungen und Eingangsdaten über Eingangsports erwartet. Die Ausgabedaten werden über Ausgangsports zurück an die Metrik geleitet. Je nach Konfiguration nehmen die Routingblöcke Änderungen am Modell vor, indem sie das Routingergebnis auf das Modell anwenden.

6.2.14 Metrikausführer

Das Metrikausführer-Artefakt ist kein Block, welches im Metrikdiagramm dargestellt wird oder als Block ausgeführt wird. Es ist ein Artefakt, welches eine Klammer um eine Metrik oder einen Teil einer Metrik bildet und mehrere Startpunkte und einen Endpunkt der Metrik definiert. Zusätzlich wird der Metrikausführer in einen externen Kontext eingebunden, in welchem die Metrik ausgeführt wird und an welchen die Ergebnisse ausgegeben werden. Als externe Kontexte kommen Tabellen, Aufrufe über das Kontextmenü auf anderen Artefakten, Änderungen an Artefakten oder ein Doku-

mentengenerator in Frage.

Der externe Kontext liefert an den durch den Metrikausführer festgelegten Startpunkten der Metrik die zu übergebenden Informationen. Zum Ausführungszeitpunkt wird der referenzierte Endpunkt der Metrik berechnet. Durch die überschriebenen Startpunkte rechnet die Metrik ein kontextspezifisches Ergebnis aus und liefert es an den externen Kontext zurück.

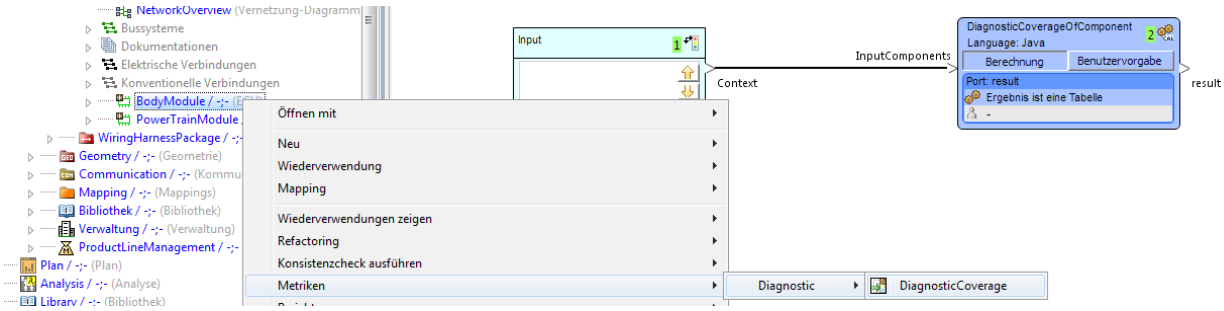


Abbildung 6.16: Aufruf eines Metrikausführers über das Kontextmenü

In Abbildung 6.16 ist als Beispiel der Aufruf eines Metrikausführer *DiagnosticCoverage* über das Kontextmenü auf dem Artefakt *BodyModule* gezeigt. Im Hintergrund des Kontextmenüs ist die entsprechende Metrik zu sehen. In diesem konkreten Beispiel wird der Metrik-Kontextblock, der in diesem Beispiel in der Metrik leer ist, mit dem Artefakt *BodyModule* überschrieben und die Metrik wird somit auf diesem Artefakt ausgeführt.

6.2.15 Konzeption Sensitivitätsanalyse

Die Sensitivitätsanalyse ermöglicht eine Metrikberechnung anhand von Intervallen anstelle konkreter Werte. Berechnungs- und Parameterblöcke können dazu entsprechend umgeschaltet werden. Ausgehend vom Ausgabewert (dies kann beim Berechnungsblock je nach aktivem Modus ein berechneter Wert oder ein benutzerdefiniertes Ergebnis sein) eines Datenquellports können absolute oder relative Ober- und Untergrenzen sowie Schrittweiten angegeben werden.

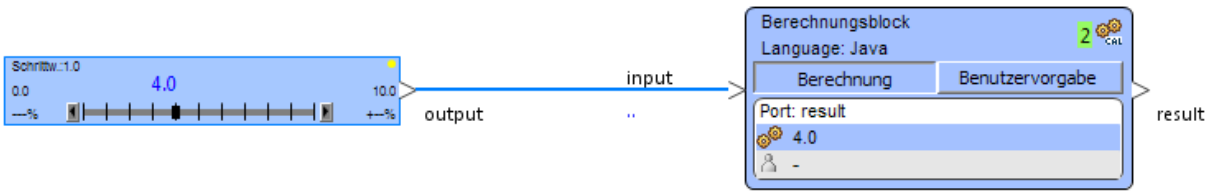


Abbildung 6.17: Sensitivitätsanalyse

Abbildung 6.17 zeigt einen Parameterblock (vgl. Abschnitt 6.4.8) mit aktivierter und vorberechne-

ter Sensitivitätsanalyse. Bei der Berechnung werden in einem erstem Schritt alle möglichen Werte ermittelt und zwischengespeichert, was je nach Anzahl der Blöcke mit eingeschalteter Sensitivitätsanalyse und der Anzahl zu berechnender Schritte etwas dauern kann. In einer nachgelagerten Schritt ist es möglich, mit dem Schiebebalken Wertänderungen anzuwenden und den Effekt auf die Gesamtmetrik zu analysieren.

6.3 Visualisierung von Metriken

In diesem Abschnitt soll die Visualisierung kompletter Metriken detailliert beschrieben werden. Zentrale Stelle ist die in dieser Arbeit entwickelte graphische Notation für Metriken in Form von Metrikdiagrammen. Die graphische Notation der einzelnen Artefakte einer Metrik sind im vorangegangenen Kapitel 6.2 beschrieben. Im folgenden Abschnitt 6.3.1 wird auf die Visualisierung kompletter Metriken eingegangen.

Ergänzend dazu gibt es tabellarische Ansichten, welche spezifische Teilbereiche von Metriken beleuchten und den Metrikentwickler so bei seiner Arbeit unterstützen. Diese werden im Kapitel 6.3.2 beschrieben.

6.3.1 Diagrammorientierte Darstellung

Wie bereits in Abschnitt 6.2 beschrieben, ist ein Metrikdiagramm Datenfluss-orientiert. Im allgemeinen ist daher die Leserichtung von links nach rechts, die Datenquellen sind auf der linken und die Datensenken auf der rechten Seite angeordnet. Hierbei handelt es sich aber um eine Konvention und nicht um eine zwingende Vorgabe. Die im Rahmen dieser Arbeit vorgestellten Metriken wurden nach dieser Konvention modelliert.

Technisch gesehen basiert der Metrikdiagramm-Editor auf dem in Kapitel 2.9.3 vorgestellten Generic Diagram Framework GDF und beinhaltet somit über die Grundfunktionalitäten.

Abbildung 6.18 zeigt einen Metrikdiagramm-Editor mit einer berechneten Beispielmetrik. Das zuschaltbare Raster erleichtert Skalierung und Anordnung der Metrikblöcke. Im rechten Teil ist die Palette zu sehen, über welche Metrikartefakte per Drag and Drop in das Diagramm gezogen und neu angelegt werden. Im oberen rechten Teil ist die Toolbar des Diagrammeditors eingeblendet. Sie enthält zahlreiche Funktionen für Zoom, Raster, Ausrichtungs- und Linienassistenten, graphische Filter und Metrik-Berechnungsfunktionen. Darüber hinaus gibt es ein Kontextmenü mit selektionsspezifischen Aktionen.

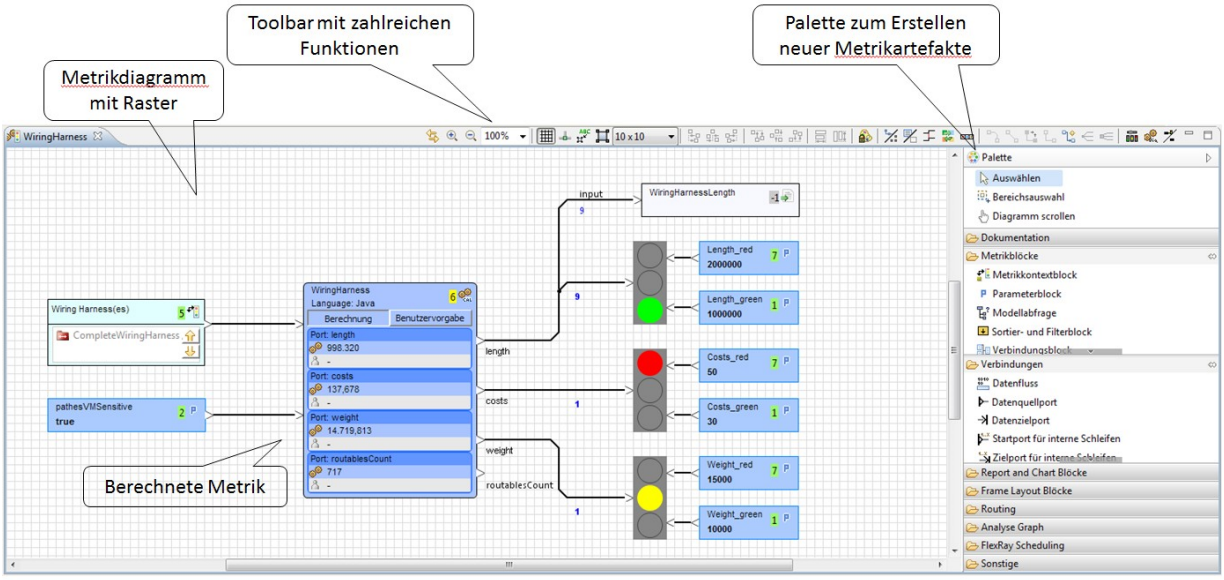


Abbildung 6.18: Metrikdiagrammeditor mit einer berechneten Metrik

Bei der in der Abbildung 6.18 dargestellten Metrik handelt es sich um die Leitungssatz-Analysemetrik, die in Abschnitt 7.1.3.2 näher behandelt wird.

Metrikdiagramme stellen, je nach Modellierung, komplette Metriken oder Teilmetriken dar. Die graphische Repräsentation hat keinen Einfluss auf die Berechnung der Metrik, die ausschließlich durch die Struktur der Metrikartefakte und deren Vernetzung abhängt. Somit ist es möglich, für eine Metrik mehrere Metrikdiagramme zu erstellen, die für den jeweiligen Anwendungszweck zugeschnitten sind.

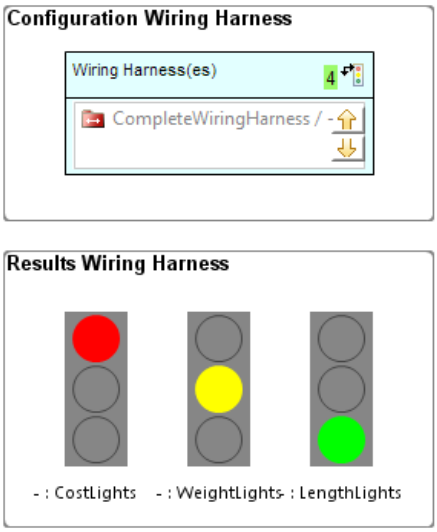


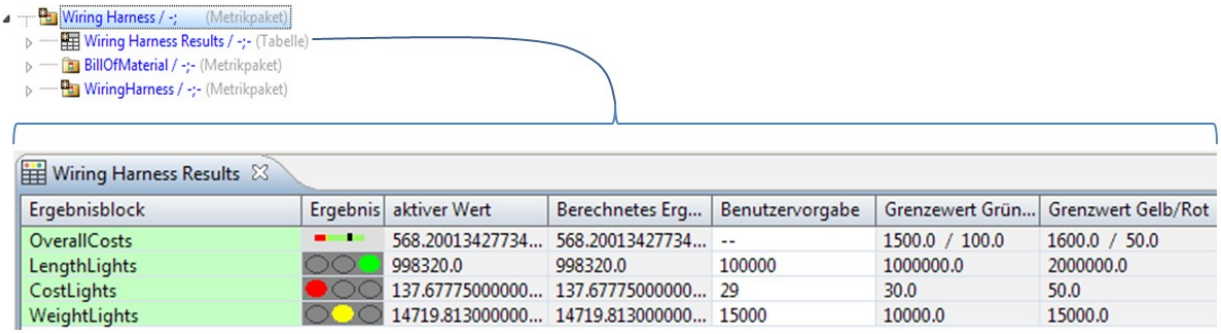
Abbildung 6.19: Ergebnisorientierte Darstellung einer Metrik

In Abbildung 6.19 ist eine ergebnisorientierte Darstellung der Leitungssatz-Analysemetrik zu sehen.

Im oberen Kasten ist der Metrik-Kontextblock zur Konfiguration der Metrik dargestellt, im unteren Kasten sind die Ergebnisblöcke visualisiert. Alle anderen Artefakte der Metrik wie Berechnungs- und Parameterblöcke, Ports und Datenflüsse sind für diese Darstellung nicht relevant und sind daher ausgeblendet. Dennoch kann die Metrik in dieser Darstellung komplett berechnet werden.

6.3.2 Tabellenorientierte Ergebnisdarstellung

Zur kompakten Darstellung vieler Berechnungsergebnisse eignet sich die Metrikergebnistabelle, wie sie in Abbildung 6.20 im unteren Teil dargestellt ist. Die Tabelle selbst wird im Metrikmodell als eigenständiges Artefakt unterhalb eines Metrikpakets repräsentiert, im oberen Teil von Abbildung 6.20 als *Wiring Harness Results* zu sehen. Diese Tabellenmodellierung geht auf das Konzept der Analysetabellen (MAT09 Seite 139ff.) zurück.



Ergebnisblock	Ergebnis	aktiver Wert	Berechnetes Erg...	Benutzervorgabe	Grenzewert Grün...	Grenzwert Gelb/Rot
OverallCosts		568.20013427734...	568.20013427734...	--	1500.0 / 100.0	1600.0 / 50.0
LengthLights		998320.0	998320.0	100000	1000000.0	2000000.0
CostLights		137.67775000000...	137.67775000000...	29	30.0	50.0
WeightLights		14719.813000000...	14719.813000000...	15000	10000.0	15000.0

Abbildung 6.20: Metrikergebnistabelle

In der Tabelle wird für alle Ergebnisblöcke in dem Teilbaum, welcher durch das übergeordnete Metrikpaket des Tabellenartefakts aufgespannt wird, eine Zeile angelegt. Als Spalten kommen relevante Informationen über die Ergebnisblöcke zum Einsatz. In diesem Beispiel stammt der Ergebnisblock *OverallCosts* aus der *BillOfMaterial*-Metrik im gleichnamigen Metrikpaket. Die anderen Ergebnisblöcke kommen aus der aus Abschnitt 6.3.1 bekannten Leitungssatz-Analysemetrik.

6.4 Metamodell

In diesem Kapitel wird das Metrik-Metamodell beschrieben. Bei dem Metamodell handelt es sich um ein eigenständiges MOF-basiertes (siehe 2.1.1) Metamodell. Die in Abschnitt 6.2 vorgestellten Artefakte von Metriken werden durch dieses Metamodell spezifiziert. Instanzen dieses Metamodells sind Metrikmodelle.

Obwohl das Metrik-Metamodell als eigenständigs Metamodell entworfen ist, ist es in das in Abschnitt 3.3 beschriebene Metamodell zur Beschreibung von Elektrik-/ Elektronikarchitekturen inte-

griert. Berechnungsergebnisse von des domänenspezifischen Modells (E/E-Architekturen) sind nachvollziehbar, da die Bewertungsvorschriften in Form des Metrikmodells mit dem domänenspezifischen Modell verknüpft sind[16].

Graphische Informationen von Metrikdiagrammen, wie zum Beispiel Position, Breite und Höhe von Blöcken werden in eigenständigen Modellartefakten beschrieben und mit den metrikspezifischen Artefakten verknüpft. Das entsprechende technische Konzept ist in Abschnitt 2.9.3 beschrieben (vgl. Abbildung 2.29).

6.4.1 Kompositionshierarchie

Die Kompositionshierarchie im Metrik-Metamodell unterstützt sowohl das Container-Konzept (siehe 3.2.2.2) als auch das paketorientierte Hierarchisierungskonzept (siehe 3.2.2.3). Wie in Abbildung 6.21 oben links dargestellt, bietet das Metrik-Metamodell eine eigene Modellwurzel *MetricModelRoot* an, um eigenständige, unabhängige Instanzen eines Metrikmodells zu ermöglichen (siehe Abschnitt 3.2.2.1).

Über die Metaklasse *MetricModel* wird das eingangs angesprochene Container-Konzept umgesetzt. Über die vererbte Komposition zu *AbstractModelRoot* wird die Containerklasse in den Modellbaum eingehängt und dient als Ausgangspunkt für das komplette Metrikmodell. Ferner erbt diese Klasse von *MetricContentOwner* und kann somit über die Komposition zu *MetricContent* Metrikinhalte (siehe Abschnitt 6.4.2) aufnehmen.

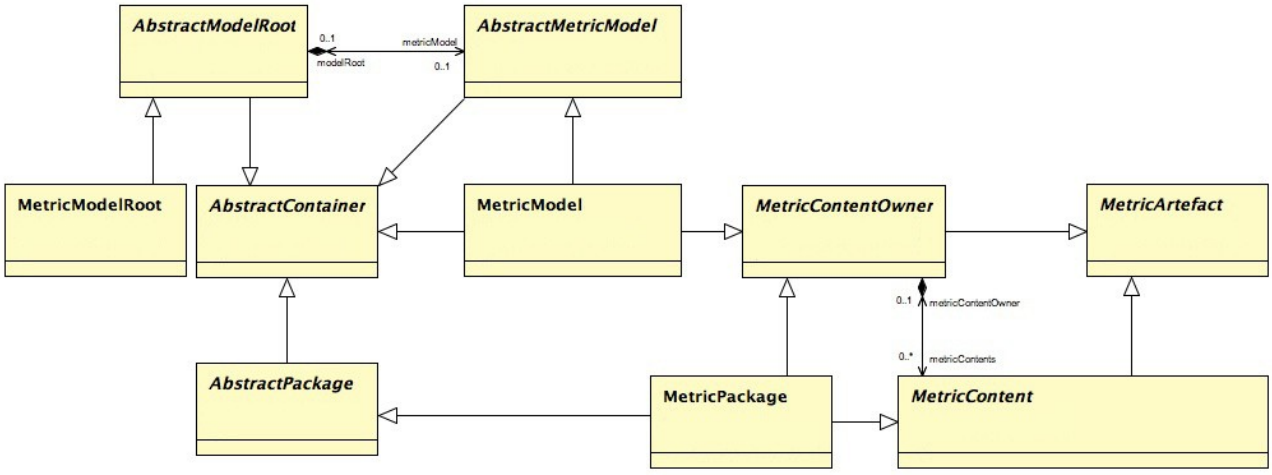


Abbildung 6.21: Kompositionshierarchie im Metrik-Metamodell

16 Dies gilt für einen spezifischen Modellstand. Durch Änderungen des domänenspezifischen Modells oder der Berechnungsvorschriften sind andere Berechnungsergebnisse möglich.

Die Metaklasse *MetricPackage* realisiert die paketorientierte Kompositionshierarchie. Dazu erbt sie sowohl von *MetricContentOwner*, um Metrikinhalte aufnehmen zu können, als auch von *MetricContent* selbst und ist somit selbst ein Metrikinhalt.

6.4.2 Metrikblöcke

Metrikblöcke stellen die Knoten in einem Metrikdiagramm dar und repräsentieren semantisch Übergänge, an denen die Ausführungsschicht (siehe 6.5) Berechnungen durchführt und so Zustandsänderungen innerhalb des Berechnungsfortschritts einer Metrik herbeiführt.

Im Metamodell erben alle Metrikblöcke von der abstrakten Klasse *AbstractMetricBlock*. Diese erbt wiederum von der oben beschriebenen Metaklasse *MetricContent* und ist so Teil des Hierarchisierungskonzepts eines Metrikmodells.

Grundsätzlich lassen sich Metrikblöcke in drei Kategorien einteilen, die im Folgenden näher beschrieben werden.

- *DataSourceBlock*: Allgemeine Superklasse für Metrikblöcke, die ausschließlich als Datenquelle dienen und somit nur über *DataSourcePorts* verfügen können.
- *DataTargetBlock*: Allgemeine Superklasse für Metrikblöcke, die ausschließlich als Datenziel dienen und somit nur über *DataTargetPorts* verfügen können.
- *DataSourceTargetBlock*: Allgemeine Superklasse für alle Metrikblöcke, die sowohl über Datenquell- als auch über Datenzielports verfügen können. Diese Klasse erbt diese Konzepte daher direkt von den oben beschriebenen Metaklassen *DataSourceBlock* und *DataTargetBlock*.

Die meisten Metrikblöcke erben von *DataSourceTargetBlock*, da für sie die Notwendigkeit besteht, Daten empfangen und verschicken zu können.

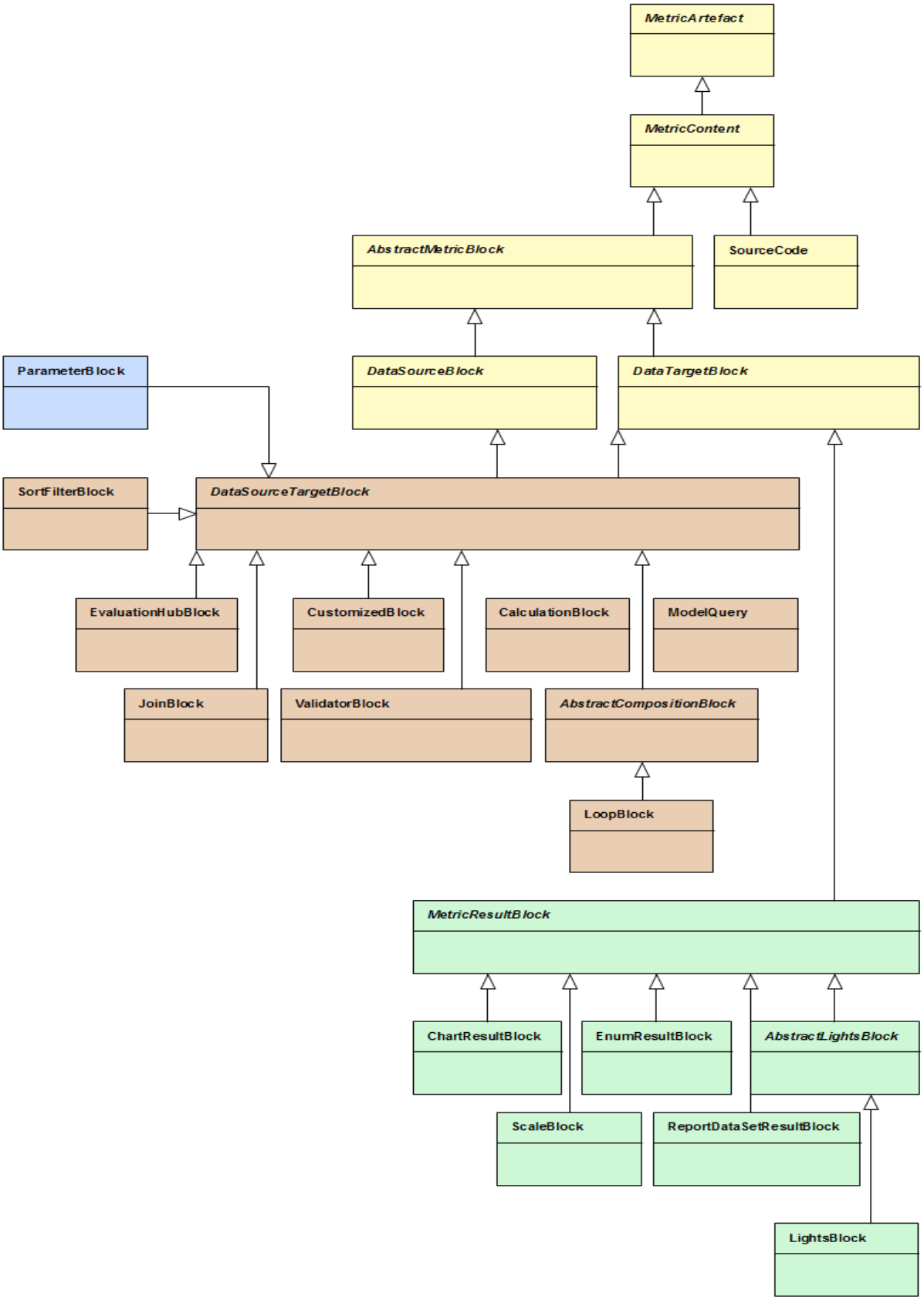


Abbildung 6.22: Vererbungshierarchie im Metrik-Metamodell

In Abbildung 6.22 ist die Vererbungshierarchie für alle Metrikblöcke zu sehen. Darin zeigt sich die Aufteilung der Metrikblöcke in die oben beschriebenen drei Kategorien. Die meisten Metrikblöcke können sowohl Daten empfangen als auch Daten versenden. Diese erben von *DataSourceTarget-Block* und sind im Diagramm links unten angeordnet. Der einzige Block, der nur Daten senden kann, ist der *ParameterBlock*, dieser erbt direkt von *DataSourceBlock*. Metrikergebnisblöcke können nur Daten empfangen, erben daher direkt von *DataTargetBlock*. Sie sind im rechten unteren Teil des Diagramms dargestellt.

Im oberen Teil des Diagramms ist die allgemeine Vererbungshierarchie der Metrikblöcke dargestellt. Alle Artefakte des Metrik-Metamodells erben von *MetricArtefact*. Die Metaklasse Metric-Content dient, wie in Abschnitt 6.4.1 beschrieben, für die Realisierung der Kompositionshierarchie, der alle Metrikblöcke (*AbstractMetricBlock*) unterworfen sind.

Die Metaklasse *SourceCode* repräsentiert keinen Metrikblock und wird im Abschnitt 6.4.9 detailliert beschrieben.

6.4.3 Datenfluss

Datenflüsse realisieren die Kommunikationsstrecken zwischen Blöcken. Dabei ist ein Datenfluss immer mit genau einem Datenquellport verbunden und kann beliebig viele Datenzielports bedienen. Dieser Umstand erlaubt die in Abbildung 6.23 dargestellte Modellierung von Datenflüssen.

Der Datenfluss wird per Komposition dem Datenquellport zugeordnet. Dies hat zur Folge, dass im Falle der Löschung des Datenquellports der Datenfluss mitgelöscht wird. Es kann demnach keine Datenflüsse geben, die keinem Datenquellport zugeordnet sind.

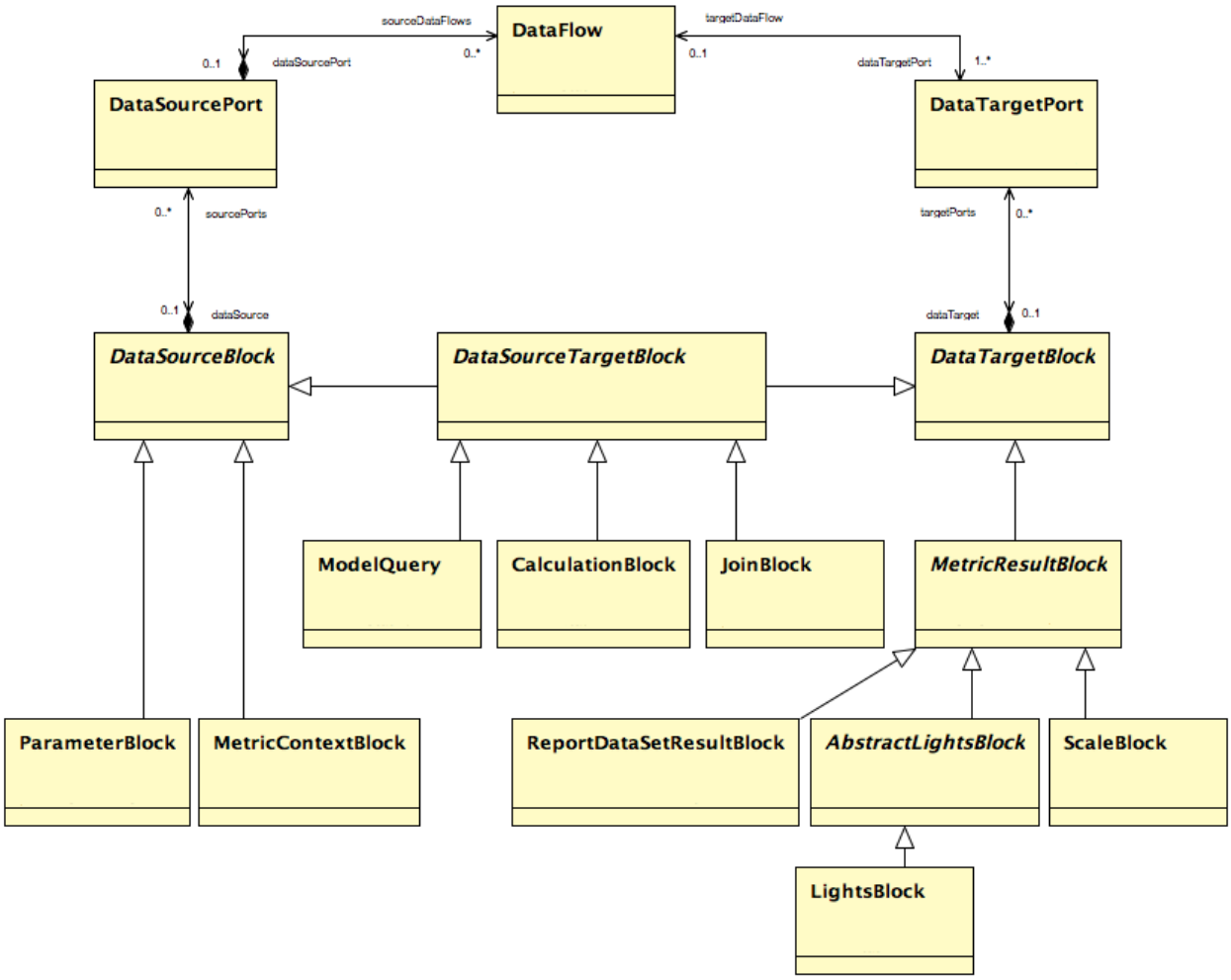


Abbildung 6.23: Datenfluss im Metrik-Metamodell

Die Datenzielports werden vom Datenfluss aus assoziiert. Im Gegensatz zum Datenquellport wird der Datenfluss nicht mitgelöscht, wenn ein Zielport aus dem Modell entfernt wird.

6.4.4 Metrik-Kontextblock

Der Metrik-Kontextblock wird im Metrik-Metamodell durch die Metaklasse *MetricContextBlock* abgebildet. Durch die in Abbildung 6.24 dargestellte Vererbung zu *DataSource* dient dieser Block lediglich als Datenquelle und kann nicht als Ziel eines Datenflusses eingesetzt werden.

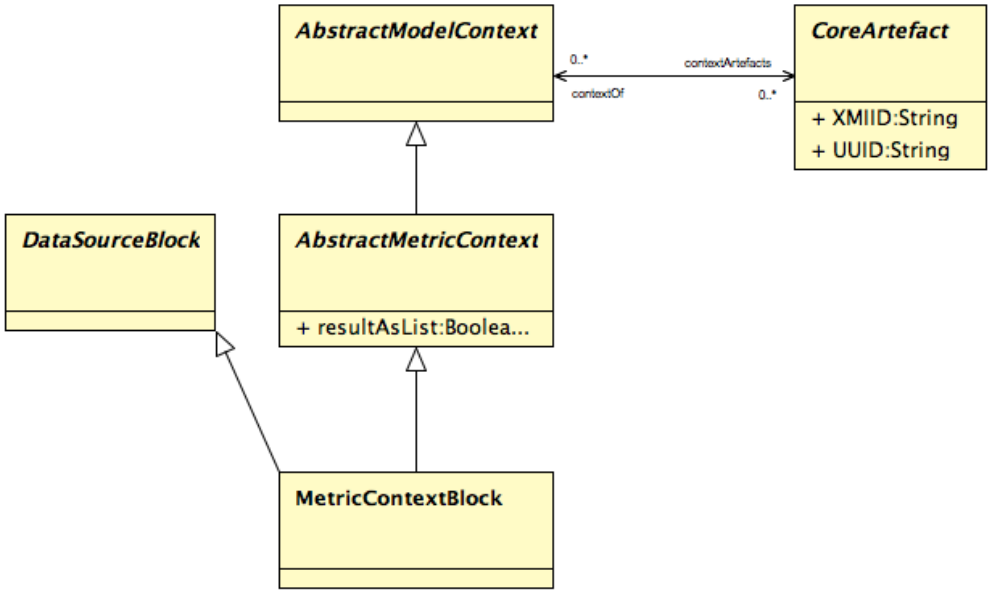


Abbildung 6.24: Kontextblöcke im Metrik-Metamodell

Die Vererbung zur Metaklasse *AbstractMetricContext* realisiert die Anbindung an das generische Konzept für Modellkonzepte. Dieses Konzept erlaubt es durch die Vererbung zu *AbstractModelContext*, beliebige Artefakte (*CoreArtefact*) einer allgemeinen Kontextklasse (*AbstractModelContext*) über eine Assoziation zuzuordnen. Die so referenzierten Artefakte des Modells bilden den Inhalt des Metrik-Kontextblocks.

6.4.5 Modellabfragen

Bei Modellabfragen (*ModelQuery*) handelt es sich um eine Gruppierung von Abfrageregeln (*QueryRule*). Abfrageregeln können beliebig durch Modellabfragen gruppiert und ggf. in anderen Modellabfragen wiederverwendet werden. Die Abfrageregel selbst ist dabei lediglich ein Stellvertreter auf eine Regel in einem externen Regelmodell, wo die genaue Semantik der Abfrageregel spezifiziert wird. Die Abfrageregel verfügt dazu über das vererbte Attribut *id*, über das die Regel eindeutig identifiziert werden kann. Sowohl die Modellabfragen, als auch die Abfrageregeln werden durch die Vererbung zu *QueryPackageContent* in Abfragepaketen (*QueryPackage*) hierarchisiert.

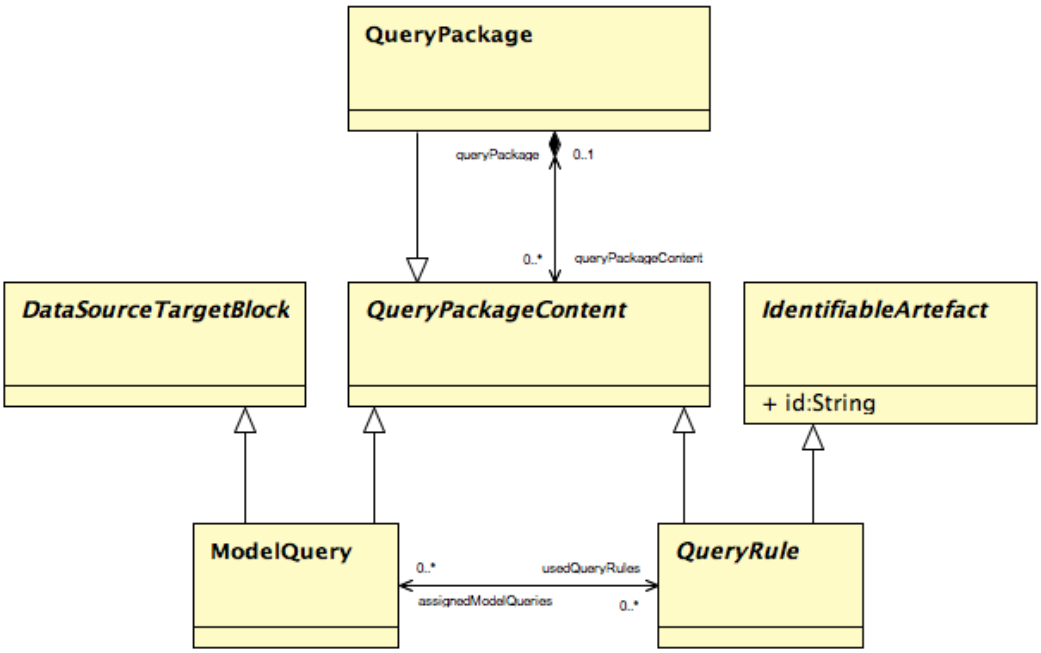


Abbildung 6.25: Modellabfragen im Metrik-Metamodell

Die berechneten Ergebnisse der Modellabfrage sind prinzipiell volatil, das heißt, dass es im Gegensatz zum Metrik-Kontextblock (siehe 6.4.4) weder im Modell noch im Metamodell einen Bezug zwischen einem getroffenen Artefakt und der Modellabfrage selbst bzw. der ausgeführten Abfrageregel gibt.

6.4.6 Joinblock

Der Joinblock ist in der Lage, zwei Tabellen, die über zwei Datenzielports eingehen, bezüglich eines Kriteriums zu vereinigen (Join, aus dem englischen für vereinigen) und als Tabelle weiterzugeben. Im Metrik-Metamodell gibt es dazu eine eigene Metaklasse JoinBlock, die in Abbildung 6.26 dargestellt ist.

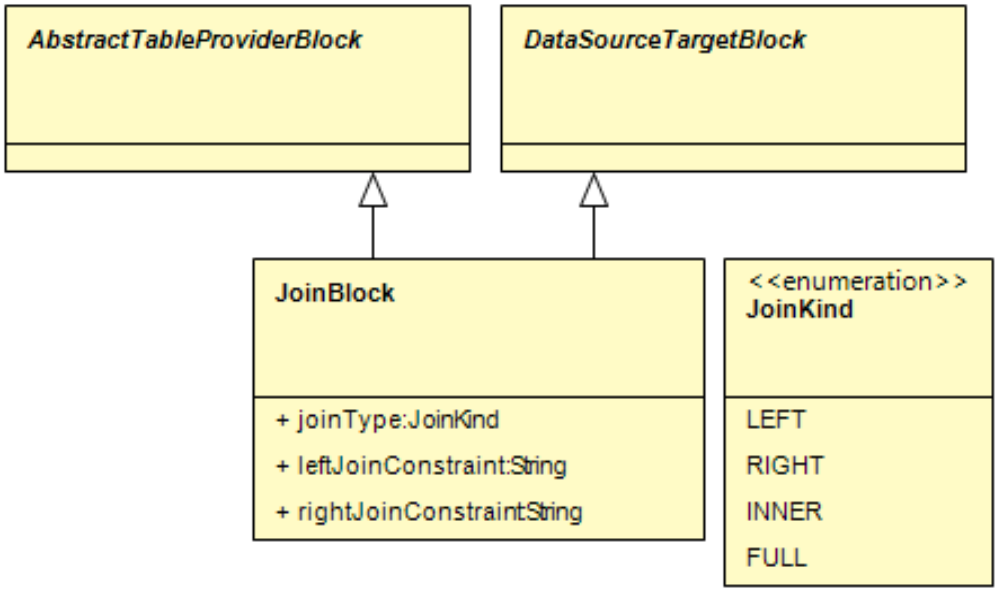


Abbildung 6.26: Joinblock im Metrik-Metamodell

JoinBlock erbt von *AbstractTableProviderBlock*, um Filter- und Sortiereinstellungen für die Ergebnistabelle modellieren zu können. Als *DataSourceTargetBlock* kann der Joinblock sowohl Datenziel- als auch Datenquellports beinhalten.

Im Attribut *joinType* des *JoinBlock* wird die Vereinigungsart eingestellt. Dieses Attribut ist vom Typ *JoinKind*, einer Enumeration, welche die Werte *FULL*, *INNER*, *LEFT* und *RIGHT* annehmen kann. Die Semantik der einzelnen Werte und auch der anderen Attribute ist in Abschnitt 6.2.4 beschrieben. Die Eigenschaften *leftJoinConstraint* und *rightJoinConstraint* enthalten Informationen darüber, welche Daten der linken und rechten Eingangstabelle für die Join-Operation herangezogen werden sollen.

6.4.7 Interne Schleifen

Interne Schleifen dienen vor allem dazu, vorhandene Metriken innerhalb von komplexen Metrikblöcken wiederzuverwenden. Metrikblöcke können auf diese Weise während ihrer Berechnung Ergebnisse anderer Metriken anfordern, die sie für ihre weitere Berechnung benötigen. Durch die spezialisierten Datenports *InternalLoopDataSourcePort* und *InternalLoopDataTargetPort* werden interne Schleifen, wie in Abbildung 6.27 dargestellt, an das normale Datenfluss-Konzept angeschlossen.

Diese spezialisierten Datenports sind mit einer internen Schleife (*InternalLoop*) assoziiert. Damit werden die zusammengehörenden Ports einer Schleife in Bezug gesetzt. Über die Komposition von

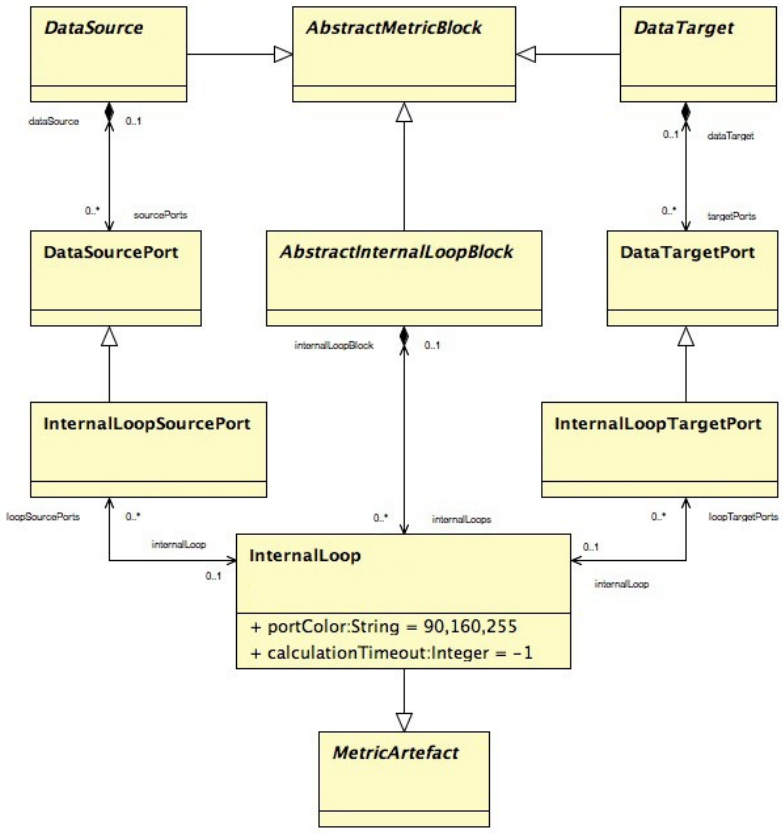


Abbildung 6.27: Konzept für interne Schleifen

InternalLoop zu *AbstractInternalLoopBlock* (siehe Abbildung 6.27) kann ein Block somit über mehrere voneinander unabhängigen internen Schleifen verfügen. Über das Attribut *portColor* kann den Ports einer internen Schleife eine einheitliche Farbe gegeben werden, womit die graphischen Repräsentationen mehrerer interner Schleifen eines Blocks farblich unterschieden werden können. Das Attribut *calculationTimeout* bestimmt die maximal zulässige Berechnungszeit der internen Schleife. Wird diese Zeit bei der Berechnung überschritten, so wird die Berechnung des Blocks mit einem Fehler unterbrochen.

6.4.8 Parameterblock

Der Parameterblock dient als Quellblock für Konstanten, Steuerungsvariablen und fachmodellfremde Informationen innerhalb einer Metrik. In Abbildung 6.28 ist das Konzept für Parameterblöcke im Metamodell dargestellt.

Da der konkrete Typ des im Parameterblock zu speichernden Datums vom Anwendungsfall abhängt, wird die Information als String abgelegt. Die Implementierung des Parameterblock verfügt jedoch über eine Funktion, um anderweitige Daten für Speicher- und Leseoperationen entsprechend

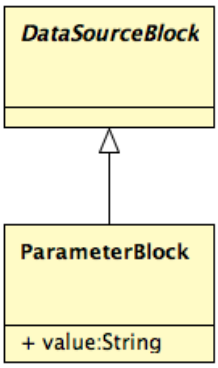


Abbildung 6.28: Konzept für Parameterblöcke im Metrik-Metamodell

zu konvertieren.

6.4.9 Berechnungsblock

Der Berechnungsblock ist, wie in Abbildung 6.29 dargestellt, ein Block einer Metrik, dem benutzerspezifischer Quellcode zugeordnet wird, der bei der Berechnung der Metrik ausgeführt wird. Dazu ist der Berechnungsblock (*CalculationBlock*) mit einem Quellcode (*SourceCode*) assoziiert. Da der Quellcode von beliebig vielen Berechnungsblöcken assoziiert werden kann, ist die Wiederverwendbarkeit der Quelltexte gegeben.

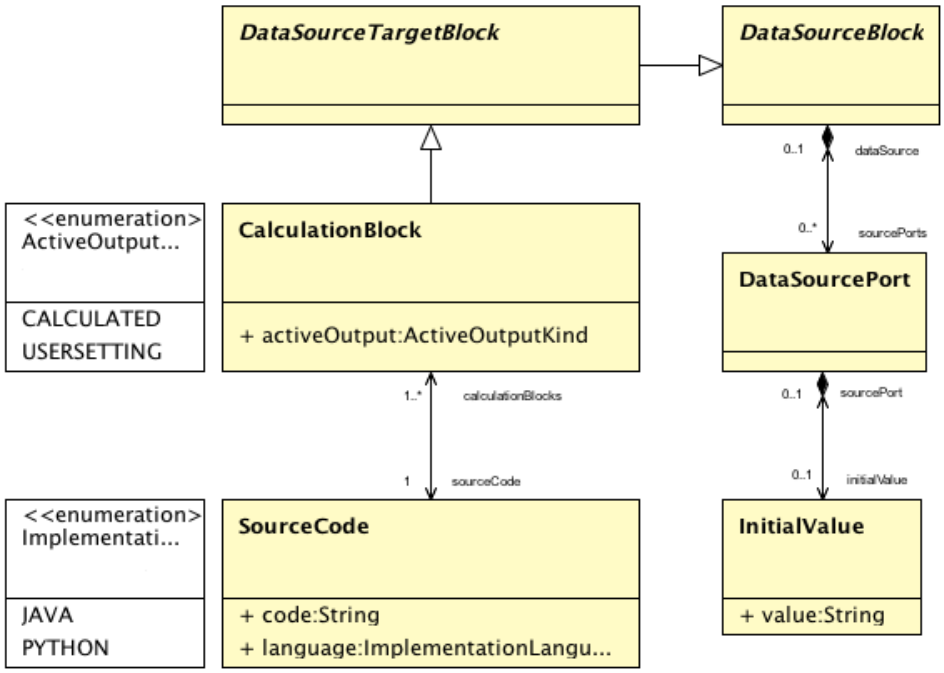


Abbildung 6.29: Konzept für Berechnungsblöcke im Metrik-Metamodell

Die Sprache des Quelltextes wird dem *SourceCode* als Attribute *language* hinterlegt. Dieses Attribut ist durch die Enumeration *ImplementationLanguage* typisiert, die zur Zeit die Literale *JAVA* und *PYTHON* für die unterstützten Sprachen beinhaltet. Die Enumeration kann bei Bedarf erweitert werden. Auf der Implementierungsseite steht eine entsprechende API zur Verfügung. Der Quellcode selbst wird im Attribut *code* als String abgelegt und ist somit Teil des Modells.

Der Berechnungsblock selbst verfügt über das Attribut *activeOutput*, das durch die Enumeration *ActiveOutputKind* typisiert ist. Dieses Konzept dient zur Umsetzung der in Abschnitt 6.2.5 beschriebenen „Was wäre wenn...“-Szenarien. Ist der Wert dieses Attributs auf *CALCULATED* gestellt, so werden die vom Berechnungsblock berechneten Ergebnisse an die Ausgangsports weitergegeben. Ist der Wert auf *USERSETTING* gestellt, so werden die vom Benutzer gesetzten Werte an die Ausgangsports geleitet. Diese Werte sind im Attribut *value* der am jeweiligen Datenquellport (*DataSourcePort*) angehängten Metaklasse *InitialValue* abgelegt.

6.4.10 Schleifenblock

Der Schleifenblock (*LoopBlock*) erbt von der abstrakten Metaklasse *AbstractCompositionBlock* (siehe Abbildung 6.30), welche im wesentlichen die Konzepte für interne und externe Ports zur Verfügung stellt, wie sie in Abschnitt 6.2.11 beschrieben sind.

Die internen Ports werden durch die Metaklassen *InternalSourcePort*, *InternalDataPort* und *InternalEndPort* beschrieben. Der Schleifenblock verfügt über genau einen internen Startport (*InternalSourcePort*), der die Eingangsdaten liefert, über die der Schleifenport iteriert. Zusätzliche, schleifeninvariente Daten werden dem Schleifenblock durch interne Datenports (*InternalDataPort*) zur Verfügung gestellt. Die Teilergebnisse der Schleifendurchläufe werden durch interne Endports (*InternalEndPort*) nach außen geleitet.

Die externen Ports werden durch Stellvertreter (Proxy) zu den entsprechenden internen Ports repräsentiert. Dazu gibt es die Metaklasse *DataTargetPortProxy* für interne Quellports (*AbstractInternalSourcePort*, eine Verallgemeinerung für *InternalSourcePort* und *InternalDataPort*). Interne Endports werden durch die Metaklasse *DataSourcePortProxy* nach außen vertreten. Die Stellvertreterklassen nehmen die gegensätzliche Semantik zu ihren Ursprungsports ein. Ein interner Quellport nimmt externe Daten entgegen, der entsprechende Proxy agiert extern als Datenzielport. Umgekehrt liefert ein interner Zielport Daten nach außen, so dass der entsprechende Proxy extern als Datenquellport agiert. Diese unterschiedlichen Charaktere werden durch die Vererbungen zu *DataSour-*

cePort und *DataTargetPort* sichtbar. Die Stellvertreterports referenzieren ihre Ursprungsports durch eine 1-zu-1 Assoziation.

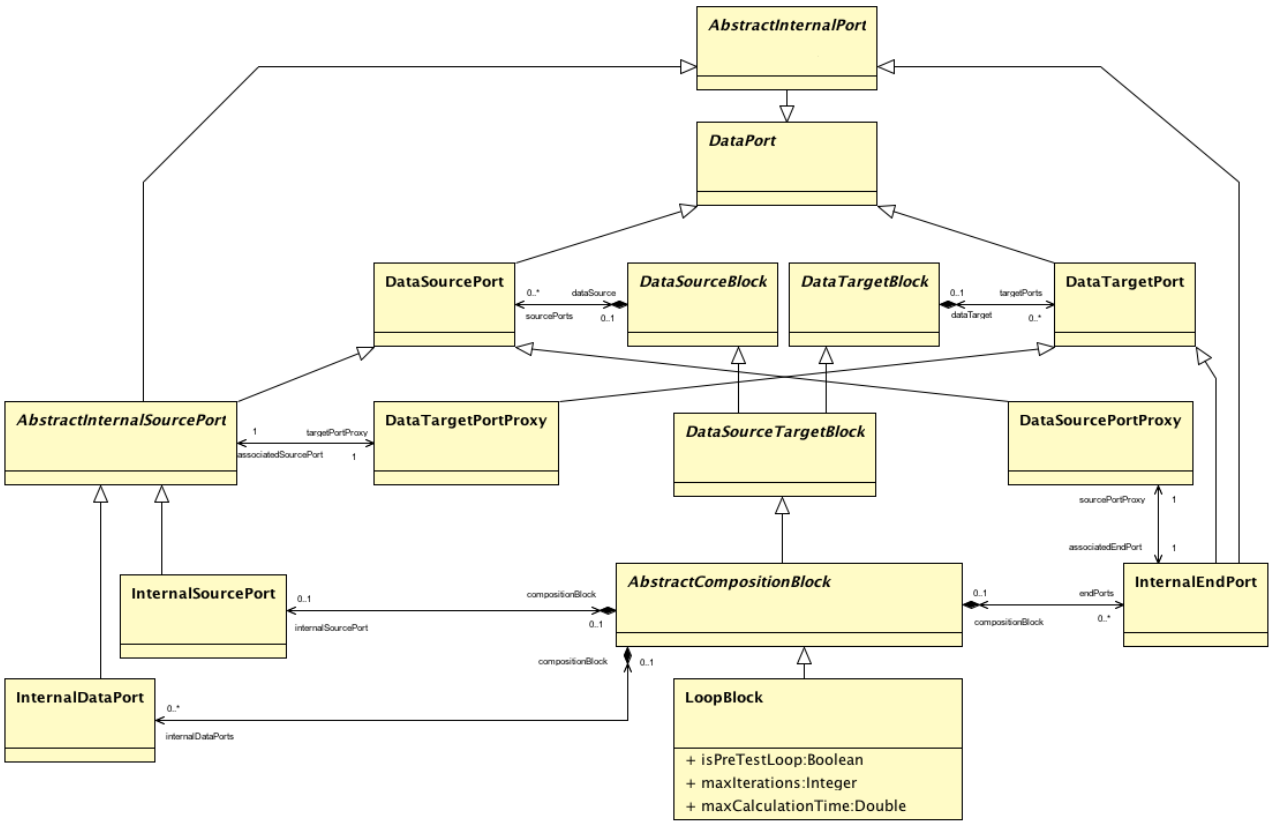


Abbildung 6.30: Konzept für Schleifenblöcke im Metrik-Metamodell

Der Schleifenblock (*LoopBlock*) selbst verfügt über drei Attribute. Das boolesche Attribut *isPreTestLoop* legt fest, ob die Schleifenbedingung am Anfang oder am Ende der Schleife ausgewertet wird. *MaxIterations* ist eine ganze Zahl und legt die maximale Anzahl der Iterationen fest. Das Double-Attribut *maxCalculationTime* legt die maximal zulässige Zeit für die Berechnung aller Schleifendurchläufe in Millisekunden fest. Ist diese Zeit erreicht, beendet der Schleifenblock mit einem entsprechenden Fehler.

6.4.11 Benutzerspezifischer Block

Der benutzerspezifische Block wird durch die Metaklasse *CustomizedBlock* repräsentiert (siehe Abbildung 6.31). Durch die Vererbung zu *DataSourceTargetBlock* kann dieser Block sowohl Datenquell- als auch Datenzielports und darüber hinaus Ports für interne Schleifen aufnehmen. Das von *IdentifieableArtefact* vererbte Attribut *id* speichert die Identität des implementierenden Eclipse-Plugins. Die Vererbung zu OperationRunnableArtefact lässt den benutzerspezifischen Block am

Konzept zur Ausführung von Modelloperationen, durch die Veränderungen des Modells ermöglicht werden, teilnehmen.

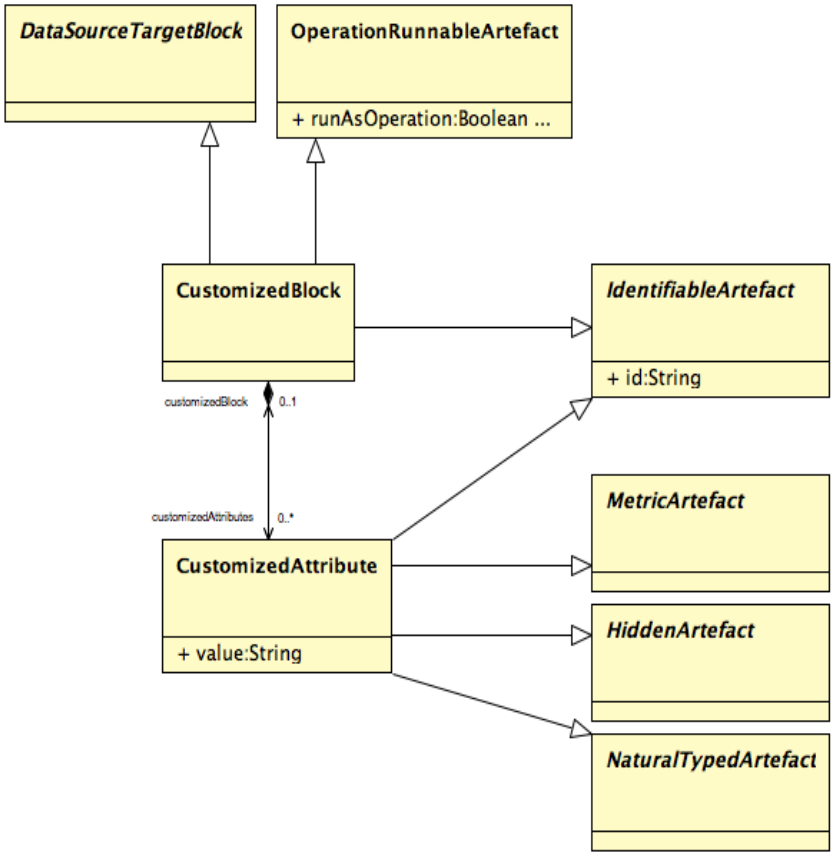


Abbildung 6.31: Konzept für benutzerspezifische Blöcke im Metrik-Metamodell

Spezifische Attribute des Blocks werden durch die Metaklasse *CustomizedAttribute* beschrieben. Diese verfügt über das Attribut value, um den Wert des spezifischen Attributs als Zeichenfolge abzulegen. Über die Vererbung zu *IdentifieableArtefact* verfügt *CustomizedAttribute* zudem über eine Identität, welche die Eindeutigkeit des spezifischen Attributs sichert. Die Vererbung zu *NaturalTypesArtefact* ermöglicht bei Bedarf eine (nicht im Diagramm dargestellte) Assoziation zu einem natürlichen Typen, der die Einheit des spezifischen Attributs repräsentiert.

6.4.12 Sensitivitätsanalyse

Für das in Abschnitt 6.2.15 beschriebene Konzept zur Sensitivitätsanalyse gibt es im Metrik-Metamodell eine entsprechende Repräsentation. Wie in Abbildung 6.32 dargestellt, gibt es eine Metaklasse *SensitivityArtefact*, die über eine Komposition zu einem *SensitivityDescriptor* besitzt. Dieser Deskriptor verfügt über alle Eigenschaften, die notwendig sind, um die Eckdaten für die Sensitivi-

tätsanalyse für das *SensitivityArtefact* durchzuführen.

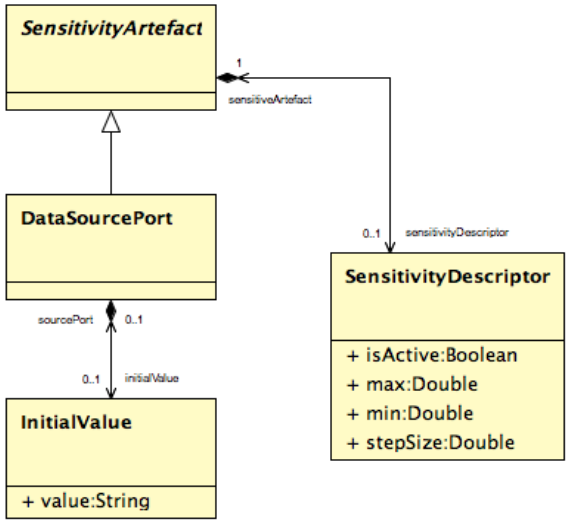


Abbildung 6.32: Sensitivitätsbeschreibung im Metrik-Metamodell

Das Attribut *isActive* beschreibt, ob der Sensitivitätsmodus für das Artefakt aktiviert ist. Mit den Attributen *min* und *max* wird das für die Sensitivitätsanalyse relevante Intervall festgelegt. Die *stepSize* bestimmt die Schrittlänge, mit der bei der Analyse gearbeitet werden soll.

Artefakte, für welche die Sensitivitätsanalyse zur Verfügung stehen sollen, erben von der abstrakten Metaklasse *SensitivityArtefact*. Dies ist beim Datenquellport (*DataSourcePort*) der Fall. Die darin enthaltene Metaklasse *InitialValue* legt den Referenzwert fest, auf den sich die Angaben im *SensitivityDescriptor* beziehen. Eine Ausnahme bildet der Berechnungsblock (*CalculationBlock*, siehe Abschnitt 6.4.9), für den je nach Modus der Referenzwert auch das berechnete Ergebnis sein kann.

6.4.13 Report-Dokumentationblock

Der Report- und Dokumentationsblock erlaubt, wie in Abschnitt 6.2.10 beschrieben, die Anbindung einer Metrik an einen Dokumentengenerator. Im Metamodell wird der Report- und Dokumentationsblock durch die Metaklasse *ReportDataSetResultBlock* repräsentiert. Durch die Vererbung zu *MetricResultBlock* gehört er, wie in Abbildung 6.33 gezeigt, zur Kategorie der Ergebnisblöcke.

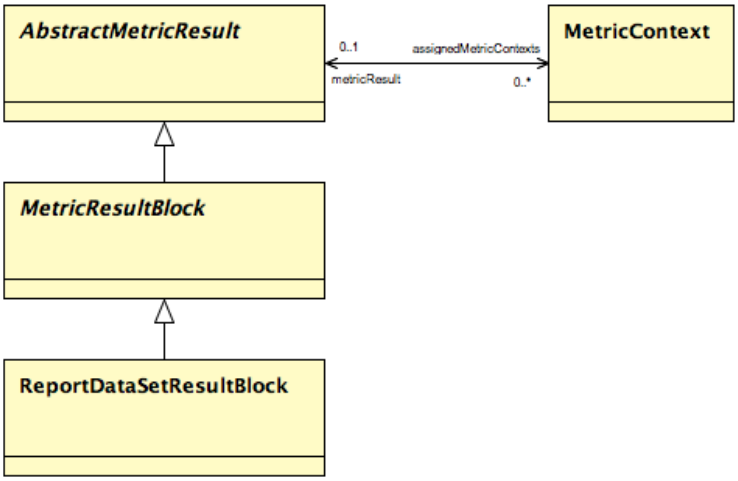


Abbildung 6.33: Report- und Dokumentationsblock im Metrik-Metamodell

Ergebnisblöcke verfügen über eine Assoziation zu einem Metrikkontext (MetricContext). Dabei handelt es sich um ein Stellvertreterartefakt zu dem Metrikergebnis, das vom Dokumentengenerator dazu verwendet wird, das assoziierte Metrikergebnis als Datenquelle in die Dokumentengenerierung einzubinden. Dieses Konzept steht allen Ergebnisblöcken, deren Metamodell-Konzepte in den folgenden Abschnitten beschrieben werden, zur Verfügung.

6.4.14 Diskreter Ergebnisblock

Der diskrete Ergebnisblock ermöglicht den Abgleich des eingehenden Ergebnisses mit einer Menge an gültigen Werten und die entsprechende Visualisierung im Metrikdiagramm. Im Metrik-Metamodell wird dieser Block, wie in Abbildung 6.34 gezeigt, durch die Metaklasse *EnumResultBlock* (der Name leitet sich aus seiner im folgenden näher erklärten Beziehung zu Enumerationen ab) repräsentiert.

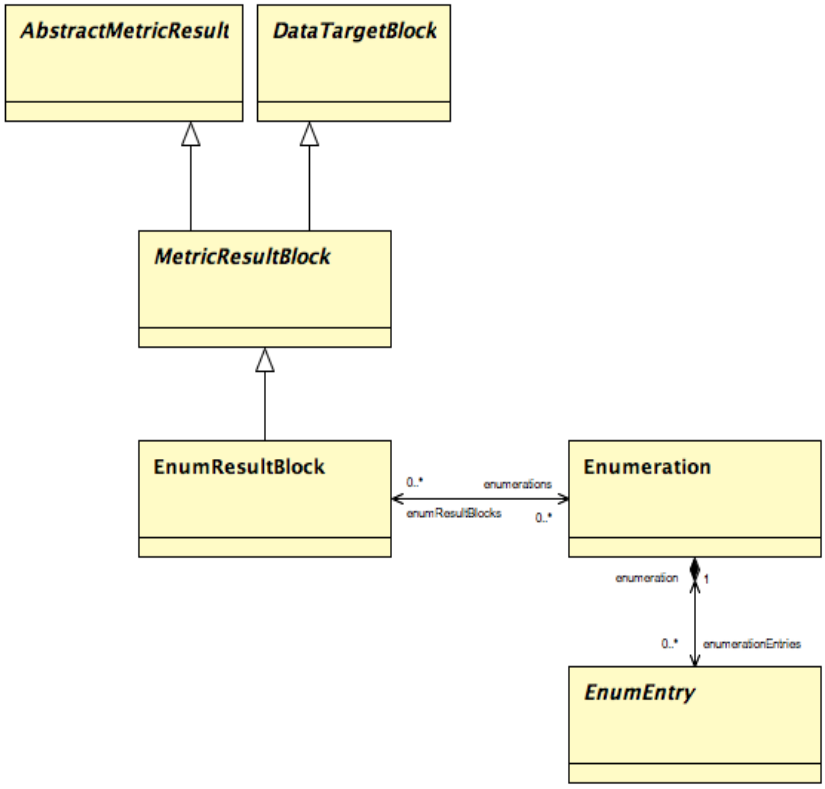


Abbildung 6.34: Diskreter Ergebnisblock im Metrik-Metamodell

EnumResultBlock erbt von der abstrakten Metaklasse MetricResultBlock und ist somit ein Metrikergebnis (*AbstractMetricResult*) sowie eine Zielblock für Datenflüsse (*DataTargetBlock*). Die Assoziation zu Enumerationen erlaubt die Spezifikation der erlaubten Ergebniswerte, die durch die Metaklasse EnumEntry repräsentiert werden.

6.4.15 Ampel- und Skalablock

Ampelblöcke ermöglichen die Visualisierung eines Ergebnisses als Ampel. Dazu steht im Metrik-Metamodell, wie in Abbildung 6.35 dargestellt, die Metaklasse *LightsBlock* zur Verfügung.

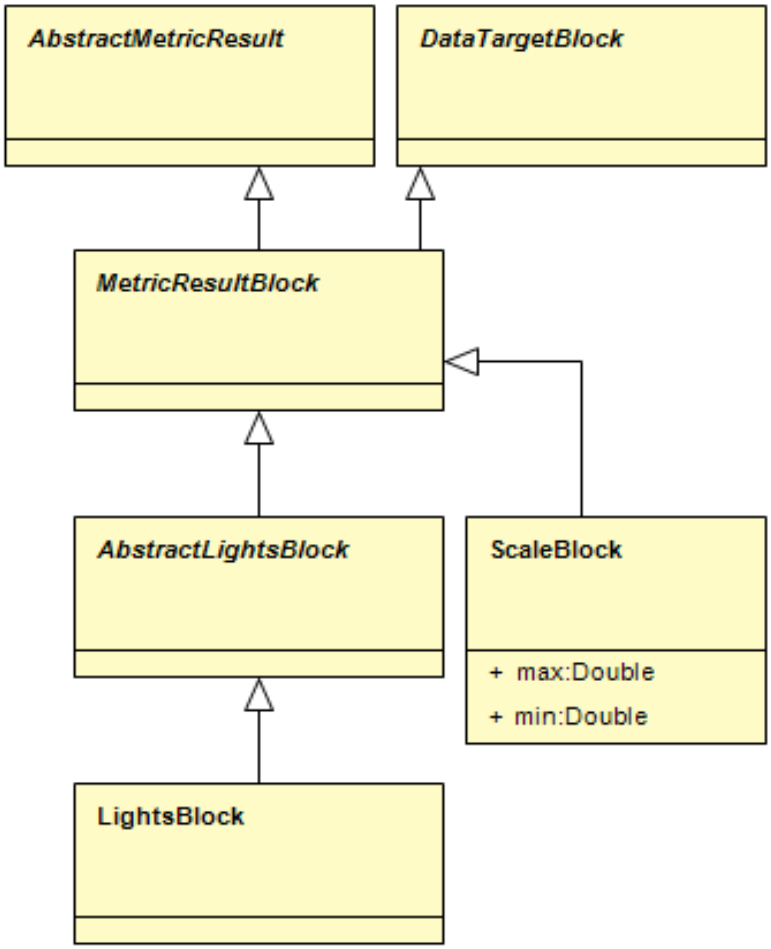


Abbildung 6.35: Ampel- und Skalablock im Metrik-Metamodell

Durch die Vererbung zu *MetricResultBlock* wird festgelegt, dass ein Ampelblock ein Metrikergebnis ist (*AbstractMetricResult*) und nur über Datenzielports (*DataTargetBlock*) verfügt. Neben den geerbten (und nicht dargestellten) Standardattributen wie Name und Beschreibung hat der Ampelblock keine weiteren Attribute.

Der Skalablock wird durch die Metaklasse *ScaleBlock* umgesetzt, die von *MetricResultBlock* erbt und somit analog zu den anderen Ergebnisblöcken nur über Datenzielports (*DataTargetBlock*) verfügt. Die Attribute *min* und *max* bilden die obere und untere Grenze des Skalenblocks ab.

6.4.16 Filter- und Sortierblock

Die Filter- und Sortierung von Ergebnissen und Zwischenergebnissen stellt einen wesentlichen Aspekt bei der Entwicklung von Metriken dar. Im Metrik-Metamodell gibt es dazu entsprechende Konzepte, die in Abbildung 6.36 dargestellt sind. Ausgangspunkt ist, dass Metrikergebnisse vor allem als Tabellen organisiert sind, wobei Listen und Vektoren als einspaltige Tabelle interpretiert

werden.

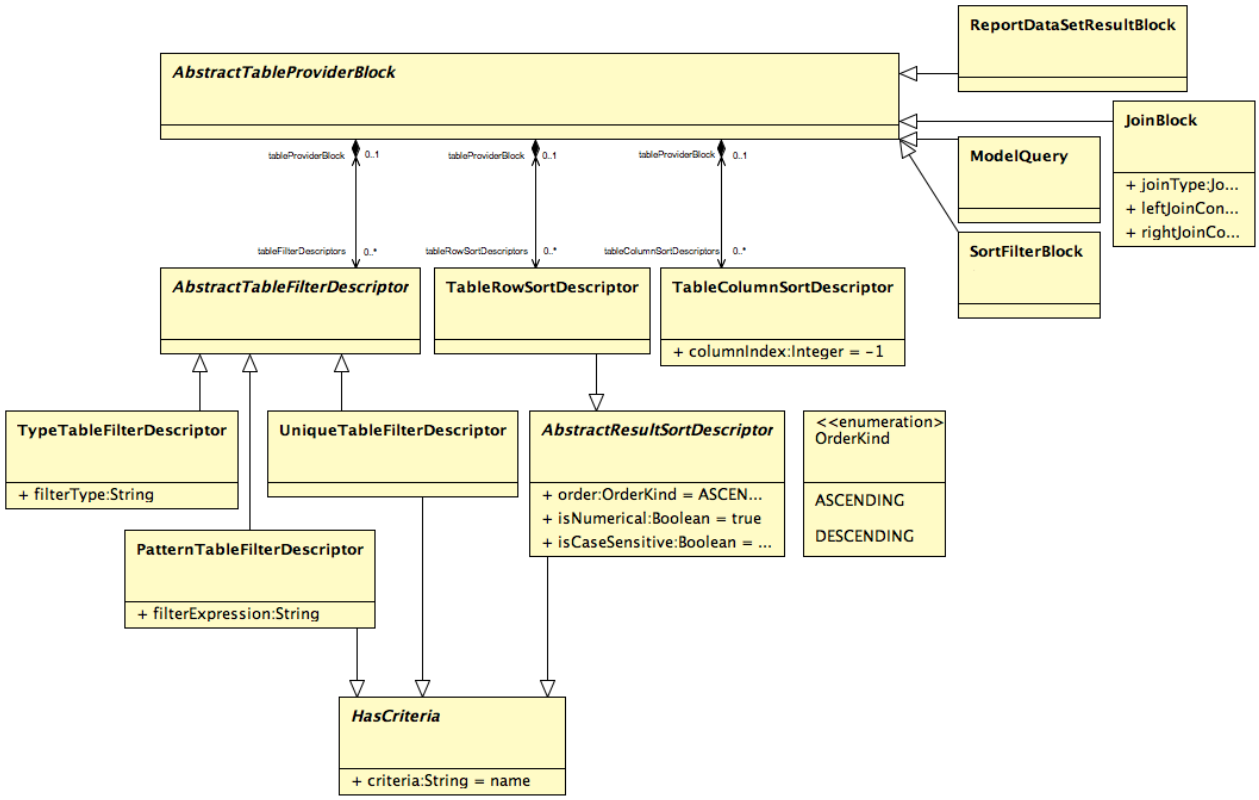


Abbildung 6.36: Filter- und Sortierung im Metrik-Metamodell

Dreh- und Angelpunkt ist die abstrakte Metaklasse *AbstractTableProviderBlock*, von der alle Metrikblöcke Erben, die Tabellen als Ergebnis zurückliefern können. Diese sind der *JoinBlock*, der Modellabfrageblock (*ModelQuery*) und der Filter- und Sortierblock (*SortFilterBlock*). Der Report- und Dokumentationsblock nimmt in diesem Konzept eine Sonderrolle ab, da er, wie in Abbildung 6.22 dargestellt, nur von *DataTargetBlock* erbt, so dass er in einer Metrik nicht als Datenquelle fungieren kann. Dieser Block stellt als Tabelle organisierte Daten für einen Dokumentengenerator zur Verfügung, die durch die Metrik gefiltert und sortiert werden können.

Die Filter- und Sortierungsbeschreibungen lassen sich in insgesamt drei Kategorien einteilen. Ein abstrakter Filterdeskriptor (*AbstractTableFilterDescriptor*) erlaubt die Filterung von Tabellen, wobei die Filterung entweder nach Typ (*TypeTableFilterDescriptor*), nach Mustern in Zeichenfolgen (*PatternTableFilterDescriptor*) oder nach Duplikaten (*UniqueTableFilterDescriptor*) erfolgt. Die Attribute filterType und filterExpression speichern die Informationen für den zu filternden Typ bzw. der zu filternde Ausdruck. Das Filterkriterium (z.B. der Name eines Artefakts) wird von mehreren Deskriptoren verwendet, so dass die Eigenschaft von der abstrakten Superklasse geerbt wird.

In der zweiten Kategorie befinden sich Deskriptoren zur Sortierung von Tabellenzeilen (*TableRowSortDescriptor*). Die geerbten Attribute legen die Sortierrichtung (*order*) fest, beschreiben, ob nummerisch sortiert werden soll (*isNummerical*) und ob die Groß- und Kleinschreibung eine Rolle spielen sollen (*isCaseSensitive*).

Die dritte Kategorie repräsentiert Deskriptoren zur Sortierung von Tabellenspalten (*TableColumnSortDescriptor*). Dabei weiß der Deskriptor über das Attribut columnIndex, welche Spalte er vertritt. Für jede Spalte, die im Ergebnis enthalten sein soll, wird ein entsprechender Deskriptor angelegt. Diese Eigenschaft macht den Sortierdeskriptor gleichzeitig zu einem Filter für Spalten. Die Spalten werden nach den jeweiligen Indizes vom *AbstractTableProviderBlock* aus sortiert.

6.4.17 Evaluations-Hubblock

Der Evaluations-Hubblock dient als Knotenblock, um einen über einen Eingangs-Datenfluss übertragenen Wert durch Evaluationsmetriken zu prüfen. Der Name Hubblock lehnt sich an Knotenpunkte wie USB-Hub aus der Computertechnik an. Dieser ermöglicht den Anschluss mehrerer USB-Endgeräte an einen USB-Host. Die Endgeräte sind als Evaluationsmetriken zu interpretieren, der Eingangs-Datenfluss führt zu einem Datenquellblock, der als Host zu interpretieren ist.

Die Anzahl der Evaluationsmetriken ist beliebig. Der Hubblock legt den angeschlossenen Evaluationsmetriken eine Information zur Überprüfung vor. Diese werten den übergebenen Wert aus und liefern ein bool'sches Ergebnis, welche von einem an den Hubblock angeschlossenen Validatorblock (siehe 6.4.18) durchgeführt wird.

In Abbildung 6.37 ist die Abbildung des Konzeptes für Evaluations-Hubblöcke dargestellt. Der Hubblock unterstützt die zwei Ausführungsmodi *parallel* und *serial*, der durch das Attribut *executionType* festgelegt wird.

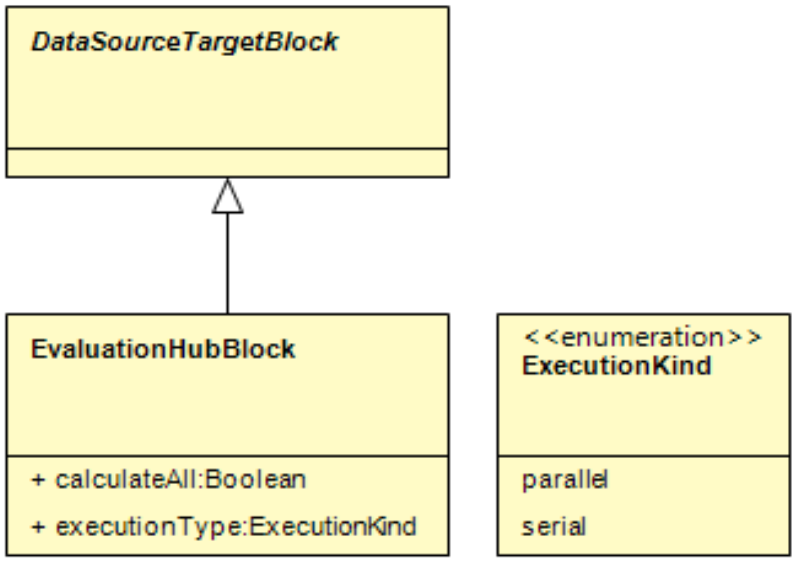


Abbildung 6.37: Evaluations-Hubblock im Metrik-Metamodell

Im parallelen Ausführungsmodus werden alle angeschlossenen Evaluationsmetriken zur parallelen Verarbeitung freigegeben (siehe 6.5.2), im seriellen Ausführungsmodus werden sie nacheinander zur Verarbeitung freigegeben. Die Ergebnisse fertig berechneter Evaluationsmetriken werden an den angeschlossenen Validatorblock übermittelt (siehe folgenden Abschnitt 6.4.18).

Im bool'schen Attribut *calculateAll* wird festgelegt, ob der Hubblock trotz negativer Ergebnisse einzelner Evaluationsmetriken alle angeschlossenen Evaluationsmetriken berechnen soll oder nicht.

6.4.18 Validatorblock

Der Validatorblock (Abbildung 6.38) wertet die Ergebnisse der Evaluationsmetriken des im voran gegangenen Kapitels 6.4.17 aus und errechnet daraus in Abhängigkeit des Attributes *validation* ein Resultat. Dieses Attribut kann die Werte der bool'schen Operatoren *AND*, *XOR* und *OR* annehmen.

Der Evaluations-Hubblock und der Validatorblock arbeiten zusammen und sind in der Lage, die Ergebnisse einer beliebigen Anzahl von Evaluationsmetriken zu kombinieren und daraus ein Gesamtresultat zu bestimmen. Benötigt werden diese Blöcke beispielsweise bei Optimierungsmetriken, um eine mögliche Lösung vorab auf Machbarkeit zu überprüfen.

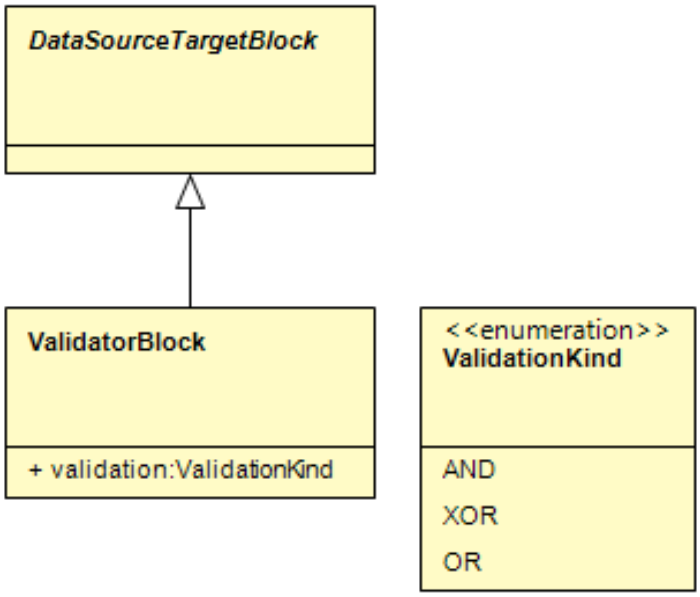


Abbildung 6.38: Validatorblock im Metrik-Metamodell

6.4.19 Metrikausführer

Wie in Abschnitt 6.2.14 beschrieben, stellt das Metrikausführer-Artefakt eine Möglichkeit dar, Metriken oder Teilmetriken aus anderen Kontexten heraus zu starten und dort zu verwenden.

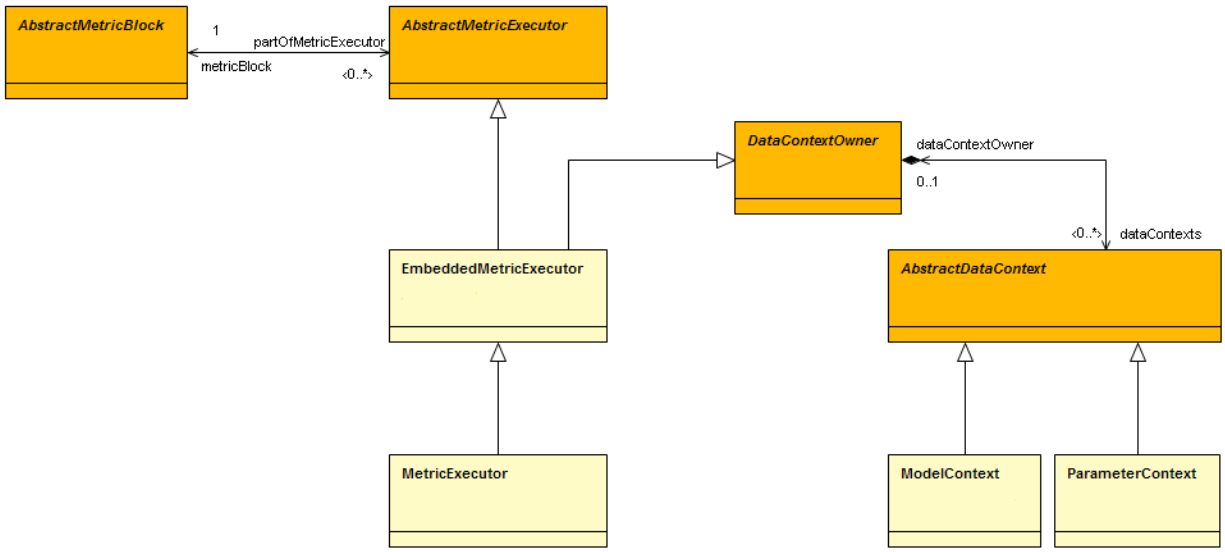


Abbildung 6.39: Metrikausführer im Metrik-Metamodell

Das Artefakt des Metrikausführers wird im Metrik-Metamodell, wie in Abbildung 6.39 dargestellt, durch die Metaklasse *MetricExecutor* repräsentiert. Über die von *AbstractMetricExecutor* vererbte Beziehung zu *AbstractMetricBlock* verfügt der Metrikausführer zu einer Referenz zu einem Metrik-

block (*AbstractMetricBlock*), welcher den Endpunkt der Metrik oder Teilmetrik, die es zu berechnen gilt, markiert.

Über die Vererbung zu *DataContextOwner* kann ein Metrikausführer zudem sogenannte Datenkontexte beinhalten. Diese können konkret Modell- oder Parameterkontexte sein (*ModelContext* bzw. *ParameterContext*). Über den Modellkontext werden die vom Metrik-Kontextblock (siehe Abschnitt 6.4.4) an die nachfolgenden Blöcke übergebenen Artefakte dynamisch (d.h. dies ist keine modellverändernde Maßnahme) überschrieben. Somit wird erreicht, dass die über einen Metrikausführer berechnete Metrik dynamisch an den aufrufenden Kontext angepasst und berechnet wird.

6.5 Ausführungsschicht für modellbasierte Metriken

Analog zum Metrik-Metamodell, welches unabhängig von Anwendungsdomänen entwickelt worden ist, ist auch die Implementierung der Ausführungsschicht unabhängig implementiert. Somit steht der Metrikeditor mit identischer Metamodell- und Codebasis in mehreren unterschiedlichen Anwendungen zur Verfügung.

6.5.1 Plug-in Struktur

Abbildung 6.40 zeigt die prinzipielle Plug-in Struktur der Implementierung als UML-Komponentendiagramm (siehe Kapitel 2.3.4), wobei ein Plug-in als Komponente repräsentiert wird. Das Plug-in *aquintos.mm.metricmm* beinhaltet das Metrik-Metamodell (Kapitel 6.4) und stellt somit als Instanz das Metrikmodell zur Verfügung. Davon abhängig ist das Plug-in *aquintos.mm.engine*, wo die komplette Semantik zur Ausführung von Metriken implementiert ist. Die blau eingefärbten Plug-ins links unterhalb beherbergen die jeweiligen Editoren und Ansichten für die graphische Darstellung und um Benutzerinteraktionen entgegen zu nehmen. Die grün hinterlegten Plug-ins rechts unten beinhalten die Implementierungen für die Java-Ausführungsschicht von Berechnungsblöcken und für die automatisierte Durchführung ausführbarer Einheiten (siehe 6.2.13.2).

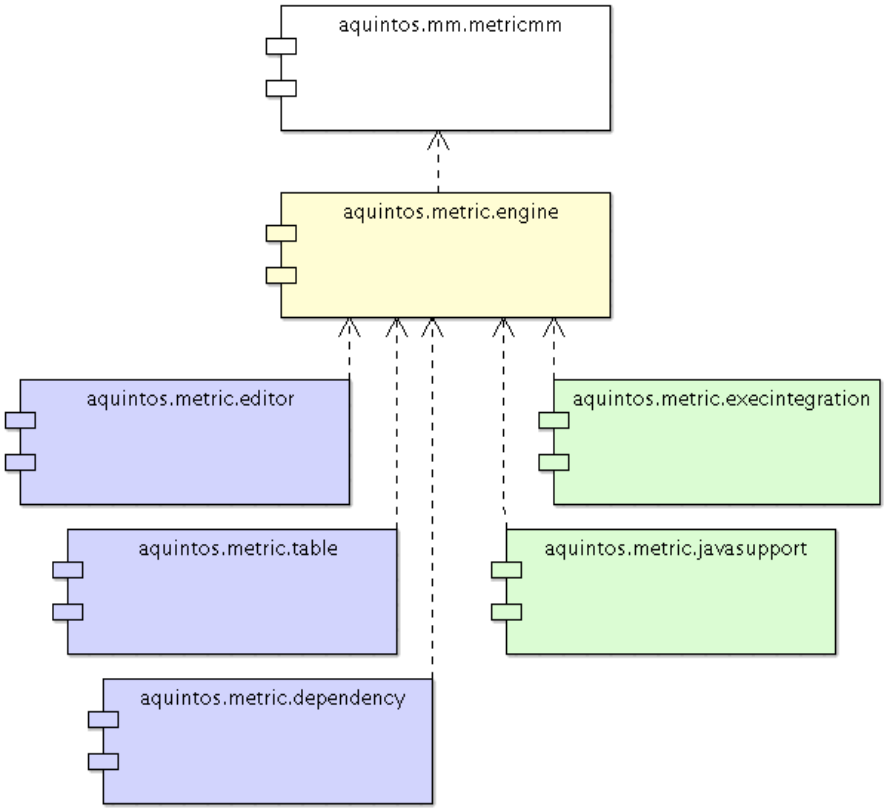


Abbildung 6.40: Plug-in Struktur des Metrik-Frameworks

6.5.2 Ausführungssemantik

Die Ausführungssemantik des Metrikframeworks teilt die Ausführung einer Metrik in zwei Schritte auf. Im ersten Schritt werden iterativ die Vorgängerblöcke (siehe Abschnitt 6.2.1) bestimmt. Die Bestimmung der Vorgängerblöcke wird in Abschnitt 6.5.2.2 detailliert erklärt. Die Vorgängerblöcke müssen berechnet werden, bevor die Ergebnisse der als Eingabe übergebenen zu berechnenden Ergebnisblöcke ermittelt werden können. Die gefundenen Vorgängerblöcke werden zusammen mit den Ergebnisblöcken in einer Menge M gesammelt und der Metrikengine übergeben, welche die eigentliche Berechnung durchführt. Dieser Sachverhalt wird im oberen Teil von Abbildung 6.41 dargestellt.

Abbildung 6.41: Ablauf der Berechnung einer Metrik

Im zweiten Schritt, dargestellt im unteren Teil von Abbildung 6.41, wird die aus dem ersten Schritt ermittelte Menge M an Metrikblöcken zur Berechnung auf die verfügbare Anzahl an parallel arbeitenden Berechnungskernen übergeben. Diese Anzahl ist standardmäßig die Anzahl der auf der verwendeten CPU verfügbaren Kerne, kann aber vom Benutzer auf einen kleineren Wert begrenzt werden. Zunächst wir in einem Berechnungskern der nächste berechenbare Block abgeholt und berechnet. Ein Block ist genau dann berechenbar, wenn er keine nicht berechneten Vorgängerblöcke mehr

hat. Gibt es keinen solchen Block mehr in der Menge M, so begibt sich der Kern in einen Idle-Zustand. Dieser wird verlassen um einen neuen zu berechnenden Block aus der Menge M zu holen, sobald ein anderer Kern die Berechnung eines Blocks beendet hat. Befinden sich alle Kerne gleichzeitig im Idle-Zustand, so bedeutet dies, dass es keine Blöcke mehr in der Menge M gibt, die berechnet werden könnten. Ist die Menge M leer, so wurden alle Blöcke ordnungsgemäß berechnet. Sind Blöcke in der Menge verblieben, so wird ein Fehlerzustand erreicht da nicht alle Blöcke berechnet werden konnten. Tritt während der Berechnung eines Blocks ein Fehler auf, so wird ebenfalls ein Fehlerzustand erreicht.

6.5.2.1 Berechnungszustände von Metrikblöcken

Metrikblöcke können unterschiedliche Zustände annehmen, je nachdem, ob sie noch nicht berechnet wurden, in Berechnung sind oder ein Berechnungsergebnis vorliegt. Der Zustandsautomat ist in Abbildung 6.42 dargestellt. Beim Erstellen eines Metrikblocks findet sofort der Übergang vom Initialzustand in den Zustand *Nicht berechnet* statt.

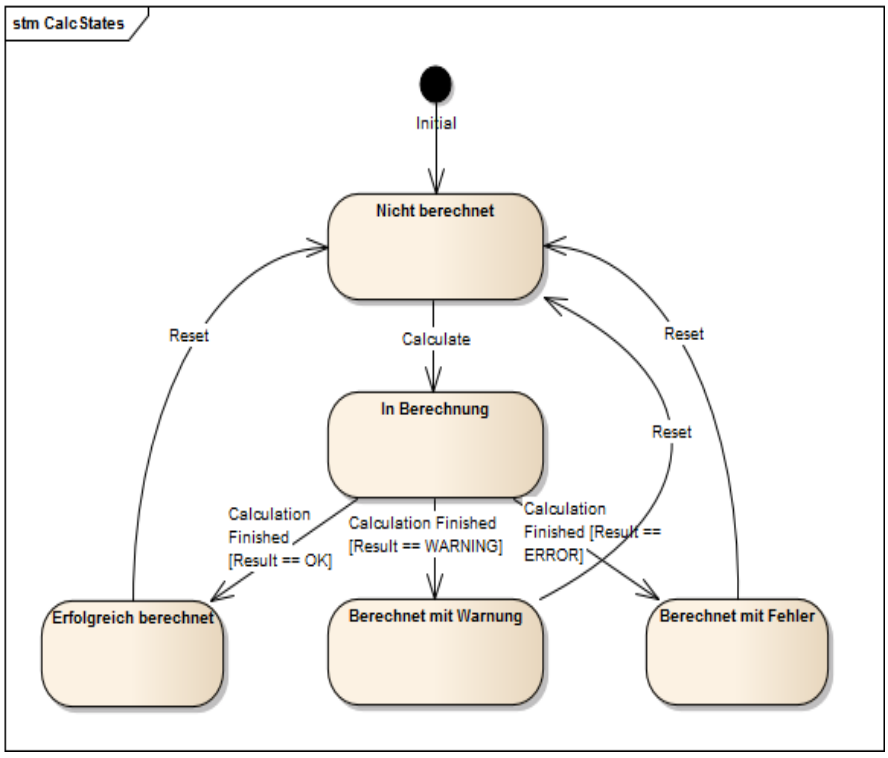


Abbildung 6.42: Zustandsautomat für Berechnungszustände für Metrikblöcke

Wird die Berechnung der Metrik angestoßen, so wechselt der Block in den Zustand *In Berechnung*. Wenn diese durchgeführt ist hängt der Folgezustand vom Berechnungsergebnis ab, wobei zwischen *Erfolgreich berechnet*, *Berechnet mit Warnung* und *Berechnet mit Fehler* unterschieden wird. Von

diesen Ergebniszuständen kann wiederum der Zustand *Nicht berechnet* erreicht werden.

Dieser Zustandsautomat verfügt über keinen expliziten Endzustand, da sich ein Metrikblock jederzeit in einem der angegebenen Zustände befindet.

Jedem der Zustände ist eine Farbe zugeordnet, die bei der Visualisierung des Blocks im Metrikdiagramm verwendet wird (siehe Abschnitt 6.2.1). Es ergibt sich folgende Farbzuordnung:

- Nicht berechnet: Grau
- In Berechnung: Blau
- Erfolgreich berechnet: Grün
- Berechnet mit Warnung: Orange
- Berechnet mit Fehler: Rot

6.5.2.2 Bestimmung der Vorgängerblöcke

Zur Bestimmung der Vorgängerblöcke eines Blocks kommen Modellabfragen (siehe Abschnitt 2.4.2.1) zum Einsatz, die lokal auf den zu untersuchenden Block angewendet werden.

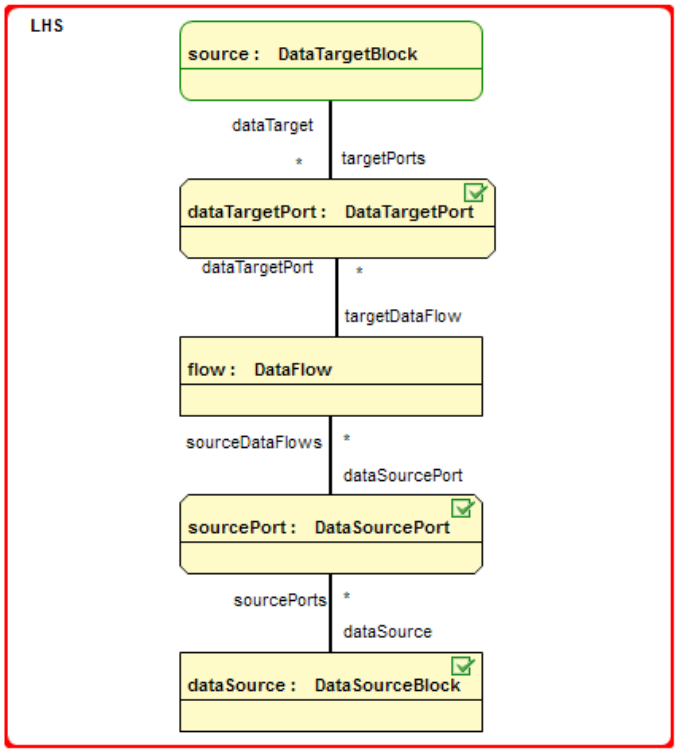


Abbildung 6.43: Beispielregel zur Bestimmung des Vorgängerblocks

Im Gegensatz zu den Modellabfragen, wie sie als Block innerhalb einer Metrik verwendet werden (siehe Kapitel 6.2.3), sind diese Abfragen in einem separaten Regelmodell abgelegt und so vor Modifizierung geschützt. Die in Abbildung 6.43 gezeigte Regel dient zur Bestimmung des Vorgängerblocks *datasource* sowie der beteiligten Ports *sourcePort* und *dataTargetPort*, ausgehend von einem dem Datenzielport *source*. Insgesamt umfasst dieses Regelmodell 23 Abfrageregeln, um alle unterschiedlichen Möglichkeiten abzudecken.

6.5.2.3 Zyklen innerhalb einer Metrik

In Metriken können zwei Typen von Zyklen auftreten. Der erste Typ ist ein Zyklus durch den Einsatz von internen Schleifen. Wie in Abschnitt 6.4.7 beschrieben, werden auf diese Weise interne Berechnungen eines Blocks nach außen zu anderen Blöcken delegiert um so die Wiederverwendbarkeit von Metriken zu erhöhen.

Der zweite Typ eines Zyklus entsteht durch eine Verschaltung von Metrikblöcken, bei denen Datenflüsse über Quell- und Zielports Blöcke derart miteinander verbinden, dass ein Zyklus entsteht. Die Anzahl der am Zyklus beteiligten Blöcke ist unerheblich.

6.5.3 Datentransport zwischen Metrikblöcken

Wie bereits in Abschnitt 6.2.1 beschrieben, erfolgt die formale Modellierung für Transportwege auszutauschender Daten durch Datenflüsse. Diese sind über Ports mit den Metrikblöcken verbunden. Im Folgenden wird darauf eingegangen, wie der Transport der Daten über die modellierten Datenflüsse während der Ausführungszeit realisiert ist.

Zunächst gilt, dass Datenflüsse beliebige Objekte transportieren können. Diese sind, wie in Abbildung 6.44 zu sehen, in einem speziellen Statusobjekt eingepackt, welches von dem Vorgängerblock erzeugt wird und den Nachfolgeblöcken zur Verfügung gestellt wird. Das allgemeine Statuskonzept wird von dem Eclipse-Framework zur Verfügung gestellt. Die daran beteiligten Klassen und Interfaces sind beige dargestellt. Ein Status verfügt über einen Schweregrad, einer Mitteilung und ggf. über ein allgemeines Ausnahmeobjekt (Throwable). Zusätzlich gibt es das Interface IMultiStatus, welches eine Hierarchisierung von Statusobjekten ermöglicht. Metrikergebnisse sind solche MultiStatus-Objekte und implementieren zusätzlich das Interface IMetricResultStatus. Damit verfügt es über eine Komposition zu einem allgemeinen Objekt, welches das Ergebnis repräsentiert.

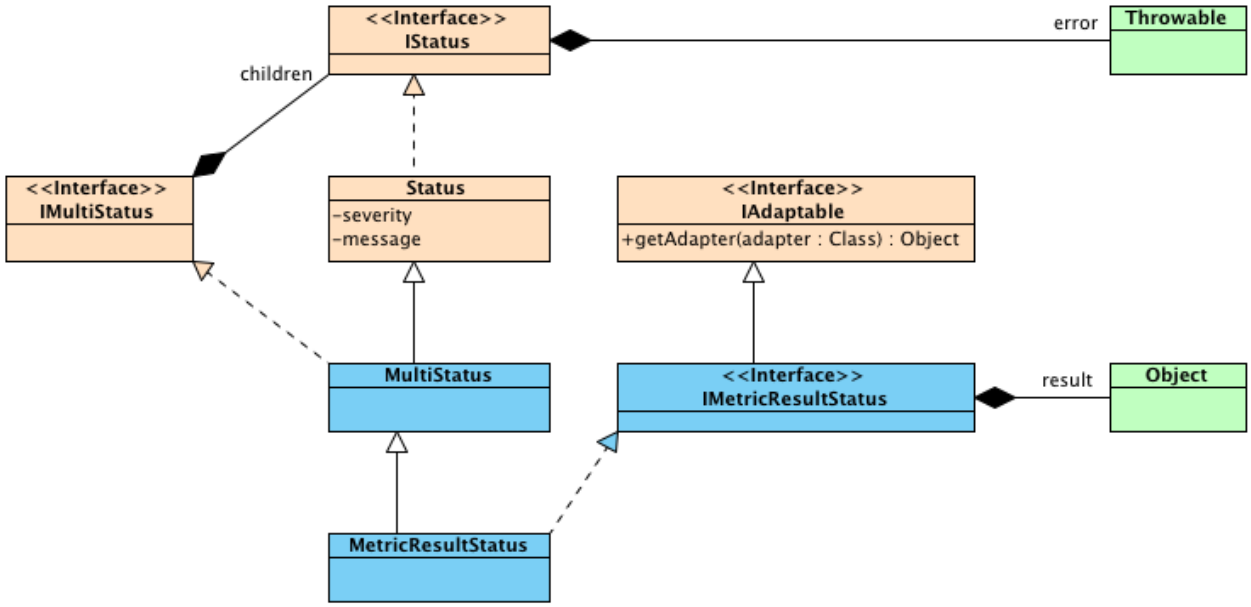


Abbildung 6.44: Implementierungskonzept MetricResultStatus

Weder Datenfluss, noch die beteiligten Ports verfügen über die Information, wie die übertragenen Objekte typisiert sind. Diese Informationen sind erst bekannt, wenn der Datenquellblock berechnet worden ist. Demnach gibt es auch keine Möglichkeit, zum Modellierungszeitpunkt einer Metrik den Benutzer darauf aufmerksam zu machen, dass zwei miteinander verbundenen Blöcke nicht zueinander passen, d.h. das Datenformat der auf den Datenfluss gelegten Information des Quellblocks kann vom verbundenen Zielblock nicht interpretiert bzw. verarbeitet werden. Der Fehler würde erst zum Berechnungszeitpunkt der Metrik bemerkt werden.

Es zeigt sich jedoch, dass die Anzahl der unterschiedlichen Datentypen, die über Datenflüsse transportiert werden, einen begrenzten Umfang haben. Zudem unterstützen die Ausleseroutinen der Eingangsparameter der Metrikblöcke oft mehrere Datentypen, die bei Bedarf in ein einheitliches blockinternes Format umgewandelt werden. Ein Beispiel hierfür ist der Ampelblock, der in Abschnitt 6.2.7 näher beschrieben ist. Das Format der Eingangsdaten darf skalar oder mehrdimensional sein, das konkrete Zahlenformat (z.B. Float, Double oder Integer) ist unerheblich. In den folgenden Abschnitten werden gängige Datenformate für Informationsübertragung über Datenflüsse beschrieben sowie Beispiele dafür gegeben.

6.5.3.1 Austausch von primitiven Datentypen

Zu den primitiven Datentypen gehören die standardmäßigen Zahlenformate wie Integer, Float, Double, Long, etc. und Zeichenketten (String). In der eingesetzten Programmiersprache Java können

diese Datentypen komfortabel ineinander überführt werden, wenn auch nicht immer verlustfrei (z.B. von Double nach Integer.

In der Regel erwarten Metrikergebnisblöcke wie der Skalablock oder der Ampelblock Zahlenwerte als Eingangsparameter. Diese Werte können dabei in komplexeren Strukturen, wie sie im Abschnitt 6.5.3.4 beschrieben sind, eingebettet sein.

6.5.3.2 Austausch von Modellierungsartefakten

Über Datenflüsse können beliebige Modellierungsartefakte oder typisierte Untermengen (wie zum Beispiel alle Steuergeräte vom Typ ECU (siehe Kapitel 3.3.5)) transportiert werden. Oft werden mehrere Artefakte auf einmal übertragen. Sie sind dann in komplexen Datenstrukturen, wie sie im Abschnitt 6.5.3.4 beschrieben sind, organisiert. Als Beispiel hierfür ist der Modellabfrageblock anzuführen. Diese Datenquelle liefert je nach Anwendungsfall Listen von Modellierungsartefakten oder Tabellen (beschrieben im folgenden Abschnitt 6.5.3.3) bestehend aus Modellierungsartefakten zum angeschlossenen Datenfluss.

6.5.3.3 Austausch von tabellarisch organisierten Informationen

In einigen Anwendungsfällen ist es notwendig, primitive Datentypen (siehe 6.5.3.1) bzw. Modellierungsartefakte (siehe 6.5.3.2) oder eine Kombination daraus tabellarisch zu übertragen. Dabei ist der Struktur der Tabelle eine Semantik hinterlegt.

Ein einfaches Beispiel hierfür ist eine Tabelle bestehend aus zwei Spalten, wobei in der ersten Spalte Leitungen (Modellierungsartefakte) und in der zweiten Spalte die Längen der Leitungen (als primitiver Datentyp Double), berechnet aus einer entsprechenden Leitungssatz-Bewertungsmetrik, hinterlegt sind.

Da Java hier keine geeigneten Bordmittel zur Verfügung stellt, wurde eine Standardimplementierung entworfen. In Abbildung 6.45 ist ein UML-Klassendiagramm zu sehen, welches die Interfaces der Implementierung zeigt.

Dreh- und Angelpunkt ist das Interface *ITempTable*. Dieses kann über beliebig viele Spalten (*ITempTableColumn*) und Zeilen (*ITempTableRow*) verfügen. Tabelle, Spalten und Zeilen können über Namen verfügen. Die Zellen der Tabelle sind durch *ITempTableCell* abgebildet, die jeweils einer Spalte und einer Zeile zugeordnet sind. Damit ist deren Position innerhalb der Tabelle eindeutig. Die Zelle ist ein Stellvertreterobjekt für den eigentlichen Inhalt der Zelle. Dies kann beispielsweise

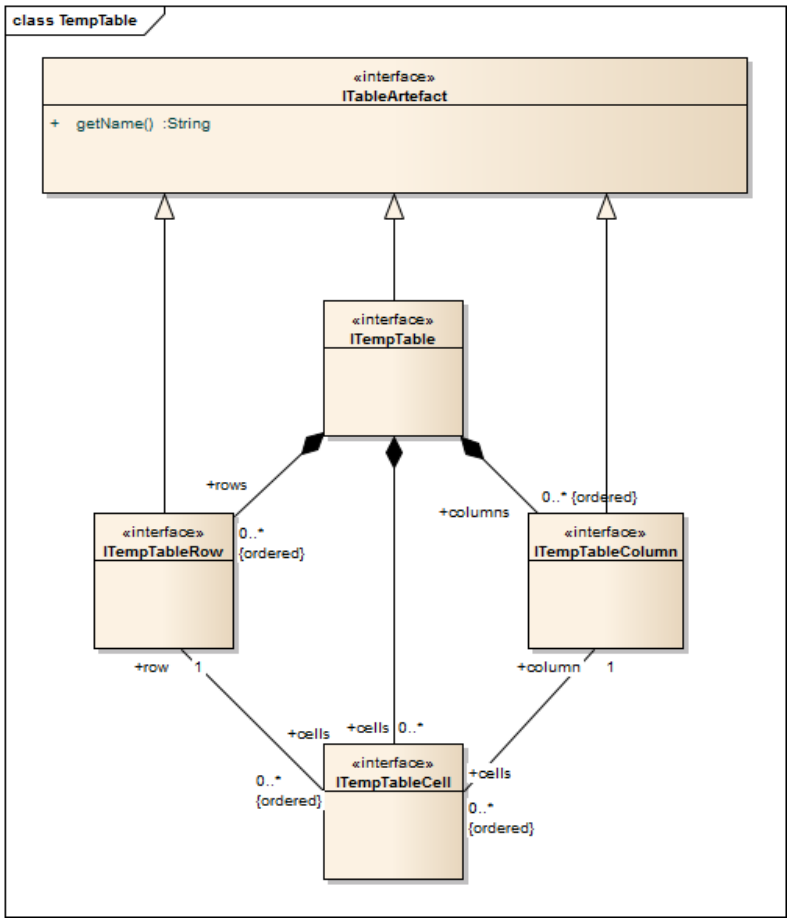


Abbildung 6.45: ITempTable: tabellarisch organisierte Informationen

ein Modellierungsartefakt oder ein primitives Datenobjekt sein.

6.5.3.4 Komplexe Strukturen

Häufig werden nicht einzelne Artefakte von Metrikblöcken übertragen. Stattdessen werden viele Artefakte in komplexe Datenstrukturen zusammengefasst und gebündelt auf den Ausgangsport gelegt. Zu diesen komplexen Datenstrukturen gehören beispielsweise Listen [JAVA6List], Mengen [JAVA6Collection], Zuordnungen [JAVA6Map] und Felder (Array). Für diese Datenstrukturen stehen auch von Java zur Verfügung gestellte Sortier- und Filterkriterien zur Verfügung. Diese werden durch den Sortier- und Filterblock (siehe 6.2.6) verwendet.

In einem Metrikkontextblock (siehe 6.2.2) werden beispielsweise beliebige Modellierungsartefakte vom Benutzer zusammengestellt. Bei der Berechnung der Metrik werden diese Artefakte dann in eine Liste verpackt und auf den Ausgangsport gelegt.

Dem Metrikentwickler steht es frei, eigene Programmbibliotheken in eine Metrik zu integrieren und dort vorhandene Datenstrukturen zu verwenden. Hierbei empfiehlt es sich, dem MetricResultStatus

Adapter zu registrieren, um das spezifische Ergebnis in allgemein unterstützte Datenformate umzuwandeln. Anderenfalls können diese Ergebnisdaten nur von spezifischen Metrikblöcken (in Form von Berechnungsblöcken (siehe 6.2.5) oder benutzerspezifischen Blöcken (siehe 6.2.13)) verarbeitet werden.

6.5.4 Metrik-Debugging

In diesem Abschnitt wird zunächst auf allgemeine Debugging-Methoden eingegangen. Basierend darauf werden Debugging-Strategien und prototypische Implementierungen vorgestellt, welche zum Debugging von modellbasierten Metriken generell geeignet sind.

6.5.4.1 Debugging-Methoden

Unter Debugging versteht man eine Methode zur Fehlerlokalisation und -analyse. Für die unterschiedlichen Anwendungen und Werkzeuge gibt es entsprechende Debugging-Verfahren. Dabei unterscheidet man mehrere hierarchische Analysestrategien [ZEL03] Experimentieren, Induktion, Beobachtung und Deduktion. Diese führen mit den wachsenden Anforderungen zu einer umfassenden Debuggingstrategie [GRA09]. Die gängigste Strategie ist das Experimentieren, wobei ein Programmablauf angehalten wird und Entwickler in die Lage versetzt werden, relevante Programminformationen interaktiv abzufragen. Diese Strategie lässt sich auch auf das Debugging von modellbasierten Metriken applizieren (siehe auch WEI10).

Die Ausführung von Metriken wird an definierten Punkten, den sogenannten Haltepunkten (Breakpoint) angehalten, um dort den Ausführungszustand der Metrik, z.B. zur Fehlersuche, zu analysieren. An dieser Stelle sind zwei unterschiedliche Arten des Debuggings möglich. Das modellbasierte Metrik-Debugging unterstützt des Debugging des Metrikablaufs im Metrikmodell und wird im folgenden Abschnitt 6.5.4.2 detailliert beschrieben. Das Quellcodebasierte Metrikdebugging unterstützt das Debugging des ausführbaren Quellcodes innerhalb eines Berechnungblocks und ist das Analogon zu bekannten Debug-Funktionen gängiger Software-Entwicklungsumgebungen. Es wird näher in Abschnitt 6.5.4.3 beschrieben.

6.5.4.2 Modellbasiertes Metrik-Debugging

Modellbasiertes Metrik-Debugging ermöglicht die Analyse des Metrikablaufs und Datenflusses. Dazu werden Haltepunkte auf Modellartefakte gesetzt. Die Ausführung der Metrik wird angehalten, wenn bei der Metrikausführung ein Haltepunkt erreicht wird oder ein mit einem Haltepunkt anno-

tiertes Artefakt an der Metrikausführung beteiligt ist.

Haltepunkt	**Artefakt**	**Beschreibung**
Vor Berechnung	Alle Metrikblöcke außer Schleifen- und Berechnungsblöcke	Dieser Haltepunkt stoppt die Ausführung einer Metrik bevor der markierte Metrikblock ausgeführt wird.
Nach Berechnung	Alle Metrikblöcke außer Schleifen- und Berechnungsblöcke	Dieser Haltepunkt stoppt die Ausführung einer Metrik nachdem der markierte Metrikblock ausgeführt wurde.
Auf Artefakt	Artefakt	Dieser Haltepunkt stoppt die Ausführung der Metrik immer dann, wenn das markierte Artefakt das Ergebnis oder Teil des Ergebnisses eines Metrikblocks ist, das über einen Datenquellport auf einen Datenfluss innerhalb der ausgeführten Metrik gelegt wird.
Auf Typ	Typ-Artefakt	Dieser Haltepunkt stoppt die Ausführung der Metrik immer dann, wenn das markierte Typ-Artefakt der Typ eines Artefaktes ist, welches das Ergebnis oder Teil des Ergebnisses eines Metrikblocks ist, das über einen Datenquellport auf einen Datenfluss innerhalb der ausgeführten Metrik gelegt wird.
Artefakt oder Typ auf Datenquellport	Datenquellport	Dieser Haltepunkt stoppt die Ausführung der Metrik analog zu den Haltepunkten „Auf Artefakt“ und „Auf Typ“ mit dem Unterschied, dass ein konkret angegebener Datenquellport betroffen sein muss.
Auf Schleifendurchlauf	Metrikschleife	Dieser Haltepunkt stoppt die Ausführung der Metrik, wenn die angegebene Iteration des Scheifenblocks erreicht wurde.

Tabelle 4: Arten von Haltepunkten für Modellbasiertes Metrik-Debugging

In Tabelle 4 sind die möglichen Arten von Haltepunkten aufgeführt. Neben der Analyse des reinen Metrikablaufs können Bedingungen an den Haltepunkten annotiert werden, so dass diese Haltepunkte nur dann aktiv werden, wenn die Bedingung erfüllt ist.

Obwohl Metriken parallel berechnet werden können (siehe Kapitel 6.5.2), werden im Debug-Modus Metriken sequenziell berechnet. Dadurch ist eine eindeutige Berechnungsreihenfolge gegeben, die bei Bedarf (sofern durch die Metrikstruktur realisierbar) während des Debuggings modifiziert werden kann. Durch die in Kapitel 6.5.2 beschriebene Ausführungssemantik ist gewährleistet, dass sich die Metrikergebnisse im parallelen und im seriellen Modus bei identischen Anfangsbedingungen nicht unterscheiden.

6.5.4.3 Quellcodebasiertes Metrik-Debugging

Ist in der Algorithmik eines Berechnungsblocks ein Fehler enthalten, so führen die Methoden des modellbasierten Metrikdebuggings nicht zum Erfolg. Hierzu wird das quellcodebasierte Metrik-Debugging verwendet, welches analog zu gängigen integrierten Software-Entwicklungsumgebungen (IDE) funktioniert.

Wie in Abschnitt 2.7.2 beschrieben, funktioniert üblicherweise das Java-Debugging verteilt über zwei Applikationen. Das zu debuggende Programm wird durch die debuggende Applikation gesteuert. Beide laufen in unterschiedlichen virtuellen Maschinen.

Die Ausführung von Metriken ist in PREEvision, der Modellierungssoftware für E/E-Architekturen, integriert. Folglich soll das Debugging ebenfalls in diese Software integriert sein. Dies bedeutet, dass sowohl die debuggende Metrik als auch der Metrik-Debugger innerhalb eines Programms in derselben virtuellen Java-Maschine laufen.

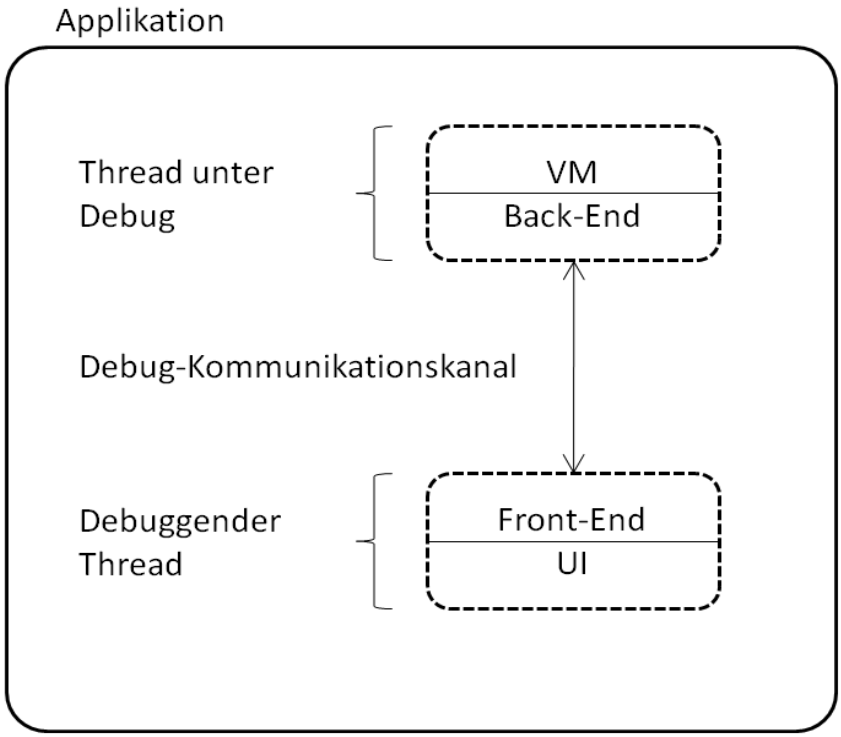


Abbildung 6.46: Debugging innerhalb einer Applikation

Dies wird erreicht, indem sowohl der debuggende als auch der gedebuggte Applikationsteil thread-basiert in einer Applikation eingebettet werden und über den Java-Debugging-Mechanismus verbunden werden [WEI10]. Dies ist in Abbildung 6.46 dargestellt.

Das Debugging selbst unterstützt Funktionen wie Haltepunkte, diverse Schrittoperationen sowie das

Auslesen und Setzen von Programmvariablen.

6.6 Modellierung und Interpretation von Metriken

In diesem Abschnitt wird darauf eingegangen, wie Metriken auf Basis dieser Arbeit modelliert und interpretiert werden. Zunächst wird die Hierarchisierung und Partitionierung von Metriken erläutert. Aufbauend darauf wird auf die Modellierung einfacher Bewertungsmetriken, z.B. die Erfassung von Kosten oder Gewichten eingegangen, gefolgt von der Modellierung komplexer Metriken mit Iterationen sowie den Aufbau von Quellcode-Bibliotheken.

6.6.1 Hierarchisierung und Partitionierung

Die Hierarchisierung von Metriken erfolgt nach der in Kapitel 6.4.1 beschriebenen Kompositionshierarchie, das dazugehörige Klassendiagramm des Metamodells ist in Abbildung 6.21 gezeigt.

Metriken liegen, organisiert in einem *Metrikmodell*, in einem Teilbaum unterhalb der Verwaltung *Administration* (Abbildung 6.47, linker Teil). Das Metrikmodell (Metaklasse *MetricModel*) wird über Metrikpakete (Metaklasse *MetricPackage*) weiter aufgeteilt. Die einzelnen Metrikartefakte sind in den Metrikpaketen einsortiert (Abbildung 6.47, rechter Teil).

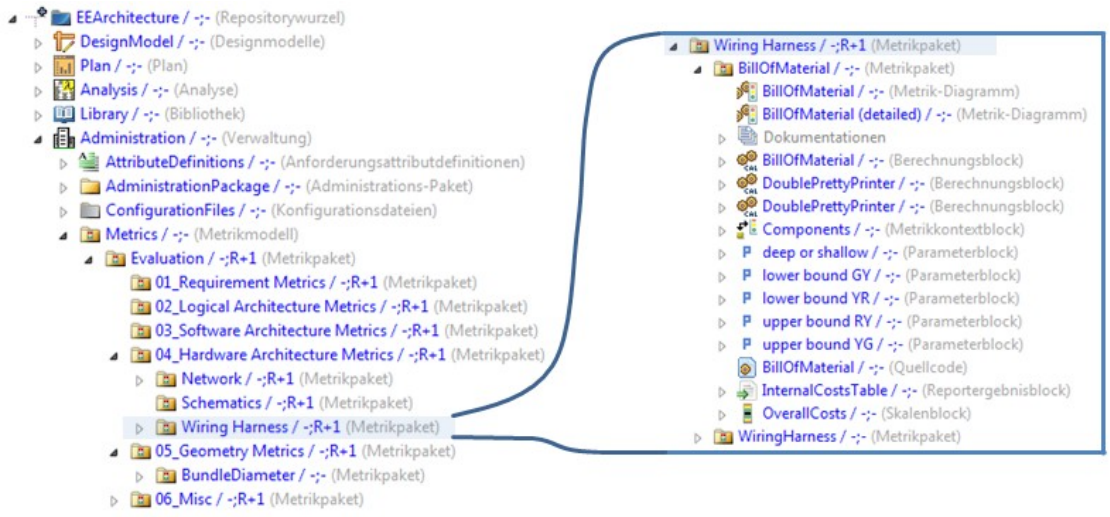


Abbildung 6.47: Hierarchisierung von Metriken

Im gezeigten Beispiel wird das Metrikmodell durch Metrikpakete strukturiert, welche sich am PREEvision-Ebenenmodell orientieren. Die spezifischen Metriken (z.B. *BillOfMaterial*) werden wiederum durch ein Metrikpaket gekapselt. Darin sind alle zur Metrik gehörenden Artefakte angeordnet.

Metrikübergreifend verwendete Artefakte werden idealerweise in allgemeinen Metrikpaketen (z.B. *06_Misc*) abgelegt. Diese Ordnungsstruktur ist jedoch beliebig und wird nicht vorgegeben. Für die Berechnung einer Metrik ist die Hierarchisierung nicht relevant, da kommt es lediglich auf die Modellierung der Metrikblöcke an.

7 Bewertung und Vergleich von Architekturmodellen

7.1 Allgemeine Bewertungsmetriken

7.1.1 Zählmetriken

Zählmetriken dienen u.a. zur Erstellung von Stücklisten (vgl. Abschnitt 4.2.1). Die in Abbildung 7.1 vorgestellte Zählmetrik gibt einen groben Überblick über die aktuelle Architektur, indem Summen für die relevanten Bauteile gebildet werden und in eine Tabelle exportiert werden. Bei den betrachteten Bauteilen handelt es sich um Steuergeräte (ECU), Sensoren, Aktuatoren, Sicherungen, Bussysteme, leitungssatzseitige Pins und komponentenseitige Pins.

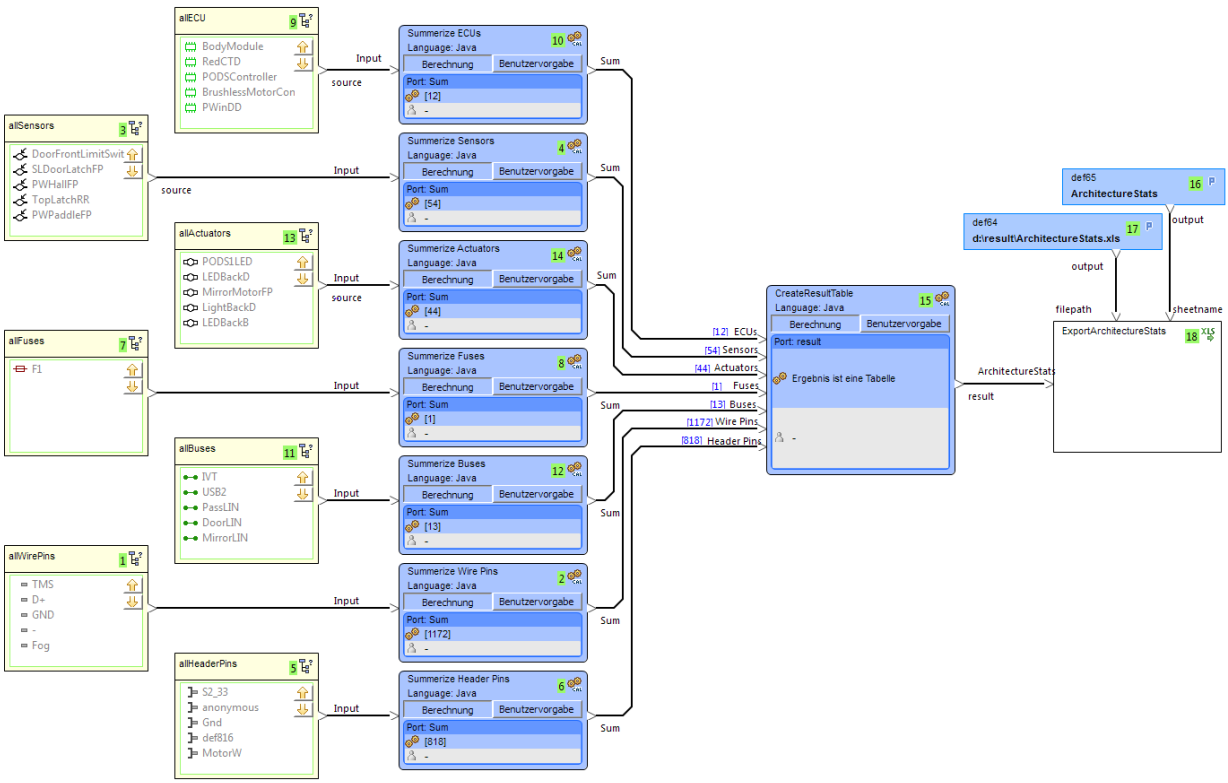


Abbildung 7.1: Zählmetrik zur Erfassung relevanter Summen

Im linken Teil der Metrik liefern Modellabfragen die für die Metrik relevanten Artefakte zurück. Die hier verwendeten Modellabfragen ähneln sich alle, so dass exemplarisch nur die Modellabfrage *allECU* in Abbildung 7.2 gezeigt wird. Es wird nach allen Artefakten vom Typ *ECU* gesucht, die Teil der aktiven Variante sind.

In den folgenden *Summerize*-Berechnungsblöcken werden die Summen der jeweiligen Modellab-

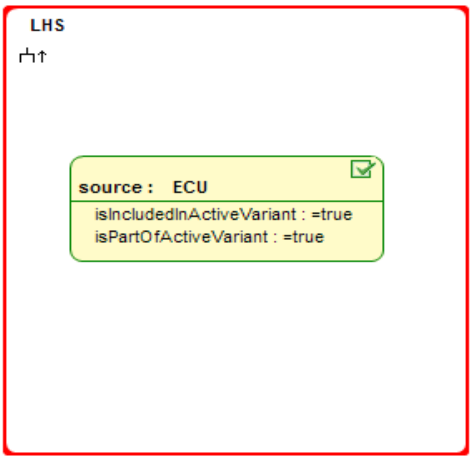


Abbildung 7.2: Modellabfrage-Regel allECU

fragen gebildet. Zwar sind in der Metrik insgesamt sieben dieser Berechnungsblöcke vorhanden, jedoch verweisen sie alle auf dasselbe Quellcode-Artefakt. Dieses Konzept im Metamodell wird in Abschnitt 6.4.9 näher beschrieben.

Die Summen der Artefakttypen werden an einen weiteren Berechnungsblock übergeben, welcher die Ergebnistabelle aufbereitet und an den Excel-Exportblock weiterleitet.

ECUs	Actuators	Sensors	Fuses	Busses	Header Pins	Wire Pins
12	44	54	1	13	818	1172

Tabelle 5: Exportiertes Metrikergebnis

In Tabelle 5 ist das Metrikergebnis zu sehen. Die Formatierung wurden nachträglich eingefügt. Die dargestellten Zahlen beziehen sich auf das Elypsis-Modell [ELY11] ohne eingeschalteter Variante, d.h. die Werte beziehen sich auf alle Artefakte im Modell.

7.1.2 Materialkosten

Ziel der Metrik ist die Bestimmung der Materialkosten gemäß Abschnitt 4.2.2 in einem angegebenen Teilbaum (alternativ dazu im gesamten Modell). Ferner soll die Kostenart (geschätzt / berechnet / verhandelt) berücksichtigt werden und das Ergebnis als Tabelle exportiert werden.

Die Metrik ist in Abbildung 7.3 dargestellt. Der Modellabfrageblock *CostEvaluation* (siehe Abschnitt 6.2.3) ermittelt alle Artefakte, welche über eine Kostenbeschreibung verfügen. Die Abfrageregel ist in Abbildung 7.4 dargestellt. Diese Regel beruft sich auf das Kostenkonzept im EEA-ADL Metamodell, wie es in Abschnitt 3.2.2.4 im Rahmen des Deskriptor-Konzeptes beschrieben ist.

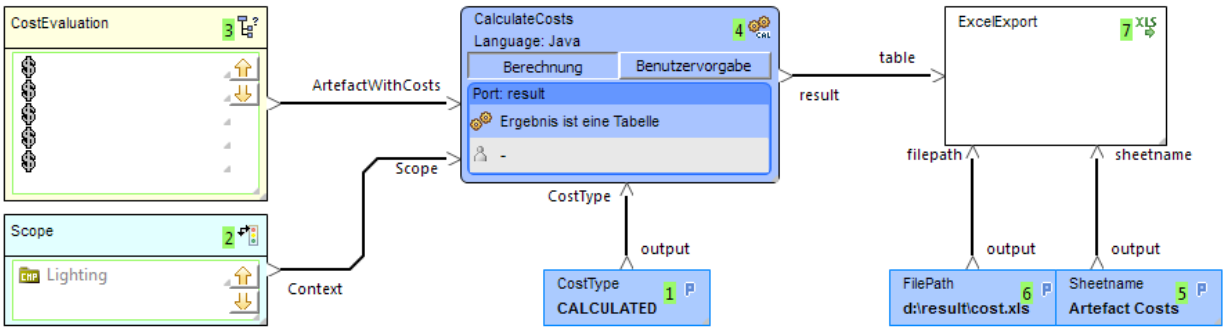


Abbildung 7.3: Metrik zur Berechnung der Materialkosten

Der zu betrachtende Teilbaum wird durch den Metrik-Kontextblock *Scope* (siehe 6.2.2) spezifiziert, im dargestellten Beispiel wird die Modellierung der Fahrzeugbeleuchtung beachtet. Ist in diesem Block kein Kontext angegeben, so wird das komplette Architekturmodell betrachtet.

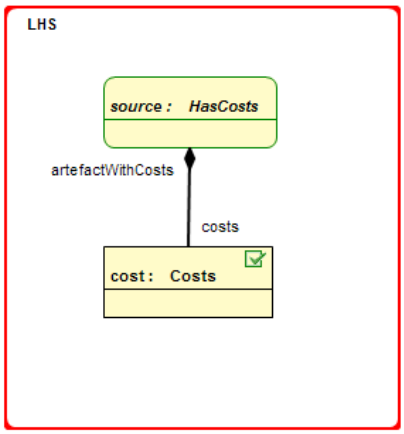


Abbildung 7.4: Modelabfrage-Regel CostEvaluation

Der folgende Berechnungsblock *CalculateCosts* wird die eigentliche Berechnung der Kosten vorgenommen. Dazu werden alle Artefakte mit Kosten, welche in der aktiven Variante enthalten und im selektierten Teilbaum enthalten sind, zur Berechnung herangezogen. Der zu verwendende Kostentyp wird durch den Parameterblock *CostType* angegeben. Als Ergebnis liefert der Berechnungsblock eine Tabelle mit den auszugebenden Rohdaten. Die Implementierung in Java umfasst 105 Zeilen Quellcode, aufgeteilt in drei Methoden. Darin enthalten sind bereits das Lesen und Setzen der Ports, notwendige Konvertierungen, Fehlerbehandlung, import-Statements sowie Klassen- und Instanzvariablendeklarationen.

Den Abschluss der Metrik bildet der Exportblock nach Excel (siehe 6.2.13.1). Diesem Block werden die darzustellenden Daten vom Berechnungsblock als Tabelle (siehe 6.5.3.3).

Artefact	Cost
Total Cost	52
XenonUnitFR	20
XenonUnitFL	20
LEDBackA	3
LEDBackB	3
LEDBackD	3
LEDBackC	3

Tabelle 6: Exportiertes Metrik-Ergebnis

Das für das dargestellte Beispiel mit der aktiven Variante des Elypsis-Modells [ELY11] *Advanced Lights* generierte Excel-Dokument ist in Tabelle 6 dargestellt, Formatierung und Sortierung sind nachträglich angepasst. Die im Modell eingetragenen Kostenwerte sind fiktiv.

7.1.3 Bewertung mechanischer Eigenschaften

7.1.3.1 Gewichte

Aufgrund ähnlicher Konzepte zur Beschreibung von Kosten und Gewichten im EEA-ADL-Metamodell (siehe Abschnitt 3.2.2.4), ist die Metrik zur Berechnung der Gewichte nach 4.2.3 sehr ähnlich zur vorgestellten Metrik zur Berechnung der Materialkosten in Abschnitt 7.1.2.

Deshalb berechnet die in Abbildung 7.5 gezeigte Metrik die Gewichte aller vorhandenen Varianten und Realisierungsalternativen und legt das Ergebnis tabellarisch in einer Gegenüberstellung ab.

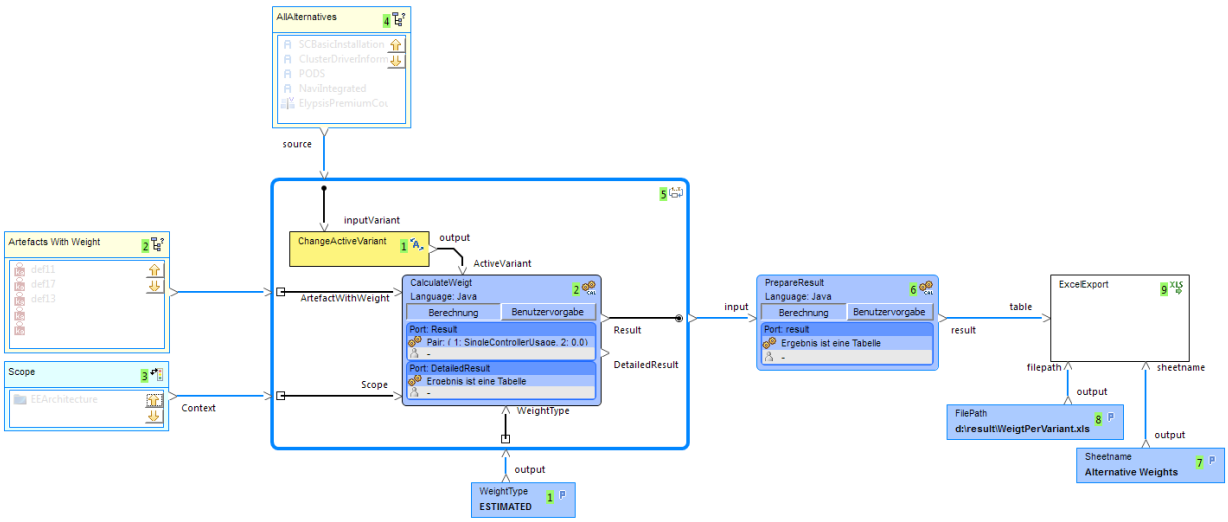


Abbildung 7.5: Metrik zur Berechnung der geschätzten Kosten in allen vorhandenen Varianten

Die Blöcke *Artefact With Weight*, *Scope*, *WeightType*, *FilePath* und *Sheetname* sind weitgehend aus

der Materialkosten-Metrik übernommen und sind ggf. an das leicht unterschiedliche Metamodell-Konzept zur Modellierung der Gewichte angepasst.

Der Berechnungsblock *CalculateWeight* ist seinem Analogon in der Materialkosten-Metrik ebenfalls sehr ähnlich. Es wurde lediglich ein zusätzlicher Eingang *ActiveVariant* implementiert. Ferner wurde die Ausgabe derart erweitert, dass zwei Arten von Ergebnissen ausgegeben werden. Während der Ausgang *DetailedResult* weitgehend dem Ergebnis der Kostenmetrik entspricht, beinhaltet der Ausgang *Result* nur noch das Gesamtgewicht für die jeweils aktive Variante. Der besseren Übersichtlichkeit halber wird das detaillierte Ergebnis in dieser nicht weiter betrachtet.

Der Modellabfrage-Block *AllAlternatives* ermittelt alle im Modell vorhandenen Architekturalternativen. Dies können Varianten oder Realisierungsalternativen sein. Die zugehörige Abfrageregel ist in Abbildung 7.6 dargestellt.

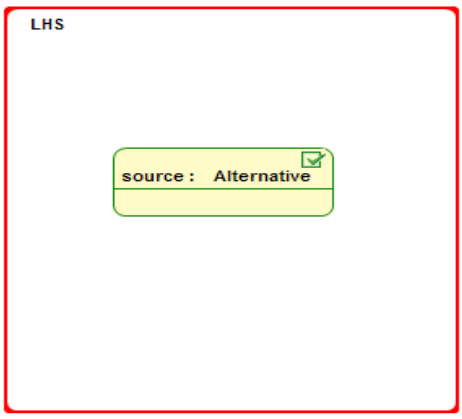


Abbildung 7.6: Modellabfrage-Regel AllAlternatives

Das Ergebnis dieses Abfrageblocks befüllt den Eingangsport des Schleifenblocks (siehe 6.2.11). Dieser Schleifenblock iteriert über alle vorhandenen Varianten und setzt diese über den Block *ChangeActiveVariant* als aktive Variante. Die übrigen Eingangsports der Schleife sind iterationsinvariant und werden durchgeführt.

Der Berechnungsblock *CalculateWeight* berechnet für jede aktive Variante das Gesamtgewicht und gibt es als Ergebnis weiter. Nachdem alle Iterationen berechnet sind, verpackt der Schleifenblock die Ergebnisse aller Durchläufe und gibt sie an den nachfolgenden Berechnungsblock *PrepareResult* weiter. Dort wird das Ergebnis für den Excel-Exportblock aufbereitet und weitergegeben. Varianten ohne gesetzte Gewichte werden dabei aussortiert.

Active Variant	Total Weight
ElypsisPremiumCoupeA	0,5
ElypsisBasicCoupeB	0,5
KeylessEntry	0,02
CentralLocking	0,02
DCHardBasic	0,48
DCHardPremium	0,48
DCSmartPremium	0,48
ElypsisBasicCoupeA	0,02
ElypsisPremiumCoupeB	0,5

Tabelle 7: Exportiertes Metrik-Ergebnis

Tabelle 7 zeigt das Ergebnis der Metrik. Die Formatierungen wurden nachträglich bearbeitet. Analog zur Kostenmetrik beziehen sich die dargestellten Werte auf das Elypsis-Modell [ELY11], die konkreten Werte sind fiktiv.

Die Implementierungen der Berechnungsblöcke betragen für *CalculateWeight* 117 Zeilen Quellcode (gegenüber 105 Zeilen für die Materialkosten) und für *PrepareResult* 39 Zeilen.

7.1.3.2 Längen

Die Berechnung von Kabel- und Leitungslängen unterliegt vielen Besonderheiten. Es müssen Splices, Trennstellen, Aufgabelungen von Kabelsträngen, etc. unterstützt werden. Des Weiteren müssen spezifische Eigenschaften der unterschiedlichen Führungen (Tape, Kabelkanal, Clips) beachtet werden. Zudem haben Ausstattungs- und Fahrzeugvarianten eine starke Auswirkung auf den Leitungssatz, da oft nicht benötigte Leitungen weggelassen werden.

Die in Abbildung 7.7 gezeigte Metrik berechnet wichtige Leitungssatzkriterien wie Längen, Kosten, Gewicht und Anzahl der geschnittenen Leitungen nach 4.2.3. Optional dazu kann die Metrik mit Varianten umgehen, d.h. es wird beachtet, ob eine Leitung und deren Verlegeweg zur aktiven Variante gehören oder nicht. Dies erfolgt über den Parameterblock *pathesVMSensitive*. Der Kontextblock *Wiring Harness (es)* beinhaltet die Leitungssatz-Pakete, deren Inhalte in der Metrik betrachtet werden sollen. Die eigentliche Berechnung erfolgt über den Berechnungsblock *WiringHarness*. Dieser Block verfügt über vier Ausgangsport, welche die Ergebnisse der Metrik ausgeben und an Ergebnisaufbereitungsblöcke wie Ampeln und Report-Ergebnisblock weitergegeben werden. Sowohl die Werte der Datengrundlage (und damit die Ergebnisse der Metrik) als auch die Werte für die Schranken der Ampelblöcke beziehen sich auf das Elypsis-Modell [ELY11] und sind fiktiv.

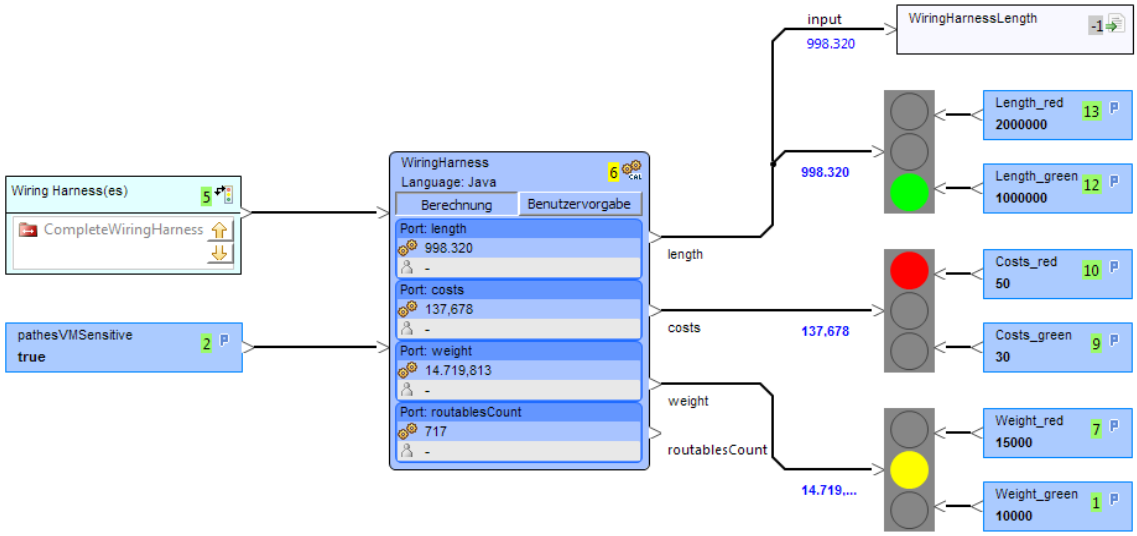


Abbildung 7.7: Metrik zur Berechnung wichtiger Leitungssatzkriterien

7.1.3.3 Befüllungsgrade von Segmenten

Ein wichtiges Kriterium bei der Auslegung von Leitungssätzen in Fahrzeugen ist der Befüllungsgrad von Topologiesegmenten. Die Frage ist, ob ein Leitungs- und Kabelstrang des Leitungssatzes in das dafür vorgesehene Topologiesegment passt. Diese Fragestellung lässt sich zum Kreis-Paketierungsproblem zuordnen und ist NP-Schwer [OLI10].

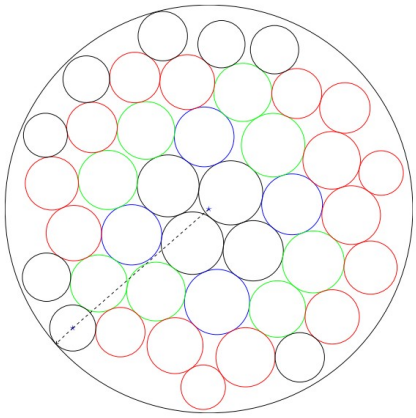


Abbildung 7.8: Beispiel: Das Kreis-Paketierungsproblem [OLI10]

In Abbildung 7.8 wird dieses Problem visualisiert. Die inneren Kreise mit unterschiedlichen Radien repräsentieren Leitungen und Kabel. Der äußere Kreis ist der kleinstmögliche Kreis, in welchen alle inneren Kreise hineinpassen. Somit sagt sein Radius etwas darüber aus, wie groß die engste Stelle des Topologiesegmentes sein muss, damit alle Leitungen und Kabel des Leitungssatzes an dieser Stelle hindurchpassen. Bedingt durch Fertigungstoleranzen und Montageanforderungen muss der

minimale Radius des Segmentes in der Realität größer sein, was sich durch einen Korrekturfaktor bewerkstelligen lässt.

Während der Konzeptions- und Entwicklungszeit des Fahrzeug ist daher der exakte Befüllungsgrad des Segmentes noch nicht erforderlich, eine gute Annäherung ist zu diesem Zeitpunkt ausreichend. Diesen Umstand macht sich die hier vorgestellte Metrik zur Ermittlung des Befüllungsgrades zunutze, da exakte Algorithmen für dieses Problem mitunter sehr lange Laufzeiten haben können. Die Metrik entstand in einer im Rahmen dieser Arbeit betreuten Diplomarbeit [FES07].

Kernidee ist, bereits berechnete Lösungen [HYDR] zu nutzen. Diese Lösungen vereinfachen das Problem derart, dass die inneren Kreise denselben Durchmesser haben. Es hat sich jedoch gezeigt, dass im Automotive-Bereich diese Vereinfachung durchaus zu vertretbaren Ergebnissen führt, da hier es hier sehr viele Leitungen mit ähnlichen Durchmessern gibt. Vergleichsmessungen mit reellen Leitungssatzsträngen ergaben bis zu 6% Abweichung [FES07].

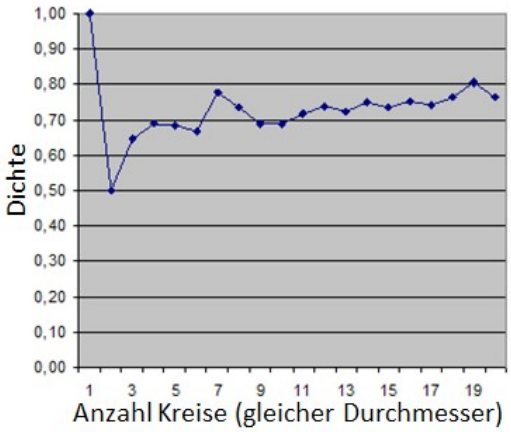


Abbildung 7.9: Dichteverteilung für die ersten 20 Kreise

Die Berechnung der Bündelfläche (Fläche des minimalen äußeren Kreises, siehe Abbildung 7.8) erfolgt durch

$$Bündelfläche = \frac{Nettofläche\left(\sum innere\ Kreise\right)}{Dichte}$$

Daraus resultiert ein Bündeldurchmesser von

$$Bündeldurchmesser = \sqrt{\frac{Bündelfläche * 4}{\pi}}$$

Für die Dichte wird in der Metrik eine lineare Annäherungsfunktion

$$f_{Dichte}(n) = 0{,}004\,n + 0{,}685 \quad \text{[FES07]}$$

verwendet. Die Steigung und der Offset wurden anhand der Dichteverteilung aus Abbildung 7.9 bestimmt. Für Leitungsstränge mit weniger als vier Leitungen greift die Metrik auf die berechneten Lösungen [HYDR] zu, da die Annäherungsgerade in diesem Bereich eine zu große Abweichung aufweist.

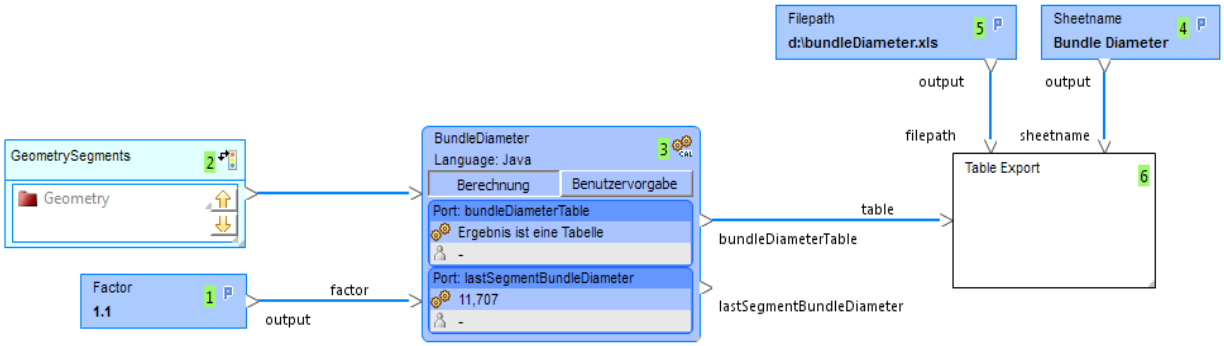


Abbildung 7.10: Metrik zur Berechnung des Bündeldurchmessers

Abbildung 7.10 zeigt das Diagramm der Befüllungsgrad-Metrik. Im Kontextblock *GeometrySegments* werden die zu berücksichtigenden Topologiesegmente übergeben. Der Parameter *Factor* ist der Korrekturfaktor, der auf das Ergebnis angewendet wird. Im Berechnungsblock BundleDiameter wird der Algorithmus angewendet. Die Implementierung in Java ist 204 Zeilen lang. Im rechten Teil der Metrik wird das Ergebnis als Tabelle exportiert.

Segment	Diameter	Cross Section Area
BodyFront_TunnelFront::TopologySegment	10,64016904	73,4854853
Body_DashFR::TopologySegment	15,14405662	148,8637519
Body_FrontCR::TopologySegment	19,70581265	252,0537773
BodyFront_DashFR::TopologySegment	12,31260586	98,40212252
Body_DashDrive::TopologySegment	9,428753305	57,70494839
BodyFront_DashFL::TopologySegment	23,78117974	367,0889416
BodyRL_BodyFL::TopologySegment	15,84347456	162,931619
BodyFront_Body::TopologySegment	10,64016904	73,4854853
BodyRear_TunnelRear::TopologySegment	15,80749388	162,1924207
Body_DashFL::TopologySegment	7,407081554	35,61219672
BodyFL_DashFL::TopologySegment	19,73478731	252,7955422

Tabelle 8: Ergebnis Befüllungsgradmetrik (gefiltert)

Tabelle 8 zeigt das Ergebnis der Befüllungsgradmetrik. Die Formatierungen wurden nachträglich bearbeitet, ferner wurde ein Filter angewendet, um die Anzahl der Zeilen zu reduzieren.

7.1.4 Bewertung elektrischer und elektronischer Eigenschaften

7.1.4.1 Buslastanalysen

Die hier vorgestellte Analysemetrik zur Ermittlung von Buslasten unterstützt CAN-, LIN-, FlexRay- und MOST-Bussysteme. Die Metrik ist in Abbildung 7.11 dargestellt, wobei nur der allgemeine Teil und die Erweiterung zur Berechnung des CAN-Busses abgebildet (in den mit *CAN-Berechnung* benannten Rechtecken) sind, um die graphische Darstellung nicht zu komplex werden zu lassen. Die Rechtecke an den freien Ports zeigen, welche Medium-spezifische Submetrik dort angeschlossen sind.

Das zu untersuchende Bussystem wird der Metrik über den Metrikkontextblock *Bussystems* als Eingabeartefakt übergeben. Im folgenden Berechnungsblock *BusLoadHandler* werden die Eingangsartefakte verarbeitet und die Struktur der Ergebnisports vorbereitet. Die konkrete Art des Bussystems wird über das assoziierte Bustyp-Artefakt (siehe Abschnitt 3.3.5.4) ermittelt und an die folgenden Berechnungsblöcke *SignalLoadShare* und *LoadCumulation* übergeben. Diese beiden Blöcke sind als interne Schleifen (siehe Abschnitt 6.4.7) angebunden, d.h. deren Berechnung wird vom *BusLoadHandler* während seiner eigenen Berechnung gesteuert.

Der Berechnungsblock *SignalLoadShare* erhält als Eingabe alle auf diesem Bus übertragenen SignalTransmissionen (siehe Abschnitt 3.3.4), wertet den Bustyp aus und delegiert die weitere Berechnung der statischen und dynamischen Buslast an die Medium-spezifischen Submetriken. Es werden FlexRay [ISO17458], CAN, und LIN unterstützt.

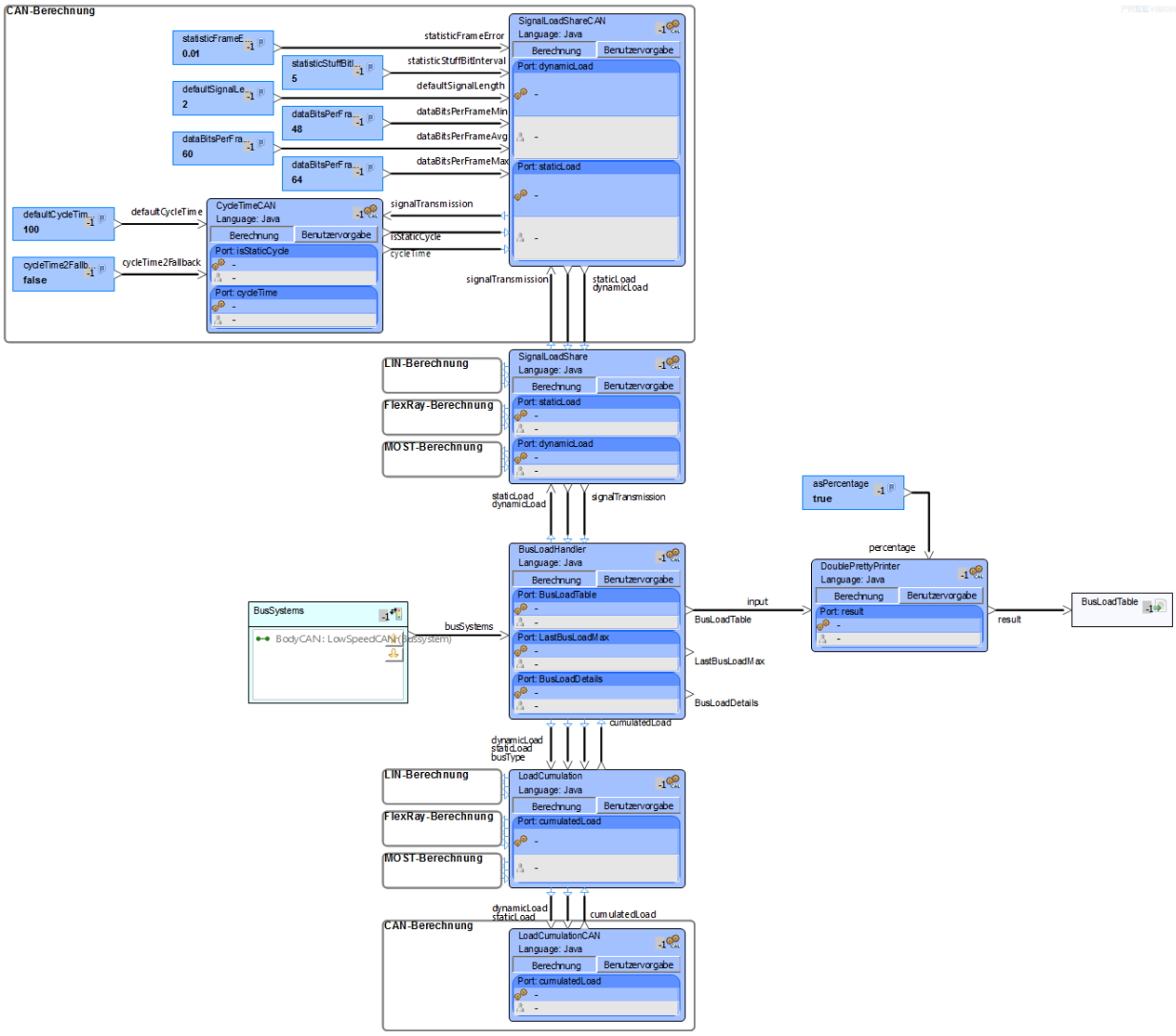


Abbildung 7.11: Analysemetrik zur Ermittlung der Buslast

7.2 FlexRay-Parametrisierung

Im Rahmen einer Kooperation zwischen der aquintos GmbH (heute Teil der Vector Informatik GmbH) und der Porsche AG im Jahre 2008 wurde im Rahmen der Diplomarbeit von Thibault Schneider [SCH08] untersucht, in wie weit sich FlexRay-Parametrisierungen mit Hilfe des im Rahmen dieser Arbeit vorgestellten Bewertungs-, Analyse- und Vergleichsframeworks berechnen und interpretieren lassen. Dazu wurde eine frühe Version des damals noch in Entwicklung befindlichen Frameworks verwendet. Im folgenden werden die Ergebnisse dieser Diplomarbeit vorgestellt.

7.2.1 FlexRay

Die Anfänge von FlexRay [ISO17458] gehen auf das FlexRay-Konsortium, einem Zusammenschluss von mehreren Firmen im Jahre 2000 zurück. Ziel war die Spezifikation eines

schnellen, echtzeitfähigen, fehlertoleranten Feldbussystems für den Einsatz in Kraftfahrzeugen. Im Jahre 2010 löste sich das Konsortium auf, FlexRay wurde in die ISO-Standards ISO 17458 Teil 1 bis 5 überführt.

FlexRay ist ein fehlertolerantes, deterministisches, serielles echtzeitfähiges Multi-Masternetzwerk. Es sieht zwei redundant arbeitende Kanäle mit je 10Mbit/s vor, die Kommunikation erfolgt über elektrische Medien im Zeitscheibenverfahren TDMA (*Time Division Multiple Access*). Es werden Bus-, Stern- und Hybridtopologieformen unterstützt.

Als zeitgesteuertes System ist die Konfiguration eines FlexRay-Bussystems deutlich komplexer als die anderer Bussysteme. Im FlexRay-Standard sind insgesamt 74 Parameter verankert, die einen theoretischen Lösungsraum mit 10^{48} möglichen Konfigurationen aufspannen [ASH06]. Des weiteren gibt es 43 mathematische Bedingungen und 19 Gleichungen, die eine gültige Buskonfiguration erfüllen muss [BRO10].

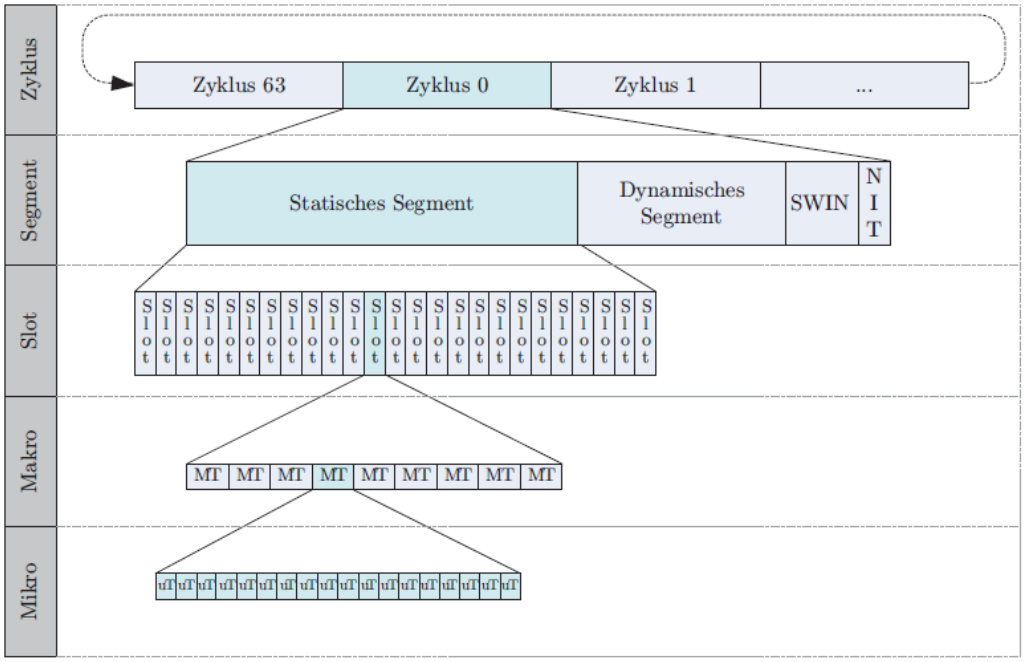


Abbildung 7.12: Das fünfstufige Zeitkonzept von FlexRay [BRO10]

Ein großer Teil der Parameter befassen sich mit der Auslegung und Konfiguration des FlexRay-Zeitkonzeptes, das in Abbildung 7.12 gezeigt ist. Zunächst gibt es 64 Zyklen, die periodisch übertragen werden. Jeder Zyklus ist unterteilt in ein statisches Segment mit einer konstanten Zeitscheibenzuordnung, ein dynamisches Segment mit einer variablen Zeitscheibenzuordnung, einem Seg-

ment für Symbole[17] sowie einer Ruhezeit, die zur Resynchronisation des FlexRay-Clusters genutzt wird. Die Segmente werden in Zeitschlitze, den sogenannten Slots zerlegt. Sie sind im statischen Segment einheitlich lang, im dynamischen Segment unabhängig definierbar. Die weitere Verfeinerung erfolgt durch Makro- und Microticks, die über den gesamten FlexRay-Cluster eine definierte, von der Konfiguration abhängige Größe haben.

7.2.2 Berechnung und Interpretation von FlexRay-Parametrisierungen

In [SCH08] werden die FlexRay-Parameter in insgesamt neun Gruppen unterteilt. Konstanten und Eingangsparameter, die von keinem anderen Parameter abhängig sind, werden mit Hilfe von Modellabfragen (vgl. Abschnitt 6.2.3) auf Modellierungsartefakte des EEA-Metamodells abgebildet. Dabei wird die Variantenmodellierung (vgl. Abschnitt 3.3.8) im E/E-Architekturmodell beachtet, um beispielsweise die Auswirkungen unterschiedlicher FlexRay-Bustopologien zu untersuchen und zu vergleichen.

Die anderen Parameter werden innerhalb der jeweiligen Gruppe durch Bewertungsmetriken berechnet. Parameter innerhalb einer Gruppe können von Parametern anderer Gruppen abhängen, dadurch ergibt sich die Berechnungsreihenfolge der Parameter. Die Gruppen sind im Einzelnen:

- Microtick und Macrotick: Beinhaltet Parameter zur Bestimmung der Micro- und Macroticks. Dabei handelt es sich um wichtige Zeiteinheiten im FlexRay-Protokoll.
- Statische Slots: Beinhaltet Parameter zur Auslegung des statischen Segments. Im statischen Segment ist die Zuordnung von zu übertragenden Signalen konstant.
- Dynamische Slots: Beinhaltet Parameter zur Konfiguration des optionalen dynamischen Segments. Hier werden meist Event-getriggerte Nachrichten übertragen.
- SymbolWindow und NIT: Beinhaltet Parameter zur Bestimmung von Korrekturwerten etc.
- Symbole: Parameter, die zur Definition der im SymbolWindow übertragenen Symbole beitragen.
- PhysicalLayer: Parameter mit physikalischem Hintergrund.
- Initialisierung der Uhr: Fasst Parameter zusammen, die benötigt werden, um die für die Bussynchronizität benötigten Uhren zu initialisieren.

17 Das Symbol-Segment wird nur bei Einsatz eines sogenannten Buswächters zur Erhöhung der Fehlertoleranz verwendet. Da dieses Konzept zur Zeit bei Kraftfahrzeugen nicht verwendet wird, wurde es nicht weiter beachtet [BRO10].

- Synchronisation der Uhr: Fasst Parameter zusammen, die benötigt werden, um die Uhren zu synchronisieren.
- Präzision: Beinhaltet Parameter zur Bestimmung relevanter Eckwerte der FlexRay-Konfiguration.

Für jede dieser Gruppen ist ein Diagramm und eine Ergebnistabelle vorhanden. Der Gesamtüberblick aller Parameter wird durch eine Ergebnistabelle über alle Parametergruppen gegeben.

Abbildung 7.13 zeigt beispielhaft die Berechnung der Nutzdateneffizienz im statischen Segment (nach [SCH08]). Die Berechnungsblöcke auf der linken Seite sind berechnete FlexRay-Parameter, die wiederum von anderen Blöcken abhängig sind, aber der Übersichtlichkeit halber nicht dargestellt werden. Analog dazu werden die anderen Parameter berechnet.

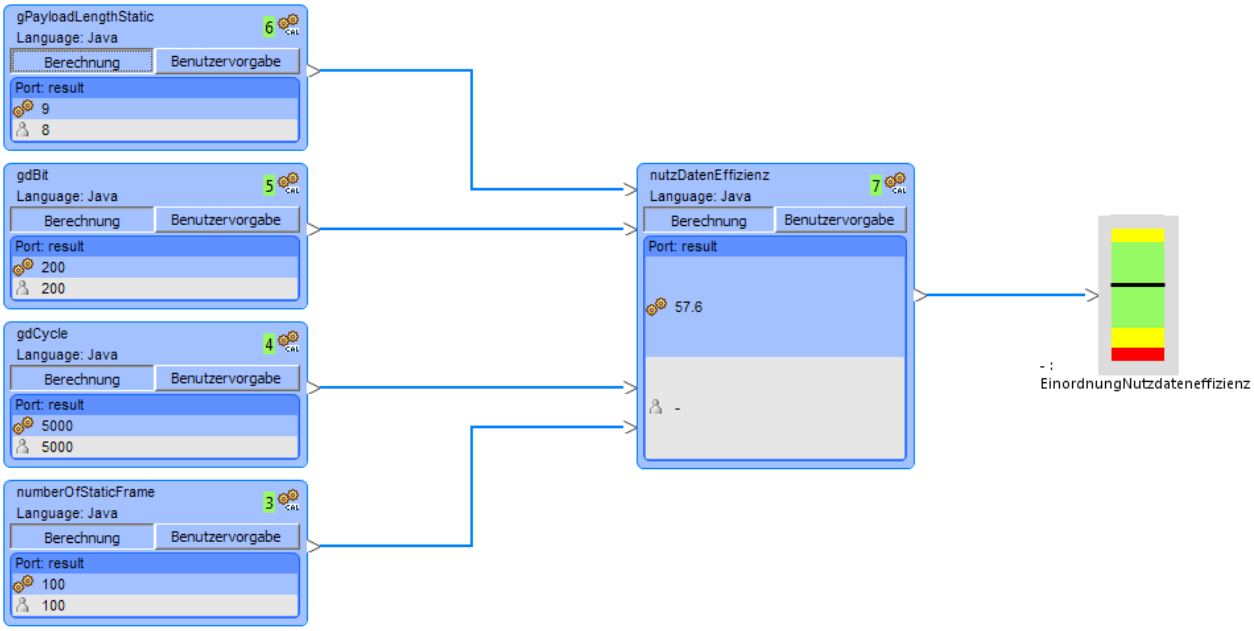


Abbildung 7.13: Ermittlung der Nutzdateneffizienz einer FlexRay-Parametrisierung (angelehnt an [SCH08])

Der Berechnungsblock *nutzDatenEffizienz* berechnet folgende Formel:

$$Nutzdateneffizienz_{StatischesSegment} = \frac{Länge_{StatischesSegment}}{Länge_{Slots}} * Nutzdaten_{Slot}$$

Weitere Details zur Formel sind in [BRO10] ausführlich beschrieben.

Die folgende Abbildung 7.14 zeigt den Ausschnitt einer FlexRay-Parametrisierung in tabellarischer Form. In den unterschiedlichen Spalten werden sowohl die Zahlenwerte als auch die aus den Diagrammen bekannte graphische Darstellung angezeigt.

Parametername	Grenzbereiche	Aktiver Wert	Berechneter Wert	Benutzerdefinierte Werte	Schwelle Positiv/Neutral	Schwelle Neutral/Negativ
gdMaxInitializationError		9.769000004470348	9.769000004470348	8	11.7 / 0.0	11.7 / 0.0
gdActionPointOffset_Robust		20.0	20.0		63.0 / 1.0	63.0 / 1.0
gdStaticSlot		212.0	212.0	24	661.0 / 4.0	661.0 / 4.0
aFrameLengthStatic		269.0	269.0	653	2638.0 / 0.0	2638.0 / 0.0
pdAcceptedStartupRange		[190.0, 379.0, 757.0]	[190.0, 379.0, 757.0]		1875.0 / 0.0	1875.0 / 0.0
pLatestTx		[234.0, 234.0, 234.0]	[234.0, 234.0, 234.0]		7980.0 / 0.0	7980.0 / 0.0
aFrameLengthDynamic		[5169.0, 5169.0, 5169.0]	[5169.0, 5169.0, 5169.0]	2	2638.0 / 0.0	2638.0 / 0.0
gdSymbolWindow		0	22.0	0	142.0 / 0.0	142.0 / 0.0
gdMaxPropagationDelay		0.055000002902323	0.055000002902323	2.5	2.5 / 0.0	2.5 / 0.0
gdMinPropagationDelay		0.19600000447034838	0.19600000447034838	0	0.8550000029802323	0.8550000029802323
gClusterDriftDamping		2.0	2.0	12	5.0 / 0.0	5.0 / 0.0
gdActionPointOffset_Normal2		7.0	7.0	39	63.0 / 1.0	63.0 / 1.0
gdMinislotActionPointOffset		7.0	7.0		31.0 / 1.0	31.0 / 1.0
gPayloadLengthStatic		9.0	9.0	8	127.0 / 0.0	127.0 / 0.0
gAssumedPrecision		9.110000005960465	9.110000005960465	2.1	3.0589999985098846	11.7 / 0.15
gOffsetCorrectionMax		10.690437507450508	10.690437507450508		393.567 / 0.15	393.567 / 0.15
gdMinislot		15.0	15.0	5	63.0 / 2.0	63.0 / 2.0
gMacroPerCycle		3636.0	3636.0		16000.0 / 10.0	16000.0 / 10.0

Abbildung 7.14: Ausschnitt der Ergebnistabelle einer FlexRay-Parametrisierung [BRO10]

7.2.3 Ergebnisse

In [SCH08] wurde gezeigt, dass eine modellbasierte, automatisierte Parametrisierung von FlexRay-Bussystemen durchgeführt werden kann. Durch die graphische Darstellung der Ergebnisse und Parameterabhängigkeiten sind Zusammenhänge schnell und intuitiv erfassbar. Durch die integrierte Beachtung von Realisierungsalternativen und Varianten konnten mit wenig Mehraufwand unterschiedliche Bustopologien und Anfangskonfigurationen betrachtet und miteinander verglichen werden. Auf diesen Ergebnissen aufbauend wurde in [BRO10] eine Fallstudie durchgeführt, in der sechs Fahrzeugderivate aufgebaut und miteinander verglichen wurden (Abbildung 7.15).

Sportwagen (HM) (Basis)	Sportwagen (HM) (Premium)	Sportwagen (MM) (Sport)	Sportwagen (MM) (Premium)	SUV (Komfort)	SUV (Sport)
Fahrwerk	Integriert	Fahrwerk	Integriert	Bremsen	Traktions-management
Dezentral	Low-End-Cluster	Low-End-Cluster	High-End-Cluster	Low-End-Cluster	Low-End-Cluster
Unabhängig	Unabhängig	Unabhängig	Unabhängig	DC/DC-Wandler	Unabhängig
Gruppe B	Vollständig	Gruppe B	Vollständig	Gruppe A	Gruppe B
Passiver Bus	4-fach Stern	4-fach Stern	8-fach Stern	4-fach Stern	4-fach Stern
LQ-Quarz (ink.PLL)	HQ-Quarz (ink.PLL)	LQ-Quarz	Plattformquarz	HQ-Quarz (ink.PLL)	HQ-Quarz
Zulieferer B	Zulieferer B	Zulieferer A	Zulieferer A	Zulieferer B	Zulieferer A
Klemmen-konzept B	Klemmen-konzept B	Klemmen-konzept B	Klemmen-konzept A	Klemmen-konzept A	Klemmen-konzept B

Abbildung 7.15: In [BRO10] untersuchte Fahrzeugderivate

Die Ergebnisse der untersuchten Fahrzeugderivate ergeben das in Abbildung 7.16 gezeigte Netzdia-

gramm:

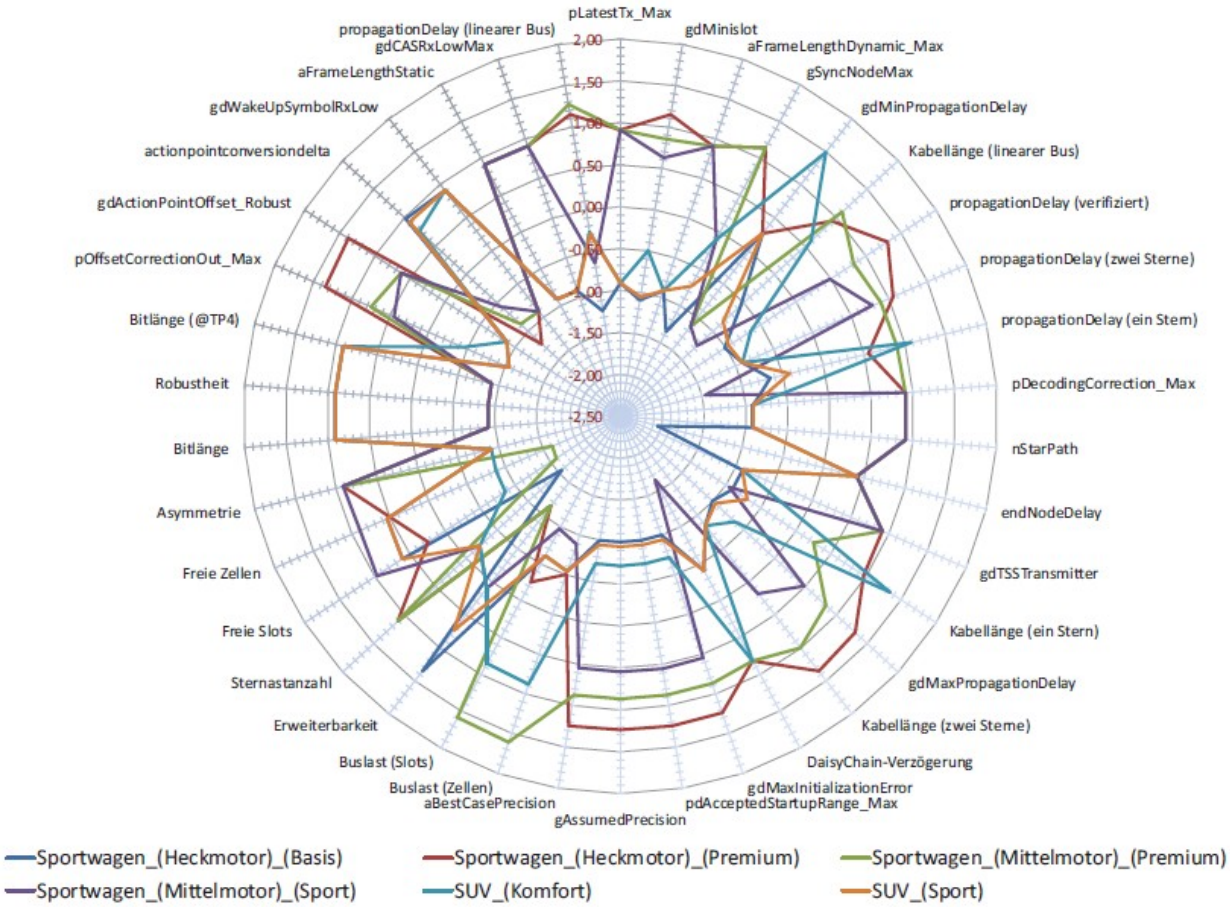


Abbildung 7.16: Ausgewählten FlexRay- und Qualitätsparameter sechs untersuchter Fahrzeugederivate in der prozentualen Abweichung zueinander [BRO10]

Durch die Verknüpfung von FlexRay-relevanten Architekturdaten und der Berechnung der Parameter konnten Zusammenhänge analysiert, miteinander verglichen und optimiert werden. So wurden zum Beispiel Einflüsse unterschiedlicher Topologie-Umsetzungen (mit/ohne Sternkoppler / Hinzunahme eines zweiten Sternkopplers) oder die Auswirkungen unterschiedlicher Sensorik- und Uhrenkonzepte gegenübergestellt und miteinander verglichen. Für weitere Details sei hier an [BRO10] verwiesen.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Die Komplexität von Elektrik/Elektronik-Architekturen im Fahrzeug hat in den letzten Jahren kontinuierlich zugenommen. Durch die Einführung immer neuer Systeme, vor allem in den Bereichen assistiertes/automatisiertes Fahren ([AUD14],[DAI14],[HER14]), Sicherheit, hybride- und elektrische Antriebe sowie Steigerung der Effizienz wird sich dieser Trend, vor allem auch getrieben durch Kundenwünsche, in den nächsten Jahren unvermindert fortsetzen [GMRMG08]. Um im globalen Markt bestehen zu können, herrscht bei Herstellern und Zulieferern ein hoher Kosten- und Zeitdruck. Auch müssen Qualität und Zuverlässigkeit der Produkte den Kundenvorstellungen entsprechen.

Dieses Umfeld stellt mit den gegebenen Randbedingungen hohe Ansprüche an die Entwicklung künftiger Fahrzeuge, insbesondere im Hinblick auf die Elektrik/Elektronik-Architektur.

Mit dieser Arbeit wird ein Verfahren zur Bewertung, Analyse und zum Vergleich von Elektrik/Elektronik-Architekturen vorgestellt, um bereits von der frühen Entwurfs- und Entwicklungsphase an wichtige Kennzahlen im Blick zu haben und um Effekte von Änderungen, Realisierungsalternativen und Optimierungsmaßnahmen nachvollziehen zu können. Die Ergebnisse dieser Arbeit sind Bestandteil des Werkzeugs PREEvision [PREE] und bereits produktiv im Einsatz. Folgende Merkmale stellen den Mehrwert dieser Arbeit heraus:

8.1.1 Modellbasierte Beschreibung

Durch die modellbasierte Beschreibung, die auf der Grundlage eines MOF-konformen [MOF241], formalen Metamodell beruht, wird eine domänenspezifische Sprache (Domain Specific Language DSL [FOW10]) geschaffen. Diese Sprache konzentriert sich auf wesentliche Aspekte der Bewertung, Analyse und Vergleichs und bildet somit eine einheitliche Basis zum übergreifenden Verständnis von Bewertungs-, Analyse- und Vergleichsvorschriften sowie deren Ergebnisse.

Für die wichtigsten Aufgaben stehen spezialisierte Blöcke zur Verfügung. Für die Suche nach relevanten Modellierungsartefakten werden modellbasierte Modellabfragen gemäß der M²TOS-Technologie [REI05] herangezogen. Für weitere Aufgaben wie die Darstellung von Ergebnissen, Umschaltung von Varianten, Durchführung von Schleifen, Im- und Exporte für unterschiedliche Formate

stehen weitere spezialisierte Blöcke zur Verfügung. Somit wird die Notwendigkeit an zu programmierenden Quellcode auf ein Minimum reduziert.

Die Integration in das Werkzeug PREEvision [PREE], welches die modellbasierte Beschreibung von Elektrik/Elektronik-Architekturen im Fahrzeug erlaubt, ermöglicht die direkte Verarbeitung von E/E-Modellierungsartefakten in Bewertungs-, Analyse- und Vergleichsvorschriften.

8.1.2 Graphische Notation

Die graphische Notation zur Repräsentation der domänenspezifischen Sprache ermöglicht das intuitive Verständnis und die komfortable Editierung von Bewertungs-, Analyse- und Vergleichsvorschriften. Die Darstellung wesentlicher Zusammenhänge, Abläufe und Ergebnisse ist ein wichtiges Kriterium für Nachvollziehbarkeit und Nachhaltigkeit der Berechnungsvorschriften.

Bei der graphischen Notation handelt es sich um Blockdiagramme mit einer Datenfluss-orientierten Sicht. Die einzelnen Blöcke repräsentieren die spezialisierten Aufgaben, die über ein Portkonzept mit generischen Datenflüssen miteinander verknüpft werden. Berechnungsabläufe lassen sich intuitiv erfassen.

8.1.3 Transparenz von Bewertungs-, Analyse- und Vergleichsvorschriften

Modellierungsartefakte, die für die Bewertungsvorschriften als Ein- und Ausgangsdaten herangezogen werden, sind direkt in die graphische Notation integriert, navigierbar und dienen somit der Verifikation der Berechnungsergebnisse. Hierzu verfügt das Framework über vollen lesenden und schreibenden Zugriff auf Modellierungsartefakte.

Zur Analyse des Ablaufs von Berechnungsvorschriften werden wichtige Informationen wie Berechnungsreihenfolge und Berechnungsergebnis der einzelnen Blöcke angezeigt. Die Ergebnisse aller Ports aller Blöcke können nach der Berechnung dargestellt werden. Handelt es sich bei den Ergebnissen um Modellierungsartefakte, so sind diese transparent als navigierbare Links ausgeführt.

8.1.4 Effiziente Unterstützung von Architekturvarianten und Realisierungsalternativen

Moderne E/E-Architekturen in Fahrzeugen unterstützen eine Vielzahl von Funktionen und werden oft für eine oder mehrere Plattformen einschließlich etwaiger Derivate samt länder- und ausstat-

tungsspezifischer Varianten entwickelt. Damit ergibt sich eine sehr hohe Anzahl theoretisch möglicher (ca. 10^{27} [JAEN12]) und tatsächlich angebotener Varianten (z.B. $7{,}53 \cdot 10^{12}$ bereits bei einem Kleinfahrzeug [JAEN12]). In der Realität wird eine kleine Anzahl von Varianten gebildet, die ein möglichst breites Spektrum der Anwendungsfälle und häufig produzierter Konfigurationen abbilden. Bewertungen, Analysen und Optimierungen werden auf diesen Varianten durchgeführt, die dann verglichen werden.

Mit dem in dieser Arbeit vorgestellten automatisierten Framework kann eine höhere Anzahl definierter Varianten nach festgelegten Kriterien in kürzerer Zeit bewertet, analysiert und verglichen werden. Mit diesem Verfahren werden nicht nur Länder- und Austattungsvarianten behandelt, sondern auch Architekturalternativen verarbeitet, um Fragestellungen während des Architekturdesigns zu beantworten.

8.1.5 Sensitivität und Was-Wäre-Wenn-Szenarien

Eine weitere, wichtige Fragestellung während der Architekturentwicklung ist die Ermittlung von Einflüssen bestimmter Eingangsgrößen, um so den Effekt von Architekturentscheidungen besser abschätzen zu können.

Mit den Ergebnissen der hier vorgestellten Arbeit können Sensitivität und War-Wäre-Wenn-Szenarien für Parameter und berechneter Größen konfiguriert und durchgerechnet werden. Die Ergebnisse können nach erfolgter Berechnung betrachtet und bezüglich aufgetretener Auffälligkeiten wie Maxima und Minima untersucht werden.

8.2 Ausblick

Das im Rahmen dieser Arbeit vorgestellte Verfahren zur Bewertung, Analyse und Vergleich von E/E-Architekturen im Fahrzeug setzt vor allem auf einen modellbasierten, domänenspezifischen Ansatz mit graphischer Notation und die Integration mit den Modellierungsartefakten der zu bewertenden E/E-Architektur. Dazu kommt vor allem der lesende Zugriff auf die Modellierungsartefakte zum Einsatz.

Über den schreibenden Zugriff, der bereits technisch von den Ergebnissen dieser Arbeit unterstützt wird, eröffnen sich vollkommen neue Möglichkeiten, die weit über Bewertungs-, Analyse- und Vergleichsaspekte hinaus gehen. Damit ist die Grundlage für automatisierte und automatische Architekturoptimierungen geschaffen.

Diese Automatismen sind in unterschiedlichen Ebenen denkbar. Der Anfang sind kleine, lokal arbeitende Modellaktualisierungen. Als Beispiel sei hier die automatisierte Anpassung von zu verwendenden Kabel- und Adertypen genannt, um zum Beispiel Aluminium-/Kupfervergleiche durchzuführen. Darüber hinaus sind wesentlich weitreichendere, ebenenübergreifende Optimierungen vorstellbar. Hier sind vor allem die Methoden und die in Frage kommenden Algorithmen, wie zum Beispiel evolutionäre Algorithmen, Ziel weiterer Forschungen. Wesentliche Punkte sind dabei die Beachtung der Komplexität und der Größe von E/E-Architekturmodellen und deren Auswirkung auf die Laufzeit und Ressourcenbedarf von Optimierungsmaßnahmen. Erste Schritte hierzu wurden durch die beiden, im Rahmen dieser Arbeit betreuten Abschlussarbeiten [KIR14] und [AKS15], die technologisch aufeinander aufsetzen, gemacht.

Bei dem Einsatz nicht deterministischer Algorithmen, von denen bekannt ist, dass sie „gute“ Lösungen liefern, aber keine Aussage getroffen werden kann, „wie gut“ diese Lösungen im Lösungsraum sind, ist die Akzeptanz und Glaubwürdigkeit der Ergebnisse ein sehr wichtiges Kriterium für die Anwender solcher Technologien.

Definitionen

Def. 1 Abstrakte Syntax

Die Struktur einer Modellierungssprache ist ihre abstrakte Syntax. (in Anlehnung an [GRA09, S.16])

Def. 2 Abstraktionsebene

„Eine Abstraktionsebene beschreibt die für diese Ebene wesentlichen Modellierungsartefakte, den Aufbau der Modellierungsartefakte und die Wechselwirkungen zwischen den Artefakten.“ [MAT09, S.196]

Def. 3 Aktor/Aktuator

„Aktoren führen durch Eingangssignale bestimmte Aktionen aus. Meist sind Aktoren eigenständige Motoren.“ [MAT09, S.196]

Def. 4 Ampelblock

Ein Ampelblock ist ein Metrikblock (Def. 40), der eingehende Daten mit vorgegebenen Schwellwerten vergleicht und das Ergebnis als Ampel visualisiert.

Def. 5 Architektur

„The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.“ [IEEE1471, S.3]

Def. 6 Artefakt (Modelle)

Ein Artefakt ist ein nicht weiter typisiertes Element eines Modells oder eines Metamodells.

Def. 7 Assoziation (Modelle)

Eine Assoziation ist eine kennt-Beziehung zwischen Artefakten, die durch Multiplizitäten und Rollennamen an ihren Enden näher spezifiziert wird.

Def. 8 Ausschlusskriterium

Ein Ausschlusskriterium ist ein Bewertungskriterium, welches bei Nichterfüllung eine vor der

Bewertung festgelegte Entscheidung zur Folge hat.

Def. 9 Berechnungsblock

Ein Berechnungsblock ist ein Metrikblock (Def. 40), der vorgegebenen Java-Quellcode ausführen kann. Benötigte Daten werden über eingehende Metrikblockports (Def. 41) erwartet, Berechnungsergebnisse werden an ausgehende Metrikblockports übergeben.

Def. 10 Bewertung

„Bewertung heißt die Verknüpfung der zugänglichen Information eines Sachverhaltes mit dem persönlichen Wertesystem zu einem Urteil über den entsprechenden Sachverhalt. Eine Bewertung ist ein Hilfsmittel der Entscheidungsfindung, stellt jedoch noch nicht die Entscheidung selbst dar.“ [GIEG91, S.19]

Def. 11 Bewertungslogik

„Eine Methode der Bewertungslogik ist ein formalisiertes Regelwerk, das vorschreibt, wie zugängliche Informationen und Werthaltungen zu einem Bewertungsurteil verknüpft werden müssen. Eine Bewertungslogik muss unabhängig von den jeweiligen Sachinhalten anwendbar sein.“ [GIEG91, S.20]

Def. 12 Bewertungsprozess

„Ablauf der Bewertung eines komplexen Sachverhaltes, mit der mehrere Personen und/oder Grup[p]en einen bestimmten Zeitraum beschäftigt sind.“ [GIEG91, S.19]

Def. 13 Bewertungsurteil

Das Bewertungsurteil ist das Ergebnis einer Bewertung.

Def. 14 Bewertungsverfahren

„Konkreter, methodisch festgelegter Ablauf einer Bewertung mit klaren Regeln, der von der Problemdefinition bis zum Bewertungsergebnis reicht. Ein Bewertungsverfahren gilt für einen bestimmten Anwendungszweck.“ [GIEG91, S.20]

Def. 15 Bewertungsziel

Das Bewertungsziel beschreibt die Motivation einer oder mehrerer Bewertungen und Interpretationen für deren Bewertungsurteile.

Def. 16 Bordnetz

„Das Bordnetz eines Kfz besteht aus dem Generator als Energiewandler, einer oder mehrerer Batterien als Energiespeicher und den elektrischen Geräten als Verbraucher.“ [BOS14, S.1102]

Def. 17 Bussystem

„Unter dem Begriff Bussystem wird ... die Zusammenfassung verschiedener Charakteristika der Kommunikation mehrerer Teilnehmer verstanden. Diese Charakteristika beinhalten die Art der Verknüpfung der Teilnehmer (Topologie des Bussystems),die Regeln, die die Kommunikation unter den Teilnehmern festlegen (Protokoll des Bussystems) bis hin zur Spezifikation der physikalischen Realisierung.“ [REI12, S.2]

Def. 18 Diskreter Ergebnisblock

Ein diskreter Ergebnisblock ist ein Metrikblock (Def. 40), der eingehende Daten bezüglich vorgegebener, diskreter Werte vergleicht und das Ergebnis im Fall einer Übereinstimmung grün und andernfalls rot darstellt.

Def. 19 Dokumentation (E/E-Architektur)

Die Dokumentation einer E/E-Architektur umfasst versionierte, archivierbare, auf Basis einer oder mehrerer E/E-Architekturen zusammengetragene Informationen in einem geeigneten Format. Dazu gehören das Lastenheft, Unterlagen zu Audits, Bewertungs- und Vergleichsergebnisse von E/E-Architekturen, etc.

Def. 20 Domain Specific Language DSL

„a computer programming language of limited expressiveness focused on a particular domain.“ [FOW10, S.27]

Def. 21 Eingebettetes System

„Ein eingebettetes System ... ist eine Software-/Hardware-Einheit, die über Sensoren und Aktuatoren mit einem Gesamtsystem verbunden ist und darin Überwachungs-, Steuerungs- beziehungsweise Regelungsaufgaben übernimmt. In der Regel handelt es sich bei eingebetteten Systemen um reaktive, häufig auch um hybride verteilte Systeme mit Echtzeitanforderungen. Typischerweise sind solche Systeme dem menschlichen Benutzer nicht direkt sichtbar, er interagiert unbewußt mit dem eingebetteten System.“ [BRO98, S.13]

Def. 22 E/E-Architektur

„Eine E/E-Architektur beschreibt fahrzeugweit alle E/E-Systeme hinsichtlich derer Struktur und Interaktion, d.h. der Schnittstellen, der funktionalen, logischen, elektrischen sowie physikalischen Vernetzung, der Kommunikation, der elektrischen Versorgung sowie der Topologie.“ [JAEN12, S.16]

Def. 23 E/E-Architektur-Bewertungskriterium

Ein E/E-Architektur-Bewertungskriterium ist ein Bewertungskriterium. Es definiert als Eingangsgröße eine Menge von Artefakten einer E/E-Architektur. Als Ausgangsgröße definiert es eine oder mehrere Kennzahlen, die für eine Entscheidung relevant sind. Ein Sonderfall von E/E-Architektur-Bewertungskriterien sind Ausschlusskriterien.

Def. 24 E/E-Architekturmetrik, Metrik (angelehnt an Softwarequalitätsmetrik)

Eine E/E-Architekturmetrik ist eine Berechnungsvorschrift, mit deren Hilfe die Bewertung einer E/E-Architektur bezüglich vorgegebener Bewertungskriterien in eine oder mehrere Kennzahlen durchgeführt wird.

Def. 25 Elektrik/Elektronik (E/E-Architektur)

Die Elektrik/Elektronik umfasst alle elektrischen und elektronischen Bauteile eines Systems zur Erfüllung von Funktionen und Funktionalitäten im Fahrzeug. Unter den elektrischen Bauteilen versteht man die analogen Komponenten des Systems, unter elektronischen Bauteilen versteht man die digital arbeitenden Bauteile des Systems.

Def. 26 Entwurfsmuster

Entwurfsmuster sind Lösungsansätze für wiederkehrende Problemstellungen in der Softwarearchitektur. Sie werden als Vorlage zur Lösung des Problems in Abhängigkeit des jeweiligen spezifischen Kontextes verwendet.

Def. 27 Generalisierung/Vererbung (Modelle)

„Klassen werden in eine Hierarchie eingeordnet, die eine Weiterleitung von Informationen von oben nach unten ermöglicht. Eine Unterklasse verfügt dann über die Eigenschaften und das Verhalten der Oberklasse. Sie „erbt“ diese Charakteristika. Entsprechend wird die Hierarchie auch Vererbungsstruktur oder Klassenhierarchie genannt.“ [FOR07, S.26]

Def. 28 Joinblock

Ein Joinblock ist ein Metrikblock (Def. 40), der zwei Ausgangstabellen (rechte und linke Tabelle) miteinander bezüglich eines gemeinsamen Spaltennamens als Verknüpfungskriterium zu einer Ergebnistabelle verknüpft.

Def. 29 Komponente / E/E-Komponente

Unter E/E-Komponente versteht man Steuergeräte, Sensoren, Aktoren eines Fahrzeugs sowie alle anderen Bauteile, die als Einheit verbaut werden. (in Anlehnung an [FREE07, S.1])

Def. 30 Komposition (Modelle)

Eine Komposition ist eine besitzt-Beziehung zwischen Artefakten, bei denen ein Artefakt aus anderen Artefakten zusammengesetzt wird.

Def. 31 Konkrete Syntax

„Die konkrete Syntax definiert die Symbole einer Modellierungssprache und ihre zulässige Zusammenstellung.“ [MAT09, S.201]

Def. 32 Kosten (E/E-Architektur)

Die Kosten einer E/E-Architektur setzen sich aus unterschiedlichen Kostenteilen zusammen. Neben den reinen Materialkosten gibt es Kosten für Entwicklung, Montage, Test, Wartung, Aftermarket, etc, die z.T. nicht unerheblich sind. Meist werden Kosten monetär betrachtet, es kann jedoch auch andere Sichtweisen, z.B. physikalische Eigenschaften, geben. Des weiteren gibt es, auf die einzelnen Bauteile der E/E-Architektur, geschätzte, berechnete und verhandelte Kosten.

Def. 33 Kriterium, Bewertungskriterium

„Ein Kriterium … ist ein Merkmal, das bei einer Auswahl zwischen … Objekten (Gegenständen, Eigenschaften, .. usw.) relevant für die Entscheidung ist.“ [WIBK]

Def. 34 Machbarkeit (E/E-Architektur)

Die Machbarkeit einer E/E-Architektur wird durch eine Reihe von Bewertungs- oder Ausschlusskriterien beschrieben, die deren Ergebnis die Entscheidungsgrundlage bilden, ob die bewertete E/E-Architektur realisiert werden kann.

Def. 35 Mapping

„Ein Mapping ist die bidirektionale Abbildung eines Artefakts in die nächst abstraktere oder konkretere Ebene.“ [MGRMG08]

Def. 36 Mechanische Eigenschaften (E/E-Architektur)

Die mechanischen Eigenschaften einer E/E-Architektur beschreiben physikalische Eigenschaften von Bauteilen der Elektrik/Elektronik selbst als auch deren direkte relevante Umgebung im Fahrzeug.

Def. 37 Mechatronisches System

„Elektronik (, die) zur Steuerung und Regelung mechanischer Vorgänge räumlich eng mit mechanischen Systembestandteilen verbunden (ist).“ [BRO98, S.13].

Def. 38 Metamodell

„Ein Metamodell ist eine formale Beschreibung eines Modells. Es ist ein Modell eines Modells.“ [MAT09, S.202]

Def. 39 Metrik

Siehe Def. 24: E/E-Architekturmetrik, Metrik (angelehnt an Softwarequalitätsmetrik)

Def. 40 Metrikblock

Ein Metrikblock kapselt einen Ausführungsschritt einer Metrik. Ein Metrikblock verfügt über Metrikblockports (Def. 41), um benötigte Daten zu empfangen und seine Ergebnisse weiterzugeben.

Def. 41 Metrikblockport (Port)

Ein Metrikblockport stellt die Verbindung zwischen einem Metrikblock (Def. 40) und einem Metrikdatenfluss (Def. 42) her. Man unterscheidet zwischen Eingangports, über welche benötigte Daten dem Metrikblock zur Verfügung gestellt werden und Ausgangports, über welche der Metrikblock seine Ergebnisse an folgende Metrikblöcke weitergibt.

Def. 42 Metrikdatenfluss (Datenfluss)

Ein Metrikdatenfluss verbindet einen ausgehenden Metrikblockport (Def. 41) mit vielen eingehenden Metrikblockports. Der Metrikdatenfluss übermittelt das Ergebnis des Metrikblocks

(Def. 40) am ausgehenden Metrikblockport zum Zeitpunkt der Ergebnisbereitstellung an die eingehenden Metrikblockports.

Def. 43 Metrikkontextblock

Ein Metrikkontextblock referenziert auf beliebige Artefakte des Modells und übergibt diese über einen ausgehenden Metrikblockport (Def. 41) an nachfolgende Metrikblöcke (Def. 40). Metrikkontextblöcke können zur Laufzeit dynamisch, mit einem erst zur Ausführungszeit bekannten Artefakt überschrieben werden. In diesem Fall werden die referenzierten Artefakte des Modells ignoriert.

Def. 44 Mikrocontroller

„Mikrocontroller sind spezielle Mikrorechner auf einem Chip, die auf spezifische Anwendungsfälle zugeschnitten sind. Meist sind dies Steuerungs- oder Kommunikationsaufgaben, die einmal programmiert und dann für die Lebensdauer des Mikrocontrollers auf diesem ausgeführt werden“ [BRI10, S.73]

Def. 45 Modell

„Ein Modell ist eine Menge von Aussagen über das zu analysierende System. Dabei wird von der Realität abstrahiert.“ [REI05, S.22]

Def. 46 Modellabfrage

„Eine Modellabfrage ist eine Zusammenstellung von einer oder mehreren Suchregeln.“ [MAT09, S.202], in Anlehnung an [GMKMG09]

Def. 47 Modellabfrageblock

Ein Modellabfrageblock stellt die Funktionalität einer Modellabfrage (Def. 46) in einer E/E-Architekturmetrik (Def. 24) zur Verfügung.

Def. 48 Modellartefakt

„Ein Modellartefakt ist ein Element, das zur Modellierung innerhalb eines Modells verwendet wird.“ [MAT09, S.202]

Def. 49 Modelltransformation

„Eine Modelltransformation ist über eine Menge von Modellartefakten X, die man Quellm-

odell nennt, in eine andere Menge von Modellartefakten Y, die man Zielmodell nennt, durch ein Regelwerk (eine Menge von Regeln) bestimmt, das den bestimmten x aus X ein y aus Y zuordnet." [REI05, S.74]

Def. 50 Multiplizität (Modelle)

Eine Multiplizität drückt die Kardinalität an den Beziehungsenden von Relationen aus. Die Multiplizität beinhaltet eine untere und eine obere Grenze.

Def. 51 Optimierung

„Das Gebiet der Optimierung in der angewandten Mathematik beschäftigt sich damit, optimale Parameter eines - meist komplexen - Systems zu finden." [WIOPT10]

Def. 52 Optimum

„Unter einem Optimum (lateinisch optimum, Neutrum von optimus ‚Bester, Hervorragendster', Superlativ von bonus ‚gut') versteht man das beste erreichbare Resultat im Sinne eines Kompromisses zwischen verschiedenen Parametern oder Eigenschaften unter dem Aspekt einer Anwendung, einer Nutzung oder eines Zieles." [WIOPTU14]

Def. 53 Pareto-Optimalität

„A point $x^* \in C$ [18] *is said to be (globally) Pareto optimal or a (globally)efficient point or a non-dominated or a non-inferior point for (MOP[19]) if and only if* $\nexists x \in C$ *such that* $F(x) \leq F(x^*)$ *with at least one strict inequality (the* $\leq$ *implies term-by-term inequality)." [DD96, S.1]*

Def. 54 Paket (Softwaretechnik, Modelle)

„Ein Paket ist ein Hierarchisierungsartefakt welches gemäß seiner spezifischen konkreten Spezifikation sich selbst und andere Artefakte aufnimmt."

Def. 55 Qualität

„Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt" [ISO9000, S.18]

18 *C* ist definiert als $C = \{ x : h(x) = 0, g(x) \leq 0, a \leq x \leq b \}$ mit $h(x)$ als Gleichheitsrandbedingung und $g(x)$ als Ungleichheitsrandbedingung (siehe Abschnitt 4.3.1)

19 MOP = Multicriteria Optimization Problem (siehe Abschnitt 4.3)

Def. 56 Qualitätsattribut

„Quality attributes represent architecture quality requirements. Besides a requirement itself, additional information on how to understand the requirement and evaluate regarding it has to be provided.“ [FLO08, S.57]

Def. 57 Relation (Modelle)

Eine Relation drückt eine nicht weiter typisierte Beziehung zwischen Artefakten aus.

Def. 58 Report- und Dokumentationsblock

Ein Report- und Dokumentationsblock ist ein Metrikblock (Def. 40), der eingehende Daten einem Dokumentengenerator übergibt.

Def. 59 Rollenname (Modelle)

Der Rollenname benennt beide Enden einer Relation, über welchen das Artefakt der einen Seite dem Artefakt auf der anderen Seite in Erscheinung tritt.

Def. 60 Schleifenblock

Ein Schleifenblock ist ein Metrikblock (Def. 40), der über beinhaltete Metrikblöcke bezüglich einer vorgegebenen Anzahl von Elementen iteriert. Die Teilergebnisse der Iterationsläufe werden als Gesamtergebnis zusammengefasst und weitergegeben.

Def. 61 Sensor

„Ein Sensor wandelt eine meist nichtelektrische Messgröße (Eingangsgröße) in ein elektrisches Ausgangssignal um. Häufig werden Sensoren auch als Wandler (Transducer) bezeichnet: Sie wandeln eine Messgröße von einer Energieform in eine andere.“ [REI12, S.95]

Def. 62 Skalenblock

Ein Skalenblock ist ein Metrikblock (Def. 40), der eingehende Daten relativ und maßstäblich zu konfigurierbaren Bezugsgrößen in Form eines vertikalen Bandes visualisiert.

Def. 63 Softwarequalitätsmetrik

„A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.“ [IEEE1061, S.3]

Def. 64 Sortier- und Filterblock

Ein Sortier- und Filterblock ist ein Metrikblock (Def. 40), der Eingangsdaten bezüglich einer vorgegebenen Konfiguration sortiert und/oder filtert und das Ergebnis weitergibt.

Def. 65 Steuergerät

„Ein Steuergerät (engl. Electronic Control Unit, ECU) ist (insbesondere in der Automobiltechnik) die physikalische Umsetzung (Implementierung) eines eingebetteten Systems. In mechatronischen Systemen bilden Steuergerät und Sensorik/Aktuatorik oft eine Einheit.“ [BRO98, S.13]

Def. 66 System

„A collection of components organized to accomplish a specific function or set of functions.“ [IEEE610, S.73]

Abbildungsverzeichnis

Literaturverzeichnis

ALE77: C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksfahl-King, S. Angel: A Pattern Language: Towns, Buildings, Construction, 1977

ATEANA10: ATESST2: Guideline for Analysis-driven modeling and architecture evaluation method, Technischer Bericht, European Commission ICT Research FP7, Deliverable D5.2.1, 2010

ASH06: Armengaud, E. ; Steininger, A. ; Horauer, M.: Automatic Parameter Identification in FlexRay Based Automotive Communication Networks, 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'06), September 2006

AUTROA09: Fürst, Simon et al.: AUTOSAR – A Worldwide Standard is on the Road., 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden 2009

AUTVFB08: AUTOSAR: Specification of the Virtual Functional Bus, Document Version 1.0.2, Release 3.1, Revision 0001

BAS12: Gasser, T.M.: Rechtsfolgen zunehmender Fahrzeugautomatisierung, Forschung kompakt 11/12, Bundesamt für Straßenwesen Bergisch Gladbach, 2012

BFM05: Belschner, R.; Freess, J.; Mroßko, M.: Gesamtheitlicher Entwicklungsansatz für Entwurf, Dokumentation und Bewertung von E/E-Architekturen, 12. Internationaler Kongress Elektronik im Kraftfahrzeug, Baden-Baden 2005

BIS09: Biskup, T.: Agile fachmodellgetriebene Softwareentwicklung für mittelständische IT-Projekte, Dissertation, Universität Oldenburg 2009

BOO95: G. Booch, J. Rumbaugh, Unified Method for Object-Oriented Development, 1995

BOS14: Bosch GmbH, Kraftfahrttechnisches Taschenbuch, 28. Auflage, Springer Vieweg, 2014

BRI10: Brinkschulte, U.; Ungerer, T.: Mikrocontroller und Mikroprozessoren, 3. Auflage, Springer Verlag, 2010

BRO98: Broy M.; von der Beeck, M.; Krüger: SOFTBED: Problemanalyse für ein Großverbundprojekt „Systemtechnik Automobil -Software für eingebettete Systeme"; Universität München, 1998

BRO10: Broy, J.: Modellbasierte Entwicklung und Optimierung flexibler zeitgesteuerter Architekturen im Fahrzeugserienbereich, Dissertation, Karlsruher Institut für Technologie, 2010

BUR06: Burgstahler, L.: An Enhanced Reference Point Method for the Evaluation of

Network Performance Aspects, Proceedings of the 4th Polish-German Teletraffic Symposium, PGTS 2006, Seiten 177–186, Wrocław, Polen, September 2006

BUR08: Burgstahler, L.: Bewertung von Mess- und Abschätzverfahren zur Unterstützung dienstgüteorientierter Verkehrslenkung in verbindungslosen Datennetzen, Dissertation, Universität Stuttgart 2008

DAIS13: Daimler AG, Die E/E-Architektur der Mercedes Benz S-Klasse 2013

DAV13: Davey C.: Automotive Software Systems Complexity: Challenges and Opportunities, INCOSE International Workshop, MBSE Workshop, Januar 2013

DD96: Das, I.; Dennis J.: Normal-Boundary Intersection: An alternate method for generating pareto optimal points in multicriteria optimization problems. NASA Langley Research Center 1996.

DD97: Das, I.; Dennis J.: A Closer Look at Drawbacks of MinimizingWeighted Sums of Objectives for Pareto Set Generation inMulticriteriaOptimization Problems. StructuralOptimization, 1997.

EAS04: The East-EEA Project: Definition of language for automotive embedded electronic architecture approach, Technischer Bericht, ITEA, Deliverable D3.6, 2004

EAS07: Brentsson et al, EAST ADL 2.0 Specification, 2007

ELY11: PREEvision 5.0 Demomodell „Elypsis“ vom 30.11.2011

FES07: Fesseler, Bertram: Entwicklung von ausgewählten Bewertungsmetriken auf einem E/E-Architektur-Modell, Diplomarbeit, Hochschule Ulm 2007

FIB09: ASAM: ASAM MCD-2 NET Data Model for ECU Network Systems (Field Bus Data Exchange Format), Version 3.1.0, 2009

FLO08: Florentz, Bastian: Software and System Architecture Evaluation and Analysis in the Automotive Domain, Dissertation, Technische Universität Braunschweig 2008

FOR07: Forbrig, P.: Objektorientierte Softwareentwicklung mit UML, Hanser Verlag, München 2007

FOW10: Fowler, M.; Parsons, R.: Domain-Specific Languages, Addison Wesley 2010

FREE07: Freess, Jascha: Modelle zur Beschreibung und Evaluierung von Architekturkonzepten der Elektrik und Elektronik in Kraftfahrzeugen, Dissertation, Universität Karlsruhe (TH) 2007

GIEG91: Giegrich, Jürgen, Ansätze zur Bewertung von Konzepten und Maßnahmen in der Abfallwirtschaft, 1991

GOF95: E. Gamma, R. Helm, R. E. Johnson, J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, 1995

GR07AB: Große-Rohde, Martin et al.: Architekturbewertung, ISST Bericht 82/07, Fraunhofer Institut Software- und Systemtechnik 2007

GR07RP: Große-Rohde, Martin et al.: Grobentwurf des VEIA-Referenzprozesses, ISST Bericht 80/07, Fraunhofer Institut Software- und Systemtechnik 2007

GRA09: Graf, Philipp: Entwurf eingebetteter Systeme: Ausführbare Modelle und Fehlersuche, Dissertation, Universität Karlsruhe (TH) 2009

GSCK06: Greenfield, J.; Short, K.; Cook, S.; Kent, S.: Software Factories: Moderne Software-Architekturen mit SOA, MDA, Patterns und agilen Methoden, Mitp-Verlag, 2006

HAR87: Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: Science of Computer Programming 8 (1987), S. 231#274

HAU04: Hauser J., Development of Highly complex Control Systems at BMW Group, 2004

JAC92: I. Jacobson, Object Oriented Software Engineering, 1992

JAEN12: Jaensch, M.: Modulorientiertes Produktlinien Engineering für den modellbasierten Elektrik/Elektronik-Architekturentwurf, Dissertation, Karlsruher Institut für Technologie (KIT), 2012

JECK00: Jeckle, M.: Konzepte der Metamodellierung – Zum Begriff Metamodell, 2000

JECK03: Jeckle M., C. Rupp, J. Hahn, B. Zengler, S. Queins, UML 2 glasklar, 2003

JECK04: Jeckle M.: Herkunft und Historie der UML, Vorlesung, Fachhochschule Furtwangen, Sommersemester 2004

KOH12: Kohl J.M.: Effiziente Diagnose von verteilten Funktionen automobiler Steuergeräte, Dissertation, Technische Universität München 2012

KUG12: S.M. Kugele: Model-Based Development of Software-intensive Automotive Systems, Dissertation, Technische Universität München 2012

LOU07: Louis, Dirk; Müller, Peter: Java 6 – Praxis der objektorientierten Programmierung, Markt+Technik Verlag, 2007

MAN08: Mann, Stefan: Anwendung von Kohäsions und Kohärenzmetriken auf VEIA Architekturmodelle zur Bewertung von Produktlinien, ISST Bericht 86/08, Fraunhofer Institut Software- und Systemtechnik 2008

MARTUT07: MARTE Tutorial - 13th SDL Forum, Sep 2007, Paris

MART11: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, Stand Juni 2011

MAT09: Matheis, Johannes: Abstraktionsebenenübergreifende Darstellung von Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262, Dissertation, Universität Karlsruhe (TH) 2009

POT96: Potel, Mike: MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java, Taligent Inc. 1996

OLI10: Oliveira, Washington A. et al.: A heuristic for the nonidentical circle packing problem, Publikation, CNMAC 2010 (XXXIII Congresso Nacional De Matemática Aplicada E Computacional)

OMG10DD: OMG, Diagram Definition; Version 1.0, 2012,

OMG231OCL:OMG, Object Constraint Language; Version 2.3.1, 2012,

OML: Firesmith, D.; Henderson-Sellers B.; Graham I., OPEN modeling language (OML) reference manual, Cambridge University Press New York, NY, USA, 1998

PAR10: Parker, David James: Multi-Objective Optimisation of Safety-Critical Hierarchical Systems, University of Hull (U.K.), Februar 2010

PET62: C.A. Petri, Kommunikation mit Automaten, Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn, 1962

POM97: Pomberger, G.; Rechenberg, P., Handbuch der Informatik, 1997

PRO01: ProSTEP-IVIP, AP 212 Electonical Design and Installation, VDA, 2001

QUA05: Quast, Johannes: Entwicklung und Realisierung eines Konzeptes zur generischen Verwendung modellbasierter Symboldefinitionen, Diplomarbeit, Forschungszentrum Informatik FZI / Universität Karlsruhe (TH), 2005

REI05: Reichmann, Clemens: Grafisch notierte Modell-zu-Modell-Transformationen für den Entwurf eingebetteter elektronischer System, Dissertation, Universität Karlsruhe (TH), 2005

REI12: Konrad Reif: Automobilelektronik - Eine Einführung für Ingenieure, 4., überarbeitete Auflage, Vieweg Teubner, 2012

RUM98: J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, 1998

SCH06: Scharnhorst, T.: Systemarchitektur Gesamtfahrzeug, AUTOUNI Wolfsburg, Carmeq GmbH, 2006

SPE10: The SPEEDS Project: SPEEDS L-1 Meta-Model, Deliverable D2.1.5 Revision 1.0.1, 2009

STA05 Stahl, Thomas ; Völter, Markus: Modellgetriebene Softwareentwicklung. dpunkt.verlag, 2005

STA06: Stahl, T.; Völter, M.; et. al.: Model-Driven Software Development: Technology, Engineering, Management, Wiley 2006

UML15: OMG, OMG Unified Modeling Language (OMG UML), Version 1.5, 2003

UML20: OMG, OMG Unified Modeling Language (OMG UML), Version 2.0, 2005

UML241Inf: OMG, OMG Unified Modeling Language (OMG UML), Infrastructure; Version 2.4.1, 20011

UML241Sup: OMG, OMG Unified Modeling Language (OMG UML), Superstructure; Version 2.4.1, 20011

VOS11: Vossen, Gottfried ; Witt, Kurt-Ulrich: Grundkurs theoretische Informatik. Eine anwendungsbezogene Einführung. Vieweg+Teubner, 2011

WEI10: Weiser, Dominic: Unterstützung der Verifikation und Analyse von benutzerdefinierten Metriken in einem Elektrik/Elektronik-Architekturbewertungswerkzeug, Hochschule Karlsruhe Technik und Wirtschaft 2010

WIE00: Wierzbicki Andrzej; Makowski, Marek; Wessels, Jaap: Model-Based Decision Support Methodology with Environmental Applications, Kluwer Academic publishers, Dordrecht, NL, 2000.

ZEL03: Zeller, Andreas: Causes and Effects in Computer Programs. AADEBUG'03: Proceedings of the sixth international symposium on Automated analysis-driven debugging, 2003

Internet-Quellen

ARVL: Institut für Programmierung und Reaktive Systeme, Universität zu Braunschweig; Homepage ArchiVal-Projekt: https://www.tu-braunschweig.de/ips/research/focuses/archival, zuletzt aufgerufen am 21.11.2012

ASAM: ASAM e.V.: http://www.asam.net/, zuletzt aufgerufen am 16.06.2011

ATE11: ATESST, http://www.atesst.net, zuletzt aufgerufen am 19.12.2011

ATEANA11: ATESST: Overview of Analysis-Driven Architecture and Optimization Concepts in EAST-ADL,

http://www.atesst.net/home/liblocal/docs/ConceptPresentations/13_EAST-ADL_AnalysisandOptimizationConcepts.pdf, zuletzt aufgerufen am 21.12.2011

AUD14: Audi zum Thema „Fahrerassistenzsysteme und pilotiertes Fahren“ auf der CES 2014 in Las Vegas, USA, https://www.audi-mediaservices.com/publish/ms/content/de/public/hintergrundberichte/2014/01/07/next_generation__/fahrerassistenzsysteme.html, zuletzt aufgerufen am 14.10.2014

AUT: Automotive Open System Architecture, http://www.autosar.org/, zuletzt aufgerufen am 28.07.2010

BMW14: BMW präsentiert ein autonom fahrendes Forschungsfahrzeug, „CHAUFFEUR INKLUSIVE: BMW ACTIVE ASSIST.“, http://www.bmw.de/de/topics/faszination-bmw/connecteddrive-2013/hochautomatisiertes-fahren.html, zuletzt aufgerufen am 14.10.2014

DAI: Daimler AG, http://www.daimler.de, zuletzt aufgerufen am 08.01.2014

DAI13: Daimler präsentiert ein autonom fahrendes Forschungsfahrzeug, „S 500 „INTELLIGENT DRIVE“ fährt autonom auf den Spuren von Bertha Benz“, http://www.daimler.com/dccom/0-5-1625161-49-1625165-1-0-0-1625162-0-0-135-7163-0-0-0-0-0-0-0.html, zuletzt aufgerufen am 14.10.2014

DAI14: Daimler zum Thema „Die Mobilität der Zukunft“, http://www.daimler.com/dccom/0-5-1742887-49-1743248-1-0-0-0-0-0-0-0-0-0-0-0-0-0.html, zuletzt aufgerufen am 14.10.2014

DFT25: Präsentation des Daimler Future Truck 2025, „Weltpremiere für den Transport der Zukunft“, http://www.daimler.com/dccom/0-5-1714412-49-1714446-1-0-0-1743248-0-0-135-0-0-0-0-0-0-0-0.html, zuletzt aufgerufen am 14.10.2014

EAA11: The EAST-ADL Association, http://www.east-adl.info, zuletzt aufgerufen am 19.12.2011

ECLI: Die Eclipse Foundation: https://www.eclipse.org/, zuletzt aufgerufen am 15.02.2014

EMF: The Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/, zuletzt aufgerufen am 08.09.2013

FRP7: Das 7. Forschungsrahmenprogramm: http://cordis.europa.eu/fp7/home_en.html, zuletzt aufgerufen am 27.01.2014

FZI: Forschungszentrum Informatik in Karlsruhe, www.fzi.de/, zuletzt aufgerufen am 05.08.2013

GEF: Eclipse Graphical Editing Framework, http://www.eclipse.org/gef/, zuletzt aufgerufen am 28.07.2010

GOO14: Google Self-Driving Car Project, https://plus.google.com/+GoogleSelfDrivingCars/posts, zuletzt aufgerufen am 14.10.2014

HER14: Herrtwich, R.G., Leiter Fahrassistenz- und Fahrwerksysteme in Konzernforschung und Vorentwicklung bei Daimler, in einem Interview am 25.09.2014 zum Thema „Autonomes Fahren wird stufenweise Realität“, http://www.daimler.com/dccom/0-5-1742887-49-1743293-1-0-0-1743248-0-0-135-0-0-0-0-0-0-0-0.html, zuletzt aufgerufen am 14.10.2014

HIP: HiP-HOPS, http://www.hip-hops.eu/, zuletzt aufgerufen am 26.01.2014

HIS: Herstellerinitiative Software: http://www.automotive-his.de/, zuletzt aufgerufen am 17.02.2011

HYDR: The best known packings of equal circles in a circle: http://hydra.nat.uni-magdeburg.de/packing/cci/cci.html, Universität Magdeburg, zuletzt aufgerufen am 12.08.2012

IEEE610: IEEE610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, https://standards.ieee.org/findstds/standard/610.12-1990.html, zuletzt aufgerufen am 16.02.2014

IEEE1061: IEEE1061, IEEE Standard for a Software Quality Metrics Methodology, http://standards.ieee.org/findstds/standard/1061-1998.html, zuletzt aufgerufen am 16.02.2014

IEEE1471: Systems and software engineering - Architecture description (auch bekannt als ISO/IEC 42010): http://www.iso-architecture.org/ieee-1471/defining-architecture.html, zuletzt aufgerufen am 09.01.2014

INC11: The International Council on Systems Engineering (INCOSE), http://www.incose.org/, zuletzt aufgerufen am 23.12.2011

ISO9000: ISO/IEC 9000:2005 Quality management systems -- Fundamentals and vocabulary: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=42180, zuletzt aufgerufen am 21.07.2014

ISO17458: ISO 17458-1..5:2013 Road vehicles -- FlexRay communications system: http://www.iso.org/iso/home/search.htm?qt=17458&published=on&active_tab=standards&sort_by=rel, zuletzt aufgerufen am 07.10.2014

ISO19505-1: ISO/IEC 19505-1:2012 Information technology -- Object Management Group Unified Modeling Language (OMG UML) -- Part 1: Infrastructure: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=32624, zuletzt aufgerufen am 20.11.2013

ISO19505-2: ISO/IEC 19505-2:2012 Information technology -- Object Management Group

Unified Modeling Language (OMG UML) -- Part 2: Superstructure: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=52854, zuletzt aufgerufen am 20.11.2013

ITIV: Institut für Technik der Informationsverarbeitung, http://www.itiv.kit.edu/, zuletzt aufgerufen am 08.01.2014

JAVA: Die JAVA-Programmiersprache, http://www.java.com/de/download/whatis_java.jsp, zuletzt aufgerufen am 15.02.2014

JAPI7: Java API-Dokumentation, Class Number, http://docs.oracle.com/javase/7/docs/api/java/lang/Number.html, zuletzt aufgerufen am 24.06.2013

JYT: Python Software Foundation: The Jython Project. http://www.jython.org, zuletzt aufgerufen am 05.08.2013

LQN: Layered Queueing Research Resource Page: http://www.layeredqueues.org/, zuletzt aufgerufen am 05.08.2013

MAE11: MAENAD, http://www.maenad.eu, zuletzt aufgerufen am 19.12.2011

MAEAW: MAENAD Analysis Workbench Presentation, http://www.maenad.eu/public_pw/conceptpresentations/MAENAD_Analysis_Workbench_2011.pdf, zuletzt aufgerufen am 27.01.2014

MAED5: MAENAD Analysis Workbench, Deliverable D5.2.1, Version 3.0, http://www.maenad.eu/public_pw/MAENAD_Deliverable_D5.2.1_V3.0.pdf, zuletzt aufgerufen am 26.01.2014

MAR99: K. Marczinzik, Petri-Netze und Fuzzy-Petri-Netze, https://kik.informatik.fh-dortmund.de/abschlussarbeiten/fuzzyPetriNetze/semadead.html, zuletzt aufgerufen am 05.03.2016

MARTE: UML MARTE: http://www.omgmarte.org/, zuletzt aufgerufen am 03.08.2013

MDA: Model Driven Architecture: http://www.omg.org/mda/specs.htm, zuletzt aufgerufen am 07.09.2013

MOF14: Der MOF 1.4 Standard, http://www.omg.org/spec/MOF/1.4/, zuletzt aufgerufen am 07.09.2013

MOF241: Der MOF 2.4.1 Standard, http://www.omg.org/spec/MOF/2.4.1/, zuletzt aufgerufen am 07.09.2013

OMG: Object Management Group, http://www.omg.org, zuletzt aufgerufen am 10.01.2014

OSGi: OSGi Alliance, http://www.osgi.org/, zuletzt aufgerufen am 28.11.2010

PAPY: Papyrus, http://www.eclipse.org/papyrus/, zuletzt aufgerufen am 27.01.2014

PREE: PREEvision, Vector Informatik GmbH: http://vector.com/vi_preevision_de.html, zuletzt aufgerufen am 03.01.2014

PROS: PROSTEP AG: http://www.prostep.com/, zuletzt aufgerufen am 17.02.2011

PYT: Python Software Foundation: The Python Programming Language. http://www.python.org/, zuletzt aufgerufen am 05.08.2013

RCP11: Eclipse Rich Client Platform, http://wiki.eclipse.org/index.php/Rich_Client_Platform, zuletzt aufgerufen am 23.12.2011

REQIF: The Requirement Interchange Format: http://www.omg.org/spec/ReqIF/, zuletzt aufgerufen am 17.02.2011

SIM: Mathworks Simulink, http://www.mathworks.de/products/simulink/, zuletzt aufgerufen am 26.01.2014

SPIN: Das SPIN-Software-Verifikationswerkzeug, http://spinroot.com/spin/whatispin.html, zuletzt aufgerufen am 26.01.2014

SPEE: The SPEEDS Project, http://www.speeds.eu.com/, zuletzt aufgerufen am 16.02.2011

SWT: The Standard Widget Toolkit, http://www.eclipse.org/swt/, zuletzt aufgerufen am 28.07.2010

SYSML: OMG Systems Modeling Language, http://www.omgsysml.org/, zuletzt aufgerufen am 22.12.2010

UML241: Der UML 2.4.1 Standard, http://www.omg.org/spec/UML/2.4.1/

UPP: Modellierung, Validierung und Verifikation vom Echtzeitsystemen, http://www.uppaal.org/, zuletzt aufgerufen am 26.01.2014

VEIA: Fraunhofer Institut Software- und Systemtechnik; Homepage VEIA-Projekt: http://veia.isst.fraunhofer.de/ zuletzt aufgerufen am 23.11.2012

W3CRIF: The W3C Rule Interchange Format, http://www.w3.org/TR/rif-overview/, zuletzt aufgerufen am 17.02.2011

W3CXHT: The W3C XHTML Standard: http://www.w3.org/TR/xhtml1/, zuletzt aufgerufen am 17.02.2011

WAR94: Dombacher, Christian: Warteschlangen; http://www.telecomm.at/documents/Warteschlangen.pdf, zuletzt aufgerufen am 05.08.2013

WIBK: Wikipedia Kriterium, http://de.wikipedia.org/wiki/Kriterium, zuletzt aufgerufen am 16.02.2014

WIJT11: Wikipedia Java (Technik), http://de.wikipedia.org/wiki/Java_(Technologie) , zuletzt aufgerufen am 08.02.2011

WIOPT10: Wikipedia Optimierung, http://de.wikipedia.org/wiki/Optimierung_%28Mathematik%29, zuletzt aufgerufen am XXX

WIOPTU14: Wikipedia Optimum, http://de.wikipedia.org/wiki/Optimum, zuletzt aufgerufen am 26.05.2014

WISCHA10: Wikipedia Schaltplan, http://de.wikipedia.org/wiki/Schaltplan, zuletzt aufgerufen am 27.07.2010

WISIG10: Wikipedia Signalflussdiagramm, http://de.wikipedia.org/wiki/Signalflussdiagramm, zuletzt aufgerufen am 27.07.2010

Eigene Veröffentlichungen

GMKMG09: Daniel Gebauer, Johannes Matheis, Markus Kühl, Klaus D. Müller-Glaser, „Integrierter, graphisch notierter Ansatz zur Bewertung von Elektrik/Elektronik-Architekturen im Fahrzeug“, 29. Tagung „Elektronik im Kraftfahrzeug“ vom Haus der Technik Essen e.V., Dresden 2009

GSKMG05: Daniel Gebauer, Barbara Scherrer, Markus Kühl, Klaus D. Müller-Glaser, „Verfahren zur Aufstellung formaler Metamodelle“, Kongreßbeitrag/Proceeding zu Design & Elektronik Entwicklerforum: Softwareentwicklung, München 2005

MGRMG08: Johannes Matheis, Daniel Gebauer, Clemens Reichmann, K. D. Müller-Glaser, „Ganzheitliche abstraktionsebenenübergreifende Beschreibung konsistenter Elektrik/Elektronik-Architekturen“, in Systems Engineering Infrastructure Conference (SEISCONF), München 2008

GMRMG08: Daniel Gebauer, Johannes Matheis, Clemens Reichmann, Klaus D. Müller-Glaser, Ebenenübergreifende, variantengerechte Beschreibung von Elektrik/Elektronik-Architekturen, Beitrag in Diagnose in mechatronischen Fahrzeugsystemen, Expert Verlag 2008

MGKRM06: Johannes Matheis, Daniel Gebauer, Markus Kühl, Clemens Reichmann, Klaus D. Müller-Glaser, Vorstellung einer Methodik zur E/E-Architekturmodellierung und -bewertung in der frühen Konzepthase, 26. Tagung „Elektronik im Kraftfahrzeug“ vom Haus der Technik Essen e.V., Dresden 2006

Betreute Studien- und Abschlussarbeiten

AKS15: Aksoy, Sinem: Anwendung von genetischen Algorithmen zur Optimierung der Platzierung von Splices innerhalb einer E/E-Architektur im Kraftfahr-zeug, Hochschule für Technik Stuttgart, 2015

BRE10: Bremer, Holger: Entwicklung und Realisierung einer Syntheseschritt Modellierung und deren automatischer Ausführung in E/E-Architekturen, Diplomarbeit, Universität Karlsruhe (TH) 2010

FES07: Fesseler, Bertram: Entwicklung von ausgewählten Bewertungsmetriken auf einem E/E-Architektur-Modell, Diplomarbeit, Hochschule Ulm 2007

HOM08: Hommel, Bernhard: Entwicklung graphisch notierter Konzepte für komplexe, modellbasierte Metrikberechnungen von E/E-Architekturen. Diplomarbeit, Universität Karlsruhe (TH) 2008

KIR14: Kirchner, Anja: Optimierungsalgorithmen im Projektmanagement, Masterthesis, Hochschule Reutlingen 2014

KOH06: Kohlbecker, Michael: Implementierung eines Werkzeugs zur Erstellung modellbasierter Bewertungsmetriken für Elektrik/Elektronik-Architekturen, Diplomarbeit, Universität Karlsruhe (TH) 2006

KOU06: Kouam, Prudence: Implementierung von Regelprüfungen für Elektrik/Elektronik-Architekturmodelle von Kraftfahrzeugen, Diplomarbeit, Universität Karlsruhe (TH), 2006

LEH06: Konzeption und Implementierung eines Moduls zur modellbasierten Konfiguration widgetbasierter Ansichten, Diplomarbeit, Hochschule Karlsruhe Technik und Wirtschaft

MAU15: Maugg, Gordian: Konzeption und Erweiterung der PREEvision REST-Schnittstelle um einen transaktionalen schreibenden Modellzugriff über mobile Plattformen, Masterthesis, Hochschule Mannheim, 2015

QUA05: Quast, Johannes: Entwicklung und Realisierung eines Konzeptes zur generischen Verwendung modellbasierter Symboldefinitionen, Diplomarbeit, Forschungszentrum Informatik FZI / Universität Karlsruhe (TH), 2005

SCH08: Schneider, Thibault: Analyse und Konzeption von Berechnungsvorschriften zur Parametrisierung der FlexRay-Bussysteme, Diplomarbeit, Universität Karlsruhe (TH) 2008

WIS16: Wisznewski, Niklas: Anwendungsspezifisches Benchmarking in einer 3-Tier Client-Server-Architektur, Hochschule Karlsruhe Technik und Wirtschaft, 2016