# Automatisierte Qualitätsbewertung am Beispiel von MATLAB Simulink-Modellen in der Automobil-Domäne

**Dissertation**

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inf. Jan Scheible
aus Göppingen

Tübingen
2012

| | |
|---|---|
| Tag der mündlichen Qualifikation: | 16.07.2012 |
| Dekan: | Prof. Dr. Wolfgang Rosenstiel |
| 1. Berichterstatter: | Prof. Dr. Herbert Klaeren |
| 2. Berichterstatter: | Prof. Dr. Wolfgang Rosenstiel |

# Zusammenfassung

In der Automobilindustrie wird in den letzten Jahren verstärkt auf die modellbasierte Softwareentwicklung von eingebetteten Systemen gesetzt. Die Modelle nehmen dabei den Stellenwert ein, den zuvor der handgeschriebene Quellcode eingenommen hat. Sie sind das zentrale Entwicklungsartefakt und werden über die Codegenerierung und Kompilierung direkt auf den eingebetteten Systemen ausgeführt. Der Umfang und die Komplexität der verwendeten Modelle steigen stetig. Das macht eine Qualitätssicherung der Modelle sehr aufwendig und teuer.

Diese Arbeit stellt ein Verfahren zur automatisierten Bewertung der Modellqualität vor. Auf Basis der Qualitätsbewertung werden Handlungsempfehlungen gegeben. Das Befolgen dieser Handlungsempfehlungen führt zu einer Steigerung der Modellqualität. Das entwickelte Verfahren erlaubt die Qualitätssicherung von großen und komplexen Modellen. In dieser Arbeit wird das Verfahren exemplarisch anhand von MATLAB Simulink-Modellen konzipiert und prototypisch umgesetzt.

Die Grundlage für die Definition der Modellqualität stellt ein Qualitätsmodell dar. Dieses Qualitätsmodell besteht aus verschiedenen Qualitätsaspekten und kann dadurch projektspezifisch angepasst werden. Das entwickelte Qualitätsmodell baut auf existierenden Modellen für die herkömmliche handcodierte Softwareentwicklung auf und wird in dieser Arbeit um Teile erweitert, die für die modellbasierte Entwicklung spezifisch sind. Das Qualitätsmodell betrachtet nicht nur die Simulink-Modelle selbst, sondern auch andere im modellbasierten Entwicklungsprozess beteiligten Artefakte. Das Qualitätsmodell hat die Struktur eines Baums und enthält auf unterster Ebene Metriken. Diese Metriken werden auf die Simulink-Modelle und die anderen relevanten Artefakte angewendet. Um zu einer Bewertung der Modellqualität zu gelangen, werden die Messwerte der Metriken bewertet und schließlich im Qualitätsmodell aggregiert. Die Aggregation verdichtet die Bewertungen, um einen einfachen Überblick zu erlauben. Die Bewertung wird mithilfe verschiedener Ansätze vorgenommen. Ein Referenzmodell definiert, wie ein typisches Simulink-Modell aussieht und Regeln geben vor, welche Bedingungen die Messwerte einhalten müssen. Zur Präsentation der Ergebnisse werden die Messwerte mit ihren erlaubten Grenzen und die aggregierten Bewertungen geeignet visualisiert. Aus der Qualitätsbewertung wird schließlich eine Handlungsempfehlung abgeleitet.

Anhand einer Fallstudie im PKW-Bereich der Daimler AG wird gezeigt, dass die Qualitätsbewertung zutreffend ist. Dies wird durch einen Vergleich der Qualitätsbewertung mit Einschätzungen von Experten erreicht.

# Abstract

The automotive industry has been moving towards model-based software development of embedded systems in recent years. Now models are the main artifacts, a position which was previously held by source code. Source code is generated from the models using a code generator and as such, source code is losing its importance within the development cycle. Size and complexity of the models rise steadily which makes a quality rating of models very complex and costly.

This work presents a method for an automated rating of model quality. Based on the result of the rating recommendations for quality improvements are given. The developed method allows rating the quality of large and complex models. In this work a prototype for rating MATLAB Simulink models is developed.

A quality model is used to define the notion of model quality. This quality model consists of several quality aspects and can be adapted for each project. The quality model developed in this work enriches existing models for traditional hand-coded software by parts that are specific to the model-based development. The quality model addresses not only the quality of the Simulink models themselves but also of other artifacts which are involved in the development process. The quality model has a tree structure and contains metrics at its lowest level. These metrics are applied to the Simulink models and other relevant artifacts. For getting a final rating of a model's quality, the measured values are evaluated first. Then, following the tree structure of the quality model, the metric's evaluations are aggregated. This allows an easy overview of a model's quality. The evaluation of the measurements is performed using different approaches: A reference model defines how a typical Simulink model looks like and rules make statements regarding the desired properties of the measurements. For the presentation of quality ratings the measurements and the aggregated ratings are displayed in a compact manner. Finally recommendations for improvements are derived from the quality rating.

The correctness of the presented approach is validated in a case study in the passenger cars department of Daimler AG. This is achieved by comparing quality ratings with quality estimates from experts.

# Danksagung

Die vorliegende Dissertation entstand im Zeitraum von 03/2009 bis 03/2012 während meiner Tätigkeit als Doktorand im Bereich Forschung und Vorentwicklung der Daimler AG. Sie wurde bei der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen als Dissertation eingereicht.

Mein besonderer Dank gilt meinem Doktorvater Herrn Prof. Dr. Herbert Klaeren, der stets großes Interesse an dem Thema meiner Dissertation zeigte. Unsere angeregten Diskussionen, die sowohl am Institut für Programmiersprachen und Übersetzer in Tübingen als auch im Daimler Forschungszentrum in Ulm stattfanden, lieferten viele wichtige Impulse für das Gelingen meiner Arbeit. Herrn Prof. Dr. Wolfgang Rosenstiel danke ich für das Interesse an meiner Arbeit und der Übernahme des Zweitgutachtens.

Ein weiterer besonderer Dank geht an meinen Betreuer und Teamleiter Dr. Ingo Kreuz bei der Daimler AG. Er sorgte während der letzten drei Jahre stets für ein Umfeld, welches mir die Erstellung dieser Arbeit überhaupt erst ermöglichte. Außerdem stand er mir immer als begeisterter Diskussionspartner zur Verfügung und half mir mit vielen guten Ideen und seiner konstruktiven Kritik bei allen Aspekten der Erstellung dieser Arbeit. Auch möchte ich mich bei Dr. Rainer König bedanken, der immer größten Wert darauf legte, dass wir Doktoranden in seiner Abteilung geschätzt und gefördert wurden. Des Weiteren möchte ich mich bei allen Mitgliedern des Teams Modellbasierte Entwicklung und der ganzen Abteilung GR/PSS für die vielen konstruktiven Diskussionen und die schöne Zeit bedanken. Besonders hervorheben möchte ich Dr. Ernst Richter, der sich stets für Diskussionen aller Art Zeit nahm und mir immer mit Rat und Tat zur Seite stand.

Außerdem möchte ich mich bei Dr. Hartmut Pohlheim von MES für die vielen interessanten Diskussionen und unsere gemeinsame Veröffentlichung bedanken. Eine unschätzbare Hilfe bei der prototypischen Implementierung des in dieser Arbeit vorgestellten Verfahrens war Wolfgang Holoch. Er leistete während seiner Masterarbeit und Werkstudententätigkeit eine großartige Arbeit bei der Implementierung des Modell-Exporters und der Modellmetriken seiner Masterarbeit. Eine wertvolle Hilfe waren mir außerdem Christian Dziobek und Dr. Florian Wohlgemuth aus der PKW-Entwicklung der Daimler AG. Sie hatten immer ein offenes Ohr für mich und gaben mir wertvolle Einblicke in ihr Tagesgeschäft und in die „richtig interessanten“ Problemstellungen der modellbasierten Entwicklung mit Simulink.

Mein herzlicher Dank gilt auch meiner Frau Silke für stundenlanges Korrekturlesen und wertvolles Feedback. Meinen Eltern Gerda und Martin, die mich stets bei meinem Werdegang unterstützen, möchte ich ebenfalls danken. Vor allem danke ich meiner Mutter für die Bewältigung der schier endlos erscheinenden Textmengen bei der Rechtschreibkorrektur dieser Dissertation. Zuletzt möchte ich mich bei allen anderen Familienmitgliedern und Freunden bedanken. Jeder von euch hat auf seine ganz eigene Weise am Gelingen dieser Arbeit mitgewirkt.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Metrikenverzeichnis

# 1. Einleitung

In diesem Kapitel wird zuerst die Motivation für diese Arbeit vorgestellt und dann die Zielsetzung genauer beschrieben. Anschließend wird auf den Lösungsansatz eingegangen und auf die Abschnitte verwiesen, in denen die einzelnen Aspekte im Detail beschrieben werden. Zuletzt wird die Arbeit relativ zu anderen Ansätzen in dem Themenbereich eingeordnet.

## 1.1. Motivation

Durch die modellbasierte Softwareentwicklung eingebetteter Systeme kann man im Vergleich zur Entwicklung mit textuellen Programmiersprachen unter anderem eine höhere Abstraktion und letztendlich eine höhere Softwarequalität erreichen [SVEH07, S. 16 ff.]. Die Modelle stellen bei der modellbasierten Entwicklung das zentrale Artefakt dar. Das bedeutet, dass die Modellqualität direkten Einfluss auf die Softwarequalität hat [FHR08]. Das heißt aber auch, dass durch die Qualitätssicherung von Modellen eine frühzeitige Absicherung erreicht werden kann: Fehler werden bereits im frühen Entwicklungsstadium erkannt und können noch kostengünstig behoben werden [FLS01]. Außerdem ist es im Automobilbau sehr wichtig, die Qualität eines Softwarestandes zu kennen, bevor er als Seriencode freigegeben werden kann, da es sich um eine sicherheitsrelevante Domäne handelt.

Der Umfang der Modelle steigt stetig. So hat momentan ein großes Modell typischerweise ca. 15.000 Blöcke, 700 Subsysteme und eine Subsystemhierarchie mit 16 Ebenen. Außerdem geben Werkzeuge wie MATLAB Simulink den Modellierern sehr viele Freiheiten. So wird beispielsweise in [CD06] vorgeschlagen, den erlaubten Sprachumfang für sicherheitsrelevante Systeme einzuschränken. Sowohl die großen Umfänge als auch die Freiheiten der Modellierer, die sich aus der herkömmlichen und zusätzlich aus der modellbasierten Softwareentwicklung ergeben, führen zu vielen potentiellen Fehlerquellen.

Die höhere Abstraktion unterstützt die Software-Entwicklung zwar vor allem bezüglich nichtfunktionaler Qualitätsmerkmale, weil Strukturiertheit, einheitliche Architektur, Wiederverwendbarkeit, Lesbarkeit und Übersichtlichkeit durch die grafische Notation gefördert werden. Allerdings gibt die grafische Notation allein noch keine Garantie dafür, dass hohe Softwarequalität erreicht wird. So kann beispielsweise trotz grafischer Repräsentation eine ungünstige Architektur gewählt werden, Subsysteme falsch aufgeteilt werden oder Blöcke gewählt werden, die bei der Codegenerierung zu ineffizientem Code führen.

Abbildung 1.1 und Abbildung 1.2 zeigen jeweils eine schematische Darstellung eines Simulink-Modells. Simulink-Modelle sind Blockdiagramme mit einer kombinierten Daten- und Kontrollflusssemantik. Die Daten- und Kontrollflüsse werden mittels Linien dargestellt. In den Blöcken findet eine Verarbeitung der Daten- und Kontrollflüsse statt. Blöcke können beliebig viele Ein- und Ausgänge besitzen. Die in den Abbildungen grün und blau dargestellten Blöcke sind spezielle Blöcke, die ein- oder ausgehende Signale in ein Subsystem darstellen. Ein- und ausgehende Signale gibt es auf der Ebene jedes Subsystems. Die Subsysteme, deren Namen in den Abbildungen grau dargestellt sind, dienen zur hierarchischen Aufteilung der Modelle

in handhabbare Einheiten. Da ein Subsystem wieder ein Subsystem enthalten kann, entsteht eine baumartige Subsystemhierarchie.

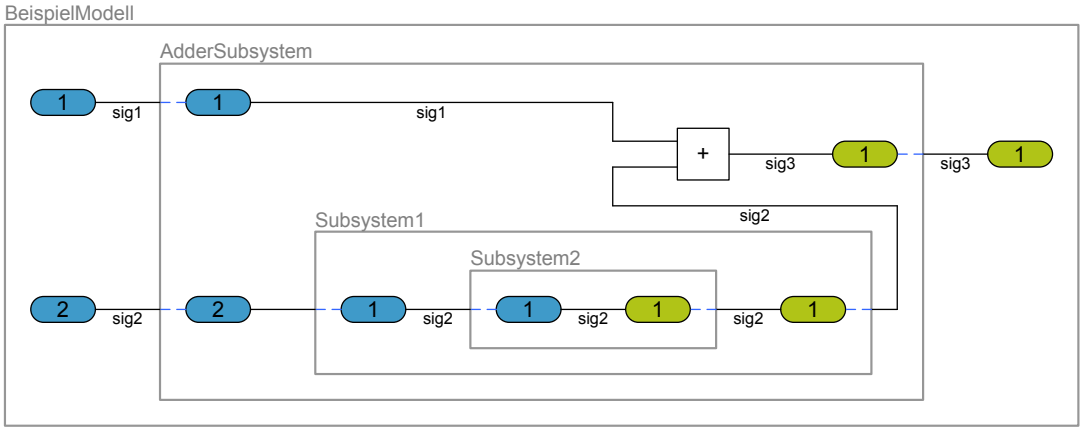



Abbildung 1.1.: Beispiel eines nicht optimalen Simulink-Modells

Die Modelle in Abbildung 1.1 und Abbildung 1.2 liefern bei gleicher Eingabe die gleiche Ausgabe. Es werden jeweils zwei Eingangssignale addiert und das Ergebnis ausgegeben. Allerdings unterscheiden sich die Modelle bezüglich ihrer Wartbarkeit und Verständlichkeit. Das Modell in Abbildung 1.2 verwendet eine minimale Anzahl an Blöcken und Signalen, um die gleiche Funktionalität wie in Abbildung 1.1 umzusetzen. Wohingegen in dem Modell in Abbildung 1.1 unnötig viele Blöcke und Subsysteme verwendet werden. Dadurch wird die Wartbarkeit und Verständlichkeit des Modells verschlechtert.

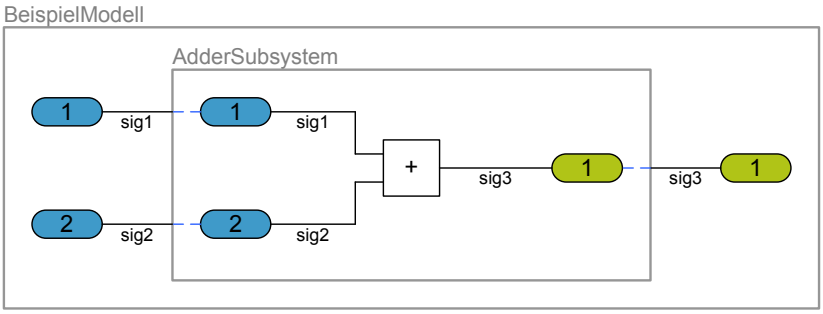



Abbildung 1.2.: Beispiel eines optimalen Simulink-Modells

Für die Ausführung der Modelle auf einem Steuergerät wird im Falle der modellbasierten Softwareentwicklung mit MATLAB Simulink zuerst mittels eines Codegenerators C-Code generiert, der anschließend von einem Compiler in Binärcode für den Zielprozessor übersetzt wird (siehe Abbildung 1.3). Zwar wird der Codegenerator die unnötigen Teile mit hoher Wahrscheinlichkeit entfernen, aber da bei der modellbasierten Entwicklung das Modell das primäre Arbeitsartefakt ist, muss jede Änderung an dem schwer zu verstehenden Modell vorgenommen werden.

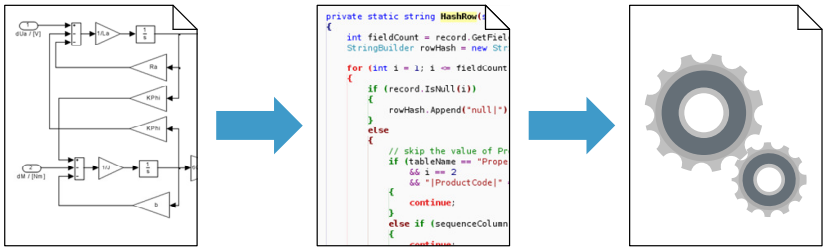


Abbildung 1.3.: Schematischer Überblick über die Code-Generierung

Somit kann man sagen, dass das Modell aus Abbildung 1.1 eine niedrigere Modellqualität als das Modell aus Abbildung 1.2 hat. Daher ist es wünschenswert, (Qualitäts-)Faktoren mit einem negativen Einfluss auf die Modellqualität erkennen zu können.

Diese Erkennung kann in Abbildung 1.1 beispielsweise durch das Zählen der Nutzblöcke pro Subsystem erreicht werden. Nutzblöcke sind in diesem Beispiel Blöcke, welche Signale verändern. So hat das erste Modell nur $0,3$ Nutzblöcke pro Subsystem, wobei das zweite Modell 1 Nutzblock pro Subsystem hat. Des Weiteren kann man die maximale Länge der Signalpfade ohne Änderungen zählen. Signalpfade ohne Änderungen sind Signalpfade zwischen Blöcken, die keine Nutzblöcke sind. Im ersten Beispiel hat das Signal sig2 6 Segmente und im zweiten Beispiel nur 2 Segmente. Dieses Beispiel illustriert, wie mithilfe von Messwerten und Qualitätsfaktoren darauf geschlossen werden kann, dass das zweite Modell eine höhere Qualität als das erste Modell hat.

## 1.2. Zielsetzung

Die Zielsetzung dieser Arbeit ist die Entwicklung eines Verfahrens zur automatisierten Qualitätsbewertung von MATLAB Simulink-Modellen, welche zur modellbasierten Softwareentwicklung von Steuergeräten in der Automobilindustrie verwendet werden. Auf Basis der Qualitätsbewertung sollen Handlungsempfehlungen abgeleitet werden, die als Ziel die Beseitigung der erkannten Probleme haben und somit zu einer Erhöhung der Modellqualität führen.

Es werden im Allgemeinen zwei Arten von Simulink-Modellen unterschieden: Simulationsmodelle und Funktionsmodelle. Simulationsmodelle können nur simuliert werden, Funktionsmodelle können simuliert und codegeneriert werden. In dieser Arbeit werden primär Funktionsmodelle untersucht, bei denen in jedem Modell eine unabhängige, in sich geschlossene Funktion modelliert ist. Im Fahrzeug werden eine oder mehrere dieser Funktionen auf einem Steuergerät ausgeführt. Reine Simulationsmodelle können mit Abstrichen mit den gleichen Methoden behandelt werden.

Im Falle der modellbasierten Entwicklung kann durch die Bewertung der Modellqualität direkt auf die Softwarequalität geschlossen werden [FHR08], da der C-Code automatisch vom Code-Generator erzeugt wird. Der generierte C-Code kann dabei über zahlreiche Einstellungen in den Modellen beeinflusst werden.

Abbildung 1.4 zeigt einen schematischen Überblick über das zu entwickelnde Verfahren. Die Eingabe sollen die Simulink-Modelle selbst und andere für die modellbasierte Entwicklung relevanten Artefakte sein. Die Ausgabe ist eine Modellqualitätsbewertung für das zu untersuchende Simulink-Modell. Die Simulink-Modelle sollen automatisiert und ohne Ausführung

auf dem Zielsteuergerät untersucht werden. Anhand der Untersuchungsergebnisse soll auf objektive Art und Weise bewertet werden, wie hoch die Qualität der Simulink-Modelle ist.

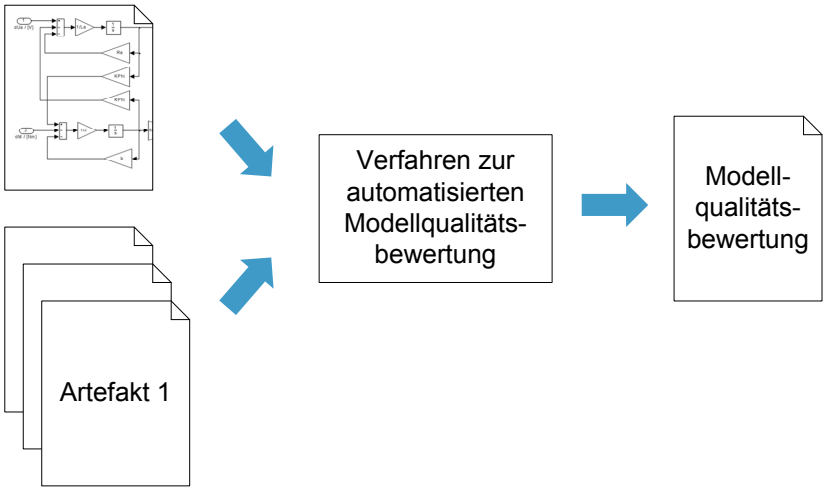



Abbildung 1.4.: Schematischer Überblick über das zu entwickelnde Verfahren

**Anwendungsfälle** Die Modellqualitätsbewertung soll für folgende Zielgruppen und deren Anwendungsfälle verwendbar sein:

**Modellierer** Ein Modellierer soll die Qualitätsbewertung verwenden können, um bei einer schlechten Bewertung Verbesserungspotential in seinem Modell zu identifizieren.

**Einkauf** Im Einkauf soll die Qualitätsbewertung für eine Unterstützung der Eingangskontrolle verwendet werden. Bei einer schlechten Bewertung können Modelle gereviewt werden und im Falle von zu vielen Auffälligkeiten zurückgewiesen und eine Nachbesserung verlangt werden.

**Manager** Einen Manager soll die Qualitätsbewertung bei der Freigabeentscheidung unterstützen.

### Anforderungen

**[A1]** Da die Modelle typischerweise sehr groß sind, soll das Verfahren automatisierbar sein.

**[A2]** Die prototypische Umsetzung soll anhand von MATLAB Simulink erfolgen. Somit soll auch die Semantik der Modelle berücksichtigt werden.

**[A3]** Die Qualitätsbewertung soll nachvollziehbar sein, das heißt, es soll immer ersichtlich sein, wie das Ergebnis zustande kommt.

**[A4]** Die Qualitätsbewertung soll wiederholbar sein, das heißt, sie muss bei wiederholter Ausführung zu dem gleichen Ergebnis kommen.

**[A5]** Die Qualitätsbewertung soll erweiterbar und anpassbar sein, da in unterschiedlichen Projekten unterschiedliche Richtlinien befolgt werden müssen.

**[A6]** Die Qualitätsbewertung soll ohne die Zielhardware möglich sein.

**[A7]** Die Modelle sollen durch die Qualitätsbewertung nicht verändert werden. Änderungen werden nur vom Modellierer auf Basis von Handlungsempfehlungen durchgeführt.

**Fragestellungen** Aus den Anwendungsfällen und den Anforderungen ergeben sich folgende zentrale Fragestellungen:

- Welche Artefakte müssen zusätzlich zum Simulink-Modell untersucht werden?
- Wie kann die Modellqualität auf eine objektive Art und Weise definiert werden?
- Wie kann die Qualität eines Modells bewertet werden?
- Wie kann die Qualität von Modellen mit niedriger Qualität erhöht werden?

Zusammenfassend ist das Ziel der Arbeit die Entwicklung eines Verfahrens zur Modellqualitätsbewertung und die Untersuchung, ob diese Qualitätsbewertung zutreffend ist.

## 1.3. Lösungsansatz und Übersicht

Im Folgenden wird ein Überblick über den Lösungsansatz gegeben, der in der Arbeit verwendet wird. Das Ziel ist die Entwicklung und prototypische Umsetzung eines Verfahrens zur automatisierten Qualitätsbewertung von Simulink-Modellen, welches die Anforderungen aus Abschnitt 1.2 erfüllt und die dort gestellten zentralen Fragestellungen beantwortet.

In diesem Ansatz wird ein Qualitätsmodell mit Metriken verwendet. Das Qualitätsmodell stellt eine abstrahierte Sicht auf die Simulink-Modelle dar. Mithilfe dieser abstrahierten Sicht können Aussagen über die Modellqualität gemacht werden. Der Aufbau des verwendeten Qualitätsmodells und dessen Metriken werden in Kapitel 4 und 3 im Detail beschrieben. Die Beschreibung umfasst sowohl die Herleitung des Qualitätsmodells als auch die Begründung seines Aufbaus.

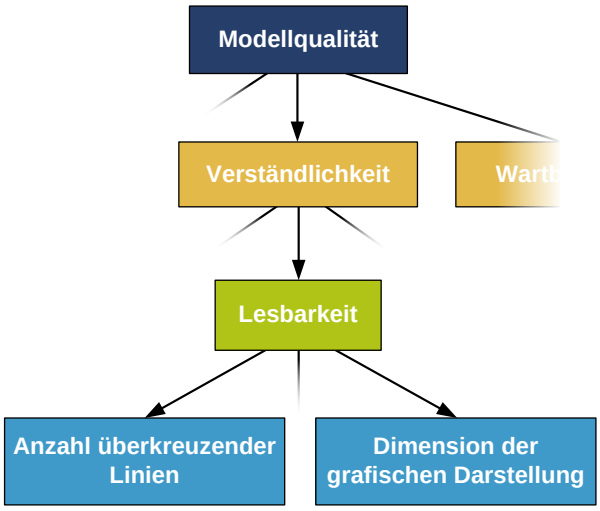



Abbildung 1.5.: Aufbau des Qualitätsmodells

Die Struktur und die Benennung der einzelnen Elemente des Qualitätsmodells sind von Cavano und McCall übernommen [CM78]. Die Definition der Modellqualität wird durch sogenannte Qualitätskriterien formalisiert. Diese Kriterien werden dann wiederum zu Qualitätsfaktoren zusammengefasst. Die Faktoren erlauben es, eine Aussage beispielsweise über die Wartbarkeit oder die Verständlichkeit eines Modells zu machen. So wird z. B. der Faktor Verständlichkeit in Abbildung 1.5 unter anderem durch das Kriterium Lesbarkeit verfeinert.

Die Lesbarkeit eines Simulink-Modells wird wiederum bewertet, indem die Anzahl der überkreuzenden Linien gezählt und die Dimension der grafischen Darstellung in Pixel geprüft wird.

Abbildung 1.6 zeigt die zentralen Schritte des Verfahrens. Jeder dieser Schritte wird im Folgenden kurz beschrieben und es wird auf das Kapitel bzw. den Abschnitt verwiesen, in dem der Schritt im Detail erläutert wird.

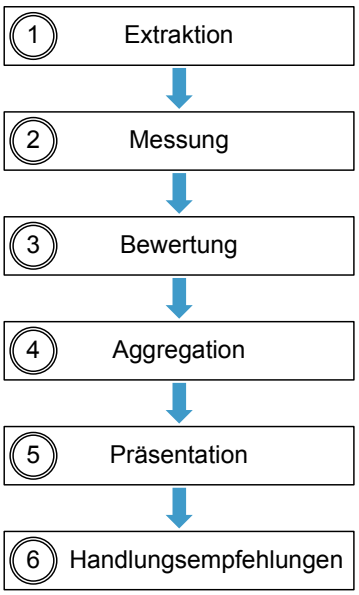



Abbildung 1.6.: Schritte zur Modellqualitätsbewertung

**Extraktion** Im ersten Schritt muss auf die Modelle zugegriffen werden, um Informationen extrahieren zu können. In Rahmen der Implementierung des Verfahrens in Kapitel 6 wird in Abschnitt 6.1 beschrieben, wie mithilfe eines Meta-Modells auf die Simulink-Modelle zugegriffen werden kann.

**Messung** Dieser Zugriff auf die Simulink-Modelle wird für die Ausführung der Metriken verwendet. Ein allgemeiner Überblick über Metriken findet sich in Abschnitt 3.1 und eine Vorstellung der Modellmetriken, die die Semantik von Simulink-Modellen berücksichtigen, findet sich ab Abschnitt 3.3. Die in dem Ansatz verwendeten Metriken sind dabei primär Modellmetriken, können aber auch Informationen aus beliebigen anderen Artefakten des modellbasierten Entwicklungsprozesses verwenden. Der modellbasierte Entwicklungsprozess und dessen Artefakte sind in Abschnitt 2.1 beschrieben.

**Bewertung** Die Erfüllung der Qualitätskriterien des Qualitätsmodells (siehe Kapitel 4) wird mittels Metriken geprüft. Nach dem Auswerten der Metriken müssen deren Messwerte bewertet werden und somit eine Aussage getroffen werden, ob ein Messwert in Ordnung ist oder nicht. Die Aussage zur Bewertung wird entweder mittels eines Referenz-Modells getroffen, das vorgibt, wie ein typisches Simulink-Modell aussieht (siehe Abschnitt 5.1.1). Oder es kann eine Aussage über die Gültigkeit eines Messwerts mittels einer Regel (siehe Abschnitt 5.1.2) getroffen werden.

**Aggregation** Nach Bewertung der Metriken muss dann auf die Qualitätskriterien und Qualitätsfaktoren aggregiert werden. Diese Aggregation wird in Abschnitt 5.2 beschrieben.

**Präsentation** Nach der Erstellung der Modellqualitätsbewertung wird das Ergebnis präsentiert. Die Visualisierung wird in Abschnitt 5.2.1 beschrieben.

**Handlungsempfehlungen** Aus der Modellqualitätsbewertung werden in Abschnitt 5.3 Handlungsempfehlungen abgeleitet. Sie geben Hinweise darauf, wie mithilfe der Ergebnisse der Modellqualitätsbewertung die Qualität der Modelle durch den Modellierer erhöht werden kann. Im Sinne einer kontinuierlichen Integration kann die Qualitätsbewertung über die Zeit betrachtet werden, um eine Verbesserung oder Verschlechterung der Modellqualität aufzuzeigen. Das Nachvollziehen der Bewertung über die Zeit wird in Abschnitt 5.4 beschrieben.

Der implementierte Prototyp aus Kapitel 6 wird in Kapitel 7 für die Evaluation des Ansatzes verwendet. Dazu werden mehrere Modelle aus dem PKW-Bereich der Daimler AG mit der Qualitätsbewertung bewertet und nach ihrer Bewertung sortiert. Das Ergebnis der Bewertung wird mit den Einschätzungen von Experten verglichen.

Abschließend wird in Kapitel 8 zuerst eine Zusammenfassung des Verfahrens zur automatisierten Modellqualitätsbewertung gegeben. Es folgt ein Ausblick auf offene Fragestellungen und mögliche Erweiterungen des Ansatzes.

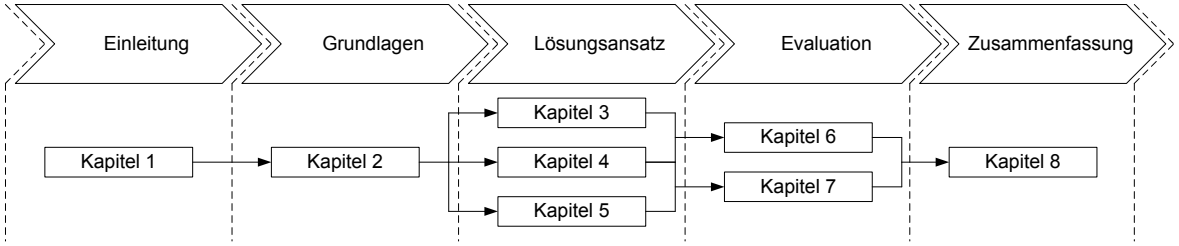



Abbildung 1.7.: Gliederung der Arbeit

Abbildung 1.7 zeigt nochmals zusammenfassend die Gliederung der Arbeit. Nach der Einleitung in Kapitel 1, in der auch in Abschnitt 1.4 die Einordnung in die Literatur vorgenommen wird, werden in Kapitel 2 die Grundlagen gelegt. Dazu wird in Abschnitt 2.1 zuerst auf den modellbasierten Entwicklungsprozess im Allgemeinen eingegangen. Abschnitt 2.2 beschreibt dann die modellbasierte Entwicklung mit MATLAB Simulink im Speziellen. In den Kapiteln 3, 4 und 5 werden das Qualitätsmodell, die Metriken und die darauf basierende Modellqualitätsbewertung vorgestellt. Danach werden in Kapitel 6 und Kapitel 7 der Prototyp und die Evaluation des Ansatzes beschrieben. Kapitel 8 gibt abschließend eine Zusammenfassung über den Inhalt der Arbeit.

## 1.4. Einordnung in die Literatur

In diesem Abschnitt wird das Verfahren zur automatisierten Modellqualitätsbewertung relativ zu anderen Ansätzen eingeordnet. Auf die Metriken für Simulink-Modelle wird in diesem Abschnitt nicht eingegangen, da in Kapitel 3 bei den jeweiligen Metriken auf eventuell bereits bestehende Metriken verwiesen wird. Des Weiteren wird eine detaillierte Gegenüberstellung des ursprünglichen Qualitätsmodells von Cavano und McCall mit dem Qualitätsmodell dieser Arbeit in Kapitel 4 vorgenommen.

Zuerst wird in Abschnitt 1.4.1 auf Ansätze eingegangen, die sich direkt mit der Modellqualität von MATLAB Simulink-Modellen beschäftigen. Ab Abschnitt 1.4.2 werden weitere Arbeiten vorgestellt, die mit dem Ansatz dieser Arbeit verwandt sind. Dazu wird zuerst in Abschnitt

1.4.2 auf andere Qualitätsmodelle eingegangen. Außerdem gibt es Normen und Standards, die sich mit dem Begriff der Qualität beschäftigen. Auch sie definieren unter anderem Qualitätsmodelle. In Abschnitt 1.4.3 wird auf Arbeiten zum Thema Qualitätsbewertung von Modellen eingegangen und mögliche Ansätze für Transfers beleuchtet.

### 1.4.1. Qualitätssicherung von Simulink-Modellen

Abbildung 1.8 gibt einen Überblick über bestehende Ansätze zur Qualitätssicherung von Simulink-Modellen. Die Ansätze werden in Bezug auf die Kriterien *konstruktiv*, *analytisch*, *automatisiert* und *manuell* eingeordnet. Konstruktive Ansätze verändern das Simulink-Modell auf eine Art und Weise, damit die Qualität höher wird. Analytische Ansätze untersuchen das Modell und geben Hinweise auf Probleme, verändern das Modell aber nicht. Automatisierte Ansätze ändern bzw. untersuchen das Modell automatisiert. Allerdings bedeutet eine automatisierte Durchführung nicht, dass keine manuelle Vorarbeit notwendig ist, zum Beispiel Festlegung von Konfigurationseinstellungen oder Erstellung von Testfällen. Vielmehr ist nach einem initialen Aufwand eine wiederholte automatisierte Ausführung möglich. Manuelle Verfahren sind nicht automatisiert und müssen immer vom Modellierer durchgeführt werden.

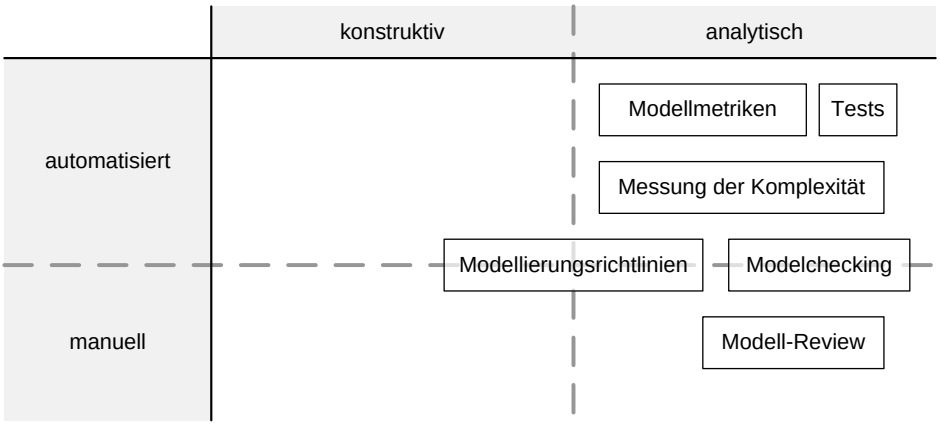



Abbildung 1.8.: Einordnung der verwandten Ansätze

**Modellierungsrichtlinien** Modellierungsrichtlinien geben Regeln vor, die bei der Modellierung eingehalten werden müssen (siehe [Con+05], [LRS07], [SDP08] und [Stü+07]). Ein Teil der Regeln kann automatisiert geprüft werden. Jede Regel muss eine Vorschrift enthalten, wie ein Verstoß gegen die Regel beseitigt werden kann. Ein Teil der Regelverstöße kann automatisiert behoben werden. Beispiele für Modellierungsrichtlinien sind die Vorgabe eines Farbschemas für die Blöcke oder eine Vorschrift über die Platzierung der Ein- und Ausgangsblöcke. Für die Prüfung von Modellierungsrichtlinien kann beispielsweise das Tool MXAM der Firma MES[1] (vergleiche [SDP08]) verwendet werden.

**Modellmetriken** Modellmetriken bilden Eigenschaften von Simulink-Modellen auf Messwerte ab. Bisher wurden nur einige generische Modellmetriken (zum Beispiel Anzahl der Blöcke oder Anzahl der Subsysteme) für Simulink-Modelle vorgestellt [Rau01]. In [MA05] verwenden Menkhaus und Andrich zwar eine Metrik zum Finden eines Subsystems in Simulink-Modellen für die FMEA, allerdings ist diese Metrik auch generisch und bezieht

[1] `http://www.model-examiner.com`

sich nur auf die Anzahl der Ein- und Ausgänge eines Subsystemes. Für handcodierte Software werden hingegen seit vielen Jahren Softwaremetriken verwendet. Bekannte Beispiele sind beispielsweise Lines of Code oder die zyklomatische Komplexität von McCabe [McC76].

**Messung der Komplexität** Ähnlich wie beim Quellcode kann die Komplexität der Modelle berechnet werden. Beispielsweise verwendet das Tool MXRAY von der Firma MES[2] (vergleiche [SPR10]) einen für Simulink-Modelle angepassten Halstead-Algorithmus [Hal77].

**Modelchecking** Erlaubt das Überprüfen von Bedingungen in Simulink-Modellen. So kann beispielsweise für Signale geprüft werden, ob sie ihre erlaubten Wertebereiche einhalten. Für Zustandsautomaten kann beispielsweise geprüft werden, ob bestimmte Zustände erreicht werden können. Diese Prüfung erfolgt automatisch. Allerdings müssen die Bedingungen zuvor von Hand formalisiert werden (siehe beispielsweise [EFJ10] und [LL08]).

**Manuelle Modell-Reviews** Ein manuelles Modell-Review ist eine strukturierte Durchsicht eine Modells. Bei dem Review wird ein Review-Protokoll erstellt, welches alle Auffälligkeiten dokumentiert. Auffälligkeiten sind beispielsweise Verstöße gegen die Modellierungsrichtlinien, unnötig komplexe bzw. schwer verständliche Subsysteme oder funktionale Fehler, das heißt nicht oder falsch umgesetzte Anforderungen (siehe [Con+05]).

**Tests** Beim Testen der Simulink-Modelle werden Testfälle ausgeführt. Jeder Testfall kann entweder erfolgreich sein oder fehlschlagen. Die Ausführung der Testfälle ist automatisierbar. Das Erstellen der Testfälle hingegen ist eine manuelle Aufgabe. Aus den Anforderungen werden Testspezifikationen abgeleitet, die wiederum in Testimplementierungen umgesetzt werden. Diese Testimplementierungen werden dann ausgeführt (siehe [CDFY99], [CDSS02] und [CFS04]).

Die Modellqualitätsbewertung in dieser Arbeit hingegen ist ein übergreifender Ansatz. Es wird ein Qualitätsmodell (vergleiche Abschnitt 1.4.2) verwendet, welches erlaubt, eine Aussage zu treffen, ob ein Modell eine hohe Qualität hat.

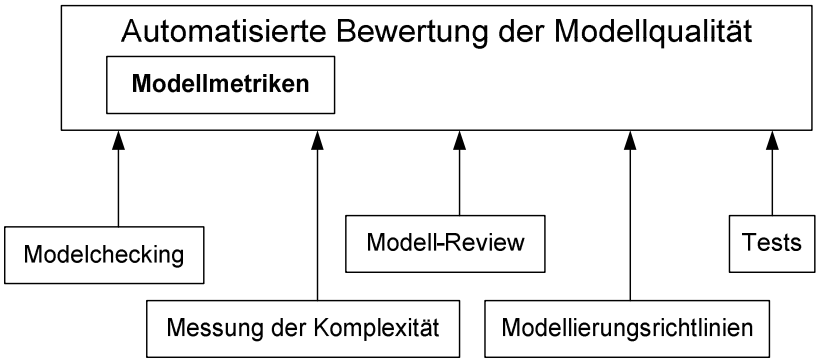



Abbildung 1.9.: Schematische Übersicht über die Einordnung der Ansätze

Dazu werden Metriken und Informationen aus den zuvor vorgestellten Ansätzen verwendet. Abbildung 1.9 zeigt eine schematische Übersicht über die Einordnung der Ansätze. Da es sich bei der Modellqualitätsbewertung um einen automatisierten Ansatz handelt, werden bei

[2] `http://www.m-xray.com`

den in Abbildung 1.8 als teil-automatisch oder manuell eingeordneten Ansätzen nur Informationen aus den Ergebnissen verwendet, wenn diese zuvor erzeugt wurden. Im Beispiel der Modellierungsrichtlinien wird der erstellte Report analysiert. Zusammenfassend baut diese Arbeit auf den Arbeiten zum Thema Qualitätsmodelle auf und berücksichtigt praktisch alle Arbeiten zum Thema Qualitätssicherung von Simulink-Modellen.

### 1.4.2. Qualitätsmodelle

Qualitätsmodelle machen den Begriff der Qualität fassbar. Sie ermöglichen eine objektive und einheitliche Bewertung der Qualität einer Software. Sie bilden den Grundstein vieler Arbeiten zum Thema Software- und Modellqualität, da ohne die Aufstellung eines Qualitätsmodells keine gemeinsame Vorstellung existiert, wann eine hohe oder niedrige Qualität vorliegt. Die meisten Qualitätsmodelle sind hierarchische Modelle, d. h. sie zergliedern den Begriff der Qualität hierarchisch in kleinere Teile. Beispiele für hierarchische Qualitätsmodelle sind das in dieser Arbeit als Grundlage verwendete Modell von Cavano und McCall [CM78], das Modell von Boehm [BBL76] und das Modell in der ISO 9126[3]. Das Modell der ISO 9126 baut auf die Modelle von McCall und Boehm auf. Die ISO 9126 wurde im März 2011 durch die Veröffentlichung der ISO/IEC 25010[4] obsolet. Letztere enthält gegenüber der ISO 9126 zusätzliche Faktoren und Kriterien. Allerdings sind die zusätzlichen Faktoren nicht neu, sondern entweder in identischer oder in ähnlicher Form in den anderen Qualitätsmodellen aus Abbildung 1.11 enthalten. Die Standards der 25.000er Reihe werden als SQuaRE (Software product Quality Requirements and Evaluation) bezeichnet.

In der Literatur werden diese hierarchischen Qualitätsmodelle meist als FCM-Modelle zusammengefasst, da die Elemente in den Qualitätsmodellen Faktoren, Kriterien und Metriken genannt werden. Abbildung 1.10 zeigt den schematischen Aufbau eines FCM-Modells.

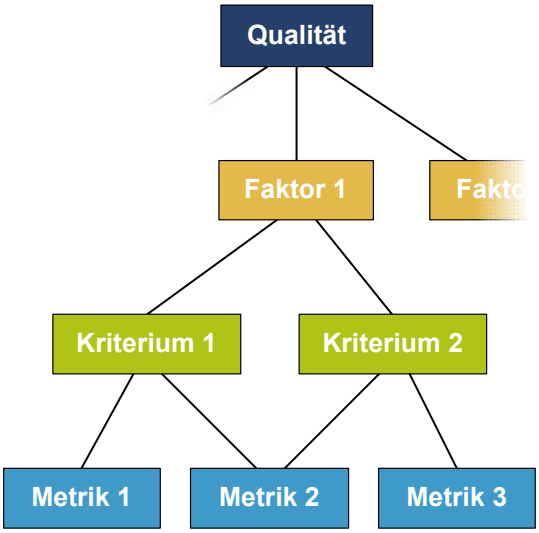



Abbildung 1.10.: Schematischer Aufbau eines FCM-Modells

Die Linien zwischen den Faktoren, Kriterien und Metriken stellen die Beziehungen zwischen den einzelnen Elementen dar. Die Faktoren geben gewünschte Eigenschaften für qualitativ

[3] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749
[4] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733

hochwertige Software bzw. Modelle vor. Die Kriterien sind Indikatoren für die gewünschten Eigenschaften und die Metriken wiederum prüfen die Erfüllung der Kriterien.

Obwohl die Qualitätsmodelle einige Gemeinsamkeiten haben, unterscheiden sie sich im Detail. Abbildung 1.11 zeigt einen Überblick über die Überschneidungen und Unterschiede der Faktoren der genannten Qualitätsmodelle. Allen gemeinsam sind die Faktoren Zuverlässigkeit, Effizienz, Wartbarkeit und Portabilität. Unterschiede gibt es jedoch bei den weiteren spezifischen Faktoren und dem Zusammenspiel der Kriterien. Die in fetter Schrift dargestellten Faktoren sind im Qualitätsmodell dieser Arbeit enthalten (vergleiche Kapitel 4). Der Faktor *Codegenerierbarkeit* (Code Generation Ability) ist für die modellbasierte Entwicklung hinzugekommen.

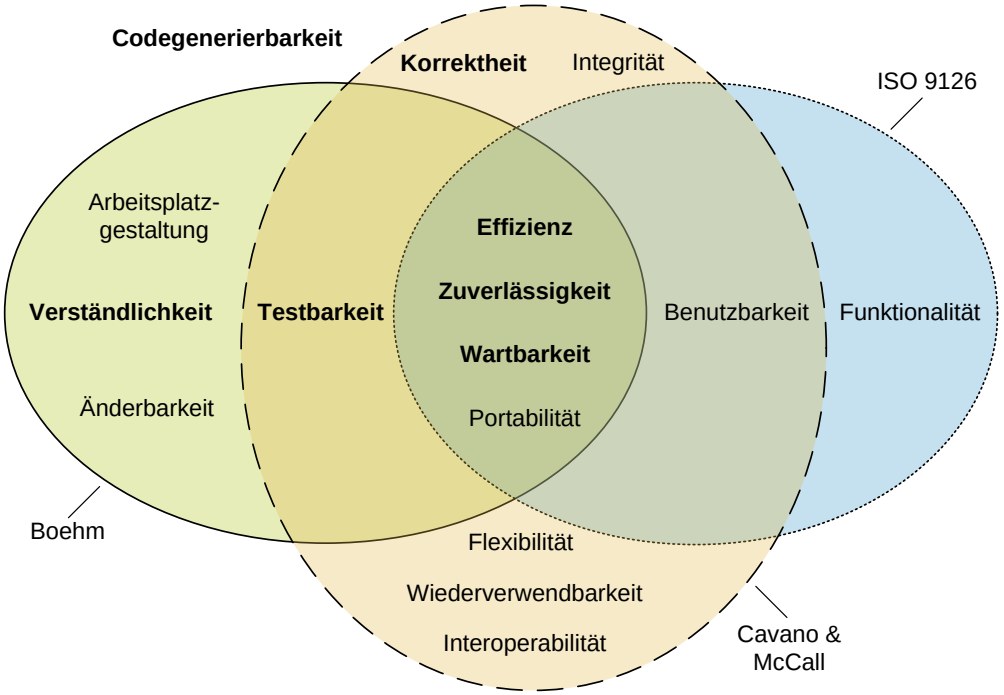



Abbildung 1.11.: Überschneidungen und Unterschiede der Qualitätsmodelle

Alle Qualitätsmodelle verfolgen das gleiche Ziel: den Begriff der Qualität fassbar zu machen. Dass sie im Detail unterschiedlich sind, liegt hauptsächlich an zwei Gründen. Zum einen unterscheiden sich die Qualitätsmodelle in Hinblick auf ihre Konzepte, d. h. welche Zusammenhänge zwischen den einzelnen Elementen bestehen. Zum anderen unterscheiden sie sich in Bezug auf die hierarchische Aufteilung, d. h. beispielsweise welche Kriterien auf welche Faktoren Einfluss haben. Die Unterschiede bei der Aufteilung lassen sich dadurch erklären, dass sich manche Kriterien mehreren Faktoren zuordnen lassen.

Der Goal-Question-Metric-Ansatz (GQM), der von Basili et al. in [BCR94] beschrieben wird, ist ein generisches Vorgehen, das die Erstellung eines individuellen Qualitätsmodells unterstützt. Das Ergebnis der Anwendung des Goal-Question-Metric-Ansatzes ist eine Liste von Fragen, denen Metriken zugeordnet sind. Die Metriken messen Werte, die es erlauben, die Fragen zu beantworten. Durch den Ansatz wird, wie bei der Verwendung eines FCM-Modells, die Qualität von Software bzw. Modellen objektiv erfassbar.

Bobkowska stellt in [Bob09] einen generischen Ansatz vor, der Qualitätskriterien und die Wahl der Methode zur Bewertung berücksichtigt. In einer Fallstudie zur Vorhersage der Qualität

eines Softwareprojekts, welches UML verwendet, wird von ihr ein FCM-Modell nach Cavano und McCall [CM78] verwendet.

Speziell für Simulink sind die beiden Arbeiten von Deissenboeck et al. und Ahrens et al. relevant. Deissenboeck et al. [Dei+07] stellen ein Qualitätsmodell für die Wartbarkeit auf. Sie modellieren nicht nur die Qualitätskriterien, sondern auch ihren Einfluss auf die nötigen Aktivitäten. In einer Fallstudie erweitern sie ihr Qualitätsmodell um Simulink-spezifische Kriterien und Aktivitäten. Eine vollständig automatisierte Qualitätsbewertung steht nicht in ihrem Fokus und das Qualitätsmodell konzentriert sich auf die Wartbarkeit. Die Aktivitäten sind vergleichbar mit den Handlungsempfehlungen im Ansatz dieses Beitrags. Ahrens et al. [AFPB10] haben ein Bewertungsverfahren für Softwarearchitekturen erstellt. Sie transformieren die konkreten Modelle in ein generisches Datenmodell, auf dem dann die Metriken operieren. Momentan unterstützen sie eine Transformation von Simulink-Modellen und ASCET-Modellen[5]. Ihr Ansatz ist automatisiert, allerdings beschränkt sich das Qualitätsmodell auf Architekturaspekte.

Zusammenfassend kann festgestellt werden, dass sehr viele Arbeiten über Qualitätsmodelle existieren. Es gibt viele Qualitätsmodelle für die herkömmliche handcodierte Softwareentwicklung und einige für sehr spezifische Anwendungsdomänen wie z. B. die Bewertungen von Web-Anwendungen. Allerdings gibt es sehr wenige Arbeiten für Daten- und Kontrollflussmodelle im Allgemeinen und noch weniger für Simulink-Modelle im Speziellen. Daher stellt das in dieser Arbeit vorgestellte Qualitätsmodell für diesen Anwendungsbereich einen Mehrwert dar.

### 1.4.3. Qualitätsbewertung

In diesem Abschnitt wird zuerst auf allgemeine Ansätze zur Qualitätsbewertung eingegangen. Danach werden Ansätze für Simulink-Modelle, UML-Modelle und Geschäftsprozessmodelle beschrieben. Bei den UML-Modellen und den Geschäftsprozessmodellen werden außerdem mögliche Ansätze für Transfers beleuchtet.

#### Allgemeine Ansätze

In diesem Abschnitt werden allgemeine Ansätze vorgestellt, die nicht spezifisch für Simulink sind. Das ConQAT-Framework der TU München[6] visualisiert verschiedene Qualitätscharakteristiken sowohl von Quellcode als auch von Modellen. Es bietet allerdings weder eine Unterstützung beim Finden der erlaubten Grenzen von Metriken, noch wird eine vergleichbare Aggregation wie in Abschnitt 5.2 unterstützt. Bisher sind nur einfache Aggregationen wie z. B. Minimum, Maximum, Mittelwert und Median integriert.

Im Rahmen des QuaMoCo-Projekts[7] wird ein in der Praxis anwendbarer Qualitätsstandard entwickelt. Deißenböck et al. beschreiben in [Dei+11] die in dem Projekt verwendete Toolkette. Die Toolkette enthält einen Qualitätsmodell-Editor für die Erstellung bzw. Anpassung eines Qualitätsmodells. In dem verwendeten Qualitätsmodell werden die benötigten Aktivitäten explizit modelliert, wie es von Deissenboeck et al. in [Dei+07] beschrieben wird. Abbildung 1.12 zeigt einen Überblick über die Toolkette. Das erstellte Qualitätsmodell wird

[5] `http://www.etas.com/de/products/ascet_software_products.php`
[6] `http://conqat.in.tum.de/index.php/ConQAT`
[7] `http://quamoco.in.tum.de`

in das oben beschriebene ConQAT-Framework übertragen und ausgeführt. Abschließend wird das ConQAT-Framework für die Visualisierung der Ergebnisse verwendet.

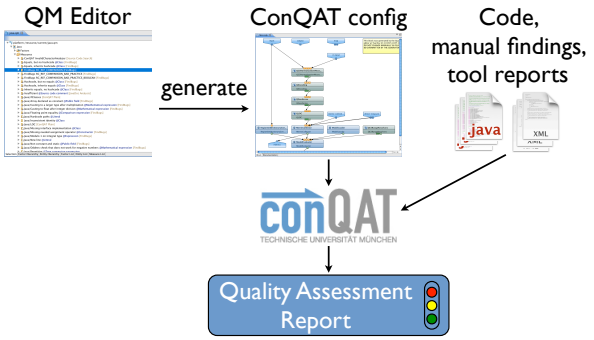



Abbildung 1.12.: QuaMoCo-Toolkette [Dei+11, S. 2]

Lilburne et al. verwenden in [LDK04] ein Qualitätsmodell, das eine vergleichbare Struktur wie das Qualitätsmodell in dieser Arbeit hat. Sie verwenden das Qualitätsmodell für die Bewertung von Web-Anwendungen. Allerdings kennen die Autoren die Maximalwerte der Metriken durch ihre Konstruktion und somit entfällt eine vergleichbare Auswertung der Messwerte. Außerdem verwenden sie zur Aggregation eine einfache Mittelwertbildung.

Khoshgoftaar und Allen verwenden in [KA99] ein Qualitätsmodell in Form eines Klassifikationsbaums. Die Blätter geben an, ob ein Fehler zu erwarten ist. Mit diesem Ansatz bewerten sie die Qualität von eingebetteten Systemen. In dem Klassifikationsbaum ist gut erkennbar, welche Metriken unabhängig voneinander sind und welche nicht. Sie beschreiben einen Algorithmus zum Aufbauen des Baums und verwenden diesen Baum in einer Fallstudie. Dank des Klassifikationsbaums ist keine Aggregation notwendig. Allerdings kann das Ergebnis für jedes Softwaremodul ausschließlich die Werte *fehleranfällig* oder *nicht fehleranfällig* annehmen.

Die in dieser Arbeit vorgestellte Modellqualitätsbewertung bewertet Messwerte eines Modells im Vergleich zu empirischen Daten und ob ein Modell bestimmte Regeln einhält. Viele andere Untersuchungen wählen meist eine Menge von Metriken aus, von denen angenommen wird, dass ihre Messwerte die gewünschten Rückschlüsse zulassen. Dann wird geprüft, ob eine Korrelation zwischen den Messwerten und den tatsächlichen Problemen besteht (zum Beispiel in [RPRB97], [MPKS00], [AM96] und [BBM96]).

### Ansätze für Simulink

In diesem Abschnitt werden Ansätze für Simulink vorgestellt. Kemmann et al. [KKT11] verwenden in ihrem INProVE-Ansatz benutzerdefinierte Qualitätsattribute, um die Anforderungen der unterschiedlichen Anwendungsdomänen abzudecken. Dies wird dadurch erreicht, dass dem Benutzer Werkzeuge zur Erstellung von angepassten Qualitätsindikatoren zur Verfügung gestellt werden. Die Qualitätsindikatoren prüfen das Vorhandensein von gewünschten Qualitätsattributen. Sie unterstützen verschiedene Modellierungssprachen, indem sie Adapter für den Zugriff auf ihr generisches Datenflussmodell verwenden. Ein Adapter ist zum Beispiel ein MATLAB Simulink Adapter. Die Festlegung der erlaubten Bereiche der Messwerte und die Aggregation der Bewertungen werden in ihrem Ansatz auf Ebene der Qualitätsindikatoren vorgenommen. Somit wird sowohl die Ermittlung der erlaubten Bereiche als auch die Aggregation der Bewertungen dem Benutzer überlassen.

Hu et al. beschreiben in [HLW12] einen Ansatz, der die ISO 9126 als Ausgangsbasis hat. Sie konzentrieren sich vor allem auf die interne Modellqualität, d. h. nicht auf die anderen am modellbasierten Entwicklungsprozess beteiligten Artefakte. Der Fokus ihres Qualitätsmodells liegt wie bei Deissenboeck et al. auf der Wartbarkeit der Simulink-Modelle. Der Ansatz von Hu et al. enthält momentan keine Konzepte zum Festlegen der erlaubten Grenzen für die Messwerte der Metriken. Somit steht auch eine automatisierte Qualitätsbewertung der untersuchten Simulink-Modelle nicht im Mittelpunkt.

#### Ansätze für die UML

In diesem Abschnitt werden Ansätze für die UML vorgestellt und ein möglicher Transfer auf Simulink diskutiert. Für UML-Diagramme gibt es bereits sehr viele Arbeiten, die entweder bestehende Metriken sammeln oder neue vorschlagen (z. B. für Klassendiagramme [GPC05], Zustandsdiagramme [GMP02], [Huh+05], Komponentendiagramme [ML05] und Sequenzdiagramme [VALC07]). Eine Sonderstellung nimmt die Arbeit von Purchase et al. ein. Sie befassen sich in [PMCC01] mit der Ästhetik und Verständlichkeit von UML-Klassendiagrammen. Dabei werden Aspekte wie eine minimale Anzahl an Linienknicken, eine optimale Verteilung der Klassen und eine einheitliche Pfeilrichtung berücksichtigt. Allerdings lassen sich UML-Metriken im Allgemeinen nicht ohne Weiteres auf den kombinierten Daten- und Kontrollfluss von Simulink-Modellen übertragen.

Arbeiten über die Qualität von UML-Modellen haben einen anderen Fokus, da die UML-Modelle meist nicht vollständig codegenerierbar sind. Vielmehr wird bei dem Paradigma der modellgetriebenen Architektur ein zweistufiges Konzept verwendet, das zuerst die Erstellung eines PIM (Platform Independent Model) und dann erst eine Transformation in ein PSM (Platform Specific Model) vorsieht. Simulink hingegen verwendet eine proprietäre Blockdiagrammnotation, die PIM und PSM nicht unterscheidet. Diese Blockdiagramme werden um Informationen zur Codegenerierung angereichert und dann von einem Codegenerator in C-Code überführt, der für das Zielsteuergerät optimiert ist.

Bei vielen Arbeiten über die Qualität von UML-Modellen stehen die Klassendiagramme im Mittelpunkt. Allerdings lassen sich Klassendiagramme nicht auf Simulink-Modelle übertragen, da es kein Klassen- und Vererbungskonzept in Simulink gibt. Simulink-Modelle sind kombinierte Datenfluss- und Kontrollflussdiagramme und sind somit Aktivitätsdiagrammen am ähnlichsten. Allerdings sind dem Autor keine Arbeiten bekannt, die sich mit der Qualität von Aktivitätsdiagrammen beschäftigen. Ein Grund dafür ist, dass Aktivitätsdiagramme nicht einfach ausgeführt werden können, da ihre Semantik nur informal in der UML-Spezifikation definiert ist. Ein Ansatz, der dennoch eine Ausführung von Aktivitätsdiagrammen erlaubt, ist die Abbildung von Aktivitätsdiagrammen auf abstrakte Zustandsmaschinen in [Sar06]. Allerdings handelt es sich bei der ActiveChartsIDE um einen universitären Forschungsprototypen. Die Arbeiten auf dem Gebiet der UML-Modelle lassen sich daher nicht direkt auf Simulink-Modelle übertragen.

Ein mit dem Ansatz dieses Beitrags vergleichbares Framework zur Qualitätsbewertung für UML-Modelle beschreiben Lange und Chaudron [LC05]. Sie unterscheiden zwischen Entwicklung und Wartung, da bei UML-Modellen in den jeweiligen Phasen andere Qualitätskriterien relevant sind. Sie verwenden zum Großteil OO-Metriken und nur einige modellspezifische Metriken wie z. B. Anzahl der überkreuzenden Linien.

### Ansätze für Geschäftsprozessmodelle

Geschäftsprozessmodelle verwenden meist eine Notation wie die BPMN[8] (Business Process Modeling Notation). Da Geschäftsprozessmodelle sowohl zur Dokumentation der Prozesse als auch als Hilfsmittel zur Abarbeitung der Prozesse verwendet werden, sind Geschäftsprozessmodelle oft simulierbar und ausführbar. In Geschäftsprozessmodellen wird vor allem der Kontrollfluss modelliert. Somit unterscheiden sich die Geschäftsprozessmodelle stark von den datenflusslastigen Simulink-Modellen. Außerdem enthält der Kontrollfluss der Geschäftsprozessmodelle oft Konstrukte zur Modellierung von Nebenläufigkeiten und Synchronisationspunkten, die nicht auf Simulink abbildbar sind.

SEQUAL [KSJ06] ist beispielsweise ein Qualitätsmodell für Geschäftsprozessmodelle. Es berücksichtigt sehr viel mehr Aspekte als nur die Eigenschaften des Geschäftsprozessmodells an sich. So spielen z. B. soziale Aspekte bei der Benutzung der Geschäftsprozesse und die Organisation, deren Geschäftsprozesse abgebildet werden, eine Rolle. SEQUAL definiert zentrale Qualitätsaspekte und deren Beziehungen zueinander und bildet somit einen abstrakten Rahmen für die Qualitätsbewertung von Geschäftsprozessmodellen.

Des Weiteren beschäftigen sich einige Arbeiten mit unterschiedlichen Aspekten der Qualität von Geschäftsprozessmodellen. Gruhn und Laue stellen in [GL06] Metriken vor, die die Komplexität von Geschäftsprozessmodellen messen. Mit der Verständlichkeit von Geschäftsprozessmodellen beschäftigen sich beispielsweise Mendling et al. in [MRC07] und Vanderfeesten et al. in [Van+08]. Die Identifikation von ähnlichen Geschäftsprozessmodellen, die Dongen et al. in [DDM08] beschreiben, ist vergleichbar mit der Suche von Klonen in Simulink-Modellen.

[8] `http://www.bpmn.org/`

# 2. Grundlagen

In diesem Kapitel werden die zum Verständnis dieser Arbeit benötigten Grundlagen beschrieben. Dazu wird in Abschnitt 2.1 zuerst der modellbasierte Entwicklungsprozess und die wichtigsten beteiligten Artefakte beschrieben. In Abschnitt 2.2 wird dann auf die konkrete Umsetzung der modellbasierten Entwicklung mit MATLAB Simulink eingegangen.

## 2.1. Modellbasierter Entwicklungsprozess

Im Kontext dieser Arbeit ist das Ziel des modellbasierten Entwicklungsprozesses die modellbasierte Erstellung einer Steuergerätesoftware. Der modellbasierte Entwicklungsprozess untergliedert sich dabei in mehrere große Phasen (siehe Abbildung 2.1). Es werden im Allgemeinen zwei Arten von Simulink-Modellen unterschieden: Simulationsmodelle und Funktionsmodelle. Simulationsmodelle können nur simuliert werden, Funktionsmodelle können simuliert und codegeneriert werden.

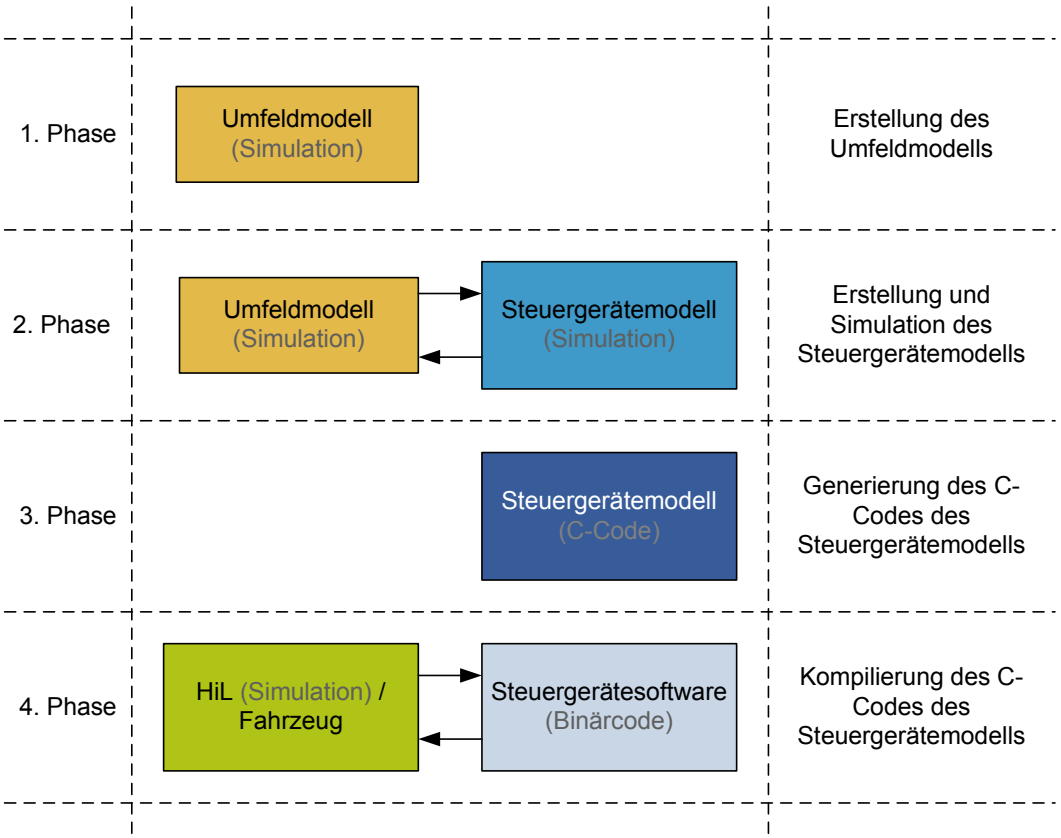



Abbildung 2.1.: Phasen des modellbasierten Entwicklungsprozesses

In der ersten Phase wird ein Umfeldmodell erstellt. In einem Umfeldmodell wird das Umfeld der Steuergerätesoftware modelliert. Das Umfeld sind die für die Steuergerätesoftware relevanten angrenzenden Systeme im Fahrzeug. Der Begriff System umfasst hierbei sowohl andere Steuergeräte als auch mechanische Komponenten. Im Falle eines Fensterhebers sind das z. B. sowohl das Türsteuergerät als auch der Motor des Fensters und die Sensoren zur Erkennung des oberen und unteren Anschlags. Das Umfeldmodell ist ein reines Simulationsmodell.

In der nächsten Phase wird das Modell erstellt, in welchem die gewünschte Funktionalität der Steuergerätesoftware umgesetzt wird. In dieser Phase sind beide Modelle noch reine Simulationsmodelle, die zusammen in der Simulation getestet werden können.

Nach Abschluss dieser Phase wird in der dritten Phase das Simulationsmodell um Informationen für die Codegenerierung erweitert. Ab diesem Zeitpunkt ist es möglich Code zu generieren. In den meisten Fällen wird C-Code generiert, da es für die Sprache C die meisten Compiler für die Prozessoren gibt, die typischerweise auf Steuergeräten verwendet werden.

In der vierten und letzten Phase wird der kompilierte Binärcode auf das Steuergerät übertragen und kommt dort zur Ausführung. Die Stelle des Umfeldmodells wird jetzt entweder von einem HiL-Teststand (Hardware in the Loop), wie beispielsweise dem dSPACE Simulator[1], oder von einem realen Fahrzeug eingenommen. Ein HiL-Teststand erlaubt eine Echtzeitsimulation des Umfelds. In dieser Simulation können sowohl reale Steuergeräte als auch in Software implementierte Anteile zum Einsatz kommen. Gegen Ende des Entwicklungsprozesses werden alle in Software implementierten Anteile durch ihre entsprechenden realen Steuergeräte ersetzt.

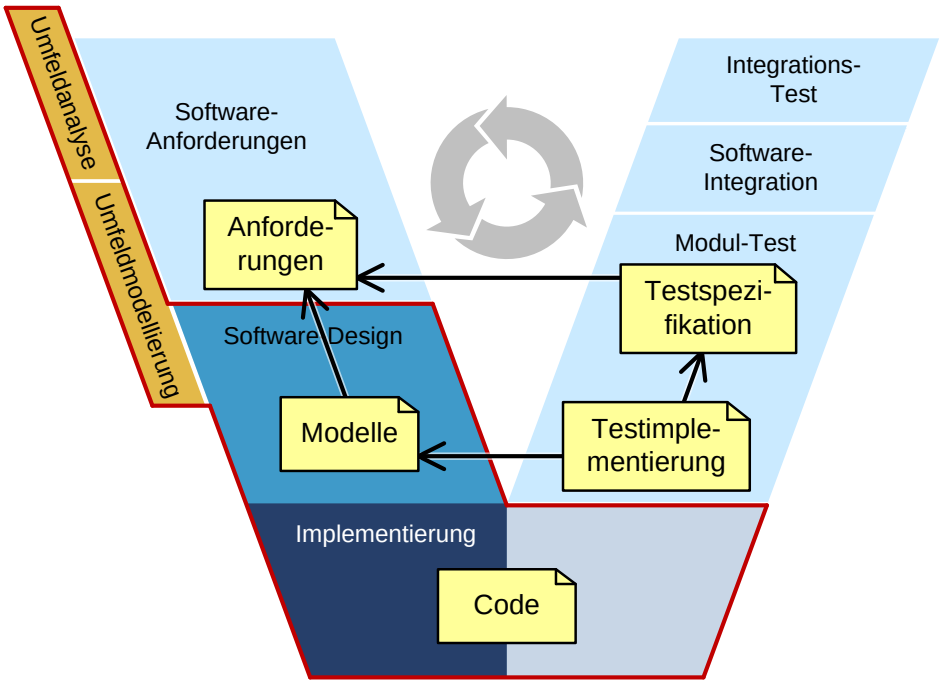



Abbildung 2.2.: Modellbasierter Entwicklungsprozess mit Artefakten im V-Modell

Abbildung 2.2 zeigt die Phasen des modellbasierten Entwicklungsprozesses eingebettet in das V-Modell. Dabei sind die aus Modellsicht relevanten Teile des V-Modells, die in Abbildung

[1] http://www.dspace.de/de/gmb/home/applicationfields/other_fields/ecutesting.cfm

2.1 beschrieben wurden, in der gleichen Farbe wie in Abbildung 2.1 dargestellt und rot umrandet. Der Wiederholungspfeil in der Mitte der Abbildung deutet an, dass es sich um einen iterativen Prozess handelt. Zusätzlich werden in den gelben Boxen die wichtigsten Artefakte dargestellt, die für die Modellierung eine Rolle spielen. Die Pfeile zwischen den Artefakten geben an, welche Artefakte miteinander in Beziehung stehen. So werden beispielsweise in dem Modell die Anforderungen implementiert und in der Testspezifikation werden Testfälle für die Anforderungen erstellt. Im Folgenden sind die Artefakte aus Abbildung 2.2 beschrieben:

**Anforderungen** Die Anforderungen beschreiben die gewünschte Funktionalität der Steuergerätesoftware.

**Modell** Mithilfe des Modells wird eine Steuergerätesoftware erstellt, die die Anforderungen erfüllt.

**Code** Der Code ist ein Zwischenprodukt, das aus dem Modell generiert und anschließend zu Binärcode kompiliert wird.

**Testspezifikation** Die Testspezifikation beschreibt die Testfälle, die für das Abprüfen der Anforderungen benötigt werden.

**Testimplementierung** Die Testimplementierung implementiert die in der Testspezifikation beschriebenen Testfälle.

In diesem Abschnitt wurde der modellbasierte Entwicklungsprozess nur insofern beschrieben, wie er für diese Arbeit relevant ist. Hierbei wurde ein Überblick über die Phasen und die beteiligten Artefakte gegeben. Den Artefakten im modellbasierten Entwicklungsprozess kommt im Rahmen dieser Arbeit die größte Bedeutung zu. Andreas Rau beschreibt in seiner Dissertation [Rau02, S. 123 ff.] die Konzeption eines Entwicklungsprozesses für die modellbasierte Entwicklung im Detail.

## 2.2. Modellbasierte Entwicklung mit MATLAB Simulink

MATLAB Simulink[2] ist ein Tool zur Simulation und Analyse von linearen und nichtlinearen mathematischen Modellen mit einer grafischen Notation. Es eignet sich sehr gut für das Lösen von regelungstechnischen Aufgabenstellungen, die in mechatronischen Systemen auftreten. Zusätzlich zu den für die Modellierung von Reglern benötigten Konstrukten enthält Simulink auch Konstrukte, um den Kontrollfluss zu modellieren, d. h. Bedingungen und Fallunterscheidungen. Daher kann es nicht nur für die Reglerentwicklung selbst, sondern für die Entwicklung der gesamten Steuergerätesoftware verwendet werden. MATLAB Simulink erlaubt auf einfache Art und Weise Simulink-Modelle zu simulieren, d. h. direkt in Simulink auszuführen. Es wird zwischen diskreten und kontinuierlichen Systemen unterschieden. Diskrete Systeme werden in festen Zeitintervallen abgetastet, d. h. im Falle eines Modells zyklisch ausgewertet. Die Signale kontinuierlicher Systeme hingegen existieren zu jedem beliebigen Zeitpunkt, da sie analog sind. Im Rahmen dieser Arbeit werden allerdings nur diskrete Systeme betrachtet, da Steuergerätesoftware immer nur mit digitalen Daten arbeitet und somit ein diskretes System darstellt.

[2] `http://www.mathworks.com/products/simulink/index.html`

### 2.2.1. Notation und Semantik

Simulink-Modelle sind kombinierte Daten- und Kontrollflussdiagramme. Sie bestehen aus *Blöcken* und *Linien*. Blöcke haben beliebig viele *Inports* und *Outports*. Nicht angeschlossene Inports werden durch ein kleines offenes Dreieck symbolisiert, nicht angeschlossene Outports durch ein kleines geschlossenes Dreieck (siehe Abbildung 2.3).

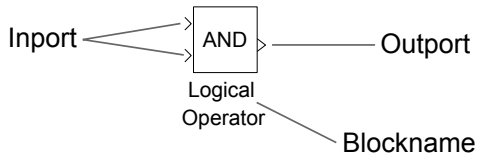



Abbildung 2.3.: Nicht angeschlossener Simulink-Block

Abbildung 2.4 zeigt, wie der logische Und-Block aus Abbildung 2.3 angeschlossen aussieht. Die Blöcke werden angeschlossen, indem sie mit Linien verbunden werden.

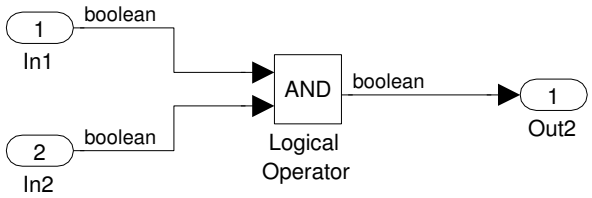



Abbildung 2.4.: Einfaches Simulink-Modell

Linien transportieren Signale in die Richtung, in die der Pfeil der Linie weist. Eine Linie führt immer von einem Outport eines Blocks zu einem Inport eines anderen Blocks. Linien können sich verzweigen, d. h. ein Outport eines Blocks ist mit mehren Inports anderer Blöcke verbunden. Signale sind Größen, die sich über die Zeit verändern und zu jedem Zeitpunkt einen Wert besitzen. Signale haben Eigenschaften wie zum Beispiel Name, Datentyp oder Dimensionalität. Im Folgenden wird der Begriff Signal aus Gründen der Einfachheit aber auch für eine konkrete Linie zwischen Blöcken verwendet. So müsste z. B. die Formulierung „ein Integer-Signal“ eigentlich „eine Linie, über die ein Signal vom Typ Integer fließt“ heißen.

Im Beispiel aus Abbildung 2.4 werden zwei boolesche Eingangssignale durch einen logischen Und-Block verknüpft und ausgegeben. Die Signale in Abbildung 2.4 sind alle skalare Signale, d. h. es wird nur ein einzelner Wert übertragen. Es gibt weitere Typen von Signalen wie zum Beispiel nicht-skalare Signale, Busse oder Kontrollsignale. Kontrollsignale (im Simulink-Modell als gestrichelte Linien dargestellt) initiieren die Ausführung anderer Blöcke. So ist die Ausgabe des Modells in Abbildung 2.5 entweder -1 oder 1. Je nachdem, ob der Eingangswert kleiner oder größer als null ist, wird der obere oder der untere Pfad aktiviert. Es gibt verschiedene Arten von Blöcken (siehe Tabelle 2.4 am Ende des Abschnitts). Sie unterscheiden sich in ihrer Semantik, das heißt in der Art und Weise, wie sie ihre Eingangssignale mit ihren Ausgangssignalen in Beziehung setzen. So geben zum Beispiel die quadratischen Blöcke über und unter dem If-Block immer einen konstanten Signalwert zurück. Der Merge-Block auf der rechten Seite hingegen leitet nur das momentan aktive Signal weiter. Zusätzlich besitzt jeder Block einen Namen. Dieser Name hat keinen Einfluss auf die Semantik des Modells, er erleichtert lediglich die Verständlichkeit.

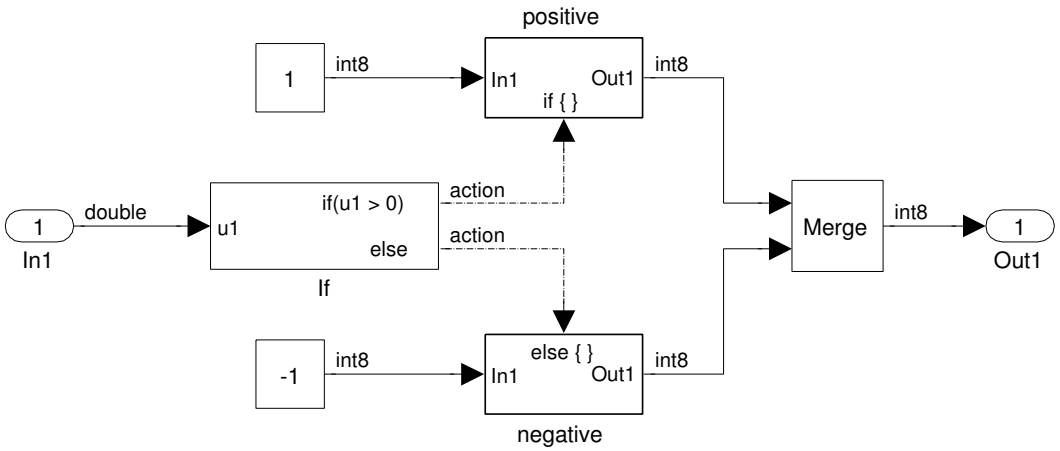



Abbildung 2.5.: Simulink-Modell mit Kontrollfluss

Ein weiterer Signaltyp ist der Bus. Ein Bus fasst beliebige Signale zu einem gebündelten Signal zusammen. Abbildung 2.6 zeigt ein Beispiel eines Busses, der drei Signale enthält. Entnommen werden aber nur zwei Signale aus dem Bus. Busse werden mit BusCreator-Blöcken erstellt und einzelne Signale werden mit BusSelector-Blöcken wieder entnommen. Busse können wiederum Busse enthalten, d. h. es sind dann mehrere BusSelector-Blöcke notwendig, um ein Signal aus einem Bus wieder zu entnehmen.

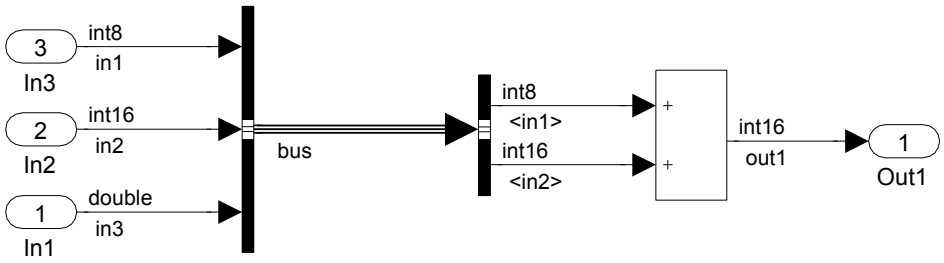



Abbildung 2.6.: Simulink-Modell mit einem Bus

Die Inport- und Outport-Blöcke, welche in Abbildung 2.4, 2.5 und 2.6 auf der linken bzw. rechten Seite zu sehen sind, sind spezielle Blöcke. Auf oberster Ebene stellen sie die Eingabe- und Ausgabeschnittstelle des Modells nach außen dar. Bei der Verwendung von sogenannten Subsystemen stellen sie innerhalb des Subsystems über ihre Nummern eine Verbindung zu den In- und Outports auf der Ebene des Subsystem-Blocks dar. Die Nummern der In- und Outports auf der Ebene des Subsystem-Blocks ergeben sich durch Nummerieren der In- und Outports von oben nach unten. Abbildung 2.7 illustriert diesen Zusammenhang. Die Namen der Signale in dem Subsystem müssen den Namen außerhalb nicht entsprechen. Dies ist z. B. bei wiederverwendbaren Subsystemen sinnvoll, da sie an verschiedenen Stellen im Modell verwendet werden. Bei Subsystemen, die zur hierarchischen Aufteilung verwendet werden, bietet es sich aus Gründen der Verständlichkeit hingegen an, die Signale außerhalb und innerhalb gleich zu benennen.

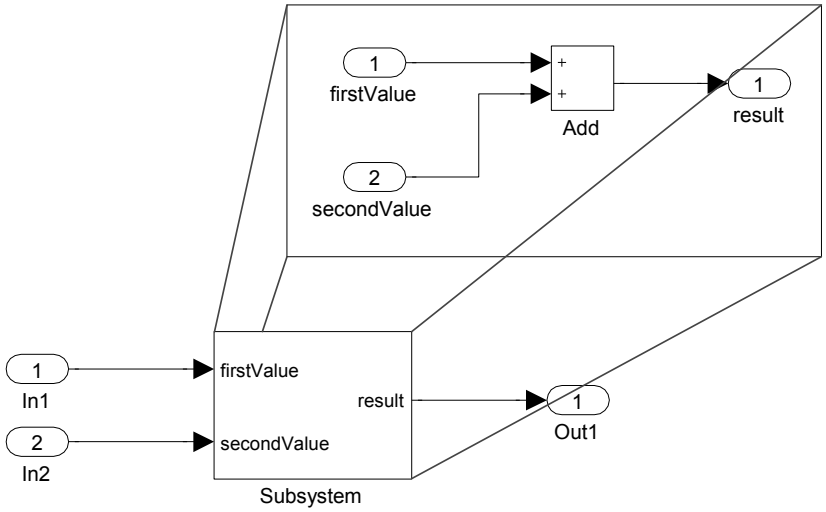



Abbildung 2.7.: Simulink-Modell mit dargestelltem Subsystem

Subsystem-Blöcke können wiederum andere Subsystem-Blöcke enthalten. Dadurch entsteht eine Subsystemhierarchie mit einer Baumstruktur. Die Blätter der Subsystemhierarchie sind die Subsysteme, die keine weiteren Subsystem-Blöcke enthalten.

In Simulink können Blöcke und Subsysteme, die wiederverwendbar sind, in Bibliotheken verwaltet werden. Wird in einem Modell ein Block aus einer Bibliothek verwendet, wird eine Verknüpfung mit dem Block im Modell erstellt und nicht der Block ins Modell kopiert. Dadurch ist es möglich, Änderungen an einer zentralen Stelle durchzuführen. Abbildung 2.8 zeigt, wie in Simulink ein Block dargestellt wird, der aus einer Bibliothek stammt.

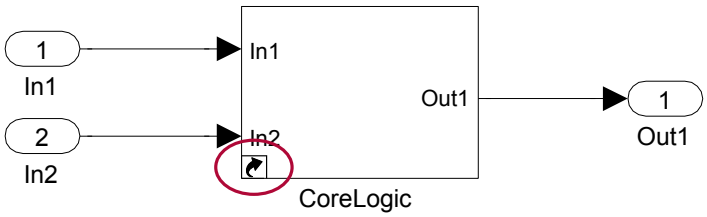



Abbildung 2.8.: Block aus einer Bibliothek

Zusammenfassend geben Tabelle 2.1 und 2.2 einen Überblick über die wichtigsten Eigenschaften der Blöcke und Linien in Simulink.

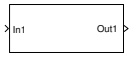
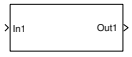
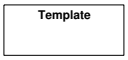
| **Eigenschaft** | **Beschreibung** |
|---|---|
| Typ | Der Typ des Blocks bestimmt die Anzahl seiner In- und Outports und seine Semantik. |
| Name | Der Name dient zur Verständlichkeit des Modells. |
| Subblöcke | Nur Blöcke vom Typ Subsystem können weitere Blöcke enthalten. |
| Inports | Die eingehenden Verbindungen des Blocks. |
| Outports | Die ausgehenden Verbindungen des Blocks. |

Tabelle 2.1.: Eigenschaften der Blöcke in Simulink

| Eigenschaft | Beschreibung |
|---|---|
| Name | Der Name des Signals dient zur Verständlichkeit des Modells. |
| Datentyp | Der Datentyp des Signals. |
| Typ | Der Typ des Signals, z. B. *Skalar*, *Nicht-Skalar*, *Kontrollsignal* oder *Bus*. |
| Dimensionalität | Die Anzahl der Dimensionen des Signals. |

Tabelle 2.2.: Eigenschaften der Linien in Simulink

Tabelle 2.4 zeigt die wichtigsten Simulink-Blöcke und ihre Semantik. In der letzten Spalte wird für jeden Block angegeben, ob er virtuell ist und deshalb keine Veränderung an den Signalen vornimmt und nur ein Hilfsmittel zur Strukturierung der Simulink-Modelle ist. Nicht in der Tabelle enthalten sind Stateflow-Blöcke, diese werden im Abschnitt 2.2.2 beschrieben.

| Grafische Darstellung | | Semantik | Virtuell |
|---|---|---|---|
|  | In1 | Eintrittspunkt der Signale auf jeder Hierarchieebene. | ja |
|  | Out1 | Austrittspunkt der Signale auf jeder Hierarchieebene. | ja |
|  | Subsystem | Block, der wiederum Blöcke enthalten kann und somit zur hierarchischen Aufteilung von Simulink-Modellen verwendet wird. | ja |
|  | Atomic Subsystem | Spezielles Subsystem, bei dem sichergestellt wird, dass es als atomare Einheit ausgeführt wird. | nein |
|  | Configurable Subsystem | Spezielles Subsystem, bei dem der konkrete Inhalt durch die zur Verfügung stehenden Alternativen ausgetauscht werden kann. | ja |
|  | Bus Creator | Block zum Erstellen eines Busses. | ja |
|  | Bus Selector | Block zum Entnehmen von Signalen aus einem Bus. | ja |
|  | Terminator | Dieser Block beendet ein Signal. | nein |
|  | Ground | Dieser Block erdet ein Signal, d. h. setzt es dauerhaft auf einen neutralen Wert (0). | nein |
|  | Switch | Dieser Block erlaubt eine Umschaltung zwischen zwei Signalen. | nein |
|  | Constant | Dieser Block gibt dauerhaft einen konstanten Wert zurück. | nein |
|  | Add | Dieser Block addiert die Werte zweier Signale. | nein |
|  | Logical Block | Dieser Block führt eine ausgewählte logische Operation durch. In diesem Fall eine logische Und-Verknüpfung der beiden Eingangssignale. | nein |
|  | Unit Delay | Dieser Block speichert den Wert eines Signals aus dem letzten Zyklus zwischen und stellt ihn im aktuellen Zyklus zur Verfügung. | nein |

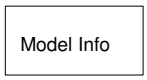
| | | | |
|---|---|---|---|
|  | Saturation | Dieser Block begrenzt den Wert eines Signals auf den vorgegebenen Minimal- und Maximalwert. | nein |
|  | Data Store Memory | Dieser Block speichert einen Wert. | nein |
|  | Data Store Read | Dieser Block liest aus einem DataStoreMemory-Block. | nein |
|  | Data Store Write | Dieser Block schreibt in einen DataStoreMemory-Block. | nein |
|  | Model Info | Dieser Block stellt Versionsinformationen über ein gesamtes Modell oder ein Subsystem bereit. | nein |
|  | Goto | Dieser Block leitet ein Signal zu den angegebenen From-Blöcken weiter. | ja |
|  | From | Dieser Block empfängt ein Signal von einem Goto-Block. | ja |

Tabelle 2.4.: Überblick über die wichtigsten Simulink-Blöcke

### 2.2.2. Zustandsautomaten

Stateflow[3] ist ein Zusatzprodukt von Simulink und stellt einen weiteren Blocktyp zur Verfügung, der es erlaubt, Zustandsautomaten in Simulink-Modellen zu verwenden. Abbildung 2.9 zeigt im Hintergrund ein Simulink-Modell.

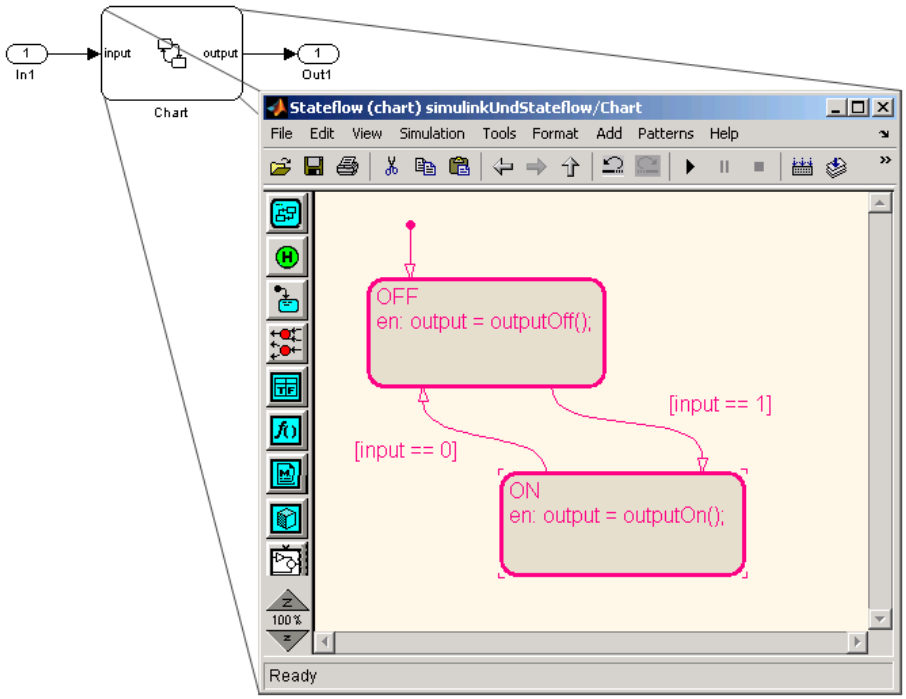



Abbildung 2.9.: Simulink und Stateflow

[3] http://www.mathworks.de/products/stateflow/index.html

Es enthält einen Stateflow-Block, der mit einem Stateflow-Diagramm verknüpft ist. Das Stateflow-Diagramm ist im Vordergrund sichtbar. In Stateflow-Diagrammen können Zustände und Zustandsübergänge verwendet werden. Im Gegensatz zu der datenflussorientierten Modellierung in Simulink erlauben die Stateflow-Diagramme eine zustandsorientierte Modellierung.

Ein Stateflow-Diagramm interagiert mit Simulink über die In- und Outports seines Stateflow-Blocks. Die In- und Outports des Stateflow-Blocks verhalten sich wie normale In- und Outports anderer Simulink-Blöcke. Auf einer abstrakten Ebene ist ein Stateflow-Block ein normaler Simulink-Block mit einer (potentiell) komplexen Logik. Dadurch werden Stateflow-Diagramme nahtlos in Simulink integriert.

### 2.2.3. Codegenerierung

Zur Ausführung auf einem Steuergerät muss Binärcode für die jeweilige Zielhardware produziert werden, da die Simulink-Laufzeitumgebung für die Ausführung auf dem PC konzipiert ist. Sie verbraucht sehr viele Ressourcen und ist auf ein Betriebssystem wie Windows oder Linux angewiesen.

Da ein direktes Erzeugen von Binärcode die Integration eines Compilers erfordern würde, wird der Umweg über generierten C-Code gegangen. Dies ermöglicht zum einen einen einfachen Austausch des Compilers für unterschiedliche Zielprozessoren und zum anderen entsteht in Form des generierten C-Codes ein untersuchbares Zwischenprodukt. So erlaubt der generierte C-Code, im Gegensatz zu direkt generiertem Binärcode, z. B. eine einfache Analyse möglicher Probleme. Der in dieser Arbeit betrachtete C-Codegenerator ist dSPACE Targetlink[4]. Bei der Verwendung von Targetlink bekommen die Simulink-Blöcke zusätzliche für die Codegenerierung notwendige Eigenschaften: Datentypen, Skalierung und Wertebereiche. Sie sind unabhängig von den Simulink-Datentypen, die nur für die Simulation relevant sind.

Außer den zusätzlichen Eigenschaften für die bestehenden Blöcke kommen auch einige neue für Targetlink spezifische Blöcke hinzu. Tabelle 2.6 zeigt einen Überblick über die wichtigsten zusätzlichen Targetlink-Blöcke.

| Grafische Darstellung | Semantik |
|---|---|
|  Custom Code Block | Dieser Block bindet beliebigen C-Code in ein Simulink-Modell ein. |
|  Function | Dieser Block sorgt dafür, dass für ein Subsystem eine separate C-Datei generiert wird. |

Tabelle 2.6.: Überblick über die wichtigsten zusätzlichen Targetlink-Blöcke

Abbildung 2.10 zeigt ein einfaches Simulink-Modell, welches zwei Werte addiert und den dazugehörigen C-Code. Bei der Codegenerierung kommen einige Aspekte hinzu, die für die Simulation nicht relevant waren. So stehen auf dem Steuergerät beispielsweise keine Fließkommazahlen zur Verfügung und es muss mit Festkommazahlen gerechnet werden. Außerdem wird die Ablage der Variablenwerte im Speicher relevant, da die Steuergeräte meist nur einen sehr begrenzten Speicher besitzen. Auch wenn bestimmte Zwischenwerte, d. h. Werte von Signalen an bestimmten Blöcken, in einem Modell zur Laufzeit beobachtet werden sollen,

[4] `http://www.dspace.de/de/gmb/home/products/sw/pcgs/targetli.cfm`

müssen Schritte unternommen werden, damit bekannt ist, wo im Speicherlayout die Variable liegt, in der der Zwischenwert gespeichert wird. Dies kann beispielsweise dadurch erreicht werden, dass die zu beobachtenden Signale bei der Codegenerierung auf globale Variablen mit festen Speicheradressen abgebildet werden.

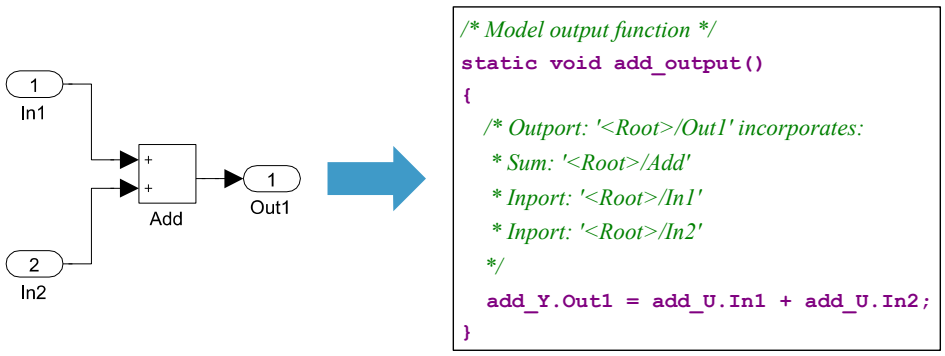



Abbildung 2.10.: Simulink-Modell und generierter Code

Die zusätzlichen Eigenschaften der Blöcke zur Codegenerierung können bei der Verwendung von Targetlink entweder lokal in den einzelnen Blöcken gespeichert werden oder zentral in dem so genannten Data Dictionary. Bei der zentralen Speicherung im Data Dictionary wird in den Blöcken nur ein Verweis auf einen Eintrag im Data Dictionary gespeichert. Abbildung 2.11 zeigt das Data Dictionary. In der Abbildung ist die zentrale Definition der verfügbaren Basistypen zu sehen. Einsortiert in eine Baumstruktur werden zentral Datentypen, Variablentypen und andere für die Codegenerierung relevante Einstellungen gespeichert.

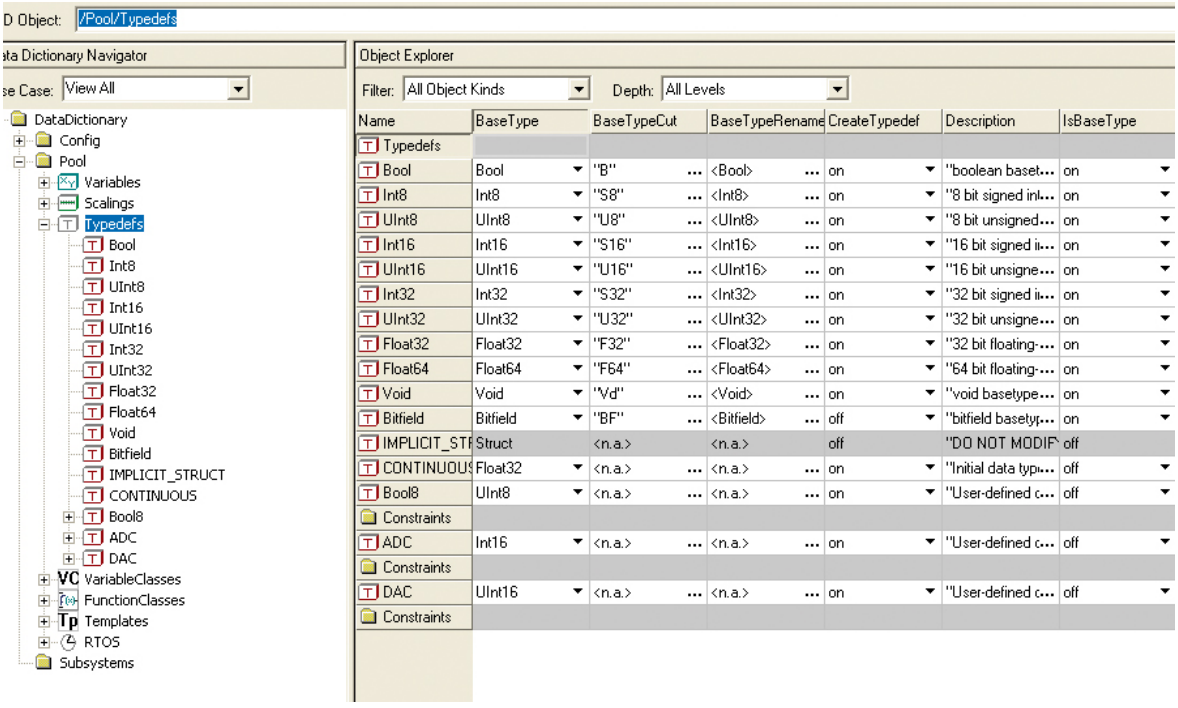



Abbildung 2.11.: Ausschnitt aus einem Targetlink Data Dictionary

Gegenüber der dezentralen Speicherung in den Blöcken hat die zentrale Speicherung den Vor-

teil einer einfachen Wiederverwendung und einer einfachen Änderbarkeit. Anstatt in jedem Block einzeln müssen die Änderungen nur einmal zentral im Data Dictionary vorgenommen werden.

### 2.2.4. Steuergerätesoftware

Sowohl das Modell in der Simulation als auch der C-Code auf dem Steuergerät werden zyklisch ausgeführt. In der Simulation wird das Modell von der Simulink-Laufzeitumgebung ausgeführt. Auf dem Steuergerät ruft der Scheduler des Betriebssystems den generierten Code zyklisch auf. In jedem Zyklus können sich die Werte am Eingang ändern und somit auch das Ergebnis am Ausgang.

Allerdings sind die Werte an den Ausgängen nur dann direkt von den Werten an den Eingängen abhängig, wenn das Modell keine internen Zustände besitzt. Ein Beispiel für einen internen Zustand ist ein UnitDelay-Block, der die Signalwerte aus dem jeweiligen vorherigen Zyklus zwischenspeichert.

Da es sich bei Steuergeräten im Automobilumfeld meist um Systeme mit Echtzeitanforderungen handelt, wird auf den Steuergeräten häufig ein Scheduling mit festen Zeitscheiben verwendet. Das bedeutet, dass bereits im Vorfeld bekannt ist, wie viel Zeit das Modell in jedem Zyklus zur Verfügung hat. Die Größe der Zeitscheibe ist von der Worst-Case-Execution-Time des Modells abhängig. Die Worst-Case-Execution-Time ist die Zeit, die das Modell im schlechtesten Fall zur Ausführung benötigt. Des Weiteren ist bekannt, wie lange der Zeitraum ist, bis das Modell wieder zur Ausführung kommt. Wie groß der Zeitraum ist, hängt von der Anzahl der Modelle auf dem Steuergerät und deren Worst-Case-Execution-Time ab.

### 2.2.5. Geforderte Eigenschaften der Modelle

In dieser Arbeit werden Simulink-Modelle betrachtet, die zur Funktionsmodellierung dienen. Enthält das Modell zusätzlich Targetlink-Blöcke, dann können auch Qualitätsaussagen über die Codegenerierbarkeit des Modells gemacht werden. Außerdem müssen die Modelle folgende Eigenschaften erfüllen:

- ein Modell-Update muss möglich sein
- keine unverbundenen In- und Outports

Bei einem Modell-Update bereitet Simulink ein Modell auf die Simulation vor. Dieser Vorgang ist grob mit dem Kompilieren von Source-Code zu vergleichen. Es werden z. B. die Ausdrücke in den Blockparametern ausgewertet, die Datentypen im Modell propagiert und die Ausführungsreihenfolge der Blöcke berechnet. Grundvoraussetzung für ein Modell-Update ist, dass das Modell keine syntaktischen Fehler enthält.

# 3. Metriken

In diesem Kapitel werden zunächst einige grundlegende Begriffe definiert. Dazu werden bestehende Konzepte von Softwaremetriken auf Modelle übertragen. In Abschnitt 3.2 wird eine Formalisierung der Simulink-Metriken vorgestellt. Diese Formalisierung wird von den Metriken in den Abschnitten ab 3.3 verwendet. Dabei wird für jeden Abschnitt die Formalisierung aus Abschnitt 3.2 vorausgesetzt. Sind spezifische Konstrukte notwendig, werden diese am Anfang der jeweiligen Abschnitte eingeführt. Die Modellmetriken sind speziell an Simulink-Modelle angepasst. Durch Berücksichtigung der Semantik der Simulink-Modelle können sie spezifische Aussagen über die Modelle machen. Die Modellmetriken werden in diesem Kapitel vorgestellt, ohne dass eine Aussage über die Interpretation ihrer Messwerte getroffen wird. Diese Interpretation und eine allgemeine Einbettung in das Thema Modellqualität werden in Kapitel 4 vorgenommen. Die in diesem Kapitel vorgestellten Metriken sind zu einem großen Teil in dem in [SK10] veröffentlichten Qualitätsmodell enthalten.

## 3.1. Definitionen

Die Metriken in dieser Arbeit sind Modellmetriken, d. h. die Metriken machen Aussagen über grafische Modelle. Die Modelle rücken gegenüber der handcodierten Softwareentwicklung in den Vordergrund, da das Modell in der modellbasierten Entwicklung das zentrale Artefakt ist. Zwar wäre es möglich, Code-Metriken auf den generierten C-Code der Modelle anzuwenden, jedoch ist dazu immer eine Rückverfolgung von dem C-Code in das Modell notwendig. Diese Rückverfolgung ist aufwendig und u. U. gar nicht möglich, da die Struktur der Modelle nicht 1:1 auf den Quellcode abgebildet wird. Des Weiteren gehen alle grafischen Informationen, wie z. B. die Aufteilung in Subsysteme, verloren und können daher nicht berücksichtigt werden.

Um die Unterschiede und Gemeinsamkeiten mit Softwaremetriken aufzuzeigen, werden die Definitionen von [Mil88] verwendet und an Modellmetriken angepasst beziehungsweise abgegrenzt. Diese Einordnung wurde bereits in [Sch10] beschrieben. Die Arbeit von Mills wurde gewählt, da er viele Vorschläge zur Charakterisierung und Einordnung von Metriken macht und sich dadurch die Modellmetriken gut beschreiben lassen. Mills definiert Metriken wie folgt: „software metrics deal with the measurement of software product and the process by which it is developed“. Somit lautet die Definition für Modellmetriken:

**Definition 1 (Modellmetrik).** *Eine Modellmetrik bildet eine oder mehrere Eigenschaften eines Modells oder des modellbasierten Entwicklungsprozesses auf einen Messwert ab.*

Durch das Messen findet eine Abstraktion von der Wirklichkeit statt. Es werden daher nur die Eigenschaften eines Modells berücksichtigt, welche von den jeweiligen Metriken erfasst wer-

den. Jede Metrik liefert einen *Messwert* zurück. Mehrere Messwerte bilden das *Messergebnis* eines *Messlaufs*. Daher gelten folgende Definitionen:

**Definition 2 (Messwert).** *Ein Messwert einer Metrik besteht aus einer Maßzahl und einer Einheit (Messwert = Maßzahl [**Einheit**]).*

**Definition 3 (Messergebnis).** *Ein Messergebnis besteht aus einem oder mehreren Messwerten.*

**Definition 4 (Messobjekt).** *Ein Messobjekt ist entweder das Modell selbst oder ein anderes Artefakt des modellbasierten Entwicklungsprozesses.*

**Definition 5 (Messlauf).** *Ein Messlauf ist die Anwendung einer oder mehrerer Metriken.*

Des Weiteren stellt Mills in [Mil88] eine Reihe von Forderungen an die Metriken:

**Einfachheit** Die Metriken sollen exakt definierbar sein, so dass es klar ist, wie die Metrik berechnet werden kann.

**Objektivität** Die Metriken sollen so objektiv wie möglich sein.

**Einfache Erhältlichkeit** Die Berechnung der Metriken darf keine hohen Kosten erzeugen.

**Gültigkeit** Die Metriken sollen das messen, für das sie entworfen wurden.

**Robustheit** Die Metriken sollen möglichst unempfindlich gegenüber Änderungen an den Prozessen oder Produkten sein.

Im Ansatz dieser Arbeit werden *Einfachheit*, *Objektivität* und *Einfache Erhältlichkeit* von den Metriken durch die automatisierte Ermittlung erreicht. *Gültigkeit* wird durch die Formalisierung in diesem Kapitel und durch die in Kapitel 6 beschriebene Implementierung erreicht. Auch die *Robustheit* wird durch die Implementierung erreicht. So ist es z. B. die Aufgabe der Implementierung, Unterschiede in verschiedenen Tool-Versionen vor den Metriken zu verbergen.

Mills unterscheidet zwischen *primitiven* und *berechneten* Metriken. Davon abweichend wird in dieser Arbeit eine etwas andere Unterteilung in folgende Gruppen verwendet:

**Zählmetrik** Zählmetriken liefern die Anzahl bestimmter Konstrukte zurück. Die Einheit ihrer Messwerte sind die gezählten Anzahlen (z. B. [**Blöcke**] bei *#Blöcke*). Das Symbol '#' wird dabei als Abkürzung für „Anzahl der...“ verwendet.

**Durchschnittsmetrik** Durchschnittsmetriken liefern Anzahlen im Verhältnis zu einer Referenz zurück. Die Einheit ihrer Messwerte sind Brüche aus den gezählten Anzahlen und der Referenz (z. B. [**Signal**]/[**Bus**] bei $\varnothing Busgröße$). Das Symbol '$\varnothing$' wird dabei als Abkürzung für „Durchschnittliche...“ verwendet.

**Prozentmetrik** Prozentmetriken liefern einen Prozentwert zurück, d. h. der Messwert ist beschränkt auf das Intervall $[0, 1]$ bzw. $[0, 100]$. Die Messwerte sind ohne Einheit bzw. tragen die Hilfseinheit [**Prozent**], da sich die Einheiten wegkürzen. Das ist beispielsweise bei der Metrik $\%ErfolgreicherTestfälle$ der Fall ([**(erfolgreiche) Testfälle**] / [**(alle) Testfälle**] = 1). Das Symbol '%' wird dabei als Abkürzung für „Prozentsatz...“ verwendet.

Die Gruppen sind so gewählt, dass sie in der Qualitätsbewertung in Kapitel 5 verwendet werden können. Sie sind auch auf die beiden Kategorien von Mills abbildbar. So sind Zählmetriken primitiv und Durchschnittsmetriken und Prozentmetriken berechnet. Zusätzlich liegen die Messwerte der Metriken auf einer Rationalskala, da in Kapitel 5 Mittelwerte der Messwerte gebildet werden.

Außerdem kann zwischen *direkten* und *indirekten* Metriken unterschieden werden. Diese Unterscheidung wird von Mills nicht vorgenommen. Direkte Metriken messen auf dem Modell, indirekte verwenden eine externe Datenquelle, welche aufbereitete Informationen über das Modell bereitstellt. Ein Beispiel für eine externe Datenquelle sind die Ergebnisse eines Prüfers für Modellierungsrichtlinien, welche Rückschlüsse auf die Standardkonformität eines Modells zulassen. Allerdings ist diese Unterscheidung nur für die Implementierung relevant. Auf konzeptioneller Ebene ist es irrelevant, wie die Informationen über die Einhaltung der Modellierungsrichtlinien zustande gekommen sind. Sie können sowohl durch eine direkte Analyse des Modells als auch durch ein externes Tool zustande gekommen sein.

Jede Metrik hat einen *Typ*, welcher bestimmt, wie die Messwerte erhoben werden. Metriken, die nur einen einzelnen Messwert pro Modell berechnen können, sind vom Typ *atomar*. Alle anderen Metriken können hingegen pro Subsystem einen Wert messen. Aus den Werten pro Subsystem wird dann der endgültige Messwert wie folgt berechnet: Bei Metriken vom Typ *akkumulierend* werden die Messwerte pro Subsystem aufaddiert. Beim Typ *arithmetisches Mittel* wird zusätzlich noch durch die Anzahl der Subsysteme geteilt.

Im Folgenden wird das Schema vorgestellt, in dem alle Metriken in dieser Arbeit notiert werden. Als erstes wird der Namen der Metrik aufgeführt. In der zweiten Zeile steht auf der linken Seite die Einheit der Messwerte der Metrik und auf der rechten Seite der Faktor der Metrik aus dem Qualitätsmodell in Kapitel 4. Darunter steht die Formel zur Berechnung der Metrik. Anhand des tiefgestellten Symbols an der Variablen *messwert* kann erkannt werden, ob die Metrik eine Zählmetrik, eine Durchschnittsmetrik oder eine Prozentmetrik ist. Dabei beginnen die Namen von Zählmetriken im Allgemeinen mit „Anzahl der...“ und die Namen von Durchschnittsmetriken immer mit „Durchschnittliche...“. Zuletzt folgt, falls vorhanden, eine textuelle Beschreibung der Metrik.

**Metrik 0:** Anzahl der Beispiele

| Einheit: [**Beispiele**] | Faktor: Beispielfaktor |
|---|---|

$$messwert_{\#} = |\{beispiel \in Beispiele\}|$$

Der Typ und das Messobjekt, die entscheiden, ob eine Metrik direkt oder indirekt ist, sind für die Definition und das Verständnis der Metriken nicht relevant. Für die Implementierung des Prototyps in Kapitel 6 hingegen schon. Daher werden diese Eigenschaften den Metriken erst in Abschnitt 6.2 zugeordnet.

## 3.2. Formalisierung der Simulink-Modelle

In diesem Abschnitt wird eine alternative Sicht auf die Simulink-Modelle vorgestellt: die Sicht als Graph. Diese Sicht als Graph erlaubt es, die Metriken in den folgenden Abschnitten in einer einheitlichen Notation darzustellen. Diese Notation verwendet Mengen als Datenstrukturen und Funktionen für die Abbildungen zwischen den Mengen. Mengen werden durch geschweifte Klammern dargestellt. Für die Anzahl der Elemente, d. h. Kardinalität der Mengen, werden Betragsstriche verwendet. Außerdem werden in diesem Abschnitt die Begriffe Signalpfad, unabhängiger Berechnungspfad und Modellklon definiert.

Jedes Modell stellt einen Graph $G$ dar. Die Blöcke sind die Knoten $V$ und die Linien sind die Kanten $E$. Statt der üblicherweise für Graphen verwendeten Notation $G = (V, E)$ wird im Folgenden die Schreibweise $\mathit{Modelle} = (\mathit{Blöcke}, \mathit{Linien})$ verwendet, da sie in Bezug auf Simulink-Modelle intuitiver ist. Dabei ist $\mathit{Blöcke}$ die Menge aller Blöcke und $\mathit{Linien} \subseteq \mathit{Blöcke} \times \mathit{Blöcke}$ die Menge aller Linien eines Simulink-Modells $\mathit{modell} \in \mathit{Modelle}$.

Zur hierarchischen Aufteilung werden Simulink-Modelle in Subsysteme aufgeteilt. Die Menge aller Subsysteme eines Modells wird mit $\mathit{Subsysteme}$ bezeichnet. In der Darstellung als Graph ist die Hierarchie der Subsysteme nicht explizit modelliert, alle Subsysteme liegen als Teilgraphen nebeneinander. Daher zerfällt das Modell $\mathit{modell}$ in paarweise disjunkte Teilmengen und es gilt $\mathit{modell} = \bigcup_{s \in \mathit{Subsysteme}} s$. Der Zusammenhang zwischen einem Subsystem und den Blöcken, die es enthält, wird über die Funktion $\mathit{subblöcke} : \mathit{Subsysteme} \to \mathit{Blöcke}$ hergestellt. Die Subsystemhierarchie kann auch als Baum beziehungsweise als ein weiterer gerichteter Graph $G^\star = (V^\star, E^\star)$ interpretiert werden. Dabei gilt $V^\star = \mathit{Subsysteme}$ und für $E^\star$ gilt:

$$E^\star = \{(s_1, s_2) \mid s_1 \in \mathit{Subsysteme} \wedge s_2 \in \{b \in \mathit{subblöcke}(s_1) \mid \mathit{typ}(b) = \text{‘Subsystem‘}\}\}$$

Da die Reihenfolge der Linien eine Auswirkung auf die Semantik der Blöcke hat, wird als zusätzliche Information die Portnummer der Linien benötigt. Somit müssen zwei zusätzliche Abbildungen definiert werden:

$$\begin{aligned} \mathit{inportNummer} &: \mathit{Blöcke} \times \mathit{Linien} \to \mathbb{N}^+ \\ \mathit{outportNummer} &: \mathit{Linien} \to \mathbb{N}^+ \end{aligned}$$

Diese beiden Abbildungen ordnen jeder Line eine Inport-Nummer und eine Outport-Nummer zu. Da sich Linien verzweigen können, muss bei Inport-Nummern zusätzlich der zu betrachtende Block übergeben werden. Außerdem wird jedem Block und jeder Linie über die Abbildungen $\mathit{name} : \mathit{Blöcke} \to \mathit{Namen}$ und $\mathit{name} : \mathit{Linien} \to \mathit{Namen}$ ein Name aus der Menge aller möglichen Namen $\mathit{Namen}$ zugeordnet. Dabei gilt $\mathit{name}(b_1) \neq \mathit{name}(b_2)$ für $b_1, b_2 \in s_1$, d. h. pro Subsystem sind die Namen der Blöcke eindeutig.

Somit ergibt sich eine Darstellung als gerichteter Graph mit Mehrfachkanten, welche über die beiden Abbildungen $\mathit{inportNummer}(b, l)$ und $\mathit{outportNummer}(l)$ ihre Port-Nummern als natürliche Zahlen und über die Abbildung $\mathit{name}(b)$ ihren Namen zugeordnet bekommen. Die

Knoten bekommen ihren Namen ebenfalls durch die Abbildung $name(b)$ zugeordnet. Somit ergibt sich folgender Graph:

$$modell = (Blöcke, Linien, Subsysteme, name, inportNummer, outportNummer)$$

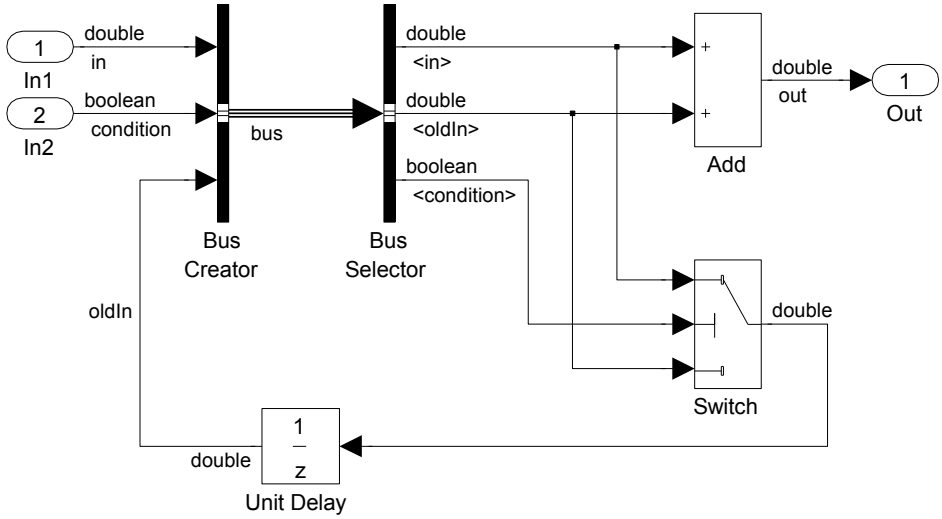



Abbildung 3.1.: Modell in Blockdiagrammdarstellung

Abbildung 3.1 zeigt ein Simulink-Modell in der normalen Blockdiagrammdarstellung. Abbildung 3.2 hingegen zeigt das gleiche Modell in der Graphendarstellung. Am Beispiel des BusSelector-Blocks sind die Mehrfachkanten gut erkennbar. Unterschieden werden sie durch die ihnen zugeordneten Portnummern.

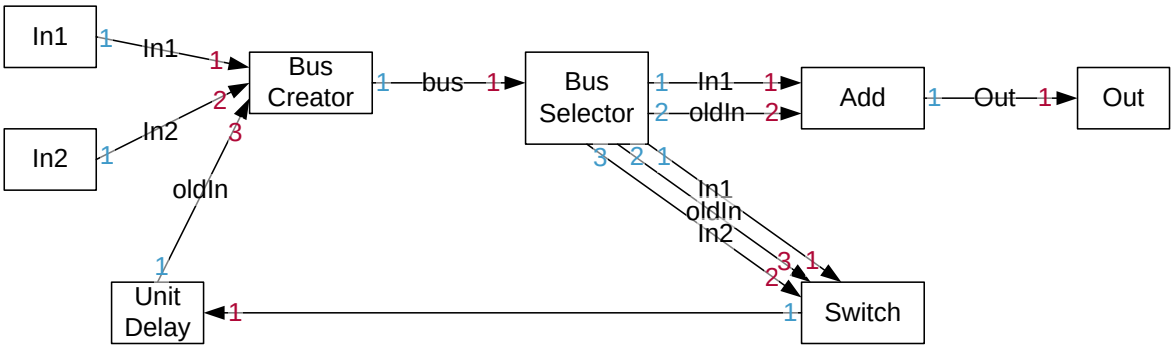



Abbildung 3.2.: Modell in Graphendarstellung

Analog zu dem Begriff Teilgraph kann der Begriff Teilmodell auf Simulink-Modelle angewendet werden. Ein Teilmodell $modell'$ enthält nur Blöcke und Linien aus $modell$, somit gilt $modell' \subseteq modell$. Ein Teilmodell kann zusammenhängend im Sinne eines zusammenhängenden Graphen sein oder in mehrere Zusammenhangskomponenten $z \in modell'$ zerfallen. Jedes Subsystem $s$ eines Modells $modell$ ist auch ein Teilmodell und besteht aus einer Zusammenhangskomponente oder aus mehreren Zusammenhangskomponenten. Somit gilt $modell = \bigcup_{s \in Subsysteme} s = \bigcup_{z \in Zusammenhangskomponenten} z$.

Zusammengefasst werden folgende Datenstrukturen und Funktionen zur Beschreibung eines Simulink-Modells als Graph benötigt:

**(1) Datenstrukturen**

***modell*** Ein konkretes Simulink-Modell.

***Blöcke*** Die Menge aller Blöcke $b \in \mathit{Blöcke}$ in einem Simulink-Modell *modell*.

***Linien*** Die Menge aller Linien $l \in \mathit{Linien}$ in einem Simulink-Modell *modell*.

***Subsysteme*** Die Menge aller Subsysteme $s \in \mathit{Subsysteme}$ in einem Simulink-Modell *modell*.

***Annotationen*** Die Menge aller Annotationen $\mathit{annotation} \in \mathit{Annotationen}$ in einem Simulink-Modell *modell*. Annotationen sind Textblöcke, die frei in Subsystemen platziert werden können.

**(2) Funktionen**

***name*** $(b)$ Gibt den Namen eines Blocks zurück.

***name*** $(l)$ Gibt den Namen einer Linie zurück. Wurde kein Name vergeben, wird eine leerer Name zurückgegeben.

***typ*** $(b)$ Gibt den Typ eines Blocks zurück.

***subblöcke*** $(s) \subseteq$ ***Blöcke*** Gibt die Menge aller Blöcke eines Subsystems $s$ zurück.

***tiefe*** $(s) \in \mathbb{N}^+$ Gibt die Länge des Pfads vom Wurzelknoten der Subsystemhierarchie bis zu dem betrachteten Subsystem zurück.

***linien*** $(s) \subseteq$ ***Linien*** Gibt alle in einem Subsystem $s$ enthaltenen Linien zurück.

***annotationen*** $(s) \subseteq$ ***Annotationen*** Gibt alle in einem Subsystem $s$ enthaltenen Annotationen zurück.

***inportNummer*** $(b, l) \in \mathbb{N}^+$ Gibt die Inport-Nummer einer Linie $l$ zurück, die mit dem Block $b$ verbunden ist.

***outportNummer*** $(l) \in \mathbb{N}^+$ Gibt die Outport-Nummer einer Linie $l$ zurück.

Somit enthält Abbildung 3.2 nur einen Teil der Informationen, die in einem Simulink-Modell gespeichert werden. So sind zum Beispiel in Abbildung 3.1 Datentypen an den Linien sichtbar, die in Abbildung 3.2 nicht dargestellt sind. Auch hat jeder Simulink-Block weitere Eigenschaften wie z. B. einen Typ. Daher werden in den folgenden Abschnitten weitere Datenstrukturen und Funktionen eingeführt, die zur Berechnung der vorgestellten Metriken benötigt werden.

### 3.2.1. Signalpfade

In Simulink wird durch Linien der Signalfluss modelliert. Signale fließen durch ein Simulink-Modell und werden mit anderen Signalen verrechnet. Um ein Simulink-Modell zu verstehen, muss der Signalfluss in dem gesamten Modell verstanden werden. Daher ist es von Interesse, Aussagen treffen zu können, wie lange Signale nur durch Modelle geleitet werden, bis sie für eine Berechnung verwendet werden. Dazu dient die Definition des *Signalpfads*.

Ein Signalpfad $\mathit{signalpfad} \in \mathit{Signalpfade}$ ist eine Menge von Linien $\mathit{Linien}_1 = (l_1, l_2, ..., l_n)$ und eine injektive Abbildung $r : \mathit{Linien}_1 \to \mathbb{N}$, die die Reihenfolge der Linien vorgibt. Durch die Vorgabe der Reihenfolge wird die Menge der Linien zu einer Liste, in der jeder Linie ein eindeutiger Index zugeordnet werden kann. Ein Signalpfad verbindet ausschließlich *Signalträger* miteinander. Signalträger sind dabei Blöcke, die den Wert des Signals auf seinem Weg nicht verändern. Das bedeutet, dass jede Linie $l$ eines Signalpfads den gleichen Datentyp, Typ und die gleiche Dimensionalität besitzt. Signalträger sind alle in Tabelle 2.4 als

virtuell markierten Blöcke, Switch-Blöcke und einige Sonderfälle, die nur einen Inport und einen Outport haben. Sonderfälle sind zum Beispiel der Gain-Block, der Not-Block und der Saturation-Block. Obwohl sie den Wert des Signals ändern, ist kein anderer Signalwert involviert, dessen Zustandekommen auch nachvollzogen werden müsste. Ein Signalpfad beginnt an einer *Signalquelle* und endet an einer *Signalsenke*. Abbildung 3.3 zeigt ein Beispiel eines Signalpfads.

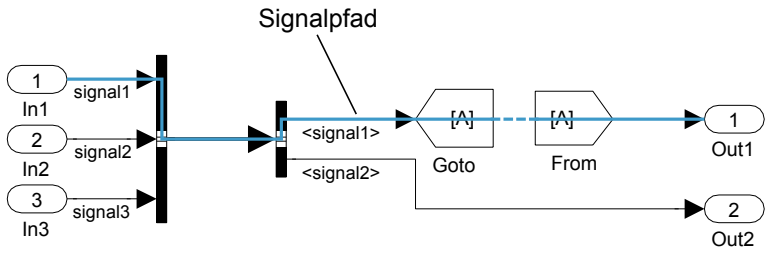



Abbildung 3.3.: Beispiel eines Signalpfads

Ein *Maximaler-Identitäts-Signalpfad* (MISP) ist ein Signalpfad, der von einem betrachteten Block $b$ aus die maximal mögliche Länge bis zum nächsten Block besitzt, der kein Signalträger ist. *MISP-Erzeuger* sind alle Blöcke, die keine Signalträger sind. MISPs sind nicht auf ein Subsystem beschränkt, sondern können sich über Subsystemgrenzen hinweg erstrecken.

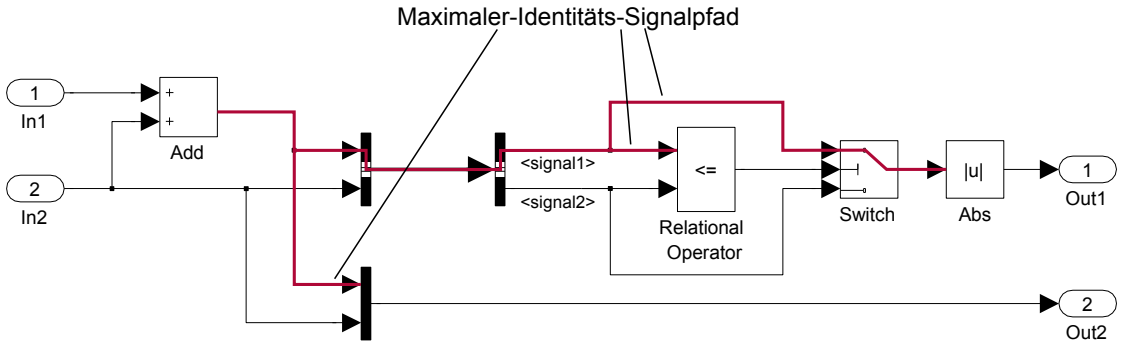



Abbildung 3.4.: Beispiel eines Maximalen-Identitäts-Signalpfads

Da jeder Block $b$ sowohl mehr als einen Outport besitzen kann als auch sich die Linien verzweigen können, existiert für jeden Block eine Menge von MISPs. Abbildung 3.4 zeigt ein Beispiel eines Add-Blocks, der drei MISPs besitzt. Der Algorithmus zur Berechnung der Signalpfade wird in [Hol11] im Detail beschrieben.

#### Datenstrukturen

***MISPs*** Die Menge aller Maximaler-Identitäts-Signalpfade $misp \in MISPs$ in einem Simulink-Modell *modell*.

### 3.2.2. Unabhängige Berechnungspfade

Subsysteme sind logische Einheiten, die eine hierarchische Aufteilung eines Problems erlauben. In Abbildung 3.5 wird ein Beispiel eines Subsystems mit zwei unabhängigen Berechnungspfaden gezeigt. Unabhängige Berechnungspfade sind von Interesse, da sie ein möglicher

Hinweis auf voneinander unabhängige Funktionalitäten in einem Subsystem sind. Diese unabhängigen Funktionalitäten könnten aus Gründen der Übersichtlichkeit und Verständlichkeit in separate Subsysteme aufgeteilt werden.

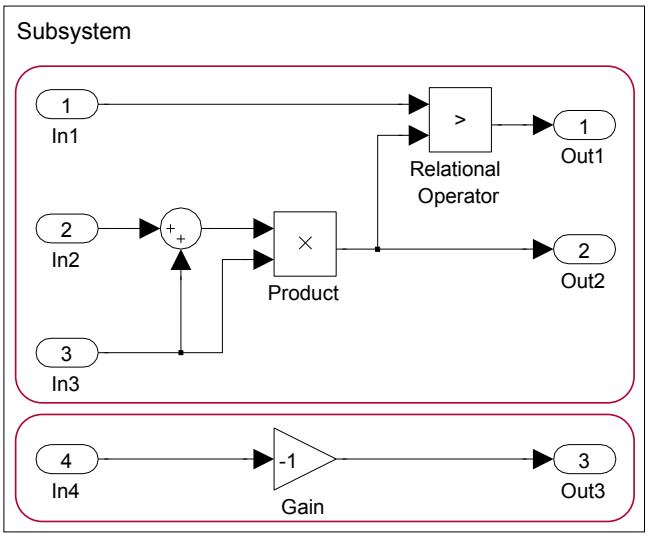



Abbildung 3.5.: Subsystem mit zwei unabhängigen Berechnungspfaden [Hol11, S. 43]

Die Anzahl der unabhängigen Berechnungspfade wird durch das Zählen der Zusammenhangskomponenten berechnet. Der Algorithmus zur Berechnung der Anzahl der Zusammenhangskomponenten wird in [Hol11] im Detail beschrieben.

#### Funktionen

***berechnungspfade*** $(s) \subseteq$ ***modell*** Gibt die Menge aller unabhängigen Berechnungspfade eines Subsystems $s$ zurück.

### 3.2.3. Modellklone

Geklonte Modellteile sind identische Modellteile, die durch Kopieren und Einfügen entstanden sind. Sie erschweren das Beseitigen von Fehlern oder die Erweiterung um neue Funktionalität, da die Änderungen nicht nur an einer zentralen Stelle, sondern in allen kopierten Modellteilen vorgenommen werden müssen.

Zwei äquivalente Modellteile $K = (Blöcke_K, Linien_K)$ und $K' = (Blöcke_{K'}, Linien_{K'})$ werden Klon genannt, wenn ihre Teilgraphen zusammenhängend sind, eine äquivalente Blockmenge besitzen, überschneidungsfrei sind, eine äquivalente Linienmenge besitzen, eine vergleichbare Ausdehnung haben und eine Mindestgröße besitzen. Die beiden Teile $K$ und $K'$ können im selben Subsystem enthalten sein, müssen aber nicht.

Abbildung 3.6 zeigt schematisch einen Klon. Er besteht aus den Teilen $K$ und $K'$, die sich in verschiedenen Subsystemen befinden. Jeder Block in $K$ ist einem Block in $K'$ zugeordnet. Aus Gründen der Übersichtlichkeit sind die Linien zwischen den Blöcken in der Abbildung nicht dargestellt.

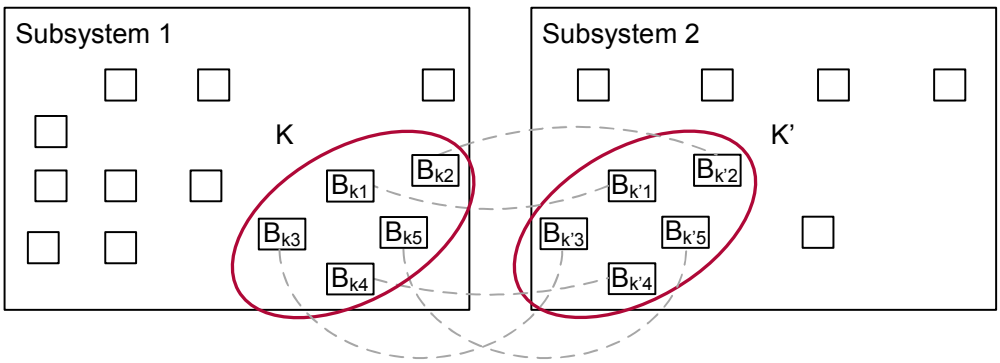



Abbildung 3.6.: Schematische Darstellung eines Klons

In diesem Abschnitt werden die geforderten Eigenschaften der beiden Teilmodelle $K$ und $K'$ im Detail beschrieben:

**Zusammenhängend** Sowohl der Teilgraph $K$ als auch $K'$ sind zusammenhängend.

**Überschneidungsfrei** Es muss $K \cup K' = \emptyset$ gelten.

**Äquivalente Blockmenge** Für jeden Block $b \in \mathit{Blöcke}_K$ existiert ein äquivalenter Block $b' \in \mathit{Blöcke}_{K'}$ und andersherum. Somit existiert eine bijektive Abbildung zwischen $\mathit{Blöcke}_K$ und $\mathit{Blöcke}_{K'}$. Zwei Blöcke sind genau dann äquivalent, wenn ihr Typ, die Anzahl der Subblöcke, die Anzahl der Inports, die Anzahl der Outports und der Operator übereinstimmen. Der Operator ist nur im Falle von Blöcken relevant, die logische oder relationale Operationen durchführen.

**Äquivalente Linienmenge** Für jede Line $l \in \mathit{Linien}_K$ existiert eine äquivalente Linie $l' \in \mathit{Linien}_{K'}$. Linien sind äquivalent, wenn ihr Quellblock und Zielblock äquivalent und ihre Inport- und Outport-Nummern identisch sind.

**Vergleichbare Ausdehnung** Die Größe der kleinsten möglichen Rechtecke, die sowohl $K$ als auch $K'$ einschließen, müssen vergleichbar groß sein (zum Beispiel maximal 10% Abweichung).

**Mindestgröße** $|K|$ und $|K'|$ müssen größer als eine gewählte minimale Blockanzahl sein. Der Wert fünf hat sich als ein in der Praxis bewährter Wert herausgestellt.

Ein Klonteil kann sich nur innerhalb eines Subsystems befinden, das heißt, weder $K$ noch $K'$ kann sich über mehr als ein Subsystem erstrecken. Des Weiteren kann keine Aussage darüber getroffen werden, ob der Modellteil $K$ oder $K'$ das Original ist. Diese Aussage kann nur durch die Betrachtung der Bearbeitungshistorie des Modells getroffen werden. Der Algorithmus zur Berechnung der Modellklone wird in [Hol11] im Detail beschrieben.

### Funktionen

***klone*(*modell*)** Gibt die Menge aller gefundener Klone in dem Modell *modell* zurück. Dabei ist die Menge aller Klone eine Liste von Paaren $(K, K')$.

## 3.3. Blockmetriken

In diesem Abschnitt werden Metriken vorgestellt, die primär Blöcke und deren Anzahlen berechnen. Die Metriken werden aus Gründen der Übersichtlichkeit in zwei Gruppen aufgeteilt. Zuerst werden die Gruppen der einfachen und dann der erweiterten Blockmetriken vorgestellt. Einfache Blockmetriken berechnen Blockanzahlen auf Grund des Blocktyps. Erweiterte Blockmetriken hingegen berücksichtigen die verbundenen Blöcke oder andere Zusammenhänge zwischen Blöcken, wie z. B. ein Leser-Schreiber-Verhältnis. Die Metriken in diesem Abschnitt verfolgen ein ähnliches Ziel, wie die von Andreas Rau in seiner Dissertation vorgeschlagenen Metriken zur Messung der Modellgröße (siehe [Rau02, S. 83 ff.]). Daher wurden einige dieser Metriken in diese Arbeit übernommen und entsprechend formalisiert.

### 3.3.1. Einfache Blockmetriken

Einfache Blockmetriken berechnen Blockanzahlen aufgrund des Blocktyps. Da sie alle ähnlich aufgebaut sind und sich nur im Blocktyp unterscheiden, enthalten die einzelnen Metriken keine eigene Beschreibung.

**(1) Datenstrukturen** »keine neuen Datenstrukturen«

**(2) Funktionen** »keine neuen Funktionen«

**(3) Metriken**

**Metrik 1:** Anzahl der Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke\}|$$

**Metrik 2:** Anzahl der TargetlinkFunction-Blöcke

| Einheit: [**Blöcke**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘TargetlinkFunction‘}\}|$$

**Metrik 3:** Anzahl der CustomCode-Blöcke

| Einheit: [**Blöcke**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘CustomCode‘}\}|$$

**Metrik 4:** Anzahl der Ground-Blöcke

| Einheit: [**Blöcke**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘Ground‘}\}|$$

**Metrik 5:** Anzahl der Terminator-Blöcke

| Einheit: [**Blöcke**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘Terminator‘}\}|$$

**Metrik 6:** Anzahl der Saturation-Blöcke

| Einheit: [**Blöcke**] | Faktor: Robustheit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘Saturation‘}\}|$$

**Metrik 7:** Anzahl der DataStoreMemory-Blöcke

| Einheit: [**Blöcke**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘DataStoreMemory‘}\}|$$

**Metrik 8:** Anzahl der DataStoreRead-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘DataStoreRead‘}\}|$$

**Metrik 9:** Anzahl der DataStoreWrite-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘DataStoreWrite‘}\}|$$

**Metrik 10:** Anzahl der ModelInfo-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘ModelInfo‘}\}|$$

**Metrik 11:** Anzahl der BusCreator-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘BusCreator‘}\}|$$

**Metrik 12:** Anzahl der BusSelector-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in \mathit{Blöcke} \mid \mathit{typ}\,(b) = \text{‘BusSelector‘}\}|$$

**Metrik 13:** Anzahl der Goto-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in \mathit{Blöcke} \mid \mathit{typ}\,(b) = \text{‘Goto‘}\}|$$

**Metrik 14:** Anzahl der From-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in \mathit{Blöcke} \mid \mathit{typ}\,(b) = \text{‘From‘}\}|$$

**Metrik 15:** Anzahl der UnitDelay-Blöcke

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in \mathit{Blöcke} \mid \mathit{typ}\,(b) = \text{‘UnitDelay‘}\}|$$

### 3.3.2. Erweiterte Blockmetriken

Erweiterte Blockmetriken betrachten nicht nur einzelne Blöcke und ihre Typen, sondern auch z. B. die Vorgänger, Nachfolger und weitere Blockeigenschaften.

**(1) Datenstrukturen** »keine neuen Datenstrukturen«

**(2) Funktionen**

***schreiber*(*b*) ⊆ *Blöcke*** Gibt alle Blöcke vom Typ ‘DataStoreWrite‘ zurück, die in den Block $b$ vom Typ ‘DataStoreMemory‘ schreiben.

***leser*(*b*) ⊆ *Blöcke*** Gibt alle Blöcke vom Typ ‘DataStoreRead‘ zurück, die aus dem Block $b$ vom Typ ‘DataStoreMemory‘ lesen.

***blockset*(*b*)** Gibt den Namen des Blocksets von $b$ zurück (z. B. ‘Simulink‘ oder ‘Targetlink‘).

***saturierung*(*b*)** Gibt zurück, ob die Saturierung des Ausgangssignals eines Targetlink-Blocks eingeschaltet (‘on‘) oder ausgeschaltet (‘off‘) ist.

***orientation*(*b*)** Gibt die Ausrichtung des Blocks $b$ zurück (‘right‘, ‘left‘, ‘up‘ oder ‘down‘).

***sichtbarkeit*(*b*)** Gibt die Sichtbarkeit eines Blocks vom Typ ‘Goto‘ zurück (‘local‘, ‘scoped‘ oder ‘global‘).

***vorgänger*** $(b) \subseteq$ ***Blöcke*** Gibt eine Menge von Blöcken $Blöcke' \subseteq Blöcke$ zurück, die über einen Inport von $b$ verbunden sind. Somit gilt für alle $b' \in Blöcke'$ der Zusammenhang $(b', b) \in Linien$.

***nachfolger*** $(b) \subseteq$ ***Blöcke*** Gibt eine Menge von Blöcken $Blöcke' \subseteq Blöcke$ zurück, die über einen Outport von $b$ verbunden sind. Somit gilt für alle $b' \in Blöcke'$ der Zusammenhang $(b, b') \in Linien$.

### (3) Metriken

**Metrik 16:** Anzahl der DataStoreMemory-Blöcke ohne Schreibzugriff

| Einheit: [**Blöcke**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{`DataStoreMemory`} \wedge schreiber(b) = \emptyset\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der nicht geschriebenen DataStoreMemory-Blöcke. Nicht geschriebene DataStoreMemory-Blöcke haben keine dazugehörigen DataStoreWrite-Blöcke, von denen Werte in den Speicher geschrieben werden.

**Metrik 17:** Anzahl der DataStoreMemory-Blöcke ohne Lesezugriff

| Einheit: [**Blöcke**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{`DataStoreMemory`} \wedge leser(b) = \emptyset\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der nicht gelesenen DataStoreMemory-Blöcke. Nicht gelesene DataStoreMemory-Blöcke haben keine dazugehörigen DataStoreRead-Blöcke, von denen Werte aus dem Speicher gelesen werden.

**Metrik 18:** Anzahl der aktivierten Targetlink Block-Saturierungen

| Einheit: [**Blöcke**] | Faktor: Robustheit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid blockset(b) = \text{`Targetlink`} \wedge saturierung(b) = \text{`on`}\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Targetlink-Blöcke, bei denen eine Saturierung des Ausgangssignals stattfindet. Das bedeutet, dass bei diesen Blöcken bei einem Über- oder Unterlauf dafür gesorgt wird, dass das Ausgangssignal den größten bzw. kleinsten erlaubten Wert annimmt.

**Metrik 19:** Anzahl der UnitDelay-Blöcke in Vorwärtsrichtung

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{`UnitDelay`} \wedge orientation(b) = \text{`left`}\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der UnitDelay-Blöcke, deren Signalfluss von links nach rechts zeigt, d. h. in Vorwärtsrichtung stattfindet. Die Anzahl dieser Blöcke ist von Interesse, da es sich bei einem UnitDelay-Block um eine Rückkopplung handelt. Rückkopplungen sollten zur besseren Erfassbarkeit von rechts nach links modelliert werden.

**Metrik 20:** Anzahl der globalen Goto-Blöcke

| Einheit: [**Blöcke**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{‘Goto‘} \wedge sichtbarkeit(b) = \text{‘global‘}\}|$$

**Beschreibung**: Diese Metrik zählt die Anzahl der Goto-Blöcke mit einem globalen Geltungsbereich. Das bedeutet, dass von From-Blöcken an einer beliebigen Stelle des Modells zu diesen Goto-Blöcken gesprungen werden kann. Dieser Sprung ist unabhängig von der Subsystemhierarchie.

**Metrik 21:** Durchschnittliche Anzahl an nachfolgenden Blöcken

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{b \in Blöcke} |nachfolger(b)|}{|Blöcke|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Anzahl an nachfolgenden Blöcken pro Block. Die Anzahl der nachfolgenden Blöcke ist für jeden Block größer gleich der Anzahl seiner Outports. Ein Block hat mehr Nachfolger als Outports, wenn sich Linien an seinen Outports aufspalten (Mehrfachkanten).

**Metrik 22:** Durchschnittliche Blockinstabilität

| Einheit: [**Änderungswahrscheinlichkeiten**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{b \in Blöcke} \frac{|vorgänger(b)|}{|vorgänger(b)| + |nachfolger(b)|}}{|Blöcke|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Stabilität der Blöcke eines Simulink-Modells. Die Stabilität eines Blocks sagt aus, wie wahrscheinlich es ist, dass der Block geändert werden muss. Dazu wird die Anzahl der Vorgängerblöcke durch die Summe der Anzahl der Vorgängerblöcke und der Nachfolgerblöcke geteilt. Daher hat ein Block mit mehr Vorgängerblöcken als Nachfolgerblöcken eine höhere Änderungswahrscheinlichkeit. Denn je mehr Vorgänger ein Block besitzt, desto eher muss er aufgrund einer Änderung an den Vorgängerblöcken auch angepasst werden. Diese Metrik wurde ursprünglich von Menkhaus und Andrich in [MA05] für eine Fehler-Möglichkeits- und Einflussanalyse von Simulink-Modellen verwendet und in [Hol11, S. 63] für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

**Metrik 23:** Anzahl der verwendeten Blocktypen

| Einheit: [**Blocktypen**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = \left| \left( \bigcup_{b \in Blöcke} typ\,(b) \right) \right|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der in einem Simulink-Modell verwendeten Blocktypen.

## 3.4. Konstantenmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit Constant-Blöcken und deren Eigenschaften beschäftigen.

### (1) Datenstrukturen

***MagischeKonstantenBlöcke*** $\subseteq$ ***Blöcke*** Die Menge *MagischeKonstantenBlöcke* enthält alle Constant-Blöcke mit magischen Konstanten. Magische Konstanten sind numerische Werte, die magisch genannt werden, da sie keinen Namen besitzen und ihre Bedeutung daher nicht direkt ersichtlich ist. Numerische Konstanten mit einem Wert von 1 und 0 werden nicht als magische Konstanten behandelt, da ihre Verwendung als *true* und *false* in Vergleichsoperationen o. Ä. Konvention ist. Die Menge ist definiert als:

$$MagischeKonstantenBlöcke = \{ b \in Blöcke \mid typ\,(b) = \text{‘Constant‘} \wedge istNumerisch\,(wert\,(b)) \wedge \neg\,(wert\,(b) = 0 \vee wert\,(b) = 1) \}$$

### (2) Funktionen

***wert*** $(b)$ Gibt den Konstantenwert eines Blocks vom Typ ‘Constant‘ zurück.

***istNumerisch*** $(wert\,(b))$ Gibt als Wahrheitswert zurück, ob der Konstantenwert des Blocks $b$ ein numerischer Wert ist.

### (3) Metriken

**Metrik 24:** Anzahl der magischen Konstanten

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |MagischeKonstantenBlöcke|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Constant-Blöcke, die einen magischen Wert verwenden. Magische Werte sind numerische Werte, deren Bedeutung ohne weitere Erläuterungen nicht ersichtlich ist.

**Metrik 25:** Anzahl der Werte magischer Konstanten

| Einheit: [**Konstantenwerte**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = \left| \bigcup_{b \in MagischeKonstantenBlöcke} wert(b) \right|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl verwendeter Werte magischer Konstanten, d.h. es wird die Anzahl unterschiedlicher numerischer Werte gezählt.

**Metrik 26:** Anzahl der Konstanten mit Namen

| Einheit: [**Blöcke**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid typ(b) = \text{`Constant'}\} \setminus MagischeKonstantenBlöcke|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Constant-Blöcke, die keinen numerischen Wert, sondern einen Bezeichner mit sprechendem Namen verwenden.

## 3.5. Linienmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit den Linien und deren Eigenschaften beschäftigen.

### (1) Datenstrukturen

***LinienPaare*** $\subseteq$ ***Linien*** $\times$ ***Linien*** Die Menge aller eindeutigen Linienpaare, wobei $l_1, l_2 \in Linien$ sind und $f\colon Linien \to \mathbb{N}$ eine injektive Abbildung ist. Somit ergibt sich:

$$LinienPaare = \{(l_1, l_2) \in Linien \times Linien \mid f(l_1) < f(l_2)\}$$

Die Menge *LinienPaare* ist das Kreuzprodukt der Menge aller Linien *Linien* mit sich selbst, ohne die Paare, in denen dieselbe Linie zweimal enthalten ist und ohne die Paare, die durch Vertauschung von $l_1$ und $l_2$ eines anderen Paares entstehen.

### (2) Funktionen

***anzahlÜberkreuzungen*** $(\langle \boldsymbol{l}_1, \boldsymbol{l}_2 \rangle)$ Gibt die Anzahl der Schnittpunkte der beiden Linien $l_1$ und $l_2$ zurück.

### (3) Metriken

**Metrik 27:** Anzahl der Linien

| Einheit: [**Linien**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{l \in Linien\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Linien in einem Simulink-Modell.

**Metrik 28:** Anzahl der Linien-Annotationen

| Einheit: [**Liniennamen**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{l \in Linien \mid name(l) \neq \text{“}\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der nicht leeren Linien-Annotationen in einem Simulink-Modell.

**Metrik 29:** Anzahl der überkreuzenden Linien

| Einheit: [**Linienüberschneidungen**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = \sum_{\langle l_1, l_2 \rangle \in LinienPaare} anzahlÜberkreuzungen(\langle l_1, l_2 \rangle)$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der überkreuzenden Linien in einem Simulink-Modell. Dazu werden für jedes Linienpaar die Überschneidungen berechnet und aufsummiert.

## 3.6. Subsystemmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit den Subsystemen in einem Simulink-Modell beschäftigen.

### (1) Datenstrukturen

***TestbareSubsysteme*** $\subseteq$ ***Subsysteme*** Die Menge aller einzeln testbarer Subsysteme. In dieser Arbeit werden die einzeln testbaren Subsysteme dadurch definiert, dass sie nicht leer und auf tiefster Ebene in der Subsystemhierarchie sind. Daher ergibt sich folgende Definition:

$$TestbareSubsysteme = \{s \in Subsysteme \mid subblöcke(s) \neq \emptyset \wedge subblöcke(subblöcke(s)) = \emptyset\}$$

Bei nicht leeren Subsystemen auf tiefster Ebene wird davon ausgegangen, dass sie einzeln in Unit-Tests testbar sind.

### (2) Funktionen

***istAtomar*** $(s)$ Gibt als Wahrheitswert zurück, ob das Subsystem von Simulink in der Simulation als atomare Einheit behandelt werden soll oder nicht. Bei atomaren Subsystemen wird vor der Ausführung sicher gestellt, dass alle Eingangssignale zur Verfügung stehen.

***arithmetischeBlöcke*** $(s) \subseteq$ ***subblöcke*** $(s)$ Gibt die Menge aller Subblöcke eines Subsystems zurück, die arithmetische Operationen durchführen.

***logischeBlöcke*** $(s) \subseteq$ ***subblöcke*** $(s)$ Gibt die Menge aller Subblöcke eines Subsystems zurück, die logische Operationen durchführen.

***signalführendeBlöcke*** $(s) \subseteq$ ***subblöcke*** $(s)$ Gibt die Menge aller Subblöcke eines Subsystems zurück, die nur der Signalführung dienen. Das sind zum Beispiel Blöcke vom Typ ‘BusCreator‘ oder ‘Goto‘.

***schaltendeBlöcke*** $(s) \subseteq$ ***subblöcke*** $(s)$ Gibt die Menge aller Subblöcke eines Subsystems zurück, die zwischen verschiedenen Signalen umschalten. Das sind zum Beispiel Blöcke vom Typ 'Switch' oder 'MultiportSwitch'.

***datenTyp*** $(b)$ Gibt den Simulink-Datentyp des Ausgangssignals eines Blocks zurück. Wurde kein Datentyp explizit gewählt, ist er standardmäßig 'Inherit: auto'.

***höheInPixel*** $(s)$ Gibt die Höhe der grafischen Darstellung eines Subsystems $s$ in [**Pixel**] zurück.

***breiteInPixel*** $(s)$ Gibt die Breite der grafischen Darstellung eines Subsystems $s$ in [**Pixel**] zurück.

***istBusSignal*** $(b)$ Gibt als Wahrheitswert zurück, ob das Signal eines Blocks $b$ vom Typ 'Inport' ein Bussignal ist.

### (3) Metriken

**Metrik 30:** Anzahl der Subsysteme

| Einheit: [**Subsysteme**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{s \in Subsysteme\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Subsysteme in einem Simulink-Modell.

**Metrik 31:** Anzahl der konfigurierbaren Subsysteme

| Einheit: [**Subsysteme**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\#} = |\{s \in Subsysteme \mid typ(s) = \text{'ConfigurableSubsystem'}\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der konfigurierbaren Subsysteme in einem Simulink-Modell.

**Metrik 32:** Anzahl der atomaren Subsysteme

| Einheit: [**Subsysteme**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |\{s \in Subsysteme \mid typ(s) = \text{'Subsystem'} \wedge istAtomar(s)\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der atomaren Subsysteme in einem Simulink-Modell.

**Metrik 33:** Durchschnittliche Anzahl an Subsystemen pro Subsystem

| Einheit: [**Subsysteme**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{s \in Subsysteme} |\{b \in subblöcke(s) \mid typ(b) = \text{'Subsystem'}\}|}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Anzahl an Subsystemen pro Subsystem, d.h. wie viele Subsysteme durchschnittlich in einem Subsystem enthalten sind.

**Metrik 34:** Anzahl der arithmetischen Subsysteme

| Einheit: [**Subsysteme**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = \left| \left\{ s \in Subsysteme \mid \frac{|arithmetischeBlöcke\,(subblöcke\,(s))|}{|subblöcke\,(s)|} > schwellwert \right\} \right|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der arithmetischen Subsysteme. Subsysteme werden als arithmetisch klassifiziert, wenn ihr Anteil an Blöcken mit arithmetischen Funktionen größer ist als von dem Schwellwert *schwellwert* gefordert.

**Metrik 35:** Anzahl der logischen Subsysteme

| Einheit: [**Subsysteme**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = \left| \left\{ s \in Subsysteme \mid \frac{|logischeBlöcke\,(subblöcke\,(s))|}{|subblöcke\,(s)|} > schwellwert \right\} \right|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der logischen Subsysteme. Subsysteme werden als logisch klassifiziert, wenn ihr Anteil an Blöcken mit logischen Funktionen größer ist als von dem Schwellwert *schwellwert* gefordert.

**Metrik 36:** Anzahl der signalführenden Subsysteme

| Einheit: [**Subsysteme**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = \left| \left\{ s \in Subsysteme \mid \frac{|signalführendeBlöcke\,(subblöcke\,(s))|}{|subblöcke\,(s)|} > schwellwert \right\} \right|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der signalführenden Subsysteme. Subsysteme werden als signalführend klassifiziert, wenn ihr Anteil an Blöcken, die nur zur Führung der Signale dienen, größer ist als von dem Schwellwert *schwellwert* gefordert.

**Metrik 37:** Anzahl der schaltenden Subsysteme

| Einheit: [**Subsysteme**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = \left| \left\{ s \in Subsysteme \mid \frac{|schaltendeBlöcke\,(subblöcke\,(s))|}{|subblöcke\,(s)|} > schwellwert \right\} \right|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der schaltenden Subsysteme. Subsysteme werden als schaltend klassifiziert, wenn ihr Anteil an Blöcken, die nur zwischen Signalen umschalten, größer ist als von dem Schwellwert *schwellwert* gefordert.

**Metrik 38:** Anzahl der testbaren Subsysteme mit impliziten Inport-Datentypen

| Einheit: [**Subsysteme**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\#} = |\{s \in TestbareSubsysteme \mid \exists b\,(b \in subblöcke\,(s) \wedge typ\,(b) = \text{‘Inport‘} \wedge datenTyp\,(b) = \text{‘Inherit: auto‘})\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der testbaren Subsysteme, bei denen für mindestens einen Inport-Block kein expliziter Datentyp festgelegt wurde.

**Metrik 39:** Anzahl der testbaren Subsysteme mit eingehenden Bussen

| Einheit: [**Subsysteme**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\#} = |\{s \in TestbareSubsysteme \mid \exists b\,(b \in subblöcke\,(s) \wedge typ\,(b) = \text{‘Inport‘} \wedge istBusSignal\,(b))\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der testbaren Subsysteme, die einen oder mehrere eingehende Busse besitzen.

**Metrik 40:** Tiefe der Subsystem-Hierarchie

| Einheit: [**Subsystemhierarchieebenen**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = \max_{s \in Subsysteme} (tiefe\,(s))$$

**Beschreibung**: Diese Metrik berechnet die Tiefe der Subsystem-Hierarchie. Dabei wird die Subsystemhierarchie als Baum betrachtet. Zurückgeliefert wird die maximale Tiefe dieses Baums.

**Metrik 41:** Durchschnittliche Höhe der Subsysteme auf dem Bildschirm

| Einheit: [**Pixel**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{s \in Subsysteme} höheInPixel\,(s)}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Höhe der Subsysteme auf dem Bildschirm.

**Metrik 42:** Durchschnittliche Breite der Subsysteme auf dem Bildschirm

| Einheit: [**Pixel**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{s \in Subsysteme} breiteInPixel\,(s)}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Breite der Subsysteme auf dem Bildschirm.

## 3.7. Annotationsmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit den Annotationen in einem Simulink-Modell beschäftigen. Annotationen sind Textboxen mit Kommentaren, die in Subsystemen platziert werden können. Sie verfolgen die gleiche Idee, wie die von Andreas Rau in seiner Dissertation vorgeschlagenen Metriken zur Messung der Lesbarkeit eines Simulink-Modells (siehe [Rau02, S. 85]).

**(1) Datenstrukturen** »keine neuen Datenstrukturen«

**(2) Funktionen**

***länge*** (***annotation***) $\in \mathbb{N}$ Gibt die Länge einer Annotation als Anzahl der verwendeten Zeichen zurück.

**(3) Metriken**

**Metrik 43:** Anzahl der Subsystem-Annotationen

| Einheit: [**Kommentare**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |Annotationen|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Subsystem-Annotationen in einem Simulink-Modell.

**Metrik 44:** Durchschnittliche Länge der Subsystem-Annotationen

| Einheit: [**Zeichen**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{annotation \in Annotationen} länge\,(annotation)}{|Annotationen|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Kommentarlänge aller Kommentare eines Simulink-Modells. In [Hol11, S. 41] wurde die Metrik für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

## 3.8. Bibliotheksmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit den Bibliotheken in einem Simulink-Modell beschäftigen. Dabei steht die Art und Weise der Verwendung von Blöcken aus Bibliotheken im Fokus und nicht der Aufbau der Bibliotheken an sich.

**(1) Datenstrukturen** »keine neuen Datenstrukturen«

**(2) Funktionen**

***istAusBibliothek*(*b*)** Gibt als Wahrheitswert zurück, ob der Block $b$ aus einer Bibliothek stammt. Dabei ist die Standard-Bibliothek von Simulink ausgeschlossen, da deren Blöcke nicht als Bibliotheksblöcke gezählt werden. Sie stellt die grundlegenden Bausteine für alle anderen Blöcke bereit.

***istVerknüpfungDeaktiviert*(*b*)** Gibt als Wahrheitswert zurück, ob die Verknüpfung des Bibliotheksblocks $b$ deaktiviert ist. Sobald eine Verknüpfung deaktiviert wird, wird der referenzierte Block aus der Bibliothek mit all seinen Subblöcken in das Modell kopiert.

***istBibliothekUnauffindbar*(*b*)** Gibt als Wahrheitswert zurück, ob die Bibliothek unauffindbar ist, aus der der Block $b$ stammt.

***istVerändert*(*b*)** Gibt als Wahrheitswert zurück, ob der Bibliotheksblock $b$ verändert wurde. Bibliotheksblöcke können nur verändert werden, wenn für sie $istVerknüpfungDeaktiviert(b)$ gilt.

***bibliotheksname*(*b*)** Gibt den Namen der Bibliothek zurück, aus der der Block $b$ stammt.

### (3) Metriken

**Metrik 45:** Anzahl der verwendeten Bibliotheksverknüpfungen

| Einheit: [**Blöcke**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid istAusBibliothek(b)\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der verwendeten Bibliotheksverknüpfungen, d. h. die Anzahl der Blöcke aus Bibliotheken.

**Metrik 46:** Anzahl der deaktivierten Bibliotheksverknüpfungen

| Einheit: [**Blöcke**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid istAusBibliothek(b) \wedge istVerknüpfungDeaktiviert(b)\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Bibliotheksverknüpfungen mit deaktivierter Verknüpfung.

**Metrik 47:** Anzahl der deaktivierten und unauffindbaren Bibliotheksverknüpfungen

| Einheit: [**Blöcke**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid istAusBibliothek(b) \wedge istVerknüpfungDeaktiviert(b) \wedge istBibliothekUnauffindbar(b)\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Bibliotheksverknüpfungen mit deaktivierter Verknüpfung und unauffindbarem Originalblock.

**Metrik 48:** Anzahl der deaktivierten Bibliotheksverknüpfungen mit veränderter Struktur

| Einheit: [**Blöcke**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid istAusBibliothek(b) \wedge istVerknüpfungDeaktiviert(b) \wedge istVerändert(b) \wedge \neg istBibliothekUnauffindbar(b)\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Bibliotheksverknüpfungen mit deaktivierter Verknüpfung und geändertem Inhalt. Bibliotheksverknüpfungen mit verändertem Inhalt werden dadurch erkannt, dass ihre Struktur mit der Struktur des Originalblocks aus der Bibliothek verglichen wird. Sobald die Anzahl der Blöcke oder Linien unterschiedlich ist, wurde der Block entweder im Modell oder in der Bibliothek verändert.

**Metrik 49:** Anzahl der verwendeten Bibliotheken

| Einheit: [**Bibliotheksnamen**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = \left| \left( \bigcup_{b \in \{b \in Blöcke | istAusBibliothek(b)\}} bibliotheksname(b) \right) \right|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der verwendeten Bibliotheken. Dazu werden die Namen aller verwendeter Bibliotheken gesammelt und gezählt.

## 3.9. Signalflussmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit dem Signalfluss in einem Simulink-Modell beschäftigen. Die Metriken in diesem Abschnitt können detaillierte Aussagen über die Struktur der Modelle machen, da sie die Semantik der Simulink-Modelle berücksichtigen.

### (1) Datenstrukturen

***BusCreators*** $\subseteq$ ***Blöcke*** Die Menge aller BusCreator-Blöcke mit:

$$BusCreators = \{b \in Blöcke \mid typ(b) = \text{‘BusCreator‘}\}$$

***Inports*** $\subseteq$ ***Blöcke*** Die Menge aller Inport-Blöcke mit:

$$Inports = \{b \in Blöcke \mid typ(b) = \text{‘Inport‘}\}$$

***Outports*** $\subseteq$ ***Blöcke*** Die Menge aller OutportBlöcke mit:

$$Outports = \{b \in Blöcke \mid typ(b) = \text{‘Outport‘}\}$$

### (2) Funktionen

***alleSignaleVerfallen*** (***b***) Gibt als Wahrheitswert zurück, ob alle Signale, die einen Block verlassen, direkt oder indirekt in einem Terminator-Block enden.

***länge*** (***misp***) $\in \mathbb{N}^+$ Gibt die Länge, d. h. die Anzahl der Linien eines MISPs zurück.

***längeInPixel*** (***misp***) $\in \mathbb{N}^+$ Gibt die Länge eines MISPs auf dem Bildschirm zurück.

***signalpfadVorgänger*** (***misp***) $\subseteq$ ***MISPs*** Gibt alle MISPs zurück, die direkt oder indirekt Einfluss auf den übergebenen MISP haben. Zur Ermittlung, welche MISPs Einfluss auf einen untersuchten MISP haben, wird von dem untersuchten MISP aus rekursiv nach vorne gegangen. Dabei werden die Vorgänger-MISPs über Subsystemgrenzen hinweg betrachtet.

***misp*** (***s***) $\subseteq$ ***MISPs*** Gibt alle MISPs zurück, die in dem Subsystem $s$ beginnen.

***anzahlSignale*** (***b***) $\in \mathbb{N}^+$ Liefert die Anzahl der Signale, die aus einem Block heraus oder in ihn hinein gehen. Bei Inport-Blöcken und BusCreator-Blöcken wird das ausgehende Signal, bei Outport-Blöcken und BusSelector-Blöcken das eingehende Signal betrachtet. Ist das betrachtete Signal nicht vom Typ 'Bus' und 'Nicht-Scalar', ist die Anzahl der Signale 1. Bei Signalen vom Typ 'Nicht-Scalar' ist die Anzahl die Dimension des Signals. Im Falle eines Typs 'Bus' wird rekursiv die Anzahl der Signale in dem Bus gezählt.

***zyklen*** (***s***) Gibt die Menge aller Signalpfade eines Subsystems zurück, die einen Zyklus bilden. Für einen Signalpfad mit Zyklus gilt $zyklus = (l_1, l_2, ..., l_{n-1}, l_1)$.

***interneZustände*** (***modell***) Gibt die Menge aller internen Zustände eines Simulink-Modells zurück.

***wertebereichsverletzungen*** (***modell***) Gibt die Menge der gefundenen Wertebereichsverletzungen von Signalen zurück, die an den Outport-Blöcken eines Modell auftreten.

### (3) Metriken

**Metrik 50:** Durchschnittliche Anzahl der unabhängigen Berechnungspfade

| Einheit: [**(unabhängige) Berechnungspfade**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\#} = \frac{\sum_{s \in Subsysteme} |berechnungspfade\,(s)|}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Anzahl der unabhängigen Berechnungspfade (vergleiche Abschnitt 3.2.2) pro Subsystem. Dazu analysiert die Metrik alle Subsysteme und summiert die Anzahl der gefundenen unabhängigen Berechnungspfade auf. In [Hol11, S. 42] wurde die Metrik für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

**Metrik 51:** Anzahl der Zyklen

| Einheit: [**Zyklen**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = \sum_{s \in Subsysteme} |zyklen\,(s)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Zyklen in allen Subsystemen eines Simulink-Modells. In [Hol11, S. 73] wurde die Metrik für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

**Metrik 52:** Anzahl der verfallenen Ergebnisse

| Einheit: [**Ergebnisse**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid alleSignaleVerfallen(b)\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der verfallenen Ergebnisse in einem Simulink-Modell. Dazu wird für jeden Outport aller MISP-Erzeuger (vergleiche Abschnitt 3.2.1) geprüft, ob deren Signale alle direkt oder indirekt in einem Terminator-Block enden und somit verfallen.

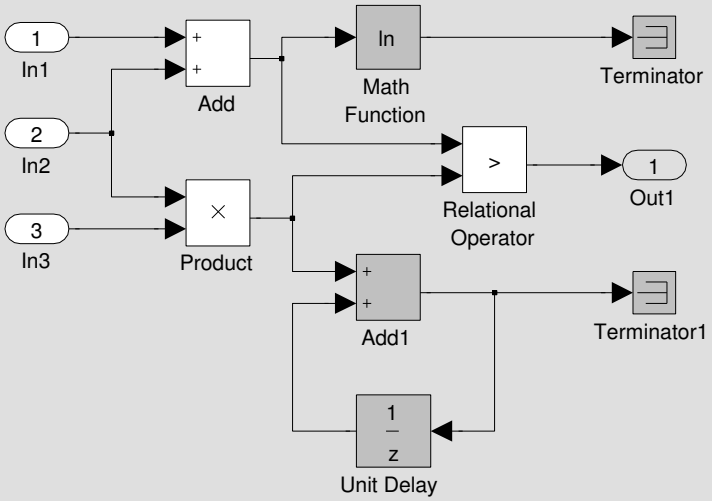



Abbildung 3.7.: Beispiel verfallener Ergebnisse [Hol11, S. 67]

Abbildung 3.7 zeigt beispielhaft verfallene Ergebnisse, die in den Terminator-Blöcken enden, ohne verwendet zu werden. In [Hol11, S. 67] wurde die Metrik für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

**Metrik 53:** Durchschnittlicher Vernetzungsgrad

| Einheit: [**Signalpfade**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{s \in Subsysteme} \frac{|signalpfadVorgänger(misp(s))|}{|misp(s)|}}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet pro Subsystem die Anzahl der MISPs, die durchschnittlich Einfluss auf einen einzelnen MISP haben. Dazu werden für jeden MISP alle Vorgänger ermittelt, die Einfluss auf den betrachteten MISP haben. Unter Einfluss wird hier verstanden, dass ein anderer MISP in einem Block endet, in dem der betrachtete beginnt.
Dieser Ansatz ist vergleichbar mit dem Vorgehen des Rückwärts Slicings in der Compilertechnik, bei dem alle Programmteile identifiziert werden, die Einfluss auf die gewählte Programmstelle haben. In [Hol11, S. 80] wurde die Metrik für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

**Metrik 54:** Durchschnittliche Signalpfadlänge

| Einheit: [**Signalpfade**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{misp \in MISPs} länge\,(misp)}{|MISPs|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Signallänge aller Maximalen-Identitäts-Signalpfade. In [Hol11, S. 77] wurde die Metrik für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

**Metrik 55:** Durchschnittliche Länge der Signalpfade auf dem Bildschirm

| Einheit: [**Pixel**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{misp \in MISPs} längeInPixel\,(misp)}{|MISPs|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Signallänge aller Maximalen-Identitäts-Signalpfade auf dem Bildschirm.

**Metrik 56:** Durchschnittliche Breite der Eingangsschnittstelle

| Einheit: [**Signale/Subsystem**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{b \in Inports} anzahlSignale\,(b)}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Breite der Eingangsschnittstelle. Dazu wird für jeden Inport die Anzahl der Signale gezählt. Diese Metrik wurde bereits von Andreas Rau in seiner Dissertation [Rau02, S. 84] vorgeschlagen.

**Metrik 57:** Durchschnittliche Breite der Ausgangsschnittstelle

| Einheit: [**Signale/Subsystem**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{b \in Outports} anzahlSignale\,(b)}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Breite der Ausgangsschnittstelle. Dazu wird für jeden Outport die Anzahl der Signale gezählt. Diese Metrik wurde bereits von Andreas Rau in seiner Dissertation [Rau02, S. 84] vorgeschlagen.

**Metrik 58:** Durchschnittliche Busgröße

| Einheit: [**Signale/Bus**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{b \in BusCreators} anzahlSignale\,(b)}{|BusCreators|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Größe der Busse in einem Simulink-Modell. Dazu werden alle ausgehenden Signale der BusCreator-Blöcke untersucht, da nur BusCreator-Blöcke neue Busse erzeugen können.

**Metrik 59:** Anzahl der internen Zustände

| Einheit: [**Interne Zustände**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |interneZustände\,(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der internen Zustände eines Simulink-Modells. Interne Zustände kommen dadurch zustande, dass Werte zwischen zwei Abtastzyklen des Modells zwischengespeichert werden, d. h. die Ausgabe des Modells ist nicht mehr ausschließlich von der aktuellen Eingabe abhängig. Interne Zustände kommen z. B. durch die Verwendung von UnitDelay-Blöcken oder Integrator-Blöcken zustande.

**Metrik 60:** Anzahl der Wertebereichsverletzungen

| Einheit: [**Wertebereichsverletzungen**] | Faktor: Robustheit |
|---|---|

$$messwert_{\#} = |wertebereichsverletzungen\,(modell)|$$

**Beschreibung**: Die Metrik berechnet die Anzahl der Wertebereichsverletzungen in einem Simulink-Modell. Die Wertebereichsverletzungen werden mittels Model Checking ermittelt (vergleiche beispielsweise [EFJ10]). Durch das Durchführen des Model Checkings direkt auf dem Simulink-Modell werden auch die gefundenen Probleme direkt im Modell angezeigt. Eine Wertebereichsverletzung tritt beispielsweise auf, wenn ein Signalwert an einem Outport-Block größer wird als der größte mögliche Wert des verwendeten Datentyps.

## 3.10. Strukturmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit strukturellen Eigenschaften eines Simulink-Modells beschäftigen. Das umfasst beispielsweise die Architektur und die Komplexität der Modelle.

**(1) Datenstrukturen** »keine neuen Datenstrukturen«

**(2) Funktionen**

***modellierungsmuster*(*modell*)** Gibt die Menge aller in einem Modell *modell* verwendeter Modellierungsmuster zurück.

***globaleKomplexität*** (***modell***) $\in \mathbb{N}^+$ Gibt die Höhe der globalen Komplexität eines Simulink-Modells *modell* zurück.

***lokaleKomplexität*** (***s***) $\in \mathbb{N}^+$ Gibt die Höhe der lokalen Komplexität des Subsystems *s* zurück.

***architekturverletzungen*** (***modell***) Gibt die Menge aller gefundenen Architekturverletzungen eines Simulink-Modells *modell* zurück.

***richtlinienverletzungen*** (***modell***, **'Kategorie'**) Gibt die Richtlinienverletzungen eines Modells *modell* zurück. Dabei kann die gewünschte Kategorie, d. h. die zu verwendenden Richtlinien, angegeben werden.

### (3) Metriken

**Metrik 61:** Anzahl der Modellklone

| Einheit: [**Klone**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = |klone\,(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der geklonten Modellteile in einem Modell. In [Hol11, S. 50] wurde die Metrik für das Framework dieses Ansatzes angepasst (siehe Kapitel 6).

**Metrik 62:** Anzahl der verwendeten Modellierungsmuster

| Einheit: [**Modellierungsmuster**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = |modellierungsmuster\,(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der verwendeten Modellierungsmuster. Dazu werden Muster aus einem Katalog in dem Modell gesucht und ihr Vorkommen gezählt. Der Katalog enthält bewährte Muster, d. h. Modellausschnitte mit bestimmten Konstellationen aus Blöcken und Linien. Er kann beispielsweise aus Richtliniensammlungen wie z. B. den MAAB (MathWorks Automotive Advisory Board) Richtlinien[a] entnommen werden. Abschnitt A.1.1 im Anhang enthält einige Beispielmuster.

[a] `http://www.mathworks.de/automotive/standards/maab.html?`

**Metrik 63:** Höhe der globalen Komplexität

| Einheit: [**Komplexitätspunkte**] | Faktor: Wartbarkeit |
|---|---|

$$messwert_{\#} = |globaleKomplexität\,(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Komplexität eines Simulink-Modells. Dazu wird ein für Simulink-Modelle angepasster Halstead-Algorithmus verwendet. Die Anpassung beschreiben Stürmer et al. in [SPR10] im Detail.

**Metrik 64:** Durchschnittliche lokale Komplexität

| Einheit: [**Komplexitätspunkte**] | Faktor: Testbarkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{s \in Subsysteme} |lokaleKomplexität(s)|}{|Subsysteme|}$$

**Beschreibung**: Diese Metrik berechnet die Komplexität eines Subsystems. Es wird wie in Metrik 63 der angepasste Halstead-Algorithmus verwendet.

**Metrik 65:** Anzahl der Architekturverletzungen

| Einheit: [**Architekturverletzungen**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |architekturverletzungen(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Architekturverletzungen. Dazu wird die tatsächlich im Simulink-Modell enthaltene Architektur mit einer vorgegebenen Architektur-Beschreibung verglichen und Abweichungen gezählt. In den Simulink-Modellen findet sich die Architektur in Form von vorgegebenen Subsystemen und erlaubten Linien wieder. Die vorgegebenen Subsysteme sind Komponenten und die erlaubten Linien stellen Schnittstellen zwischen den Komponenten dar.

**Metrik 66:** Anzahl der verletzten dSPACE-Richtlinien

| Einheit: [**Richtlinienverletzungen**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |richtlinienverletzungen(modell, \text{`dSPACE'})|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der verletzten dSPACE-Richtlinien[a], die sich direkt auf Probleme bei der Codegenerierung beziehen. Tabelle A.1 im Anhang listet die verwendeten Richtlinien auf.

[a] `http://www.dspace.de/de/gmb/home/support/kb/kbtl/tlmodguide/tlapp_modelguide.cfm`

**Metrik 67:** Anzahl der verletzten MISRA TL-Richtlinien

| Einheit: [**Richtlinienverletzungen**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |richtlinienverletzungen(modell, \text{`MISRA TL'})|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der verletzten MISRA TL-Richtlinien[a], die sich direkt auf Probleme bei der Codegenerierung beziehen. Tabelle A.2 im Anhang listet die verwendeten Richtlinien auf.

[a] `http://www.misra.org.uk/Publications/tabid/57/Default.aspx`

## 3.11. Stateflow-Metriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit den Stateflow-Diagrammen in einem Simulink-Modell beschäftigen.

### (1) Datenstrukturen

***StateflowBlöcke*** Die Menge aller Stateflow-Blöcke:

$$StateflowBlöcke = \{b \in Blöcke \mid typ(b) = \text{‘Stateflow‘}\}$$

### (2) Funktionen

***zustände*** (***stateflow***) Gibt die Menge aller Zustände zurück, die in dem Stateflow-Diagramm des übergebenen Stateflow-Blocks $stateflow \in StateflowBlöcke$ verwendet werden.

***unerreichbareZustände*** (***stateflow***) Gibt die Menge aller nicht erreichbaren Zustände zurück, die in dem Stateflow-Diagramm des übergebenen Stateflow-Blocks $stateflow \in StateflowBlöcke$ enthalten sind.

### (3) Metriken

**Metrik 68:** Anzahl der Stateflow-Blöcke

| Einheit: [**Stateflow-Diagramme**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\#} = |StateflowBlöcke|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Stateflow-Blöcke und somit der Stateflow-Digramme in einem Simulink-Modell.

**Metrik 69:** Durchschnittliche Anzahl an Stateflow-Zuständen

| Einheit: [**Stateflow-Zustände/Stateflow-Diagramme**] | Faktor: Verständlichkeit |
|---|---|

$$messwert_{\varnothing} = \frac{\sum_{stateflow \in StateflowBlöcke} |zustände(stateflow)|}{|StateflowBlöcke|}$$

**Beschreibung**: Diese Metrik berechnet die durchschnittliche Anzahl an Zuständen pro Stateflow-Diagramm.

**Metrik 70:** Anzahl der unerreichbaren Zustände

| Einheit: [**(unerreichbare) Stateflow-Zustände**] | Faktor: Robustheit |
|---|---|

$$messwert_{\#} = \sum_{stateflow \in StateflowBlöcke} |unerreichbareZustände(stateflow)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der unerreichbaren Zustände in den Stateflow-Diagrammen. So verwenden beispielsweise Hamon and Rushby in [HR07] ihre Laufzeitsemantik von Stateflow für eine Überführung in eine Model Checking-Sprache. Mithilfe des Model Checkers kann dann geprüft werden, welche Zustände unerreichbar sind.

## 3.12. Codegeneratormetriken

In diesem Abschnitt werden die Metriken vorgestellt, die sich mit Eigenschaften beschäftigen, die für die Codegenerierbarkeit eines Simulink-Modells wichtig sind.

### (1) Datenstrukturen

***RechenintensiveBlöcke*** $\subseteq$ ***Blöcke*** Die Menge *RechenintensiveBlöcke* enthält alle rechenintensiven Blöcke. Rechenintensive Blöcke sind Blöcke, deren Bearbeitung viele Prozessorzyklen benötigen. Die Menge ist definiert als:

$$RechenintensiveBlöcke = \{b \in Blöcke \mid typ(b) = \text{‘Math‘} \vee typ(b) = \text{‘TrigonometricFunction‘}\}$$

### (2) Funktionen

***targetlinkFehler*** (***modell***) Gibt die Menge der Targetlink-Fehler zurück, die bei der Codegenerierung des Modells *modell* auftreten.

***targetlinkWarnungen*** (***modell***) Gibt die Menge der Targetlink-Warnungen zurück, die bei der Codegenerierung des Modells *modell* auftreten.

***ddVariable*** (***b***) Gibt den Namen der verwendeten Targetlink-Variablen aus dem Data Dictionary zurück. Wenn keine Variable verwendet wird, ist der Name leer.

***ddSkalierung*** (***b***) Gibt den Namen der verwendeten Targetlink-Skalierung aus dem Data Dictionary zurück. Wenn keine Skalierung verwendet wird, ist der Name leer.

***ddTyp*** (***b***) Gibt den Namen des verwendeten Targetlink-Typs aus dem Data Dictionary zurück. Wenn kein expliziter Typ verwendet wird, ist der Name leer.

***typisierungsprobleme*** (***modell***) Gibt die Menge der gefundenen Typisierungsprobleme zurück.

***worstCaseExecutionTime*** (***modell***) Gibt die Worst-Case-Execution-Time des Modells auf der Zielhardware zurück.

### (3) Metriken

**Metrik 71:** Anzahl der Targetlink-Fehler

| Einheit: [**Codegenerator Fehler**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |targetlinkFehler(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Targetlink-Fehler, die bei der Codegenerierung des Modells auftreten.

**Metrik 72:** Anzahl der Targetlink-Warnungen

| Einheit: [**Codegenerator Warnungen**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |targetlinkWarnungen(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Targetlink-Warnungen, die bei der Codegenerierung des Modells auftreten.

**Metrik 73:** Anzahl der Port-Blöcke ohne Data Dictionary-Eintrag

| Einheit: [**Blöcke**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid (typ(b) = \text{‘Inport‘} \vee typ(b) = \text{‘Outport‘}) \wedge ddVariable(b) = \text{“}\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Inport- und Outport-Blöcke, die einen Datentyp verwenden, der nicht zentral im Data Dictionary hinterlegt ist.

**Metrik 74:** Anzahl der verwendeten Skalierungen ohne Data Dictionary-Eintrag

| Einheit: [**Blöcke**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |\{b \in Blöcke \mid ddSkalierung(b) = \text{“}\}|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Blöcke, die eine Skalierung verwenden, die nicht zentral im Data Dictionary hinterlegt ist.

**Metrik 75:** Anzahl der verwendeten Typen aus dem Data Dictionary

| Einheit: [**Codegenerator Datentypen**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = \left| \left( \bigcup_{b \in Blöcke} ddTyp(b) \right) \setminus \{\text{“}\} \right|$$

**Beschreibung**: Diese Metrik berechnet, wie viele Typen aus dem Data Dictionary verwendet werden.

**Metrik 76:** Anzahl der verwendeten Variablen aus dem Data Dictionary

| Einheit: [**Codegenerator Variablen**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = \left| \left( \bigcup_{b \in Blöcke} ddVariable(b) \right) \setminus \{\text{“}\} \right|$$

**Beschreibung**: Diese Metrik berechnet, wie viele Variablen aus dem Data Dictionary verwendet werden.

**Metrik 77:** Anzahl der rechenintensiven Blöcke

| Einheit: [**Blöcke**] | Faktor: Effizienz |
|---|---|

$$messwert_{\#} = |RechenintensiveBlöcke|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Blöcke in einem Simulink-Modell, die in rechenintensiven Code übersetzt werden. Die Berechnung von mathematischen Funktionen ist meist rechenintensiv, da oft ein Näherungsverfahren verwendet wird, das die Lösung in mehreren Iterationen berechnet.

**Metrik 78:** Anzahl der potentiellen Datentyp-Probleme

| Einheit: [**Typisierungsprobleme**] | Faktor: Codegenerierbarkeit |
|---|---|

$$messwert_{\#} = |typisierungsprobleme\,(modell)|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl potentieller Probleme mit der Typisierung der Daten. Typisierungsprobleme sind potentielle Probleme mit den Datentypen der Blöcke. Das umfasst beispielsweise unzulässige Typkonvertierungen oder Wertebereichsverletzungen bei arithmetischen Operationen, nicht initialisierte Variablen oder unpassende Verwendungen von Signalen an Modulschnittstellen. Zum Finden der Typisierungsprobleme werden die Eingänge und Ausgänge der Blöcke untersucht. Wird ein Problem gefunden, kann kein konkretes Beispiel gegeben werden, das zeigen könnte, für welche Signalwerte das Problem auftritt. Daher sind die gefundenen Probleme nur potentielle Probleme, da sie sich zwar in dem generierten Code widerspiegeln, aber unter Umständen nur bei einer bestimmten Kombination von Eingangssignalen auftreten. Gefunden werden können potentielle Datentyp-Probleme beispielsweise durch Prüfung auf Einhaltung der Strong Data Typing-Richtlinien[a]. Tabelle A.3 im Anhang listet die verwendeten Richtlinien auf.

[a] `http://www.model-engineers.com/fileadmin/mes/PUBLIC/MXAM/MES_MXAM-SDT-Guidelines_V1.3.pdf`

**Metrik 79:** Worst-Case-Execution-Time

| Einheit: [**Millisekunden**] | Faktor: Effizienz |
|---|---|

$$messwert_{\#} = worstCaseExecutionTime\,(modell)$$

**Beschreibung**: Diese Metrik berechnet die Worst-Case-Execution-Time des Modells auf der Zielhardware. Dazu wird ein Modell des Zielprozessors verwendet. Das Modell des Prozessors enthält beispielsweise die Caches und die Pipelines des realen Prozessors. Zusammen mit dem kompilierten C-Code, der bei der Codegenerierung des Modells entstanden ist, kann eine Aussage über den längsten möglichen Ausführungsweg und somit die maximal benötigten Taktzyklen gemacht werden. Die Anzahl der Taktzyklen wird abschließend anhand der Taktfrequenz des Prozessors in eine Zeitangabe umgerechnet. So beschreiben beispielsweise Ferdinand et al. in [Fer+01] eine solche WCET-Analyse für einen Motorola ColdFire MCF 5307 Prozessor, der typischerweise in eingebetteten Systemen zum Einsatz kommt.

## 3.13. Artefaktmetriken

In diesem Abschnitt werden die Metriken vorgestellt, die keine direkten Aussagen über das Simulink-Modell an sich treffen, sondern weitere Artefakte aus dem modellbasierten Entwicklungsprozess analysieren. Die Artefakte des modellbasierten Entwicklungsprozesses werden in Abschnitt 2.1 beschrieben.

### (1) Datenstrukturen

***Anforderungen*** Die Menge aller Anforderungen, die für ein Simulink-Modell *modell* vorgegeben sind.

***Testfälle*** Die Menge aller Testfälle, die für ein Simulink-Modell *modell* existieren.

### (2) Funktionen

***erfolgreicheTestfälle*** (***Testfälle***) $\subseteq$ ***Testfälle*** Gibt die Menge der erfolgreichen Testfälle zurück.

***testabdeckung*** (***modell***, ***Testfälle***) $\in \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ Gibt die Testabdeckung als Prozentzahl zurück, d. h. wie viel Prozent des Modells *modell* durch die Testfälle *Testfälle* getestet werden.

***anforderungsabdeckung*** (***modell***, ***Anforderungen***) $\in \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ Gibt die Anforderungsabdeckung als Prozentzahl zurück, d. h. wie viel Prozent der gewünschten Anforderungen *Anforderungen* im Modell umgesetzt werden.

### (3) Metriken

**Metrik 80:** Anzahl der Testfälle

| Einheit: [**Testfälle**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |Testfälle|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Testfälle.

**Metrik 81:** Erfolgreiche Testfälle

| Einheit: [**(erfolgreiche) Testfälle/(alle) Testfälle**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\%} = \frac{|erfolgreicheTestfälle\,(Testfälle)|}{|Testfälle|}$$

**Beschreibung**: Diese Metrik berechnet den Prozentsatz der erfolgreichen Testfälle.

**Metrik 82:** Testabdeckung

| Einheit: [**(getestete) Blöcke/(ungetestete) Blöcke**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\%} = testabdeckung\,(modell, Testfälle)$$

**Beschreibung**: Diese Metrik berechnet die Testabdeckung des Modells, d. h. wie viel Prozent des Modells durch Testfälle abgedeckt werden. Informationen über die Testabdeckung müssen während der Testausführung mitprotokolliert und anschließend abgespeichert werden.

**Metrik 83:** Anzahl der Anforderungen

| Einheit: [**Anforderungen**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\#} = |Anforderungen|$$

**Beschreibung**: Diese Metrik berechnet die Anzahl der Anforderungen.

**Metrik 84:** Anforderungsabdeckung

| Einheit: [**(umgesetzte) Anforderungen/ (alle) Anforderungen**] | Faktor: Korrektheit |
|---|---|

$$messwert_{\%} = anforderungsabdeckung\,(modell, Anforderungen)$$

**Beschreibung**: Diese Metrik berechnet die Anforderungsabdeckung eines Modells. Jede Anforderung ist mit einer ID versehen. Zur Berechnung wird die Menge aller IDs der Anforderungen ermittelt. Im Modell werden diese IDs an den Stellen annotiert, an denen sie umgesetzt werden. Dann wird ermittelt, wie viel Prozent der Anforderungen im Modell umgesetzt werden.

## 3.14. Zusammenfassung

In diesem Kapitel wurden zuerst die zentralen Begriffe für Modellmetriken definiert. Danach wurde eine Formalisierung der Simulink-Modelle vorgestellt. Die Formalisierung wurde anschließend für die Vorstellung der Modellmetriken verwendet. Die Modellmetriken wurden nach ihren Funktionalitäten in Gruppen eingeteilt und beschrieben.

Ein Teil der Metriken in diesem Kapitel wurde aus der Literatur übernommen. Der andere Teil der Metriken ist von Fragestellungen aus der Praxis motiviert. Stammt eine Metrik aus der Literatur, wurde ein Verweis bei der jeweiligen Metrik vorgenommen. Alle Metriken aus [Hol11] sind in einer im Rahmen dieser Arbeit betreuten Masterarbeit entstanden. Die Metriken, die aus der Praxis motiviert sind, dienen entweder dazu, Elemente zu zählen oder tiefere Einblicke in nicht leicht zu erfassende Sachverhalte zu erhalten. Wenn nur Elemente gezählt werden, ist das Ziel meist ein schneller Überblick über ein Simulink-Modell. Metriken, die tiefere Einblicke erlauben, bereiten hingegen Sachverhalte auf, die von Hand nur sehr aufwendig zu ermitteln sind. Ein Beispiel dafür ist die durchschnittliche Signallänge.

# 4. Qualitätsmodell

In diesem Kapitel wird das erarbeitete Qualitätsmodell für Simulink-Modelle vorgestellt. Dazu wird zuerst das zugrunde liegende Qualitätsmodell beschrieben und die Anpassungen für die modellbasierte Entwicklung erläutert. Das Qualitätsmodell besteht aus Qualitätsfaktoren, Qualitätskriterien und Metriken. Durch das Qualitätsmodell wird der Begriff der Modellqualität hierarchisch zergliedert und dadurch greifbar und überprüfbar gemacht. Dann wird das im Rahmen dieser Arbeit entstandene Qualitätsmodell im Detail vorgestellt. Dabei spiegelt sich die Struktur des Qualitätsmodells in der Struktur der Abschnitte wider. In diesem Kapitel liegt der Fokus auf der Herleitung des Qualitätsmodells und der Begründung des Aufbaus. Die enthaltenen Metriken wurden bereits in Kapitel 3 vorgestellt.

## 4.1. Ursprung und Konstruktion

In diesem Abschnitt wird die Konstruktion des Qualitätsmodells beschrieben. Die Struktur und der Inhalt des Qualitätsmodells folgen dem Ansatz von Cavano und McCall, die ihr FCM-Qualitätsmodell (FCM steht für *Factors*, *Critera* und *Metrics*) für die Bewertung von Softwarequalität in [CM78] vorstellten. Das Qualitätsmodell wurde so angepasst und erweitert, dass es auf grafische Daten- und Kontrollflussmodelle angewendet werden kann.

Cavano und McCall zerlegen den Begriff der Softwarequalität in *Faktoren* (Factors) wie z. B. Wartbarkeit oder Testbarkeit. Die Faktoren wiederum setzen sich aus *Kriterien* (Criteria) zusammen, welche die Faktoren konkretisieren. Diese Faktoren wurden in dieser Arbeit z. B. um den Faktor *Codegenerierbarkeit* erweitert, der für grafische Modelle hinzukommt. Tabelle 4.1 zeigt die Gegenüberstellung der Faktoren des Qualitätsmodells von Cavano und McCall und den Faktoren des Qualitätsmodells aus dieser Arbeit. In der ersten Spalte stehen Kategorien, in die Cavano und McCall die Faktoren eingeteilt haben. In der zweiten Spalte stehen die ursprünglichen Faktoren mit jeweils einer Frage, die Cavano und McCall zu jedem Faktor formuliert haben.

| **Cavano und McCall** | | **Qualitätsmodell in dieser Arbeit** |
|---|---|---|
| Änderung | **Wartbarkeit** (Can I fix it?) | **Wartbarkeit** |
| | **Flexibilität** (Can I change it?) | Faktor wird nicht verwendet, ist in Wartbarkeit und Verständlichkeit enthalten |
| | **Testbarkeit** (Can I test it?) | **Testbarkeit** |
| Portierung | **Portabilität** (Will I be able to use it on another machine?) | fallen weg, da durch die modellbasierte Entwicklung von der konkreten Hardware abstrahiert wird und dadurch Wiederverwendung erleichtert wird |
| | **Wiederverwendbarkeit** (Will I be able to reuse some of the software?) | |
| | **Interoperabilität** (Will I be able to interface it with another system?) | entfällt, da auf Softwareseite durch AUTOSAR und auf Hardwareseite durch standardisierte Bussysteme, wie z.B. CAN-Bus, sichergestellt |

| | | |
|---|---|---|
| Betrieb | **Korrektheit** (Does it do what I want?) | **Korrektheit** |
| | **Zuverlässigkeit** (Does it do it accurately all of the time?) | **Robustheit**, wurde umbenannt |
| | **Effizienz** (Will it run on my hardware as well as it can?) | **Effizienz** |
| | **Integrität** (Is it secure?) | fallen weg, da das Modell nicht für den Endbenutzer sichtbar ist |
| | **Benutzbarkeit** (Can I run it?) | |
| | | **Codegenerierbarkeit**, zusätzlicher Faktor wegen modellbasierter Entwicklung |
| | | **Verständlichkeit**, zusätzlicher Faktor wegen grafischer Notation |

Tabelle 4.1.: Gegenüberstellung der Qualitätsmodelle

Die Faktoren in der Kategorie *Portierung* sind für diese Arbeit nicht relevant, da bei der modellbasierten Entwicklung ein Austausch der Hardwareplattform bereits vom Konzept her vorgesehen ist. Außerdem steht die Domäne fest, in der die Modelle verwendet werden. Dadurch ist die Bandbreite an eingesetzten Technologien sehr viel geringer. So haben sich beispielsweise AUTOSAR als Komponentenmodell und der CAN-Bus als Kommunikationsbus durchgesetzt. Zusätzlich wurde beispielsweise der Faktor *Flexibilität* entfernt und in die Faktoren *Wartbarkeit* und *Verständlichkeit* aufgeteilt, um das Qualitätsmodell übersichtlich zu halten. Dies war möglich, da manche der Kriterien nicht nur einem Faktor zuzuordnen sind, sondern zu mehreren Faktoren passen.

Die Kriterien sind Indikatoren für gewünschte Eigenschaften, die von den Faktoren auf oberster Ebene gefordert werden. Die Kriterien ermöglichen das Identifizieren von Auffälligkeiten, die dazu führen, dass das Modell die gewünschten Eigenschaften nicht besitzt. Diese Auffälligkeiten sind die modellbasierten Analogien der *Code Smells* von Fowler und Beck [Fow99, S. 75 ff.]. Fowler und Beck haben eine Liste von Auffälligkeiten für Quellcode zusammengestellt, die einen Hinweis auf mögliche Probleme geben. Das umfasst beispielsweise zu lange Methoden, Methoden mit zu vielen Parametern, unvollständige Bibliotheksklassen oder eine zu starke Kopplung zwischen zwei Klassen. Mens et al. übertragen den Begriff der Code Smells in [MTM07] auf die modellbasierte Entwicklung. Sie definieren *Model Smells* wie folgt: „typical model smells have to do with redundancies, ambiguities, inconsistencies, incompleteness, non-adherence to design conventions or standards, abuse of the modelling notation, and so on“. Im Qualitätsmodell dieser Arbeit wird z. B. das Vorhandensein von Redundanzen in den Simulink-Modellen durch das Kriterium *Vermeidung von Redundanz* des Faktors Wartbarkeit abgeprüft.

Die *Metriken* (Metrics) prüfen die Erfüllung der Kriterien. Kriterien können durch eine oder mehrere Metriken gemessen werden. So wird zum Beispiel der Qualitätsfaktor Verständlichkeit durch die Kriterien Lesbarkeit, sprechende Bezeichner, kognitive Erfassbarkeit und ähnliche konkretisiert. Hierbei kann das Kriterium Lesbarkeit durch eine Metrik zur Messung der Schachtelungstiefe und der Anzahl der Rückkopplungen innerhalb eines Subsystems gemessen werden. Es ergibt sich eine Baumstruktur. Abbildung 4.1 zeigt den schematischen Aufbau des Qualitätsmodells.

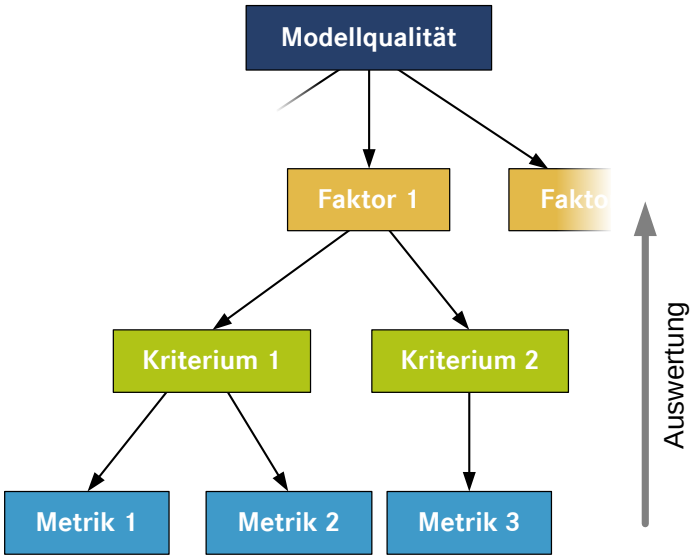



Abbildung 4.1.: Schematischer Aufbau des Qualitätsmodells

Für die einzelnen Elemente des Qualitätsmodell ergeben sich folgende Definitionen:

**Definition 6 (Faktoren).** *Faktoren geben gewünschte Eigenschaften für qualitativ hochwertige Modelle vor.*

**Definition 7 (Kriterien).** *Kriterien sind Indikatoren für die gewünschten Eigenschaften (Faktoren).*

**Definition 8 (Metriken).** *Metriken prüfen die Erfüllung der Kriterien.*

Das Qualitätsmodell ist nicht monolitisch. Es besteht aus einzelnen Qualitätsaspekten. Einzelne Qualitätsaspekte können an- oder abgeschaltet sowie neue Qualitätsaspekte hinzugefügt werden. Ein Qualitätsaspekt gruppiert zusammengehörige Metriken. Ein Qualitätsaspekt besteht aus einem Teilbaum mit Faktoren, Kriterien und Metriken. Abbildung 4.2 zeigt, wie zwei Qualitätsaspekte zu einem Qualitätsmodell vereinigt werden. Bei der Vereinigung werden Knoten mit gleichen Namen als identisch angesehen.

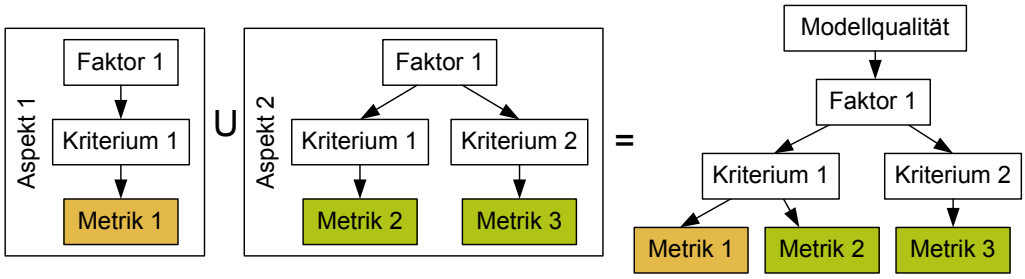



Abbildung 4.2.: Schematische Darstellung der Vereinigung zweier Aspekte

Tabelle 4.2 zeigt einen Überblick über die verwendeten Qualitätsaspekte. Abschnitt A.2 im Anhang listet die Zuordnung der Metriken zu den Qualitätsaspekten auf. Das in diesem Kapitel vorgestellte Qualitätsmodell ist vollständig, d. h. alle verfügbaren Aspekte sind ausgewählt.

| **Aspekt** | **Beschreibung der Eigenschaft** |
|---|---|
| Anforderungen | Treffen Aussagen über die Anforderungen eines Simulink-Modells. |
| Architektur | Treffen Aussagen über die Architektur eines Simulink-Modells. |
| Laufzeit | Treffen Aussagen über die Laufzeit eines Simulink-Modells. |
| Simulink | Treffen allgemeine Aussagen über ein Simulink-Modell. |
| Stateflow | Treffen Aussagen über die Stateflow-Anteile eines Simulink-Modells. |
| Targetlink | Treffen Aussagen über die Targetlink-Anteile eines Simulink-Modells. |
| Test | Treffen Aussagen über die Tests eines Simulink-Modells. |

Tabelle 4.2.: Verwendete Qualitätsaspekte für die Gruppierung der Metriken

In den folgenden Abschnitten wird das Qualitätsmodell beschrieben. Dabei spiegelt die Struktur der Abschnitte die Baumstruktur des Qualitätsmodells wider. Für alle Metriken ist am Ende ihrer Beschreibungen angegeben, welchen Einfluss ihre Messwerte auf das jeweilige Kriterium haben. Außerdem ist hinter dem Namen jeder Metrik angegeben, wie hoch ihr Einfluss auf die Modellqualität ist. Die einzelnen Metriken können einen unterschiedlich hohen Einfluss haben, da die betrachteten Eigenschaften sehr unterschiedlich sind. So ist zwar beispielsweise eine hohe Anzahl an Ground-Blöcken ein möglicher Hinweis auf noch nicht vollständig modellierte Funktionalität, allerdings sind die Auswirkungen auf die Software doch gering. Werden hingegen Zustände in Stateflow-Diagrammen nie erreicht, liegt ein ernsthaftes Problem vor. Folgende Notation wird für den Einfluss der Metriken verwendet:

(+) geringer Einfluss (++) mittlerer Einfluss (+++) hoher Einfluss

Tabelle 4.3 zeigt einen Überblick über die Faktoren, Kriterien und Metriken des Qualitätsmodells in tabellarischer Form. Die Reihenfolge der Faktoren, Kriterien und Metriken ist identisch mit der Reihenfolge der ausführlichen Beschreibung ab Abschnitt 4.2. Die Nummerierung der Metriken wurde aus Kapitel 3 übernommen.

| **Faktor** | **Kriterium** | **Metrik** |
|---|---|---|
| Codegenerierbarkeit | Explizite Datentypen | Metrik 78: Anzahl der potentiellen Datentyp-Probleme |
| | Modularisierung | Metrik 2: Anzahl der TargetlinkFunction-Blöcke |

| Faktor | Kriterium | Metrik |
|---|---|---|
| | Nebeneffektfreiheit | Metrik 3: Anzahl der CustomCode-Blöcke |
| | Umgehung bekannter Probleme | Metrik 66: Anzahl der verletzten dSPACE-Richtlinien |
| | | Metrik 67: Anzahl der verletzten MISRA TL-Richtlinien |
| | Verwendung des Data Dictionaries | Metrik 73: Anzahl der Port-Blöcke ohne Data Dictionary-Eintrag |
| | | Metrik 74: Anzahl der verwendeten Skalierungen ohne Data Dictionary-Eintrag |
| | | Metrik 75: Anzahl der verwendeten Typen aus dem Data Dictionary |
| | | Metrik 76: Anzahl der verwendeten Variablen aus dem Data Dictionary |
| | Verwendung von Targetlink | Metrik 71: Anzahl der Targetlink-Fehler |
| | | Metrik 72: Anzahl der Targetlink-Warnungen |
| Effizienz | Maximale Laufzeit | Metrik 77: Anzahl der rechenintensiven Blöcke |
| | | Metrik 79: Worst-Case-Execution-Time |
| Korrektheit | Ausreichende Tests | Metrik 80: Anzahl der Testfälle |
| | | Metrik 81: Erfolgreiche Testfälle |
| | | Metrik 82: Testabdeckung |
| | Einhaltung der Architektur | Metrik 65: Anzahl der Architekturverletzungen |
| | Funktionale Korrektheit | Metrik 83: Anzahl der Anforderungen |
| | | Metrik 84: Anforderungsabdeckung |
| | Funktionale Vollständigkeit | Metrik 16: Anzahl der DataStoreMemory-Blöcke ohne Schreibzugriff |
| | | Metrik 17: Anzahl der DataStoreMemory-Blöcke ohne Lesezugriff |
| | | Metrik 4: Anzahl der Ground-Blöcke |
| | | Metrik 5: Anzahl der Terminator-Blöcke |
| | | Metrik 52: Anzahl der verfallenen Ergebnisse |
| Robustheit | Behandlung von Unter-/Überläufen | Metrik 60: Anzahl der Wertebereichsverletzungen |
| | | Metrik 6: Anzahl der Saturation-Blöcke |
| | | Metrik 18: Anzahl der aktivierten Targetlink Block-Saturierungen |
| | Konsistente Zustandsautomaten | Metrik 70: Anzahl der unerreichbaren Zustände |
| Testbarkeit | Aufteilung | Metrik 58: Durchschnittliche Busgröße |
| | | Metrik 50: Durchschnittliche Anzahl der unabhängigen Berechnungspfade |
| | | Metrik 56: Durchschnittliche Breite der Eingangsschnittstelle |
| | | Metrik 38: Anzahl der testbaren Subsysteme mit impliziten Inport-Datentypen |
| | | Metrik 64: Durchschnittliche lokale Komplexität |

| Faktor | Kriterium | Metrik |
|---|---|---|
| | | Metrik 39: Anzahl der testbaren Subsysteme mit eingehenden Bussen |
| | Vermeidung impliziter Konstrukte | Metrik 31: Anzahl der konfigurierbaren Subsysteme |
| | | Metrik 7: Anzahl der DataStoreMemory-Blöcke |
| | | Metrik 20: Anzahl der globalen Goto-Blöcke |
| Verständ-lichkeit | Dokumentation | Metrik 10: Anzahl der ModelInfo-Blöcke |
| | | Metrik 43: Anzahl der Subsystem-Annotationen |
| | | Metrik 44: Durchschnittliche Länge der Subsystem-Annotationen |
| | | Metrik 28: Anzahl der Linien-Annotationen |
| | Kognitive Erfassbarkeit | Metrik 11: Anzahl der BusCreator-Blöcke |
| | | Metrik 12: Anzahl der BusSelector-Blöcke |
| | | Metrik 51: Anzahl der Zyklen |
| | | Metrik 8: Anzahl der DataStoreRead-Blöcke |
| | | Metrik 9: Anzahl der DataStoreWrite-Blöcke |
| | | Metrik 13: Anzahl der Goto-Blöcke |
| | | Metrik 14: Anzahl der From-Blöcke |
| | | Metrik 21: Durchschnittliche Anzahl an nachfolgenden Blöcken |
| | | Metrik 15: Anzahl der UnitDelay-Blöcke |
| | | Metrik 19: Anzahl der UnitDelay-Blöcke in Vorwärtsrichtung |
| | | Metrik 59: Anzahl der internen Zustände |
| | Lesbarkeit | Metrik 29: Anzahl der überkreuzenden Linien |
| | | Metrik 41: Durchschnittliche Höhe der Subsysteme auf dem Bildschirm |
| | | Metrik 42: Durchschnittliche Breite der Subsysteme auf dem Bildschirm |
| | Umfang | Metrik 32: Anzahl der atomaren Subsysteme |
| | | Metrik 1: Anzahl der Blöcke |
| | | Metrik 40: Tiefe der Subsystem-Hierarchie |
| | | Metrik 27: Anzahl der Linien |
| | | Metrik 68: Anzahl der Stateflow-Blöcke |
| | | Metrik 30: Anzahl der Subsysteme |
| | | Metrik 69: Durchschnittliche Anzahl an Stateflow-Zuständen |
| | | Metrik 33: Durchschnittliche Anzahl an Subsystemen pro Subsystem |
| | | Metrik 23: Anzahl der verwendeten Blocktypen |
| | Verwendung sprechender Bezeichner | Metrik 24: Anzahl der magischen Konstanten |
| | | Metrik 25: Anzahl der Werte magischer Konstanten |

| Faktor | Kriterium | Metrik |
|---|---|---|
| | | Metrik 26: Anzahl der Konstanten mit Namen |
| Wartbarkeit | Architektur | Metrik 34: Anzahl der arithmetischen Subsysteme |
| | | Metrik 35: Anzahl der logischen Subsysteme |
| | | Metrik 36: Anzahl der signalführenden Subsysteme |
| | | Metrik 37: Anzahl der schaltenden Subsysteme |
| | Komplexität | Metrik 63: Höhe der globalen Komplexität |
| | | Metrik 22: Durchschnittliche Blockinstabilität |
| | | Metrik 53: Durchschnittlicher Vernetzungsgrad |
| | | Metrik 57: Durchschnittliche Breite der Ausgangsschnittstelle |
| | | Metrik 54: Durchschnittliche Signalpfadlänge |
| | | Metrik 55: Durchschnittliche Länge der Signalpfade auf dem Bildschirm |
| | Standardkonformität | Metrik 62: Anzahl der verwendeten Modellierungsmuster |
| | Vermeidung von Redundanz | Metrik 61: Anzahl der Modellklone |
| | | Metrik 46: Anzahl der deaktivierten Bibliotheksverknüpfungen |
| | | Metrik 47: Anzahl der deaktivierten und unauffindbaren Bibliotheksverknüpfungen |
| | | Metrik 48: Anzahl der deaktivierten Bibliotheksverknüpfungen mit veränderter Struktur |
| | | Metrik 45: Anzahl der verwendeten Bibliotheksverknüpfungen |
| | | Metrik 49: Anzahl der verwendeten Bibliotheken |

Tabelle 4.3.: Überblick über das Qualitätsmodell

## 4.2. Codegenerierbarkeit

Dieser Faktor ist gegenüber dem ursprünglichen Qualitätsmodell von Cavano und McCall neu, da das Konzept des generierten Codes erst mit der modellbasierten Softwareentwicklung hinzugekommen ist. In der handcodierten Softwareentwicklung gab es diesen Schritt nicht. Der Faktor fasst Kriterien zusammen, die erfüllt werden müssen, damit reibungslos Code aus den Modellen erzeugt werden kann. In den folgenden Abschnitten werden die Kriterien vorgestellt, die zur Codegenerierbarkeit von Simulink-Modellen beitragen.

### 4.2.1. Explizite Datentypen

Dieses Kriterium fordert eine explizite Angabe der zu verwendenden Datentypen. Das bedeutet, dass die Einstellungen für die Codegenerierung, wie zum Beispiel Datentypen, Skalierungen und Wertebereiche der Blöcke explizit vorgegeben werden. Denn die Einstellungen der jeweiligen Blöcke müssen zueinander passen. Werden beispielsweise zwei Signale mit einem Wertebereich von $[2, 8]$ in einem Add-Block addiert, muss der Wertebereich des Ausgangs des Add-Blocks mindestens $[4, 16]$ betragen, ansonsten kommt es zu einem Unter- oder Überlauf. Derartige Probleme können durch explizite Angabe und Plausibilitätsprüfung der Datentypen umgangen werden. Diese Prüfung ist eine statische Prüfung und findet nicht zur Laufzeit des Modells statt.

**Anzahl der potentiellen Datentyp-Probleme** (++)
Je mehr potentielle Datentyp-Probleme gefunden werden, desto wahrscheinlicher ist es, dass es Inkonsistenzen bei den eingestellten Datentypen und deren Eigenschaften gibt. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

### 4.2.2. Modularisierung

Dieses Kriterium fordert eine Modularisierung der Simulink-Modelle. Modularisierung im Sinne der Codegenerierbarkeit bezieht sich auf die Aufteilung des C-Codes in den generierten Dateien. Wird beispielsweise nur eine große C-Datei generiert, ist es sehr schwierig, einen Überblick über den generierten Code zu erhalten.

**Anzahl der TargetlinkFunction-Blöcke** (+)
Die Anzahl der verwendeten TargetlinkFunction-Blöcke lässt keinen Rückschluss zu, ob richtig modularisiert wird. Allerdings wird pro TargetlinkFunction-Block eine eigene C-Datei generiert und dadurch wird es erleichtert, sich einen Überblick über den generierten Code zu verschaffen. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

### 4.2.3. Nebeneffektfreiheit

Dieses Kriterium fordert Nebeneffektfreiheit in Bezug auf den generierten Code. Das betrifft beispielsweise die Einbettung von altem bestehendem C-Code in die Simulink-Modelle. Dieser C-Code kann z. B. beliebige Betriebssystemaufrufe enthalten und Daten an jeder Stelle im Speicher verändern, ohne dass diese Veränderungen durch Simulink-Blöcke und Linien modelliert sind. Das kann zu sehr schwer nachvollziehbaren Fehlern in dem generierten Code führen.

**Anzahl der CustomCode-Blöcke** (+++)
Enthält ein Targetlink-Modell CustomCode-Blöcke, kann in diesen Blöcken beliebiger C-Code ausgeführt werden, der beliebige Seiteneffekte haben kann. Daher sollten möglichst keine CustomCode-Blöcke verwendet werden. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik den Wert 0 annimmt.

### 4.2.4. Umgehung bekannter Probleme

Dieses Kriterium fordert, dass bekannte Probleme in Bezug auf die Codegenerierung bei der Modellierung umgangen werden. Bekannte Probleme entstehen durch die Integration von Targetlink in Simulink. Es ist z. B. möglich, dass in einer neuen Version von Simulink ein neues Feature eingeführt wird, das von Targetlink noch nicht unterstützt wird. Durch die Vermeidung dieses neuen Features kann dafür gesorgt werden, dass bei der Codegenerierung keine Probleme auftreten.

**Anzahl der verletzten dSPACE-Richtlinien** (++) und
**Anzahl der verletzten MISRA TL-Richtlinien** (++)
Je größer die Anzahl der Richtlinienverletzungen ist, desto wahrscheinlicher ist es, dass schwerwiegende Probleme bei der Codegenerierung auftreten. Somit erfüllen die Metriken ihr Kriterium umso besser, je kleiner die Messwerte der Metriken sind.

### 4.2.5. Verwendung des Data Dictionaries

Dieses Kriterium fordert die korrekte Verwendung des Data Dictionaries, welches eine zentrale Rolle bei der Codegenerierung spielt. Wie in Abschnitt 2.2.3 beschrieben, erleichtert die Verwendung des Data Dictionaries Änderungen an den Datentypen, die zur Codegenerierung benötigt werden.

**Anzahl der Port-Blöcke ohne Data Dictionary-Eintrag** (++) und
**Anzahl der verwendeten Skalierungen ohne Data Dictionary-Eintrag** (++)
Je mehr Datentypen und Skalierungen nicht zentral im Data Dictionary definiert sind, desto aufwendiger sind Änderungen, da jeder Block einzeln angepasst werden muss. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metrik den Wert 0 annehmen.

**Anzahl der verwendeten Typen aus dem Data Dictionary** (+) und
**Anzahl der verwendeten Variablen aus dem Data Dictionary** (+)
Sowohl sehr wenige als auch sehr viele Einträge sind fragwürdig: Bei sehr wenigen Einträgen stellt sich die Frage, ob die Einstellungen direkt in den Blöcken vorgenommen wurden. Bei sehr vielen Einträgen hingegen stellt sich die Frage, ob so viele verschiedene Datentypen und Variablen notwendig sind. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken in einem zulässigen Bereich liegen.

### 4.2.6. Verwendung von Targetlink

Dieses Kriterium fordert Targetlink möglichst so zu verwenden, wie es vorgesehen ist. Bei der Codegenerierung werden Warnungen und Fehler ausgegeben. Fehler müssen auf jeden Fall beseitigt werden und Warnungen sollten ebenfalls vermieden werden, da sie potentielle Fehler darstellen.

**Anzahl der Targetlink-Fehler** (+++) und
**Anzahl der Targetlink-Warnungen** (++)
Targetlink-Fehler deuten auf Probleme hin. Targetlink-Warnungen hingegen sind nur potentielle Fehler. Viele Warnungen beziehen sich auf Probleme mit erlaubten Maximal- und Minimalwerten von Signalen. Diese Warnungen können unter anderem zu mehr Auffälligkeiten bei einer statischen Analyse des generierten C-Codes führen, da der Codegenerator im Falle von definierten Grenzen Kommentare in den generierten Code schreiben kann, die bei

der statischen Analyse als zusätzliche Bedingungen verwendet werden können. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken klein bzw. im besten Fall sogar 0 sind.

## 4.3. Effizienz

Dieser Faktor fasst Kriterien zusammen, die erfüllt werden müssen, damit ein Simulink-Modell auf der gewünschten Zielhardware ausführbar ist. In den folgenden Abschnitten werden die Kriterien vorgestellt, die zur Effizienz von Simulink-Modellen beitragen.

### 4.3.1. Maximale Laufzeit

Dieses Kriterium fordert, dass die Modelle bei der Ausführung auf der Zielhardware eine vorgegebene maximale Laufzeit nicht überschreiten. Die Einhaltung einer maximalen Laufzeit ist für Fahrzeugsoftware von zentraler Bedeutung, da sie meist Echtzeitanforderungen einhalten muss (vergleiche 2.2.4).

**Anzahl der rechenintensiven Blöcke** (++)
Enthält ein Modell rechenintensive Blöcke, dann erhöht sich die Wahrscheinlichkeit, dass die Ausführung des Modells auf der Zielhardware zu lange dauert. Es bietet sich häufig an, anstatt der aufwendigen Berechnungen Lookup-Tabellen zu verwenden. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Worst-Case-Execution-Time** (+++)
In einem Steuergerät bekommen die Modelle für jeden Aufrufzyklus eine maximal erlaubte Laufzeit vorgeschrieben. Sobald die Worst-Case-Execution-Time eines Modells größer als die erlaubte Laufzeit wird, können z. B. alle weiteren Aufrufe aus dem Takt geraten. Dadurch ist das Steuergerät nicht mehr echtzeitfähig und muss Notfallmaßnahmen, wie z. B. einen Neustart, einleiten. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik einen Wert $WCET < schedulerZeitscheibe$ annimmt. Dabei bezeichnet *schedulerZeitscheibe* die zur Verfügung stehende Zeit, die das Modell auf dem jeweiligen Steuergerät zugeteilt bekommt.

## 4.4. Korrektheit

Dieser Faktor fasst Kriterien zusammen, die erfüllt werden müssen, damit in einem Simulink-Modell die gewünschte Funktionalität umgesetzt wird. In den folgenden Abschnitten werden die Kriterien vorgestellt, die zur Korrektheit von Simulink-Modellen beitragen.

### 4.4.1. Ausreichende Tests

Dieses Kriterium fordert, dass die Modelle ausreichend getestet werden. Ausreichend bedeutet in diesem Fall in erster Linie, dass alle Anforderungen einen entsprechenden Testfall besitzen, der die Anforderung abprüft.

**Anzahl der Testfälle** (+)
Die Anzahl der Testfälle muss in einer Relation zu der Größe und der Komplexität des Simulink-Modells stehen. Ein großes Modell benötigt viele Testfälle und ein kleineres Modell kommt mit weniger Testfällen aus. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Erfolgreiche Testfälle** (+++)
Der Prozentsatz der erfolgreichen Testfälle lässt einen direkten Rückschluss auf die umgesetzte Funktionalität des Modells zu. Im Idealfall sind alle Testfälle erfolgreich. Schlagen Testfälle fehl, gibt es dafür meist zwei Gründe. Entweder ist im Simulink-Modell ein Fehler enthalten, der beseitigt werden muss, oder es sind Fehler in den Testfällen enthalten. In beiden Fällen muss das Problem beseitigt werden. Da es sich bei dem Messwert der Metrik bereits um einen Prozentsatz handelt, trägt der Messwert der Metrik direkt zu der Erfüllung ihres Kriteriums bei.

**Testabdeckung** (+++)
Der Prozentsatz der Testabdeckung gibt an, wie viel Prozent des Modells durch Testfälle geprüft werden. Gegen Ende des Entwicklungsprozesses sollte dieser Wert gegen 100% gehen. Da es sich bei dem Messwert der Metrik bereits um einen Prozentsatz handelt, trägt der Messwert der Metrik direkt zu der Erfüllung ihres Kriteriums bei.

### 4.4.2. Einhaltung der Architektur

Dieses Kriterium fordert, dass in den Modellen eine vorgegebene Architektur eingehalten wird. Vor der Erstellung des Simulink-Modells wird eine Architektur erstellt, die den Aufbau des Simulink-Modells auf oberster Ebene vorgibt. In der Architektur wird festgelegt, in welche Module sich die gewünschte Funktionalität aufteilen lässt und welche Module miteinander kommunizieren dürfen. Die Einhaltung der Architektur trägt dazu bei, die gewünschte Funktionalität zu erhalten.

**Anzahl der Architekturverletzungen** (++)
Je mehr Architekturverletzungen gefunden werden, desto mehr weicht die konkrete Umsetzung des Simulink-Modells von der ursprünglichen Vorgabe ab. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

### 4.4.3. Funktionale Korrektheit

Dieses Kriterium fordert, dass die gewünschten Anforderungen in dem Simulink-Modell umgesetzt sind. Nur wenn alle Anforderungen umgesetzt sind, kann das Simulink-Modell die gewünschte Funktionalität erfüllen.

**Anzahl der Anforderungen** (+)
Es ist auffällig, wenn besonders viele oder besonders wenig Anforderungen umgesetzt werden sollen. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Anforderungsabdeckung** (+++)
Am Ende der Entwicklung muss jede Anforderung im Modell umgesetzt werden. Allerdings ist es vor allem bei Anforderungen auf einem hohen Abstraktionslevel sehr schwer, eine direkte

Abbildung auf das Modell zu erreichen. Daher ist nur eine Aussage über die Anforderungen auf niedrigstem Abstraktionslevel sinnvoll. Außerdem kann eine unvollständige Anforderungsabdeckung in den frühen Entwicklungsphasen unproblematisch sein, da noch nicht alle Anforderungen umgesetzt wurden. Da es sich bei dem Messwert der Metrik bereits um einen Prozentsatz handelt, trägt der Messwert der Metrik direkt zu der Erfüllung ihres Kriteriums bei.

### 4.4.4. Funktionale Vollständigkeit

Dieses Kriterium fordert, dass die gewünschte Funktionalität vollständig in dem Simulink-Modell umgesetzt ist. Das umfasst beispielsweise, dass keine Platzhalter mehr in den Modellen enthalten sind und alle in den Speicher geschriebenen Werte auch wieder gelesen werden.

**Anzahl der DataStoreMemory-Blöcke ohne Schreibzugriff** (++) und
**Anzahl der DataStoreMemory-Blöcke ohne Lesezugriff** (++)
Nicht gelesene oder geschriebene DataStoreMemory-Blöcke stellen Fehler dar und müssen behoben werden. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken klein bzw. im besten Fall sogar 0 sind.

**Anzahl der Ground-Blöcke** (+) und
**Anzahl der Terminator-Blöcke** (+)
Ground und Terminator-Blöcke können dazu verwendet werden, unverbundene Inports bzw. Outports zu verbinden und so eine Warnung von Simulink zu vermeiden. Daher sind Ground- und Terminator-Blöcke ein möglicher Hinweis auf noch nicht modellierte Funktionalität. Allerdings gibt es auch Ausnahmen. So kann im Falle eines Bit-Decoders, also einem Block, der ein Byte in seine Bits zerlegt, nur ein Ausgangsbit von Interesse sein. In diesem Fall würden die anderen 7 Bits terminiert werden, ohne dass eine noch nicht modellierte Funktionalität vorliegt. Somit erfüllen die Metriken ihr Kriterium umso besser, je kleiner die Messwerte der Metriken sind.

**Anzahl der verfallenen Ergebnisse** (++)
Durch das Zählen der verfallenen Ergebnisse kann ein Rückschluss daraus gezogen werden, wie viele Berechnungen in einem Modell unnötigerweise durchgeführt werden. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

## 4.5. Robustheit

Dieser Faktor wurde gegenüber dem Qualitätsmodell von Cavano und McCall umbenannt. Er hieß ursprünglich Zuverlässigkeit. Der Faktor fasst Kriterien zusammen, die erfüllt sein müssen, damit ein Simulink-Modell zur Laufzeit das gewünschte Verhalten besitzt. In den folgenden Abschnitten werden die Kriterien vorgestellt, die zur Robustheit von Simulink-Modellen beitragen.

### 4.5.1. Behandlung von Unter-/Überläufen

Dieses Kriterium fordert, dass Unter- und Überlaufe explizit behandelt werden. Werden Unter- und/oder Überläufe in einem Simulink-Modell nicht explizit behandelt, dann drohen schwer zu findende Laufzeitfehler, da diese nur in Abhängigkeit von bestimmten Signalwerten auftreten.

**Anzahl der Wertebereichsverletzungen** (+++)
Gefundene Wertebereichsverletzungen sind Fehler, die behoben werden müssen. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik den Wert 0 annimmt.

**Anzahl der Saturation-Blöcke** (+) und
**Anzahl der aktivierten Targetlink Block-Saturierungen** (+)
Die Verwendung von expliziten Saturation-Blöcken oder der Saturierung in den Targetlink-Blöcken verhindert zur Laufzeit Unter- oder Überläufe. Dadurch wird vermieden, dass es zu einer groben Fehlfunktion kommt. Somit erfüllen die Metriken ihr Kriterium umso besser, je größer die Messwerte der Metriken sind.

### 4.5.2. Konsistente Zustandsautomaten

Dieses Kriterium fordert, dass die Zustandsautomaten in einem Simulink-Modell konsistent sind, das umfasst beispielsweise, dass alle Zustände erreichbar sind. Gibt es einige Zustände, die nie erreichbar sind, liegt ein Fehler vor und das Simulink-Modell wird sich zur Laufzeit nicht wie gewünscht verhalten.

**Anzahl der unerreichbaren Zustände** (+++)
Gefundene, nicht erreichbare Zustände sind Fehler, die behoben werden müssen. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik den Wert 0 annimmt.

## 4.6. Testbarkeit

Dieser Faktor fasst Kriterien zusammen, die erfüllt sein müssen, damit ein Simulink-Modell im Ganzen und in Teilen testbar ist. In den folgenden Abschnitten werden die Kriterien vorgestellt, die zur Testbarkeit von Simulink-Modellen beitragen.

### 4.6.1. Aufteilung

Dieses Kriterium fordert, dass die Modelle so aufgeteilt werden, dass ihre einzelnen Teile gut testbar sind. Denn nur, wenn sich die Funktionalität eines Simulink-Modells in einzelnen, überschaubaren Modulen testen lässt, kann das gesamte Modell einfach und frühzeitig getestet werden.

**Durchschnittliche Busgröße** (+++)
Je mehr Signale in einem Bus enthalten sind, desto breiter und unübersichtlicher werden die Schnittstellen. Außerdem müssen große Busse komplett nachgebildet werden, wenn nur Teile eines Modells getestet werden sollen. Werden hingegen sehr viele sehr kleine Busse verwendet, müssen beim Test unnötig viele Busse erstellt werden, deren Signale auch einzeln geführt werden könnten. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Durchschnittliche Anzahl der unabhängigen Berechnungspfade** (+++)
Mithilfe dieser Metrik wird die Kohäsion der Subsysteme ermittelt. Haben die Subsysteme eine hohe Kohäsion, das heißt wenige unabhängige Berechnungspfade (vergleiche Abschnitt 3.2.2), deutet das darauf hin, dass pro Subsystem genau eine Funktion umgesetzt ist. Viele unabhängige Berechnungspfade deuten hingegen darauf hin, dass die Subsysteme eine schwache Kohäsion haben und daher pro Subsystem viele unterschiedliche Funktionen realisiert sind,

die möglicherweise nichts miteinander zu tun haben. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

**Durchschnittliche Breite der Eingangsschnittstelle** (++)
Je breiter die Eingangsschnittstelle der Subsysteme ist, desto aufwendiger ist der Test einzelner Modellteile, da alle Signale stimuliert werden müssen. Allerdings kann eine bestimmte Mindestbreite auch nicht unterschritten werden, da jedes Subsystem ein Minimum an benötigten Informationen besitzt, die notwendig sind, damit die gewünschte Funktionalität umgesetzt werden kann. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Anzahl der testbaren Subsysteme mit impliziten Inport-Datentypen** (+++)
Testbare Subsysteme mit explizit gesetzten Datentypen in den Inport-Blöcken können unabhängig vom Rest des Simulink-Modells getestet werden. Sind die Datentypen hingegen nicht angegeben, wird beim Testen von Simulink fälschlicherweise der Standard-Datentyp *double* für die eingehenden Signale angenommen. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Durchschnittliche lokale Komplexität** (++)
Ist die durchschnittliche Komplexität pro Subsystem zu groß, deutet das darauf hin, dass pro Subsystem zu viel Funktionalität umgesetzt wird. Dadurch wird das Testen erschwert. Auf der anderen Seite sollte die durchschnittliche Komplexität aber auch nicht zu niedrig sein, da sehr viele Subsysteme mit nur sehr wenigen Blöcken schwer zu überschauen sind. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Anzahl der testbaren Subsysteme mit eingehenden Bussen** (+++)
Testbare Subsysteme mit eingehenden Bussen sind schwieriger zu testen, da für den Test die Struktur des Busses nachgebaut werden muss. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

### 4.6.2. Vermeidung impliziter Konstrukte

Dieses Kriterium fordert, dass möglichst wenige implizite Konstrukte verwendet werden. Implizite Konstrukte sind Konstrukte, die nicht auf den ersten Blick erkennbar sind. So haben z. B. ein Goto- und From-Paar die gleiche Semantik wie eine Linie, sind aber visuell nicht verbunden. Die Verwendung von impliziten Konstrukten wirkt der Modularisierung entgegen und hat somit einen negativen Einfluss auf die Testbarkeit. Denn ohne eine ausreichende Modularisierung muss immer das gesamte Modell getestet werden, was sehr umfangreiche und komplexe Testfälle zur Folge hat.

**Anzahl der konfigurierbaren Subsysteme** (++)
Konfigurierbare Subsysteme erlauben es, den Inhalt eines Subsystems durch ein anderes Subsystem mit gleicher Eingangs- und Ausgangsschnittstelle auszutauschen. Dadurch wird die Testbarkeit verschlechtert, da Testfälle für alle Kombinationen der zur Verfügung stehenden Subsysteme erstellt und ausgeführt werden müssen. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Anzahl der DataStoreMemory-Blöcke** (++)
DataStoreMemory-Blöcke verschlechtern die Testbarkeit eines Modells, da sie Abhängigkeiten über Subsystemgrenzen hinweg erzeugen. Dadurch haben Subsysteme, die eigentlich einzeln getestet werden könnten, Abhängigkeiten zueinander und der Test einzelner Subsysteme wird erschwert. Außerdem ist der Signalfluss zwischen den DataStoreMemory-Blöcken und ihren

DataStoreWrite- bzw. DataStoreRead-Blöcken nicht explizit über die Schnittstelle sichtbar. Allerdings ist in manchen Fällen eine bestimmte Anzahl an DataStoreMemory-Blöcken unerlässlich, da sie z. B. für den Zugriff auf den EEPROM-Speicher des Steuergeräts verwendet werden. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Anzahl der globalen Goto-Blöcke** (++)
Goto-Blöcke mit einer globalen Sichtbarkeit verschlechtern die Testbarkeit eines Modells, da sie Abhängigkeiten über Subsystemgrenzen hinweg erzeugen. Dadurch haben Subsysteme, die eigentlich einzeln getestet werden könnten, Abhängigkeiten zueinander und der Test einzelner Subsysteme wird erschwert. Außerdem ist der Signalfluss, der durch die Goto-Blöcke modelliert wird, nicht explizit über die Schnittstelle sichtbar. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

## 4.7. Verständlichkeit

Dieser Faktor ist gegenüber dem ursprünglichen Qualitätsmodell von Cavano und McCall hinzugekommen, da die Verständlichkeit bei einer grafischen Notation eine zentrale Rolle spielt. Der Faktor fasst Kriterien zusammen, die erfüllt werden müssen, damit ein Simulink-Modell eine hohe Verständlichkeit hat. In den folgenden Abschnitten werden die Kriterien vorgestellt, die zur Verständlichkeit von Simulink-Modellen beitragen.

### 4.7.1. Dokumentation

Dieses Kriterium fordert, dass die Simulink-Modelle ausreichend dokumentiert werden. Die Dokumentation von Modellen trägt einen großen Teil zur Verständlichkeit bei, da in vielen Fällen die Funktion ganzer Subsysteme in wenigen Sätzen beschrieben werden kann. Dadurch wird die benötigte Zeit zum Verständnis eines Modells drastisch reduziert.

**Anzahl der ModelInfo-Blöcke** (++)
ModelInfo-Blöcke stellen Versionsinformationen über ein gesamtes Modell oder ein Subsystem bereit. Sie enthalten Informationen wie den Autor der letzten Änderung, das Erstellungsdatum und eine Beschreibung der Funktionalität des Modells oder Subsystems. Je mehr ModelInfo-Blöcke in einem Modell enthalten sind, desto besser dokumentiert ist ein Modell. Somit erfüllt die Metrik ihr Kriterium umso besser, je größer der Messwert der Metrik ist.

**Anzahl der Subsystem-Annotationen** (++)
Je mehr Kommentare in den Subsystemen eines Simulink-Modells enthalten sind, desto besser dokumentiert ist das Modell. Somit erfüllt die Metrik ihr Kriterium umso besser, je größer der Messwert der Metrik ist.

**Durchschnittliche Länge der Subsystem-Annotationen** (++)
Mithilfe dieser Metrik wird gemessen, wie ausführlich die Dokumentation eines Modells ist. Somit erfüllt die Metrik ihr Kriterium umso besser, je größer der Messwert der Metrik ist.

**Anzahl der Linien-Annotationen** (++)
Durch die Benennung der Linien in einem Simulink-Modell wird die Semantik der Signale explizit dokumentiert. Daher wird das Modell durch mehr Linien-Annotationen besser dokumentiert und dadurch verständlicher. Somit erfüllt die Metrik ihr Kriterium umso besser, je größer der Messwert der Metrik ist.

### 4.7.2. Kognitive Erfassbarkeit

Dieses Kriterium fordert, dass die Simulink-Modelle so aufgebaut sind, dass deren Funktionalität leicht von den Modellierern nachvollzogen werden kann. Das bedeutet, dass Simulink im Allgemeinen und bestimmte Konstrukte im Speziellen so verwendet werden, dass auf den ersten Blick ersichtlich ist, welche Funktionalität in dem Modell umgesetzt wird.

**Anzahl der BusCreator-Blöcke** (++) und
**Anzahl der BusSelector-Blöcke** (++)
BusCreator-Blöcke werden zum Erstellen von Bussen und BusSelector-Blöcke zum Entnehmen von Signalen aus Bussen verwendet. Auf der einen Seite verbessert die Verwendung von Bussen in einem Modell die Verständlichkeit. Auf der anderen Seite wird die Verständlichkeit durch zu breite Busse und durch zu viele BusSelector-Blöcke verschlechtert. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken in einem zulässigen Bereich liegen.

**Anzahl der Zyklen** (+++)
Zyklen sind Rückkopplungen, bei denen die Werte eines vorherigen Schritts in den Berechnungen des aktuellen Schritts eingehen. Zyklen verschlechtern die Verständlichkeit eines Modells, da beim Nachvollziehen der Funktionalität nicht nur die aktuellen Eingaben des Modells, sondern auch die der vorherigen Schritte berücksichtigt werden müssen. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

**Anzahl der DataStoreRead-Blöcke** (++) und
**Anzahl der DataStoreWrite-Blöcke** (++)
DataStoreRead-Blöcke lesen Werte aus DataStoreMemory-Blöcken und DataStoreWrite-Blöcke schreiben Werte in DataStoreMemory-Blöcke. Da bei der Verwendung von DataStoreMemory-Blöcken der Signalfluss nicht explizit modelliert wird und somit nicht grafisch nachvollziehbar ist, ist ein Modell mit vielen DataStoreRead- und DataStoreWrite-Blöcken schwerer zu verstehen. Wenn aus technischen Gründen auf DataStoreRead- und DataStoreWrite-Blöcke nicht verzichtet werden kann, sollte jeder DataStoreMemory-Block nur einmal geschrieben und gelesen werden. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metrik mindestens den Wert »Anzahl der DataStoreMemory-Blöcke« annehmen.

**Anzahl der Goto-Blöcke** (++) und
**Anzahl der From-Blöcke** (++)
Zwischen zwei zusammengehörigen Goto- und From-Blöcken befindet sich eine virtuelle Linie. Sie ist virtuell, da sie nicht dargestellt wird. Dadurch, dass sie nicht dargestellt wird, ist ein Simulink-Modell mit vielen From- und Goto-Blöcken schwerer nachvollziehbar. Somit erfüllen die Metriken ihr Kriterium umso besser, je kleiner die Messwerte der Metriken sind.

**Durchschnittliche Anzahl an nachfolgenden Blöcken** (+++)
Wenn ein Simulink-Modell im Durchschnitt sehr viele nachfolgende Blöcke hat, verzweigen sich seine Linien stark. Dadurch müssen beim Nachvollziehen der Funktionalität des Modells viele Pfade gleichzeitig berücksichtigt werden. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

**Anzahl der UnitDelay-Blöcke** (++) und
**Anzahl der UnitDelay-Blöcke in Vorwärtsrichtung** (+++)
UnitDelay-Blöcke speichern einen Signalwert aus dem vorherigen Zyklus. Sie erschweren das Verständnis des Modells, da nicht nur die aktuellen Werte der Eingangssignale berücksichtigt werden müssen, sondern auch die des vorherigen Abtastzyklus. Eine Möglichkeit, die Verständlichkeit zu erhöhen, ist eine grafische Konvention: UnitDelay-Blöcke werden immer in

Rückwärtsrichtung modelliert. Dadurch kann leichter erkannt werden, dass ein Wert aus einem vorherigen Zyklus verwendet wird. Wird diese Konvention nicht eingehalten, verschlechtert sich die Verständlichkeit des Modells. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken klein bzw. im besten Fall sogar 0 sind.

**Anzahl der internen Zustände** (++)
Ein UnitDelay-Block hat beispielsweise einen internen Zustand. Er speichert den Signalwert des vorherigen Zyklus. Auch kommen interne Zustände hinzu, wenn man Integrator-Blöcke oder einen Puffer zur Berechnung eines gleitenden Mittelwerts verwendet. Eine weitere Möglichkeit, wie zu viele interne Zustände in einem Modell zustande kommen, ist der Verzicht auf Stateflow. Wird auf die explizite Modellierung der Zustände eines Modells verzichtet, müssen die Zustandsinformationen anderweitig gespeichert werden. Das kann beispielsweise in FlipFlop-Blöcken geschehen, die jeweils einen internen Zustand besitzen. Viele interne Zustände in einem Modell machen ein Modell schwer verständlich, da nicht nur die aktuellen Werte der Eingangssignale berücksichtigt werden müssen, sondern auch die Werte der internen Zustände. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

### 4.7.3. Lesbarkeit

Dieses Kriterium fordert, dass Simulink-Modelle so modelliert werden, dass sie möglichst gut lesbar sind. Dies adressiert die Eigenschaften einer grafischen Notation, die bei einer textuellen Notation keine Rolle spielen, also beispielsweise die Anzahl der überkreuzenden Linien oder die Größe der Diagramme auf dem Bildschirm.

**Anzahl der überkreuzenden Linien** (++)
Linien, die sich überkreuzen, sind optisch schwer zu verfolgen. Daher haben viele sich überkreuzende Linien einen negativen Einfluss auf die Lesbarkeit eines Simulink-Modells. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

**Durchschnittliche Höhe der Subsysteme auf dem Bildschirm** (++) und
**Durchschnittliche Breite der Subsysteme auf dem Bildschirm** (++)
Um eine optimale Lesbarkeit zu gewährleisten, sollten Subsysteme weder zu wenig noch zu viele Blöcke enthalten. Wird ein Subsystem zu groß, kann es nicht mehr ohne die Wahl eines kleinen Zoomfaktors auf dem Bildschirm dargestellt werden und ist somit schlechter lesbar. Viele zu kleine Subsysteme hingegen erschweren ebenfalls den Überblick über das Gesamtmodell. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken in einem zulässigen Bereich liegen.

### 4.7.4. Umfang

Dieses Kriterium fordert, dass der Umfang der Simulink-Modelle sich in einem Bereich bewegt, in dem eine gute Verständlichkeit gegeben ist. Denn sobald der Umfang eines Modells zu groß wird oder sehr ungleich verteilt ist, nimmt die Verständlichkeit eines Modells ab. Ungleich verteilt ist der Umfang eines Modells beispielsweise, wenn in wenigen Subsystemen sehr viele Blöcke enthalten sind und in den anderen sehr wenige.

**Anzahl der atomaren Subsysteme** (+)
Atomare Subsysteme werden als atomare Einheit ausgeführt. Das heißt, die Werte aller Eingangssignale müssen berechnet sein, bevor ein atomares Subsystem ausgeführt wird. Daher

stellen atomare Subsysteme meistens eine zusammengehörige logische Einheit dar. Werden zu viele verwendet, verliert Simulink viele Freiheiten bei der Berechnung der Signale. Werden zu wenige verwendet, wird es schwierig, eine Aussage darüber zu machen, wann welches Signal bereit steht. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Anzahl der Blöcke** (++)
Die Anzahl der Blöcke in einem Simulink-Modell ist nur relativ zu anderen Messwerten interessant. Wird beispielsweise die Anzahl der Blöcke in Relation zu der Anzahl der Anforderungen gesetzt, kann eine Aussage getroffen werden, ob besonders viele oder besonders wenige Blöcke verwendet wurden. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Tiefe der Subsystem-Hierarchie** (++)
Die Tiefe der Subsystem-Hierarchie gibt einen Hinweis darauf, ob der Baum, den die Subsysteme bilden, balanciert ist. Dies ist allerdings nur relativ zu einer Referenz möglich. Es kann beispielsweise die »Anzahl der Subsysteme« verwendet werden. Dann kann eine Aussage getroffen werden, ob die Subsystem-Hierarchie zu tief ist oder nicht. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Anzahl der Linien** (++),
**Anzahl der Stateflow-Blöcke** (++) und
**Anzahl der Subsysteme** (++)
Die Anzahl der Linien, Stateflow-Blöcke und Subsysteme in einem Simulink-Modell ist nur relativ zu anderen Messwerten aussagekräftig. So können sie beispielsweise in Relation zu der Metrik »Anzahl der Blöcke« gesetzt werden. Dann kann eine Aussage getroffen werden, ob auffällig viele oder wenige Linien, Stateflow-Blöcke und Subsysteme in einem Modell enthalten sind. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken in einem zulässigen Bereich liegen.

**Durchschnittliche Anzahl an Stateflow-Zuständen** (++)
Eine sehr hohe Anzahl an Stateflow-Zuständen pro Stateflow-Chart deutet auf zu komplexe Zustandsautomaten hin. Sind im Durchschnitt nur sehr wenige Zustände in den Stateflow-Charts enthalten, kann das ein Indiz dafür sein, dass mehrere Stateflow-Charts zusammengefasst werden können. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Durchschnittliche Anzahl an Subsystemen pro Subsystem** (++)
Eine sehr hohe Anzahl an Subsystemen pro Subsystem kann auf eine zu starke Aufteilung in Subsysteme hinweisen. Hingegen kann eine sehr geringe Anzahl auf zu große Subsysteme mit zu vielen Blöcken hindeuten. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Anzahl der verwendeten Blocktypen** (+)
Eine hohe Anzahl an verwendeten Blocktypen kann ein Hinweis auf die Verwendung besonders exotischer Blocktypen sein. Werden nur sehr wenige Blocktypen verwendet, wurde u. U. Funktionalität nachmodelliert, die bereits in Form von vorgefertigten Blöcken zur Verfügung steht. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

### 4.7.5. Verwendung sprechender Bezeichner

Dieses Kriterium fordert, dass für die Bezeichner in einem Simulink-Modell sprechende Namen verwendet werden sollen. Das umfasst beispielsweise die Namen von Konstanten. Die Verwendung von sprechenden Bezeichnern erhöht die Verständlichkeit enorm, da durch sie zusätzliche Informationen über die Bedeutung der einzelnen Konstrukte im Modell enthalten sind.

**Anzahl der magischen Konstanten** (+++)
Magische Konstanten sind Konstanten ohne Namen. Das heißt, sie besitzen nur einen Wert. Da die Semantik einer Konstanten ohne Namen nicht ersichtlich ist, verringert eine hohe Anzahl an Konstanten mit magischen Werten die Verständlichkeit. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Anzahl der Werte magischer Konstanten** (++)
Die Anzahl an verwendeten magischen Werten macht eine Aussage darüber, wie viele verschiedene magische Werte in einem Simulink-Modell verwendet werden. Die Verständlichkeit wird stärker verringert, wenn nicht nur viele Constant-Blöcke mit magischen Konstanten verwendet werden, sondern auch viele verschiedene magische Werte. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Anzahl der Konstanten mit Namen** (+)
Die Verwendung von Konstanten mit Namen ist prinzipiell erwünscht. Es können nur sehr wenige verwendet werden, wenn z. B. keinerlei Parametrisierung notwendig ist. Ein Modell kann aber auch sehr viele Konstanten enthalten, wenn im Modell sehr viele Parameter benötigt werden. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

## 4.8. Wartbarkeit

Dieser Faktor fasst Kriterien zusammen, die erfüllt sein müssen, damit ein Simulink-Modell eine hohe Wartbarkeit hat, d. h. Änderungen sich mit geringem Aufwand durchführen lassen. In den folgenden Abschnitten werden die Kriterien vorgestellt, die zur Wartbarkeit von Simulink-Modellen beitragen.

### 4.8.1. Architektur

Dieses Kriterium fordert, dass ein Simulink-Modell bestimmte typische Subsysteme enthält. Für gewöhnlich ist ein Simulink-Modell derart aufgebaut, dass es zuerst einige Subsysteme gibt, die die Signale aufbereiten und verteilen. Dann gibt es einige Subsysteme, die zwischen verschiedenen Eingaben umschalten. Zuletzt gelangen die Signale dann in Subsysteme mit der eigentlichen Logik. Weicht ein Modell von diesem typischen Aufbau ab, werden meist verschiedene Aspekte vermischt und dadurch die Wartbarkeit verringert.

**Anzahl der arithmetischen Subsysteme** (+),
**Anzahl der logischen Subsysteme** (+),
**Anzahl der signalführenden Subsysteme** (++) und
**Anzahl der schaltenden Subsysteme** (++)
Die Klassifikationen und Zählungen dieser Metriken ermöglichen eine Aussage darüber, ob ein Simulink-Modell über eine typische Verteilung der spezialisierten Subsysteme verfügt. So ist beispielsweise ein Simulink-Modell auffällig, das vor allem signalführende Subsysteme und praktisch keine arithmetischen und logischen Subsysteme enthält, da es fraglich ist, wo die gewünschte Funktionalität umgesetzt ist. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken in einem zulässigen Bereich liegen.

### 4.8.2. Komplexität

Dieses Kriterium fordert, dass die Komplexität eines Simulink-Modells möglichst niedrig ist, damit eine gute Wartbarkeit gewährleistet ist. Sehr komplexe Modellteile erschweren Änderungen und verringern somit die Wartbarkeit. Dabei kann allerdings eine gewisse Komplexität, die von der Komplexität der jeweiligen Problemstellung vorgegeben wird, nicht unterschritten werden. Daher fordert das Kriterium eine dem Problem angemessene Komplexität.

**Höhe der globalen Komplexität** (++)
Die globale Komplexität eines Simulink-Modells ist nur relativ zu anderen Messwerten interessant. Wird beispielsweise die globale Komplexität in Relation zu der »Anzahl der Anforderungen« gesetzt, kann eine Aussage getroffen werden, ob die gemessene globale Komplexität gerechtfertigt ist oder nicht. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Durchschnittliche Blockinstabilität** (++)
Wie in [Hol11] beschrieben, ist die Intention der Metrik die Folgende: „Jeder Block in einem Simulink-Modell wird nur direkt von seinen Quellblöcken beeinflusst und beeinflusst selbst nur seine direkten Zielblöcke. Dadurch ist es umso wahrscheinlicher, dass ein Block geändert wird, je mehr er von anderen Blöcken abhängt.“. Der Messwert der Metrik kann daher als Änderungswahrscheinlichkeit interpretiert werden. Er sollte zwar nicht zu hoch sein, kann aber einen bestimmten Minimalwert nicht unterschreiten, da jeder Block mindestens einen Vorgänger haben muss. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Durchschnittlicher Vernetzungsgrad** (+++)
Je größer der durchschnittliche Vernetzungsgrad ist, desto mehr Blöcke haben Einfluss auf einen Signalwert an einer beliebigen Stelle im Modell. Dadurch wird die Komplexität erhöht, da eine lokale Betrachtung nicht ausreicht. Allerdings liegt es auch in der Natur der Datenflussmodellierung, dass Blöcke Einfluss auf andere Blöcke haben. Lediglich für ein triviales System, das beispielsweise nur aus einem Eingang und einem Ausgang besteht, ist der Einfluss sehr klein. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Durchschnittliche Breite der Ausgangsschnittstelle** (++)
Sehr breite Ausgangsschnittstellen sind ein Hinweis darauf, dass in den einzelnen Subsystemen unterschiedliche Funktionalitäten vermischt werden. Allerdings sind Subsysteme mit nur sehr wenigen Ausgängen auch auffällig, da nur wenige Ergebnisse generiert werden. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik in einem zulässigen Bereich liegt.

**Durchschnittliche Signalpfadlänge** (++) und
**Durchschnittliche Länge der Signalpfade auf dem Bildschirm** (+++)
Maximale-Identitäts-Signalpfade (vergleiche Abschnitt 3.2.1) sind Signal-Pfade mit maximaler Länge, auf denen der Wert eines Signals nicht verändert wird. MISPs werden nur durch die Hierarchieebenen eines Simulink-Modells geleitet, ohne dass Berechnungen mit anderen Signalwerten stattfinden. Das bedeutet, umso länger diese Signalpfade sind, auf denen das Signal unverändert bleibt, desto schwieriger ist das Modell zu verstehen. Ist die durchschnittliche Länge der MISPs allerdings sehr klein, enthält das Modell höchstwahrscheinlich zu wenige Subsysteme. Somit erfüllen die Metriken ihr Kriterium, wenn die Messwerte der Metriken in einem zulässigen Bereich liegen.

### 4.8.3. Standardkonformität

Dieses Kriterium fordert, dass ein Simulink-Modell Vorgaben zur Standardkonformität einhält. Standardkonforme Modelle können leichter von unterschiedlichen Modellierern bearbeitet werden. Dadurch sind Änderungen einfacher durchzuführen und die Wartbarkeit wird erhöht.

**Anzahl der verwendeten Modellierungsmuster** (+)
Die Verwendung etablierter Modellierungsmuster vereinfacht Änderungen an einem Modell. Somit erfüllt die Metrik ihr Kriterium umso besser, je größer der Messwert der Metrik ist.

### 4.8.4. Vermeidung von Redundanz

Dieses Kriterium fordert, dass Redundanzen in den Simulink-Modellen vermieden werden. Wenig Redundanz bewirkt, dass Änderungen nur an einer Stelle durchgeführt werden müssen. Somit ist auch die Gefahr geringer, dass Stellen vergessen werden. Daher wird durch eine geringe Redundanz die Wartbarkeit der Simulink-Modelle erhöht.

**Anzahl der Modellklone** (+++)
Durch das Zählen der Anzahl der Modellklone (vergleiche Abschnitt 3.2.3) kann ein Rückschluss auf die Wartbarkeit eines Modells getroffen werden. Denn Änderungen an kopierten Modellteilen sind aufwendig, da die Änderungen an allen geklonten Modellteilen durchgeführt werden müssen. Außerdem verursachen vergessene geklonte Modellteile bei Änderungen schwer zu findende Fehler. Dass eine Suche nach Redundanzen zum Auffinden von Fehlern verwendet werden kann, haben z. B. Xie und Engler in [XE02] gezeigt. Sie zeigen, dass in Code-Modulen mit vielen Redundanzen mit hoher Wahrscheinlichkeit auch schwerwiegendere Fehler zu finden sind. Somit erfüllt die Metrik ihr Kriterium umso besser, je kleiner der Messwert der Metrik ist.

**Anzahl der deaktivierten Bibliotheksverknüpfungen** (++)
Blöcke mit deaktivierten Bibliotheksverknüpfungen werden zusammen mit allen Subblöcken aus der Bibliothek in das Modell kopiert. Allerdings bleiben eventuelle Änderungen in der Bibliothek unberücksichtigt, die zum Beispiel durchgeführt wurden, um einen Fehler zu beseitigen. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Anzahl der deaktivierten und unauffindbaren Bibliotheksverknüpfungen** (+++)
Nicht auffindbare Bibliotheksverknüpfungen sind ein Problem, das beseitigt werden muss. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Anzahl der deaktivierten Bibliotheksverknüpfungen mit veränderter Struktur** (+++)
Blöcke mit deaktivierten Bibliotheksverknüpfungen werden zusammen mit allen Subblöcken aus der Bibliothek in das Modell kopiert. Dadurch sind lokale Änderungen möglich. Eine hohe Zahl an deaktivierten Bibliotheksverknüpfungen mit veränderter Struktur kann ein Hinweis auf Probleme mit der vorgegebenen Bibliothek sein, da individuelle Anpassungen notwendig sind. Somit erfüllt die Metrik ihr Kriterium, wenn der Messwert der Metrik klein bzw. im besten Fall sogar 0 ist.

**Anzahl der verwendeten Bibliotheksverknüpfungen** (++)
Eine niedrige Anzahl an verwendeten Bibliotheksverknüpfungen ist ein Indikator dafür, dass zu viel Funktionalität neu modelliert und nicht aus den vorhandenen Bibliotheken wiederverwendet wurde. Somit erfüllt die Metrik ihr Kriterium umso besser, je größer der Messwert der Metrik ist.

**Anzahl der verwendeten Bibliotheken** (+)
Eine niedrige Anzahl an unterschiedlichen Bibliotheken, die in dem Modell verwendet werden, deutet auf eine zu geringe Wiederverwendung bestehender Blöcke hin. Somit erfüllt die Metrik ihr Kriterium umso besser, je größer der Messwert der Metrik ist.

## 4.9. Zusammenfassung

Das Qualitätsmodell basiert auf der Arbeit von Cavano und McCall [CM78]. Allerdings wird nur der grundlegende Aufbau ihres Qualitätsmodells und die Faktoren auf oberster Ebene übernommen. Die Faktoren haben sich in der Praxis bewährt und haben Einzug in verschiedene Standards gefunden (vergleiche Abschnitt 1.4.2). Der Inhalt des Qualitätsmodells wurde komplett im Rahmen dieser Arbeit erstellt und ist speziell auf die modellbasierte Entwicklung mit MATLAB Simulink und Targetlink zugeschnitten.

Da sowohl manche Kriterien verschiedenen Faktoren als auch manche Metriken unterschiedlichen Kriterien zugeordnet werden können, sind viele verschiedene Variationen des Qualitätsmodells denkbar. Das in dieser Arbeit vorgeschlagene Qualitätsmodell bietet den Vorteil, dass es eine gemeinsame Basis für das Verständnis des Begriffs der Modellqualität von Simulink-Modellen bietet. Es kann projektspezifisch um neue Faktoren, Kriterien und Metriken erweitert werden oder bestehende können entfernt werden. Wichtig ist, dass ein gemeinsames Qualitätsmodell verwendet wird. Damit ist eine Vergleichbarkeit der Qualität verschiedener Simulink-Modelle gegeben.

# 5. Modellqualitätsbewertung

In diesem Kapitel wird die Modellqualitätsbewertung vorgestellt. Sie stellt das Bindeglied zwischen dem Qualitätsmodell aus Kapitel 4 und den Messwerten der Metriken aus Kapitel 3 dar. Mithilfe der Modellqualitätsbewertung werden die absoluten Messwerte der Metriken auf relative Bewertungen abgebildet. Diese Abbildung ist essentiell, da mit den absoluten Messwerten der Metriken alleine keine Aussage getroffen werden kann, ob ein Messwert in Ordnung ist oder nicht. Abbildung 5.1 zeigt schematisch, welche Schritte für die Bewertung eines Modells vorgenommen werden.

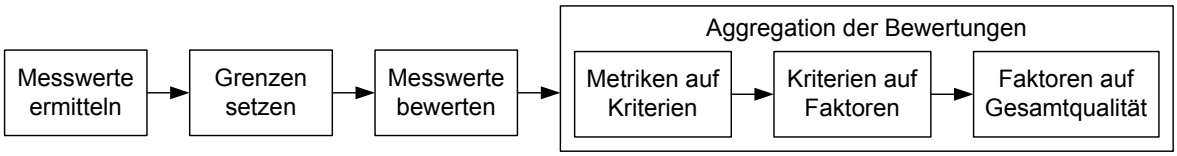



Abbildung 5.1.: Schritte der Modellqualitätsbewertung

Zuerst werden die Messwerte der Metriken aus Kapitel 3 ermittelt. Dann werden die erlaubten Grenzwerte für jede Metrik festgelegt und ihre Messwerte bewertet. Diese Bewertung der Messwerte wird in Abschnitt 5.1 beschrieben. Dazu wird in Abschnitt 5.1.1 zum einem ein Referenzmodell verwendet, das beschreibt, wie ein typisches Simulink-Modell aussieht. Durch einen Vergleich mit dem Referenzmodell werden Ausreißer gefunden. Zum anderen werden in Abschnitt 5.1.2 Regeln definiert, die Zusammenhänge zwischen den Messwerten der Metriken fordern. Nach der Bewertung der Messwerte werden schließlich die Bewertungen in dem Baum des Qualitätsmodells nach oben aggregiert, um einen einfachen Überblick über die Qualität eines Modells zu ermöglichen. Die Aggregation der Bewertungen wird in Abschnitt 5.2 im Detail dargestellt. In dieser Arbeit gelten folgende Definitionen:

**Definition 9 (Modellqualitätsbewertung).** *Die Modellqualitätsbewertung ordnet einem Simulink-Modell mithilfe des Qualitätsmodells eine Bewertung zu, die eine Aussage über die Qualität des Modells erlaubt.*

**Definition 10 (Bewertung der Messwerte).** *Die Bewertung der Messwerte ordnet jedem Messwert einer Metrik eine Bewertung von 0% bis 100% zu.*

**Definition 11 (Aggregation der Bewertungen).** *Die Aggregation der Bewertungen fasst die Bewertungen der Metriken im Qualitätsmodell von unten nach oben (von den Metriken zu den Faktoren) zusammen.*

Sowohl für die Darstellung der Messwerte mit ihren erlaubten Grenzen (siehe Abschnitt 5.1.3) als auch für die aggregierten Bewertungen (siehe Abschnitt 5.2.1) wird eine Visualisierung vorgestellt. Aus der Modellqualitätsbewertung können dann Handlungsempfehlungen abgeleitet werden. Sie geben Hinweise darauf, wie mithilfe der Ergebnisse der Modellqualitätsbewertung die Qualität der Modelle durch den Modellierer erhöht werden kann. Abschnitt 5.3 enthält exemplarisch einige Beispiele abgeleiteter Handlungsempfehlungen. Zuletzt wird in Abschnitt 5.4 vorgestellt, wie die Qualitätsbewertung über die Zeit betrachtet und interpretiert werden kann. Das Nachvollziehen der Bewertung über die Zeit erlaubt es, eine Verbesserung oder Verschlechterung der Modellqualität zu erkennen.

## 5.1. Bewertung der Messwerte

Für die Abbildung der absoluten Messwerte auf relative Bewertungen werden zwei verschiedene Ansätze verwendet. Der erste Ansatz basiert auf einem Referenzmodell und wird in Abschnitt 5.1.1 beschrieben. Das Referenzmodell wird auf Basis von empirisch gewonnenen Daten erstellt. Es beschreibt daher, wie ein typisches Simulink-Modell in einem Projekt aussieht. Der zweite Ansatz verwendet Regeln, die in Abschnitt 5.1.2 beschrieben werden. Die Regeln machen Aussagen über gewünschte Eigenschaften der Modelle. Somit können mit dem Referenzmodell Ausreißer gefunden und mit den Regeln das Vorhandensein gewünschter Zusammenhänge formuliert werden.

### 5.1.1. Referenzmodell

Ein Referenzmodell wird aus den Messwerten beliebig vieler Modelle aufgebaut. Je mehr Modelle verwendet werden, desto präziser werden die Aussagen des Referenzmodells. Das Referenzmodell ist kein reales Simulink-Modell, sondern besteht aus einer Menge von Messwerten, die sich für ein typisches Simulink-Modell ergeben würden.

Bevor ein Modell in ein Referenzmodell aufgenommen werden kann, müssen die Messwerte seiner Metriken normiert werden. Eine Normierung ist notwendig, damit Modelle unterschiedlicher Größe verglichen werden können, d. h. durch die Normierung wird das Referenzmodell größenunabhängig. Für die Normierung der Messwerte kann pro Referenzmodell theoretisch jede beliebige Metrik verwendet werden. Die Metrik, deren Messwert für die Normierung verwendet wird, wird im Folgenden *Referenzmetrik* genannt. Als Referenzmetriken kommen beispielsweise die Metriken »Anzahl der Blöcke« oder »Höhe der globalen Komplexität« in Frage, da sie eine Aussage über die Größe bzw. Komplexität eines Modells machen.

Ein Simulink-Modell wird mit einem Referenzmodell verglichen, um Ausreißer zu finden. Dazu werden alle Werte im Referenzmodell mit dem Messwert der Referenzmetrik des Simulink-Modells multipliziert. Dadurch wird das Referenzmodell so skaliert, dass es mit dem zu untersuchenden Simulink-Modell vergleichbar ist. Es ergeben sich erlaubte Werte für das Simulink-

Modell, wie z. B. die Anzahl der Subsysteme, der erlaubte durchschnittliche Subsystem-Kinder-Grad oder die maximal erlaubte Subsystemtiefe. Es wird vereinfacht von einem linearen Zusammenhang ausgegangen. Bei der Erstellung des Referenzmodells werden alle Messwerte der Modelle, die für das Referenzmodell verwendetet werden, durch den jeweiligen Messwert der Referenzmetrik geteilt. Für jede Metrik wird dann der Durchschnittswert der normierten Messwerte im Referenzmodell gespeichert. Bei der Bewertung der Qualität eines Modells ist das zu bewertende Modell nicht in der Datenbasis des Referenz-Modells enthalten. Wäre das zu bewertende Modell enthalten, würden sich die Mittelwerte und die Standardabweichung zugunsten des betrachteten Modells verändern.

Um erlaubte Grenzen für die einzelnen Metriken zu erhalten, enthält das Referenzmodell nicht nur Durchschnittswerte, sondern auch Minimal- und Maximalwerte. Die Minimal- und Maximalwerte werden durch die Verwendung der korrigierten Stichprobenvarianz und des arithmetischen Mittelwerts berechnet. Tabelle 5.1 zeigt einen Ausschnitt aus einem Referenzmodell für verschiedene Blockanzahlen. Die Datenbasis für das Referenzmodell besteht aus den Modellen, die im Kapitel 7 für die Evaluation verwendet werden. Die Modelle haben einen Umfang zwischen 1.000 und 13.800 Blöcken. Als Referenzmetrik wurde die Metrik »Anzahl der Blöcke« verwendet. Dadurch können für ein beliebiges Modell mit $n$ Blöcken die erlaubten Minimal- und Maximalwerte jeder Metrik berechnet werden, für deren Messwerte sich ein sinnvoller Mittelwert bilden lässt.

| | **200 Blöcke** | | **1.000 Blöcke** | | **5.000 Blöcke** | |
|---|---|---|---|---|---|---|
| | **Min.** | **Max.** | **Min.** | **Max.** | **Min.** | **Max.** |
| #Subsysteme | 14 | 46 | 70 | 230 | 348 | 1.149 |
| #Linien | 224 | 256 | 1.120 | 1.280 | 5.600 | 6.400 |
| #überkreuzenden Linien | 28 | 96 | 144 | 480 | 720 | 2.400 |

Tabelle 5.1.: Auszug aus dem Referenzmodell für verschiedene Blockanzahlen

Für die Abbildung der absoluten Messwerte auf die relativen Bewertungen muss für jede Metrik geprüft werden, ob ihr Messwert innerhalb des erlaubten Bereichs liegt oder nicht. Dabei kann entweder eine einfache Bereichsprüfung, bei der entweder 0% oder 100% zurückgeliefert wird, oder ein komplexerer Ansatz wie bei den Regeln im nächsten Abschnitt gewählt werden.

Die Verwendung des arithmetischen Mittels und der korrigierten Stichprobenvarianz für die Berechnung der Werte des Referenzmodells sind nicht die einzigen Möglichkeiten. Sie wurden wegen ihrer intuitiven Verständlichkeit gewählt, sind aber durch andere Verfahren ersetzbar, die u. U. besser geeignet sind. Das gesamte Konzept des Referenzmodells, das aus empirischen Daten aufgebaut wird, kann durch ein anderes Verfahren ausgetauscht werden. Würde es beispielsweise einen Ansatz geben, der noch besser ausdrücken könnte, wie ein optimales Simulink-Modell einer vorgegebenen Größe aussieht, dann könnte dieser Ansatz anstelle des Referenz-Modells verwendet werden. Die einzige Aufgabe ist es, die absoluten Messwerte der Metriken auf relative Bewertungen abzubilden, wie es mit dem Referenzmodell möglich ist.

### 5.1.2. Regeln

Für die Messwerte der Metriken, für die sich kein sinnvoller Mittelwert bilden lässt, müssen die Grenzen auf andere Art und Weise formuliert werden. Dies wird durch Regeln erreicht, die bestimmte Zusammenhänge zwischen den Messwerten fordern. So muss es z. B. in einem

Simulink-Modell gleich viele oder mehr Lese-Blöcke (DataStoreRead-Blöcke) als Speicher-Blöcke (DataStoreMemory-Blöcke) geben. Denn wenn es mehr Speicher- als Lese-Blöcke gibt, wird der Wert einiger Speicher-Blöcke im Modell nie ausgelesen. Dieser Zusammenhang wird beispielsweise durch folgende Regel beschrieben:

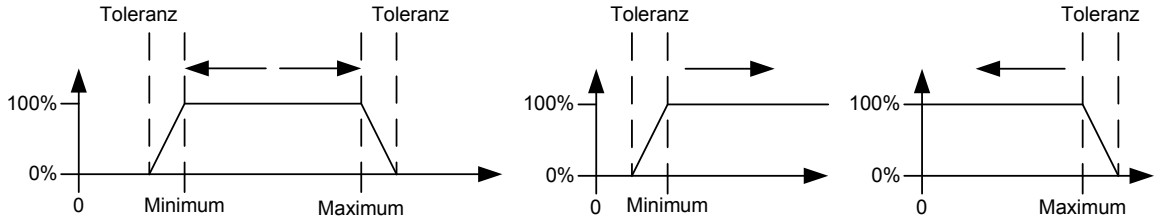
$$\texttt{Anzahl der DataStoreMemory-Blöcke} \leq \texttt{Anzahl der DataStoreRead-Blöcke}$$

Regeln können auch parametrisiert werden. Dadurch können sie z. B. an die jeweiligen Projektanforderungen angepasst werden. So kann ein Projekt durchschnittlich einen Goto-Block pro Subsystem erlauben und ein anderes Projekt Goto-Blöcke komplett verbieten. Folgende Regeln sind Beispiele für parametrisierte Regeln:

$$0 \leq \texttt{Anzahl der UnitDelay-Blöcke} \leq 2{,}0 \cdot \texttt{Anzahl der Subsysteme}$$
$$\texttt{Anzahl der CustomCode-Blöcke} = 0$$

Die erste Regel erlaubt im Durchschnitt maximal zwei UnitDelay-Blöcke pro Subsystem. Die zweite Regel hingegen verbietet die Verwendung von CustomCode-Blöcken komplett. Alle für die Evaluation in Kapitel 7 verwendeten Regeln werden im Anhang in Abschnitt A.4 aufgelistet.

Aus den Regeln werden Grenzwerte abgeleitet. Bei einer einfachen Auswertung der Regeln wird nur deren Erfüllung geprüft, d. h. die Bewertung für jede Metrik ist entweder 0% oder 100%. Um eine bessere Differenzierung der Bewertungen zu ermöglichen, können verschiedene Arten von Grenzen angegeben werden. Durch die Grenzen können Ergebnisse im Intervall [0%..100%] zurückgeliefert werden, wobei komplett erfüllte Metriken mit 100% und komplett unerfüllte Metriken mit 0% gewertet werden. Abbildung 5.2 zeigt die verschiedenen Arten von Grenzen.




(a) Intervall (b) Minimum (c) Maximum

Abbildung 5.2.: Verschiedene Arten von Grenzen

Abbildung 5.2(a) stellt ein erlaubtes Intervall zwischen zwei Werten dar. Dabei ist der Toleranzbereich auf der linken und rechten Seite ein Bereich, in dem die Grenzen zwar verletzt sind, aber die Metrik noch nicht als komplett unerfüllt gewertet werden soll. Die Art von Grenzen in Abbildung 5.2(b) und 5.2(c) hingegen kommt zum Einsatz, falls ein Wert größer oder kleiner als ein einzelner Wert sein soll. Auch hier gibt es wieder einen Toleranzbereich, in dem die Metrik als noch nicht komplett unerfüllt bewertet wird. Somit sind für jede Auswertung einer Metrik die Minimum- und Maximumwerte und die Größe des Toleranzbereichs notwendig. Das Ergebnis in den Toleranzbereichen wird durch lineare Interpolation berechnet, wobei beliebige andere Interpolationsverfahren denkbar sind.

Die verwendeten Regeln (vergleiche Abschnitt A.4 im Anhang) haben nur vereinfachte Rampen. Rampen sind mit den Minimum- und Maximumregeln aus den Abbildungen 5.2(b) und 5.2(c) vergleichbar, wobei sie nur aus dem Toleranzbereich bestehen. Das Minimum spielt dabei keine Rolle, da es immer 0 ist und die Messwerte keine Werte kleiner null annehmen

können. Eine Rampe wird zum Beispiel in der Regel über die erlaubte Anzahl der UnitDelay-Blöcke verwendet:

$$0 \leq \texttt{Anzahl der UnitDelay-Blöcke} \leq 2{,}0 \cdot \texttt{Anzahl der Subsysteme}$$

Die Rampe bedeutet, dass die Metrik mit 100% bewertet wird, wenn kein einziger UnitDelay-Block in dem bewerteten Modell enthalten ist. Sind hingegen genau doppelt so viel oder mehr UnitDelay-Blöcke wie Subsystem-Blöcke in dem Modell enthalten, so wird die Metrik mit 0% bewertet. Enthält das Modell aber beispielsweise gleich viele UnitDelay-Blöcke wie Subsystem-Blöcke, dann wird die Metrik mit 50% bewertet.

### 5.1.3. Visualisierung

Abbildung 5.3 zeigt einige Messwerte eines Modells und ihre Grenzen in einem Kiviat-Diagramm (vergleiche [Dra96]). Ein Kiviat-Diagramm ermöglicht, schnell einen Überblick über viele Messwerte zu erhalten. Jede Achse stellt den Messwert einer Metrik dar. Dabei ist die Skalierung jeder Achse so gewählt, dass ihr erlaubter minimaler Wert auf dem inneren Kreis und ihr erlaubter maximaler Wert auf dem äußeren Kreis liegt.

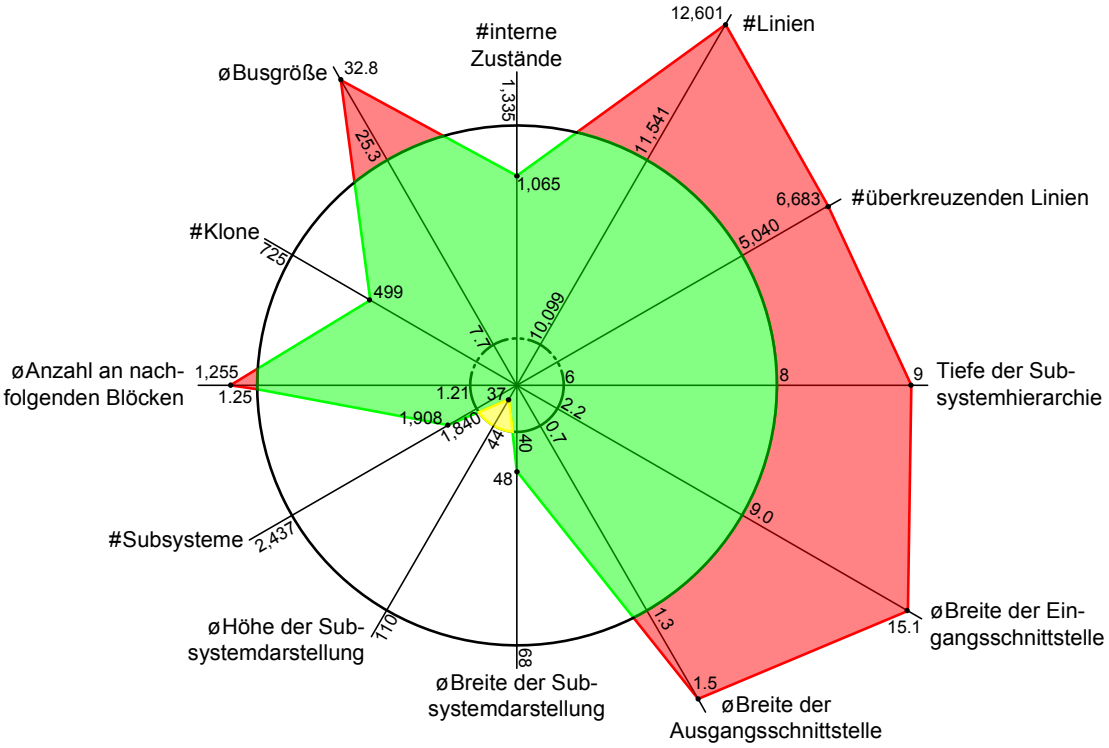



Abbildung 5.3.: Messwerte und ihre Grenzen

Durch das Verbinden der Messpunkte entsteht ein Polygon, dessen rot bzw. gelb eingefärbten Flächen die Ausreißer hervorheben. Rot eingefärbte Flächen überschreiten den erlaubten maximalen Wert und gelbe Flächen unterschreiten den erlaubten minimalen Wert. Aus Gründen der Übersicht sind in Abbildung 5.3 keine Toleranzbereiche dargestellt.

## 5.2. Aggregation der Bewertungen

In Abschnitt 5.1 wurde die Bewertung der Messwerte beschrieben. Nach der Bewertung haben alle Blätter im Qualitätsmodell eine Bewertung zwischen 0% und 100%. Der nächste Schritt ist die Aggregation der Bewertungen im Qualitätsmodell, d. h. es wird eine Vorschrift benötigt, die beschreibt, wie die Bewertungen der Metriken zuerst auf die Kriterien und dann wiederum auf die Faktoren aggregiert werden können. Die Aggregation dient dazu, einen schnellen Überblick über die Qualität eines Modells zu ermöglichen.

Bei der Aggregation soll beispielsweise ein Kriterium mit vier Metriken, von denen drei mit 100% und eine mit 0% erfüllt sind, nicht als zu 75% erfüllt bewertet werden. Denn wenn eine oder mehrere Metriken gar nicht bzw. nur sehr schlecht erfüllt sind, soll das zu einer zusätzlichen Abwertung führen. Ansonsten können einzelne Probleme in einem sonst guten Modell u. U. nicht erkannt werden. Somit muss die Bewertung des Kriteriums schlechter als der Mittelwert sein. Das kann durch Multiplikation des arithmetischen Mittels mit dem Prozentsatz der Bewertungen erreicht werden, die größer als ein geforderter *Schwellwert* sind. Somit ergibt sich folgende Formel zur Berechnung der Gesamtbewertung:

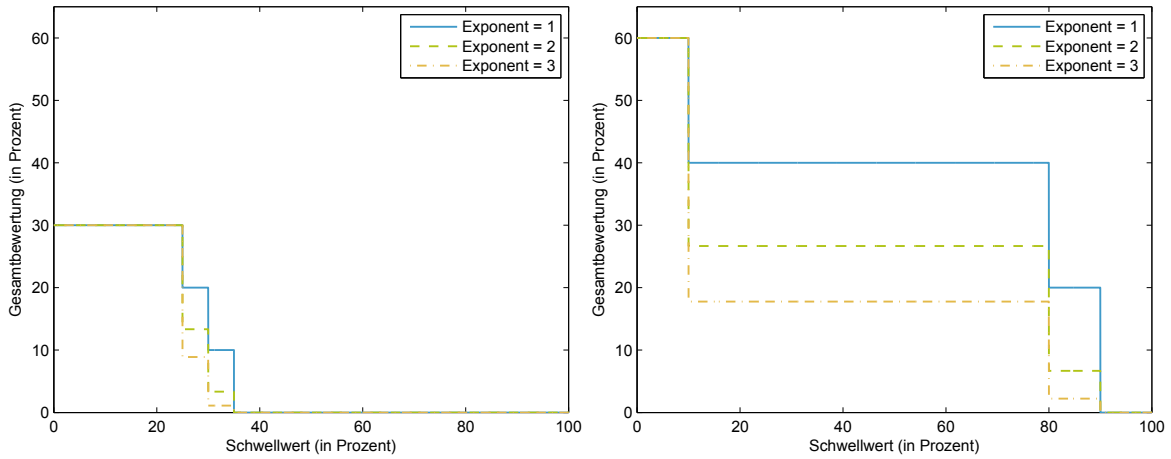
$$gesamtbewertung = \underbrace{\frac{\sum_i bewertung_i}{\#bewertungen}}_{arithmetisches\ Mittel} \cdot \underbrace{\left(\frac{\#bewertungen\ddot{U}berSchwelle}{\#bewertungen}\right)^{exponent}}_{Dämpfungsfaktor} \tag{5.1}$$

Der Dämpfungsfaktor kann zusätzlich mittels eines Exponenten noch weiter verkleinert werden, damit Bewertungen unter dem geforderten Schwellwert zu einer noch deutlich schlechteren Gesamtbewertung führen. Liegen alle Bewertungen über dem Schwellwert, entspricht die Gesamtbewertung dem arithmetischen Mittel. Dabei kann der Schwellwert entweder global oder pro Aggregationsschritt gewählt werden. Werden die zu aggregierenden Bewertungen gewichtet, muss anstatt des einfachen arithmetischen Mittels ein gewichtetes arithmetisches Mittel verwendet werden. Gewichtungen können dazu verwendet werden, um auszudrücken, ob eine nicht erfüllte Metrik großen Einfluss auf die Gesamtqualität hat oder nicht. So kann abgebildet werden, dass beispielsweise ein Messwert größer null bei der Metrik »Anzahl der unerreichbaren Zustände« einen großen Einfluss zur Laufzeit hat, wohingegen viele Linienüberkreuzungen zwar unschön, aber längst nicht so problematisch sind. Die verwendeten Gewichte für die Evaluation in Kapitel 7 sind in Abschnitt A.3 im Anhang aufgelistet.

Die Wahl des Schwellwerts ist projektspezifisch und kann z. B. in Abhängigkeit des ASIL (Automotive Safety Integrity Level) der ISO 26262[1] gewählt werden. Der ASIL bestimmt, wie viele der geforderten Maßnahmen der ISO erfüllt werden müssen.

Abbildung 5.4 zeigt zwei Gesamtbewertungen mit jeweils drei unterschiedlichen Exponenten. Die einzelnen Bewertungen sind unter den jeweiligen Abbildungen angegeben. Auf der x-Achse ist der Schwellwert und auf der y-Achse die Gesamtbewertung aufgetragen. Gut sichtbar ist, dass wenn keine der Bewertungen über dem Schwellwert liegt, der Dämpfungsfaktor den Wert 0 annimmt und somit die Gesamtbewertung 0 ist. Ebenso erkennbar ist, dass je höher der Exponent gewählt wird, desto kleiner werden die Gesamtbewertungen bei Nichterreichung des Schwellwerts.

[1] `http://www.iso.org/iso/catalogue_detail?csnumber=51362`




(a) Bewertungen [25%, 30%, 35%]

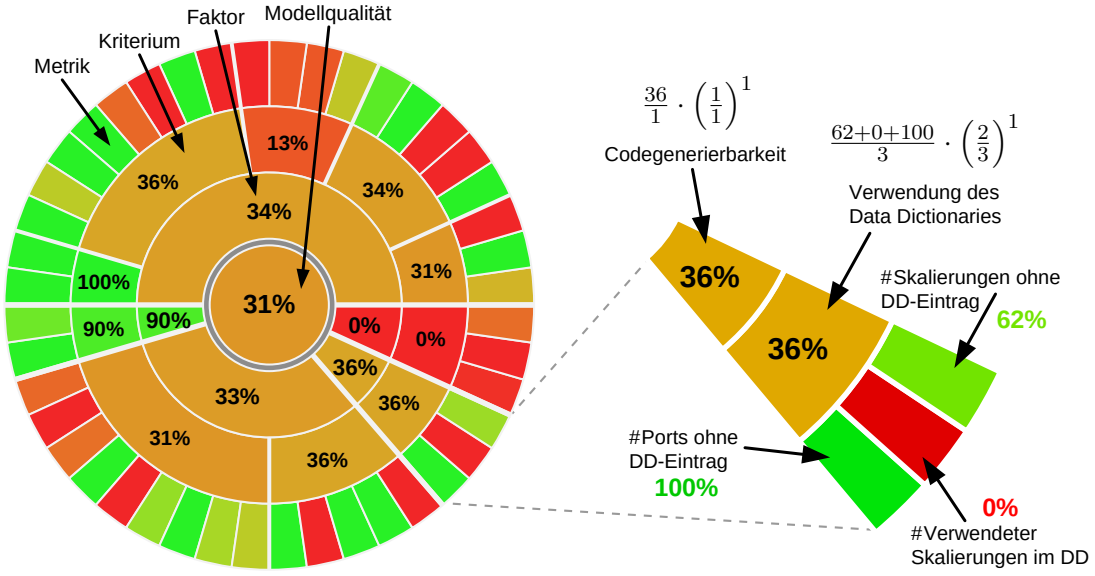
(b) Bewertungen [10%, 80%, 90%]

Abbildung 5.4.: Verhalten der Aggregation

Analog zur Aggregation von den Metriken auf die Kriterien wird die Aggregation von den Kriterien auf die Faktoren vorgenommen. Falls gewünscht, ist es auch möglich, einen finalen Wert für die Modellqualitätsbewertung zu berechnen, der schließlich durch die Aggregation der Faktoren zustande kommt. Allerdings ist die Aussagekraft dieses Werts nicht mehr sehr hoch, da nur bedingt ersichtlich ist, welche Faktoren für die Bewertung verantwortlich sind.

### 5.2.1. Visualisierung

Abbildung 5.5 zeigt die Aggregation der Bewertungen im Qualitätsmodell. Es wird eine scheibenförmige Darstellung des Qualitätsmodells verwendet.




(a) Komplette Modellqualitätsbewertung

(b) Bewertung eines Faktors

Abbildung 5.5.: Aggregation der Bewertungen mit Schwellwert = 20% und Exponent = 1

Dabei stellt der äußere Ring die Bewertungen der Metriken dar. Im nächsten Ring nach innen hin, werden die zu den Metriken gehörenden Kriterien dargestellt. Ab hier wird zusätzlich die Bewertung in Prozent angegeben. Auf die Kriterien folgen die Faktoren. In der Mitte wird schließlich die Gesamtbewertung der bewerteten Faktoren angegeben. Die Farbe jedes Segments kodiert die Bewertung des Segments, indem die Bewertung auf einen linearen Übergang zwischen den Farben Grün und Rot abgebildet wird. So sind z. B. vollständig erfüllte Segmente grün, halb erfüllte orange und nicht erfüllte rot.

Abbildung 5.5(a) zeigt die Bewertung eines kompletten Qualitätsmodells. Abbildung 5.5(b) hingegen zeigt exemplarisch, wie die Bewertung eines Faktors zustande kommt. Die Bewertung von 36% kommt dadurch zustande, dass der Dämpfungsfaktor aus Formel 5.1 bei der Aggregation der Bewertungen der Metriken den Wert $^2/_3$ annimmt, weil eine der Bewertungen nicht über dem geforderten Schwellwert von 20% liegt.

## 5.3. Handlungsempfehlungen

Handlungsempfehlungen geben Hinweise, wie die Qualität eines Simulink-Modells aus Sicht der Modellqualitätsbewertung verbessert werden kann. Dazu werden die Metriken mit einer schlechten Bewertung betrachtet und zusammen mit dem Qualitätsmodell wird festgelegt, in welche Richtung sich die Messwerte bewegen müssen. Somit tragen Handlungsempfehlungen per Definition zu einer Erhöhung der Modellqualität bei, da sie Empfehlungen geben, die die Messwerte in den erlaubten Bereich bringen und somit zu einer höheren Modellqualitätsbewertung führen. In diesem Abschnitt wird exemplarisch anhand von drei Metriken gezeigt, wie Handlungsempfehlungen abgleitet werden.

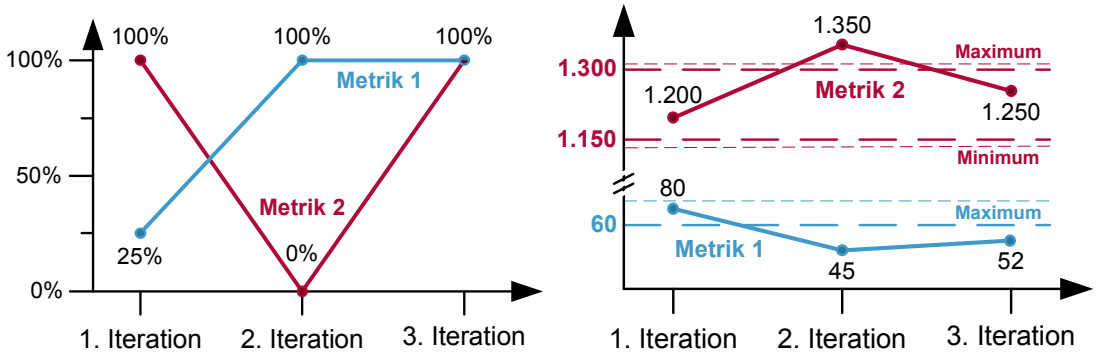
Die Metrik »Tiefe der Subsystem-Hierarchie« berechnet die Tiefe der Subsystem-Hierarchie. Das Qualitätsmodell gibt vor, dass die Messwerte dieser Metrik in einem bestimmten Bereich liegen sollen. Das bedeutet, sie sollen weder zu klein noch zu groß sein. Ist der Messwert dieser Metrik außerhalb des erlaubten Bereichs, so muss er durch Modell-Refactoring in den erlaubten Bereich gebracht werden. Modell-Refactoring beschreibt in diesem Fall, analog zu Code-Refactoring, den strukturellen Umbau eines Modells ohne eine Veränderung der von außen wahrnehmbaren Funktionalität. Stürmer et al. beschreiben beispielsweise in [Stü+07] einen Ansatz für das Refactoring von Simulink-Modellen mittels Graphentransformationen. Somit müssen je nach dem, ob der Messwert zu klein oder zu groß ist, Subsysteme entfernt oder hinzugefügt werden.

Analog kann eine Handlungsempfehlung für die Metrik »Durchschnittliche Busgröße« abgeleitet werden. Im Qualitätsmodell wird ebenfalls gefordert, dass ihre Messwerte in einem vorgegebenen Bereich liegen. Auch hier muss wieder durch Modell-Refactoring dafür gesorgt werden, dass entweder mehr Signale zu Bussen zusammengefasst werden oder große Busse in kleinere aufgeteilt werden.

Bei der Metrik »Anzahl der Subsystem-Annotationen« wird im Qualitätsmodell gefordert, dass die Messwerte so groß wie möglich sind. Denn je mehr Kommentare in den Subsystemen eines Simulink-Modells enthalten sind, desto besser dokumentiert ist das Modell. Ist der Messwert der Metrik zu gering, müssen mehr Kommentare, d. h. Annotationen, in die Subsysteme eingefügt werden.

## 5.4. Nachvollziehen der Bewertung über die Zeit

Bei der Trendanalyse werden zwei oder mehr Stände eines Modells miteinander verglichen. Abbildung 5.6 zeigt beispielhaft die Bewertung zweier Metriken über drei Iterationen des Entwicklungsprozesses. In Abbildung 5.6(a) ist der relative Trend dargestellt. Wenn allerdings nur die Bewertungen der Metriken aus Abschnitt 5.1 dargestellt sind, ist kein Rückschluss möglich, wie weit die Messwerte einer Metrik von ihren erlaubten Grenzen entfernt sind. Daher zeigt Abbildung 5.6(b) den absoluten Trend der Metriken. In dieser Darstellung ist sichtbar, wie die Messwerte der Metriken im Verhältnis zu ihren erlaubten Grenzen und Toleranzbereichen liegen. So ist z. B. erkennbar, warum Metrik 2 in der 2. Iteration den Wert 0% hat. Außerdem kann mit der absoluten Darstellung ermittelt werden, ob sich ein Messwert in Richtung seiner erlaubten Grenzen bewegt, obwohl er in der relativen Darstellung immer 0% ist. Somit kann mithilfe der Trendanalyse nachvollzogen werden, wie die Qualitätsbewertung zu den einzelnen Zeitpunkten zustande gekommen ist.




(a) Relative Darstellung

(b) Absolute Darstellung

Abbildung 5.6.: Trendanalyse eines Modells

Des Weiteren lassen sich in der Darstellung des relativen Trends nicht nur Metriken darstellen, sondern jedes Element des Qualitätsmodells. So ist es auch möglich, sich einen Überblick über die Entwicklung bestimmter Kriterien oder Faktoren zu verschaffen.

## 5.5. Zusammenfassung

Die Modellqualitätsbewertung wird mithilfe des Qualitätsmodells und der Metriken automatisiert durchgeführt. Dazu werden zuerst die Metriken bewertet und dann die Bewertungen im Qualitätsmodell aggregiert. Die Modellqualitätsbewertung ist von der Größe des Modells unabhängig und ermöglicht somit eine automatisierte Modellqualitätsbewertung großer und komplexer Modelle.

Der Algorithmus zur Aggregation der Bewertungen im Qualitätsmodell wurde gewählt, da er sehr intuitiv ist. Die Verwendung des arithmetischen Mittels ist leicht nachvollziehbar und eine Berücksichtigung besonders schlechter Bewertungen bietet sich an. Allerdings sind auch andere Algorithmen zur Aggregation denkbar. Des Weiteren müssen bei der Modellqualitätsbewertung die für das Referenzmodell verwendeten Modelle sorgfältig ausgewählt werden. Denn diese Modelle dienen als Referenz für die Bewertungen des zu untersuchenden Modells. Nur wenn die für das Referenzmodell ausgewählten Modelle qualitativ hochwertig sind, ist

ein Vergleich sinnvoll. Eine robustere Möglichkeit für die Erstellung eines Referenzmodells wäre ein konstruktives Vorgehen, das die Kontrolle über alle zur Erstellung verwendeten Parameter erlaubt.

Die in Abschnitt 5.4 vorgestellte Nachvollziehbarkeit über die Zeit ist ideal im Zusammenspiel mit dem Konzept der *kontinuierlichen Integration*. Bei der kontinuierlichen Integration wird nach jeder Änderung ein neues Release der Software erzeugt. Übertragen auf die modellbasierte Entwicklung wird für jedes Modell geprüft, ob fehlerfrei Code erzeugt werden kann. Zusätzlich kann eine Modellqualitätsbewertung durchgeführt werden. Durch die langfristige Beobachtung der Messwerte und Bewertungen ist es möglich, bei Problemen geeignete Gegenmaßnahmen einzuleiten (vergleiche Abschnitt 5.3 über Handlungsempfehlungen).

# 6. Implementierung

In diesem Kapitel wird die Implementierung des Verfahrens beschrieben. Zuerst wird in Abschnitt 6.1 geschildert, wie der Prototyp mithilfe eines Metamodells auf die Simulink-Modelle zugreift. In Kapitel 3 wurden die verwendeten Metriken vorgestellt. Dabei wurden die Datenquellen der Metriken noch außen vor gelassen, da von der konkreten Umsetzung der Metriken abstrahiert wurde. In Abschnitt 6.2 hingegen wird aufgezeigt, aus welchen Quellen die einzelnen Metriken ihre Daten in der Implementierung beziehen. Die Architektur des Prototyps und seine einzelnen Module werden in Abschnitt 6.3 beschrieben. Abschließend wird in Abschnitt 6.4 auf Einschränkungen des Prototyps eingegangen.

## 6.1. Metamodell für Simulink

Momentan gibt es keine Möglichkeit, auf generische Art und Weise auf MATLAB Simulink-Modelle zuzugreifen. Für einen generischen Zugriff wird ein Metamodell der Simulink-Modelle und eine Infrastruktur zur Kontaktaufnahme mit MATLAB benötigt. Zwar wird versucht, in dem MAJA-Projekt[1] eine solche Zugriffsmöglichkeit zu schaffen, allerdings befindet sich das Projekt noch in der Entwicklung. MathWorks selbst stellt weder die benötigte Infrastruktur für einen externen Zugriff noch ein Metamodell bereit.

Die in der Literatur verwendeten Metamodelle sind immer an ihre jeweilige Aufgabe angepasst. So stellen beispielsweise Giese et al. in [GMW06] ein Metamodell vor, das für die Richtlinienprüfung und Transformation von Simulink-Modellen verwendet wird. Die Transformation wird für die Reparatur gefundener Richtlinienverletzungen verwendet. Auch das von Amelunxen et al. in [AKRS06] verwendete Simulink-Metamodell wird zur Modell-zu-Modell-Transformation eingesetzt. Neema et al. hingegen verwenden in [NSK03] ein Metamodell von Simulink, um aus einzelnen Modellteilen mithilfe eines Verfahrens zur Modellsynthese komplette Simulink-Modelle zu generieren. Die verwendeten Metamodelle ähneln sich meist und unterscheiden sich nur in Details.

Da ein allgemeines Metamodell fehlt, wurde im Rahmen dieser Arbeit ein Metamodell erstellt, das für die Anwendung von Modellmetriken optimiert ist. Abbildung 6.1 zeigt einen Überblick über das in dieser Arbeit verwendete Metamodell für Simulink und Stateflow-Anteile. Die Abbildung enthält einen Ausschnitt aus dem Metamodell und ist in Bezug auf die gezeigten Blocktypen nicht vollständig.

[1] www.es.tu-darmstadt.de/forschung/domaenenspezifische-sprachen-methoden/maja

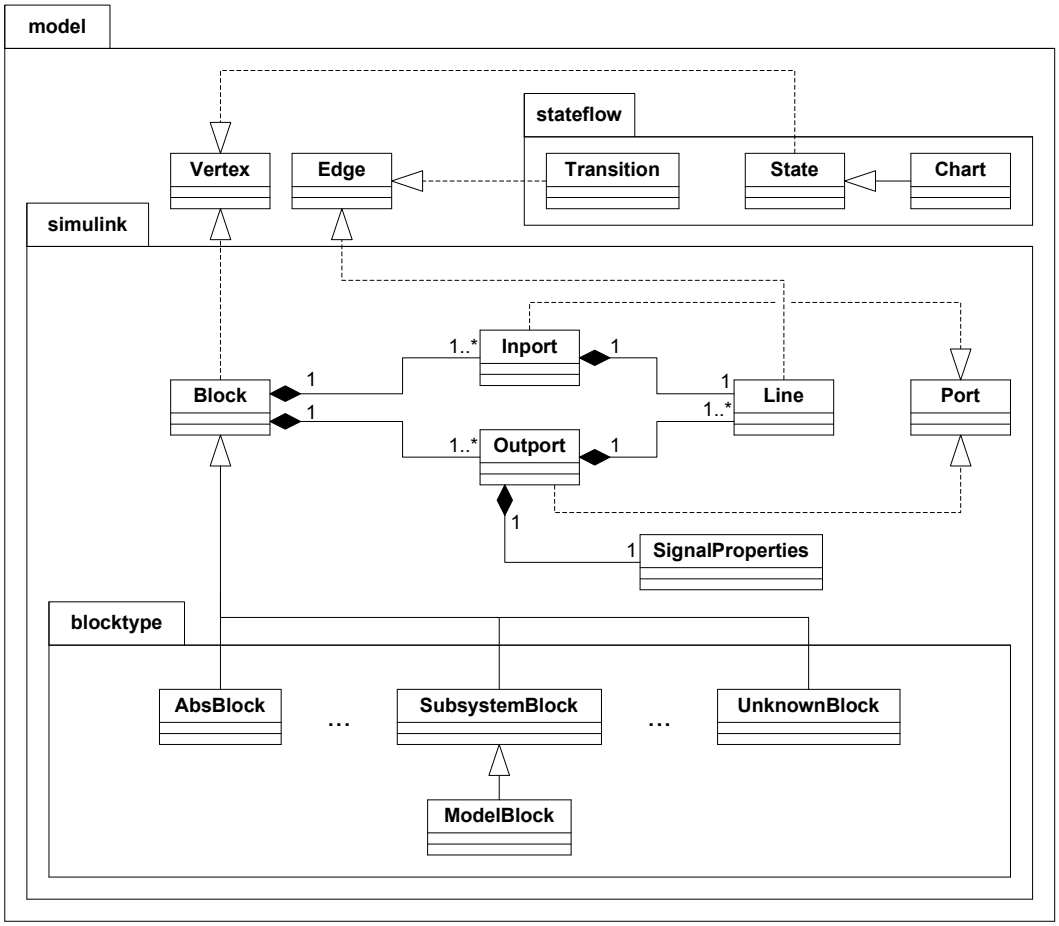



Abbildung 6.1.: Metamodell für Simulink-Modelle

Das Metamodell wird für den Zugriff auf die Simulink-Modelle in Java benötigt. Es beschreibt, wie ein Simulink-Modell im Speicher repräsentiert wird. Für die Simulink-Anteile eines Modells sind die Blöcke und Linien die zentralen Elemente, wohingegen für die Stateflow-Anteile eines Modells die Zustände und Übergänge die zentralen Elemente sind. Jeder unterstützte Simulink-Blocktyp besitzt eine korrespondierende Klasse mit den Attributen des Blocktyps. Abbildung 6.2 zeigt die Klassen `Block` und `SubsystemBlock` mit ihren wichtigsten Eigenschaften.

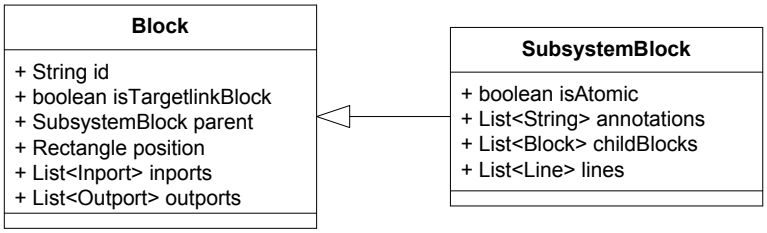



Abbildung 6.2.: Auszug aus dem Metamodell

Das in dieser Arbeit verwendete Metamodell ist nicht vollständig in Bezug auf die Blocktypen und Eigenschaften von Simulink. Es wurden nur die von den Metriken benötigten Umfänge modelliert.

## 6.2. Datenquellen der Metriken

Einige Metriken benötigen komplexe Algorithmen, um Aussagen über die Simulink-Modelle treffen zu können. Diese komplexen Algorithmen lassen sich nicht ohne Weiteres im Rahmen eines Prototyps implementieren, da der Aufwand sehr hoch wäre. Außerdem existieren u. U. bereits Produkte, die diese Algorithmen implementieren und deren Ergebnisse verwendet werden können. Ein Beispiel dafür ist das Verfahren des Model Checkings, wie es in der Metrik »Anzahl der Wertebereichsverletzungen« in Abschnitt 3.9 verwendet wird. Model Checker für Simulink sind beispielsweise der Simulink Design Verifier[2] und der EmbeddedValidator[3]. Die Schnittstelle zu den externen Tools bilden die Reports, die von den Tools generiert werden. Diese Reports werden im Folgenden *Datenquellen* genannt. Tabelle 6.1 listet alle Metriken auf, deren Datenquellen nicht ausschließlich das Modell selbst sind. Nach der Klassifikation aus Kapitel 3 sind somit alle Metriken, die direkt auf dem Modell messen, *direkt* und alle Metriken, die auf externe Datenquellen angewiesen sind, *indirekt*.

| **Datenquelle** | **Metrik** |
|---|---|
| Modellierungsmustersuchtool | Metrik 62: Anzahl der verwendeten Modellierungsmuster |
| Anforderungsverwaltungstool | Metrik 83: Anzahl der Anforderungen<br>Metrik 84: Anforderungsabdeckung |
| Testtool | Metrik 80: Anzahl der Testfälle<br>Metrik 81: Erfolgreiche Testfälle<br>Metrik 82: Testabdeckung |
| Model Checker | Metrik 60: Anzahl der Wertebereichsverletzungen<br>Metrik 70: Anzahl der unerreichbaren Zustände |
| Modellierungsrichtlinienprüfer | Metrik 66: Anzahl der verletzten dSPACE-Richtlinien<br>Metrik 67: Anzahl der verletzten MISRA TL-Richtlinien<br>Metrik 78: Anzahl der potentiellen Datentyp-Probleme |
| Komplexitätsanalysetool | Metrik 63: Höhe der globalen Komplexität<br>Metrik 64: Durchschnittliche lokale Komplexität |
| `sldiagnostics` | Metrik 59: Anzahl der internen Zustände |
| `export_tl_errors` | Metrik 71: Anzahl der Targetlink-Fehler<br>Metrik 72: Anzahl der Targetlink-Warnungen |
| WCET-Analysetool | Metrik 79: Worst-Case-Execution-Time |

Tabelle 6.1.: Datenquellen der Metriken

Der Prototyp enthält z. B. eine Anbindung an den Modellierungsrichtlinienprüfer MXAM[4] und das Komplexitätsanalysetool MXRAY[5]. Außerdem können die Reports des Simulink-Befehls `sldiagnostics` und des Targetlink-Befehls `export_tl_errors` eingelesen werden.

## 6.3. Architektur und Module

In diesem Abschnitt wird zuerst ein Überblick über die Architektur des Prototyps gegeben und dann auf die einzelnen Module eingegangen. Dazu werden im Folgenden zur Illustra-

[2] `http://www.mathworks.de/products/sldesignverifier/index.html`
[3] `http://www.btc-es.de/index.php?lang=2&idcatside=5`
[4] `http://www.model-engineers.com/de/our-products/model-examiner.html`
[5] `http://www.model-engineers.com/de/our-products/m-xray.html`

tion vereinfachte Klassendiagramme der Implementierung verwendet. Abbildung 6.3 zeigt einen Überblick über die Module des Prototyps und die beteiligten Artefakte. Artefakte mit einer gestrichelten Linie sind temporäre Artefakte, die nur zur Laufzeit eines Moduls existieren. Als erstes wird mithilfe des Modell-Exporters das zu untersuchende Simulink-Modell als XML-Datei gespeichert. Dann werden im *Model Measurer* die Messungen durchgeführt. Die Metriken verwenden entweder das Simulink-Modell oder externe Datenquellen. Die Messwerte werden abgespeichert. Im *Measurement Database Builder* wird dann eine Messwertdatenbank erstellt, die später zur Erzeugung des Referenz-Modells dient. Für die Erzeugung der Messwertdatenbank werden Messwerte von anderen Modellen benötigt. Im *Quality Rater* wird die Modellqualitätsbewertung vorgenommen. Dazu werden das aus der Messwertdatenbank erstellte Referenzmodell, das Qualitätsmodell und die Regeln verwendet. Das Ergebnis ist eine Modellqualitätsbewertung, die dann im *Quality Investigator* visualisiert werden kann.

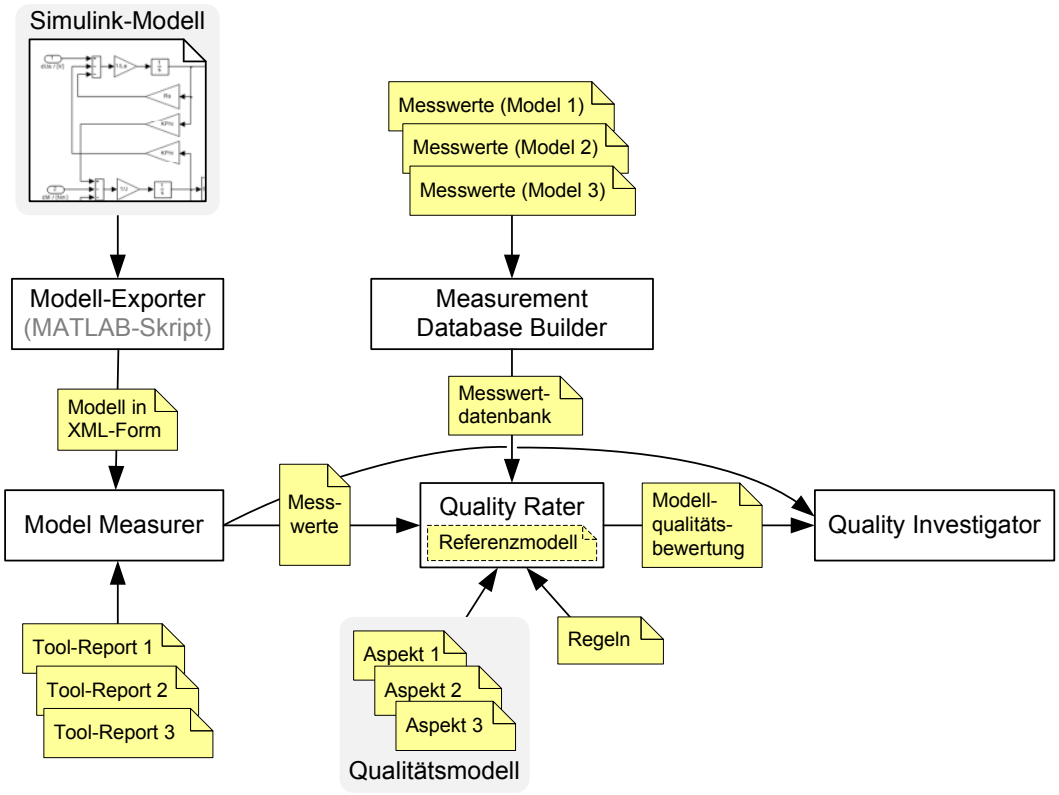



Abbildung 6.3.: Module und Artefakte des Prototyps

Ein Großteil des Prototyps wurde in Java implementiert. Einzig der Modell-Export aus Simulink in Abschnitt 6.3.1 wurde als MATLAB-Skript umgesetzt. Zur Speicherung der Artefakte wird meist XML oder ein eigenes, einfaches textuelles Format verwendet. Die Visualisierung in Abschnitt 6.3.5 wurde mit einer Swing-Oberfläche realisiert. Im Folgenden werden die einzelnen Module und die beteiligten Artefakte im Detail beschrieben.

### 6.3.1. Modell-Exporter

**Eingaben**

| | |
|---|---|
| Simulink-Modell | Ein updatefähiges Simulink-Modell, das bereit für die Simulation oder Codegenerierung ist. |

**Ausgaben**

| | |
|---|---|
| Modell in XML-Form | Als XML-Datei serialisierte Form des Modells. Enthält Blöcke, Linien und Informationen, die erst nach einen Modell-Update verfügbar sind. |

Der Modell-Exporter stellt eine serialisierte Form eines Simulink-Modells zur Verfügung. Es gibt zwei Hauptgründe, warum die Modelle serialisiert werden müssen. Zum einen sollen die Modelle, auf denen gemessen werden soll, einfach abgelegt und archiviert werden können. Zum anderen sollen die Metriken und die Logik für die Qualitätsbewertung in Java implementiert werden, da größere Projekte in Java sehr viel leichter umzusetzen sind als in MATLAB-Skript.

Für den Export der Modelle wird in einem MATLAB-Skript über ein in Simulink geladenes und update-fähiges Modell iteriert. Die Erzeugung des XML-Exports aus Simulink heraus ist notwendig, da die MDL-Dateien von Simulink zwar im Klartext vorliegen, aber nicht alle benötigten Informationen enthalten. Wenn nur die Modell-Dateien mit einem Parser, wie beispielsweise der ConQAT Simulink Library[6], gelesen werden, fehlen alle Informationen, die von Simulink erst zur Laufzeit ermittelt werden. Das umfasst beispielsweise den Inhalt von Bussen oder im Modell propagierte Datentypen (vergleiche Abschnitt 2.2.5). Außerdem müsste die Logik von Simulink für die Behandlung von Bibliotheken nachgebildet werden, da die Bibliotheken in eigenen Modell-Dateien gespeichert werden. Zwar kann MATLAB in neueren Versionen die Modell-Dateien als XML-Datei speichern, jedoch enthalten diese nur die Informationen, die auch in den MDL-Dateien selbst enthalten sind. Aus diesen Gründen wird der Modell-Exporter für den Export der Modelle verwendet.

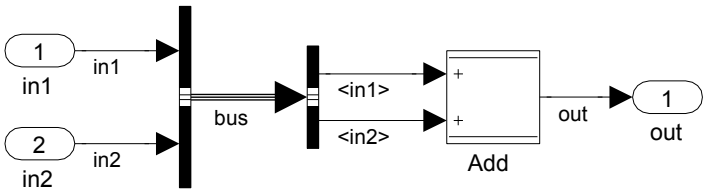



Abbildung 6.4.: Beispiel-Export in Simulink

Während der Iteration über die Subsystemhierarchie werden die Blöcke und Linien des Modells auf XML-Tags abgebildet. Abbildung 6.4 zeigt ein Simulink-Modell und Listing 6.1 zeigt Ausschnitte aus dem dazugehörigen XML-Export. In Zeile 3 in Listing 6.1 beginnt der Tag für alle Subblöcke des darüberliegenden Subsystems, in diesem Fall des Modells selbst. Ab Zeile 5 ist die Repräsentation des Add-Blocks zu finden. Sie enthält ab Zeile 8 die Inports und ab Zeile 16 die Outports. Die Linien der Subblöcke werden in dem Tag ab Zeile 27 gespeichert. Zusätzlich wird für jeden Block vermerkt, ob er ein Targetlink- oder Simulink-Block ist.

```
1 <modelBlock xmlId="1" id="xmlExportSample" name="xmlExportSample"
2         isTargetlinkBlock="false">
3     <childBlocks xmlId="2">
```

[6] http://conqat.in.tum.de/index.php/Simulink_Library

```
4         ...
5         <addBlock xmlId="48" id="xmlExportSample/Sum" name="Sum"
6                 isTargetlinkBlock="true" date="25-Nov-2011 11:00:52">
7             <position xmlId="49" x="330" y="335" width="40" height="40"/>
8             <inports xmlId="50">
9                 <inport xmlId="51" portNumber="1" type="INPORT">
10                    <position xmlId="52" x="315" y="345"/>
11                </inport>
12                <inport xmlId="53" portNumber="2" type="INPORT">
13                    <position xmlId="54" x="315" y="365"/>
14                </inport>
15            </inports>
16            <outports xmlId="55">
17                <outport xmlId="56" portNumber="1" type="OUTPORT">
18                    <position xmlId="57" x="375" y="355"/>
19                    <signalProperties xmlId="59" signalName="out" lsb="1" offset="0"
20                            scaling="" variable="" checkMin="false" checkMax="false"
21                            type="Int16" memoryClass="default"/>
22                </outport>
23            </outports>
24        </addBlock>
25        ...
26    </childBlocks>
27    <lines xmlId="65">
28        <line xmlId="66">
29            <source xmlReference="44"/>
30            <destination xmlReference="53"/>
31            <points xmlId="67">
32                <point xmlId="68" x="280" y="365"/>
33                <point xmlId="69" x="325" y="365"/>
34            </points>
35        </line>
36        ....
37</modelBlock>
```

Listing 6.1: Beispiel-Export in XML

Listing 6.2 zeigt einen weiteren Ausschnitt aus dem XML-Export des Modells aus Abbildung 6.4. Enthalten sind Informationen über die Outports des BusCreator-Blocks. In Zeile 7 und Zeile 11 beginnen die Tags, in denen Informationen über das Signal am Ausgang des BusCreator-Blocks gespeichert sind. Der Tag in Zeile 11 enthält die Namen der Signale in dem Bus. Wohingegen der Tag in Zeile 7 die Simulink-interne Beschreibung des Signals enthält. Die $-2$ als erster Eintrag der Dimensionalität des Signals bedeutet, dass es sich um einen Bus handelt. Diese Simulink-interne Darstellung wird in den Java-Klassen aufbereitet und in eine intuitivere Form umgewandelt.

```
1 ...
2     ...
3         <outports xmlId="24">
4             <outport xmlId="25" portNumber="1" type="OUTPORT">
5                 <block xmlReference="17"/>
6                 <position xmlId="26" x="225" y="355"/>
7                 <dimensions xmlId="27">
8                     <int>-2</int><int>2</int><int>1</int>
9                     <int>1</int><int>1</int><int>1</int>
10                </dimensions>
```

```
11                  <signalProperties xmlId="28" signalName="bus">
12                      <busSignals>
13                          <signalProperties xmlId="29" signalName="in1"/>
14                          <signalProperties xmlId="30" signalName="in2"/>
15                      </busSignals>
16                  </signalProperties>
17              </outport>
18          </outports>
19      ...
20  ...
```

Listing 6.2: Bus-Informationen

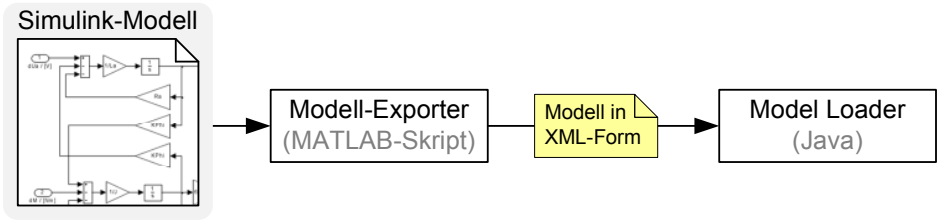



Abbildung 6.5.: Überblick über den Modell-Export und Import

Abbildung 6.5 zeigt einen Überblick über den Export eines Simulink-Modells und den Import in einer Java-Anwendung. Nach dem Laden der XML-Datei in Java, d. h. dem Erzeugen der korrespondierenden Instanzen des Metamodells aus Abschnitt 6.1, werden in Java noch einige Nachbearbeitungsschritte durchgeführt:

**Verbinden der Ports:** Da in den verwendeten XML-Dateien nur Rückwärtsreferenzen auf Objekte möglich sind, sind die Quell- und Zielports nach dem Laden nur in den Linien gespeichert. In diesem Nachbearbeitungsschritt werden auch in den Ports die Referenzen auf die Linien gespeichert.

**Aufbereiten der MemoryStore-Zugriffe:** In diesem Nachbearbeitungsschritt wird die Verbindung zwischen den DataStoreMemory-Blöcken und deren Read- und Write-Blöcken hergestellt.

**Aufbereiten der Bibliotheksinformationen:** In diesem Nachbearbeitungsschritt wird die Liste der verwendeten Bibliotheken gesammelt. Außerdem wird die Liste der ignorierten Bibliotheken um alle Bibliotheken bereinigt, die nicht in dem Modell verwendet werden. Beim Modell-Export ignorierte Bibliotheken enthalten Modellteile, die nicht zum Modell selbst gezählt werden, sondern beispielsweise als gemeinsame verwendete Modellbibliotheken in einem Projekt vorgegeben sind.

**Berechnung der Maximalen-Identitäts-Signalpfade:** In diesem Nachbearbeitungsschritt werden alle MISPs (vergleiche Abschnitt 3.2.1) eines Modells berechnet. Diese Berechnung wird einmalig nach dem Laden des Modells durchgeführt, da mehrere Metriken auf diese Information angewiesen sind und somit die Liste nur einmal berechnet werden muss.

Nach Durchführung der Nachbearbeitungsschritte kann das Modell für Analysen verwendet werden, d. h. für die Berechnung der Metriken.

### 6.3.2. Ausführen der Metriken

**Eingaben**

| | |
|---|---|
| Modell in XML-Form | Das serialisierte Simulink-Modell. |
| Tool-Reports | Reports von Tools, die Modellanalysen durchführen. |

**Ausgaben**

| | |
|---|---|
| Messwerte | Die Messwerte der Metriken des Simulink-Modells. |

Abbildung 6.6 zeigt einen Überblick über den *Model Measurer*. Der Model Measurer wertet die Metriken aus und speichert ihre Messwerte. Daher nimmt die abstrakte Klasse `Metric` eine zentrale Bedeutung ein. Sie ist die Basisklasse aller Metriken. Jede Metrik hat bestimmte Eigenschaften wie z. B. `Range`, `Priority` und `Type` (vergleiche Kapitel 3 und 4).

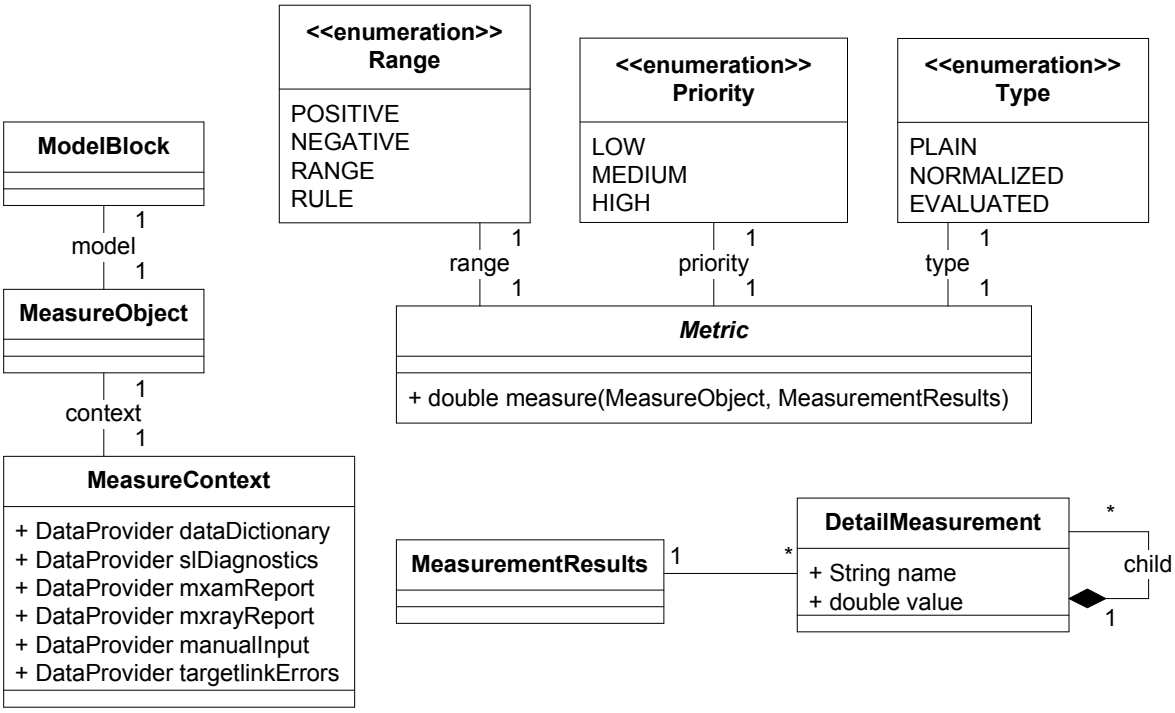



Abbildung 6.6.: Überblick über die wichtigsten Klassen des Model Measurers

Zum Ausführen einer Metrik benötigt die Methode `measure(...)` der Klasse `Metric` eine Instanz der Klasse `MeasureObject` und eine Instanz der Klasse `MeasurementResults`. Das Messobjekt enthält sowohl alle `DataProvider` als auch eine Referenz auf das geladene Modell. Die `DataProvider` dienen zur Anbindung der in Abschnitt 6.2 beschriebenen Datenquellen. Die Messergebnisse werden dann in einem Objekt der Klasse `MeasurementResults` gespeichert. Pro Metrik wird ein Objekt der Klasse `DetailMeasurement` instanziiert, das wiederum beliebig viele `DetailMeasurement`-Instanzen als Kinder haben kann. Diese hierarchische Struktur kann dazu verwendet werden, detaillierte Informationen über die betrachteten Elemente einer Metrik abzuspeichern. Abbildung A.5 im Anhang zeigt ein Beispiel, wie die Metrik »Anzahl der Modellklone« ihre Informationen als `DetailMeasurement`-Instanzen verwaltet. Listing 6.3 zeigt einen Überblick über den grundlegenden Algorithmus des Model Measurers. Es wird über alle Metriken iteriert (Zeile 5) und jede Metrik ausgeführt (Zeile 6). Als Parameter bekommt jede Metrik eine Referenz auf das Messobjekt und die Messergebnisse übergeben. Zurückgegeben wird das Messergebnis.

```
1  public class ModelMeasurer {
2      public MeasurementResults measure(MeasureObject mo, List<Metric> metrics) {
3          MeasurementResults mr = new MeasurementResults();
4
5          for(Metric metric : metrics) {
6              metric.measure(mo, mr);
7          }
8
9          return mr;
10     }
11 }
```

Listing 6.3: Model Measurer

### 6.3.3. Erzeugung der Messwertdatenbank

**Eingaben**

| | |
|---|---|
| Messwerte verschiedener Modelle | Die Messwerte der Modelle, die als Basis für die Messwertdatenbank dienen sollen. |

**Ausgaben**

| | |
|---|---|
| Messwertdatenbank | Enthält alle Messwerte der Modelle, deren Mittelwerte und erlaubten Minimal- und Maximalwerte. |

Abbildung 6.7 zeigt einen Überblick über den *Measurement Database Builder*. Der Measurement Database Builder erstellt die Datenbank, die für die Erzeugung eines Referenzmodells benötigt wird. Die zentralen Klassen sind die Klassen `MeasurementsDatabase` und `MeasurementsTable`.

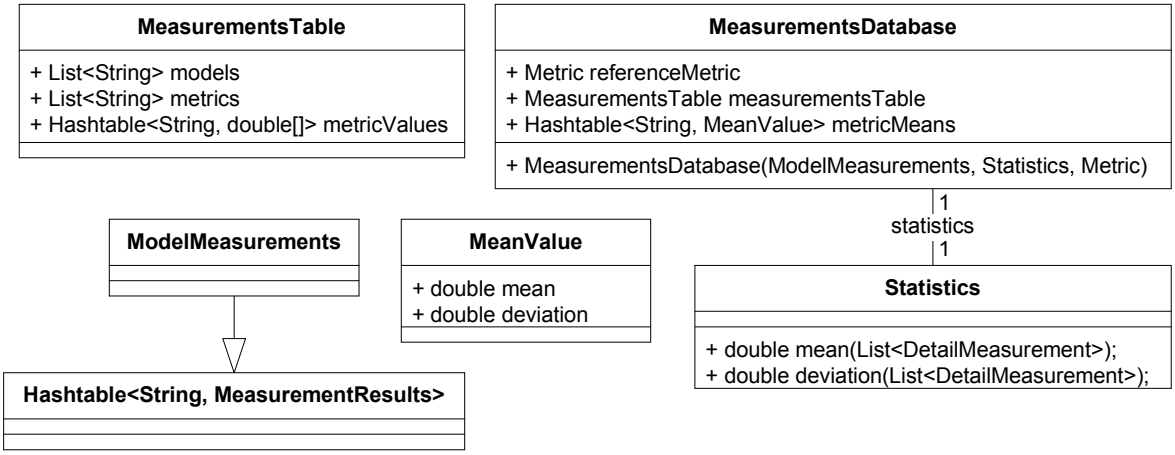



Abbildung 6.7.: Überblick über die wichtigsten Klassen des Measurement Database Builders

Die Instanzen der Klasse `MeasurementsDatabase` enthalten die Messwerte aller Modelle, deren Mittelwerte und erlaubten Minimal- und Maximalwerte. Dabei werden die Messwerte der Modelle in einer Instanz der Klasse `MeasurementsTable` verwaltet. Die Klasse `Statistics` implementiert die Berechnung des arithmetischen Mittels und der korrigierten Stichprobenvarianz. Listing 6.4 zeigt einen Überblick über den grundlegenden Algorithmus des Measurement Database Builders. Eine Messwertdatenbank wird aus einer Menge von Messwerten

erstellt (Zeile 6). Dabei werden eine Instanz der verwendeten statistischen Berechnungen und die verwendete Referenzmetrik übergeben. Zurückgegeben wird die Messwertdatenbank.

```
1  private Metric referenceMetric;
2  private Statistics statistics;
3
4  public class MeasurementDatabaseBuilder {
5      public MeasurementsDatabase build(ModelMeasurements mm) {
6          MeasurementsDatabase md = new MeasurementsDatabase(mm,
7              statistics, referenceMetric);
8          return md;
9      }
10 }
```

Listing 6.4: Measurement Database Builder

### 6.3.4. Qualitätsbewertung

**Eingaben**

| | |
|---|---|
| Messwerte | Die Messwerte des zu bewertenden Simulink-Modells. |
| Messwertdatenbank | Die Messwertdatenbank, die zur Erzeugung des Referenzmodells verwendet wird. |
| Regeln | Die zur Bewertung verwendeten Regeln. |
| Qualitätsmodell | Das Qualitätsmodell mit seinen einzelnen Aspekten. |

**Ausgaben**

| | |
|---|---|
| Modellqualitätsbewertung | Die Ergebnisse der Modellqualitätsbewertung. |

Abbildung 6.8 zeigt einen Überblick über den *Quality Rater*. Der Quality Rater nimmt die Modellqualitätsbewertung vor. Die dazu benötigten zentralen Klassen sind `ReferenceModel` und `RuleLimits`.

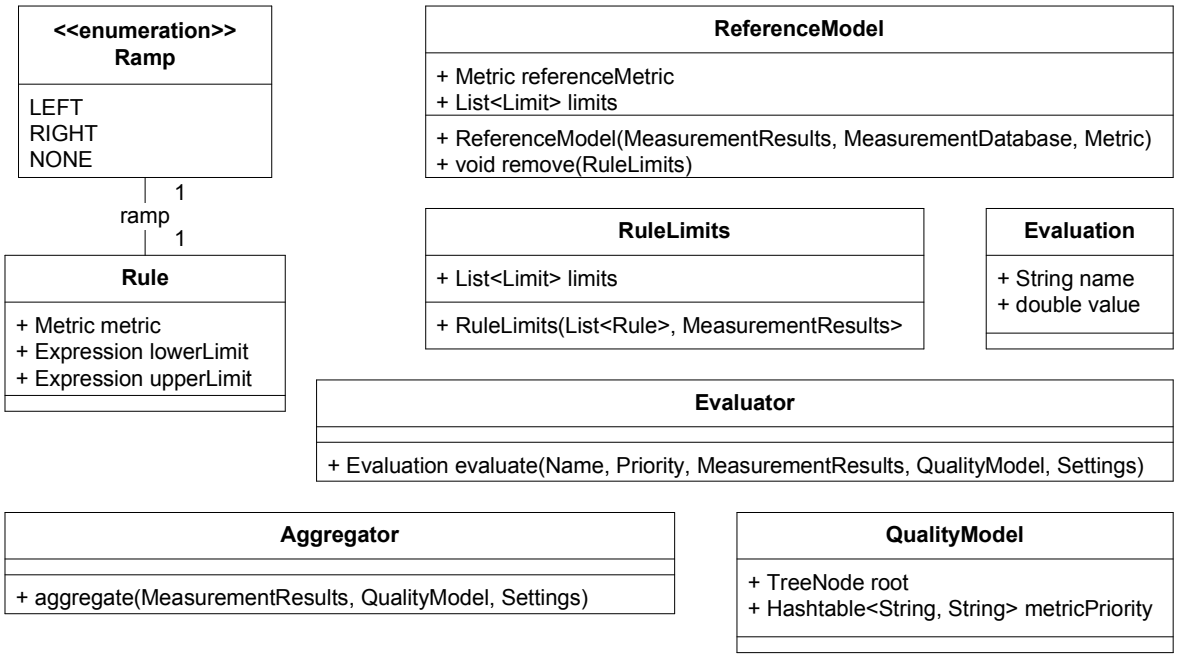



Abbildung 6.8.: Überblick über die wichtigsten Klassen des Quality Raters

Die erlaubten Bereiche des Referenzmodells werden mithilfe einer übergebenen Instanz der Klasse `MeasurementResults` aus Abbildung 6.6 und einer Instanz der Klasse `MeasurementsDatabase` aus Abbildung 6.7 erstellt. Die erlaubten Bereiche der Regeln hingegen werden aus den Regeln an sich und den Messwerten des aktuellen Modells in einer Instanz der Klasse `RuleLimits` erzeugt. Nachdem die erlaubten Bereiche entfernt wurden, die sowohl in einer Regel als auch im Referenzmodell enthalten sind, werden die verbleibenden erlaubten Bereiche ausgewertet. Dazu werden sie einer Instanz der Klasse `Evaluator` übergeben. Das Ergebnis sind Instanzen der Klasse `Evaluation`, die anschließend von einer Instanz der Klasse `Aggregator` im Qualitätsmodell aggregiert werden. Listing 6.5 zeigt einen Überblick über den grundlegenden Algorithmus des Quality Raters. Zuerst wird aus den Messwerten des zu untersuchenden Modells und der Messwertdatenbank das Referenzmodell erstellt (Zeile 15). Dann werden die Regeln geladen (Zeile 16) und alle Referenzmodellgrenzen verworfen, für die es auch Regeln gibt (Zeile 17). Regeln haben eine höhere Priorität als Referenzmodellgrenzen. Als nächstes wird über alle Grenzen iteriert (Zeile 19) und aus jeder Grenze und dem dazugehörigen Messwert eine Bewertung erstellt (Zeile 21). Nach der Erstellung der Bewertungen werden diese schließlich im Qualitätsmodell aggregiert (Zeile 24). Zurückgegeben werden die Bewertungen für die Faktoren, Kriterien und Metriken.

```
1  private Settings settings;
2  private QualityModel qm;
3  private List<Rule> rules;
4
5  private Metric referenceMetric;
6
7  private Evaluator evaluator;
8  private Aggregator aggregator;
9
10 public class QualityRater {
11     public List<Evaluation> rate(MeasurementResults mr,
12             MeasurementsDatabase md) {
13         List<Evaluation> evaluations = new List<Evaluation>();
14
15         ReferenceModel rm = new ReferenceModel(mr, md, referenceMetric);
16         RuleLimits rl = new RuleLimits(rules, mr);
17         rm.remove(rl);
18
19         for(Limit limit : new CompoundIterable<Limit>(rm, rl)) {
20             String priority = qualityModel.getPriority(limit.getName());
21             evaluations.add(evaluator.evaluate(limit, priority, mr));
22         }
23
24         aggregator.aggregator(evaluations, qualityModel, settings);
25
26         return evaluations;
27     }
28 }
```

Listing 6.5: Quality Rater

### 6.3.5. Grafische Oberfläche

**Eingaben**

| | |
|---|---|
| Messwerte | Die darzustellenden Messwerte. |
| Modellqualitätsbewertung | Die darzustellende Modellqualitätsbewertung. |

Abbildung 6.9 und 6.10 zeigen den *Quality Investigator*. Der Baum auf der linken Seite enthält in beiden Abbildungen das Qualitätsmodell. In Abbildung 6.9 werden die Messwerte in einem Kiviat-Diagramm (vergleiche Abschnitt 5.1.3) dargestellt. Abbildung 6.10 zeigt die Visualisierung der Modellqualitätsbewertung (vergleiche Abschnitt 5.2.1) im Prototyp.

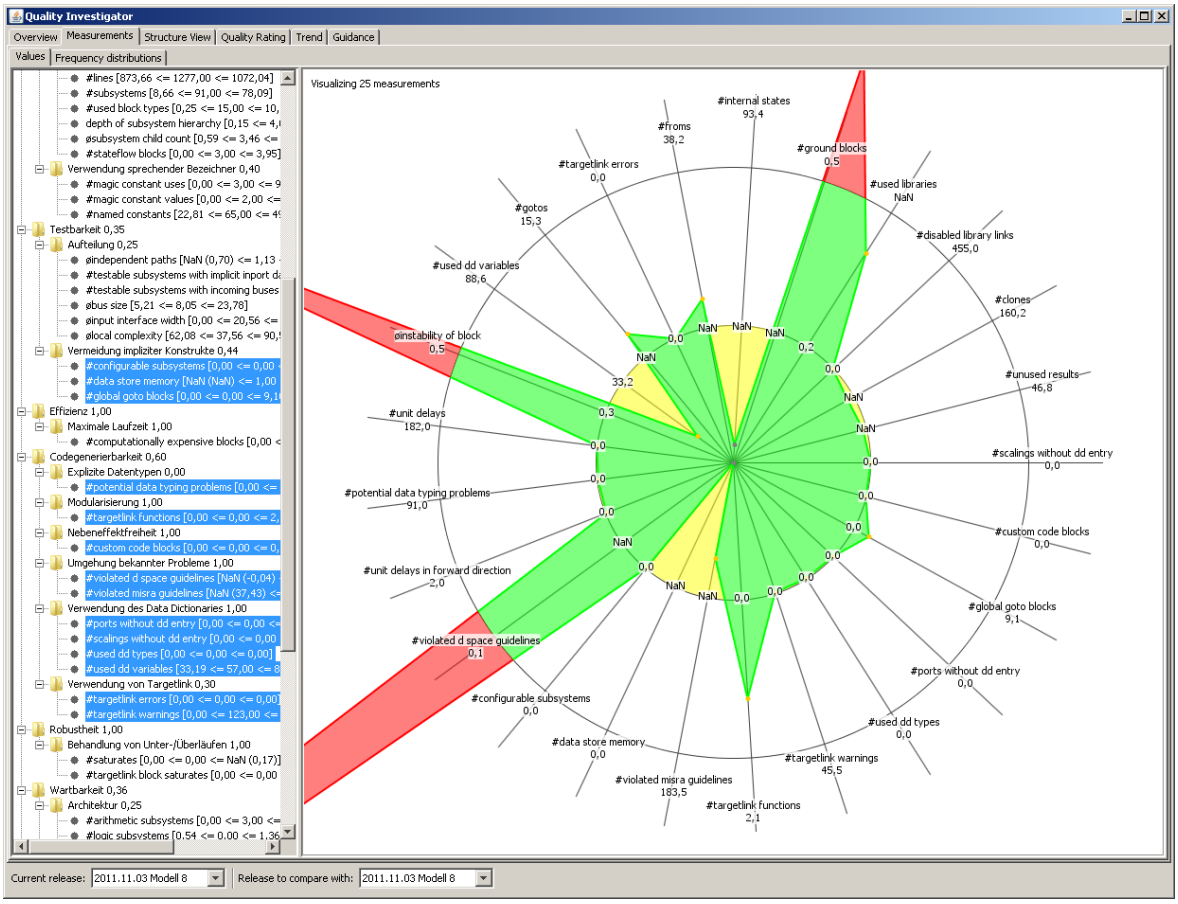



Abbildung 6.9.: Darstellung der Messwerte als Kiviat-Diagramm im Prototyp

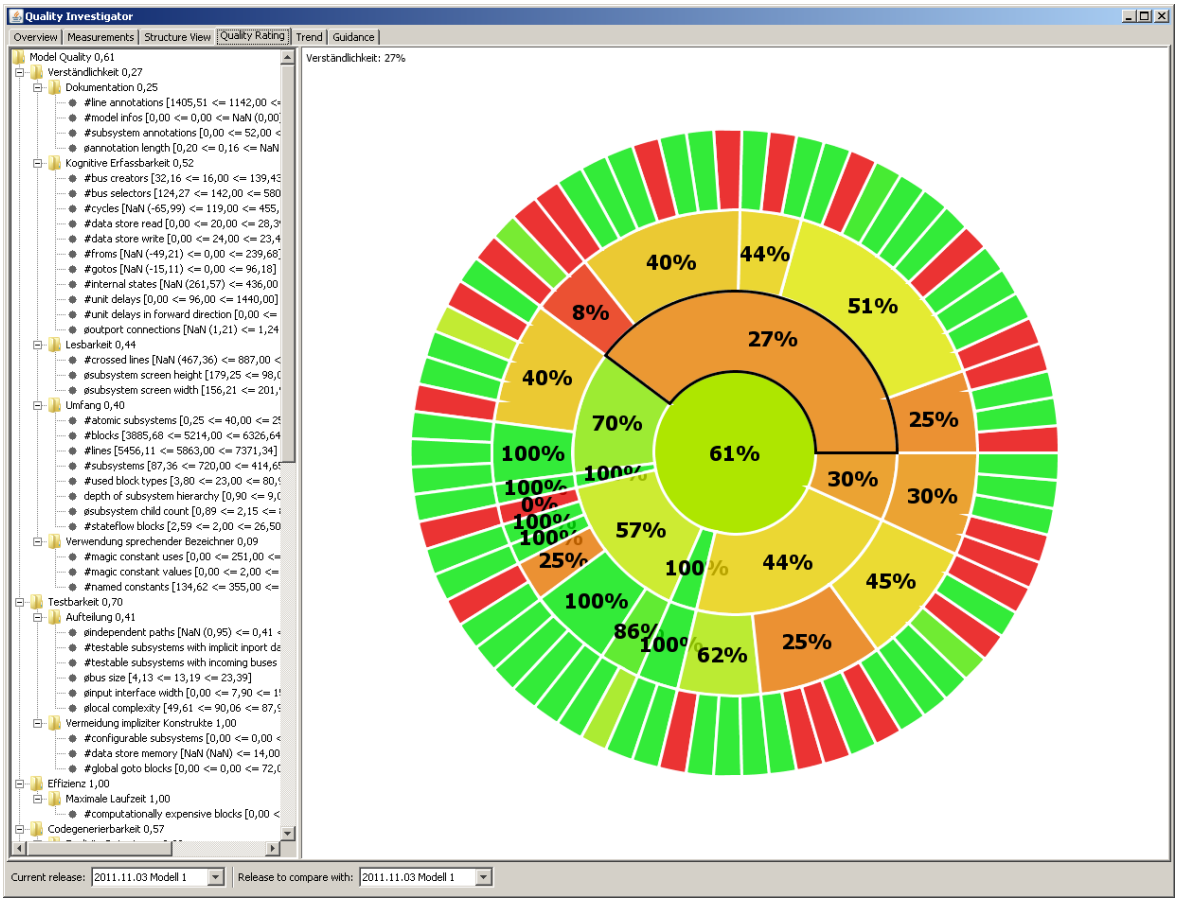



Abbildung 6.10.: Darstellung der Modellqualitätsbewertung im Prototyp

Außerdem gibt es im Prototyp beispielsweise eine Ansicht, in der die erweiterten Informationen der Metriken aus Abschnitt 6.3.2 dargestellt werden können. Diese Darstellung wird in Abbildung A.4 im Anhang gezeigt.

## 6.4. Einschränkungen

Der Prototyp implementiert alle in dieser Arbeit beschriebenen Konzepte und, bis auf wenige Ausnahmen, alle beschriebenen Metriken. Das umfasst zum einen einige Metriken, die auf die Verwendung spezifischer Tools angewiesen sind, und zum anderen Metriken, für die weitere Informationen notwendig sind.

Zwar ist die Anbindung von externen Tools im Prinzip kein Problem (siehe Abschnitt 6.2), allerdings entstehen dadurch auch einige Nachteile. Sind für eine Metrik spezifische Tools notwendig, werden weitere Lizenzen für die Tools benötigt. Auch wird die Implementierung der Metrik sehr spezifisch. Außerdem müssen die Tools oft noch pro Modell aufwendig konfiguriert werden und sind meist nicht automatisiert ausführbar. Ein Beispiel für eine Metrik, die auf ein spezifisches Tool angewiesen ist, ist die Metrik »Worst-Case-Execution-Time«. Tools für die Ermittlung der WCET benötigen häufig Annotationen im Code und müssen speziell für einen Zielprozessor konfiguriert werden.

Metriken, die auf weitere Informationen angewiesen sind, benötigen weitere Artefakte bzw. Dokumente, damit sie verwendet werden können. Ein Beispiel für eine Metrik, die auf weitere Informationen angewiesen ist, ist die Metrik »Anzahl der Architekturverletzungen«. Um die Architektur auf Verletzungen prüfen zu können, muss ein Artefakt bzw. Dokument vorhanden sein, das auf ausreichend formale Art und Weise beschreibt, wie die Architektur aussehen soll. Ist die Beschreibung zwar vorhanden, aber nicht ausreichend formalisiert, kann sie nicht automatisiert ausgewertet werden.

Eine Übersicht über die nicht implementierten Metriken findet sich im Anhang in Abschnitt A.6. Allerdings hat das Fehlen der Metriken keinen negativen Einfluss auf das Gesamtkonzept dieser Arbeit, da das Qualitätsmodell erweiterbar konzipiert ist. Das Hinzufügen und Entfernen von Faktoren, Kriterien und Metriken ist durch den Aufbau des Qualitätsmodells vorgesehen.

## 6.5. Zusammenfassung

Der Prototyp implementiert den in dieser Arbeit vorgestellten Ansatz vollständig. Dank des modularen Aufbaus ist er sehr flexibel einsetzbar. So kann z. B. der Model Measurer auf einem Server ausgeführt werden, damit regelmäßig Messungen vorgenommen werden. Der Prototyp bietet viele Konfigurationsmöglichkeiten, die in Abschnitt A.5 im Anhang aufgelistet sind. Dadurch kann er projektspezifisch angepasst werden. Dank des Metamodells kann aus Java heraus auf die Modelle zugegriffen werden. Der Prototyp bietet eine Visualisierung der Messwerte und der Qualitätsbewertung.

# 7. Evaluation

In diesem Kapitel wird die Modellqualitätsbewertung aus Kapitel 5 evaluiert. Dazu wird sie in einer Fallstudie im PKW-Bereich der Daimler AG angewendet und auf gewünschte Eigenschaften untersucht. Für die Evaluation wird der Prototyp aus Kapitel 6 verwendet. Die verwendeten Modelle und die Konfiguration des Prototyps werden in Abschnitt 7.1 beschrieben.

Die Evaluation des Verfahrens zur automatisierten Bewertung der Modellqualität wird analog zu dem von Kläs et al. vorgestellten Verfahren in [KLH11] vorgenommen. In [KLH11] wird ein Qualitätsmodell untersucht, welches Aussagen über die Qualität von Java-Anwendungen macht. Kläs et al. stellen einige Anforderungen an das Qualitätsmodell und die Qualitätsbewertung. Angepasst an die Modellqualitätsbewertung ergeben sich folgende Anforderungen:

**Nachvollziehbarkeit** Die Modellqualitätsbewertung soll nachvollziehbar sein.

**Kosteneffizienz** Die Modellqualitätsbewertung soll kosteneffizient sein.

**Diversifikation** Die Modellqualitätsbewertung soll Modelle voneinander unterscheiden können, obwohl durch die Messung von den konkreten Modellen abstrahiert wird.

**Validität** Die Modellqualitätsbewertung soll zutreffend sein, d. h. beispielsweise mit der Einschätzung von Experten übereinstimmen.

Die Nachvollziehbarkeit ist durch die Konstruktion der Modellqualitätsbewertung in Kapitel 5 gegeben. Es ist ersichtlich, wie es zu einer bestimmten Bewertung gekommen ist. Des Weiteren lässt sich die Bewertung beliebig oft mit dem gleichen Ergebnis wiederholen. Die Kosteneffizienz ist dadurch erfüllt, dass die Qualitätsbewertung automatisiert stattfindet und somit keine Benutzerinteraktionen notwendig sind. Die Diversifikation der Modellqualitätsbewertung wird in Abschnitt 7.2 untersucht. Die Validität der Modellqualitätsbewertung wird in Abschnitt 7.3 untersucht. Dazu werden Modelle mit der Qualitätsbewertung bewertet und nach ihrer Bewertung sortiert. Diese Bewertung wird dann mit der Einschätzung der Modellqualität verglichen, wie sie von Experten wahrgenommen wird.

## 7.1. Ausgangspunkt

Für die Evaluation wird der Prototyp aus Kapitel 6 verwendet. Er enthält das Qualitätsmodell aus Kapitel 4 mit den Metriken aus Kapitel 3 und führt die Qualitätsbewertung durch, wie sie in Kapitel 5 beschrieben ist. Allerdings ohne die in Abschnitt 6.4 beschriebenen Metriken. Das umfasst vor allem Metriken, die entweder auf externe, nicht automatisierbare Tools oder fehlende Daten angewiesen sind.

Die betrachtete Datenbasis besteht aus 8 Modellen aus dem PKW-Bereich. Die Modelle haben verschiedene Größen und setzen unterschiedliche Funktionen um. Sie befinden sich momentan in Wartung, d. h. sie sind im Prinzip vollständig. Es werden aber immer wieder Änderungen eingepflegt. Die Modelle sind Targetlink-Modelle und erfüllen die geforderten

Eigenschaften aus Abschnitt 2.2.5. Im Folgenden werden die Modelle *Modell 1*, *Modell 2*, ..., *Modell 8* genannt. Abbildung 7.1 und 7.2 zeigen einen Überblick über einige Messwerte der Modelle.

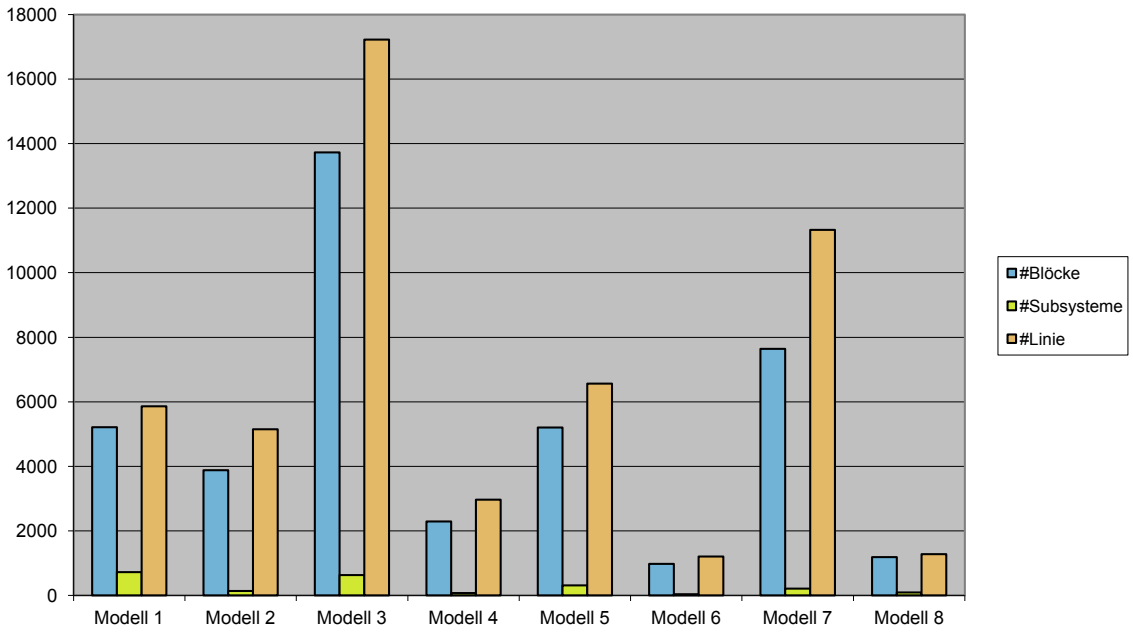



Abbildung 7.1.: Überblick über die verwendeten Modelle

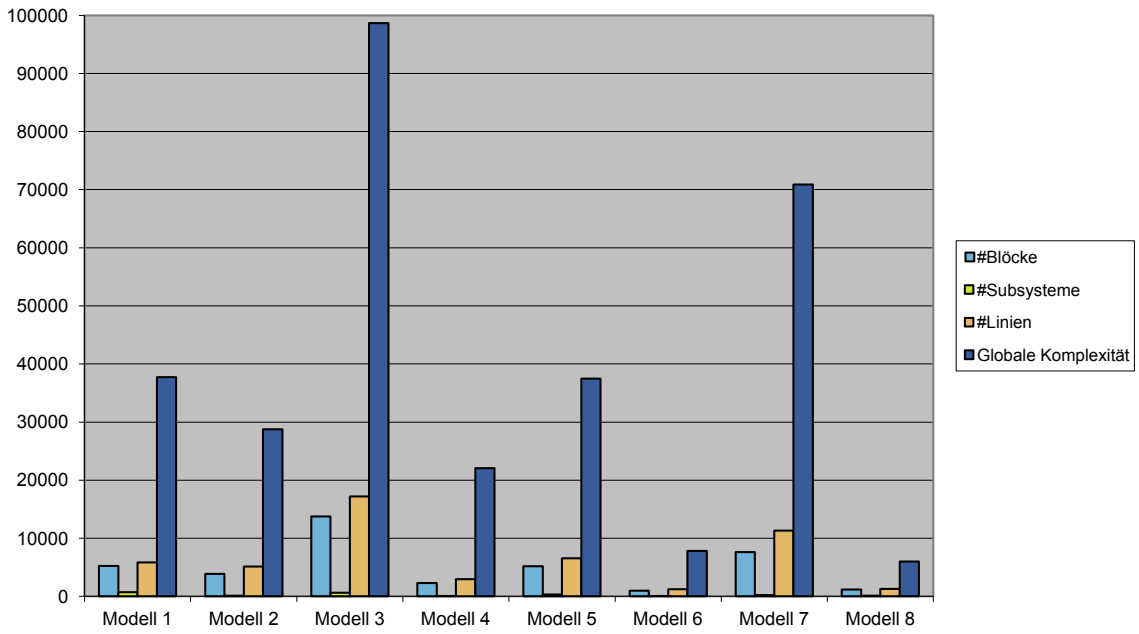



Abbildung 7.2.: Überblick über die Komplexität der Modelle

Zusätzlich zu den Anzahlen der Simulink-Konstrukte wird in Abbildung 7.2 noch die »Höhe

der globalen Komplexität« dargestellt, die für die Erstellung des Referenz-Modells verwendet wird. Da die Messwerte der globalen Komplexität um einige Größenordnungen größer sind als z. B. die Anzahlen der Blöcke, werden in der ersten Abbildung nur die Anzahlen der Simulink-Konstrukte dargestellt.

Abbildung 7.3 listet die verwendeten Parameter der Modellqualitätsbewertung (vergleiche Kapitel 5 und Abschnitt A.5 im Anhang) auf. Der Parameter *useMetricPriorities* sorgt dafür, dass die Prioritäten der Metriken berücksichtigt werden, da ihre Nichterfüllung unterschiedlich schwerwiegend ist. Das Berücksichtigen der Prioritäten wird durch unterschiedliche Gewichtung der einzelnen Metrikbewertungen erreicht. Die verwendeten Gewichtungen der Metriken sind in Abschnitt A.3 im Anhang aufgelistet. Ein *aggregationThreshold* von 15% hat sich in der Praxis bewährt. Der Schwellwert ist weder zu hoch noch zu niedrig: Wäre er zu hoch, dann würden die Gesamtbewertungen alle sehr schlecht ausfallen. Wäre er hingegen zu niedrig, würde das dazu führen, dass praktisch alle zu aggregierenden Bewertungen über dem Schwellwert sein würden und somit nie abgewertet werden würden. Der Parameter *referenceMetric* schließlich legt die Metrik »Höhe der globalen Komplexität« als Metrik für die Normierung im Referenzmodell fest (vergleiche Abschnitt 5.1.1).

```
useMetricPriorities = true
aggregationThreshold = 15
referenceMetric = GlobalComplexity
```

Abbildung 7.3.: Gewählte Parameter der Modellqualitätsbewertung

Analog zu [KLH11] wird davon ausgegangen, dass die Qualität der Modelle normalverteilt ist. Das bedeutet, die meisten Modelle sind von mittlerer Qualität und nur einige sind besonders gut oder besonders schlecht. Diese Annahme deckt sich mit den Erfahrungen aus der Praxis.

Tabelle 7.1 zeigt die Ergebnisse der Modellqualitätsbewertung absteigend sortiert nach der Gesamtbewertung. In der untersten Zeile sind die Gesamtbewertungen aufgelistet und in den Zeilen darüber die Bewertungen für die einzelnen Faktoren.

| | **Modell 5** | **Modell 3** | **Modell 2** | **Modell 1** | **Modell 6** | **Modell 4** | **Modell 8** | **Modell 7** |
|---|---|---|---|---|---|---|---|---|
| Codegenerierbarkeit | 60% | 49% | 33% | 57% | 60% | 60% | 60% | 69% |
| Effizienz | 81% | 93% | 74% | 100% | 100% | 100% | 100% | 100% |
| Korrektheit | 30% | 15% | 30% | 30% | 30% | 30% | 30% | 0% |
| Robustheit | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| Testbarkeit | 71% | 62% | 43% | 70% | 43% | 33% | 35% | 14% |
| Verständlichkeit | 63% | 68% | 57% | 27% | 34% | 33% | 43% | 22% |
| Wartbarkeit | 59% | 62% | 97% | 44% | 50% | 52% | 36% | 35% |
| Gesamt | 66,1% | 63,9% | 62,0% | 61,3% | 59,5% | 58,2% | 57,6% | 34,6% |

Tabelle 7.1.: Ergebnisse der Modellqualitätsbewertung

Bei den Bewertungen der Korrektheit aus Tabelle 7.1 fällt auf, dass sie sehr niedrig ausfallen. Für Modell 7 ist sie sogar 0%. Das ist dadurch zu erklären, dass der Faktor nur Kriterien

und Metriken enthält, die Untersuchungen auf dem Modell durchführen. So liefert beispielsweise die Metrik »Anzahl der verfallenen Ergebnisse« zwar wichtige Informationen über den Signalfluss eines Modells zurück, kann aber keine Aussagen über den Prozentsatz der umgesetzten Anforderungen machen. Und da, wie in Abschnitt 6.4 beschrieben, der Prozentsatz der umgesetzten Anforderungen fehlt, relativiert sich die schlechte Bewertung bezüglich der Korrektheit.

Auch liegen alle Bewertungen der Codegenerierbarkeit unter 69%. Das bedeutet aber nicht, dass kein Code generierbar ist. Vielmehr sagen diese Bewertungen aus, dass nicht alle Kriterien erfüllt sind, um jegliche mögliche Probleme bei der Codegenerierung im Vorhinein auszuschließen. Aus allen untersuchten Modellen kann Code generiert werden. Sobald allerdings die Bewertung des Faktors Codegenerierbarkeit kleiner wird, erhöht sich die Anzahl der möglichen Probleme. So können beispielsweise kleine Änderungen am Modell zu großen Problemen mit den Datentypen bei der Generierung des Codes führen.

## 7.2. Diversifikation

In diesem Abschnitt wird die Diversifikation der Modellqualitätsbewertung untersucht. Die Diversifikation ist ein Maß dafür, wie unterschiedlich die Modellqualitätsbewertungen für unterschiedliche Simulink-Modelle ausfallen. Diese Eigenschaft ist wichtig, da durch die Modellqualitätsbewertung von der Wirklichkeit abstrahiert wird. Wird zu stark abstrahiert, d. h. werden zu viele spezifische Details der Modelle nicht beachtet, verliert die Modellqualitätsbewertung an Aussagekraft, da dann alle Modelle gleich bewertet würden.

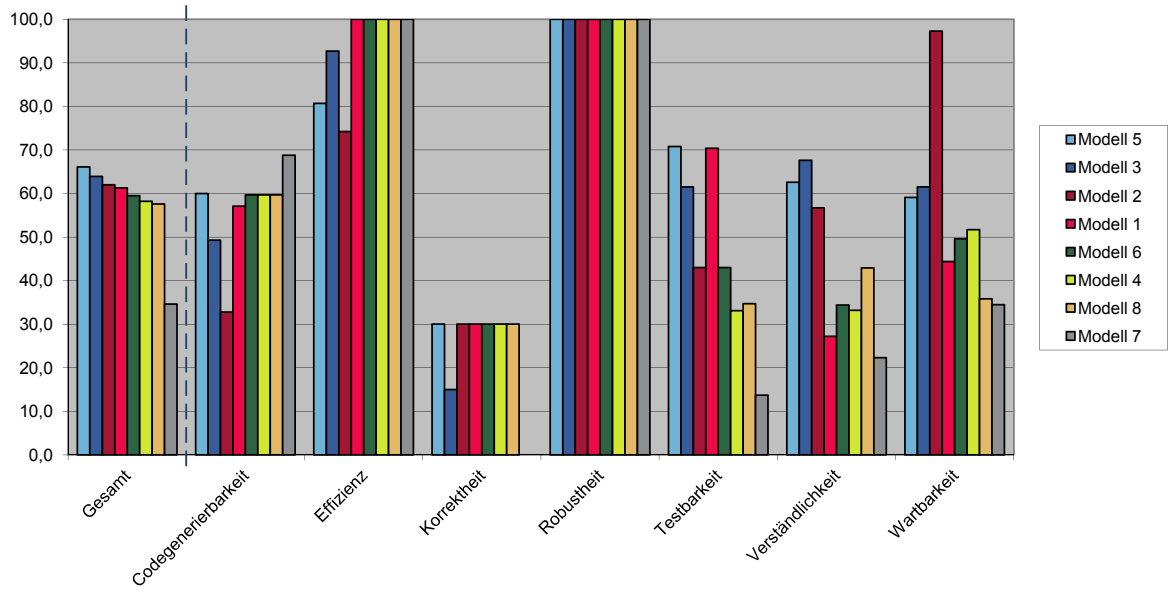



Abbildung 7.4.: Modellqualitätsbewertungen

Abbildung 7.4 zeigt einen Überblick über die Gesamtbewertungen und die einzelnen Bewertungen der Faktoren der Modelle. Die Gesamtbewertungen stimmen weitgehend mit der Annahme der Normalverteilung überein. Es gibt viele Modelle mit einer mittleren Gesamtbewertung und ein Modell mit einer relativ schlechten Bewertung. Es fehlt ein sehr gutes

Modell, allerdings kann das auf die Größe der Stichproben zurückgeführt werden. Obwohl Modell 4 und Modell 8 eine annähernd gleiche Gesamtbewertung haben, sind die Bewertungen einiger ihrer Faktoren unterschiedlich. Das heißt, die identischen Gesamtbewertungen können auf Ebene der Faktoren wieder unterschieden werden.

Abbildung 7.5 zeigt die Bewertungen in einem Boxplot (vergleiche [MTL78]). Ein Boxplot visualisiert verschiedene statistische Streuungs- und Lagemaße unimodal verteilter Daten. Durch die Annahme der Normalverteilung der Modellqualität kann ein Boxplot zur Visualisierung verwendet werden. Der horizontale Strich in den jeweiligen Boxen stellt den Median dar. Die Box beginnt beim ersten Quantil, endet beim dritten Quantil und umfasst somit den Bereich, in dem 50% der Daten liegen. Somit stellt die Größe der Box den Interquartilsabstand dar. Der durch eine vertikale Linie verbundene horizontale Strich über und unter der Box endet an den Minimal- bzw. Maximalwerten. Liegen ein oder mehrere Werte mehr als das anderthalbfache des Interquartilsabstands vom Median entfernt, werden sie nicht als Minimal- oder Maximalwert betrachtet, sondern als Ausreißer. Ausreißer werden als farbige Kreuze dargestellt.

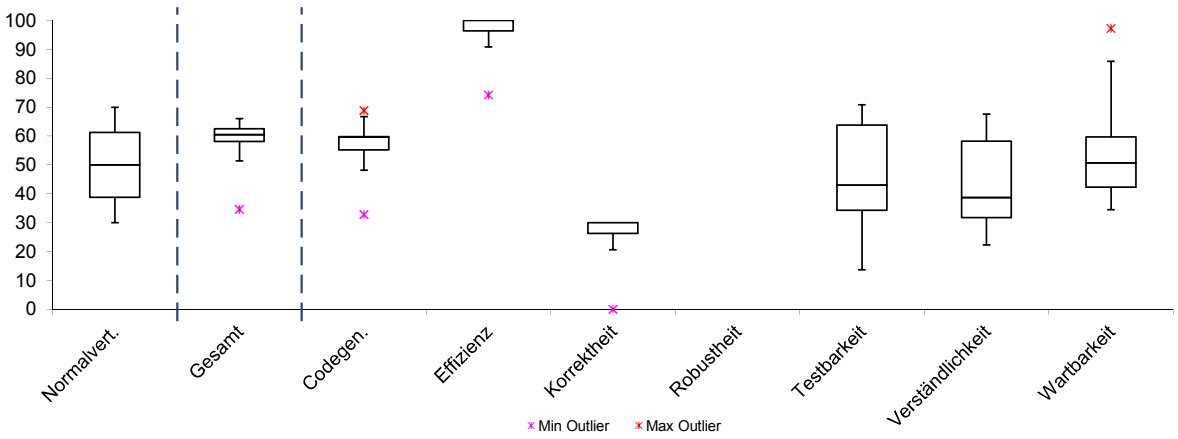



Abbildung 7.5.: Boxplot der Modellqualitätsbewertungen

Zusätzlich zu den Bewertungen der Modelle werden in der ersten Spalte des Boxplots zum Vergleich generierte Bewertungen gezeigt, die normalverteilt sind. Es ist im Boxplot gut ersichtlich, welche Faktoren der Modellqualitätsbewertung vornehmlich zur Diversifikation der Modellqualitätsbewertung beitragen. So zeigt der Faktor Wartbarkeit beispielsweise eine sehr gute Verteilung. Wohingegen der Faktor Robustheit nichts zur Diversifikation beiträgt, da er für alle Modelle mit 100% bewertet wird. Allerdings bedeutet kein Beitrag zur Diversifikation nicht, dass der Faktor keine Aussagekraft hat. Wenn die Kriterien und Metriken des Faktors korrekt sind, wird trotzdem geprüft, ob die Modelle die gewünschten Qualitätseigenschaften haben. Nur wenn die Kriterien und Metriken nicht die gewünschten Eigenschaften prüfen, ist sowohl der Beitrag zur Diversifikation als auch die Aussagekraft des Faktors hinfällig.

An dieser Stelle muss eine Einschränkung bei der Evaluation der Modellqualitätsbewertung gegenüber dem Vorgehen in [KLH11] getroffen werden. Kläs et al. zeigen die Diversifikation über die Berechnung der Entropie ihrer Faktoren. Die Entropie der Faktoren vergleichen sie dann mit der Entropie normalverteilter Ergebnisse. Durch diesen Vergleich treffen sie eine Aussage, wie unterschiedlich die Bewertungen ihrer Faktoren sind. Aus sehr unterschiedlichen Faktoren kann dann auf eine hohe Diversifikation geschlossen werden. Im Ansatz der

hier vorliegenden Arbeit ist die Berechnung der Entropie auf diese Art und Weise nicht möglich, da die Anzahl der Skalenlevel (1001) sehr viel größer als die Anzahl der betrachteten Modelle (8) ist. Im Gegensatz dazu kann in [KLH11] die Entropie berechnet werden, da für die Bewertung Schulnoten (6 Skalenlevel) verwendet werden und 13 Softwareprojekte an der Evaluation beteiligt sind. Daher wird in dieser Arbeit die Auswertung auf eine grafische Auswertung von Abbildung 7.4 und des Boxplots in Abbildung 7.5 beschränkt. Die Gesamtbewertungen sind annähernd normalverteilt und die Gesamtbewertungen, die sehr nahe beieinander liegen, haben trotzdem unterschiedlich bewertete Faktoren. Deshalb kann auf eine ausreichende Diversifikation geschlossen werden.

Erwartungsgemäß tragen Faktoren mit vielen Metriken und Kriterien stärker zur Diversifikation bei als Faktoren mit wenigen Metriken und Kriterien. Allerdings ändert das nichts an der Aussagekraft der Faktoren. Ein Faktor, dem nur ein Kriterium mit einer Metrik zugeordnet ist, kann nur den Wert der Metrik (z. B. 0% oder 100%) annehmen, ist aber dennoch ein Indikator für das Vorhandensein eines gewünschten Qualitätskriteriums. Bekommt ein Faktor oder Kriterium immer nur ein und dieselbe Bewertung, ist zu prüfen, ob entweder alle betrachteten Modelle die gewünschten Eigenschaften haben oder die Metriken falsch formuliert sind.

## 7.3. Validität

In diesem Abschnitt wird die Validität der Modellqualitätsbewertung untersucht. Dazu wurde die Qualität der Modelle von Experten eingeschätzt. Tabelle 7.2 zeigt einen Überblick der Einschätzungen, die das Ergebnis von Interviews mit Experten sind.

| Name | Qualität | Historie | Besonderheiten |
|---|---|---|---|
| Modell 3 | sehr gut | ging durch einige Hände | ist sehr groß, gute Aufteilung der Busse |
| Modell 1 | gut | wurde primär von einem Modellierer gepflegt | gute Aufteilung der Busse |
| Modell 2 | gut | wurde primär von einem Modellierer gepflegt | keine Verwendung von Stateflow, gleichartiger Modellierungsstil im ganzen Modell |
| Modell 5 | mittel | ging durch viele Hände | Wiederverwendung des Modells als linker und rechter Teil |
| Modell 4 | mittel | ging durch viele Hände, ist alt | viele Stateflow-Diagramme |
| Modell 6 | mittel | ging durch viele Hände, ist alt | Stateflow-Diagramme mit hohem Kontrollflussanteil |
| Modell 8 | mittel | ist relativ neu | verwendet große Anteile aus Bibliotheken |
| Modell 7 | mittel/schlecht | hat viele Änderungen erfahren | hat sehr viele Parameter |

Tabelle 7.2.: Modelle und Einschätzung der Modellqualität

Obwohl einige Modelle eine identische Qualitätseinschätzung bekommen haben, berücksichtigt die Reihenfolge der Modelle die relative Qualität zueinander. Das bedeutet, wenn zwei

Modelle als *gut* eingeschätzt wurden, ist das höher in der Liste geführte Modell relativ besser eingeschätzt worden als das tiefer in der Liste geführte Modell. Die Qualitätseinschätzung wird in Abschnitt 7.3.1 für einen Vergleich mit der Modellqualitätsbewertung verwendet. Zusätzlich zu der Qualitätseinschätzung enthält die Tabelle Informationen über die Historie und die Besonderheiten der Modelle. In Abschnitt 7.3.2 wird geprüft, ob die Modellqualitätsbewertung die Besonderheiten der Modelle erkannt und berücksichtigt hat. Die Einschätzung der Modellqualität wurde nicht auf Ebene der Faktoren durchgeführt, da eine intuitive Einschätzung gewünscht war. Dadurch kann sowohl die relative Einschätzung der Experten als auch die Gründe für diese Einschätzung mit der Modellqualitätsbewertung abgeglichen werden.

### 7.3.1. Aggregierte Bewertungen

Tabelle 7.3 zeigt die Rangfolgen, die durch die Modellqualitätsbewertung $rang(x_{mqb})$ und die Experteneinschätzungen $rang(y_{exp})$ entstanden sind. Die Rangdifferenz $d_i$ und die quadrierte Differenz $d_i^2$ werden für die Berechnung des Rangkorrelationskoeffizienten nach Spearman (vergleiche [Spe04]) benötigt.

| **Modell** | $\boldsymbol{rang(x_{mqb})}$ | $\boldsymbol{rang(y_{exp})}$ | $\boldsymbol{d_i}$ | $\boldsymbol{d_i^2}$ |
|---|---|---|---|---|
| Modell 2 | 3 | 3 | 0 | 0 |
| Modell 3 | 2 | 1 | 1 | 1 |
| Modell 1 | 4 | 2 | 2 | 4 |
| Modell 5 | 1 | 4 | -3 | 9 |
| Modell 4 | 6 | 5 | 1 | 1 |
| Modell 6 | 5 | 6 | -1 | 1 |
| Modell 8 | 7 | 7 | 0 | 0 |
| Modell 7 | 8 | 8 | 0 | 0 |

Tabelle 7.3.: Rangfolgen der Modellqualitätsbewertung und der Experten

Der Rangkorrelationskoeffizient gibt an, wie hoch die Übereinstimmung zweier Rangfolgen ist. Spearmans Rho kann einen Wert zwischen $-1$ und 1 annehmen (siehe Abbildung 7.6).

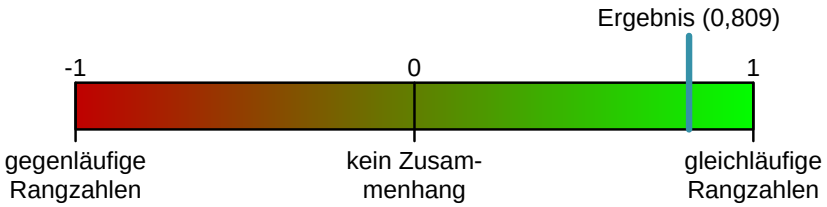



Abbildung 7.6.: Korrelation der zwei Rangfolgen

Da der Grad der Übereinstimmung geprüft werden soll, lautet die Alternativhypothese $H_A$: Es besteht eine positive Korrelation zwischen der Rangfolge der Modellqualitätsbewertung und der Rangfolge der Expertenmeinungen. Somit muss $r_s(rang(y_{exp}), rang(y_{exp})) > 0$ gelten. Im Umkehrschluss lautet die Nullhypothese $H_0$: $r_s(rang(y_{exp}), rang(y_{exp})) \leq 0$.

$$r_s = 1 - \frac{6 \sum_i d_i^2}{n \cdot (n^2 - 1)} \tag{7.1}$$

Mit den Werten aus Tabelle 7.3 und $n = 8$ ergibt sich aus Formel 7.1 der Wert $r_s = 0,80952$. Dieses Ergebnis stellt eine starke Korrelation dar. Für die Absicherung gegen null stellt Sachs in [Sac84, S. 310] für $n \leq 30$ eine Tafel für den Wert von $r_s$ bereit. Tabelle 7.4 zeigt den Ausschnitt der Tafel für $n = 8$.

| **Signifikanzniveau $\alpha$** | $0,001$ | $0,005$ | $0,010$ | $0,025$ | $0,050$ | $0,100$ |
|---|---|---|---|---|---|---|
| **Testgröße $r_s$** | $0,9286$ | $0,8571$ | $0,8095$ | $0,6905$ | $0,5952$ | $0,4762$ |

Tabelle 7.4.: Signifikanzniveau von Spearmans Rangkorrelationskoeffizient für $n = 8$

Nach Tabelle 7.4 ergibt sich für $r_s = 0,80952$ ein Signifikanzniveau von $\alpha = 0,010$. Somit kann die Alternativhypothese $H_A$ mit einer Konfidenz von 99% angenommen werden.

Bei Modell 3, Modell 4 und Modell 6 unterscheiden sich die Einschätzungen der Experten und der Modellqualitätsbewertung nur um einen Rang. Modell 1 hingegen wird von der Modellqualitätsbewertung im Vergleich zu den Experten schlechter und Modell 5 besser bewertet.

Modell 1 wird von der Modellqualitätsbewertung schlechter bewertet, da auf der einen Seite eine semantisch gute Aufteilung der Busse nicht erkannt werden kann. Auf der anderen Seite enthält das Modell überdurchschnittlich viele Konstanten-Blöcke und verhältnismäßig wenige Linien-Annotationen. Zusätzlich enthält das Modell einige DataStoreMemory-Blöcke, auf die nie lesend zugegriffen wird. Im Falle von Modell 1, wie auch von Modell 7, werden jedoch DataStoreMemory-Blöcke für den Zugriff auf das EEPROM des Steuergeräts verwendet. Das müsste bei der Modellqualitätsbewertung berücksichtigt werden, wozu allerdings individuelle Einstellungen pro Modell notwendig wären. Um eine bessere Vergleichbarkeit der Ergebnisse zu erreichen, wurde darauf in der Evaluation verzichtet.

Modell 5 hingegen wurde von der Modellqualitätsbewertung besser bewertet als von den Experten, da keine offensichtlichen Probleme erkannt wurden. Der Umstand, dass Modell 5 alt ist und durch viele Hände gegangen ist, kann durch die Modellqualitätsbewertung nicht erkannt werden.

### 7.3.2. Besonderheiten der Modelle

In diesem Abschnitt wird untersucht, ob die Modellqualitätsbewertung die von den Modellierern beschriebenen Besonderheiten der Modelle aus Tabelle 7.2 berücksichtigt hat. Dazu werden für jedes Modell die relevanten Metriken aufgelistet und ihre Bewertungen betrachtet. Für die Metriken wird der Messwert für das jeweilige Modell, das erlaubte Minimum und das erlaubte Maximum in der Form $[minimum \leq messwert \leq maximum]$ notiert. Die erlaubten Minimal- und Maximalwerte stammen entweder aus dem verwendeten Referenzmodell oder wurden aus einer Regel abgeleitet (vergleiche Abschnitt 5.1). Darf ein Messwert einer Metrik beliebig groß werden, ohne einen negativen Einfluss auf sein Kriterium zu haben, wird eine Notation der Form $[minimum \leq messwert \leq \infty\ (maximum)]$ verwendet. Der erlaubte Maximalwert wird in Klammer angegeben, um eine Aussage treffen zu können, wie der Messwert

der betrachteten Metrik im Vergleich zu den Messwerten der Modelle steht, die im Referenzmodell enthalten sind. Bei Messwerten, die kleiner als das erlaubte Minimum sind, wird das erste Kleinerzeichen durchgestrichen. Analog dazu wird bei Messwerten, die größer als das erlaubte Maximum sind, das zweite Kleinerzeichen durchgestrichen.

#### Modell 1

**Besonderheit:** gute Aufteilung der Busse
Durchschnittliche Busgröße $[4,13 \leq 13,19 \leq 23,39] \rightarrow$ die Metrik ist im erlaubten Bereich, d. h. die Busse scheinen gut aufgeteilt zu sein.

#### Modell 2

**Besonderheit:** keine Verwendung von Stateflow
Anzahl der Stateflow-Blöcke $[2,51 \not\leq 0,00 \leq 20,08] \rightarrow$ die Metrik ist nicht im erlaubten Bereich, da das Modell ungewöhnlich wenige Stateflow-Blöcke enthält. Das stellt u. U. kein Problem dar, wenn für die geforderte Funktionalität keine expliziten Zustände benötigt werden. Allerdings werden in manchen Fällen anstelle von Stateflow-Blöcken FlipFlop-Blöcke verwendet. Jeder FlipFlop-Block speichert ein Bit und die Zustandsinformationen werden somit in den Flipflops kodiert. Diese Art der Modellierung ist jedoch sehr viel schwieriger nachzuvollziehen, da im Gegensatz zu Zustandsautomaten nicht direkt ersichtlich ist, welche Zustände es gibt und was die Bedingungen für einen Zustandsübergang sind. Für die Erkennung der FlipFlop-Blöcke wäre eine Metrik notwendig, die die Anzahl der verwendeten FlipFlop-Blöcke in einem Modell berechnet.

**Besonderheit:** gleichartiger Modellierungsstil
Momentan gibt es keine Metrik, die einen gleichartigen Modellierungsstil in einem Modell identifizieren kann. Eine solche Metrik ist auch nicht ohne Weiteres möglich, da die Erkennung eines verwendeten Modellierungsstils schwer maschinell zu realisieren ist.

#### Modell 3

**Besonderheit:** ist sehr groß

- Anzahl der Blöcke $[10.147 \leq 13.729 \leq 16.526] \rightarrow$ die Metrik ist im erlaubten Bereich, das Modell wird allerdings nur als mittelgroß angesehen.
- Die Metrik »Höhe der globalen Komplexität« wird als Referenzmetrik verwendet, daher sind nur die absoluten Werte vergleichbar. In Abbildung 7.2 ist ersichtlich, dass Modell 3 den größten Messwert hat.

**Besonderheit:** gute Aufteilung der Busse
Durchschnittliche Busgröße $[4,25 \leq 12,45 \leq 23,49] \rightarrow$ die Metrik ist im erlaubten Bereich, d. h. die Busse scheinen gut aufgeteilt zu sein.

#### Modell 4

**Besonderheit:** viele Stateflow-Diagramme
Anzahl der Stateflow-Blöcke $[0,0 \leq 14,00 \not\leq 13,67] \rightarrow$ die Metrik ist etwas außerhalb des erlaubten Bereichs, d. h. die hohe Anzahl an Stateflow-Diagrammen wurde erkannt.

#### Modell 5

**Besonderheit:** Wiederverwendung des Modells als linker und rechter Teil
Die Wiederverwendung ist mithilfe der Metriken nicht erkennbar, da für die Bewertung der Modellqualität nur ein Teil verwendet wurde. Würden beide Teile verwendet werden, so würde z. B. die Anzahl der Klone drastisch ansteigen, ohne dass eine wirkliche Dopplung von Modellteilen vorliegt.

#### Modell 6

**Besonderheit:** Stateflow-Diagramme mit hohem Kontrollflussanteil
Momentan gibt es keine Metrik, die berechnet, wie groß der Kontrollflussanteil in Stateflow-Diagrammen ist. Allerdings wäre eine Metrik denkbar, die den Anteil an Zuständen dem Anteil an Kontrollflusskonstrukten gegenüberstellt.

#### Modell 7

**Besonderheit:** hat sehr viele Parameter

- Anzahl der Konstanten mit Namen $[252,31 \leq 545,00 \leq 659,16] \rightarrow$ die Metrik ist im erlaubten Bereich, aber im Vergleich zu den anderen Modellen werden überdurchschnittlich viele Konstanten mit Namen verwendet.
- Durchschnittliche Breite der Eingangsschnittstelle $[0,0 \leq 228,98 \not\leq 81,33] \rightarrow$ die Metrik ist nicht im erlaubten Bereich, die durchschnittliche Eingangsschnittstelle ist sehr viel breiter als bei den anderen Modellen, d. h. schätzungsweise werden die Parameter als Eingangssignale modelliert.

#### Modell 8

**Besonderheit:** verwendet große Anteile aus Bibliotheken

- Anzahl der verwendeten Bibliotheksverknüpfungen $[85,90 \leq 321,00 \leq \infty\ (165,14)]$ $\rightarrow$ die Metrik ist im erlaubten Bereich und in dem Modell werden sehr viel mehr Bibliotheksblöcke als bei den anderen Modellen verwendet.
- Anzahl der verwendeten Bibliotheken $[0,19 \leq 3,00 \leq \infty\ (1,34)] \rightarrow$ die Metrik ist im erlaubten Bereich und in dem Modell werden mehr Bibliotheken als bei den anderen Modellen verwendet.

## 7.4. Zusammenfassung

Die Evaluation des Ansatzes war mit den betrachteten Modellen erfolgreich. Der Vergleich der Modellqualitätsbewertung mit den Einschätzungen der Experten in Abschnitt 7.3 hat eine hohe Übereinstimmung gezeigt. Die von den Experten beschriebenen Besonderheiten der einzelnen Modelle wurden meist erkannt und können theoretisch alle zukünftig durch einige weitere Metriken erkannt werden (vergleiche Abschnitt 7.3.2). Allerdings wurde bei der Evaluation ein konkretes Qualitätsmodell evaluiert. Sobald Änderungen an dem Qualitätsmodell vorgenommen werden, müssen die gewünschten Eigenschaften erneut geprüft werden. Außerdem waren nur 8 Modelle an der Evaluation beteiligt. Um Aussagen über die Allgemeingültigkeit zu erhalten, sind weitere empirische Studien notwendig. Die Evaluation hat außerdem gezeigt, dass der Prototyp als Forschungsinstrument tauglich ist.

# 8. Zusammenfassung und Ausblick

## 8.1. Zusammenfassung

Die wachsende Verbreitung der modellbasierten Softwareentwicklung in der Automobil-Domäne und der steigende Umfang der Modelle machen automatisierte Ansätze zur Qualitätssicherung der verwendeten Modelle notwendig. Große Modelle haben in der Automobil-Domäne typischerweise ca. 15.000 Blöcke, 700 Subsysteme und eine Subsystemhierarchie mit 16 Ebenen. Die Modelle stellen bei der modellbasierten Entwicklung das zentrale Artefakt dar. Das bedeutet, dass die Modellqualität direkten Einfluss auf die Softwarequalität hat. Die höhere Abstraktion unterstützt die Software-Entwicklung zwar vor allem bezüglich nicht-funktionaler Qualitätsmerkmale, weil Strukturiertheit, einheitliche Architektur, Wiederverwendbarkeit, Lesbarkeit und Übersichtlichkeit durch die grafische Notation gefördert werden. Allerdings gibt die grafische Notation allein noch keine Garantie dafür, dass hohe Softwarequalität erreicht wird. So kann beispielsweise trotz grafischer Repräsentation eine ungünstige Architektur gewählt werden, Subsysteme falsch aufgeteilt werden oder Blöcke gewählt werden, die bei der Codegenerierung zu ineffizientem Code führen.

Ziel dieser Arbeit ist es, einen Ansatz zur automatisierten Qualitätsbewertung dieser Modelle zu entwickeln. Die Grundlage für die Definition der Modellqualität stellt ein Qualitätsmodell dar. Dieses Qualitätsmodell besteht aus verschiedenen Qualitätsaspekten und kann dadurch projektspezifisch angepasst werden. Das entwickelte Qualitätsmodell baut auf existierenden Modellen für die herkömmliche handcodierte Softwareentwicklung auf und wurde in dieser Arbeit um Teile erweitert, die für die modellbasierte Entwicklung spezifisch sind. Das Qualitätsmodell betrachtet nicht nur die Simulink-Modelle selbst, sondern auch andere im modellbasierten Entwicklungsprozess beteiligten Artefakte. Das Qualitätsmodell hat die Struktur eines Baums und enthält auf unterster Ebene Metriken. Diese Metriken werden auf die Simulink-Modelle und die anderen relevanten Artefakte angewendet. Ein Metamodell für Simulink-Modelle ermöglicht den Metriken Zugriff auf die Modelle. Um zu einer Bewertung der Modellqualität zu gelangen, werden die Messwerte der Metriken bewertet und schließlich in dem Qualitätsmodell aggregiert. Die Aggregation verdichtet die Bewertungen, um einen einfachen Überblick zu erlauben. Die Bewertung wird mithilfe der beiden folgenden Ansätze vorgenommen. Ein Referenzmodell definiert, wie ein typisches Simulink-Modell aussieht und Regeln geben vor, welche Bedingungen die Messwerte einhalten müssen. Zur Präsentation der Ergebnisse werden die Messwerte mit ihren erlaubten Grenzen und die aggregierten Bewertungen geeignet visualisiert. Aus der Qualitätsbewertung wird schließlich eine Handlungsempfehlung abgeleitet. Zusammenfassend liefert diese Arbeit folgende Beiträge zum aktuellen Stand der Forschung:

**Qualitätsmodell** Konzeption eines erweiterten und angepassten Qualitätsmodells für die modellbasierte Entwicklung mit MATLAB Simulink (vergleiche [SK10]).

- Sammlung neuer bzw. an die modellbasierte Entwicklung mit Simulink adaptierter Metriken, die die Semantik der Modelle berücksichtigen und somit spezifische Aussagen treffen können.

- Konzeption der Faktoren und Kriterien des Qualitätsmodells und Einordnung der Metriken in das Qualitätsmodell.

**Modellqualitätsbewertung** Automatisierte Ermittlung der Modellqualität (vergleiche [SP11]).

- Erstellung eines Konzepts zur Bestimmung der erlaubten Grenzen für die Messwerte der Metriken.
- Aggregation der Metrikbewertungen im Qualitätsmodell, um einen Überblick über die Gesamtqualität zu erhalten.

**Fallstudie** Durchführung einer Fallstudie im PKW-Bereich der Daimler AG mit dem implementierten Prototyp.

Die Evaluation der Modellqualitätsbewertung mit den PKW-Modellen der Daimler AG in Kapitel 7 hat erfolgreich gezeigt, dass eine hohe Übereinstimmung zwischen der Modellqualitätsbewertung und den Einschätzungen der Experten besteht. Allerdings muss bei der Interpretation der Bewertungen differenziert werden. Viele Metriken machen Aussagen über die Simulink-Modelle an sich. Das bedeutet, wenn durch diese Metriken Probleme identifiziert werden, dann können diese Probleme durch die Ableitung von Handlungsempfehlungen in den Modellen beseitigt werden (wie in Abschnitt 5.3 beschrieben). Andere Metriken hingegen machen Aussagen über sehr abstrakte Sachverhalte, wie z. B. über den Prozentsatz der im Modell bereits umgesetzten Anforderungen. Solche Aussagen sind sehr global und z. B. nicht mit einzelnen strukturellen Auffälligkeiten in den Modellen zu vergleichen. Daher muss bei jeder Modellqualitätsbewertung untersucht werden, um welche Art von Problemen es sich handelt, wenn ein Modell schlecht bewertet wurde. Gibt es viele technische Probleme in den Modellen, müssen die Modellierer die Modelle überarbeiten. Sind hingegen z. B. zu wenige Anforderungen umgesetzt, müssen Maßnahmen auf der Ebene des Projektmanagements getroffen werden. Daher sind die geforderten Anwendungsfälle der Zielsetzung aus Abschnitt 1.2 abgedeckt, wobei jede Zielgruppe ihre eigene Sicht auf die Modellqualitätsbewertung hat. Die Modellierer interessieren sich für konkrete Probleme in den Modellen, die Manager profitieren von Informationen, die für das Projektmanagement von Interessen sind, und der Einkauf bekommt einen ersten Eindruck, wie ein Modell einzuschätzen ist.

Des Weiteren erfüllt die Modellqualitätsbewertung die folgenden Anforderungen der Zielsetzung aus Abschnitt 1.2:

**[A1]** Automatisierbare Ausführung
Da die Modellqualitätsbewertung nur die Modelle selbst oder Reports von anderen Tools betrachtet, ist eine automatisierte Ausführung möglich.

**[A2]** Prototypische Umsetzung
Der in Kapitel 6 vorgestellte Prototyp setzt den in dieser Arbeit beschriebenen Ansatz um und macht ihn dadurch ausführbar.

**[A3]** Nachvollziehbarkeit der Bewertung
Das in Kapitel 5 beschriebene Konzept beschreibt die für die Bewertung der Modelle verwendeten Konzepte und macht somit transparent, wie die Bewertungen zustande kommen.

**[A4]** Wiederholbarkeit der Bewertung
Die in Kapitel 5 beschriebenen Konzepte stellen sicher, dass bei einer wiederholten Ausführung der Modellqualitätsbewertung eine identische Bewertung zustande kommt.

**[A5]** Projektspezifische Anpassbarkeit und Erweiterbarkeit
Der baumartige Aufbau des Qualitätsmodells in Kapitel 4 erlaubt beliebige Erweiterungen bzw. Änderungen an dem Qualitätsmodell. Dabei kann das in dieser Arbeit vorgestellte Qualitätsmodell als Ausgangsbasis dienen. Außerdem ermöglichen die Regeln in Kapitel 5 eine projektspezifische Anpassung der Modellqualitätsbewertung.

**[A6]** Unabhängigkeit von der Zielhardware
Die Modellqualitätsbewertung ist unabhängig von der Zielhardware, da nur das Simulink-Modell und die externen Reports benötigt werden. Die Verwendung des Prototyps ist sogar unabhängig von MATLAB Simulink, nachdem das Modell als XML-Datei serialisiert wurde (vergleiche Abschnitt 6.3.1).

**[A7]** Ausschließlich lesender Zugriff auf die Modelle
Die Metriken aus Kapitel 3 nehmen keine Änderungen an den Modellen vor. Im Prototyp sind Änderungen an den Modellen aus technischer Sicht ausgeschlossen, da nur auf das serialisierte Modell zugegriffen wird.

## 8.2. Ausblick

In diesem Abschnitt wird auf mögliche weiterführende Punkte eingegangen, die auf das Verfahren zur automatisierten Modellqualitätsbewertung aufbauen und über das in dieser Arbeit geschilderte Verfahren hinausgehen.

Zusätzlich zu der Betrachtung der Modellqualität über die Zeit in Abschnitt 5.4 ist ein direkter Qualitätsvergleich zweier Modellversionen denkbar. Denn obwohl die Gesamtbewertung der Modelle gleich geblieben ist, können sich einzelne Teile verbessert und andere Teile verschlechtert haben. In der Gesamtbewertung sind diese Veränderungen u. U. nicht sichtbar, da sie sich gegenseitig aufheben können. Würde aber auch die Qualität einzelner Teile bzw. Module betrachtet werden, könnten Hinweise gegeben werden, in welchen Teilen sich die Qualität verändert hat.

In dieser Arbeit wurde das Verfahren zur Modellqualitätsbewertung exemplarisch anhand von MATLAB Simulink-Modellen konzipiert und prototypisch umgesetzt. Der Ansatz lässt sich leicht auf andere Werkzeuge für die modellbasierte Entwicklung von Steuergeräten übertragen. In erster Linie müssen dazu die verwendeten Metriken entweder angepasst oder ersetzt werden. So würde sich beispielsweise eine Adaption für ASCET von ETAS[1] anbieten, da ASCET ebenfalls Blockdiagramme und Zustandsautomaten zur Modellierung verwendet.

Um die Anbindung weiterer Werkzeuge zu erleichtern bzw. um von anderen Werkzeugen verwendet werden zu können, würde sich die Verwendung des Software Metrics Metamodel[2] (SMM) der Object Management Group anbieten. Dazu muss untersucht werden, wie sich das Qualitätsmodell aus Kapitel 5 und die im Prototyp aus Kapitel 6 verwendeten Konzepte auf die des SMM abbilden lassen. Das SMM stellt dazu abstrakte Konzepte wie Messwerte, Messergebnisse, Messobjekte und aggregierte Bewertungen zur Verfügung.

In dieser Arbeit wurde vor allem der Kontroll- und Datenflussanteil der Modelle betrachtet. Für den zustandsorientierten Anteil, der im Fall von Simulink aus Stateflow-Diagrammen besteht, wurden nur wenige Metriken umgesetzt. Arbeiten über Metriken für Zustandsautomaten im Allgemeinen sind in der Literatur zu finden, da das Konzept des Zustandsautomaten

[1] http://www.etas.com/de/products/ascet_software_products.php
[2] http://www.omg.org/spec/SMM

sehr weit verbreitet ist (siehe beispielsweise [GMP02] oder [RCLGP08]). Daher bietet es sich an, diese bestehenden Metriken für Stateflow-Diagramme zu adaptieren. Eine ausführlichere Betrachtung der Stateflow-Diagramme ist vor allem für Modelle wichtig, die zu einem großen Teil aus Stateflow-Diagrammen bestehen.

In dieser Arbeit werden für die Erstellung des Referenzmodells empirische Daten verwendet. Es wird der Durchschnitt über mehrere Modelle in einem Projekt gebildet. Das ermöglicht, wie gefordert, auffällige Messwerte eines Modells zu finden. Allerdings wäre ein Ansatz wünschenswert, der auf konstruktivere Art und Weise die Aufstellung eines Referenzmodells ermöglicht. Das heißt, es wird eine Vorschrift benötigt, die ausdrücken kann, wie ein Modell im Idealfall auszusehen hat bzw. welche Bereiche für die Messwerte des Modells erlaubt sind.

Eine langfristige Vision für die automatisierte Modellqualitätsbewertung könnte eine Verbindung mit dem halbautomatischen Modell-Refactoring von Modellen sein. Dazu müssten nicht nur Handlungsempfehlungen aus der Modellqualitätsbewertung abgeleitet werden, sondern Vorschläge für konkrete Refactoring-Operationen. So wäre es möglich, die entdeckten Probleme mit sehr geringem Aufwand zu beseitigen. Dieser Ansatz steht auch nicht in Widerspruch mit der Anforderung [**A7**], die für die Qualitätsbewertung einen ausschließlich lesenden Zugriff auf die Modelle fordert, da ein Modellierer für jede vorgeschlagene Refactoring-Operation entscheiden muss, ob diese durchgeführt werden soll oder nicht.

# A. Anhänge

**A.1 Zusätzliche Informationen zu den Metriken** Dieser Abschnitt enthält zusätzliche Informationen zu den Metriken. Das umfasst beispielsweise Auszüge aus verwendeten Modellierungsrichtlinien.

**A.2 Zuordnung der Metriken zu den Qualitätsaspekten** In diesem Abschnitt wird die Zuordnung der Metriken zu den Qualitätsaspekten aufgelistet.

**A.3 Einfluss der Metriken** Dieser Abschnitt listet die Metriken und ihren Einfluss auf die Gesamtqualität auf. Die Evaluation in Kapitel 7 wurde mit diesen Einflüssen durchgeführt.

**A.4 Verwendete Regeln** Dieser Abschnitt enthält die verwendeten Regeln für die Modellqualitätsbewertung in Kapitel 7.

**A.5 Konfiguration des Prototyps** Dieser Abschnitt enthält die Konfigurationsdateien des Prototyps aus Kapitel 6.

**A.6 Im Prototyp fehlende Metriken** In diesem Abschnitt werden die Metriken aufgeführt, die nicht im Prototyp implementiert wurden.

**A.7 Weitere Ansichten in der grafischen Oberfläche** Dieser Abschnitt enthält weitere Ansichten des Prototyps, in denen zusätzliche Informationen über die Messwerte einer Metrik angezeigt werden.

## A.1. Zusätzliche Informationen zu den Metriken

In diesem Abschnitt werden zusätzliche Informationen zu den Metriken zur Verfügung gestellt. Das umfasst beispielsweise Auszüge aus verwendeten Modellierungsrichtlinien.

### A.1.1. Beispiele für MAAB-Modellierungsmuster

Dieser Abschnitt enthält einige Beispiele für Simulink-Modellierungsmuster, wie sie von der Metrik »Anzahl der verwendeten Modellierungsmuster« verwendet werden. Die Muster aus Abbildung A.1, A.2 und A.3 wurden aus den MAAB-Richtlinien[1] entnommen. Am Ende der Bildunterschrift steht in Klammern die ID der ursprünglichen MAAB-Richtlinie.

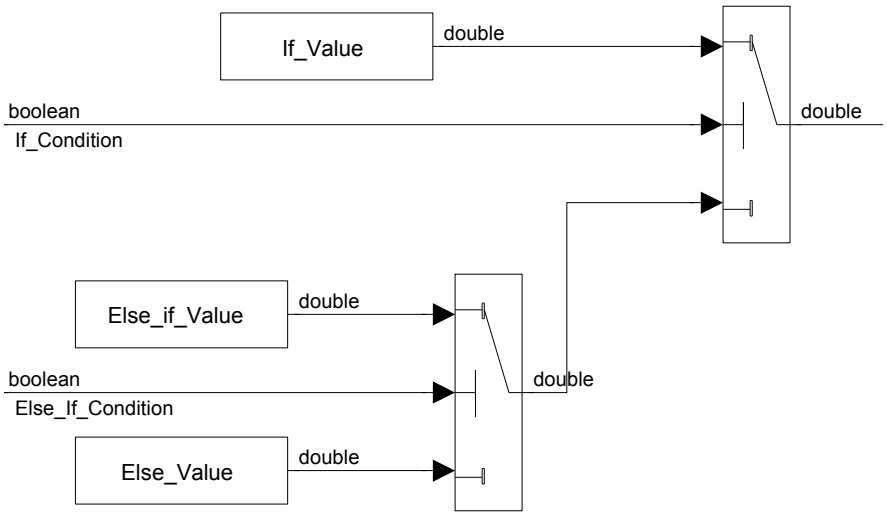



Abbildung A.1.: Muster für ein If-then-else-if-Konstrukt (db_0114)

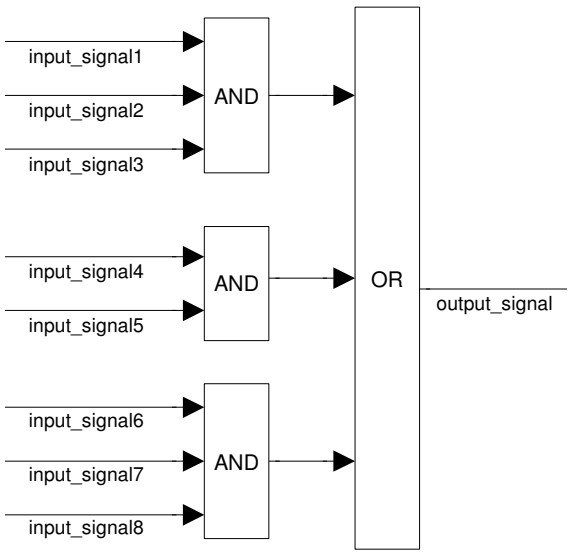



Abbildung A.2.: Muster für eine logische Formel in der disjunktiven Normalform (db_0116)

[1] http://www.mathworks.de/automotive/standards/maab.html?

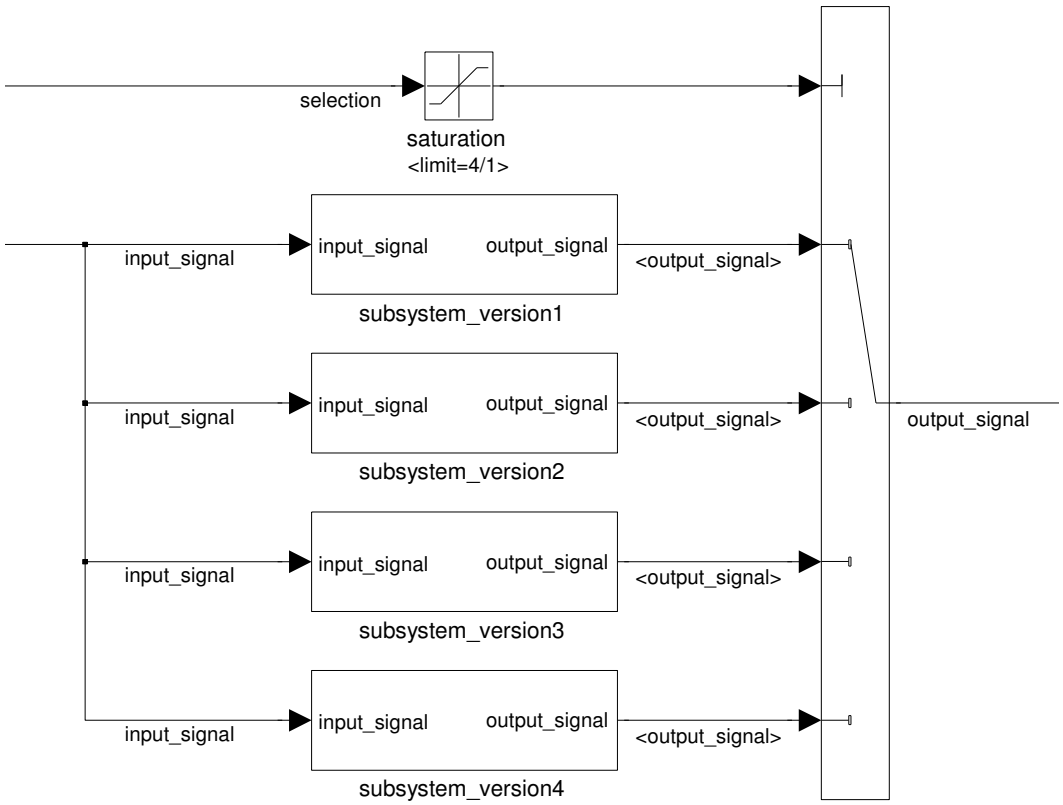



Abbildung A.3.: Muster für ein Switch-Case-Konstrukt (db_0115)

### A.1.2. Verwendete dSPACE-Richtlinien

Folgende dSPACE-Richtlinien[2] werden von der Metrik »Anzahl der verletzten dSPACE-Richtlinien« verwendet:

| | |
|---|---|
| $tl\_0003_a$ | Limitations with regard to matrix signals |
| $tl\_0003_b$ | Limitations with regard to matrix signals |
| $ds\_0023_a$ | Limitations with regard to the Discrete-Time Integrator Block: Vector Signals |
| $ds\_0023_b$ | Limitations with regard to the Discrete-Time Integrator Block: External Reset |
| ds_0025 | Limitations with regard to the Relational Operator and Logical Operator Blocks |
| ds_0027 | Limitations with regard to the Look-up Table (n-D) and Direct Look-up Table (n-D) Blocks |
| ds_0029 | Limitations with regard to operand numbers for the Product block |
| ds_0030 | Limitations with regard to parameters for the Product block |
| ds_0031 | Limitations with regard to the Sum block |
| ds_0032 | Limitations with regard to the Math Function block |
| ds_0049 | Limitations with regard to the Bus Creator block |

Tabelle A.1.: Verwendete dSPACE-Richtlinien

[2] http://www.dspace.de/de/gmb/home/support/kb/kbtl/tlmodguide/tlapp_modelguide.cfm

### A.1.3. Verwendete MISRA TL-Richtlinien

Folgende MISRA TL-Richtlinien[3] werden von der Metrik »Anzahl der verletzten MISRA TL-Richtlinien« verwendet:

| | |
|---|---|
| misra_tl_1_2 | Comments in block properties |
| misra_tl_4_2 | Permitted data types at the Stateflow Chart Interface |
| misra_tl_4_4 | Avoidance of Saturation in Integer Arithmetics |
| misra_tl_4_5 | Avoidance of Rounding Operations |
| misra_tl_5_1 | Attributes of Look-Up table outputs |
| misra_tl_5_2$_a$ | Show default case option of Switch Case block shall be set |
| misra_tl_5_3 | Use of Data Store blocks across TargetLink subsystems |
| misra_tl_5_6 | Use of the Multiport Switch block |
| misra_tl_5_7 | Use of data types for Multiport Switch block |
| misra_tl_5_8 | Use of the Switch block |
| misra_tl_6_1 | Power of two scaling |
| misra_tl_6_2 | Zero offsets |
| misra_tl_7_1 | Generating generic code without pragmas |
| misra_tl_7_2 | Avoiding target-specific assembly code |
| misra_tl_7_5 | Local macros should not be used |
| misra_tl_7_6 | Correct declaration of external variables |
| misra_tl_7_7 | Use of Include Statements in Custom Code Block |

Tabelle A.2.: Verwendete MISRA TL-Richtlinien

### A.1.4. Verwendete Strong Data Typing-Richtlinien

Folgende Strong Data Typing-Richtlinien[4] werden von der Metrik »Anzahl der potentiellen Datentyp-Probleme« verwendet:

| | |
|---|---|
| sdt_sc001 | Strong Data Typing at the TargetLink function interface |
| sdt_sc002 | Strong Data Typing at the Stateflow interface |
| sdt_sc003 | Strong Data Typing of merge blocks |
| sdt_sc004 | Strong Data Typing of arithmetic blocks |
| sdt_sa001 | Strong Data Typing at the input of Logical Operator blocks |
| sdt_sa003 | Control input of Switch- and Multiport Switch blocks |
| sdt_ic001 | Initial value of the output of merge blocks |
| sdt_ic002 | Initial value of the output of conditional subsystems |
| sdt_ic003 | Default paths of If- and Switch-Case Blocks |
| sdt_il001 | Avoidance of duplicated port signals |

Tabelle A.3.: Verwendete Strong Data Typing-Richtlinien

[3] http://www.misra.org.uk/Publications/tabid/57/Default.aspx
[4] http://www.model-engineers.com/fileadmin/mes/PUBLIC/MXAM/MES_MXAM-SDT-Guidelines_V1.3.pdf

## A.2. Zuordnung der Metriken zu den Qualitätsaspekten

In diesem Abschnitt wird die Zuordnung der Metriken zu den Qualitätsaspekten (vergleiche Abschnitt 4.1) aufgelistet. Die Unterabschnitte A.2.1, A.2.2, A.2.3, A.2.4, A.2.5, A.2.6 und A.2.7 listen die Metriken der jeweiligen Aspekte auf.

### A.2.1. Metriken des Qualitätsaspekts Anforderung

Die Metriken dieses Qualitätsaspekts treffen Aussagen über die Anforderungen eines Simulink-Modells.

| Metrik |
|---|
| Metrik 83: Anzahl der Anforderungen |
| Metrik 84: Anforderungsabdeckung |

Tabelle A.4.: Metriken des Qualitätsaspekts Anforderung

### A.2.2. Metriken des Qualitätsaspekts Architektur

Die Metriken dieses Qualitätsaspekts treffen Aussagen über die Architektur eines Simulink-Modells.

| Metrik |
|---|
| Metrik 65: Anzahl der Architekturverletzungen |

Tabelle A.5.: Metriken des Qualitätsaspekts Architektur

### A.2.3. Metriken des Qualitätsaspekts Laufzeit

Die Metriken dieses Qualitätsaspekts treffen Aussagen über die Laufzeit eines Simulink-Modells.

| Metrik |
|---|
| Metrik 77: Anzahl der rechenintensiven Blöcke |
| Metrik 79: Worst-Case-Execution-Time |

Tabelle A.6.: Metriken des Qualitätsaspekts Laufzeit

### A.2.4. Metriken des Qualitätsaspekts Simulink

Die Metriken dieses Qualitätsaspekts treffen allgemeine Aussagen über ein Simulink-Modell und sind auf jedes Simulink-Modell anwendbar.

| **Metrik** |
|---|
| Metrik 1: Anzahl der Blöcke |
| Metrik 4: Anzahl der Ground-Blöcke |
| Metrik 5: Anzahl der Terminator-Blöcke |
| Metrik 6: Anzahl der Saturation-Blöcke |
| Metrik 7: Anzahl der DataStoreMemory-Blöcke |
| Metrik 8: Anzahl der DataStoreRead-Blöcke |
| Metrik 9: Anzahl der DataStoreWrite-Blöcke |
| Metrik 10: Anzahl der ModelInfo-Blöcke |
| Metrik 11: Anzahl der BusCreator-Blöcke |
| Metrik 12: Anzahl der BusSelector-Blöcke |
| Metrik 13: Anzahl der Goto-Blöcke |
| Metrik 14: Anzahl der From-Blöcke |
| Metrik 15: Anzahl der UnitDelay-Blöcke |
| Metrik 16: Anzahl der DataStoreMemory-Blöcke ohne Schreibzugriff |
| Metrik 17: Anzahl der DataStoreMemory-Blöcke ohne Lesezugriff |
| Metrik 19: Anzahl der UnitDelay-Blöcke in Vorwärtsrichtung |
| Metrik 20: Anzahl der globalen Goto-Blöcke |
| Metrik 21: Durchschnittliche Anzahl an nachfolgenden Blöcken |
| Metrik 22: Durchschnittliche Blockinstabilität |
| Metrik 23: Anzahl der verwendeten Blocktypen |
| Metrik 24: Anzahl der magischen Konstanten |
| Metrik 25: Anzahl der Werte magischer Konstanten |
| Metrik 26: Anzahl der Konstanten mit Namen |
| Metrik 27: Anzahl der Linien |
| Metrik 28: Anzahl der Linien-Annotationen |
| Metrik 29: Anzahl der überkreuzenden Linien |
| Metrik 30: Anzahl der Subsysteme |
| Metrik 31: Anzahl der konfigurierbaren Subsysteme |
| Metrik 32: Anzahl der atomaren Subsysteme |
| Metrik 33: Durchschnittliche Anzahl an Subsystemen pro Subsystem |
| Metrik 34: Anzahl der arithmetischen Subsysteme |
| Metrik 35: Anzahl der logischen Subsysteme |
| Metrik 36: Anzahl der signalführenden Subsysteme |
| Metrik 37: Anzahl der schaltenden Subsysteme |
| Metrik 38: Anzahl der testbaren Subsysteme mit impliziten Inport-Datentypen |
| Metrik 39: Anzahl der testbaren Subsysteme mit eingehenden Bussen |
| Metrik 40: Tiefe der Subsystem-Hierarchie |
| Metrik 41: Durchschnittliche Höhe der Subsysteme auf dem Bildschirm |
| Metrik 42: Durchschnittliche Breite der Subsysteme auf dem Bildschirm |
| Metrik 43: Anzahl der Subsystem-Annotationen |
| Metrik 44: Durchschnittliche Länge der Subsystem-Annotationen |
| Metrik 45: Anzahl der verwendeten Bibliotheksverknüpfungen |
| Metrik 46: Anzahl der deaktivierten Bibliotheksverknüpfungen |
| Metrik 47: Anzahl der deaktivierten und unauffindbaren Bibliotheksverknüpfungen |
| Metrik 48: Anzahl der deaktivierten Bibliotheksverknüpfungen mit veränderter Struktur |
| Metrik 49: Anzahl der verwendeten Bibliotheken |
| Metrik 50: Durchschnittliche Anzahl der unabhängigen Berechnungspfade |

| Metrik |
|---|
| Metrik 51: Anzahl der Zyklen |
| Metrik 52: Anzahl der verfallenen Ergebnisse |
| Metrik 53: Durchschnittlicher Vernetzungsgrad |
| Metrik 54: Durchschnittliche Signalpfadlänge |
| Metrik 55: Durchschnittliche Länge der Signalpfade auf dem Bildschirm |
| Metrik 56: Durchschnittliche Breite der Eingangsschnittstelle |
| Metrik 57: Durchschnittliche Breite der Ausgangsschnittstelle |
| Metrik 58: Durchschnittliche Busgröße |
| Metrik 59: Anzahl der internen Zustände |
| Metrik 60: Anzahl der Wertebereichsverletzungen |
| Metrik 61: Anzahl der Modellklone |
| Metrik 62: Anzahl der verwendeten Modellierungsmuster |
| Metrik 63: Höhe der globalen Komplexität |
| Metrik 64: Durchschnittliche lokale Komplexität |

Tabelle A.7.: Metriken des Qualitätsaspekts Simulink

### A.2.5. Metriken des Qualitätsaspekts Stateflow

Die Metriken dieses Qualitätsaspekts setzen die Verwendung von Stateflow voraus. Sind keine Stateflow-Diagramme in dem betrachteten Modell enthalten, liefern die aufgelisteten Metriken immer den Messwert 0 zurück.

| Metrik |
|---|
| Metrik 68: Anzahl der Stateflow-Blöcke |
| Metrik 69: Durchschnittliche Anzahl an Stateflow-Zuständen |
| Metrik 70: Anzahl der unerreichbaren Zustände |

Tabelle A.8.: Metriken des Qualitätsaspekts Stateflow

### A.2.6. Metriken des Qualitätsaspekts Targetlink

Die Metriken dieses Qualitätsaspekts setzen die Verwendung von Targetlink voraus. Sind keine Targetlink-Blöcke in dem betrachteten Modell enthalten, liefern die aufgelisteten Metriken immer den Messwert 0 zurück.

| Metrik |
|---|
| Metrik 2: Anzahl der TargetlinkFunction-Blöcke |
| Metrik 3: Anzahl der CustomCode-Blöcke |
| Metrik 18: Anzahl der aktivierten Targetlink Block-Saturierungen |
| Metrik 66: Anzahl der verletzten dSPACE-Richtlinien |
| Metrik 67: Anzahl der verletzten MISRA TL-Richtlinien |
| Metrik 71: Anzahl der Targetlink-Fehler |
| Metrik 72: Anzahl der Targetlink-Warnungen |

| |
|---|
| Metrik 73: Anzahl der Port-Blöcke ohne Data Dictionary-Eintrag |
| Metrik 74: Anzahl der verwendeten Skalierungen ohne Data Dictionary-Eintrag |
| Metrik 75: Anzahl der verwendeten Typen aus dem Data Dictionary |
| Metrik 76: Anzahl der verwendeten Variablen aus dem Data Dictionary |
| Metrik 78: Anzahl der potentiellen Datentyp-Probleme |

Tabelle A.9.: Metriken des Qualitätsaspekts Targetlink

### A.2.7. Metriken des Qualitätsaspekts Test

Die Metriken dieses Qualitätsaspekts treffen Aussagen über die Tests eines Simulink-Modells.

| **Metrik** |
|---|
| Metrik 80: Anzahl der Testfälle |
| Metrik 81: Erfolgreiche Testfälle |
| Metrik 82: Testabdeckung |

Tabelle A.10.: Metriken des Qualitätsaspekts Test

## A.3. Einfluss der Metriken

Bereits in Kapitel 4 wurde bei der Einsortierung der Metriken in das Qualitätsmodell die Höhe des Einflusses jeder Metrik auf die Gesamtqualität festgelegt. Tabelle A.11 listet die Metriken und ihren Einfluss auf die Gesamtqualität nochmals auf. Aus Gründen der Verständlichkeit wurde folgende Abbildung verwendet:

| **Notation aus Kapitel 4** | **Notation in Tabelle A.11** |
|---|---|
| (+) | klein |
| (++) | mittel |
| (+++) | groß |

Dabei wird dem Einfluss *klein* ein Gewicht von 1, dem Einfluss *mittel* ein Gewicht von 2 und dem Einfluss *groß* ein Gewicht von 3 zugeordnet. Die Evaluation in Kapitel 7 wurde mit diesen Gewichten durchgeführt.

| **Metrik** | **Einfluss** |
|---|---|
| Metrik 2: Anzahl der TargetlinkFunction-Blöcke | klein |
| Metrik 4: Anzahl der Ground-Blöcke | klein |
| Metrik 5: Anzahl der Terminator-Blöcke | klein |
| Metrik 6: Anzahl der Saturation-Blöcke | klein |
| Metrik 18: Anzahl der aktivierten Targetlink Block-Saturierungen | klein |
| Metrik 23: Anzahl der verwendeten Blocktypen | klein |
| Metrik 26: Anzahl der Konstanten mit Namen | klein |
| Metrik 32: Anzahl der atomaren Subsysteme | klein |
| Metrik 34: Anzahl der arithmetischen Subsysteme | klein |

| Metrik | Einfluss |
|---|---|
| Metrik 35: Anzahl der logischen Subsysteme | klein |
| Metrik 49: Anzahl der verwendeten Bibliotheken | klein |
| Metrik 62: Anzahl der verwendeten Modellierungsmuster | klein |
| Metrik 75: Anzahl der verwendeten Typen aus dem Data Dictionary | klein |
| Metrik 76: Anzahl der verwendeten Variablen aus dem Data Dictionary | klein |
| Metrik 80: Anzahl der Testfälle | klein |
| Metrik 83: Anzahl der Anforderungen | klein |
| Metrik 1: Anzahl der Blöcke | mittel |
| Metrik 7: Anzahl der DataStoreMemory-Blöcke | mittel |
| Metrik 8: Anzahl der DataStoreRead-Blöcke | mittel |
| Metrik 9: Anzahl der DataStoreWrite-Blöcke | mittel |
| Metrik 10: Anzahl der ModelInfo-Blöcke | mittel |
| Metrik 11: Anzahl der BusCreator-Blöcke | mittel |
| Metrik 12: Anzahl der BusSelector-Blöcke | mittel |
| Metrik 13: Anzahl der Goto-Blöcke | mittel |
| Metrik 14: Anzahl der From-Blöcke | mittel |
| Metrik 15: Anzahl der UnitDelay-Blöcke | mittel |
| Metrik 16: Anzahl der DataStoreMemory-Blöcke ohne Schreibzugriff | mittel |
| Metrik 17: Anzahl der DataStoreMemory-Blöcke ohne Lesezugriff | mittel |
| Metrik 20: Anzahl der globalen Goto-Blöcke | mittel |
| Metrik 22: Durchschnittliche Blockinstabilität | mittel |
| Metrik 25: Anzahl der Werte magischer Konstanten | mittel |
| Metrik 27: Anzahl der Linien | mittel |
| Metrik 28: Anzahl der Linien-Annotationen | mittel |
| Metrik 29: Anzahl der überkreuzenden Linien | mittel |
| Metrik 30: Anzahl der Subsysteme | mittel |
| Metrik 31: Anzahl der konfigurierbaren Subsysteme | mittel |
| Metrik 33: Durchschnittliche Anzahl an Subsystemen pro Subsystem | mittel |
| Metrik 36: Anzahl der signalführenden Subsysteme | mittel |
| Metrik 37: Anzahl der schaltenden Subsysteme | mittel |
| Metrik 40: Tiefe der Subsystem-Hierarchie | mittel |
| Metrik 41: Durchschnittliche Höhe der Subsysteme auf dem Bildschirm | mittel |
| Metrik 42: Durchschnittliche Breite der Subsysteme auf dem Bildschirm | mittel |
| Metrik 43: Anzahl der Subsystem-Annotationen | mittel |
| Metrik 44: Durchschnittliche Länge der Subsystem-Annotationen | mittel |
| Metrik 45: Anzahl der verwendeten Bibliotheksverknüpfungen | mittel |
| Metrik 46: Anzahl der deaktivierten Bibliotheksverknüpfungen | mittel |
| Metrik 52: Anzahl der verfallenen Ergebnisse | mittel |
| Metrik 54: Durchschnittliche Signalpfadlänge | mittel |
| Metrik 56: Durchschnittliche Breite der Eingangsschnittstelle | mittel |
| Metrik 57: Durchschnittliche Breite der Ausgangsschnittstelle | mittel |
| Metrik 59: Anzahl der internen Zustände | mittel |
| Metrik 63: Höhe der globalen Komplexität | mittel |
| Metrik 64: Durchschnittliche lokale Komplexität | mittel |
| Metrik 65: Anzahl der Architekturverletzungen | mittel |
| Metrik 66: Anzahl der verletzten dSPACE-Richtlinien | mittel |
| Metrik 67: Anzahl der verletzten MISRA TL-Richtlinien | mittel |

| Metrik | Einfluss |
|---|---|
| Metrik 68: Anzahl der Stateflow-Blöcke | mittel |
| Metrik 69: Durchschnittliche Anzahl an Stateflow-Zuständen | mittel |
| Metrik 72: Anzahl der Targetlink-Warnungen | mittel |
| Metrik 73: Anzahl der Port-Blöcke ohne Data Dictionary-Eintrag | mittel |
| Metrik 74: Anzahl der verwendeten Skalierungen ohne Data Dictionary-Eintrag | mittel |
| Metrik 77: Anzahl der rechenintensiven Blöcke | mittel |
| Metrik 78: Anzahl der potentiellen Datentyp-Probleme | mittel |
| Metrik 3: Anzahl der CustomCode-Blöcke | groß |
| Metrik 19: Anzahl der UnitDelay-Blöcke in Vorwärtsrichtung | groß |
| Metrik 21: Durchschnittliche Anzahl an nachfolgenden Blöcken | groß |
| Metrik 24: Anzahl der magischen Konstanten | groß |
| Metrik 38: Anzahl der testbaren Subsysteme mit impliziten Inport-Datentypen | groß |
| Metrik 39: Anzahl der testbaren Subsysteme mit eingehenden Bussen | groß |
| Metrik 47: Anzahl der deaktivierten und unauffindbaren Bibliotheksverknüpfungen | groß |
| Metrik 48: Anzahl der deaktivierten Bibliotheksverknüpfungen mit veränderter Struktur | groß |
| Metrik 50: Durchschnittliche Anzahl der unabhängigen Berechnungspfade | groß |
| Metrik 51: Anzahl der Zyklen | groß |
| Metrik 53: Durchschnittlicher Vernetzungsgrad | groß |
| Metrik 55: Durchschnittliche Länge der Signalpfade auf dem Bildschirm | groß |
| Metrik 58: Durchschnittliche Busgröße | groß |
| Metrik 60: Anzahl der Wertebereichsverletzungen | groß |
| Metrik 61: Anzahl der Modellklone | groß |
| Metrik 70: Anzahl der unerreichbaren Zustände | groß |
| Metrik 71: Anzahl der Targetlink-Fehler | groß |
| Metrik 79: Worst-Case-Execution-Time | groß |
| Metrik 81: Erfolgreiche Testfälle | groß |
| Metrik 82: Testabdeckung | groß |
| Metrik 84: Anforderungsabdeckung | groß |

Tabelle A.11.: Einfluss der Metriken auf die Gesamtqualität

## A.4. Verwendete Regeln

In diesem Abschnitt werden die verwendeten Regeln für die Modellqualitätsbewertung in Kapitel 7 aufgelistet. Weitere Informationen zu den Regeln und Grenzwerten finden sich in Abschnitt 5.1.2.

```
1 ◺ 0 ≤ Anzahl der potentiellen Datentyp-Probleme ≤ 1,0 · Anzahl der Subsysteme
2 ◺ 0 ≤ Anzahl der Targetlink-Fehler ≤ 0
3 ◺ 0 ≤ Anzahl der Targetlink-Warnungen ≤ 0,5 · Anzahl der Subsysteme
4 ◺ 0 ≤ Anzahl der DataStoreMemory-Blöcke ohne Lesezugriff ≤ 0
5 ◺ 0 ≤ Anzahl der DataStoreMemory-Blöcke ohne Schreibzugriff ≤ 0
6 ◺ 0 ≤ Anzahl der testbaren Subsysteme mit impliziten Inport-Datentypen ≤ 5,0
```

```
◺ 0 ≤ Anzahl der testbaren Subsysteme mit eingehenden Bussen ≤ 5,0
◺ 0 ≤ Anzahl der konfigurierbaren Subsysteme ≤ 0
◺ 0 ≤ Anzahl der globalen Goto-Blöcke ≤ 0,1 · Anzahl der Subsysteme
◺ 0 ≤ Anzahl der UnitDelay-Blöcke ≤ 2,0 · Anzahl der Subsysteme
◺ 0 ≤ Anzahl der UnitDelay-Blöcke in Vorwärtsrichtung ≤ 2,0
◺ 0 ≤ Anzahl der magischen Konstanten ≤ 0,1 · Anzahl der Subsysteme
◺ 0 ≤ Anzahl der Werte magischer Konstanten ≤ 10,0
◺ 0 ≤ Anzahl der deaktivierten und unauffindbaren Bibliotheksverknüpfungen ≤ 0
◺ 0 ≤ Anzahl der deaktivierten Bibliotheksverknüpfungen ≤ 5,0 · Anzahl der Subsysteme
◺ 0 ≤ Anzahl der deaktivierten Bibliotheksverknüpfungen mit veränderter Struktur ≤ 0
◺ 0 ≤ Anzahl der rechenintensiven Blöcke ≤ 0,001 · Anzahl der Blöcke
```

Listing A.1: Regeln für bevorzugt kleine Messwerte, die im besten Fall sogar 0 sind

```
Anzahl der CustomCode-Blöcke = 0
Anzahl der Port-Blöcke ohne Data Dictionary-Eintrag = 0
Anzahl der verwendeten Skalierungen ohne Data Dictionary-Eintrag = 0
Anzahl der Wertebereichsverletzungen = 0
Anzahl der unerreichbaren Zustände = 0
```

Listing A.2: Regeln für Messwerte, die im besten Fall 0 sind

```
Anzahl der DataStoreMemory-Blöcke ≤ Anzahl der DataStoreRead-Blöcke
Anzahl der DataStoreMemory-Blöcke ≤ Anzahl der DataStoreWrite-Blöcke
```

Listing A.3: Regeln für Messwerte, die eine vorgegebene Relation erfüllen müssen

## A.5. Konfiguration des Prototyps

Dieser Abschnitt enthält die Konfigurationsdateien des Prototyps aus Kapitel 6. Die Konfigurationsdateien enthalten Kommentare, in denen die einzelnen Optionen beschrieben sind. Listing A.4 zeigt die allgemeine Konfiguration des Prototyps. Es können die Java-Packages angegeben werden, die die Metriken enthalten. Außerdem können Modelldateien angegeben werden, für die beim Aufruf des Prototyps keine absoluten Pfade mehr notwendig sind. Listing A.5 zeigt die Konfigurationsdatei für die Ausführung der Metriken. Abschließend zeigt Listing A.6 die Konfigurationsdatei für die Modellqualitätsbewertung.

```
#
# Diese Datei enthält die grundlegenden Einstellungen des Prototyps.
#

version = 1.0

# Java-Pakete, in denen Metriken gesucht werden
metricPackages = modelquality.modelmeasurer.metric

# Pfade zu bekannten Modellen (für die kein absoluter Pfad mehr notwendig ist)
knownModels = d:\\qualityrating\\models\\firstModel.mdl, \
        d:\\qualityrating\\models\\secondModel.mdl, \
        d:\\qualityrating\\models\\thirdModel.mdl, \
        d:\\qualityrating\\models\\fourthModel.mdl
```

Listing A.4: Allgemeine Konfigurationsdatei des Prototyps

```
#
# Diese Datei enthält die Standardwerte für den Model Measurer.
#

### <AUSWAHL der verwendeten Metriken (es muss genau eine der Möglichkeiten ge-
### wählt werden!) (nicht gewählte Möglichkeiten müssen auskommentiert werden)

# [1] verwende alle Metriken der angegebenen Aspekte
metricSelectionMethod = aspects
metricSelection = architecture, requirements, runtime, simulink, \
        stateflow, targetlink, test

# [2] verwende alle verfügbaren Metriken
metricSelectionMethod = metricsWhitelist
metricSelection = *

# [3] verwende eine Whitelist zur Auswahl der Metriken
metricSelectionMethod = metricsWhitelist
metricSelection = NumberOfBlocks, NumberOfSubsystems, NumberOfLines, \
        NumberOfBusCreators, NumberOfBusSelectors, NumberOfCrossedLines, \
        NumberOfGotos, NumberOfLibraryLinks, NumberOfNamedConstants, \
        NumberOfUnitDelays, NumberOfFroms, GlobalComplexity, \
        AverageInputInterfaceWidth, AverageOutputInterfaceWidth,  \
        AverageOutportConnections, NumberOfDataStoreRead, NumberOfDataStoreMemory

# [4] verwende eine Blacklist zur Auswahl der Metriken
metricSelectionMethod = metricsBlacklist
metricSelection = NumberOfAtomicSubsystems, NumberOfDataStoreWrite

### Ende der AUSWAHL>

# entscheidet, ob (zusätzliche) in den Regeln verwendete Metriken automatisch
# hinzugefügt werden sollen
addMetricsOfRules = true

# gibt die verwendete Referenzmetrik an
referenceMetric = GlobalComplexity

# mit diesem Wert werden alle normalisierten Werte in den Messwertdatenbanken
# multipliziert (um sehr kleine Werte zu vermeiden)
normalizedValueFactor = 1000

# über die Optionen können Parameter für einzelne Metriken vorgegeben werden
options.NumberOfClones.minimalCloneSize = 5
```

Listing A.5: Model Measurer-Konfigurationsdatei

```
#
# Diese Datei enthält die Standardwerte für den Quality Rater.
#

# für die Qualitätsbewertung (projekt-spezifisch) deaktivierten Regeln und Metriken
disabledRules =
disabledMetrics = NumberOfArchitectureViolations, NumberOfUnreachableStates, \
        NumberOfRangeViolations, NumberOfRequirements, NumberOfTestCases, \
        NumberOfUsedModellingPatterns, TestCoverage, PassedTestCases, \
```

```
10          RequirementsCoverage, WorstCaseExecutionTime
11
12  # entscheidet, ob Metriken ohne Bewertung aus dem Qualitätsmodell
13  # entfernt werden sollen
14  removeMetricsWithoutEvaluation = true
15
16  # Schwellwert der Aggregation
17  aggregationThreshold = 15
18
19  # entscheidet, ob die Prioritäten der Metriken aus dem Qualitätsmodell
20  # verwendet werden sollen
21  useMetricPriorities = true
```

Listing A.6: Quality Rater-Konfigurationsdatei

## A.6. Im Prototyp fehlende Metriken

In diesem Abschnitt werden die Metriken aufgeführt, die nicht im Prototyp implementiert wurden. Zusätzlich zu den Metriken wird analog zu Abschnitt 6.2 angegeben, ob die Datenquelle ein externes Tool oder ein Dokument ist.

| **Metrik** | **Datenquelle** |
|---|---|
| Metrik 60: Anzahl der Wertebereichsverletzungen | externes Tool |
| Metrik 62: Anzahl der verwendeten Modellierungsmuster | externes Tool |
| Metrik 65: Anzahl der Architekturverletzungen | Dokument |
| Metrik 70: Anzahl der unerreichbaren Zustände | externes Tool |
| Metrik 79: Worst-Case-Execution-Time | externes Tool |
| Metrik 80: Anzahl der Testfälle | externes Tool oder Dokument |
| Metrik 81: Erfolgreiche Testfälle | externes Tool |
| Metrik 82: Testabdeckung | externes Tool |
| Metrik 83: Anzahl der Anforderungen | externes Tool oder Dokument |
| Metrik 84: Anforderungsabdeckung | externes Tool |

Tabelle A.12.: Nicht implementierte Metriken

## A.7. Weitere Ansichten in der grafischen Oberfläche

Abbildung A.4 zeigt beispielhaft die Darstellung der zusätzlichen Informationen einer Metrik. Dabei wird das Modell im oberen Teil als TreeMap dargestellt[5]. Jedes Subsystem wird als Rechteck dargestellt, das wiederum seine Subblöcke enthält. Dadurch entsteht eine rekursive Struktur. Im unteren Teil sind die Detailinformationen der Metrik »Anzahl der Modellklone« sichtbar. Die beiden Gruppen grün markierter Blöcke sind die beiden Teile des Klons, der im unteren Teil ausgewählt ist.

Jede Metrik kann beliebige hierarchische Datenstrukturen ablegen und in der Darstellung aus Abbildung A.4 anzeigen. So speichert z. B. die Metrik »Durchschnittliche Signalpfadlänge« die an den einzelnen Signalpfaden beteiligten Blöcke und die Metrik »Durchschnittliche

[5] `http://treemap.sourceforge.net`

Anzahl der unabhängigen Berechnungspfade« die Blöcke, die jeweils einen unabhängigen Berechnungspfad bilden.

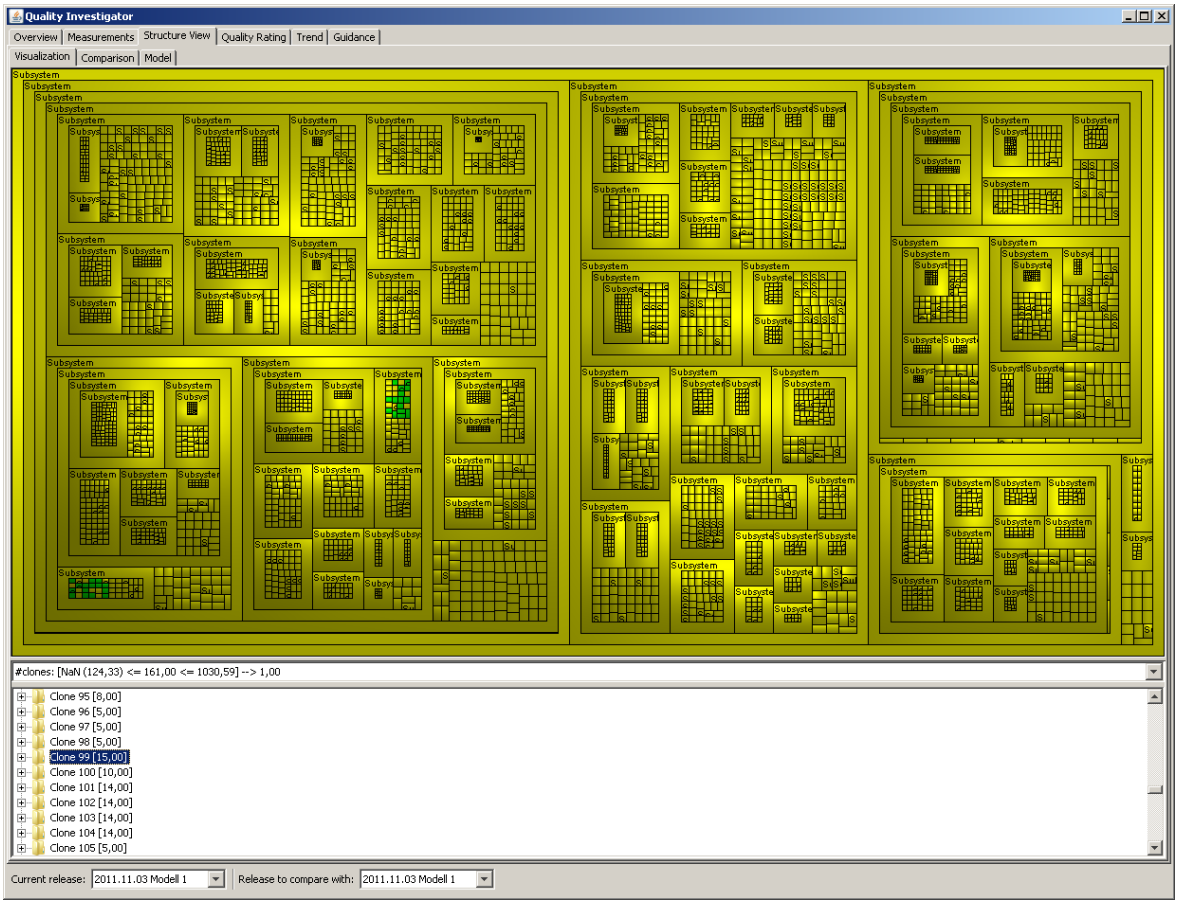



Abbildung A.4.: Darstellung der zusätzlichen Informationen der Metriken im Prototyp

Abbildung A.5 zeigt einen Auszug aus einem Objektdiagramm, das Detailinformationen der Metrik »Anzahl der Modellklone« enthält. Die Klasse `DetailMeasurement` wird in Abschnitt 6.3.2 beschreiben. Gut sichtbar ist die Hierarchie: Das Messergebnis bildet die Wurzel und dessen Kinder enthalten weitere Informationen über die Messung.

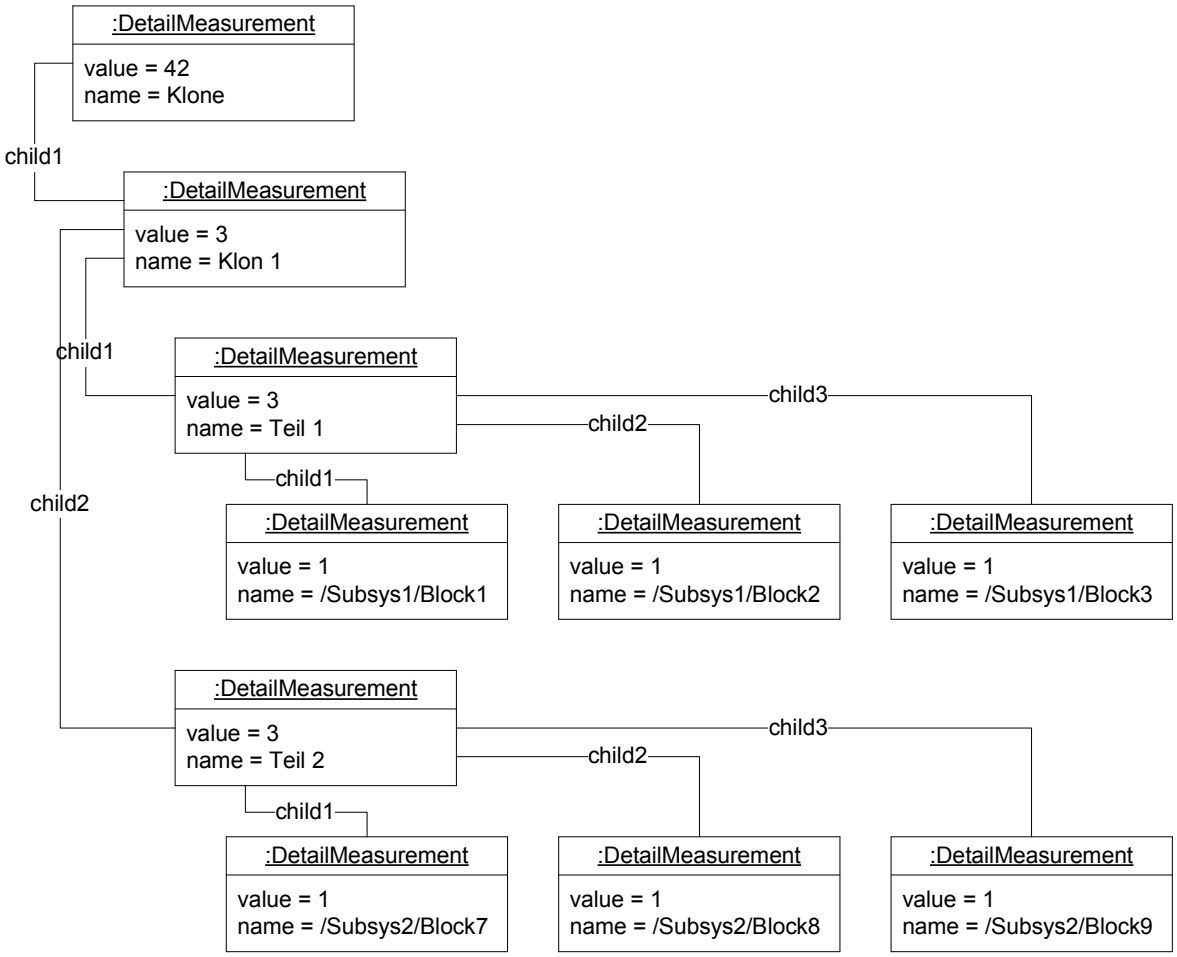



Abbildung A.5.: Detailinformationen der Metrik »Anzahl der Modellklone«

# Literatur


[AFPB10] D. Ahrens, A. Frey, A. Pfeiffer und T. Bertram. „Entwicklung eines objektiven Bewertungsverfahrens für Softwarearchitekturen im Bereich Fahrerassistenz“. In: *Software Engineering*. Bd. 159. LNI. GI, 2010, S. 201–212.

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke und A. Schürr. „MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations“. In: *Model Driven Architecture – Foundations and Applications*. Bd. 4066. Lecture Notes in Computer Science. Springer Verlag, 2006, S. 361–375.

[AM96] F. Brito e. Abreu und W. Melo. „Evaluating the Impact of Object-Oriented Design on Software Quality“. In: *METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics*. Washington, DC, USA: IEEE Computer Society, 1996, S. 90–99.

[BBL76] B. W. Boehm, J. R. Brown und M. Lipow. „Quantitative evaluation of software quality“. In: *Proceedings of the 2nd International Conference on Software Engineering*. San Francisco, California, United States: IEEE Computer Society Press, 1976, S. 592–605.

[BBM96] V. R. Basili, L. C. Briand und W. L. Melo. „A Validation of Object-Oriented Design Metrics as Quality Indicators“. In: Bd. 22. 10. IEEE Transactions on Software Engineering. Piscataway, NJ, USA: IEEE Press, 1996, S. 751–761.

[BCR94] V. R. Basili, G. Caldiera und H. Rombach. In: *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 1994. Kap. Measurement, S. 646–661.

[Bob09] A. E. Bobkowska. In: *Model-driven software development*. Information Science Reference, 2009. Kap. Integrating quality criteria and methods of evaluation for software models, S. 78–93.

[CD06] M. Conrad und H. Dörr. „Einsatz von Modell-basierten Entwicklungstechniken in sicherheitsrelevanten Anwendungen: Herausforderungen und Lösungsansätze“. In: *Proceedings of the Dagstuhl Workshop: Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*. Schloss Dagstuhl, 2006.

[CDFY99] M. Conrad, H. Dörr, I. Fey und A. Yap. „Model–based Generation and Structured Representation of Test Scenarios“. In: Proceedings of the Workshop on Software–Embedded Systems Testing, Gaithersburg, Maryland, USA. 1999.

[CDSS02] M. Conrad, H. Dörr, I. Stürmer und A. Schürr. „Graph Transformations for Model-based Testing“. In: *GI-Lecture Notes in Informatics*. 2002, S. 39–50.

[CFS04] M. Conrad, I. Fey und S. Sadeghipour. „Systematic Model-Based Testing of Embedded Control Software: The MB3T Approach“. In: *ICSE 2004 Workshop on Software Engineering for Automotive Systems, Edinburgh*. 2004.

[CM78] J. P. Cavano und J. A. McCall. „A framework for the measurement of software quality“. In: *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*. 1978, S. 133–139.

[Con+05] M. Conrad, H. Dörr, I. Fey, H. Pohlheim und I. Stürmer. „Guidelines und Reviews in der Modell-basierten Entwicklung von Steuergeräte-Software“. In: *Tagungsband der 2. Tagung Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik*. Expert-Verlag, Berlin, 2005.

[DDM08] B. Dongen, R. Dijkman und J. Mendling. „Measuring Similarity between Business Process Models“. In: *Proceedings of the 20th international conference on Advanced Information Systems Engineering*. CAiSE '08. Springer Verlag, 2008, S. 450–464.

[Dei+07] F. Deißenböck, S. Wagner, M. Pizka, S. Teuchert und J.-F. Girard. „An Activity-Based Quality Model for Maintainability“. In: *Proceedings of the 23rd IEEE International Conference on Software Maintenance*. IEEE CS Press, 2007.

[Dei+11] F. Deißenböck, L. Heinemann, M. Herrmannsdörfer, K. Lochmann und S. Wagner. „The quamoco tool chain for quality modeling and assessment“. In: *ICSE*. 2011, S. 1007–1009.

[Dra96] T. Drake. „Measuring Software Quality: A Case Study“. In: *Computer*. Bd. 29. 11. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996, S. 78–87.

[EFJ10] J. F. Etienne, S. Fechter und E. Juppeaux. „Using Simulink Design Verifier for Proving Behavioral Properties on a Complex Safety Critical System in the Ground Transportation Domain“. In: *Complex Systems Design & Management (CSDM)*. Springer Verlag, 2010, S. 61–72.

[Fer+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing und R. Wilhelm. „Reliable and Precise WCET Determination for a Real-Life Processor“. In: *Embedded Software Workshop*. Bd. 2211. Lake Tahoe, USA, 2001, S. 469–485.

[FHR08] F. Fieber, M. Huhn und B. Rumpe. „Modellqualität als Indikator für Softwarequalität: eine Taxonomie“. In: *Inform.-Spektr.* Bd. 31. 5. 2008, S. 408–424.

[FLS01] K. Frühauf, J. Ludewig und H. Sandmayr. In: *Software-Projektmanagement und –Qualitätssicherung*. Teubner Stuttgart, 2001. Kap. Software-Nutzen und -Kosten, S. 15–17.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[GL06] V. Gruhn und R. Laue. „Complexity metrics for business process models“. In: *9th international conference on business information systems (BIS 2006), volume 85 of Lecture Notes in Informatics*. 2006, S. 1–12.

[GMP02] M. Genero, D. Miranda und M. Piattini. „Defining and Validating Metrics for UML Statechart Diagrams“. In: *Proceedings of QAOOSE*. 2002.

[GMW06] H. Giese, M. Meyer und R. Wagner. „A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink“. In: *Proceedings of the 4th International Fujaba Days 2006, Bayreuth, Germany*. 2006.

[GPC05] M. Genero, M. Piattini und C. Caleron. „A survey of metrics for UML class diagrams“. In: *Journal of Object Technology*. Bd. 4. 2005, S. 59–92.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier Science Ltd, 1977.

[HLW12] W. Hu, T. Löffler und J. Wegener. „Quality Model based on ISO/IEC 9126 for Internal Quality of MATLAB/Simulink/Stateflow Models“. In: *IEEE International Conference on Industrial Technology*. Athen, Griechenland, 2012.

[Hol11] W. Holoch. „Metriken zur Qualitätsmessung von Simulink-Modellen in der modellgetriebenen Softwareentwicklung“. Masterarbeit. Universität Ulm, 2011.

[HR07] G. Hamon und J. Rushby. „An operational semantics for Stateflow“. In: *International Journal on Software Tools for Technology Transfer (STTT)*. Bd. 9. 10.1007/s10009-007-0049-7. Springer Verlag, 2007, S. 447–456.

[Huh+05] M. Huhn, J. C. Braam, M. Harms, M. Horstmann und M. Mutz. „Structured Hardware/Software Development for Enhanced Quality and Safety in Automotive Systems“. In: *6th Conference on Automation, Assistance and Embedded Real Time Platforms for Transportation*. 2005, S. 489–512.

[KA99] T. M. Khoshgoftaar und E. B. Allen. „Predicting Fault-Prone Software Modules in Embedded Systems with Classification Trees“. In: *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*. Washington, DC, USA: IEEE Computer Society, 1999, S. 105.

[KKT11] S. Kemmann, T. Kuhn und M. Trapp. „Extensible and Automated Model-Evaluations with INProVE“. In: *System Analysis and Modeling*. Bd. 6598. Lecture Notes in Computer Science. Springer Verlag, 2011, S. 193–208.

[KLH11] M. Kläs, K. Lochmann und L. Heinemann. „Evaluating a Quality Model for Software Product Assessments – A Case Study“. In: *4. Workshop zur Software-Qualitätsmodellierung und -bewertung (SQMB '11)*. 2011.

[KSJ06] J. Krogstie, G. Sindre und H. Jørgensen. „Process models representing knowledge for action: a revised quality framework“. In: *European Journal of Information Systems*. Bd. 15. 1. Basingstoke, UK, UK: Macmillan Press Ltd., 2006, S. 91–102.

[LC05] C. F. J. Lange und M. R. V. Chaudron. „Managing Model Quality in UML-Based Software Development“. In: *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*. IEEE Computer Society, 2005, S. 7–16.

[LDK04] B. Lilburne, P. Devkota und K. Khan. „Measuring Quality Metrics for Web Applications“. In: *Proceedings of 2004 IRMA International Conference*. 2004.

[LL08] F. Leitner und S. Leue. „Simulink Design Verifier vs. SPIN - A Comparative Case Study“. In: *Participant's Proceedings of FMICS 2008, ERCIM Working Group on Formal Methods for Industrial Critical Systems*. 2008.

[LRS07] E. Legros, T. Rötschke und A. Schürr. „Entwicklung eines Software-Leitstands zur Einhaltung von Modellierungsrichtlinien“. In: *Software Engineering*. 2007.

[MA05] G. Menkhaus und B. Andrich. „Metric Suite for Directing the Failure Mode Analysis of Embedded Software Systems“. In: *Proceedings of ICEIS*. 2005.

[McC76] T. J. McCabe. „A complexity measure“. In: *Proceedings of the 2nd international conference on Software engineering*. ICSE '76. San Francisco, California, United States: IEEE Computer Society Press, 1976, S. 308–320.

[Mil88] E. E. Mills. *Software Metrics - SEI Curriculum Module SEI-CM-12-1.1*. Carnegie Mellon University. 1988.

[ML05] S. Mahmood und R. Lai. „Measuring the Complexity of a UML Component Specification“. In: *QSIC '05: Proceedings of the Fifth International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2005, S. 150–160.

[MPKS00] S. Muthanna, K. Ponnambalam, K. Kontogiannis und B. Stacey. „A Maintainability Model for Industrial Software Systems Using Design Level Metrics“. In: *Proceedings of the 7. WCRE*. Washington, DC, USA: IEEE Computer Society, 2000, S. 248–256.

[MRC07] J. Mendling, H. A. Reijers und J. Cardoso. „What Makes Process Models Understandable?“ In: *Business Process Management*. 2007, S. 48–63.

[MTL78] R. McGill, J. W. Tukey und W. A. Larsen. „Variations of Box Plots“. In: *The American Statistician*. Bd. 32. 1. American Statistical Assoc., 1978, S. 12–16.

[MTM07] T. Mens, G. Taentzer und D. Müller. „Challenges in Model Refactoring“. In: *Proceedings 1st Workshop on Refactoring Tools*. University of Berlin, 2007.

[NSK03] S. Neema, J. Sztipanovits und G. Karsai. „Constraint-Based Design-Space exploration and model synthesis“. In: *In Proceedings of EMSOFT 03, Volume 2855 OF LNCS*. Springer Verlag, 2003, S. 290–305.

[PMCC01] H. C. Purchase, M. McGill, L. Colpoys und D. Carrington. „Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study“. In: *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation - Volume 9*. APVis '01. Sydney, Australia: Australian Computer Society, Inc., 2001, S. 129–137.

[Rau01] A. Rau. „Einsatz von Metriken bei der modellbasierten Software-Entwicklung“. In: *Simulationstechnik - 15. Symposium in Paderborn*. 2001.

[Rau02] A. Rau. „Model-Based Development of Embedded Automotive Control Systems“. Dissertation. Eberhard-Karls-Universität Tübingen, 2002.

[RCLGP08] L. Reynoso, J. A. Cruz-Lemus, M. Genero und M. Piattini. „Formal definition of measures for UML statechart diagrams using OCL“. In: *Proceedings of the 2008 ACM symposium on Applied computing*. SAC '08. Fortaleza, Ceara, Brazil: ACM, 2008, S. 846–847.

[RPRB97] S. Rivard, G. Poirier, L. Raymond und F. Bergeron. „Development of a measure to assess the quality of user-developed applications“. In: *SIGMIS Database*. Bd. 28. 3. New York, USA: ACM, 1997, S. 44–58.

[Sac84] L. Sachs. *Angewandte Statistik. Anwendung statistischer Methoden*. Springer Verlag, 1984.

[Sar06] S. Sarstedt. „Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams“. Dissertation. Universität Ulm, 2006.

[Sch10] J. Scheible. „Ein Framework zur automatisierten Ermittlung der Modellqualität bei eingebetteten Systemen“. In: *Proc. of the Dagstuhl Workshop: Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*. Schloss Dagstuhl, 2010.

[SDP08] I. Stürmer, C. Dziobek und H. Pohlheim. „Modeling Guidelines and Model Analysis Tools in Embedded Automotive Software Development“. In: *Proceedings of the Dagstuhl Workshop: Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*. Schloss Dagstuhl, 2008.

[SK10] J. Scheible und I. Kreuz. „Ein Qualitätsmodell zur automatisierten Ermittlung der Modellqualität bei eingebetteten Systemen“. In: *Proceedings of the 8. GI Workshop Automotive Software Engineering, 2010, Leipzig, Germany*. 2010.

[SP11] J. Scheible und H. Pohlheim. „Automated Model Quality Rating of Embedded Systems“. In: *4. Workshop zur Software-Qualitätsmodellierung und -bewertung (SQMB '11)*. 2011.

[Spe04] C. Spearman. „The Proof and Measurement of Association between Two Things“. In: *The American Journal of Psychology*. University of Illinois Press, 1904.

[SPR10] I. Stürmer, H. Pohlheim und T Rogier. „Berechnung und Visualisierung der Modellkomplexität bei der modellbasierten Entwicklung sicherheits-relevanter Software“. In: *Automotive – Safety & Security*. Shaker Verlag, 2010, S. 69–82.

[Stü+07] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A. Schürr und A. Zündorf. „Das MATE Projekt - visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen“. In: *Proceedings of the Dagstuhl Workshop: Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*. Schloss Dagstuhl, 2007, S. 83–94.

[SVEH07] T. Stahl, M. Völter, S. Efftinge und A. Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag, 2007.

[VALC07] M. F. Van Amstel, C. F. J. Lange und M. R. V. Chaudron. „Four Automated Approaches to Analyze the Quality of UML Sequence Diagrams“. In: *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2007, S. 415–424.

[Van+08] I. Vanderfeesten, H. A. Reijers, J. Mendling, W. M. Aalst und J. Cardoso. „On a Quest for Good Process Models: The Cross-Connectivity Metric“. In: *CAiSE '08: Proceedings of the 20th international conference on Advanced Information Systems Engineering*. Springer Verlag, 2008, S. 480–494.

[XE02] Y. Xie und D. Engler. „Using redundancies to find errors“. In: *SIGSOFT Software Engineering Notes*. Bd. 27. 6. New York, USA: ACM, 2002, S. 51–60.