ulm university universität
uulm

Metriken zur Qualitätsmessung von Simulink-Modellen in der modellgetriebenen Softwareentwicklung

Wolfgang Holoch

Universität Ulm

Fakultät für Ingenieurwissenschaften und Informatik

Institut für Programmiermethodik und Compilerbau

Januar 2011

Master-Thesis im Studiengang Informatik

Danksagung

An dieser Stelle möchte ich den Menschen besonders danken, die mir bei der Erstellung dieser Arbeit sehr geholfen haben:

- Prof. Dr. Helmuth Partsch für seine sehr gute Betreuung, die vielen Ratschläge und die sehr umfangreichen Rückmeldungen, die mir sehr weitergeholfen haben.
- Jan Scheible für seine sehr gute Betreuung, das stets offene Ohr, die umfassenden Rückmeldungen, die vielen anregenden Gespräche und seine große Geduld, wenn es doch wieder einmal etwas länger gedauert hat.
- Philipp Legrum für viele anregende Gespräche und Ideen, die mir oft neue Wege aufgezeigt haben, sowie die vielfältigen Rückmeldungen zu dieser Arbeit.
- Carina Schiller für ihre vielen Rückmeldungen, die Unterstützung, Rückhalt und Geduld, während dem erstellen dieser Arbeit.
- Daniel Ott für seine ausführlichen und umfassenden Rückmeldungen, sowie den aufmunternden Unterhaltungen am Morgen.

Ulm, den 1. Februar 2011 *Wolfgang Holoch*

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1
1.2 Zielsetzung 2
1.3 Gliederung der Arbeit 2

2 Grundlagen **5**

2.1 MATLAB 5
2.1.1 Simulink 5
2.1.2 TargetLink 8
2.2 Der Goal/Question/Metric-Ansatz 8
2.3 Metriken 9

3 Stand der Technik **11**

3.1 Softwaremetriken 11
3.2 Modellmetriken 12
3.2.1 Metriken für UML-Modelle 12
3.2.2 Metriken für Geschäftsprozessmodelle 14
3.2.3 Metriken für Simulink-Modelle 14
3.3 Weitere Ansätze für Metriken 15
3.4 Fazit 16

4 Simulink-Modelle **17**

4.1 Beispiel 17
4.2 Bestandteile von Simulink-Modellen 19
4.2.1 Simulink-Blöcke 19
In- und Outports 19
Subsysteme 19
Blocktypen 20
4.2.2 Kanten 26

4.2.3 Signale . . . 26
Definition: Signal, Signalquelle, Signalträger, Signalsenke . 28
Umsetzung . . . 30
4.3 Simulink-Modelle als Graphen . . . 34
4.4 Modell-Exporter . . . 34

5 Metriken – Theorie und Umsetzung für Simulink-Modelle 37
5.1 Identifikation von Metriken zur Qualitätsmessung . . . 37
5.2 Metriken für Simulink-Modelle . . . 40
5.2.1 Messung der Dokumentation . . . 41
Durchschnittliche Kommentarlänge pro Block . . . 41
5.2.2 Messung der Modularisierung . . . 42
Unabhängige Pfade . . . 42
5.2.3 Messung der Wartbarkeit . . . 50
Klonerkennung . . . 50
Blockinstabilität . . . 63
5.2.4 Messung der Effizienz . . . 64
Verfallene Ergebnisse . . . 67
5.2.5 Messung der Komplexität . . . 70
Anzahl Zyklen . . . 73
Durchschnittliche Signallänge . . . 77
Slicing . . . 80
5.2.6 Fazit . . . 83
5.3 Nicht übertragbare Metriken . . . 84
5.3.1 Softwaremetriken für generierten Quellcode . . . 84
5.3.2 Gesetz von Demeter . . . 84
5.3.3 Nebenläufigkeit . . . 85

6 Zusammenfassung und Ausblick 87
6.1 Zusammenfassung . . . 87
6.2 Ausblick . . . 89

Literatur 91

Abbildungsverzeichnis

2.1 Die Benutzeroberfläche von MATLAB. 6
2.2 Die Benutzeroberfläche von MATLAB Simulink. 7
2.3 Zusammenhang zwischen Zielen, Fragen und Metriken im Goal/ Question/Metric-Prozess. 9

4.1 Bestandteile eines Simulink-Blocks. 17
4.2 Beispiel für ein Simulink-Modell. 18
4.3 Beispiel für eine nicht erlaubte Kantenverbindung. 26
4.4 Beispiel für ein Signal in einem Simulink-Modell. 27

5.1 Beispiel für ein Subsystem mit zwei unabhängigen Pfaden. 43
5.2 Ein unerkanntes Klonpaar. 59
5.3 Beginn der Klonsuche. 60
5.4 Der erste erkannte Klon. 61
5.5 Erneute Klonsuche. 61
5.6 Der zweite erkannte Klon. 62
5.7 Das Endergebnis der Klonsuche. 62
5.8 Beispielmodell für die Berechnung der Blockinstabilität. 64
5.9 Ein Beispiel für nicht genutzte Berechnungen. 67
5.10 Ausschnitt eines Modells mit mehreren Zyklen. 76

Tabellenverzeichnis

4.1 Diskrete-Blöcke 20
4.2 Operator-Blöcke 20
4.3 Lookup-Tabellen 21
4.4 Mathematische Operationen 22
4.5 Port- und Subsystem-Blöcke 23
4.6 Routing-Blöcke 24
4.7 Routing-Blöcke Fortsetzung 25
4.8 Senken 25
4.9 Quellen 25
4.10 Sonstige-Blöcke 26
4.11 Signalquellen 28
4.12 Signalträger 29
4.13 Hilfsfunktionen des Grundalgorithmus zur Berechnung der Signale 31
4.14 Hilfsfunktionen der Funktion BerechneSignale 33

5.1 Existierende Metriken für Simulink-Modelle 40
5.2 Definition von Start- und Endblöcken 43
5.3 Ausnahmen von Start- und Endblöcken 44
5.4 Ablauf der Vereinigungen 45
5.5 Hilfsfunktionen des Grundalgorithmus zur Berechnung von unabhängigen Pfaden 46
5.6 Hilfsfunktionen der Funktion ErreichbareEndblöcke 48
5.7 Äquivalenzeigenschaften von geklonten Blöcken 52
5.8 Hilfsfunktionen des Grundalgorithmus zur Klonerkennung 54
5.9 Hilfsfunktionen der Funktion BerechneKlon 57
5.10 Hilfsfunktionen der Funktion Vereinige 59
5.11 Hilfsfunktionen des Grundalgorithmus zur Bestimmung der Blockinstabilität 66
5.12 Begriffsdefinitionen 68

5.13 Signal-Konnektor Elemente 69
5.14 Hilfsfunktionen des Grundalgorithmus zur Bestimmung der verfallenen Ergebnisse 69
5.15 Hilfsfunktionen der Funktion MarkiereAktiveSignalKonnektoren . . 72
5.16 Hilfsfunktionen des Grundalgorithmus zur Zyklensuche 74
5.17 Hilfsfunktionen der Funktion FindeZyklen 75
5.18 Verlauf der Rekursion während der Zyklensuche 76
5.19 Hilfsfunktionen des Grundalgorithmus für die durchschnittlichen Signallänge 78
5.20 Hilfsfunktionen des Grundalgorithmus für das Slicing 82
5.21 Hilfsfunktionen der Funktion BerechneSignalVorfahren 83
5.22 Zusammenfassung aller implementierten Metriken 83

6.1 Auflistung der Simulink-Metriken 89

Liste der Algorithmen

1 Der Grundalgorithmus zur Berechnung der Signale 30

2 Funktion BerechneSignale 32

3 Der Grundalgorithmus zur Berechnung von unabhängigen Pfaden . 46

4 Funktion ErreichbareEndblöcke 47

5 Funktion Vereinige 49

6 Der Grundalgorithmus der Klonerkennung 54

7 Funktion BerechneKlon 56

8 Funktion Vereinige 58

9 Grundalgorithmus zur Bestimmung der Blockinstabilität 65

10 Der Grundalgorithmus zur Bestimmung der verfallenen Ergebnisse 69

11 Funktion MarkiereAktiveSignalKonnektoren 71

12 Der Grundalgorithmus der Zyklensuche 74

13 Funktion FindeZyklen 75

14 Der Grundalgorithmus für die durchschnittlichen Signallänge 78

15 Der Grundalgorithmus des Slicing 81

16 Funktion BerechneSignalVorfahren 82

1 Einleitung

Diese Einleitung bietet eine informelle Einführung in das Thema der vorliegenden Arbeit. Am Anfang dieses Kapitels wird die Motivation zur Erstellung dieser Arbeit erläutert. Danach werden die Ziele identifiziert, die in dieser Arbeit erreicht werden sollen. Anschließend wird die Gliederung dieser Arbeit vorgestellt.

1.1 Motivation

Nach Schätzungen sind im Automobilbereich 80% aller Innovationen durch Elektronik getrieben, wobei hiervon wiederum 90% durch Softwarefunktionen bestimmt werden [Con+05, S. 3]. Aufgrund dessen steigt der Anteil von elektronischen Systemen und damit auch der Anteil von Software stetig an.

Um die Komplexität der steigenden Anzahl von Softwaresystemen in den Griff zu bekommen, findet ein zunehmender Einsatz von modellbasierter Entwicklung statt. Ein hierfür häufig eingesetztes Werkzeug ist MATLAB Simulink.

Da nicht nur die Anzahl, sondern auch die Größe der Softwaresysteme stetig steigt, wird das Bewerten der Qualität dieser Systeme immer schwieriger. In der Arbeit [Sch10, S. 1] wird der Umfang von Simulink-Modellen bei der Daimler AG so beschrieben:

> „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."

Diese Modelle können aufgrund ihrer Größe und Komplexität nicht mehr von einer Person alleine verstanden werden.

Ursprünglich wurde Simulink nicht für die modellbasierte Entwicklung entworfen, sondern wurde zur Simulation von technischen Systemen entwickelt.

Aufgrund der zunehmenden Bedeutung von Software im Auto und des Umfangs der Modelle für die Daimler AG, besteht ein Interesse, die Qualität der Modelle zu bewerten, so wie es bei der konventionellen Softwareentwicklung üblich ist.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Analyse, Adaption, Konzeption und Implementierung von Metriken für MATLAB Simulink-Modelle. Diese Metriken sollen Eigenschaften von Modellen messen, um daraus Rückschlüsse auf deren Modellqualität ziehen zu können.

Analyse Bei der Analyse werden existierende Software- und Modellmetriken untersucht, ob diese geeignet sind zur Verwendung bei MATLAB Simulink-Modellen. Die Metriken werden dahingehend bewertet, ob deren Verwendung im gegebenen Kontext Sinn macht und wie diese gegebenenfalls adaptiert werden können.

Adaption Bei der Adaption werden untersuchte und geeignete Metriken für die Verwendung bei Simulink-Modellen angepasst.

Konzeption Bei der Konzeption sollen systematisch neue Metriken entwickelt werden, die Bereiche untersuchen, die nicht durch existierende Metriken abgedeckt sind, um Qualitätsaspekte von Simulink-Modellen messen zu können.

Implementierung Die ausgewählten Metriken werden in einem existierenden Java Framework implementiert und getestet. Dieses Framework wird von Jan Scheible im Rahmen seiner Promotion entwickelt [Sch10; SK10]. Dieses Framework lädt Simulink-Modelle in Java-Objekte, ermöglicht einen einfachen Zugriff auf das Modell und verwaltet die Metriken zur Bewertung der Modelle.

1.3 Gliederung der Arbeit

In Kapitel 2 werden die notwendigen Grundlagen erklärt, die in dieser Arbeit verwendet werden. So wird erklärt was MATLAB, Simulink sowie TargetLink sind.

Des weiteren wird darauf eingegangen, was eine Metrik und der *Goal/Question/Metric*-Ansatz ist. Zum Schluss wird das in dieser Arbeit verwendete Framework zur Ausführung von Metriken kurz vorgestellt.

In Kapitel 3 wird der aktuelle Stand der Technik vorgestellt. Hierbei wird ein Überblick über die Bereiche Softwaremetriken und Modellmetriken gegeben. Sowie weitere Möglichkeiten für die Adaption von Metriken besprochen.

In Kapitel 4 werden Simulink-Modelle und dessen Elemente vorgestellt und deren Aufbau soweit erläutert wie es nötig ist, um diese Arbeit zu verstehen.

In Kapitel 5 wird der *Goal/Question/Metric-Ansatz* verwendet, um Metriken herzuleiten, welche die Qualität von Simulink-Modellen messen. Hierfür werden messbare Kriterien gesucht, um den Begriff *Qualität* zu beschreiben. Ausgehend von diesen Kriterien werden schließlich einzelne Metriken entwickelt und adaptiert, um diese Kriterien messen und bewerten zu können.

Kapitel 6 fasst die Ergebnisse dieser Arbeit zusammen und liefert einen Ausblick auf mögliche weiterführende Arbeiten.

2 Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die zum Verständnis dieser Arbeit benötigt werden. Zuerst wird erläutert was *MATLAB* und dessen Zusatzprodukte *Simulink* und *TargetLink* sind. Danach wird der Begriff einer *Metrik* definiert und anschließend der *Goal/Question/Metric*-Ansatz erklärt.

2.1 MATLAB

MATLAB ist ein Programm zur interaktiven Entwicklung von Algorithmen, numerischen Berechnungen und der Visualisierung und Analyse von Daten. Außerdem stehen mathematische Funktionen für die Bereiche lineare Algebra, Statistik, Fourieranalyse, Filtern, Optimieren und die numerische Integration bereit [Mat10a]. Dadurch erlaubt MATLAB das einfache Berechnen von Daten und die schnelle Entwicklung von Algorithmen.

In Abbildung 2.1 ist die Benutzeroberfläche von MATLAB dargestellt. Zu sehen ist eine Beispielsitzung mit einem Satz von Daten und deren Fourier-Transformation.

MATLAB wurde in dieser Arbeit als Grundlage für Simulink (siehe Kapitel 2.1.1), sowie zum Export von Simulink-Modellen benötigt (siehe Kapitel 4.4).

2.1.1 Simulink

Simulink ist ein Zusatzprodukt zu MATLAB für die Regelungstechnik. Dabei erweitert Simulink MATLAB um eine interaktive grafische Entwicklungsumgebung, mit der es möglich ist Systeme zu modellieren, zu simulieren und zu testen [Mat10b].

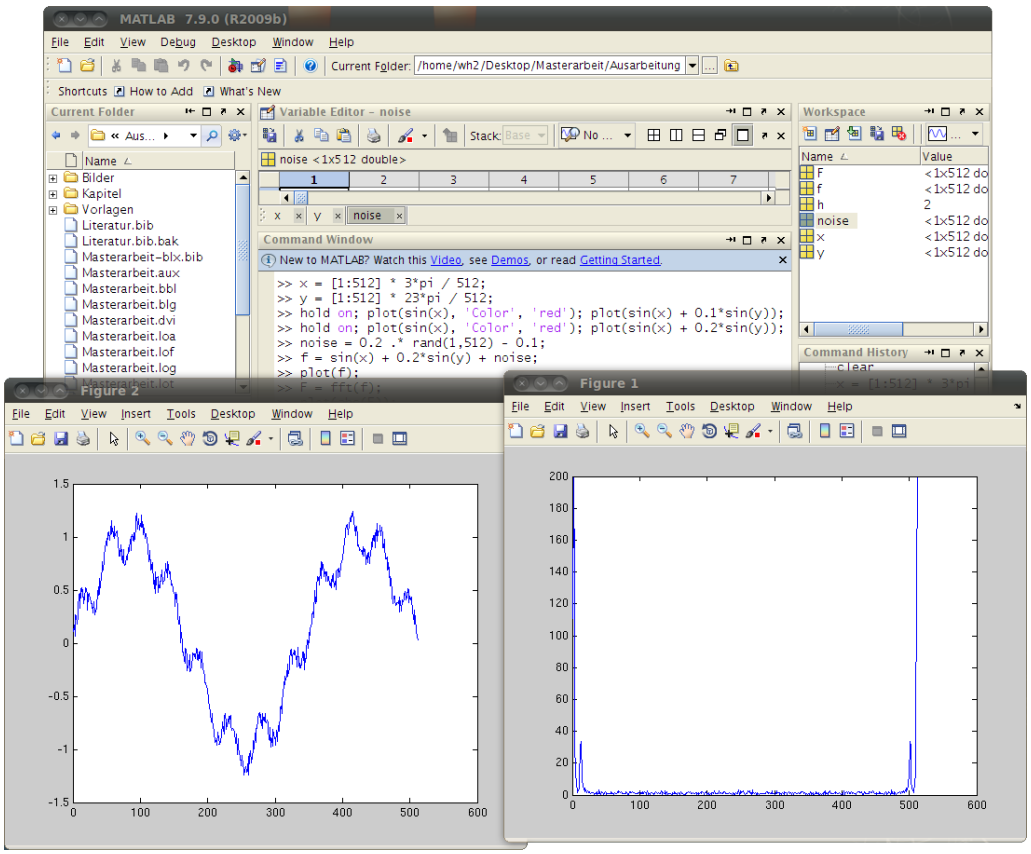


Abbildung 2.1: Die Benutzeroberfläche von MATLAB.

Im Kontext dieser Arbeit wird Simulink für die Softwareentwicklung von Steuergeräten genutzt. Da Simulink jedoch keine Möglichkeiten für die Quelltexterzeugung bietet, wird hierfür die Software TargetLink verwendet (siehe Kapitel 2.1.2).

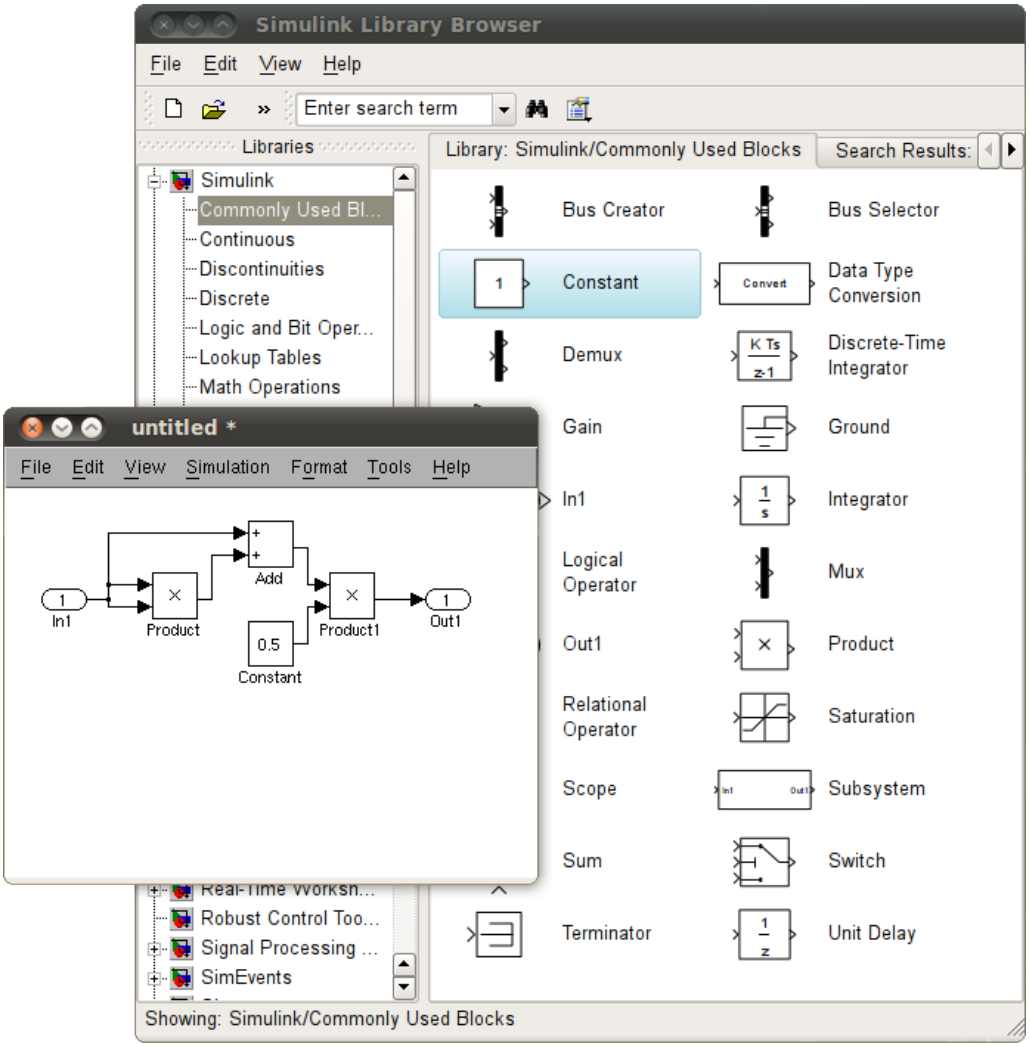


Abbildung 2.2: Die Benutzeroberfläche von MATLAB Simulink.

In Abbildung 2.2 ist die Benutzeroberfläche von Simulink dargestellt, mit einem Modell das die Formel $\frac{n^2+n}{2}$ berechnet.

Eine ausführlichere Einführung in Simulink, wird in Kapitel 4 gegeben.

2.1.2 TargetLink

TargetLink ist eine Software der Firma dSPACE, für die Programmcodegenerierung aus Simulink-Modellen. Um dies zu erreichen führt TargetLink zusätzliche Blöcke für Simulink ein die Informationen für die Programmcodegenerierung enthalten [dSP10].

Einige der TargetLink-Blöcke ersetzen, andere ergänzen die existierenden Simulink-Blöcke. Im Kontext dieser Arbeit sind nur die zusätzlichen Blöcke interessant. Ein TargetLink-Block, der einen Simulink-Block ersetzt hat die gleichen Eigenschaften wie der korrespondierende Simulink-Block, inklusive Informationen, die für die Programmcodegenerierung benötigt werden.

Ein TargetLink-Block der einen bestehenden Simulink-Block ergänzt, enthält nur Informationen für die Programmcodegenerierung. Solch ein TargetLink-Block verändert somit nicht die Funktionalität eines Simulink-Modells.

Wenn ein TargetLink-Block von spezieller Bedeutung ist, wird an der entsprechenden Stelle explizit darauf hingewiesen.

TargetLink generiert Quelltext für die Programmiersprache *C*.

2.2 Der Goal/Question/Metric-Ansatz

Der *Goal/Question/Metric*-Ansatz ist ein Mechanismus für das Definieren und Interpretieren von Betriebs- und Softwarezielen. Der gesamte Ansatz wird in der Arbeit [Bas92] beschrieben.

Der Goal/Question/Metric-Prozess ist ein *Top-Down* Prozess und ist definiert über folgende Arbeitsschritte:

1. Definition von Firmen-, Abteilungs- und Projektzielen für die Produktivität und die Qualitätskontrolle.
2. Erstellen von Fragen, welche die definierten Ziele so gut wie möglich bestimmen.
3. Spezifizierung der Messungen, die benötigt werden um die Fragen zu beantworten.

4. Entwicklung von Techniken zur Sammlung der benötigten Daten.
5. Sammlung, Validierung und Analyse der Daten in Echtzeit, um Rückmeldungen an das Projekt geben zu können.

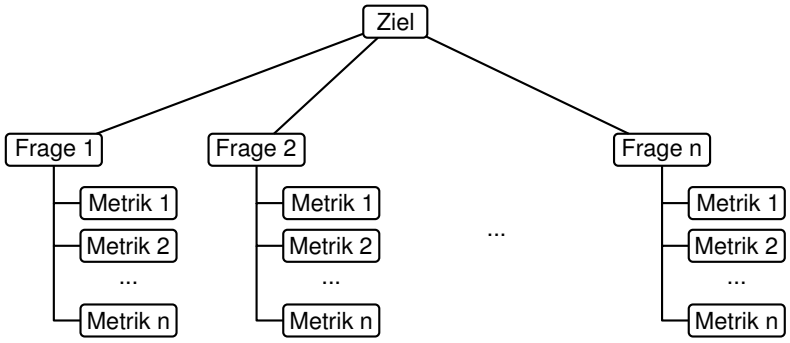


Abbildung 2.3: Zusammenhang zwischen Zielen, Fragen und Metriken im Goal/Question/Metric-Prozess.

Durch das Vorgehen bei dem Goal/Question/Metric-Ansatz ergibt sich eine Baumstruktur der Höhe zwei. Dabei steht das Ziel an der Wurzel und die daraus hergeleiteten Fragen sind die direkten Kindknoten unter dem Wurzel Knoten. Die einzelnen Blätter sind die Metriken, die aus den Fragen hergeleitet werden.

Der Goal/Question/Metric-Ansatz wird in dieser Arbeit benutzt, um den Begriff Qualität in einzelne messbare Kriterien zu zerlegen. Anschließend wird dieser Ansatz dafür verwendet, um Metriken für diese Kriterien zu identifizieren.

2.3 Metriken

Der Begriff *Metriken*, so wie er in dieser Arbeit verwendet wird, lässt sich unterteilen in die Bereiche Software- und Modellmetriken.

Eine *Softwaremetrik* ist ein Verfahren das Quelltext auf einen numerischen Wert abbildet. Durch die Interpretation des errechneten Wertes können Rückschlüsse auf den Quelltext gezogen werden. Populäre Beispiele für Softwaremetriken sind:

Anzahl der Quelltextzeilen Diese Metrik (auch *Lines of Code* genannt) zählt die Anzahl von Quelltextzeilen in einem Programm. Trotz ihrer Einfachheit ist diese Metrik sehr populär und wird viel genutzt.

Halstead-Metriken Die *Halstead-Metriken* sind zwei Metriken die versuchen die Komplexität eines Programms zu fassen [Hal77].

Zyklomatische Komplexität Diese Metrik versucht die Komplexität eines Programms über dessen Kontrollflussgraphen zu messen [McC76].

Analog wird der Begriff *Modellmetrik* in dieser Arbeit für ein Verfahren verwendet, das ein Simulink-Modell auf einen numerischen Wert abbildet.

3 Stand der Technik

In diesem Kapitel wird der aktuelle Stand der Forschung verschiedener Bereiche vorgestellt, in denen Metriken entwickelt wurden.

3.1 Softwaremetriken

Im folgenden Absatz wird ein kurzer Überblick über die verschiedenen Bereiche von Softwaremetriken gegeben. Eine kompakte Zusammenfassung über die historische Entwicklung der Softwaremetriken gibt der Artikel [FN99].

Zählmetriken sind eine der populärsten Vertreter von Softwaremetriken. Ein Beispiel hierfür ist die Anzahl der Programmzeilen (*Lines of Code*). Durch ihre einfache Funktionsweise sind solche Metriken leicht zu berechnen. Diese Zählmetriken besitzen jedoch auch Nachteile. Es ist schwierig sie zu interpretieren und Ergebnisse von Projekten die mit anderen Programmiersprachen geschrieben wurden lassen sich nicht direkt vergleichen.

Um die Problematik von verschiedenen Programmiersprachen zu umgehen gibt es auch Ansätze, die den Quelltext abstrahiert betrachten, um über diesen eine Aussage treffen zu können. Ein Beispiel hierfür sind die Metriken von Halstead [Hal77], McCabe [McC76] und Function Points [Alb79].

Ein weiterer großer Bereich von Softwaremetriken sind Metriken für spezielle Paradigmen. Besonders oft wird das objektorientierte Programmierparadigma verwendet. Ein Beispiel für diesen Ansatz ist die Arbeit von Chidamber und Kemerer [CK94].

Ob eine Softwaremetrik für Simulink-Modelle adaptiert werden kann, hängt immer vom Einzelfall ab und lässt sich nicht allgemein für alle Softwaremetriken entscheiden.

3.2 Modellmetriken

In diesem Abschnitt wird der aktuelle Stand der Forschung im Bereich Modellmetriken aufgezeigt. Dafür werden verschiedene Gebiete und Projekte, in denen Modellmetriken verwendet werden, kurz vorgestellt.

Im Folgenden betrachtet der Autor die Bereiche UML-, Geschäftsprozess- und Simulink-Modelle. Dieser Überblick hat keinen Anspruch auf Vollständigkeit und könnte um weitere Bereiche und Projekte erweitert werden. Die hier aufgeführten Bereiche wurden ausgewählt, da sie Thema aktueller Forschung sind und schon genauer untersucht wurden.

3.2.1 Metriken für UML-Modelle

Das allgemeine Ziel für Metriken auf UML-Modellen, ist die Bewertung von Projekten in einem möglichst frühen Entwicklungsstand. Dabei ist ein oft benutzter Ansatz die Verwendung von bewährten Metriken für objektorientierte Sprachen, wie z. B. die Metriken von Chidamber und Kemerer [CK94].

Diese Ansätze nutzen die Zusammenhänge zwischen Objektorientierten Sprachen und UML-Diagrammen (besonders den Klassendiagrammen), um Metriken auf UML-Diagramme zu übertragen. Außerdem ist im Bereich UML-Metriken zu beobachten, dass hauptsächlich Strukturdiagramme untersucht werden. Auch konzentrieren sich die meisten der entwickelten Metriken auf wenige Diagrammarten. Ein exemplarisches Beispiele für dieses Vorgehen ist die Arbeit [TC02]. Beispiele für Objektorientierte Metriken die auf UML-Klassendiagramme übertragen werden können, sind [CK94]:

Gewichtete Methoden pro Klasse Diese Metrik berechnet die Komplexität einer Klasse, über eine Bewertung der zugehörigen Methoden.

Tiefe des Vererbungsbaum Diese Metrik berechnet für jede Klasse von wie vielen anderen Klassen sie erbt. Damit kann der Einfluss von anderen Klassen auf die zu untersuchende Klasse bewertet werden.

Anzahl der direkten Subklassen Diese Metrik berechnet die Anzahl von Klassen, die direkt von der zu untersuchenden Klasse erben. Mit dieser Metrik kann der potentielle Einfluss auf andere Klassen gemessen werden.

Es existieren auch Metriken für Verhaltensdiagramme. Der Autor ist der Meinung, dass Metriken für Verhaltensdiagramme besser für Simulink-Modelle geeignet sind, wie Metriken für Strukturdiagramme. Dies liegt an der fehlenden Objektorientierung der Simulink-Modelle, welche sie schlecht vergleichbar mit Strukturdiagrammen macht. Da ein Simulink-Modell einen Programmfluss darstellt, sind diese Modelle eher den Aktivitätsdiagrammen ähnlich. Eine Arbeit, die sich mit Metriken für Verhaltensdiagramme beschäftigt ist [KB02].

Ein weiterer Ansatz beschäftigt sich mit der Entwicklung von neuen Metriken speziell für UML-Modelle. Hierfür liefert die Arbeit [ES03] einige Beispiele:

Informative Größe Diese Metrik berechnet den Informationsgehalt eines UML-Diagramms. Dafür wird jedes Element mit einem Wert auf dessen Informationsgehalt beurteilt und anschließend werden alle Werte aufsummiert.

Visuelle Größe Diese Metrik berechnet die Ausdehnung eines UML-Diagramms. Hieraus können Rückschlüsse über die Wartbarkeit eines Diagramms gezogen werden.

Komplexität Diese Metrik bewertet wie überschaubar und verständlich ein Klassendiagramm ist. Die Komplexität wird durch die Anzahl der Beziehungen zwischen Klassen und Schnittstellen berechnet.

Eine andere Herangehensweise ist das Messen von verschiedenen Repräsentationen (z. B. UML-Modell und entwickelter Quellcode) mit Metriken, die anschließend miteinander verglichen werden. Diese Vorgehensweise wird in der Arbeit [MP07] verwendet.

Die meisten Metriken für Strukturdiagramme auf UML-Modellen dürften für Simulink-Modelle ungeeignet sein. Dies liegt daran, dass die meisten Strukturdiagrammmetriken einen starken Bezug zur Objektorientierung haben. Da Simulink-Modellen dieser Bezug fehlt, ist es schwierig, solche Metriken zu übertragen. Jedoch können Metriken für Verhaltensdiagramme interessant für die Adaption auf Simulink-Modelle sein.

3.2.2 Metriken für Geschäftsprozessmodelle

Ein *Geschäftsprozessmodell* ist eine Abbildung von Geschäfts- oder Arbeitsabläufen in einem Diagramm [AHW03]. Diese Diagramme sind in Aufbau und Verhalten einem Flussdiagramm sehr ähnlich.

Viele Metriken für Geschäftsprozessmodelle können aus anderen Bereichen adaptiert werden. So lassen sich viele bewährte Metriken auf Geschäftsprozessmodellen einsetzen [Car+06] (LoC, zyklomatische Komplexität [McC76], etc.).

Ein Beispiel, für die Adaption von Metriken aus anderen Bereichen, sind die Bemühungen, Metriken für Graphen auf Geschäftsprozessmodelle zu adaptieren [Men08]. Einige Beispiele für Metriken auf Geschäftsprozessmodellen, die auch für Simulink-Modelle interessant sein können, sind:

Größe Die Größe eines Graphen. Diese Metrik bestimmt die Anzahl der Knoten.

Dichte Die Dichte eines Graphen. Diese Metrik misst den Grad der Vernetzung.

Durchmesser Der Durchmesser eines Graphen. Dies ist der größte Abstand zwischen zwei Knoten im Graph.

Nebenläufigkeit Die Anzahl von parallelen Arbeitsabläufen in einem Modell.

Im Bereich der Geschäftsprozessmodelle werden auch neue Metriken entwickelt, wie z. B. die Cross-Conectivity Metrik [Van+08]. Diese Metrik soll die Verständlichkeit von Geschäftsprozessmodellen messen. Weitere neue Metriken zur Messung der strukturellen Komplexität von Geschäftsprozessmodellen, werden in der Arbeit [LK01] entwickelt.

Zusammenfassend lässt sich über die Metriken für Geschäftsprozessmodelle sagen, dass diese viele interessante Ansätze liefern. Durch ihre Ähnlichkeit zu Simulink-Modellen (beide Modellarten sind den Ablaufdiagrammen sehr ähnlich) ist die Adaption vieler Metriken möglich.

3.2.3 Metriken für Simulink-Modelle

In diesem Abschnitt werden die wichtigsten Werkzeuge kurz vorgestellt, die Metriken verwenden, um die Qualität von Simulink-Modellen zu bewerten.

Das Programm *SMMC* (Simulink Model Metrics Calculator) [MA05] implementiert bekannte Metriken, um diese auf Simulink-Modelle anzuwenden. Die implementierten Metriken sind:

Zyklomatische Komplexität Eine Adaption der Komplexitätsmetrik von McCabe [McC76].

Blockinstabilität Diese Metrik berechnet, wie wahrscheinlich ein Block das Ziel einer Änderung ist.

Systeminstabilität Diese Metrik bewertet, wie wichtig ein Block in einem Modell ist.

Das Werkzeug *INProVE* (Indicator-based Non-functional Property-oriented eValuation and Evolution of software design models) [KKT10] wählt einen anderen Ansatz:

Dieses Werkzeug verwendet sogenannte *Indikatoren* um Qualitätsmerkmale von Modellen zu errechnen. Ein Indikator wird durch eine Pipeline realisiert. In einer Pipeline werden einzelne Metriken errechnet, zusammengefasst und anschließend klassifiziert. Ein Indikator kann in die folgenden Kategorien eingeteilt werden:

Pattern Suche Der Indikator zählt die Vorkommen eines Pattern und bewertet diese.

Logische Inferenz Ein Indikator errechnet aus den Ergebnissen einzelner Metriken ein neues Ergebnis.

Funktionale Kombination Ähnlich wie die *logische Inferenz*, jedoch ist die Kombination der Metriken auf numerische Operationen beschränkt.

Das Programm *M-XRAY* [SPR10] verwendet ebenfalls bewährte Metriken, um diese auf Simulink-Modelle anzuwenden. Hierfür wird die Metrik *Halstead-Volumen* [Hal77] verwendet, um die Komplexität von Simulink-Modelle zu messen.

3.3 Weitere Ansätze für Metriken

Ein weiterer Ansatz in der Entwicklung von Metriken ist die Adaption von Techniken aus anderen Bereichen der Informatik und die Interpretation oder Weiter-

verwendung deren Ergebnisse. Ein Beispiel für dieses Vorgehen ist die Arbeit [PKW06]. Dort werden verschiedene Metriken entwickelt, die mit Daten arbeiten, welche aus *Slicing* Informationen gewonnen werden. *Slicing* ist eine Technik, die aus der Programmanalyse stammt [Wei81].

3.4 Fazit

Zusammenfassend lässt sich sagen, dass je allgemeiner eine Metrik ist, sie desto leichter für Simulink-Modelle adaptierbar ist. Metriken die für spezielle Paradigmen oder Sprachen entwickelt wurden (z. B. Metriken für objektorientierte Sprachen oder Modelle) lassen sich hingegen nur schwer adaptieren, da diese spezielle Eigenschaften ausnützen, die in Simulink-Modellen nicht existieren.

4 Simulink-Modelle

In diesem Kapitel wird eine kurze Einführung in Simulink-Modelle gegeben. Dabei werden die Aspekte von Simulink erklärt, die für das Verständnis dieser Arbeit wichtig sind.

Auf den folgenden Seiten werden zuerst die einzelnen Bestandteile von Simulink-Modellen erläutert und wie solch ein Modell aufgebaut ist. Danach werden die Unterschiede und Gemeinsamkeiten von Simulink-Modellen und Graphen diskutiert.

4.1 Beispiel

In diesem Unterkapitel werden grundlegende Bestandteile von Simulink-Modellen anhand von Beispielen vorgestellt.

In Abbildung 4.1 werden die einzelnen Bestandteile eines Blocks erläutert. Die

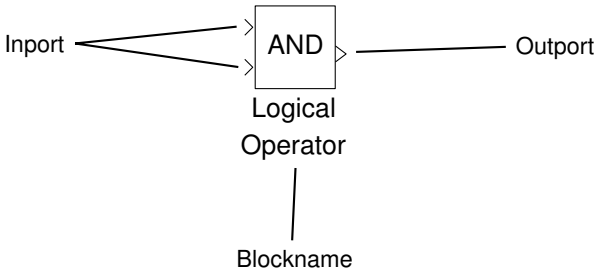


Abbildung 4.1: Bestandteile eines Simulink-Blocks.

Inports sind die Eingabeschnittstellen eines Blocks. Über diese Inports erhält ein Block Werte/Informationen die er verarbeitet und das Ergebnis über seine Outports zurück gibt. Was genau ein Block mit seinen Eingaben macht, hängt

vom jeweiligen Blocktypen ab. Die wichtigsten Blocktypen werden in Kapitel 4.2.1 näher beschrieben.

Zudem kann jeder Block einen Namen besitzen. Dieser Blockname dient nur dazu die Verständlichkeit eines Modells zu erhöhen und verändert nicht dessen Logik.

Da ein einzelner Block noch nicht viel machen kann, können mehrere Blöcke zu einem Modell/System zusammen geschaltet werden. Wie so etwas aussehen kann, ist in Abbildung 4.2 dargestellt. Jedes Simulink-Modell enthält Blöcke,

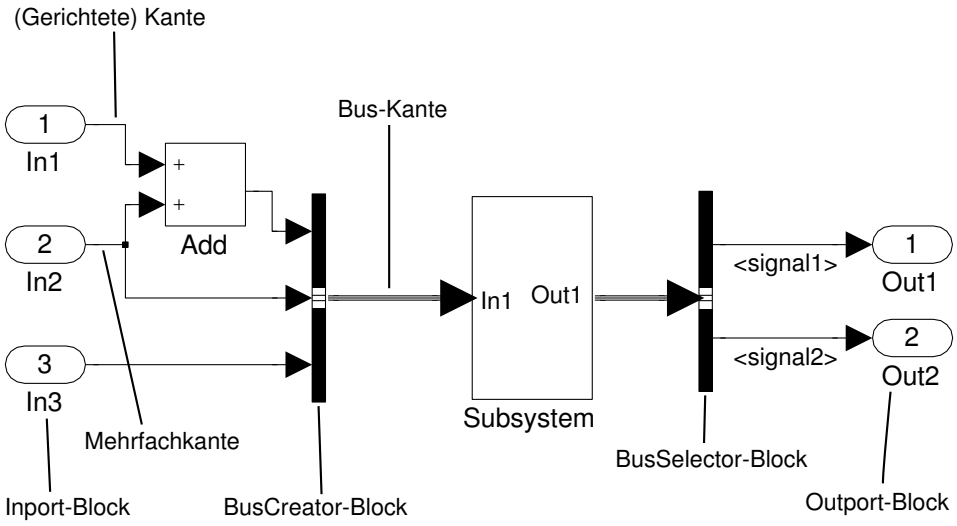


Abbildung 4.2: Beispiel für ein Simulink-Modell.

die über gerichtete Kanten miteinander verbunden sind. Des weiteren kann ein Simulink-Modell (System) beliebig viele Subsystem-Blöcke besitzen.

Ein Subsystem-Block ist ein Block der selbst ein Simulink-Modell (Subsystem) enthält. Dieses Subsystem kann wiederum beliebige Blöcke enthalten. Subsysteme dienen damit als Mittel zur Abstraktion und Kapselung von Modellteilen.

Die In- und Outports eines Systems werden in einem System als Inport- und Outport-Blöcke dargestellt.

4.2 Bestandteile von Simulink-Modellen

In den folgenden Abschnitten werden Bestandteile von Simulink-Modellen näher erläutert, die für das Verständnis für diese Arbeit benötigt werden.

4.2.1 Simulink-Blöcke

In diesem Unterkapitel werden Blöcke und genauer vorgestellt, sowie erklärt was In- und Outports von Blöcken sind.

In- und Outports

Jeder Simulink-Block hat eine gewisse Anzahl von *In-* und *Outports*. Hierbei ist zu beachten, dass es Blöcke gibt, die keine Inports (z. B. Constant-Block), keine Outports (z. B. Terminator-Block) oder keine In- und Outports (z. B. DataStore-Memory-Block) besitzen. Auch gibt es Blöcke bei denen die Anzahl der In- und Outports variabel ist (z. B. Subsystem-Block, MultiportSwitch-Block).

Mit diesen In- und Outports kann ein Block mit anderen Blöcken *kommunizieren* bzw. Daten austauschen. Ein Block empfängt Daten durch einen Inport und sendet Daten durch einen Outport. Dabei hat jeder In- und Outport seine eigene Bedeutung, die vom jeweiligen Block abhängig ist. Jeder Outport kann über Kanten mit beliebig vielen Inports verbunden werden. Jedoch kann jeder Inport nur mit genau einem Outport verbunden sein.

Sowohl die Inports als auch die Outports eines Blockes werden von oben nach unten durchnummeriert. Dieser Port-Index wird in dieser Arbeit auch als *Portnummer* bezeichnet.

Subsysteme

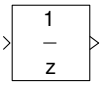
Ein Subsystem ist ein System das in einem Block gekapselt wird und damit wieder in ein anderes System eingebaut werden kann. Ein Subsystem hat dabei normale In- und Outports. Diese werden im Subsystem als Inport- und Outport-Blöcke dargestellt.

Blocktypen

In den folgenden Tabellen wird die Funktionsweise einzelner Blocktypen erläutert, die für das Verständnis dieser Arbeit benötigt werden.

In Tabelle 4.1 werden Blöcke vorgestellt, die Werte speichern können und damit einen diskreten Zustand haben. Tabelle 4.2 zeigt Blöcke für logische und

Tabelle 4.1: Diskrete-Blöcke

Blockname	Abbildung	Beschreibung
Memory		Speichert einen Wert und leitet ihn mit einem Zeitschritt Verzögerung weiter.
UnitDelay		Speichert einen Wert und leitet ihn nach einer frei wählbaren Verzögerung weiter.

relationale Operatoren. Tabelle 4.3 enthält Blöcke, die eine *Lookup-Tabelle* als

Tabelle 4.2: Operator-Blöcke

Blockname	Abbildung	Beschreibung
LogicalOperator		Errechnet mit den Eingangswerten eine wählbare logische Funktion (und, oder, nicht, ...).
RelationalOperator		Vergleicht die Eingangswerte auf eine wählbare Relation (größer, kleiner, gleich, ...) und liefert einen Wahrheitswert zurück.

Datenstruktur verwenden. Tabelle 4.4 erklärt die wichtigsten Blöcke für mathematische Berechnungen. In Tabelle 4.5 werden die wichtigsten Blöcke für Subsysteme beschrieben. Die Tabellen 4.6 und 4.7 beschreiben Blöcke die Signale (siehe Kapitel 4.2.3) weiterleiten. In Tabelle 4.8 werden Senken in Simulink-Modellen vorgestellt. Tabelle 4.9 stellt Quellen in Simulink-Modellen vor. Tabelle 4.10 beschreibt sonstige Blöcke.

Tabelle 4.3: Lookup-Tabellen

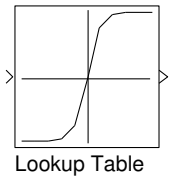
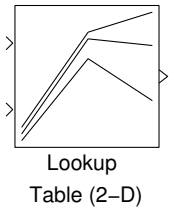
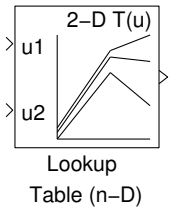
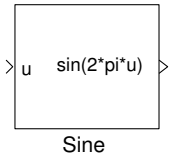
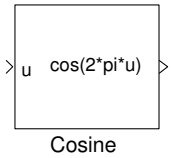
Blockname	Abbildung	Beschreibung
LookupTable		Eine eindimensionale Lookup-Tabelle.
LookupTable (2-D)		Eine zweidimensionale Lookup-Tabelle.
LookupTable (n-D)		Eine n-dimensionale Lookup-Tabelle.
Sine		Eine vorberechnete Sinus Funktion in einer Lookup-Tabelle.
Cosine		Eine vorberechnete Kosinus Funktion in einer Lookup-Tabelle.

Tabelle 4.4: Mathematische Operationen

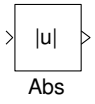
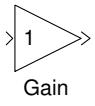
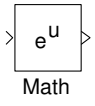
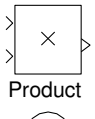
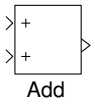
Blockname	Abbildung	Beschreibung
Abs		Berechnet den Betrag für den eingehenden Wert.
Gain		Multipliziert den eingehenden Wert mit einer Konstanten.
MathFunction		Berechnet für einen eingehenden Wert x eine wählbare mathematische Funktion (e^x, $ln(x)$, $log(x)$, x^2, ...).
Product		Multipliziert zwei eingehende Werte miteinander.
Sum, Add		Addiert zwei eingehende Werte zusammen. Dieser Block hat zwei verschiedene Darstellungen, die Blocklogik ist jedoch äquivalent.

Tabelle 4.5: Port- und Subsystem-Blöcke

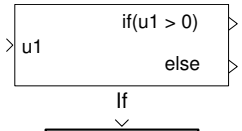
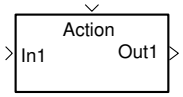
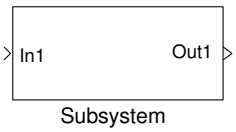
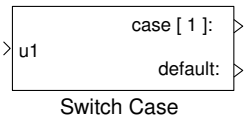
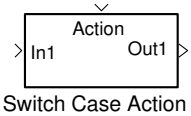
Blockname	Abbildung	Beschreibung
If		Implementierung eines *If-Else* Kontrollflusses.
IfActionSubsystem		Vorgefertigtes Subsystem für einen *If-Else* Kontrollfluss.
Inport		Eingabekanal eines Subsystems.
Outport		Ausgabekanal eines Subsystems.
Subsystem		Gekapseltes System in einem anderen System.
SwitchCase		Implementierung eines *Switch-Case* Kontrollflusses.
SwitchCaseAction-Subsystem		Vorgefertigtes Subsystem für einen *Switch-Case* Kontrollfluss.

Tabelle 4.6: Routing-Blöcke

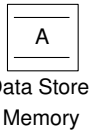
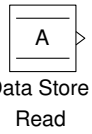
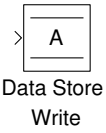
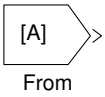
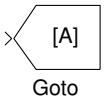
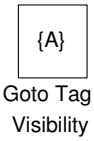
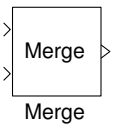
Blockname	Abbildung	Beschreibung
BusCreator		Erzeugt einen Bus aus beliebig vielen Signalen.
BusSelector		Extrahiert beliebig viele Signale aus einem Bus.
DataStoreMemory		Speichert Daten.
DataStoreRead		Liest Daten von einem DataStoreMemory-Block.
DataStoreWrite		Schreibt Daten in einen DataStoreMemory-Block.
Mux		Erzeugt einen Vektor aus allen eingehenden Werten.
Demux		Extrahiert Elemente aus einem Vektor.
From		Empfängt ein Signal von einem Goto-Block.
Goto		Sendet ein Signal an beliebig viele From-Blöcke.
GotoTagVisibility		Legt die Sichtbarkeit von Goto-Blöcken fest.
Merge		Leitet den zuletzt berechneten Wert weiter.

Tabelle 4.7: Routing-Blöcke Fortsetzung

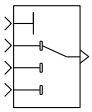
Blockname	Abbildung	Beschreibung
MultiportSwitch		Durch den ersten Inport kann ausgewählt werden, welcher andere Inport weitergeleitet wird.
Switch		Der zweite Inport entscheidet, ob der erste oder der dritte Inport weitergeleitet wird.

Tabelle 4.8: Senken

Blockname	Abbildung	Beschreibung
Outport		Ein Outport auf höchster Subsystem-Hierarchieebene ist eine Ausgabeschnittstelle. Solche Outports leiten alle eingehenden Werte an das aufrufende System zurück.
Terminator		Alle eingehenden Werte werden verworfen.

Tabelle 4.9: Quellen

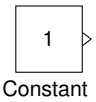
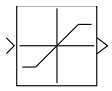
Blockname	Abbildung	Beschreibung
Constant		Generiert einen konstanten Wert.
Ground		Generiert einen passenden Wert, abhängig davon mit welchem Block er verbunden ist.
Inport		Ein Inport auf höchster Subsystem-Hierarchieebene ist eine Eingabeschnittstelle. Solche Inports erhalten alle eingehenden Werte von dem aufrufenden System.

Tabelle 4.10: Sonstige-Blöcke

Blockname	Abbildung	Beschreibung
Saturation	 Saturation	Bestimmt eine obere und untere Schranke für den eingehenden Wert.

4.2.2 Kanten

Blöcke können in Simulink über ihre In- und Outports miteinander verbunden sein. Dies geschieht über Kanten. Eine Kante ist hierbei immer gerichtet und kann sich aufteilen (Mehrfachkante).

Eine Kante verbindet immer zwei verschieden Blöcke miteinander. Es ist nicht möglich einen Block mit sich selbst zu verbinden. In Abbildung 4.3 wird dies ver-

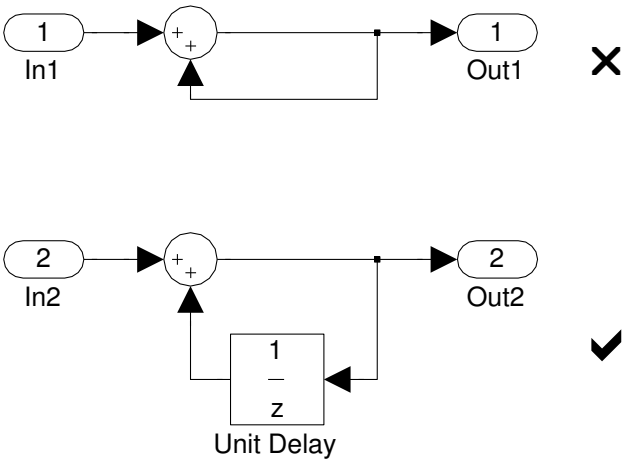


Abbildung 4.3: Beispiel für eine nicht erlaubte Kantenverbindung.

deutlicht. Im oberen Teil des Modells ist der Sum-Block mit sich selbst verbunden. Dies ist nicht erlaubt und führt zu einem Fehler. Im unteren Teil ist eine korrekte Variante dargestellt.

4.2.3 Signale

In Simulink gibt es Blöcke, deren Zweck nur die Strukturierung von Simulink-Modellen ist (z. B. Busse, Mux/Demux, Goto/From, ...). Diese Blöcke enthalten

keine Semantik und sind deshalb für viele potentielle Simulink-Metriken uninteressant. Es besteht somit ein Interesse einen Mechanismus zu entwickeln, der diese strukturellen Blöcke *ausblendet*.

Für diesen Zweck werden in dieser Arbeit *Signale* definiert. Anschaulich beschrieben ist ein Signal ein Weg in einem Simulink-Modell, auf dem ein errechneter Wert weitergeleitet wird. Dieser Weg hat per Definition einen Startblock, kann dann durch mehrere Blöcke führen und endet in einem Block. Ein Verzweigen ist möglich, dadurch wird ein neues Signal kreiert, dass den bisher zurückgelegten Weg des *Vatersignals* erbt.

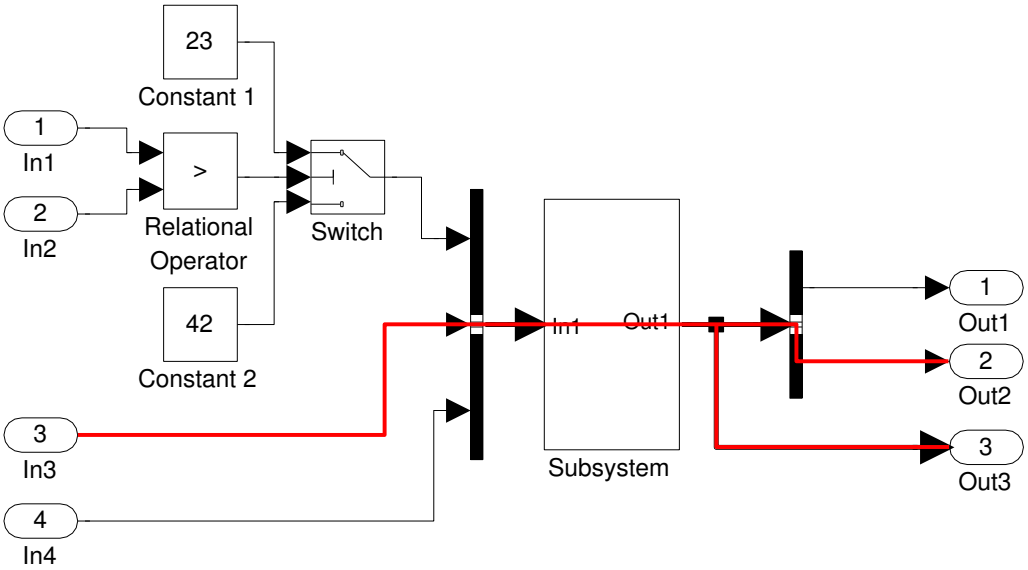


Abbildung 4.4: Beispiel für ein Signal in einem Simulink-Modell.

Ein Beispiel für einen Signalweg, der sich verzweigt, ist in Abbildung 4.4 dargestellt (rote Linie). Hierbei kommt das Signal durch den dritten Inport in das Subsystem und wird mit anderen Signalen in einem Bus gekapselt. Dieser Bus durchläuft danach ein Subsystem. Nach dem Subsystem verzweigt sich der Bus. Damit wird ein neues Signal kreiert, dass den bisher zurückgelegten Weg erbt. Eines der Signale wird wieder aus dem Bus geholt und in den zweiten Outport geleitet. Das andere Signal, dass sich immer noch im Bus befindet, wird in den dritten Outport geleitet.

Die hier vorgestellten *Signale* sind eine Definition des Autors.

Definition: Signal, Signalquelle, Signalträger, Signalsenke

Ein *Signal* ist ein Wert, der in einer Signalquelle beginnt und über mehrere Signalträger weitergeleitet wird, bis er schließlich in einer Signalsenke endet. Dabei kann sich der Weg eines Signals aufteilen.

Wenn sich der Weg eines Signals aufteilt, wird ein neues Signal erstellt. Dieses neue Signal erbt den bisher zurückgelegten Weg. Anschließend folgt jedes Signal einem der möglichen Wegverzweigungen.

Ein Signal wird von einer *Signalquelle* produziert. Eine *Signalquelle* ist ein Block, der aus keinem oder mehreren eingehenden Signalen ein neues Signal generiert. Die verschiedenen Signalquellen sind in Tabelle 4.11 aufgelistet. Alle in dieser

Tabelle 4.11: Signalquellen

Blocktyp	Beschreibung
Constant	Generiert ein Signal mit konstantem Wert
CustomCode	Bindet beliebigen Programmcode als Block ein
Ground	Liefert konstant Null als Signal
Inport	Die Inport-Blöcke der obersten Hierarchiestufe
Logic	Logische Operation (und, oder, ...)
Lookup	Bildet einen Eingabewert auf einen Ausgabewert ab
Lookup2D	Bildet zwei Eingabewerte auf einen Ausgabewert ab
LookupNDDirect	Bildet *n* Eingabewerte auf einen Ausgabewert ab
LookupNDInterp	Bildet *n* Eingabewerte auf einen interpolierten Ausgabewert ab
Math	Berechnet eine mathematische Funktion
Mux	Vereinigt mehrere Signale zu einem Vektorsignal
Demux	Spaltet ein Vektorsignal in verschiedene Signale auf
Product	Bildet das Produkt aller Eingänge
RelationalOperator	Vergleicht die Eingaben miteinander (gleich, größer, ...)
Stateflow	Enthält einen Zustandsautomaten
Sum	Summiert alle Eingänge
DataStoreRead	Liest Werte aus einem Speicher

Tabelle genannten Blocktypen generieren ein Signal an jedem ihrer Ausgänge und jedes eingehende Signal endet in ihnen. Mit den hier afugeführten Inport-Blöcken sind nur die Inport-Blöcke aus der obersten Ebene des Simulink-Modells gemeint. Alle anderen Inport-Blöcke sind Signalträger.

Einzige Ausnahme in Tabelle 4.11 ist die *nicht*-Operation des Logic-Block. Diese Operation modifiziert zwar ein Signal, wird aber nicht als Signalquelle behandelt, sondern als Signalträger. Dies liegt daran, dass dieser Block das Signal unabhängig von einem anderen Signal minimal verändert.

Neben der *nicht*-Operation des Logic-Block gibt es noch weitere *Signalträger*, die ein Signal weiterleiten. Diese Blöcke zählt Tabelle 4.12 auf. Alle Signalträger leiten ein Signal weiter. Eine genaue Erläuterung wie jeder einzelne Signalträger ein Signal weiterleitet, wird in Kapitel 4.2.3 gegeben.

Tabelle 4.12: Signalträger

Blocktyp	Beschreibung
Abs	Berechnet den Betrag eines numerischen Signals
BusCreator	Bündelt mehrere Signale zu einem Signalbus
BusSelector	Selektiert ausgewählte Signale aus einem Signalbus
BusInport	Enthält Informationen zur Codegenerierung für Busse
BusOutport	Enthält Informationen zur Codegenerierung für Busse
From	Empfängt ein Signal von einem Goto-Block
Goto	Sendet ein Signal an ein oder mehrere From-Blöcke
Gain	Multipliziert ein Signal mit einem festen Wert
Subsystem	Kapselt beliebige Blöcke/Systeme als Block
Inport	Repräsentiert einen Inport in einem Subsystem
Outport	Repräsentiert einen Outport in einem Subsystem
TLInport	Enthält Zusatzinformationen für die Codegenerierung
TLOutport	Enthält Zusatzinformationen für die Codegenerierung
Logic	Nur die logische *nicht*-Operation
Merge	Kombiniert mehrere Signale zu einem Signal
Saturate	Setzt eine Ober- und Untergrenze für ein Signal
Switch	Ein Steuersignal entscheidet welches der zwei weiteren Eingangssignale weitergeleitet wird
MultiPortSwitch	Ein Steuersignal entscheidet welches der *n* weiteren Eingangssignale weitergeleitet wird
UnitDelay	Verzögert ein Signal um eine konstante Zeitspanne

Eine *Signalsenke* ist ein Block in dem einige oder alle hineinkommenden Signale enden. Alle Signalquellen sind auch Signalsenken, ebenso wie alle hier nicht aufgeführten Blöcke. Aber auch einige Signalträger könne Signalsenken sein, z. B. endet das zweite hineinkommende Signal eines Switch-Blocks in diesem. Eine genaue Auflistung wie jeder Block eingehende Signale behandelt, wird in Kapitel 4.2.3 aufgeführt.

Umsetzung

Dieser Abschnitt erläutert, wie alle Signale in einem Simulink-Modell errechnet werden.

Der Weg eines Signals wird als Liste von Outports gespeichert, damit ist der Verlauf eines Signals eindeutig gespeichert. Ein Signal ist somit eine Liste von Outports. Wenn ein Signal zyklisch verläuft, endet es in dem zuletzt durchlaufenen Signalträger, der nicht Teil des Signalwegs ist.

Aus einer Liste von Outports lässt sich einfach die Liste der Blöcke berechnen, die der Signalweg durchläuft. Diese Liste von Blöcken wird in dieser Arbeit als *Blockweg* bezeichnet. Ein Blockweg fängt bei der Signalquelle an und endet in der Signalsenke.

In Algorithmus 1 wird dargestellt, wie der grundlegende Algorithmus zur Berechnung von Signalen funktioniert. Der Algorithmus beginnt in Zeile 1 damit, alle

Algorithmus 1 : Der Grundalgorithmus zur Berechnung der Signale

input : Die Menge aller Blöcke B
output : Alle berechneten Signale

```
// Speichere alle Blöcke die Signalquellen sind in Q
1  Q ← ∅;
2  foreach b ∈ B do
3      if IstSignalquelle(b) then
4          Q ← Q ∪ {b};
5      end
6  end
   // Berechne für jeden Ausgang eines Blocks die kreierten
      Signale
7  Signale ← ∅;
8  foreach b ∈ Q do
9      foreach outport ∈ Outports(b) do
10         Signale ← Signale ∪ BerechneSignale(outport, ∅, ∅) ;
11     end
12 end
   // Gib alle gefundenen Signale zurück
13 return Signale;
```

Signalquellen zu finden, die in Tabelle 4.11 aufgelistet sind. Anschließend wird für

Tabelle 4.13: Hilfsfunktionen des Grundalgorithmus zur Berechnung der Signale

Funktion	Beschreibung
IstSignalquelle	Gibt *True* zurück, wenn der übergebene Block eine Signalquelle ist, sonst *False*.
BerechneSignale	Siehe Algorithmus 2.
Outports	Gibt die Menge aller Outports für einen übergebenen Block zurück.

jeden Outport einer Signalquelle, die Signale berechnet, die von diesem Outport kreiert werden (ab Zeile 7).

Der Algorithmus 2 beschreibt die Berechnung der Signale, ausgehend von einem Outport. Hierbei durchläuft der Algorithmus die Wege, die von dem übergebenen Outport aus erreichbar sind, bis er auf eine Signalsenke trifft.

Hierfür ermittelt der Algorithmus zuerst den zugehörigen Block, für den aktuellen Outport. In Zeile 3 wird überprüft, ob der Block eine Signalsenke ist und dadurch das Signal beendet werden muss.

Danach wird kontrolliert, ob der aktuelle Block schon einmal besucht wurde (ab Zeile 4). Wenn der Block schon besucht wurde, wird in der Variable *signalEnde* gespeichert, dass das Signal in diesem Block endet. Dies ist nötig um zyklische Signale zu verhindern. Wenn der Block noch nicht besucht wurde, wird er zu der Liste aller besuchten Blöcke hinzugefügt.

In Zeile 9 wird, wenn das Signal zu Ende ist, das aktuelle Signal zur Menge aller Signale hinzugefügt und zurückgegeben. Wenn das Signal nicht endet, wird der aktuelle Outport dem Signal hinzugefügt (Zeile 12). Anschließend werden die Inports herausgesucht, die mit dem aktuellen Outport verbunden sind. Danach wird versucht für jeden Inport die entsprechenden Outports herauszufinden, an denen das Signal fortgeführt wird. Welche Outports dies genau sind, hängt immer vom einzelnen Block ab. Die Funktion *LeiteSignalZuNächstemOutport* in Zeile 15 erledigt die Signalverfolgung in einem Block und behandelt die verschiedenen Blocktypen folgendermaßen:

Abs-, BusInport-, BusOutport-, From-, Gain-, Merge-Blöcke Jedes eingehende Signal wird dem einzigen Outport zugeordnet.

Algorithmus 2 : Funktion BerechneSignale

input : Ein Outport eines Blocks (outport)
input : Die Menge aller bisher besuchten Blöcke V
input : Das bisherige Signal s als Liste von Outports
output : Die Menge aller Signale, die von diesem Outport ausgehen

```
1  Signale ← ∅;
2  b ← HoleBlock(outport);
   // Diese Variable speichert, ob dies der letzte Block im
      Signal ist
3  signalEnde ← IstSignalsenke(b);
   // Überprüfe ob b schon besucht wurde. Dies ist nötig um
      Zyklen zu erkennen
4  if b ∈ V then
5      signalEnde ← true;
6  else
7      V ← V ∪ {b};
8  end
   // Abbruch der Rekursion
9  if signalEnde then
10     Signale ← {s};
11 else
       // Füge den aktuellen Outport dem Signal hinzu
12     HängeOutportAn(s, outport);
       // Speichere alle nächsten Outports in einer Menge
13     nächsteOutports ← ∅;
14     foreach inport ∈ VerbundeneInports(outport) do
15         nächsteOutports ← nächsteOutports ∪
           LeiteSignalZuNächstemOutport(inport);
16     end
       // Löse die restlichen Verbindungen rekursiv
17     foreach nächsterOutport ∈ nächsteOutports do
18         Signale ← Signale ∪ BerechneSignale(nächsterOutport, V, s);
19     end
20 end
21 return Signale;
```

Tabelle 4.14: Hilfsfunktionen der Funktion BerechneSignale

Funktion	Beschreibung
HoleBlock	Gibt für einen Outport den zugehörigen Block zurück.
IstSignalsenke	Gibt *True* zurück wenn der übergebene Block eine Signalsenke ist, sonst *False*.
HaengeOutportAn	Fügt ein Outport einem Signal hinzu. Hiermit werden alle von einem Signal besuchten Outports in einer Liste gespeichert.
VerbundeneInports	Gibt die Menge von Inports zurück mit denen ein Outport verbunden ist.
LeiteSignalZuNaechstemOutport	Berechnet die Menge von Outports, die von einem bestimmten Inport aus erreicht werden können.

Saturate-, TLInport-, TLOutport- und UnitDelay-Blöcke Auch bei diesen Blöcken wird jedes eingehende Signal dem einzigen Outport zugeordnet.

BusCreator-Blöcke Jedes eingehende Signal wird dem einzigen Outport zugeordnet. Zusätzlich wird gespeichert, durch welchen Inport (Index) das Signal in den Bus gelangt ist.

BusSelector-Blöcke Dem Signal wird der Outport zugewiesen, der sich durch den gespeicherten Signal-Index und den Einstellungen des BusSelector-Blocks ergibt. Wenn sich kein Outport eindeutig dem Signal zuordnen lässt, endet das Signal in diesem Block.

Goto-Blöcke Es werden die Outports der zugehörigen From-Blöcke zugeordnet.

Logic-Blöcke Wenn der Operator das logische *nicht* ist, dann wird der einzige Outport zurückgeliefert.

Switch-Blöcke Das Signal, das am zweiten Inport ankommt, endet in diesem Block. Jedes weitere Signal wird dem einzigen Outport zugeordnet.

MultiPortSwitch-Blöcke Das Signal das am ersten Inport ankommt, endet in diesem Block. Jedes weitere Signal wird dem einzigen Outport zugeordnet.

Outport-Blöcke Bei einem Outport-Block wird das Signal an den Outport des übergeordneten Subsystems weitergeleitet.

Subsystem-Blöcke Ein Eingangssignal wird an den Outport des entsprechenden Inport-Block weitergeleitet.

Sonstige Blöcke In allen weiteren Blöcken endet jedes Signal.

Nachdem der Algorithmus die Menge von neuen Outports bestimmt hat, geht er für jeden in die Rekursion (Zeile 18). Wenn die Signale für alle Outports rekursiv berechnet sind, werden diese in einer Menge zusammengefasst und zurückgegeben.

4.3 Simulink-Modelle als Graphen

Da ein Simulink-Modell auch als Graph betrachtet werden kann, werden in diesem Absatz Gemeinsamkeiten und Unterschieden von Simulink-Modellen und Graphen erklärt.

Ein Simulink-Modell ist im Prinzip ein gerichteter Graph mit Mehrfachkanten, mit einigen Ausnahmen.

Im Gegensatz zu einem Graphen sind bei Simulink-Blöcken die Anzahl der eingehenden Kanten durch die Anzahl der Inports beschränkt. Dies bedeutet, dass ein Simulink-Block maximal so viele eingehende Kanten haben kann wie er Inports besitzt. Ebenso haben die einzelnen In- und Outports unterschiedliche Semantiken, d. h. jeder In- oder Outport eines Blocks kann eine andere Bedeutung haben.

4.4 Modell-Exporter

Im Rahmen dieser Arbeit wurde ein Programm entwickelt, das Simulink-Modelle im XML-Format exportiert. Dabei werden die Eigenschaften des Modells gespeichert die im Zuge dieser Arbeit benötigt werden. Ebenso wird eine Vereinfachung durchgeführt, indem äquivalente TargetLink- und Simulink-Blöcke zusammengefasst werden. So wird z. B. ein TargetLink Constant-Block und ein Simulink Constant-Block als der selbe Blocktyp abgespeichert.

Der Modell-Exporter ist in der in MATLAB integrierten Programmiersprache entwickelt und hat einen Umfang von ca. 2000 Zeilen. Durch die Verwendung der

integrierten Programmiersprache ist ein einfacher Zugriff auf die Elemente eines Simulink-Modells möglich.

5 Metriken – Theorie und Umsetzung für Simulink-Modelle

Dieses Kapitel beschreibt die Entwicklung und Realisierung von Metriken für Simulink-Modelle und wie diese für die vorliegende Master-These umgesetzt wurden.

Im Unterkapitel 5.1 wird erläutert, wie der Autor mit dem Goal/Question/Metric-Ansatz einzelne Kriterien identifiziert, mit denen Metriken entwickelt werden können, um die Qualität von Simulink-Modellen zu beurteilen.

In Abschnitt 5.2 werden schließlich einzelne Metriken entwickelt, um die vorher identifizierten Kriterien messen und bewerten zu können.

Zuletzt werden in 5.3 einige Metriken vorgestellt, die der Autor während dieser Arbeit untersucht hat, die aber aus verschiedenen Gründen nicht realisiert wurden.

5.1 Identifikation von Metriken zur Qualitätsmessung

In diesem Abschnitt wird beschrieben, wie der Goal/Question/Metric-Ansatz verwendet wurde, um Kriterien für Metriken zu identifizieren. Hierfür wird ein messbares Ziel definiert, unter der Verwendung des Goal/Question/Metric-Ansatzes. Aufbauend auf diesem Ziel werden Fragen (auch Kriterien genannt) entwickelt, die dieses Ziel beschreiben. Verschiedene Metriken zur Bewertung der entwickelten Kriterien, werden anschließend in Abschnitt 5.2 vorgestellt.

Der Begriff *Kriterium* wird in diesem Kapitel für Merkmale verwendet, die zusammengenommen den abstrakten Begriff *Qualität* beschreiben.

Eine kurze Beschreibung über die Funktionsweise des Goal/Question/Metric-Ansatzes befindet sich in Kapitel 2.2. In diesem Kapitel werden nur die ersten drei Prozessschritte verwendet um Metriken zu identifizieren, die Schritte vier und fünf sind nicht Teil dieser Arbeit.

Ziel

Das Ziel ist das Messen der Modellqualität von einzelnen Simulink-Modellen. Ein großes Problem ist, dass Qualität immer ein subjektiver Begriff ist. Jede Person hat von Modell- oder Software-Qualität eine andere Auffassung. Hier ist es schwierig, alle möglichen Aspekte abzudecken.

Frage

Zuerst stellt sich die Frage, was Modellqualität ist? Hierfür wird in dieser Arbeit der Begriff „Modellqualität“ aufgebrochen in feinere Fragen.

Ein möglicher Ansatzpunkt ist das Konsultieren von Normen, in denen der Begriff Qualität definiert wird. In den nächsten zwei Absätzen werden kurz die Normen *DIN 66272* und *ISO/IEC 9126* vorgestellt mit ihren einzelnen Qualitätsmerkmalen.

DIN 66272 Die Norm *DIN 66272* ist eine ehemalige DIN-Norm [Wik10a]. In dieser Norm werden Qualitätsmerkmale für Software festgelegt. Die darin definierten Hauptqualitätsmerkmale sind:

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit

ISO/IEC 9126 Die Norm *ISO/IEC 9126* ist eine Norm für die Produktqualität von Software und ist in der Norm *ISO/IEC 25000* aufgegangen [Wik10b]. Die Hauptqualitätsmerkmale sind die selben wie in der DIN-Norm 66272, jedoch gibt es Vorschläge zur Verfeinerung der einzelnen Merkmale:

- *Funktionalität*: Angemessenheit, Richtigkeit, Interoperabilität, Sicherheit
- *Zuverlässigkeit*: Reife, Fehlertoleranz, Wiederherstellbarkeit
- *Benutzbarkeit*: Verständlichkeit, Erlernbarkeit, Attraktivität
- *Effizienz*: Zeitverhalten, Verbrauchsverhalten
- *Änderbarkeit*: Analysierbarkeit, Modifizierbarkeit, Stabilität, Testbarkeit
- *Übertragbarkeit*: Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit

Kriterien

In dieser Arbeit werden jedoch nicht alle Bereiche abgedeckt, die in den Normen genannt werden. Für die Bereiche Funktionalität und Zuverlässigkeit liegen keine messbaren Daten vor. Der Bereich Benutzbarkeit ist für die untersuchten Modelle ebenfalls nicht relevant, da die Modelle in den meisten Fällen interne Prozesse steuern oder regeln und nicht vom Benutzer direkt bedient werden. Übrig bleiben die Punkte Effizienz, Änderbarkeit und Übertragbarkeit.

Die Punkte Änderbarkeit und Übertragbarkeit sind jedoch noch zu unspezifisch. Aus diesem Grund wird der Punkt Änderbarkeit in die Unterkriterien Dokumentation und Komplexität zerlegt. Der Punkt Übertragbarkeit wird in die Unterkriterien Modularisierung und Wartbarkeit aufgeteilt.

Schließlich ergeben sich die folgenden fünf Kriterien, die einzelne Qualitätsaspekte von Simulink-Modellen möglichst gut fassen sollen:

- Wie gut ist die *Dokumentation*?
- Wie *komplex* ist das Modell?
- Wie gut ist das Modell *modularisiert*?
- Wie gut ist das Modell *wartbar*?

- Wie *effizient* ist das Modell in der Ausführung?

Auch andere Aufteilungen des Begriffs *Qualität* sind denkbar. Die hier vorgestellte Aufteilung basiert auf der Meinung des Autors und dessen Verständnis was Qualität bedeutet.

5.2 Metriken für Simulink-Modelle

Alle Metriken in diesem Kapitel sind in die in Abschnitt 5.1 identifizierten Kriterien eingeteilt. Am Anfang jedes Abschnitts werden mit Hilfe des Goal/Question/Metric-Ansatzes Metriken entwickelt, um die einzelnen Kriterien abzudecken.

In Tabelle 5.1 werden die bisher bekannten Metriken für Simulink-Modelle aufgelistet. Außerdem wird jede Metrik dem Kriterium zugeordnet, dass die Metrik misst. Da durch existierende Metriken nicht alle Kriterien abgedeckt sind, werden

Tabelle 5.1: Existierende Metriken für Simulink-Modelle

Metrik	Dokumentation	Komplexität	Modularisierung	Wartbarkeit	Effizienz
Blockinstabilität				○	
Halstead-Volumen		○			
Systeminstabilität				○	
Zyklomatische Komplexität		○			

in den folgenden Unterkapiteln Metriken vorgestellt, sodass eine Abdeckung aller Kriterien gegeben ist.

Jede der folgenden Metriken wird mit dem selben Schema beschrieben: Zu Beginn wird erläutert was die Metrik macht und was mit ihr bezweckt wird. Im Abschnitt *Definition* werden alle Begriffe definiert, welche die entsprechende Metrik benötigt. Im Absatz *Umsetzung* wird detailliert beschrieben, wie der Autor die Metrik realisiert hat.

Alle Metriken wurden in der Sprache Java implementiert und getestet. Alle in diesem Kapitel vorgestellten Algorithmen wurden vom Autor persönlich entwickelt und umgesetzt.

5.2.1 Messung der Dokumentation

In diesem Unterkapitel werden Metriken zur Bewertung der Dokumentation von Simulink-Modellen vorgestellt. Hierfür werden mit dem Goal/Question/Metric-Ansatz einzelne Metriken identifiziert, die im weiteren Verlauf erläutert werden.

Goal Eine gute Dokumentation.

Question Wie ausführlich ist die Dokumentation?

Metric Durchschnittliche Kommentarlänge pro Block.

Die Implementation dieser Metrik beschreibt das nächste Unterkapitel.

Durchschnittliche Kommentarlänge pro Block

Diese Metrik errechnet die durchschnittliche Länge eines Kommentars pro Block. Der Autor dieser Arbeit vermutet, dass die durchschnittliche Kommentarlänge pro Block einen guten Hinweis gibt, wie ausführlich die Dokumentation in einem Simulink-Modell ist.

Definition: Durchschnittliche Kommentarlänge pro Block Jeder Simulink-Kommentar ist eine Zeichenkette, die beliebig in einem Subsystem platziert werden kann. Für die Menge aller Kommentare K und die Menge aller Blöcke B berechnet sich die durchschnittliche Kommentarlänge pro Block folgendermaßen:

$$\frac{\sum_{k \in K} Länge(k)}{|B|} \tag{5.1}$$

Umsetzung Da die Umsetzung dieser Metrik trivial ist, wird auf die Angabe eines genauen Algorithmus zur Berechnung dieser Metrik verzichtet.

5.2.2 Messung der Modularisierung

In diesem Unterkapitel werden Metriken zur Bewertung der Modularisierung von Simulink-Modellen vorgestellt. Hierfür werden mit dem Goal/Question/Metric-Ansatz einzelne Metriken identifiziert, die im weiteren Verlauf erläutert werden.

Goal Eine gute Modularisierung von Simulink-Modellen.

Question Wie gut bildet ein Subsystem eine Funktionalität ab?

Metric Unabhängige Pfade.

Die Implementation dieser Metrik wird in dem nächsten Unterkapitel beschrieben.

Unabhängige Pfade

Diese Metrik versucht die *Kohäsion* eines Subsystems zu messen. Dafür wird die Anzahl von *unabhängigen Pfaden* pro Subsystem gemessen. Ein *unabhängiger Pfad* stellt eine eigenständige logische Einheit dar, die in keiner Verbindung mit einer anderen logischen Einheit steht.

In Abbildung 5.1 ist ein Subsystem mit zwei unabhängigen Pfaden dargestellt. Der erste unabhängige Pfad beginnt bei den ersten drei Inports und endet im ersten und zweiten Outport. Der zweite unabhängige Pfad beginnt beim vierten Inport und endet im dritten Outport.

Bei einer guten Modularisierung soll jedes Subsystem eine Aufgabe oder logische Einheit abbilden. Wenn die Anzahl der unabhängigen Pfade größer als eins ist, werden mehrere logische Einheiten in einem Subsystem abgebildet. Nach der Meinung des Autors, sollte damit ein durchschnittlich hoher Wert von *unabhängigen Pfaden* pro Subsystem auf eine schlechte Modularisierung hindeuten.

Die Inspiration für diese Metrik ist dem Artikel [KKT10, S. 8] entnommen. In dieser Arbeit wird die Metrik nur oberflächlich erläutert. Die genaue Definition und Implementation dieser Metrik ist somit eine Idee des Autors der vorliegenden Arbeit.

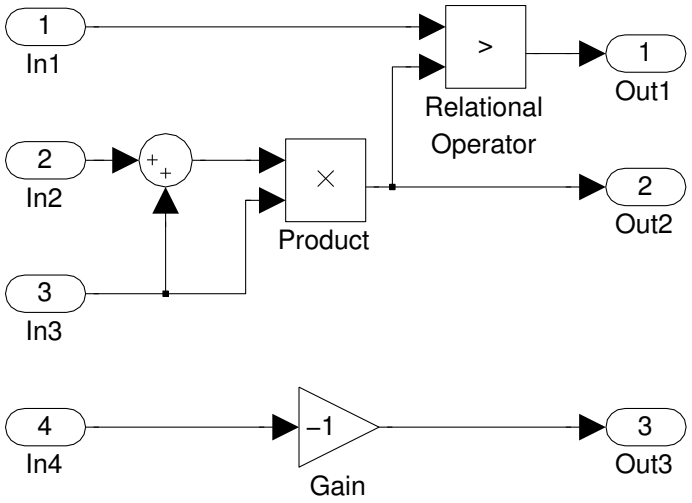


Abbildung 5.1: Beispiel für ein Subsystem mit zwei unabhängigen Pfaden.

Definition: Unabhängige Pfade, Startblöcke, Endblöcke Ein *unabhängiger Pfad* in einem Subsystem ist ein zusammenhängender, maximal großer Teilgraph. Dies bedeutet, dass zwischen zwei Blöcken aus verschiedenen unabhängigen Pfaden in einem Subsystem keine Verbindung existiert. In Abbildung 5.1 sind damit alle Blöcke zusammengenommen, die von den Inports eins, zwei und drei erreichbar sind, ein unabhängiger Pfad. Ebenso sind alle Blöcke zusammengenommen, die von Inport vier erreicht werden können, ein unabhängiger Pfad. Zwischen diesen beiden unabhängigen Pfaden existiert keine Verbindung.

Alle Eintrittspunkte in einem Subsystem werden in dieser Metrik als *Startblöcke* bezeichnet. Analog dazu werden alle Austrittspunkte aus einem Subsystem als *Endblöcke* bezeichnet.

Eine genaue Definition von Start- und Endblöcken, ist in Tabelle 5.2 gegeben. Jedoch gibt es auch zwei Blocktypen die nur unter bestimmten Bedingungen

Tabelle 5.2: Definition von Start- und Endblöcken

Blocktyp	Beschreibung
Startblock	Ein Block, der keinen Inport besitzt, aber mindestens einen Outport (Quelle in einem Graphen)
Endblock	Ein Block, der mindestens einen Inport besitzt, aber keinen Outport (Senke in einem Graphen)

Start- oder Endblöcke sind. Diese werden in Tabelle 5.3 erläutert.

Tabelle 5.3: Ausnahmen von Start- und Endblöcken

Blocktyp	Beschreibung
From-Block	Ein From-Block ist nur dann ein Startblock, wenn der zugehörige Goto-Block in einem anderen Subsystem liegt
Goto-Block	Ein Goto-Block ist nur dann ein Endblock, wenn alle zugehörigen From-Blöcke in anderen Subsystemen liegen

Beispiel In diesem Abschnitt wird das Vorgehen des Algorithmus, der in Abschnitt 5.2.2 genau erläutert wird, kurz an dem Beispiel aus Abbildung 5.1 erklärt.

In diesem Beispiel sind alle Inport-Blöcke Startblöcke. Dies sind die Blöcke mit den Namen *In1*, *In2*, *In3* und *In4*. Außerdem sind alle Outport-Blöcke Endblöcke. Dies sind die Blöcke mit den Namen *Out1*, *Out2* und *Out3*.

Der Algorithmus iteriert über alle Blöcke. Wenn er auf einen Startblock stößt, errechnet er alle Endblöcke, die von diesem Block aus erreichbar sind. Dies ergibt für die Startblöcke folgende Endblockmengen:

- *In1*: {*Out1*}
- *In2*: {*Out1*, *Out2*}
- *In3*: {*Out1*, *Out2*}
- *In4*: {*Out3*}

Anschließend werden alle überlappende Endblockmengen solange zusammengefügt, bis alle paarweise disjunkt sind. In Tabelle 5.4 werden die einzelnen Vereinigungen aufgezeigt.

Zum Schluss bleiben noch zwei disjunkte Endblockmengen übrig. Jede dieser Mengen resultiert aus einem unabhängigen Pfad, womit der Algorithmus als Ergebnis den Wert „zwei“ zurück liefert.

Tabelle 5.4: Ablauf der Vereinigungen

1. Vereinigung	2. Vereinigung
{*In1*, *In2*}: {*Out1*, *Out2*} *In3*: {*Out1*, *Out2*} *In4*: {*Out3*}	{*In1*, *In2*, *In3*}: {*Out1*, *Out2*} *In4*: {*Out3*}

Umsetzung In Algorithmus 3 ist der Grundalgorithmus dargestellt zum Berechnen von unabhängigen Pfaden in einem Simulink-Subsystem.

Er geht alle Blöcke in einem Subsystem durch und überprüft jeden Block. Wenn der Block ein Startblock ist, wird die Menge aller Endblöcke errechnet, die von diesem Block aus erreichbar sind (Zeile 11). Diese Menge wird der Menge von Mengen von Endblöcken (*Endblockmengen*) hinzugefügt.

Ab Zeile 5 wird überprüft ob der Startblock ein From-Block ist, der noch zusätzlich überprüft werden muss ob er ein Startblock ist.

Anschließend werden in Zeile 15 alle Mengen von Endblöcken aus *Endblockmengen* die sich überschneiden vereint. Damit sind alle Mengen von Endblöcken aus *Endblockmengen* zueinander disjunkt.

Jede einzelne disjunkte Menge von Endblöcken ist das Ergebnis eines unabhängigen Pfads. Somit gibt der Algorithmus zum Schluss die Anzahl der paarweise disjunkten Mengen von Endblöcken zurück, also die Anzahl von unabhängigen Pfaden. Dies funktioniert aus folgendem Grund:

Wenn sich zwei Pfade überschneiden, befindet sich mindestens ein Block in beiden Pfaden. Von diesem Block ist immer die selbe Menge von Endblöcken erreichbar. Dadurch müssen diese Endblöcke in den beiden Mengen von Pfadenden, der beiden Pfade, enthalten sein. Die beiden Pfade bilden also gemeinsam einen unabhängigen Pfad und müssen zusammengefasst werden.

In Algorithmus 4 wird gezeigt, wie die Mengen von Endblöcken berechnet werden. Dafür überprüft der Algorithmus zuerst, ob er den zu untersuchenden Block schon besucht hat (ab Zeile 2). Wenn er ihn schon besucht hat, bricht der Algorithmus ab und liefert die leere Menge zurück. Wenn der Block noch nicht besucht wurde, wird er zur Menge aller besuchten Blöcke hinzugefügt. Anschließend wird der Blocktyp verglichen.

Algorithmus 3 : Der Grundalgorithmus zur Berechnung von unabhängigen Pfaden

input : Die Menge aller Blöcke aus einem Subsystem B_s
output : Die Anzahl der unabhängigen Pfade in einem Subsystem

```
1  Endblockmengen ← ∅;
2  foreach b ∈ B_s do
       // Überprüfe ob b ein Startblock ist
3      if AnzahlInports(b) = 0 ∧ AnzahlOutports(b) > 0 then
4          istStartblock ← true;
           // Spezialbehandlung für From-Blöcke
5          if Blocktyp(b) = From then
6              istStartblock ← false;
7              goto ← VerbundenerGotoBlock(b);
8              if Subsystem(b) ≠ Subsystem(goto) then istStartblock ← true;
9          end
           // Berechne die Menge der Endblöcke, die von b aus
              erreichbar sind
10         if istStartblock then
11             Endblockmengen ← Endblockmengen ∪
               ErreichbareEndblöcke(b, ∅);
12         end
13     end
14 end
15 Endblockmengen ← Vereinige(Endblockmengen);
16 return |Endblockmengen|;
```

Tabelle 5.5: Hilfsfunktionen des Grundalgorithmus zur Berechnung von unabhängigen Pfaden

Funktion	Beschreibung
AnzahlInports	Gibt die Anzahl der Inports für einen übergebenen Block zurück.
AnzahlOutports	Gibt die Anzahl der Outports für einen übergebenen Block zurück.
Blocktyp	Ermittelt für einen Block den Blocktyp und gibt diesen als Zeichenkette zurück.
VerbundenerGotoBlock	Gibt den Goto-Block zurück, der mit dem übergebenen From-Block verbunden ist.
Subsystem	Gibt das Subsystem zurück in dem sich der übergebene Block befindet.
ErreichbareEndblöcke	Siehe Algorithmus 4.
Vereinige	Siehe Algorithmus 5.

Algorithmus 4 : Funktion ErreichbareEndblöcke

input : Ein Block b
input : Die Menge aller bisher besuchten Blöcke V
output : Die Menge aller Endblöcke für b

```
   // Endblöcke speichert alle Endblöcke die von b aus
      erreicht werden können
1  Endblöcke ← ∅;
2  if b ∉ V then
3     V ← V ∪ b;
      // Überprüfe ob ein Goto-Block ein Endblock ist
4     if Blocktyp(b) = Goto then
         // Gehe in die Rekursion für alle From-Blöcke die im
            selben Subsystem sind
5        foreach f ∈ VerbundeneFromBlöcke(b) do
6           if Subsystem(b) = Subsystem(f) then
7              Endblöcke ← Endblöcke ∪ ErreichbareEndblöcke(f, V);
8           end
9        end
         // Wenn der Goto-Block nur Verbindungen zu externen
            From-Blöcken hat, ist er auch ein Endblock
10       if VerbundeneFromBlöcke(b) = ∅ then
11          Endblöcke ← Endblöcke ∪ b;
12       end
      // Alle Blöcke ohne Outports sind Endblöcke
13    else if AnzahlOutports(b) = 0 then Endblöcke ← Endblöcke ∪ b;
14    else
         // Gehen in die Rekursion für alle nachfolgenden
            Blöcke
15       foreach b' ∈ NachfolgendeBlöcke(b) do
16          Endblöcke ← Endblöcke ∪ ErreichbareEndblöcke(b', V);
17       end
18    end
19 end
20 return Endblöcke;
```

Tabelle 5.6: Hilfsfunktionen der Funktion ErreichbareEndblöcke

Funktion	Beschreibung
Blocktyp	Ermittelt für einen Block den Blocktyp und gibt diesen als Zeichenkette zurück.
VerbundeneFromBlöcke	Gibt die Menge von From-Blöcke zurück, die mit einem Goto-Block verbunden sind.
Subsystem	Gibt das Subsystem zurück in dem sich der übergebene Block befindet.
AnzahlOutports	Gibt die Anzahl der Outports für einen übergebenen Block zurück.
NachfolgendeBlöcke	Gibt die Menge aller Blöcke zurück, die direkt mit einem Outport des übergebenen Blocks verbunden sind.

Wenn der Block vom Typ Goto ist, wird überprüft, ob die angesprungenen From-Blöcke im selben Subsystem liegen (ab Zeile 4). Wenn alle From-Blöcke in anderen Subsystemen sind, ist der Goto-Block ein Endblock und wird zur Liste aller Endblöcke hinzugefügt. Wenn jedoch mindestens ein assoziierter From-Block im selben Subsystem existiert, fährt der Algorithmus rekursiv mit ihnen fort.

Wenn der Block kein Goto-Block ist und keine Outports besitzt, ist er ein Endblock und wird der Menge von Endblöcken hinzugefügt.

In Zeile 13 wird überprüft ob der untersuchte Block keine Outports besitzt. Wenn der Block keine Outports besitzt wird er der Menge von Endblöcken hinzugefügt.

Wenn der aktuelle Block kein Endblock ist, fährt der Algorithmus ab Zeile 14 mit allen nachfolgenden Blöcken rekursiv fort.

Schließlich wird in Zeile 20 die Menge von Endblöcken zurückgegeben, die vom untersuchten Block aus erreicht werden können.

In Algorithmus 5 ist beschrieben, wie die Menge der Mengen aller Endblöcke vereinigt wird, so dass alle enthaltenen Mengen zueinander disjunkt sind. Die Grundidee des Algorithmus ist es, so lange zwei überlappende Submengen zusammenzufügen, bis es keine sich überschneidenden Mengen mehr gibt. Dafür geht der Algorithmus zuerst in eine Schleife, die erst dann verlassen wird, wenn sich nichts mehr an der Menge geändert hat.

Algorithmus 5 : Funktion Vereinige

input : Die Menge der Mengen aller Endblöcke (Endblockmengen) für jeden Startblock

output : Die paarweise disjunkte Menge der Mengen aller Endblöcke

```
1  repeat
2      mengeGeändert ← false;
3      foreach E ∈ Endblockmengen do
4          foreach E' ∈ Endblockmengen \ {E} do
               // Vereinige E und E' wenn sie sich überschneiden
5              if E ∩ E' ≠ ∅ then
                   // Entferne E, E' und füge E ∪ E' hinzu
6                  Endblockmengen ← (Endblockmengen \ {E, E'}) ∪ {E ∪ E'};
7                  mengeGeändert ← true;
                   // Verlasse die innere Schleife
8                  break;
9              end
10         end
           // Verlasse die äußere foreach Schleife
11         if mengeGeändert then break;
12     end
13 until not mengeGeändert ;
14 return Endblockmengen;
```

In der Schleife wird jede Menge mit jeder anderen Menge auf eine Überschneidung geprüft. Wenn sie sich überschneiden, werden die alten Mengen entfernt und die vereinigte neue Menge der Menge von Pfadenden hinzugefügt. Die *break*-Anweisungen in Zeile 8 und 11 veranlassen einen erneuten Durchlauf der äußersten Schleife, wenn sich etwas an der Menge geändert hat. Zum Schluss wird die paarweise disjunkte Menge von Mengen von Endblöcken zurückgegeben.

5.2.3 Messung der Wartbarkeit

In diesem Unterkapitel werden Metriken zur Bewertung der Wartbarkeit von Simulink-Modellen vorgestellt. Hierfür werden mit dem Goal/Question/Metric-Ansatz einzelne Metriken identifiziert, die im weiteren Verlauf erläutert werden.

Goal Eine gute Wartbarkeit von Simulink-Modellen.

Question Werden Modellteile wiederverwendet?

Metric Klonerkennung.

Question Wie oft wird ein Modell umgebaut?

Metric Blockinstabilität.

Die Implementation dieser Metriken wird in den nächsten Unterkapiteln beschrieben.

Klonerkennung

Das Ziel der hier vorgestellten Klonerkennung ist das Finden von kopierten Teilmodellen (Klone) in Simulink-Modellen. Diese Aufgabe ist in der Graphentheorie bekannt als das Subgraph-Isomorphie-Problem. Da dieses Problem NP-Vollständig ist [Coo71, S. 152–154], existiert kein Algorithmus der in polynomieller Zeit eine korrekte Antwort liefert.

Dennoch ist es im Fall von Simulink-Modellen möglich einen effizienten Algorithmus zu entwickeln. Um dies zu erreichen, müssen spezielle Eigenschaften von Simulink-Modellen ausgenützt werden.

Die Vorteile einer Klonerkennung werden in dem Bericht [RC07, S. 8–9] ausführlich erläutert.

In den nächsten Unterabschnitten wird der Begriff eines Klons definiert, wie der Algorithmus zur Klonerkennung funktioniert und das Verfahren an einem kurzen Beispiel erläutert.

Definition: Klone In dieser Arbeit werden zwei Teilmodelle $K = (V_K, E_K)$ und $K' = (V_{K'}, E_{K'})$ eines Simulink-Modells $G = (V, E)$ als ein Klon bezeichnet, wenn die Eigenschaften *Zusammenhängend*, *keine Überschneidung*, *äquivalente Knotenmenge*, *äquivalente Kantenmenge*, *äquivalente Ausdehnung* und *Mindestgröße* gelten. Dabei sei V die Menge aller Blöcke und E die Menge aller Kanten eines Simulink-Modells.

Da die Klone Teilmodelle des Simulink-Modells sind, gilt:

- $V_K \subset V$ und $V_{K'} \subset V$
- $E_K \subset E$ und $E_{K'} \subset E$

Zusammenhängend Sowohl K als auch K' sind zusammenhängende Teilmodelle des Models G.

Keine Überschneidungen Zwischen den Teilmodellen K und K' gibt es keine Überschneidungen, d. h. es gilt:

$$E_K \cap E_{K'} = \emptyset \quad \wedge \quad V_K \cap V_{K'} = \emptyset \tag{5.2}$$

Äquivalente Knotenmenge Für jeden Knoten aus V_K existiert genau ein zugeordneter *äquivalenter* Knoten in $V_{K'}$ und umgekehrt, d. h. es existiert eine bijektive Abbildung zwischen V_K und $V_{K'}$.

Die Äquivalenz von Blöcken wird durch den Vergleich von ausgesuchten Blockeigenschaften festgestellt. Dabei wird ausgenutzt, dass Simulink-Blöcke Eigenschaften besitzen, mit denen die Ähnlichkeit zweier Blöcke zueinander gut bestimmbar ist. Diese Eigenschaften werden in Tabelle 5.7 aufgelistet. Es werden

Tabelle 5.7: Äquivalenzeigenschaften von geklonten Blöcken

Eigenschaft	Beschreibung
Blocktyp	Jeder Block besitzt einen eindeutigen Typ, der die Funktionalität des Blocks bestimmt
Anzahl Subblöcke	Die Anzahl der Kind-Blöcke. Diese Zahl ist nur bei Subsystemen größer als Null
Anzahl Inports	Die Anzahl der Eingänge eines Blocks
Anzahl Outports	Die Anzahl der Ausgänge eines Blocks
Operator	Falls der Block ein *Logical-* oder *RelationalOperator-Block* ist, wird untersucht, ob beide Blöcke den selben Operator verwenden

die Merkmale verwendet, die die Semantik des Blocks betreffen. Sonstige Eigenschaften, wie Name oder Position des Blocks, werden ignoriert. Wenn alle Eigenschaften aus Tabelle 5.7 zwischen zwei Blöcken identisch sind, werden diese als *äquivalent* bezeichnet.

Äquivalente Kantenmenge Für jede Kante aus E_K existiert genau eine zugeordnete *äquivalente* Kante in $E_{K'}$ und umgekehrt. Somit existiert eine bijektive Abbildung zwischen E_K und $E_{K'}$.

Außerdem müssen die zugehörigen Blöcke von äquivalenten Kanten ebenfalls äquivalent sein. Genauer bedeutet dies, wenn zwei Kanten $e_K = (b_K^1, b_K^2) \in E_K$ und $e_{K'} = (b_{K'}^1, b_{K'}^2) \in E_{K'}$ äquivalent sind, dann müssen die Blöcke b_K^1 und $b_{K'}^1$ sowie b_K^2 und $b_{K'}^2$ äquivalent sein.

Bei äquivalenten Kanten müssen außerdem die Portnummern übereinstimmen. Die Portnummer gibt den Index eines Ein- oder Ausgangs eines Blocks wieder. Da verschiedene Ports eines Blocks unterschiedliche Semantiken besitzen können, könnte ein Vertauschen von Ports die Semantik eines gesamten Klons verändern.

Äquivalente Ausdehnung Die geometrische Ausdehnung der Teilmodelle K und K' soll ähnlich sein. Für K und K' wird jeweils ein minimales Rechteck r_K und $r_{K'}$ bestimmt, in dem alle Knoten von K bzw. K' enthalten sind. Danach wird

das flächenmäßig kleinere Rechteck r_{min} ermittelt und eine prozentuale Abweichung des größeren vom kleineren Rechteck über die Seitenlängen berechnet:

$$\frac{|laenge(r_K) - laenge(r_{K'})|}{laenge(r_{min})} \leq t_r \quad \wedge \quad \frac{|breite(r_K) - breite(r_{K'})|}{breite(r_{min})} \leq t_r \tag{5.3}$$

Diese Abweichung muss in der Länge und der Breite unter einem Schwellwert t_r liegen.

Diese Eigenschaft wurde der Klondefinition hinzugefügt nachdem sich gezeigt hat, dass diese eine gute Eigenschaft ist um Klone zu erkennen, die durch *Kopieren und Einfügen* entstanden sind. Klone die diese Eigenschaft nicht erfüllen sind im Kontext dieser Arbeit nicht von Interesse. Für t_r wurde der Wert 0,1 bestimmt, dies entspricht einer maximal erlaubten Abweichung von 10% in Höhe und Breite.

Mindestgröße Ein Klon muss außerdem eine Mindestanzahl von n Knoten haben. Dies ist nötig um nicht jeden Knoten als Klon eines äquivalenten Knotens zu erkennen. Die Größe eines Klons ist die Anzahl der Blöcke von V_K oder $V_{K'}$, es gilt also:

$$|V_K| = |V_{K'}| \geq n \tag{5.4}$$

Für n wurde empirisch der Wert fünf bestimmt, da kleinere äquivalente Strukturen meist keine Kopien sondern nur zufälligerweise ähnlich sind.

Umsetzung Dieser Abschnitt beschreibt die Funktionsweise des Algorithmus für die Klonerkennung. Das grundlegende Vorgehen stellt der Pseudocode in Algorithmus 6 dar. Ein Klon wird in diesem Algorithmus repräsentiert als eine Menge von Tupeln, wobei ein Tupel die zwei Blöcke enthält, die in einem Klon als Duplikate voneinander erkannt wurden. Daraus folgt, dass die Menge aller Klone eine Menge von Mengen von Tupeln ist.

Die Menge der äquivalenten Kanten eines Klon wird nicht explizit gespeichert. Diese Eigenschaft wird implizit durch das Durchlaufen des Modells entlang der Kanten sichergestellt.

Bei dem grundlegenden Algorithmus wird jeder Block einmal mit allen anderen Blöcken verglichen. In Zeile 5 werden, beginnend mit zwei Startblöcken, zwei

Algorithmus 6 : Der Grundalgorithmus der Klonerkennung

input : Die Menge aller Blöcke B
output : Die Menge aller Klone K (K ist eine Menge von Mengen von Tupeln)

```
   // Initialisiere die Menge aller Klone
1  K ← ∅;
   // Iteriere über jeden Block b aus der Menge aller Blöcke
      B
2  foreach b ∈ B do
      // Überprüfe ob b in keinem Klon ist
3     if b ∉ {c | k ∈ K, (α, β) ∈ k, c = α ∨ c = β} then
         // Für jeden weiteren Block b' ≠ b
4        foreach b' ∈ B \ {b} do
5           k ← BerechneKlon(b, b', ∅);
6           if k ≠ ∅ then
               // vereinige Klon k mit der Klonmenge K
7              K ← Vereinige(K, k);
8           end
9        end
10    end
11 end
   // Entferne alle Klone die weniger als fünf Blockpaare
      enthalten
12 K ← {c | c ∈ K, |c| ≥ 5};
13 return K
```

Tabelle 5.8: Hilfsfunktionen des Grundalgorithmus zur Klonerkennung

Funktion	Beschreibung
BerechneKlon	Siehe Algorithmus 7.
Vereinige	Siehe Algorithmus 8.

möglichst große *äquivalente* Teilgraphen berechnet. Das Ergebnis dieser Berechnung kann auch die leere Menge sein. Dies ist z. B. der Fall wenn die beiden Startblöcke unterschiedlich sind. In Zeile 7 wird der gefundene Klon mit der Menge aller Klone vereinigt. Am Ende der Berechnungen müssen alle Klone entfernt werden, die zu klein sind. Dies geschieht in Zeile 12.

Algorithmus 7 beschreibt wie, ausgehend von zwei Startblöcken, zwei möglichst große *äquivalente* Teilgraphen bestimmt werden. Der folgende Algorithmus ist prinzipiell eine Tiefensuche mit Backtracking. Im Algorithmus wird das Modell *rückwärts* durchlaufen, ausgehend von den jeweiligen Startblöcken. Dies ist nötig da sich die Kanten in Simulink-Modellen aufspalten können und damit ein Outport mit beliebig vielen Inports verbunden sein kann. Ein Inport kann hingegen immer nur mit einem Outport verbunden sein. Durch die Abarbeitung entgegen der Kantenrichtung ist damit eine eindeutige Zuordnung der nächsten zu überprüfenden Blöcke in den gerade untersuchten Teilnetzen gegeben.

Hierbei ist zu beachten, dass die Reihenfolge der Blöcke der Klon Suche willkürlich ist. Diese werden nach einer beliebigen Reihenfolge aus der Menge aller Blöcke entnommen, um ausgehend von ihnen die Suche zu beginnen. Das Durchsuchen eines Modells von dessen Ende würde eine Verbesserung der Laufzeit bringen, jedoch keinen Vorteil in der Programmierung bewirken. Deshalb wurde diese Optimierung aus Zeitgründen nicht implementiert.

Wird ein noch nicht besuchtes Blockpaar betrachtet, ist der erste Schritt, diese auf *Äquivalenz* zu untersuchen. Als nächstes erhalten die betrachteten Blöcke in Zeile 4 eine Markierung, um im Modell nicht in ein Endlosschleife zu verfallen. Danach werden die Inports mit der selben Nummer betrachtet und die damit verbundenen Outports herausgesucht. Anschließend wird überprüft, ob die zwei gefundenen Outports ebenfalls die selbe Nummer haben. Dies ist wichtig, weil die Portreihenfolge auch semantisch relevant ist und somit ein anderer Ausgang eines Blocks auch eine andere Bedeutung besitzt. Danach erfolgt die rekursive Kontrolle der nächsten verbundenen Blöcke.

Dieses Vorgehen hat ein Problem: Jedes mal wenn ein Teilmodell als Klon erkannt wird, ist es möglich, dass dies nur ein Teilklon eines noch größeren Klons ist. Die Ursache liegt in der ausschließlich rückwärts gerichteten Suche. Um diesem Problem entgegenzuwirken, ist vor dem Hinzufügen des Klons zu der Menge

Algorithmus 7 : Funktion BerechneKlon

input : Zwei Blöcke b und b' die auf ihre Kloneigenschaft untersucht werden sollen
input : Die Menge aller besuchten Blöcke V
output : Der größte Klon, ausgehend von Block b und b'

```
// Starte mit einem leeren Klon
1  k ← ∅;
2  if b ≠ b' then
       // Wurde b und b' schon besucht?
3      if b ∉ V ∧ b' ∉ V then
           // Markiere b und b' als besucht
4          V ← V ∪ {b, b'};
           // Sind b und b' äquivalent?
5          if BlockÄquivalenz(b, b') then
               // Füge b und b' zu dem Klon k hinzu
6              k ← k ∪ {(b, b')};
               // Iteriere über alle Inports von b und b' in der
                  selben Reihenfolge
7              foreach i ∈ Inports(b), i' ∈ Inports(b') ∧
               PortNummer(i) = PortNummer(i') do
                   // Hole die mit den Inports verbundenen Outports
8                  o ← VerbundenerOutport(i);
9                  o' ← VerbundenerOutport(i');
                   // Auch die Outports müssen äquivalent sein
10                 if PortNummer(o) = PortNummer(o') then
                       // Hole die mit den Inports verbundenen
                          Blöcke
11                     a ← PortZuBlock(o);
12                     a' ← PortZuBlock(o');
                       // Mache rekursiv weiter
13                     k ← k ∪ BerechneKlon (a, a', V);
14                 end
15             end
16         end
17     end
18 end
19 return k
```

Tabelle 5.9: Hilfsfunktionen der Funktion BerechneKlon

Funktion	Beschreibung
BlockÄquivalenz	Gibt *True* zurück wenn die zwei übergebenen Blöcke die Äquivalenzeigenschaft erfüllen (siehe Tabelle 5.7), sonst *False*.
Inports	Gibt die Menge aller Inports für einen übergebenen Block zurück.
PortNummer	Gibt die Nummer eines In- oder Outports zurück. Durch diese Nummer kann ein Port an einem Block eindeutig identifiziert werden.
VerbundenerOutport	Gibt den Outport zurück, der mit dem übergebenen Inport verbunden ist.
PortZuBlock	Gibt den Block zurück zu dem der übergebene In- oder Outport gehört.

aller Klone zu überprüfen, ob schon ein Teilklon von diesem besteht. Dies leistet Algorithmus 8.

Dieser Algorithmus errechnet zuerst die Menge aller bisherigen Klone, die eine Überschneidung mit dem neuen Klon haben (ab Zeile 2). Jeder Klon dieser Menge wird aus der Menge aller Klone entfernt und mit dem neuen Klon vereint. Dabei muss überprüft werden, ob die Blockreihenfolge in den Tupeln vertauscht werden muss (Zeile 7). Außerdem werden die Bounding Boxes des neuen Klon, vereint mit dem überschneidenden Klon, überprüft. Nur wenn die Abweichung der Länge und Breite kleiner als 10% ist, wird der neue Klon mit dem alten vereint.

Am Ende wird der neue Klon der Menge aller Klone hinzugefügt, wenn er aus mindestens zwei kopierten Blöcken besteht und die Abweichung der Bounding Boxes in der Länge und Breite kleiner als 10% ist.

Die Überprüfung auf eine Mindestgröße von zwei geschieht nur zu Optimierungszwecken, um die Menge aller gefundenen Klone möglichst gering zu halten. Diese Überprüfung hat nichts mit der Kloneigenschaft *Mindestgröße* zu tun. Die Überprüfung der Mindestgröße findet in Algorithmus 6 statt.

Beispiel In den nächsten Absätzen wird anhand eines Beispiels gezeigt, wie das Erkennen eines Klons abläuft. Die Ausgangssituation ist in Abbildung 5.2 zu sehen. Da der Algorithmus jeden Block mit allen anderen Blöcken auf die

Algorithmus 8 : Funktion Vereinige

input : Ein neu entdeckter Klon k (eine Menge von Tupeln)
input : Die Menge aller bisher gefundenen Klone K (eine Menge von Mengen von Tupeln)
output : Die Menge aller Klone K vereinigt mit k

```
   // Diese Menge speichert alle Klone, die sich mit dem
      neuen Klon überschneiden
1  mb ← ∅;
   // Finde alle Klone, die sich mit dem neuen Klon
      überschneiden
2  foreach (b, b') ∈ k do
      // mb ist die Menge aller Klone, die das Tupel (b, b')
         oder (b', b) enthalten
3     mb ← mb ∪{c | c ∈ K, (α, β) ∈ c, (b = α ∧ b' = β) ∨ (b = β ∧ b' = α)};
4  end
   // Vereinige alle gefundenen Klone mit dem neuen Klon
5  foreach k_alt ∈ mb do
      // Entferne den alten Klon aus der Menge aller Klone
6     K ← K \ {k_alt};
      // Überprüfe ob die Blockreihenfolge in den Tupeln
         vertauscht werden muss
7     k_alt ← ÜberprüfeBlockreihenfolge(k_alt);
8     if VergleicheBoundingBoxes(k ∪ k_alt) then
         // Vereinige den alten und neuen Klon
9        k ← k ∪ k_alt;
10    end
11 end
   // Füge den neuen Klon der Menge aller Klone hinzu
12 if |k| ≥ 2 ∧ VergleicheBoundingBoxes(k) then
13    K ← K ∪ k;
14 end
15 return K
```

Tabelle 5.10: Hilfsfunktionen der Funktion Vereinige

Funktion	Beschreibung
ÜberprüfeBlockreihenfolge	Überprüft ob die Blockreihenfolge in den Tupeln eines Klons vertauscht werden muss. Wenn die Reihenfolge vertauscht werden muss, wird eine vertauschte Kopie des übergebenen Klons zurückgegeben. Andernfalls wird der Originalklon zurückgegeben.
VergleicheBoundingBoxes	Diese Funktion bekommt einen Klon als Parameter übergeben. Für jeden der beiden Klonteile wird die Ausdehnung der umgebenden Bounding-Box berechnet und anschließend miteinander verglichen. Wenn die Abweichung größer als 10% ist wird *False* zurückgegeben, sonst *True* (siehe Kapitel 5.2.3).

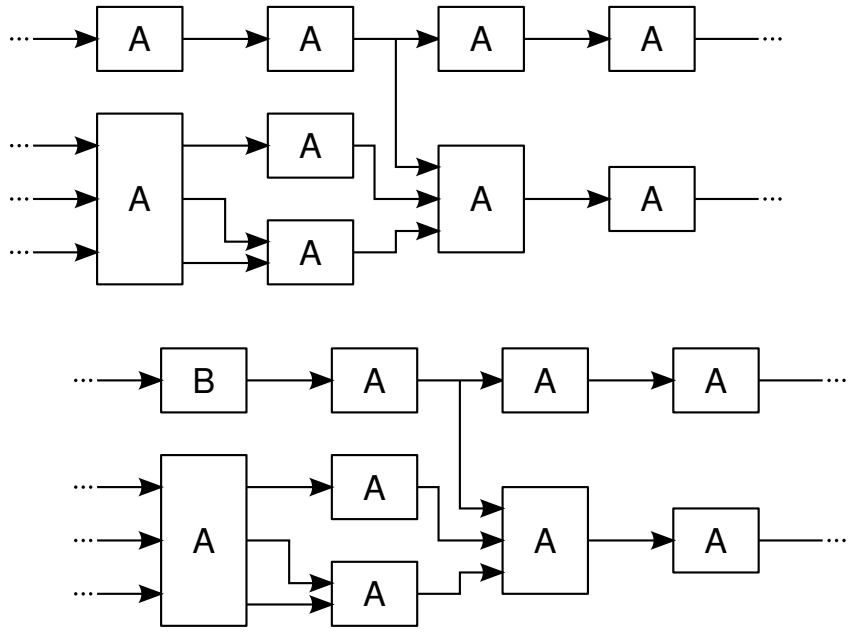


Abbildung 5.2: Ein unerkanntes Klonpaar.

Kloneigenschaften überprüft, untersucht er zu einem bestimmten Zeitpunkt die beiden Blöcke, die in Abbildung 5.3 grau hinterlegt sind. Nachdem die Klonei-

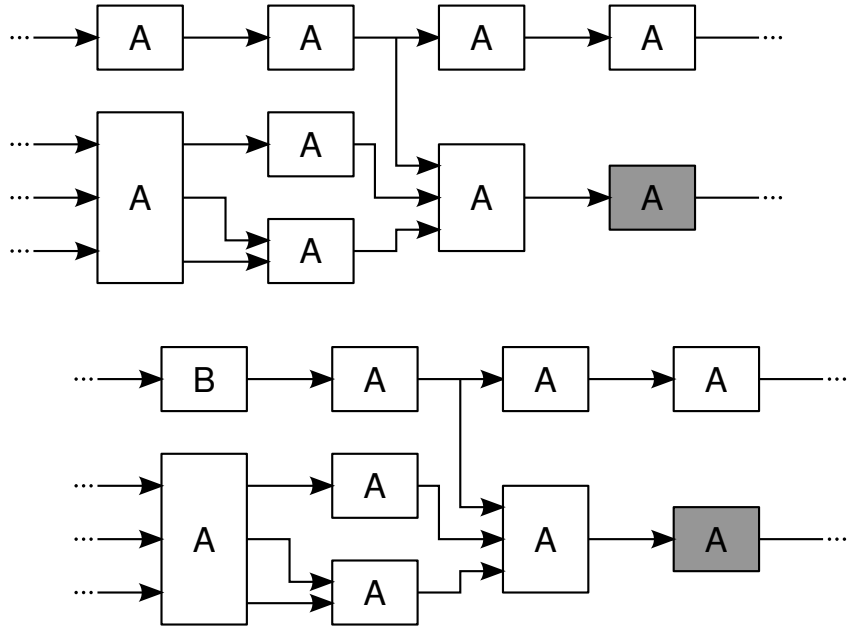


Abbildung 5.3: Beginn der Klonsuche.

genschaft für die markierten Blöcke festgestellt wurde, fährt der Algorithmus mit deren Vorgängern fort, bis die untersuchten Blöcke die Kloneigenschaft nicht mehr erfüllen. Der bisher erkannte Klon (siehe Abbildung 5.4) wird anschließend der Menge aller Klone hinzugefügt.

Zu einem späteren Zeitpunkt untersucht der Algorithmus die zwei Blöcke, die in Abbildung 5.5 gekennzeichnet sind. Ausgehend von diesen Blöcken wird wieder versucht, das größtmögliche äquivalente Netz zu finden. Dieses ist in Abbildung 5.6 markiert. Wenn dieser Klon der Menge aller Klone hinzugefügt wird, stellt der Algorithmus fest das ein äquivalentes Blockpaar bereits zu einem existierenden Klon gehört. Dadurch wird der neue Klon mit dem existierenden zusammengefügt und damit der gesamte Klon erkannt. Das Endergebnis ist in Abbildung 5.7 dargestellt.

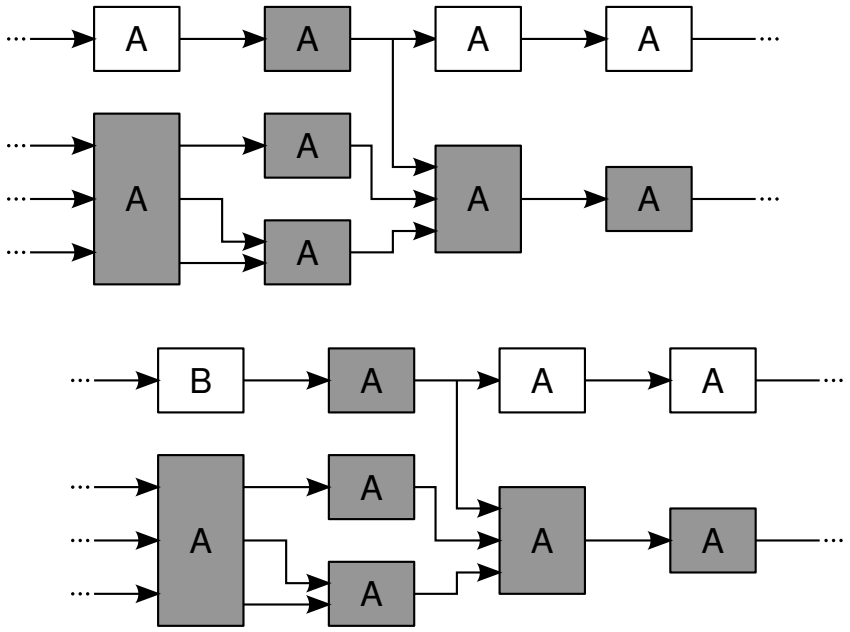


Abbildung 5.4: Der erste erkannte Klon.

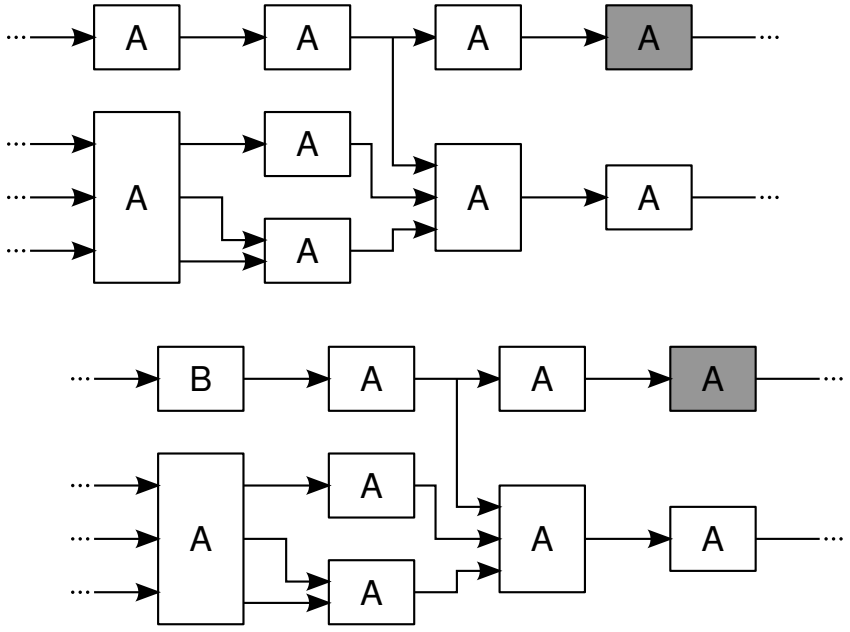


Abbildung 5.5: Erneute Klonsuche.

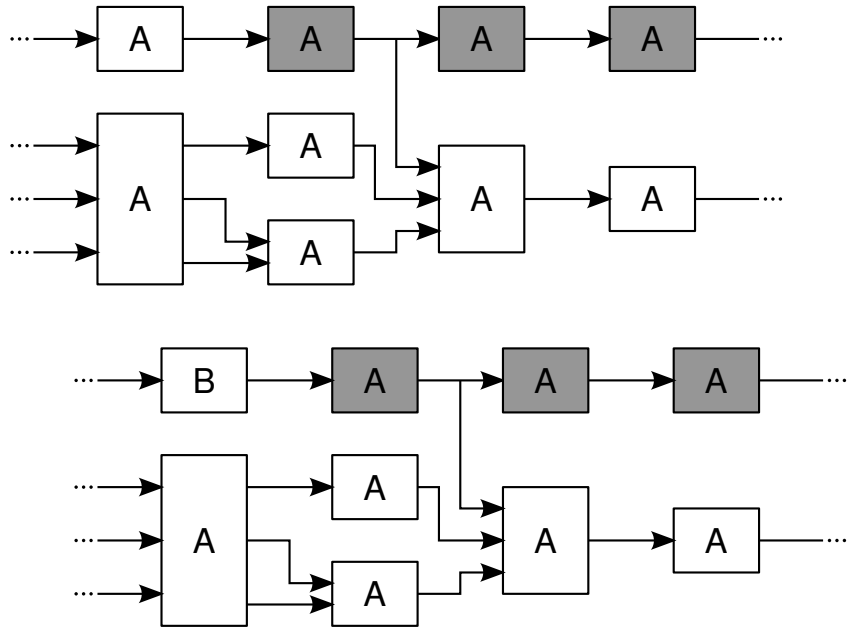


Abbildung 5.6: Der zweite erkannte Klon.

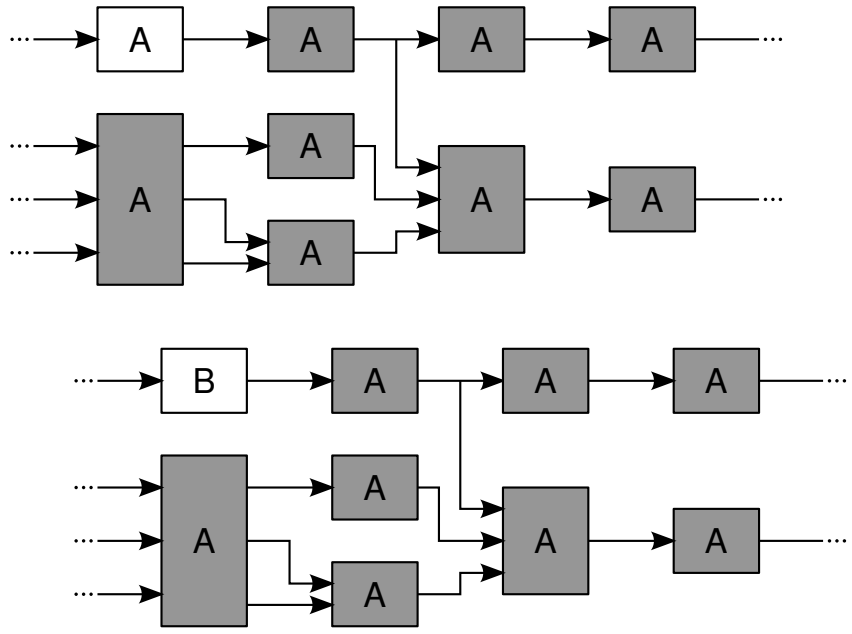


Abbildung 5.7: Das Endergebnis der Klonsuche.

Blockinstabilität

Diese Metrik versucht die Wahrscheinlichkeit zu messen, mit der ein Block geändert wird. Diese Metrik entstammt der Arbeit [MA05, S. 270–271], die wiederum Ideen aus dem Artikel [Mar94, S. 6] aufgreift.

Definition: Blockinstabilität Die *Blockinstabilität* für einen Block v wird definiert in [MA05, S. 271] als:

$$IOB(v) = \frac{CBB_e(v)}{CBB_e(v) + CBB_a(v)} \tag{5.5}$$

Dabei ist $CBB_e(v)$ die Anzahl aller Quellblöcke des Blocks v und $CBB_a(v)$ die Anzahl aller Zielblöcke des Blocks v. Ein Quellblock ist ein Vorfahre des aktuellen Blocks, er ist also mit einem Inport des aktuellen Blocks verbunden. Dann ist ein Zielblock ein Nachfahre des aktuellen Blocks, er ist also mit einem Outport des aktuellen Blocks verbunden.

Ein Wert von eins entspricht dabei einer hohen Wahrscheinlichkeit, dass ein Block geändert wird, da er nur von vorhergehenden Blöcken abhängt. Ein Wert von null entspricht hingegen einer niedrigen Wahrscheinlichkeit einer Änderung, da er von keinem vorhergehenden Block abhängt.

Die Intention dieser Metrik ist 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 das ein Block geändert wird, je mehr er von anderen Blöcken abhängt.

Beispiel Anhand des Simulink-Modells in Abbildung 5.8 soll die Berechnung der Blockinstabilität für einen Block, verdeutlicht werden. Im folgenden wird die Blockinstabilität für den grau hinterlegten Add-Block berechnet.

Die zugehörigen Quellblöcke des Add-Blocks sind die Blöcke *Product* und *In1*, damit gilt $CBB_e(Add) = 2$. Der zugehörige Zielblock ist der Block *Product1*, womit $CBB_a(Add) = 1$ ist. Nun lässt sich die Blockinstabilität für den Add-Block berechnen:

$$IOB(Add) = \frac{CBB_e(Add)}{CBB_e(Add) + CBB_a(Add)} = \frac{2}{2+1} = \frac{2}{3} \tag{5.6}$$

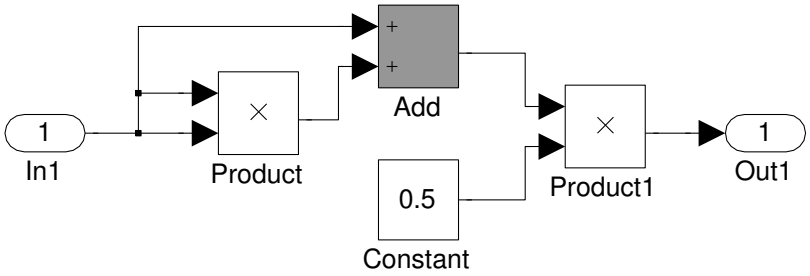


Abbildung 5.8: Beispielmodell für die Berechnung der Blockinstabilität.

Umsetzung Das Vorgehen zur Berechnung der Blockinstabilität wird durch Algorithmus 9 erläutert. Dieser Algorithmus iteriert über alle Blöcke. Zuerst ermittelt er die Anzahl der Quell- und Zielblöcke zu dem aktuellen Block.

Ab Zeile 5 wird die Anzahl der Quellblöcke ermittelt. Dabei ist zu beachten, dass einige Blöcke gesondert behandelt werden müssen. Ein DataStoreMemory-Block hat keine Inports, aber eine Anzahl von verbundenen DataStoreWrite-Blöcken. Ein DataStoreRead-Block hat ebenfalls keinen Inport, aber genau einen DataStoreMemory-Block, aus dem er lesen kann. Ein From-Block hat ebenfalls keinen Inport, dafür aber genau einen Goto-Block, von dem er benutzt wird.

Ab Zeile 14 wird die Anzahl der Zielblöcke ermittelt. Auch hier müssen einige Blöcke speziell behandelt werden. Ein DataStoreMemory-Block hat keine Outports, aber eine Menge von DataStoreRead-Blöcke, die von diesem lesen. Ein DataStoreWrite-Block hat keine Outports, aber genau einen DataStoreMemory-Block, in den er schreiben kann. Ein Goto-Block hat eine Menge von From-Blöcken, die er anspringt.

Wenn der Block Quell- und/oder Zielblöcke besitzt, wird der Blockinstabilitätswert für den aktuellen Block nach der Formel 5.5 berechnet.

Zum Schluss wird der durchschnittliche Blockinstabilität Wert über alle Blöcke errechnet und zurückgegeben.

5.2.4 Messung der Effizienz

In diesem Unterkapitel werden Metriken zur Bewertung der Effizienz von Simulink-Modellen vorgestellt. Hierfür werden mit dem Goal/Question/Metric-Ansatz einzelne Metriken identifiziert, die im weiteren Verlauf erläutert werden.

Algorithmus 9 : Grundalgorithmus zur Bestimmung der Blockinstabilität

input : Die Menge aller Blöcke B
output : Die durchschnittliche Blockinstabilität, über alle Blöcke aus B

```
1  summeIOB ← 0;
2  foreach b ∈ B do
3      quellen ← 0;
4      ziele ← 0;
5      if AnzahlInports(b) = 0 then
6          if Blocktyp(b) = DataStoreMemory then
7              quellen ← AnzahlDataStoreWriteBlöcke(b);
8          else if Blocktyp(b) = DataStoreRead then
9              quellen ← 1;
10         else if Blocktyp(b) = From then
11             quellen ← 1;
12         end
13     else quellen ← AnzahlInports(b);
14     if AnzahlOutports(b) = 0 then
15         if Blocktyp(b) = DataStoreMemory then
16             ziele ← AnzahlDataStoreReadBlöcke(b);
17         else if Blocktyp(b) = DataStoreWrite then
18             ziele ← 1;
19         else if Blocktyp(b) = Goto then
20             ziele ← AnzahlFromBlöcke(b);
21         end
22     else
23         Zielblöcke ← ∅;
24         foreach outport ∈ HoleOutports(b) do
25             foreach linie ∈ HoleLinien(outport) do
26                 Zielblöcke ← Zielblöcke ∪ HoleZielBlock(linie);
27             end
28         end
29         ziele ← |Zielblöcke|;
30     end
31     if quellen + ziele > 0 then
32         summeIOB ← summeIOB + (quellen/(quellen + ziele));
33     end
34 end
35 return summeIOB/|B|;
```

Tabelle 5.11: Hilfsfunktionen des Grundalgorithmus zur Bestimmung der Blockinstabilität

Funktion	Beschreibung
AnzahlInports	Gibt die Anzahl von Inports für einen übergebenen Block zurück.
Blocktyp	Ermittelt für einen Block den Blocktyp und gibt diesen als Zeichenkette zurück.
AnzahlDataStoreWriteBlöcke	Gibt die Anzahl von DataStoreWrite-Blöcken zurück, die mit dem übergebenen DataStoreMemory-Block assoziiert sind.
AnzahlDataStoreReadBlöcke	Gibt die Anzahl von DataStoreRead-Blöcken zurück, die mit dem übergebenen DataStoreMemory-Block assoziiert sind.
AnzahlOutports	Gibt die Anzahl von Outports für einen übergebenen Block zurück.
AnzahlFromBlöcke	Gibt die Anzahl von From-Blöcken zurück, die mit dem übergebenen Goto-Block assoziiert sind.
HoleOutports	Gibt die Menge aller Outports für den übergebenen Block zurück.
HoleLinien	Gibt die Menge aller Kanten zurück, die den übergebenen Outport mit anderen Inports verbindet.
HoleZielBlock	Gibt den Block zurück, zu dem die übergebene Kante führt.

Goal Eine gute Effizienz von Simulink-Modellen.

Question Gibt es unnötige Berechnungen in einem Modell?

Metric Verfallene Ergebnisse.

Die Implementation dieser Metrik wird in dem nächsten Unterkapitel beschrieben.

Verfallene Ergebnisse

Das Ziel dieser Metrik ist das Finden von nicht verwendeten Berechnungen. Ein Programm, das viele unnötigen Berechnungen durchführt, ist offensichtlich nicht so effizient wie ein Programm mit nur nötigen Berechnungen.

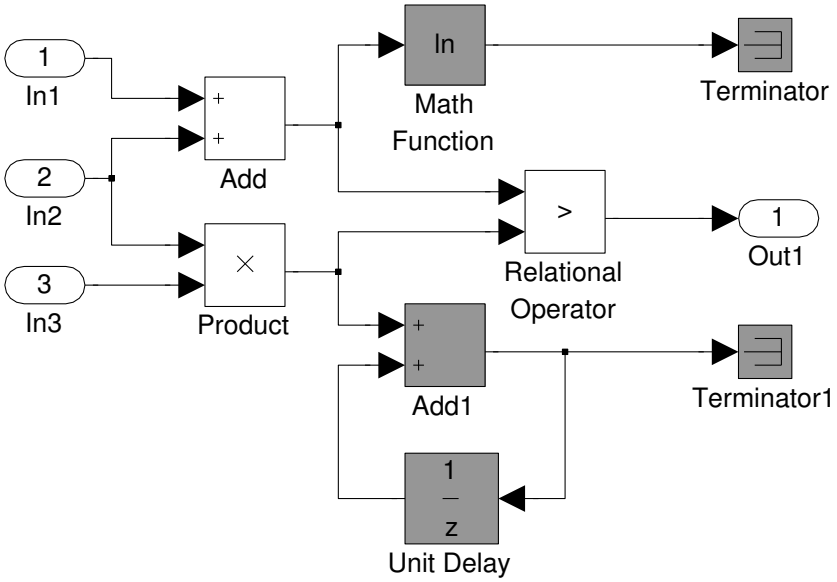


Abbildung 5.9: Ein Beispiel für nicht genutzte Berechnungen.

In Abbildung 5.9 ist ein Beispiel zu sehen, bei dem die Ergebnisse der grau hinterlegten Blöcke nicht genutzt werden.

Definition: Verfallene Ergebnisse; aktiver/inaktiver Outport, Block, Signal In diesem Abschnitt werden alle Begriffe definiert, die für diese Metrik benötigt werden.

In dieser Metrik ist ein verfallenes Ergebnis eine berechnete Information, die nicht verwendet wird. Dies ist das Selbe, wie wenn alle Signale (siehe Abschnitt 4.2.3) eines Outports nicht verwendet werden.

In Tabelle 5.12 werden die Begriffe *aktives/inaktives Signal*, *aktiver/inaktiver Outport* und *aktiver/inaktiver Block* definiert.

Tabelle 5.12: Begriffsdefinitionen

Begriff	Beschreibung
Aktiver Outport	Mindestens ein Signal von diesem Outport ist aktiv
Inaktiver Outport	Alle Signale von diesem Outport sind inaktiv
Aktiver Block	Dieser Block besitzt mindestens einen aktiven Outport
Inaktiver Block	Dieser Block besitzt nur inaktive Outports
Aktives Signal	Ein Signal, das in einem aktiven Block endet
Inaktives Signal	Ein Signal, das in einem inaktiven Block endet

Außerdem ist ein Terminator-Blöcke ein *immer inaktiver* Block. Ein Block, der kein Terminator-Block ist, der Inports aber keine Outports besitzt, ist ein *immer aktiver* Block.

Damit lässt sich definieren, was ein *verfallenes Ergebnis* ist:

Ein verfallenes Ergebnis ist ein Outport, von dem alle Signale, direkt oder indirekt, in einem Terminator-Block enden. Diese Metrik berechnet die Anzahl dieser Outports.

Umsetzung Für die Umsetzung wird eine zusätzliche Datenstruktur verwendet, um Informationen über Signale zu speichern. Diese Datenstruktur wird im weiteren als *Signal-Konnektor* bezeichnet. Diese Signal-Konnektoren werden für jeden Block angelegt, der Signale generiert oder beendet. Ein Signal-Konnektor speichert den zugehörigen Block (für den er angelegt wurde), alle Signale, die in diesem Block enden und welche Outports aktiv oder inaktiv sind.

In Algorithmus 10 ist der grundlegende Algorithmus für das Finden von nicht verwendeten Ergebnissen dargestellt. Zuerst generiert der Algorithmus für jeden Block der ein Signal generiert oder beendet einen Signal-Konnektor, wobei alle Outports als inaktiv markiert sind.

Tabelle 5.13: Signal-Konnektor Elemente

Element	Beschreibung
Block	Der mit diesem Signal-Konnektor assoziierte Block
Eingehende Signale	Alle Signale, die im zugehörigen Block enden
Aktive Outports	Ein Feld das für jeden Outport markiert, ob dieser aktiv ist oder nicht

Algorithmus 10 : Der Grundalgorithmus zur Bestimmung der verfallenen Ergebnisse

input : Die Menge aller Signale S
output : Die Anzahl von nicht verwendeten Ergebnissen

```
1  SK ← InitialisiereSignalKonnektoren(S);
2  foreach sk ∈ SK do
3      if IstAktiverEndpunkt(sk) then
4          MarkiereAktiveSignalKonnektoren(SK, sk);
5      end
6  end
7  inaktiveOutports ← 0;
8  foreach sk ∈ SK do
9      inaktiveOutports ← inaktiveOutports + AnzahlInaktiverOutports(sk);
10 end
11 return inaktiveOutports;
```

Tabelle 5.14: Hilfsfunktionen des Grundalgorithmus zur Bestimmung der verfallenen Ergebnisse

Funktion	Beschreibung
InitialisiereSignalKonnektoren	Berechnet und initialisiert alle Signal-Konnektoren, ausgehend von der Menge aller Signale.
IstAktiverEndpunkt	Gibt *True* zurück wenn der übergebene Signal-Konnektor zu einem aktiven Endpunkt gehört, sonst *False*.
MarkiereAktiveSignalKonnektoren	Siehe Algorithmus 11.
AnzahlInaktiverOutports	Die Anzahl von Outports eines Blocks, der zu dem übergebenen Signal-Konnektor gehört, die nicht als aktiv markiert wurden.

Ab Zeile 3 werden alle Signal-Konnektoren gesucht, die zwar eingehende Signale aber keine Outports haben und kein Terminator-Block sind. Diese Blöcke sind per Definition aktive Blöcke. Ausgehend von diesen Blöcken werden alle Outports markiert, die einen Einfluss auf diese Blöcke haben.

In der Schleife in Zeile 8 wird für jeden Signal-Konnektor die Anzahl der Inports aufsummiert, die nicht als aktiv markiert wurden. Die Zahl der inaktiven Outports ist die Anzahl von Signalen, die in diesem Simulink-Modell nicht verwendet werden und damit die Anzahl der verfallenen Ergebnisse. Diese Zahl wird anschließend in Zeile 11 zurückgegeben.

Algorithmus 11 markiert alle Outports, die von einem Signal-Konnektor erreicht werden können. Dabei iteriert der Algorithmus über alle Signale, die im momentanen Signal-Konnektor enden. Dann iteriert er über alle Blöcke, die ein Signal durchlaufen und überprüft, ob es zu diesem Block ebenfalls einen Signal-Konnektor gibt. Wenn es ihn gibt, markiert der Algorithmus den zugehörigen Outport als aktiv.

Anschließend wird überprüft, ob dieser Signal-Konnektor schon besucht wurde. Wenn der Signal-Konnektor davor inaktiv war, ist es das erste Mal, dass der Algorithmus diesen Signal-Konnektor besucht.

In diesem Fall, wird der aktuelle Outport und Signal-Konnektor als aktiv markiert und der Algorithmus fährt rekursiv mit diesem Signal-Konnektor fort. Durch diesen rekursiven Schritt werden alle Outports (Signale) und Signal-Konnektoren markiert, die der aktuelle Signal-Konnektor benötigt.

Wenn der Algorithmus zu einem Signal-Konnektor kommt der schon aktiv ist, dann wurden schon alle Outports (Signale) und Signal-Konnektoren als aktiv markiert, die der aktuelle Signal-Konnektor benötigt. In diesem Fall muss nur noch der aktuell benötigte Outport als aktiv markiert werden.

5.2.5 Messung der Komplexität

In diesem Unterkapitel werden Metriken zur Bewertung der Komplexität von Simulink-Modellen vorgestellt. Hierfür werden mit dem Goal/Question/Metric-Ansatz einzelne Metriken identifiziert, die im weiteren Verlauf erläutert werden.

Goal Eine Bewertung der Komplexität von Simulink-Modellen.

Algorithmus 11 : Funktion MarkiereAktiveSignalKonnektoren

input : Die Menge aller Signal-Konnektoren SK
input : Ein zu bearbeitender Signal-Konnektor sk

```
   // Iteriere über alle Signale, die in sk enden
1  foreach s ∈ HoleEingehendeSignale(sk) do
     // Iteriere über alle Outports, die das Signal
        durchläuft
2    foreach o ∈ HoleDurchlaufeneOutports(s) do
       // Hole den Block zu dem Outport gehört
3      b ← HoleBlock(o);
4      if HatSignalKonnektor(SK, b) then
         // Hole den Signal-Konnektor, der zu Block gehört
5        sk' ← HoleSignalKonnektor(SK, b);
         // Wenn schon einer der Outports des Signal-
            Konnektor als aktiv markiert wurde, markiere
            nur den aktuellen Outport
6        if IstAktiv(sk') then
7          MarkiereOutportAlsAktiv(sk', o);
8        else
           // Wenn der Signal-Konnektor noch nicht markiert
              wurde, markiere ihn und den aktuellen Outport
              als aktiv und fahre rekursiv fort
9          MarkiereSignalKonnektorAlsAktiv(sk');
10         MarkiereOutportAlsAktiv(sk', o);
11         MarkiereAktiveSignalKonnektoren(SK, sk');
12       end
13     end
14   end
15 end
```

Tabelle 5.15: Hilfsfunktionen der Funktion MarkiereAktiveSignalKonnektoren

Funktion	Beschreibung
HoleEingehendeSignale	Gibt die Menge aller Signale zurück, die in dem übergebenen Signal-Konnektor enden.
HoleDurchlaufeneOutports	Gibt die Liste von Outports zurück, die ein Signal durchläuft.
HoleBlock	Gibt den Block zurück, der zu dem übergebenen Outport gehört.
HatSignalKonnektor	Gibt *True* zurück wenn der übergebene Block einen zugehörigen Signal-Konnektor hat, sonst *False*.
HoleSignalKonnektor	Gibt den Signal-Konnektor zurück, der mit dem übergebenen Block assoziiert wird.
IstAktiv	Gibt *True* zurück wenn der übergebene Signal-Konnektor aktiv ist, sonst *False*.
MarkiereOutportAlsAktiv	Markiert den übergebenen Outport als aktiv.
MarkiereSignalKonnektorAlsAktiv	Markiert den übergebenen Signal-Konnektor als aktiv.

Question Wie verständlich/intuitiv ist das Simulink-Modell?

Metric Anzahl Zyklen.

Metric Durchschnittliche Signallänge.

Question Wie schwer ist es einzelne Subsysteme zu verstehen?

Metric Slicing.

Die Implementation dieser Metriken wird in den nächsten Unterkapiteln beschrieben.

Anzahl Zyklen

Das Ziel dieser Metrik ist das Zählen von Zyklen in einem Simulink-Modell. In dem Buchkapitel [Men08, S. 127] wird beschrieben, dass zyklische Modellteile schwerer zu verstehen sind, als sequentielle Modellteile. Ein Signal in einem Zyklus hat nicht mehr nur Auswirkungen auf folgende, sondern auch auf vorhergehende Blöcke. Damit kann die Änderung eines Blocks auch die vorangegangenen Blöcke beeinflussen. Auch hat ein Zyklus einen Zustand der sich mit jeder Iteration ändern kann. Dies erschwert das Erfassen von Zusammenhängen und Abhängigkeiten.

Nach der Meinung des Autors sollte damit eine hohe Anzahl von Zyklen auf eine erhöhte Komplexität hinweisen.

Definition: Zyklus Dieser Abschnitt definiert was ein Zyklus in einem Simulink-Modell ist.

Ein Zyklus ist ein Pfad, der den selben Start- und Endknoten (bzw. Block) besitzt.

Umsetzung Dieser Abschnitt erläutert die Umsetzung des Algorithmus zum Aufspüren von Zyklen. Algorithmus 12 beschreibt das grundlegende Vorgehen. Als Eingabe benötigt der Algorithmus die Menge aller Blöcke, die untersucht werden sollen. Anschließend iteriert der Algorithmus über jeden Block aus der Menge der übergebenen Blöcke. In Zeile 3 wird die Menge der Zyklen errechnet, die von dem aktuellen Block aus erreicht werden können und zu der Menge aller

Algorithmus 12 : Der Grundalgorithmus der Zyklensuche

input : Die Menge aller Blöcke B
output : Die Anzahl der gefundenen Zyklen

```
   // Z speichert alle gefundenen Zyklen
1  Z ← ∅;
   // Iteriere über alle Blöcke
2  foreach b ∈ B do
       // Erweitere Z um die Zyklen die von b aus gefunden
          werden können
3      Z ← Z ∪ FindeZyklen(b, ∅);
4  end
   // Gib die Anzahl aller Zyklen zurück
5  return |Z|;
```

Tabelle 5.16: Hilfsfunktionen des Grundalgorithmus zur Zyklensuche

Funktion	Beschreibung
FindeZyklen	Siehe Algorithmus 13.

Zyklen hinzugefügt. Ein Zyklus wird in diesem Algorithmus als eine Menge von Blöcken dargestellt. Am Ende gibt der Algorithmus die Anzahl aller gefundenen Zyklen zurück.

Algorithmus 13 zeigt, wie die Menge von Zyklen gefunden wird, die von einem ausgewählten Block erreicht werden können. Der Algorithmus benötigt als Eingabe einen Block und eine Menge von bereits besuchten Blöcken. Die Menge der bereits besuchten Blöcke speichert den bisher zurückgelegten Pfad bis zu diesem Block. Der Algorithmus überprüft in Zeile 2, ob der übergebene Block schon besucht wurde. Wenn er in der Menge der bereits besuchten Blöcke enthalten ist, wurde ein Zyklus gefunden. Der Zyklus muss dann noch in der Menge aller besuchten Blöcke gefunden werden. Dies erledigt die Funktion *ExtrahiereZyklus* in Zeile 3.

Diese Funktion geht rückwärts durch die Menge der besuchten Blöcke $(b_1, ..., b_n)$, solange bis der letzte Block b_n erneut an Stelle i auftaucht ($b_i = b_n$ und $1 \leq i \leq n$). Dann kann der Zyklus $(b_i, ..., b_{n-1})$ in der Menge aller Zyklen gespeichert werden.

Algorithmus 13 : Funktion FindeZyklen

input : Ein Block b von dem aus Zyklen gesucht werden
input : Eine Liste von Blöcken V, die den bisher durchlaufenen Pfad enthält
output : Eine Menge von Zyklen

```
   // Starte mit einer leeren Menge von Zyklen
1  Z ← ∅;
2  if b ∈ V then
       // In diesem Fall wurde ein Zyklus gefunden
       // Speichere den gefundenen Zyklus
3      z ← ExtrahiereZyklus(V);
4      Z ← Z ∪ {z};
5  else if HatNachfolger(b) then
       // Füge den aktuellen Block der Menge aller besuchten
          Blöcke hinzu
6      V ← V ∪ {b};
       // Gehe in die Rekursion mit allen Nachfolgern des
          aktuellen Blocks
7      foreach b' ∈ Nachfolger(b) do
8          Z ← Z ∪ FindeZyklen(b', V);
9      end
10 end
11 return Z;
```

Tabelle 5.17: Hilfsfunktionen der Funktion FindeZyklen

Funktion	Beschreibung
ExtrahiereZyklus	Berechnet aus der Menge aller besuchten Blöcke, die Liste von Blöcken die den erkannten Zyklus bilden.
HatNachfolger	Gibt *True* zurück wenn der übergebene Block über seine Outports mit mindestens einem anderen Block verbunden ist, sonst *False*.
Nachfolger	Gibt die Menge aller Blöcke zurück, die mit dem übergebenen Block über seine Outports verbunden sind.

In Zeile 5 überprüft der Algorithmus, ob der aktuelle Block Nachfolger besitzt. Der aktuelle Block wird sodann in die Liste der besuchten Blöcke aufgenommen und alle Nachfolger werden rekursiv abgearbeitet. Alle zurückgegebenen Zyklen werden in einer Menge gespeichert und am Ende zurückgegeben.

Beispiel Anhand des Beispiels in Abbildung 5.10 soll gezeigt werden, wie das Erkennen von Zyklen abläuft.

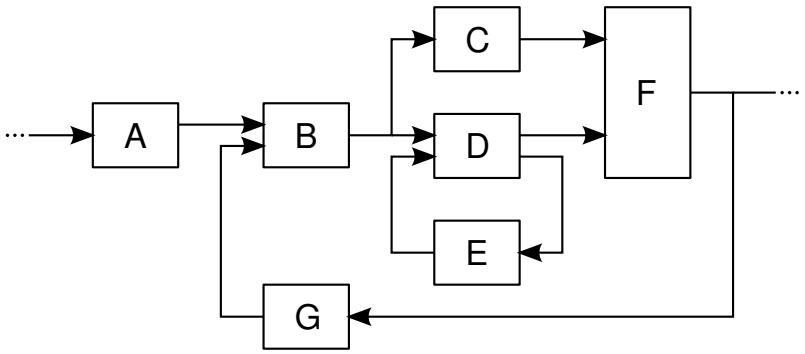


Abbildung 5.10: Ausschnitt eines Modells mit mehreren Zyklen.

Der Ausgangsknoten ist Knoten *A*. Ausgehend von diesem Knoten werden rekursiv dessen Nachfolger betrachtet. In Tabelle 5.18 sind die einzelnen Schritte des Algorithmus aufgelistet. Dabei kennzeichnen die Buchstaben zwischen den geschweiften Klammern die Liste der bisher besuchten Blöcke. Die Buchstaben hinter dem Bindestrich kennzeichnen die nächsten Blöcke im Modell.

Tabelle 5.18: Verlauf der Rekursion während der Zyklensuche

Rekursionstiefe	1. Rekursionsweg	2. Rekursionsweg	3. Rekursionsweg
0	{A} – B		
1	{A, B} – C, D		
2	{A, B, C} – F	{A, B, D} – E, F	
3	{A, B, C, F} – G	{A, B, **D**, E} – **D**	{A, B, D, F} – G
4	{A, **B**, C, F, G} – **B** **B, C, F, G**	**D, E**	{A, **B**, D, F, G} – **B** **B, D, F, G**

Da es nach den Blöcken *B* und *D* mehrere nachfolgende Blöcke gibt, teilt sich der Rekursionsweg auf. Der Algorithmus fährt so lange fort, bis er einen be-

reits besuchten Block findet, oder ein Block keine nachfolgenden Blöcke mehr besitzt.

In der dritten Rekursionstiefe findet der Algorithmus, im zweiten Rekursionsweg, den bereits besuchten Block *D*. Der Zyklus kann anschließend der Liste der besuchten Blöcke entnommen werden, indem die besuchten Blöcke vor Block *D* entfernt werden. In der vierten Rekursionstiefe werden der erste und dritte Rekursionsweg äquivalent bearbeitet. Am Ende hat der Algorithmus die folgenden Zyklen entdeckt:

- *B*, *C*, *F*, *G*
- *D*, *E*
- *B*, *D*, *F*, *G*

Durchschnittliche Signallänge

Das Ziel dieser Metrik ist das Messen von Signallängen. Hierbei ist die Meinung des Autors zu Signallängen:

Je länger ein Signal ist, desto schwieriger ist es, den Einfluss des Signals auf das gesamte Modell zu verstehen. Eine erhöhte durchschnittliche Signallänge sollte somit auch eine erhöhte Komplexität implizieren.

Definition: Signallänge Für eine genaue Definition von Signalen siehe Kapitel 4.2.3.

Die Länge eines Signals ist über die Anzahl der Kanten in dessen Pfad definiert. Hierbei ist zu berücksichtigen, dass sich ein Signal aufteilen kann. Damit kann ein Signal mehrere Längen besitzen. Dies passiert immer dann, wenn der Outport eines Blocks mit mehreren Inports verbunden ist. Es wird also dasselbe Signal an mehrere Blöcke weitergeleitet.

Algorithmus 14 : Der Grundalgorithmus für die durchschnittlichen Signallänge

input : Die Menge aller Signale
output : Die durchschnittliche Länge aller Signale

```
1  länge ← 0;
   // Iteriere über alle Signale
2  foreach s ∈ Signale do
3      istErsterBlock ← true;
       // Iteriere über alle Blöcke von dem Signal
4      foreach outport ∈ s do
           // Hole den zum Outport gehörenden Block
5          b ← HoleBlock(outport);
           // Zähle den ersten Block nicht mit
6          if not istErsterBlock then
               // Spezialbehandlung von FromBlocks
7              if Blocktyp(b) = From then
8                  länge ← länge + 2;
9              else
10                 länge ← länge + 1;
11             end
12         else istErsterBlock ← false;
13     end
14 end
15 return länge / |Signale|;
```

Tabelle 5.19: Hilfsfunktionen des Grundalgorithmus für die durchschnittlichen Signallänge

Funktion	Beschreibung
HoleBlock	Gibt den Block zurück, der zu dem übergebenen Outport gehört.
Blocktyp	Ermittelt für einen Block den Blocktyp und gibt diesen als Zeichenkette zurück.

Umsetzung In Algorithmus 14 wird die Berechnung der durchschnittlichen Signallänge erläutert. Dieser benötigt als Eingabe alle Signale in einer Menge. Die Berechnung aller Signale wird in Abschnitt 4.2.3 beschrieben.

Der Algorithmus fängt damit an, über alle Signale zu iterieren (Zeile 2). Für jedes Signal wird über die Outports iteriert, die es durchläuft. In Zeile 5 wird der zum Outport gehörende Block geholt. Anschließend wird die zurückgelegte Strecke, von dem letzten Outport bis zu diesem Block, auf die Länge aller Signale hinzu addiert. Dabei wird mit der Variable *istErsterBlock* dafür gesorgt, dass beim ersten Block nichts hinzu addiert wird.

Hierbei ist zu beachten, dass der Algorithmus sich nicht um Zyklen kümmern muss, diese wurden durch die Berechnung der Signale schon berücksichtigt.

Der hinzu addierte Zuschlag ist abhängig von der zurückgelegten Strecke zwischen dem aktuellen und dem vorhergehenden Block. Hier kommt es zu einem Sonderfall:

Es gibt Blöcke, die das Signal weiterleiten, aber selbst keinen Outport haben. Diese sind gesondert zu berücksichtigen.

Goto-Block Ein Signal, dass einen Goto-Block erreicht, macht von diesem einen Sprung zu einem oder mehreren From-Blöcken. Diese korrespondierenden From-Blöcke sind in jedem Goto-Block gespeichert. Wenn ein Goto-Block erreicht wird, liegt kein Outport vor. Deshalb ist in einem Signal kein Outport von einem Goto-Block gespeichert, sondern es folgt sofort der Outport des entsprechenden From-Blocks. Da der Goto-Block nun übersprungen wurde, muss bei einem From-Block nicht nur eine Kante, sondern zwei Kanten zur Signallänge hinzu addiert werden.

Outport-Block Ein Outport-Block in einem Subsystem hat ebenfalls keinen eigenen Outport. Hier wird das Signal an den übergeordneten Subsystem-Block weitergereicht. Dies hat keinen Einfluss auf die Signallänge.

Am Ende teilt der Algorithmus die Summe der Längen aller Signale durch die Anzahl der Signale, um die durchschnittliche Länge zu ermitteln.

Slicing

Das *Slicing* ist eine bekannte Programmanalyse, die in vielen Bereichen verwendet wird, wie z. B. bei Softwaretests, im Reverse Engineering oder im Compilerbau [Tip94, S. 4]. Sie wird in dieser Arbeit eingesetzt, um darzustellen, wie weit ein Subsystem von vorhergehenden Modellteilen beeinflusst wird.

Der Autor ist der Meinung, dass je mehr Signale auf ein Subsystem Einfluss haben, es desto schwieriger ist, das Subsystem zu verstehen.

Definition: Slicing, Slice Eine Definition für das *Slicing* wird in [Wei81, S. 439] gegeben:

> „Angefangen mit einer Teilmenge des Programmverhaltens, reduziert das Slicing das Programm zu einer minimalen Form, die immer noch dasselbe Verhalten produziert."
>
> (Original: „Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior.")

Der Begriff *Slice* wird ebenfalls in [Wei81, S. 439] definiert:

> „Ein Slice ist selbst ein ausführbares Programm, dessen Verhalten identisch sein muss zu der spezifizierten Teilmenge des ursprünglichen Programmverhaltens."
>
> (Original: „A slice is itself an executable program, whose behavior must be identical to the specified subset of the original program's behavior.")

Die kleinste Einheit in einem Simulink-Modell, die das Verhalten bestimmt, ist ein atomarer Block. Es gibt jedoch viele Blöcke, die nur strukturelle Informationen enthalten und keine Programmlogik. Aus diesem Grund ist es sinnvoll, nur die Simulink-Blöcke zu betrachten, die wirklich neue Informationen generieren. Dies entspricht genau der Definition von Blöcken, die Signale generieren (siehe Abschnitt 4.2.3).

Damit lässt sich das Slicing auch auf Signalen definieren. Ein Slicing für ein Signal ist damit die Menge aller Signale, die das gewählte Signal beeinflussen

oder von diesem beeinflusst werden. Da mit dieser Metrik eine Aussage über die Komplexität von Subsystemen gemacht werden soll, werden im weiteren nur die *Vorfahren* des Signals betrachtet.

Ein *Vorfahre V* eines Signals *S* ist ein Signal, das im Block endet, in dem *S* beginnt, oder ein Vorfahre eines Vorfahrens von *S*.

Um eine möglichst gute Aussage über die Komplexität zu machen, wird diese Metrik für jedes Subsystem berechnet. Dabei wird die durchschnittliche Anzahl von Vorfahren pro Signal errechnet, das in diesem Subsystem seinen Ursprung hat.

Umsetzung In diesem Abschnitt wird erläutert, wie der Algorithmus zur Berechnung der Slicing-Metrik funktioniert. Das grundlegende Vorgehen ist in Algorithmus 15 dargestellt. Als Eingabe benötigt er einen Subsystem-Block, für

Algorithmus 15 : Der Grundalgorithmus des Slicing

input : Die Menge aller Signale S
input : Ein Subsystem-Block (Subsystem)
output : Die durchschnittliche Anzahl von Signalvorfahren

```
// Iteriere über alle Signale
1 foreach s ∈ S do
    // Bearbeite nur Signale die in Subsystem generiert
       werden
2   if ErsterBlock(s) ∈ Subsystem then
      // Speichere die Anzahl der gefundenen
         Signalvorfahren
3     A ← BerechneSignalvorfahren(s, ∅);
4     AnzahlVorfahren ← AnzahlVorfahren + |A|;
      // Speichere die Anzahl der überprüften Signale
5     AnzahlSignale ← AnzahlSignale + 1;
6   end
7 end
8 return AnzahlVorfahren/AnzahlSignale;
```

den die Slicing-Metrik berechnet werden soll. Außerdem benötigt er die Menge aller Signale des gesamten Modells. Ein Signal ist hierbei eine Liste von Blöcken, bei denen der erste Block den Beginn des Signals darstellt und der letzte Block dessen Ende. Dann iteriert der Algorithmus über alle Signale. Nur die Signale,

Tabelle 5.20: Hilfsfunktionen des Grundalgorithmus für das Slicing

Funktion	Beschreibung
ErsterBlock	Gibt den Block zurück in dem das übergebene Signal generiert wird.
BerechneSignalvorfahren	Siehe Algorithmus 16.

die im ausgewählten Subsystem sind, werden weiter betrachtet. In Zeile 3 wird für jedes dieser Signale die Menge aller Vorfahren berechnet. Danach werden die gefundenen Vorfahren und die überprüften Signale aufsummiert. Am Ende des Algorithmus wird die durchschnittliche Anzahl von Vorfahren pro Signal zurückgegeben, das im ausgewählten Subsystem beginnt.

Der Algorithmus 16 beschreibt das Vorgehen zur Berechnung der Menge aller Vorfahren für ein ausgewähltes Signal. Neben diesem Signal benötigt er noch die Menge aller bisher bearbeiteten Signale, dies ist nötig um einen zyklischen Signalverlauf zu erkennen. Der Algorithmus überprüft in Zeile 2 ob dieses Signal

Algorithmus 16 : Funktion BerechneSignalVorfahren

input : Ein Signal s
input : Die Menge aller bisher besuchten Signale V
output : Die Menge aller Signalvorfahren von s

```
  // Diese Menge speichert alle bisher gefundenen Vorfahren
1 A ← ∅;
  // Wird s zum ersten mal besucht?
2 if s ∉ V then
3     V ← V ∪ {s};
      // Hole alle Signale aus denen s generiert wird
4     A ← DirekteVorfahren(s);
      // Kopie von A für die Schleife
5     A' ← A;
      // Iteriere über alle direkten Vorfahren
6     foreach a ∈ A' do
          // Berechne weitere Vorfahren rekursiv
7         A ← A ∪ BerechneSignalVorfahren(a, V);
8     end
9 end
10 return A;
```

schon besucht wurde. In Zeile 4 werden alle Signale geholt die in dem Block

Tabelle 5.21: Hilfsfunktionen der Funktion BerechneSignalVorfahren

Funktion	Beschreibung
DirekteVorfahren	Gibt die Menge aller Signale zurück, die in dem Block enden in dem das übergebene Signal generiert wird.

enden, in dem das aktuelle Signal beginnt. Die in Zeile 5 erstellte Kopie aller Vorfahren wird nur verwendet, um über die direkten Vorfahren zu iterieren (Zeile 6), da die ursprüngliche Menge der Vorfahren in der Schleife ständig erweitert wird. In der Schleife werden die weiteren Vorfahren rekursiv berechnet. Am Ende wird dann die Menge aller gefundenen Vorfahren zurückgeliefert.

5.2.6 Fazit

Dieser Absatz fasst die in diesem Kapitel beschriebenen Metriken in Tabelle 5.22 zusammen. Dabei wird jede Metrik dem Kriterium zugeordnet, das von der Metrik gemessen wird. Außerdem wird der Aufwand für die Adaption, Konzeption und Implementierung jeder Metrik bewertet.

Tabelle 5.22: Zusammenfassung aller implementierten Metriken

Metrik	Dokumentation	Komplexität	Modularisierung	Wartbarkeit	Effizienz	Aufwand
Klonerkennung				×		Sehr hoch
Slicing		×				Hoch
Anzahl Zyklen		×				Mittel
Durchschnittliche Signallänge		×				Mittel
Unabhängige Pfade			×			Mittel
Verfallene Ergebnisse					×	Mittel
Blockinstabilität				×		Gering
Durchschn. Kommentarlänge pro Block	×					Gering

5.3 Nicht übertragbare Metriken

In diesem Abschnitt werden Metriken oder Ansätze vorgestellt die gar nicht, oder nur sehr schwer, für Simulink-Modelle verwendet hätten werden können.

5.3.1 Softwaremetriken für generierten Quellcode

Ein Ansatz für Metriken in der modellgetriebenen Entwicklung ist das Verwenden von bewährten Metriken für Quellcode, die auf den generierten Quellcode von Modellen angewendet werden könnten.

In dieser Arbeit wurde Abstand von dieser Idee genommen. In der modellgetriebenen Entwicklung ist der generierte Quellcode nur noch eine Zwischenrepräsentation, ähnlich wie Assembler bei normalem Quellcode. Da die Qualität dieser Zwischenrepräsentation sehr stark von dem verwendeten Werkzeug abhängt (Codegenerator), wird die Qualitätsbewertung von dem untersuchten Modell stark verfälscht.

5.3.2 Gesetz von Demeter

Das *Gesetz von Demeter* (Law of Demeter) ist ein Entwurfsmuster für Objektorientierte Programmiersprachen [LH89]. Durch die Verwendung dieses Entwurfsmusters soll eine gute Modularisierung erreicht werden um damit die Wartbarkeit zu erhöhen.

Um diese Ziele zu erreichen legt das Gesetz von Demeter einige Punkte fest, die eingehalten werden müssen um die Kopplung zwischen verschiedenen Klassen möglichst gering zu halten. Die Methode M einer Klasse C darf nur Funktionen und Daten der folgenden Klassen und Oberklassen verwenden:

- C.
- Klassen in C.
- Klassen die als Parameter an M übergeben werden.
- Klassen die in M erzeugt werden.
- Klassen von globalen Variablen die in M benützt werden.

Der Autor hatte die Idee die Regeln des Gesetz von Demeter auf Simulink-Modelle zu übertragen. Anschließend könnte die Einhaltung dieser Regeln überprüft und gemessen werden.

Dieser Ansatz war jedoch nicht erfolgreich. Die starke Bindung des Gesetz von Demeter mit objektorientierten Sprachen macht eine sinnvolle Adaption für Simulink-Modelle sehr schwer. Ein Simulink-Modell entspricht eher einem prozeduralen Programm und die Assoziation mit Klassen und Objekten ist hierbei kaum möglich. Aus diesem Grund wurde die Realisierung dieser Metrik verworfen.

5.3.3 Nebenläufigkeit

Die Metrik *Nebenläufigkeit* ist eine Metrik aus der Arbeit [Men08, S. 128] für Geschäftsprozessmodelle. Diese Metrik summiert für jeden Block eines Modells, der eine parallele Bearbeitung startet, die Anzahl der ausgehenden Kanten reduziert um eins. Der somit errechnete Wert ist also die Anzahl der nebenläufigen Verarbeitungsstränge in einem Modell.

Da in Simulink-Modellen keine echte Nebenläufigkeit existiert, ist die Adaption dieser Metrik nur schwer möglich.

6 Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Punkte dieser Master-Thesis zusammengefasst. Danach werden verschiedene Bereiche vorgestellt die, aufbauend auf dieser Arbeit, vertieft werden können.

6.1 Zusammenfassung

Der erste Punkt in dieser Arbeit ist die Untersuchung des aktuellen Stand der Technik im Bereich der Metriken (Kapitel 3). In diesem Kapitel werden Softwaremetriken untersucht, um darin einzelne Metriken zu identifizieren und wie gut diese für die Adaption für Simulink-Modelle geeignet sind.

Ein anderer Punkt in Kapitel 3 ist die Untersuchung von bestehenden Modellmetriken. Hierbei wird ein Blick auf Metriken für UML-, Geschäftsprozess- und Simulink-Modelle geworfen.

Danach wird der Aufbau von Simulink-Modellen in Kapitel 4 erläutert. Es wird darauf eingegangen was Simulink-Modelle sind, welche Elemente darin vorkommen und was sie für eine Semantik haben. Außerdem wird der Begriff eines *Signals* definiert, wie er in dieser Arbeit verwendet wird.

Schließlich werden in Kapitel 5, mit Hilfe des Goal/Question/Metric-Ansatzes, einzelne Kriterien identifiziert, die einzelne Qualitätsaspekte von Simulink-Modellen abdecken. Anschließend werden für diese Kriterien Metriken gesucht, die diese Kriterien messen.

Die implementierten Metriken werden in der folgenden Aufzählung kurz zusammengefasst:

Durchschnittliche Kommentarlänge pro Block In dieser Metrik wird die durchschnittliche Kommentarlänge pro Block für alle Kommentare in einem Simulink-Modell berechnet.

Unabhängige Pfade Diese Metrik berechnet die Anzahl der unabhängigen Pfade pro Subsystem. Ein unabhängiger Pfad ist ein zusammenhängender Teilgraph, der keine Verbindung zu anderen unabhängigen Pfaden hat.

Klonerkennung Diese Metrik sucht und zählt kopierte Teilmodelle in einem Simulink-Modelle.

Blockinstabilität Diese Metrik errechnet für jeden Block, wie wahrscheinlich er einer Änderung unterliegt.

Verfallene Ergebnisse Diese Metrik berechnet die Anzahl von Signalen die nicht verwendet werden. Die Berechnung dieser Signale ist also unnötig.

Anzahl Zyklen Diese Metrik errechnet die Anzahl von Zyklen in einem Simulink-Modell.

Durchschnittliche Signallänge Diese Metrik verwendet, die in dieser Arbeit definierten, Signale und errechnet deren durchschnittliche Länge.

Slicing Diese Metrik errechnet wie viele Signale Einfluss auf einen einzelnen Block haben.

In Tabelle 6.1 werden alle in dieser Arbeit implementierten Metriken, sowie die in dieser Arbeit angesprochenen existierenden Metriken für Simulink-Modelle, aufgelistet. Alle Metriken werden zusätzlich dem Kriterium zugeordnet das sie messen. Alle adaptierten Metriken, die in dieser Arbeit implementiert wurden, sind mit einem $\times$ gekennzeichnet. Bereits bestehende Simulink-Metriken, die aber nicht in dieser Arbeit realisiert wurden, sind mit einem $\circ$ gekennzeichnet. Die Metrik *Blockinstabilität* ist eine bestehende Simulink-Metrik, die auch in dieser Arbeit implementiert wurde. Diese Metrik ist mit einem $\otimes$ gekennzeichnet.

Am Ende von Kapitel 5 wird auf Metriken eingegangen, die nicht oder nur sehr schwer auf Simulink-Modelle adaptiert werden können.

Tabelle 6.1: Auflistung der Simulink-Metriken

Metrik	Dokumentation	Komplexität	Modularisierung	Wartbarkeit	Effizienz
Anzahl Zyklen		×			
Blockinstabilität				⊗	
Durchschnittliche Kommentarlänge pro Block	×				
Durchschnittliche Signallänge		×			
Halstead-Volumen		○			
Klonerkennung				×	
Slicing		×			
Systeminstabilität				○	
Unabhängige Pfade			×		
Verfallene Ergebnisse					×
Zyklomatische Komplexität		○			

6.2 Ausblick

In den folgenden Unterpunkten werden Ansatzmöglichkeiten für zukünftige Arbeiten vorgestellt.

Weitere Metriken Die Kriterien Dokumentation, Modularisierung und Effizienz werden bisher nur durch jeweils eine Metrik bewertet. In weiteren Arbeiten könnten neue Metriken gesucht werden um diese Aspekte besser messen und bewerten zu können.

Weitere Kriterien In dieser Arbeit wurden einige Kriterien zur Messung der Qualität nicht beachtet (Funktionalität, Zuverlässigkeit und Benutzbarkeit), da diese für die untersuchten Modelle nicht relevant waren. In folgenden Arbeiten können Metriken für diese Kriterien entwickelt und umgesetzt werden.

Untersuchen der Aussagekraft Die in dieser Arbeit implementierten Metriken wurden nicht auf ihre Aussagekraft über die Qualität von Simulink-Modellen

untersucht. Die Einteilung der Metriken in die entsprechende Kriterien sind nur auf Einschätzungen des Autors gemacht worden. Eine genaue Untersuchung wie gut eine Metrik ein oder mehrere Kriterien beschreibt, wäre sinnvoll. Ein Ansatz für die Untersuchung der Aussagekraft findet sich in der Arbeit [Sch10].

Literatur

[AHW03] Wil van der Aalst, Arthur ter Hofstede und Mathias Weske. „Business Process Management: A Survey“. In: *Business Process Management*. Bd. 2678. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2003, S. 1019–1030.

[Alb79] Allan Albrecht. „Measuring application development productivity“. In: *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*. 1979, S. 83–92.

[Bas92] Victor Basili. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. Techn. Ber. University of Maryland, 1992.

[Car+06] Jorge Cardoso u. a. „A Discourse on Complexity of Process Models“. In: *Business Process Management Workshops*. 2006, S. 117–128.

[CK94] Shyam Chidamber und Chris Kemerer. „A Metrics Suite for Object Oriented Design“. In: *IEEE Transaction on Software Engineering*. 1994, S. 476–493.

[Con+05] Mirko Conrad u. a. „Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler“. In: *Informatik – Forschung und Entwicklung* 20.1–2 (Okt. 2005), S. 3–10.

[Coo71] Stephen Cook. „The Complexity of Theorem-Proving Procedures“. In: *Proceedings of the third annual ACM symposium on Theory of computing*. 1971, S. 151–158.

[dSP10] dSPACE. *dSPACE - TargetLink*. 8. Dez. 2010. URL: `http://www.dspace.de/de/gmb/home/products/sw/pcgs/targetli.cfm`.

[ES03] Heiko van Elsuwe und Doris Schmedding. „Metriken für UML-Modelle“. In: *Informatik – Forschung und Entwicklung* 18.1 (Aug. 2003), S. 22–31.

[FN99] Norman Fenton und Martin Neil. „Software Metrics: successes, failures and new directions“. In: *Journal of Systems and Software* 47.2–3 (Juli 1999), S. 149–157.

[Hal77] Maurice Halstead. *Elements of software science*. Elsevier New York, 1977.

[KKT10] Soeren Kemmann, Thomas Kuhn und Mario Trapp. „Extensible and Automated Model-Evaluations with INProVE“. Fraunhofer-Institut für Experimentelles Software Engineering. 2010.

[KB02] Hyoseob Kim und Cornelia Boldyreff. „Developing Software Metrics Applicable to UML Models“. In: *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*. 2002.

[LK01] Antti Latva-Koivisto. *Finding a complexity measure for business process models*. 2001.

[LH89] Karl Lieberherr und Ian Holland. „Assuring Good Style for Object-Oriented Programs“. In: *IEEE Software* 6.5 (Sep. 1989), S. 38–48.

[Mar94] Robert Martin. *OO Design Quality Metrics (An Analysis of Dependencies)*. 1994.

[Mat10a] The MathWorks. *MathWorks Deutschland – MATLAB – The Language of Technical Computing*. 8. Dez. 2010. URL: `http://www.mathworks.de/products/matlab/`.

[Mat10b] The MathWorks. *MathWorks Deutschland – Simulink – Simulation und Model-Based Design*. 8. Dez. 2010. URL: `http://www.mathworks.de/products/simulink/`.

[McC76] Thomas McCabe. „A Complexity Measure“. In: *IEEE Transactions on Software Engineering* 2.4 (Dez. 1976), S. 308–320.

[MP07] Jacqueline McQuillan und James Power. „On the Application of Software Metrics to UML Models“. In: *Models in Software Engineering*. Bd. 4364. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2007, S. 217–226.

[Men08] Jan Mendling. „Metrics for Process Models“. In: Bd. 6. Lecture Notes in Business Information Processing. Springer Berlin/Heidelberg, 2008. Kap. Metrics for Business Process Models, S. 103–133.

[MA05] Guido Menkhaus und Brigitte Andrich. „Metric Suite for directing the Failure Mode Analysis of embedded Software Systems“. In: *Proceedings of the 7th International Conference on Enterprise Information Systems*. 2005, S. 266–273.

[PKW06] Kai Pan, Sunghun Kim und James Whitehead. „Bug Classification Using Program Slicing Metrics“. In: *Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. 2006, S. 31–42.

[RC07] Chanchal Kumar Roy und James Cordy. *A Survey on Software Clone Detection Research*. Techn. Ber. Queen's University, 2007.

[Sch10] Jan Scheible. „Ein Framework zur automatisierten Ermittlung der Modellqualität bei eingebetteten Systemen“. In: *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*. 2010, S. 1–6.

[SK10] Jan Scheible und Ingo Kreuz. „Ein Qualitätsmodell zur automatisierten Ermittlung der Modellqualität bei eingebetteten Systemen“. In: *8. GI Workshop Automotive Software Engineering*. 2010, S. 509–514.

[SPR10] Ingo Stürmer, Hartmut Pohlheim und Thorsten Rogier. „Berechnung und Visualisierung der Modellkomplexität bei der modellbasierten Entwicklung sicherheits-relevanter Software“. In: *Automotive – Safety & Security*. 2010, S. 69–82.

[TC02] Mei-Huei Tang und Mei-Hwa Chen. „Measuring OO Design Metrics from UML“. In: *«UML» 2002 – The Unified Modeling Language*. Bd. 2460. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2002, S. 361–370.

[Tip94] Frank Tip. *A Survey of Program Slicing Techniques*. Techn. Ber. Centre for Mathematics und Computer Science, 1994, S. 121–189.

[Van+08] Irene Vanderfeesten u. a. „On a Quest for Good Process Models: The Cross-Connectivity Metric“. In: *Advanced Information Systems Engineering*. 2008, S. 480–494.

[Wei81] Mark Weiser. „Program slicing“. In: *Proceedings of the 5th international conference on Software engineering*. 1981, S. 439–449.

[Wik10a] Wikipedia – Die freie Enzyklopädie. *DIN 66272*. 5. Nov. 2010. URL: http://de.wikipedia.org/wiki/DIN_66272.

[Wik10b] Wikipedia – Die freie Enzyklopädie. *ISO/IEC 9126*. 5. Nov. 2010. URL: http://de.wikipedia.org/wiki/ISO/IEC_9126.

Name: Wolfgang Holoch Matrikelnummer: 580779

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die wörtlich oder inhaltlich übernommenen Stellen sind als solche kenntlich gemacht und die Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung wurde beachtet.

Ulm, den .....

Wolfgang Holoch