Entwurfsmethodik heterogener Systeme

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Ing. Steffen Klupsch

aus Stuttgart

Referenten der Arbeit:	Prof. Dr.-Ing. S. A. Huss Prof. Dr.-Ing. W. Glauert
Tag der Einreichung:	01. Dez. 2003
Tag der mündlichen Prüfung:	29. März 2004

Darmstädter Dissertation
D17

Inhaltsverzeichnis

Verzeichnis der Refactorings

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet für integrierte Schaltungen und Systeme der Technischen Universität Darmstadt. Sie ist das Ergebnis einer mehrjährigen kontinuierlichen Auseinandersetzung mit dem Forschungsgebiet, das durch die fachliche Diskussion mit Prof. Dr. Sorin A. Huss, meinen Kollegen und vielen Studenten sehr an Substanz gewonnen hat. Hiermit möchte ich mich bei allen bedanken, die zum Entstehen dieser Arbeit beigetragen haben. Mein besonderer Dank gilt meiner Alexandra und meinen Eltern, die sich über die Vollendung dieser Arbeit mindestens genauso gefreut haben wie ich. Ihr Verständnis und ihre Geduld gaben mir den nötigen Rückhalt.

Den Grundstein für diese Arbeit legten Prof. Dr. Gerhard Grau und Prof. Dr. Karl Reiß, deren Freude an der Forschung während meines Studiums in Karlsruhe auf mich abfärbte, sowie mein Vater, der mich stets ermutigte meinen Weg zu gehen.

Prof. Dr. Sorin A. Huss danke ich für die hervorragende Betreuung, die stets konstruktive Kritik, die fortlaufende Motivation zur Teilnahme an Workshops und zur Erstellung wissenschaftlicher Publikationen, sowie für seine Geduld in der Endphase der Arbeit. Er knüpfte auch Kontakte zur Industrie, die mich sehr motiviert haben. Jürgen Schulz stellte in meiner Anfangsphase ein Industrieprojekt aus dem Automotive Bereich vor, anhand dessen mir die Bedeutung von Mixed-Signal Entwurfsabläufen offensichtlich wurde. Dr. Heiner Flocke, Manfred Herz und Hartmut Scherner gilt mein besonderer Dank, da sie die Infrastruktur der iC-Haus GmbH und ihr Wissen zur Verfügung stellten, um den Delta-Sigma A/D Modulator (Kapitel 5.5.2) in einem CMOS ASIC zu realisieren.

Ebenso möchte ich mich bei meinen Kollegen Tom Aßmuth, Dr. Werner Bachmann, Dr. Wolfgang Bossung, Markus Ernst, Eva Glaser, Oliver Hauck, Stephan Hermanns, Abbas Laschgari, Stephan Klaus, Andreas Kühn, Dr. Ralf Rosenberger, Abdul Shoufan, Robert Strzodka und Jürgen Weber für die fruchtbaren Diskussionen, den freundschaftlichen Rat und für die vielen schönen Stunden bedanken. Im Rahmen ihrer Semester-, Studien-, und Diplomarbeiten haben die Studenten Nadeem Bhatti, Alexander Friebe, Karsten Grüner, Markus Gualeni, Tobias Kuckuck, Mateusz Majer, Armin Schmidt, Björn Seiffert, und Wolfram Stumpf mit großem Engagement wichtige Vorarbeiten zu dieser Arbeit geleistet.

Prof. Dr. Wolfram Glauert danke ich herzlich für sein Interesse an dieser Arbeit und für die Übernahme des Koreferats.

Susanne Schaude und Dr. Horst Schaude möchte ich für die tatkräftige Unterstützung bei der Suche nach Schreibfehlern und stilistischen Fehlgriffen danken; ohne Ihre Hilfe hätte ich es nicht gewagt nach den Regeln der neuen deutschen Rechtschreibung zu schreiben.

Schließlich danke ich Dr. Gottlieb Strassacker und Dr. Karsten Mühlmann, durch deren Unterstützung ich seinerzeit das Vordiplom bestanden habe.

Ober-Olm, im März 2004 Steffen Klupsch

Kapitel 1

Einleitung

Eine Vielzahl der Geräte, die wir im täglichen Leben benutzen, sind heterogene Systeme, wie beispielsweise ein Auto. Dieses enthält mechanische Komponenten, die eine kontinuierliche Steuerung des Systems ermöglichen. Es enthält ebenso ereignisdiskrete Baugruppen, z. B. Airbags, die nur bei einem speziellen Ereignis aktiviert werden. Außerdem enthält es eine Vielzahl von Steuergeräten, in denen ein Softwareprogramm zyklisch abgearbeitet wird.

Der Begriff des heterogenen Systems kann auf verschiedene Arten definiert werden. Zum einen ist ein heterogenes System ein System das Komponenten aus unterschiedlichen Anwendungsdomänen beinhaltet. Es kann beispielsweise ein System mit elektrischer Signalverarbeitung und anderen Komponenten[1], deren nichtelektrische Eigenschaften für die Funktionalität des Systems von Bedeutung sind, sein. Ein Hubschrauber mit Rotoren, Antrieb und Steuerungselektronik kann als heterogenes Gesamtsystem betrachtet werden, ebenso ein Hörgerät. Durch Fortschritte in der Mikrosystemtechnik können druckempfindliche Sensoren mittlerweile direkt in einen Mixed-Signal Chip integriert werden. Deshalb können Sensoren und die signalverarbeitenden elektronischen Schaltungen auch bezüglich lokaler Störeffekte korreliertes Fehlverhalten aufweisen. Aufgrund dessen erscheint eine gemeinsame Validierung sinnvoll.

Stellt man die Modellbildung und die Simulation des Systems an Stelle des Endproduktes in den Vordergrund, so ergibt sich aus der Analyse der Modelle eine anwendungsunabhängige Definition des heterogenen Systems, die im Rahmen dieser Arbeit verwendet wird:

Definition 1.1 (Heterogenes Simulationssystem): Ein heterogenes Simulationssystem ist ein System, in dem Komponentenmodelle unterschiedlicher Abstraktionsgrade und mit unterschiedlichen Ausführungskonzepten gemeinsam simuliert werden.

In vielen Fällen ist Heterogenität bezüglich beider Definitionen gegeben, so dass eine Zusammenfassung unter dem Begriff heterogenes System naheliegt.

Diese Arbeit hat die methodische Aufbereitung eines Entwurfsablaufs zum modellbasierten Entwurf heterogener Systeme und zur Validierung dieser Systeme zum The-

[1] Zu diesen Komponenten gehören z. B. Sensoren und Aktoren.

ma. Der Entwurf dieser Systeme ist eine besonders anspruchsvolle Aufgabe, da Expertenwissen aus mehreren Anwendungsgebieten kombiniert werden muss. Entwickler mit unterschiedlichen Ausbildungen benötigen eine gemeinsame Entwurfsplattform, um Wechselwirkungen zwischen den Systembestandteilen zu untersuchen. Solche Entwurfsplattformen existieren zur Zeit nur für spezielle Problemstellungen. Der Bedarf an einer allgemein verwendbaren Entwicklungsumgebung zeigt sich jedoch in den aktuellen Modellierungskonzepten und den dazugehörigen Notationssprachen (z. B. VHDL-AMS, SystemC, Modelica).

Modellierungskonzepte

Ein zentraler Bestandteil dieser Arbeit sind die zur Beschreibung heterogener Systeme benötigten Modellierungskonzepte. Für die einzelnen Anwendungsdomänen existieren bereits Modellierungsverfahren, die auf die Besonderheiten des jeweiligen Arbeitsgebiets zugeschnitten sind.

So basiert der Entwurf integrierter digitaler Schaltungen hauptsächlich auf textuellen ereignisdiskreten Verhaltensmodellen, die mit Hilfe von Syntheseprogrammen auf prozessspezifische Bauteilbibliotheken abgebildet werden. Bei der Entwicklung analoger elektrischer Schaltungen werden i. d. R. prozessspezifische Modelle direkt erstellt. Die Entwicklung von Sensoren und mechanischen Komponenten erfordert zusätzlich die Gestaltung der Grundelemente, so dass Modellbeschreibungen zur Evaluierung von Materialeigenschaften und zur Dimensionierung der räumlichen Ausdehnung der Komponenten benötigt werden.

Die domänentypischen Modellierungskonzepte können jedoch nicht ohne umfangreiche Erweiterungen für die Entwicklung heterogener Systeme genutzt werden. Es gibt allerdings zahlreiche äquivalente Bestandteile in den Modellierungskonzepten, so dass eine domänenunabhängige Modellbildung erarbeitet werden kann.

Qualitative Klassifizierung von Simulationsmodellen

Es gibt mehrere ursprünglich domänenspezifische Modellierungssprachen, die jetzt so erweitert wurden, dass auch die Beschreibung domänenfremder Modelle möglich ist (z. B. VHDL-AMS). In manchen Fällen kann es jedoch auch notwendig sein, ein Modell in einer Simulationssprache wie SPICE [17] nachzubilden, um eine gemeinsame Simulation zu ermöglichen. SPICE ist eine weitverbreitete auf die Simulation analoger elektrischer Schaltungen zugeschnittene Simulationssprache. Sie wird jedoch, wie in [58, 78, 72] beschrieben, auch zur Modellierung von Anwendungsszenarien genutzt, in denen optisch-, thermisch- und drucksensitive Komponenten benötigt werden. Dabei werden diese Komponenten mit Hilfe von elektrischen Ersatzmodellen nachgebildet.

Bei der Modellierung heterogener Systeme ist die Repräsentation von Teilsystemen in domänenfremden Simulationssprachen oftmals einfacher zu realisieren, als ein domänenübergreifendes Simulationssystem. Die Transformation von Modellen zwischen verschiedenen Simulationssprachen wird erleichtert, wenn es ein simulatorunabhängiges Meta-Modell gibt, welches auf die unterschiedlichen simulatorspezifischen Beschreibungssprachen abgebildet werden kann. B. P. Zeigler et. al. haben mit dem systemtheoretischen Ansatz der *Combined Discrete-Continuous System Simulation* ein solches abstraktes Beschreibungssystem erarbeitet [95]. Im Rahmen des Ptolemy-Projekts [59] hat sich die Forschergruppe um E. A. Lee ebenfalls mit dieser Problematik beschäftigt. Der Artikel [60] von E. A. Lee und A. Sangiovanni-Vincentelli beschreibt ein Meta-Modell, dass ebenfalls domänenunabhängig ist und die Modellierung von zustandsbasierten Systemen ermöglicht.

Für die Modellierung heterogener Systeme wird jedoch ein Meta-Modell benötigt, dass auch die Beschreibung eines Systems aus partiellen Differentialgleichungen erlaubt, d. h. ein Meta-Modell, in dem Zustandsgrößen über Raum und Zeit definiert werden können. Ein solches Meta-Modell wird in dieser Arbeit vorgestellt. Es beinhaltet die Modellierungsmöglichkeiten von Zeigler und Lee und ermöglicht zusätzlich die Beschreibung von raum- und zeitkontinuierlichen Feldern und Potentialgrößen.

Entwurfsmethodik

Beim Entwurf digitaler Schaltungen wird die Synthese von Verhaltensmodellen in eine Strukturbeschreibung aus vordefinierten Bibliothekselementen erfolgreich angewendet. Für viele der nichtdigitalen Komponenten eines heterogenen Systems ist eine solche automatisierte Modelltransformation jedoch nicht verfügbar. Deshalb nimmt ein methodischer Entwurf der Verhaltensmodelle und eine effiziente und sichere Arbeitsweise bei der Konkretisierung dieser Modelle einen hohen Stellenwert ein. Mit Hilfe von ausführlichen Testprogrammen kann die Wahrscheinlichkeit für Entwurfsfehler reduziert werden. Je früher diese Tests durchgeführt werden, um so besser lassen sich Entwurfsentscheidungen überprüfen. Eine Gesamtsystemsimulation mit vereinfacht beschriebenen Komponentenmodellen kann deshalb schon in der Konzeptionsphase sinnvoll eingesetzt werden.

Um Modelle aus der Gesamtsystemsimulation bei der Validierung von einzelnen konkretisierten Modellen verwenden zu können, wird eine Simulationsumgebung für heterogene Systeme benötigt. Damit ist eine Simulationsumgebung gemeint, in der Verhaltensmodelle mit unterschiedlichen Ausführungskonzepten gemeinsam ausgeführt werden können. Dabei werden für eine effektive Arbeitsweise sowohl automatisierbare, selbsttestende Simulationen, als auch interaktive Simulationen benötigt. In einer interaktiven Simulation kann der Benutzer während der Simulation Stimulidaten in das Systemmodell einspeisen und muss die Simulationsergebnisse mit einer einfach zu interpretierenden grafischen Ausgabe veranschaulicht bekommen.

Der zusätzliche Arbeitsaufwand für diese Validierungshilfsmittel soll jedoch die Entwicklungszeit des Produkts nicht verlängern. Um dieses zu erreichen muss eine flexible, effiziente und fehlerarme Methodik zur Wartung und Verfeinerung der Verhaltensmodelle zur Verfügung gestellt werden. Für den Entwurf objektorientierter Softwareprogramme existiert ein solcher Ansatz, der unter dem Namen *Refactoring* bekannt geworden ist. Die Erstellung von Verhaltensmodellen für heterogene Systeme weist viele Parallelen zur Softwareentwicklung auf, so dass eine Übertragung der Refactoring-Methodik sinnvoll erscheint. Allerdings kann ein solcher Ansatz nur dann erfolgreich sein, wenn er die Besonderheiten der Verhaltensmodell-Entwicklung berücksichtigt: Verhaltensmodelle werden selten in objektorientierten Beschreibungssprachen geschrieben und die Verhaltensmodelle müssen mehrfach konkretisiert werden, bevor eine implementierbare Beschreibung entsteht. Zusätzlich muss es möglich sein, die Verhaltensmodelle in unterschiedliche Abstraktionsebenen zu transformieren.

Für die grafische Visualisierung der Simulationsergebnisse und zur Einspeisung von Stimuli sind *Rapid Application Development Tools* geeignet, die mit unterschiedlichen Simulationskernen gekoppelt werden können. Zusätzlich zu den rein softwaregestützten Simulationskernen (SPICE, VHDL-AMS), ist eine Hardware-Emulation für digitale Schaltungen mit Hilfe von FPGAs realisierbar. Bei geeigneter Partitionierung des Systemmodells kann die Nachbildung einer digitalen Schaltung im FPGA die Simulation des Gesamtsystems um ein Vielfaches beschleunigen.

Ziele der Arbeit

In dieser Arbeit werden Hilfsmittel vorgestellt, die die Analyse und Modellbildung von Verhaltensmodellen erleichtern. Das für die Modellanalyse vorgeschlagene Meta-Modell ist speziell auf die Bedürfnisse der Zeitbereichssimulation heterogener Systeme zugeschnitten. Die erweiterten Möglichkeiten bei der Transientensimulation werden durch Demonstratoren veranschaulicht. Anhand dieser Demonstratoren werden ebenfalls die Anforderungen an ein Simulationssystem für heterogene Systeme gezeigt. Mit der in dieser Arbeit vorgestellten Entwurfsmethodik wird eine effektive, strukturierte und fehlerarme Modellbildung und Modellverfeinerung ermöglicht. Diese Arbeitsweise ermöglicht insbesondere in umfangreichen, komplexen Systemen eine zielgerichtete, simulationsbasierte Implementierung und Validierung der Systemkomponenten und des Gesamtsystems.

Kapitelübersicht

In Kapitel 2 wird ein mathematisches Meta-Modell eingeführt, in das Verhaltensmodelle aus vielen domänenspezifischen Simulationsprachen abgebildet werden können. Diese Modelle können innerhalb des Meta-Modells analysiert werden. Dazu werden

mehrere Modellklassen definiert, die sich im Zeitmodell und der Komplexität von Zustandsgrößen unterscheiden.

Im darauf folgenden Kapitel werden etablierte domänenspezifische Modellierungskonzepte und Ansätze zur Modellierung heterogener Systeme vorgestellt und diskutiert.

Kapitel 4 erläutert einen neuen qualitätsbasierten Arbeitsfluss. Dabei wird mit Hilfe von vereinfachten Systemmodellen, Gesamtsystemsimulationen, speziell für die Modellbildung heterogener Systeme angepassten Refactoring-Vorschriften, und einem Abstraktionsebenen-übergreifenden Validierungskonzept, ein für den Modellentwickler effektiver und fehlerarmer Entwurfsablauf sichergestellt. Die dafür benötigte Simulationsumgebung ist ein Komponentenmodell, dessen Konzeption eine weitgehend freie Programmierbarkeit ermöglicht. Es wird die Anbindung einer FPGA-Karte in die Simulationsumgebung vorgestellt. In dieser FPGA-Karte werden Modelle digitaler Schaltungen ausgeführt; die Simulation anderer Modelle wird mit Hilfe von Softwaresimulatoren durchgeführt. Neben der Unterstützung eines kommerziellen Simulationssystems, in dem mehrere Modellierungssprachen unterstützt werden, ist dabei die Schnittstelle zu einem *Rapid Application Development Tool* wichtig. Diese Kombination ermöglicht sowohl eine effiziente Simulation, als auch eine effektive und einfach zu interpretierende Aufbereitung der Simulationsergebnisse.

In Kapitel 5 wird die Anwendung der Entwurfsmethodik und die Tragfähigkeit des vorgeschlagenen Meta-Modells mit Hilfe von Demonstratoren nachgewiesen.
Anhand der biologischen Vorgänge beim Presslufttauchen wird die Abbildung eines Systems aus Differentialgleichungen in ein abstraktes Verhaltensmodell, das für die Implementierung in einem Mikroprozessor geeignet ist, vorgestellt. Unter Verwendung eines Delta-Sigma A/D Wandlers wird die iterative Konkretisierung von Verhaltensmodellen bis zu einem implementierbaren Komponentenmodell detailiert diskutiert.

Im abschließenden Kapitel 6 werden die Ergebnisse dieser Forschungsarbeit zusammengefasst. Außerdem wird ein Ausblick auf einige interessante Anknüpfungspunkte für zukünftige Forschungsarbeiten gegeben.

Kapitel 2

Qualitative Eigenschaften der Simulationsmodelle

2.1 Stand der Technik

Arbeiten zur Entwurfsmethodik für heterogene Systeme sind in den letzten Jahren Bestandteil vieler Forschungsprojekte gewesen. Die Anforderungen an Simulationsverfahren und Validierungsumgebungen sind vielfältig. Aus der Weiterentwicklung von eingebetteten Systemen zu intelligenten Sensoren, Mikrosystemen oder autonomen fehlertoleranten Regelsystemen sind sehr unterschiedliche anwendungsspezifische Entwurfsverfahren entstanden. Der derzeitige Automatisierungsgrad in der Modellbildung ist gering. Neben der Extraktion des Systemverhaltens ist die Abbildung des Verhaltens in ein simulierbares Modell ebenfalls eine zeitaufwendige und schwierige Aufgabe.

Andererseits ist die Bedeutung von Systembeschreibungen in Form von Modellen unstrittig. In Verbundprojekten des BMBF[1] [91, 8, 10, 9] wurde in den letzten Jahren intensiv an Verhaltensmodellen für die Systemsimulation heterogener Systeme gearbeitet. In den neuesten Forschungsprojekten aus dem Bereich der Mikrosystemtechnik wird deutlich, dass heutzutage heterogene Systeme entworfen und optimiert werden, die ausschließlich unter gleichzeitiger Betrachtung von elektrischen und mechanischen Randbedingungen beschrieben werden können. In vielen Anwendungen [61, 74, 58] sind zusätzliche Randbedingungen zu berücksichtigen (Temperatur, Druck, Magnetfelder), so dass eine systematische Modellierungsmethodik mit domänenunabhängigen Konzepten unabdingbar ist. Um eine derartige Methodik zu ermöglichen, wird in diesem Kapitel ein *universelles Meta-Modell*, sowie eine abstrakte *domänenunabhängige Klassifizierung* von Modellierungsansätzen vorgestellt, die zu einer Einteilung in Modellklassen bezüglich des Systemwissens führt.

Im Folgenden werden mathematische Eigenschaften definiert, denen Modelle genügen müssen, um einer Modellklasse anzugehören. Die Realisierung von Modellen in einer (eventuell domänenspezifischen) Modellierungssprache bedingt zusätzliche Einschränkungen in der Modellbildung. Zugunsten einer domänenübergreifenden Verwendbarkeit der Modellklassen wird im Folgenden eine Definition von Modellen erarbeitet. Anstatt der mathematischen Herleitung können die Modellklassen auch durch Abstraktion der

[1] Bundesministerium für Bildung und Forschung

physikalischen Eigenschaften, bzw. des Zeitmodells hergeleitet werden. Dieser Ansatz wird an geeigneter Stelle exemplarisch zur Veranschaulichung genutzt.

Zur Notation der Modelleigenschaften werden einige Begriffe und Definitionen benötigt. In Anlehnung an den systemtheoretischen Modellierungsansatz von Zeigler und Prähofer [95] und an das Meta-Modell für funktionale und algorithmische Modelle von Lee und Sangiovanni-Vincentelli [60] werden für die Modellbeschreibung Zustandsgrößen als Basis eingeführt. Dabei werden die Anregungen des Modells, der innere Zustand und die daraus resultierenden Observablen[2] mit den Zustandsgrößen $\mathbf{z}_i$ beschrieben.

Bei Lee und Sangiovanni-Vincentelli wird ein Meta-Modell mit Hilfe von Mengen, partiell geordneten Mengen, Relationen und Funktionen definiert. Dort wurde gezeigt, dass diese Beschreibungsform eine Anwendung auf eine Vielzahl von Modellierungskonzepten ermöglicht. Die Zustandsgrößen wurden als Mengen definiert, die eine Anzahl von Ereignissen enthalten, die über ihren Wert und einen *Tag* [3] definiert waren. Dieser Ansatz wird in dieser Arbeit übernommen.

Diese Definition ist jedoch nicht ausreichend, um alle in heterogenen Systemen auftretenden Modellierungsaufgaben zu beschreiben. So ist beispielsweise für die Optimierung eines Drucksensors ein raumkontinuierliches Modell hilfreich, mit dem mechanische Verformungen bei Druckbelastung untersucht werden können.

2.2 Ein mathematisches Meta-Modell

Im Folgenden wird gezeigt, dass durch die Erweiterung der Zustandsgrößen zu einer Datenstruktur ein universelles Meta-Modell definiert werden kann, wenn die Elemente dieser Datenstruktur aus Wert, *Tag* und einem Satz Koordinaten im n-dimensionalen Raum bestehen. In diesem können sowohl die von Lee vorgestellten Modellierungskonzepte, als auch zeit- und raumkontinuierliche Modellierungsansätze unterschieden werden. Neben der Einbeziehung räumlich verteilter Problemstellungen ermöglicht die Erweiterung der Zustandsgrößen die Beschreibung von parametrisierten Modellen, wie sie z. B. zur Ausbeute-Optimierung verwendet werden.

Ein anschauliches Beispiel ist ein SPICE-Modell eines integrierter Operationsverstärker, dessen Validierung erst nach Simulationen mit mehreren 'Worst Case Prozessparametern' und mit unterschiedlichen Betriebsbedingungen (z. B. Temperatur) abgeschlossen ist.

Aufbauend auf den Ergebnissen von Lee und Sangiovanni-Vincentelli, die gezeigt haben, dass in ihr Meta-Modell Berechnungsmodelle wie Kahn's Prozess Netzwerke, CSP, Petri-Netze und DEVS abgebildet werden konnten, wird in Kapitel 2.3 eine neue Klassifizierung von Modellierungsmöglichkeiten hergeleitet, die den Bogen von zeit- und

[2] Observable: beobachtbare Zustandsgröße

[3] *Tag* (engl.): *Tags* wurden benutzt, um Zeit, zeitliche Abfolge, Synchronisationszeitpunkte oder Vergleichbares zu modellieren

raumkontinuierlichen Modellen, z. B. zur Feldberechnung, bis zu den diskreten, zeitfreien Modellen, in denen lediglich eine Berechnungsreihenfolge festgelegt ist, z. B. Software in einem eingebetteten System, aufspannt.

Die Zustandsgrößen werden als komplexe Datenstrukturen eingeführt; sie enthalten Elemente z_i, die jeweils aus drei Bestandteilen bestehen: Einem *Tag*, einem Wert und einem Raumpunkt.[4] Dementsprechend ist $\mathbf{z}$ eine Menge solcher Elemente:

$$\mathbf{z} = \{z_1, z_2, z_3, \ldots\} \tag{2.1}$$

Die Werte einer Zustandsgröße $\mathbf{z}$ sind aus dem Definitionsbereich $\mathscr{U}$, der Wert eines Elements z wird im Folgenden mit $u(z)$ bezeichnet.

$$u(z) \in \mathscr{U} \tag{2.2}$$

Die Wertebereiche ($\mathscr{U}_{z_1}$, $\mathscr{U}_{z_2}$) zweier Zustandsgrößen ($\mathbf{z}_1$, $\mathbf{z}_2$) können verschieden sein.

Mit $t(z)$ wird eine temporale Ordnung zwischen den Elementen einer Zustandsgröße hergestellt. $t(z)$ kann anschaulich als *Zeitstempel eines Elements der Zustandsgröße* $\mathbf{z}$ interpretiert werden. Um Fehlinterpretationen bei Modellen, die keine globale zeitliche Ordnung haben, zu vermeiden, wird für $t(z)$ im Folgenden oft der Begriff *Tag* verwendet. (Für *Tags unterschiedlicher* Zustandsgrößen ist in nebenläufigen Modellen (siehe Kapitel 2.3.1) unter Umständen keine zeitliche Ordnung definiert.)

$$t(z) \in \mathscr{T} \tag{2.3}$$

Definition 2.1: Die Menge $\mathscr{T}$ muss *vollständig geordnet* sein, d. h. für alle t_i, t_j, t_k aus der Menge $\mathscr{T}$ gilt

$$(t_i < t_j) \text{ xor } (t_j < t_i) \qquad \forall t_i, t_j \in \mathscr{T} \wedge t_i \neq t_j \tag{2.4}$$

$$(t_i < t_j) \wedge (t_j < t_k) \Rightarrow t_i < t_k \qquad \forall t_i, t_j, t_k \in \mathscr{T} \tag{2.5}$$

Zur Unterscheidung von z_i mit gleichem *Tag* und gleichem Wert wird ein Tupel $\mathbf{x}(z)$ verwendet. Dieses ermöglicht eine Unterscheidung der z_i, die vielfältig genutzt werden kann.

Eine offensichtliche Anwendung für $\mathbf{x}(z)$ sind ortsabhängige Zustandsgrößen, wie sie z. B. für die Beschreibung von elektrischen Feldern benötigt werden. Bei einer ortsabhängigen Zustandsgröße wird mit dem Tupel $\mathbf{x}(z)$ ein Raumpunkt beschrieben. Wenn der dazugehörige Raum n-dimensional ist, dann gilt $\mathbf{x}(z) \in \mathscr{X}^n$. Wie für die Werte der Zustandsgrößen können zur Beschreibung eines Modells unterschiedliche Räume $\mathscr{X}^n$ verwendet werden. Dazu werden für jeden benötigten Raum eigene Zustandsgrößen definiert.

[4] Ein Raumpunkt ist durch einen Satz Koordinaten im n-dimensionalen Raum beschrieben.

Ebenso kann $\mathbf{x}(z)$ zur Festlegung einer quantifizierbaren Eigenschaft einer Zustandsgröße verwendet werden: Bei der Modellierung von parametrisierbaren Systemen können Zustandsgrößen über einem Parameterraum (z. B. Betriebsbedingungen) definiert werden, wobei der gewählte Parametersatz im Verlauf einer Simulation i. d. R. konstant bleibt. So ist es beispielsweise in SPICE-Modellen üblich eine konstante Betriebstemperatur anzugeben oder für Worst-Case Abschätzungen einige Fertigungsvarianzen über Parametersätze zu modellieren.

$$\mathscr{X}^n = \prod_{i=1}^{n} \mathscr{X}_i \qquad \mathscr{X} \in \{\mathscr{X}^n | n \in \mathbb{N}_0^+\} \tag{2.6}$$

$$\mathbf{x}(z) \in \mathscr{X} \tag{2.7}$$

$$\mathbf{x}(z) = (x_1(z), x_2(z), \ldots, x_n(z)) \tag{2.8}$$

Definition 2.2: Wenn für alle Elemente z_i einer Zustandsgröße die Komponente $\mathbf{x}(z_i)$ identisch ist, so ist die Zustandsgröße *parametrisiert, aber nicht ortsabhängig*.

Die Unterscheidung in ortsabhängige und nicht ortsabhängige Zustandsgrößen ist für die Wahl eines Simulationsverfahrens wichtig. Modelle ohne ortsabhängige Zustandsgrößen können mit einfacheren Algorithmen simuliert werden, so dass die Unterscheidung zwischen ortsabhängigen und lediglich parametrisierten, aber nicht ortsabhängigen Zustandsgrößen für eine Abschätzung des Simulationsaufwands sinnvoll ist.

Eine Zustandsgröße $\mathbf{z}$ ist somit eine Teilmenge des Raumes $\mathscr{Z}$:

$$\mathbf{z} \subseteq \mathscr{Z} = \mathscr{U} \times \mathscr{T} \times \mathscr{X} \tag{2.9}$$

$$\mathbf{z} = \{z_1, z_2, z_3, \ldots\} \qquad \text{mit } z_i = (u(z_i), t(z_i), \mathbf{x}(z_i)) \tag{2.10}$$

Definition 2.3: Eine *funktionale Zustandsgröße* beschreibt eine eindeutige Projektion $\mathscr{T} \times \mathscr{X} \to \mathscr{U}$ so, dass für alle $z_i, z_j \in \mathbf{z}$, welche die folgenden 2 Bedingungen erfüllen,

$$t(z_i) = t(z_j) \qquad \text{und} \qquad \mathbf{x}(z_i) = \mathbf{x}(z_j)$$

auch die Werte übereinstimmen müssen: $u(z_i) = u(z_j)$

Definition 2.4: Wenn für alle Elemente z_i, z_j einer funktionalen Zustandsgröße $\mathbf{z}$ mit $\mathbf{t}(z_i) = \mathbf{t}(z_j)$ auch $\mathbf{u}(z_i) = \mathbf{u}(z_j)$ gilt, so ist die Zustandsgröße ebenfalls *nicht ortsabhängig*.

Definition 2.5: Die Teilmenge einer Zustandsgröße mit dem *Tag* t_i ist $\mathbf{z}(t_i)$.

$$\mathbf{z}(t_i) = \{z | z \in \mathbf{z} \wedge t(z) = t_i\} \tag{2.11}$$

Bei einer funktionale Zustandsgröße heißt diese Teilmenge *Zustand*. Für funktionale Zustandsgrößen, deren t aus einer endlich abzählbaren Menge sind, wird $\mathbf{z}(t_i)$ als *Ereignis* bezeichnet.

Eine Zustandsgröße $\mathbf{z}$ ist stets eine vollständig zeitlich geordnete Menge, da die Menge der *Tags* $\mathscr{T}_{\mathbf{z}}$ eine vollständig geordnete Menge ist.

Definition 2.6: Für die zeitliche Ordnung von Elementen einer Zustandsgröße wird das Symbol $\preccurlyeq$ definiert.
Es sei $z_1 \preccurlyeq z_2$, genau dann wenn $t(z_1) \leq t(z_2)$. Damit gilt auch:

$$\mathbf{z}(t_1) \preccurlyeq \mathbf{z}(t_2) \quad \text{falls } t_1 \leq t_2 \qquad \forall t_1, t_2 \in \mathscr{T}_{\mathbf{z}} \tag{2.12}$$

Ein z_1 ist ein Vorgänger von z_2, wenn $z_1 \preccurlyeq z_2$ wahr ist und $z_2 \preccurlyeq z_1$ falsch ist.

Definition 2.7: Für Zustandsgrößen ohne Ortsabhängigkeit kann die verkürzte Schreibweise aus Gl. (2.13) verwendet werden, wobei die fehlenden Informationen über $\mathbf{x}$ dann als Randbedingungen definiert werden.

$$\mathbf{z}' \subseteq \mathscr{U} \times \mathscr{T} \qquad \mathbf{x}(\mathbf{z}') = \mathbf{x}_0 \tag{2.13}$$

Für die Analyse von Modellbeschreibungen ist eine Unterscheidung in sich kontinuierlich ändernde Zustandsgrößen und in sich sprunghaft ändernde Zustandsgrößen sinnvoll.

Definition 2.8: Eine Menge $\mathscr{A}$ ist *kontinuierlich*, wenn sie vollständig geordnet ist und für alle a_i, a_j aus dieser Menge, für die $a_i < a_j$ gilt, ein $a_k \in \mathscr{A}$ existiert, mit $a_i < a_k < a_j$.

Definition 2.9: Eine Menge $\mathscr{A}$ ist *stückweise kontinuierlich*, wenn es eine endliche Anzahl an kontinuierlichen Mengen $\mathscr{A}_k$ gibt, so dass

$$\mathscr{A} = \bigcup_{\forall k \leq k_0} \mathscr{A}_k \tag{2.14}$$

wobei mindestens eine der $\mathscr{A}_k$ mehr als ein Element haben muss.

Eine kontinuierliche Menge ist somit ein Spezialfall der stückweise kontinuierlichen Mengen.

Definition 2.10: Eine Menge, die weder kontinuierlich, noch stückweise kontinuierlich ist, wird *nicht kontinuierlich* genannt.

Definition 2.11: Eine Zustandsgröße $\mathbf{z}$ ist *zeitkontinuierlich*, wenn $\mathscr{T}_{\mathbf{z}}$ eine stückweise kontinuierliche Menge ist und für alle $t_0 \in \mathscr{T}_{\mathbf{z}}$ ein $z \in \mathbf{z}$ mit $t(z) = t_0$ existiert.

Definition 2.12: Eine Zustandsgröße $\mathbf{z}$ ist *ereignisdiskret*, wenn die Anzahl der $\mathbf{z}(t)$ mit $t < t_0$ für alle $t_0 \in \mathscr{T}_{\mathbf{z}}$ abzählbar endlich ist, d. h. wenn:

$$\text{mit } \mathscr{A}_{t_0} = \{\mathbf{z}(t_i) | t_i \leq t_0\} \tag{2.15}$$

$$\text{für alle } t_0 \in \mathscr{T}_{\mathbf{z}} \text{ ein } N_{t_0} \text{ existiert, mit } N_{t_0} \in \mathbb{N}^+, \text{ so dass} \tag{2.16}$$

$$|\mathscr{A}_{t_0}| < N_{t_0} \tag{2.17}$$

Definition 2.13: Eine Zustandsgröße $\mathbf{z}$ ist *stetig*, wenn für alle $z_i \in \mathbf{z}$ gilt:

$$\lim_{t(z_i)-t(z_j)\to 0}(u(z_i)-u(z_j)) = 0 \quad \text{mit } z_j \in \mathbf{z} \wedge \mathbf{x}(z_i) = \mathbf{x}(z_j) \tag{2.18}$$

Definition 2.14: Eine Zustandsgröße $\mathbf{z}$ ist *stückweise stetig*, wenn die Menge der $\mathbf{z}(t)$, welche die folgenden zwei Bedingungen erfüllen, endlich abzählbar ist:

- Es sei $t < t_0$, mit $t_0 \in \mathscr{T}_{\mathbf{z}}$ beliebig, aber fest gewählt.
- Es existiere ein $z_i \in \mathbf{z}(t)$, das Gl. (2.18) *nicht* erfüllt.

Definition 2.15: Eine Zustandsgröße $\mathbf{z}$ ist *wertdiskret*, wenn die Anzahl der Zustandswerte $u(z)$ mit $t(z) < t_0$ für alle $t_0 \in \mathscr{T}_{\mathbf{z}}$ abzählbar endlich ist, d. h. wenn:

$$\text{mit } \mathscr{A}_{t_0} = \{u(z)|t(z) \leq t_0 \wedge z \in \mathbf{z}\} \tag{2.19}$$

$$\text{für alle } t_0 \in \mathscr{T}_{\mathbf{z}} \text{ ein } N_{t_0} \text{ existiert, mit } N_{t_0} \in \mathbb{N}^+, \text{ so dass} \tag{2.20}$$

$$|\mathscr{A}_{t_0}| < N_{t_0} \tag{2.21}$$

Definition 2.16: Für die Beschreibung eines Modells werden oft mehrere Zustandsgrößen benötigt, ein Tupel aller benötigter Zustandsgrößen wird $\mathbf{Z}$ genannt:

$$\mathbf{Z} = (\mathbf{z}_i|i \in \mathbb{N}^+ \wedge i \leq N) \qquad \mathbf{Z} \in \mathscr{Z}^N = \mathscr{Z}_{I_{\text{ges}}} \tag{2.22}$$

Die Ordnung der Zustandsgrößen in $\mathbf{Z}$ ist willkürlich, aber fest, so dass jede Zustandsgröße des Tupels über ihren Index identifizierbar ist. Die Menge aller dieser $\mathbf{Z}$ ist der Zustandsraum $\mathscr{Z}_{I_{\text{ges}}}$.

Definition 2.17: Falls die Menge aller *Tags* aller Zustandsgrößen aus allen $\mathbf{Z}$ aus $\mathscr{Z}_{I_{\text{ges}}}$ eine vollständig geordnete Menge ist, dann repräsentiert diese Menge eine *globale Zeit* und jedes $\mathbf{Z}$ ist dann *vollständig zeitlich geordnet*.

Definition 2.18: Aus einem $\mathbf{Z}$ können auch kleinere Tupel $\mathbf{Z}_I$ abgeleitet werden, wobei I die darin enthaltenen Zustandsgrößen, sowie deren Sortierung bestimmt. Durch ein Tupel $\mathbf{Z}_I$ kann aus $\mathscr{Z}_{I_{\text{ges}}}$ eine Teilmenge äquivalenter $\mathbf{Z}$ bestimmt werden: Alle $\tilde{\mathbf{Z}}$, deren $\tilde{\mathbf{z}}_{i_k}$ mit denen in $\mathbf{Z}_I$ übereinstimmen, sind in $\mathscr{Z}_{I_{\text{ges}}}(\mathbf{Z}_I)$ enthalten.

$$\mathbf{Z}_I = (\mathbf{z}_i|i \in I) \qquad \mathbf{Z}_I \in \mathscr{Z}_I = \mathscr{Z}^{k_0} \tag{2.23}$$

$$\text{mit } I = (i_k|i_k, k \in \mathbb{N}^+ \wedge k \leq k_0 \wedge i_k, k_0 \leq N) \tag{2.24}$$

$$\mathscr{Z}_{I_{\text{ges}}}(\mathbf{Z}_I) = \{\tilde{\mathbf{Z}}|\tilde{\mathbf{z}}_i = \mathbf{z}_i \wedge i \in I\} \tag{2.25}$$

$$(\mathscr{Z}_{I_{\text{ges}}}(\tilde{\mathbf{Z}}_{I_1}))_{I_2} = \{\mathbf{Z}_{I_2}|\mathbf{z}_k = \tilde{\mathbf{z}}_k \forall (k \in I_1)\} \tag{2.26}$$

Definition 2.19: Wenn keine der Zustandsgrößen, die in $\mathbf{Z}_I$ enthalten sind, ortsabhängig ist, so ist $\mathbf{Z}_I$ ebenfalls *nicht ortsabhängig*. Dann wird mit $\mathbf{x}(\mathbf{Z}_I)$ der Parametersatz bezeichnet, der das Tupel der $\mathbf{x}(\mathbf{z}_i)$ enthält:

$$\begin{aligned}\mathbf{x}(\mathbf{Z}_I) = (&\{\mathbf{x}|\exists((\mathbf{x}(z) = \mathbf{x}) \wedge (z \in \mathbf{z}_{I(1)}))\},\\ &\{\mathbf{x}|\exists((\mathbf{x}(z) = \mathbf{x}) \wedge (z \in \mathbf{z}_{I(2)}))\}, \ldots)\end{aligned} \tag{2.27}$$

Eine Modellbeschreibung schränkt die erlaubten Zustände ein. Das kann genutzt werden, um eine zur Modellbeschreibung äquivalente Teilmenge des Zustandsraums zu definieren, die im folgenden $\mathscr{M}$ genannt wird. Zur Erleichterung der Notation kann $\mathscr{M}$ auch über einem Teil des Zustandsraums $\mathscr{Z}_{I_M}$ definiert werden, wobei Zustandsgrößen, die nicht in diesem Raum enthalten sind, beliebig gewählt werden dürfen. Ein $\mathbf{Z}$ erfüllt somit die Modellbeschreibung, wenn $\mathbf{Z}_{I_M} \in \mathscr{M}$.

$$\mathscr{M} \subseteq \mathscr{Z}_{I_M} \tag{2.28}$$

Definition 2.20: Enthält $\mathbf{Z}_{I_M}$ mehr als eine Zustandsgröße, so beschreibt es eine modellkonforme Relation zwischen den enthaltenen Zustandsgrößen. Es wird damit ein *Modellverhalten* beschrieben, wenn man die Zustandsgrößen als Eingangsstimuli und sich daraus ergebende Zustandsgrößen interpretiert.

Definition 2.21: Unter *Simulation einer Modellbeschreibung* versteht man die Berechnung von Zustandsgrößen, die die Modellbeschreibung erfüllen.

Ist die Modellbeschreibung durch eine Menge $\mathscr{M}$ beschrieben, deren Elemente explizit festgelegt sind, so ist die Simulation trivial. Wenn die Menge $\mathscr{M}$ statt dessen mit einem Satz an Bedingungen für $\mathscr{Z}_{I_M}$ definiert ist, so kann die Simulation, d. h. die explizite Bestimmung der Elemente der Zustandsgrößen, sehr aufwendig werden.

Definition 2.22: Eine Modellbeschreibung ist *nicht ortsabhängig*, wenn alle $\mathbf{Z}_{I_M}$, die das Modell erfüllen, den selben Parametersatz haben (siehe Gl. (2.29)) und keine der in den $\mathbf{Z}_{I_M}$ enthaltenen Zustandsgrößen $\mathbf{z}$ ortsabhängig ist.

$$\mathbf{x}(\mathbf{Z}_{I_M,i}) = \mathbf{x}(\mathbf{Z}_{I_M,j}) \quad \forall \mathbf{Z}_{I_M,i}, \mathbf{Z}_{I_M,j} \in \mathscr{M} \tag{2.29}$$

Ein Modell kann unter Verwendung von Teilmodellen beschrieben werden. Um die erlaubten Zustände im Gesamtmodell zu bestimmen, wird die Schnittmenge der Lösungsmengen der Teilmodelle berechnet. Die Schnittmenge zweier Mengen mit unterschiedlichen Indexsets I_1, I_2 wird durch implizite Expansion auf ein gemeinsames Indexset gebildet.

$$\mathscr{M}_{ges} = \bigcap_{\forall i} \mathscr{M}_i \tag{2.30}$$

Beispiel:

$$\text{mit} \quad \mathscr{M}_1 \subseteq \mathscr{Z}_{I_1} \tag{2.31}$$
$$\mathscr{M}_2 \subseteq \mathscr{Z}_{I_2} \tag{2.32}$$
$$\text{folgt} \quad \mathscr{M}_1 \cap \mathscr{M}_2 \subseteq \mathscr{Z}_{I_3} \tag{2.33}$$
$$\text{mit} \quad I_3 = I_1 \cup I_2 \tag{2.34}$$

Berechungsvorschrift:

$$\mathscr{M}_{1,I_3} = \{\mathbf{Z}_{I_3} | (\mathscr{Z}_{I_{\text{ges}}}(\mathbf{Z}_{I_3}))_{I_1} \in \mathscr{M}_1\} \tag{2.35}$$
$$\mathscr{M}_{2,I_3} = \{\mathbf{Z}_{I_3} | (\mathscr{Z}_{I_{\text{ges}}}(\mathbf{Z}_{I_3}))_{I_2} \in \mathscr{M}_2\} \tag{2.36}$$
$$\mathscr{M}_3 = \mathscr{M}_{1,I_3} \cap \mathscr{M}_{2,I_3} \tag{2.37}$$

In vielen Fällen ist es erwünscht, das Verhalten eines Modells als Reaktion auf externe Eingangsstimuli zu bestimmen. Das ist möglich, wenn man die externen Stimuli $\mathscr{S}$ in Form einer Einschränkung der erlaubten Zustände und die Lösungsmenge $\mathscr{M}$ für das Modell bestimmt hat.

Definition 2.23: Das mögliche *Verhalten des Modells* wird durch die Schnittmenge $\mathscr{V}$ beschrieben:

$$\mathscr{V} = \mathscr{S} \cap \mathscr{M} \qquad \mathscr{S} \subseteq \mathscr{Z}_{I_S} \tag{2.38}$$
$$\mathscr{V} \subseteq \mathscr{Z}_{I_V} \tag{2.39}$$

Definition 2.24: $\mathscr{M}$ ist *deterministisch* bezüglich $\mathscr{S}$, wenn

$$|\mathscr{A}(\tilde{\mathbf{Z}}_{I_S})| = \{0,1\} \qquad \forall \tilde{\mathbf{Z}}_{I_S} \in \mathscr{S} \tag{2.40}$$
$$\text{mit} \quad \mathscr{A}(\tilde{\mathbf{Z}}_{I_S}) = (\mathscr{Z}_{I_{\text{ges}}}(\tilde{\mathbf{Z}}_{I_S}))_{I_V} \cap \mathscr{V} \tag{2.41}$$

In anderen Worten: Falls ein *deterministisches Modell* die externen Stimuli $\tilde{\mathbf{Z}}_{I_S}$ verarbeiten kann, dann sind die Zustandsgrößen $\tilde{\mathbf{Z}}_{I_V}$ eindeutig bestimmt.

Definition 2.25: Die Menge der *Observablen*, die eine Teilmenge des Zustandsraums des Modells darstellt, wird $\mathscr{O}$ genannt.

$$\mathbf{O} = \mathbf{Z}_{I_O} \qquad \mathbf{O} \in \mathscr{O} \tag{2.42}$$

Definition 2.26: Ein Modell $\mathscr{M}$ ist eine *eindeutige Abbildung* $\mathscr{S} \to \mathscr{O}$, wenn für alle $\mathbf{Z}_1, \mathbf{Z}_2$, deren Stimuli identisch sind, auch die Observablen übereinstimmen:

$$(\mathbf{Z}_1)_{I_O} = (\mathbf{Z}_2)_{I_O}$$
$$\forall \{\mathbf{Z}_1, \mathbf{Z}_2 | (\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{Z}_{I_{\text{ges}}}) \wedge ((\mathbf{Z}_1)_{I_S} = (\mathbf{Z}_2)_{I_S} \in \mathscr{S})\} \tag{2.43}$$

Ein deterministisches Modell ist immer eine eindeutige Abbildung $\mathscr{S} \to \mathscr{O}$. Eine eindeutige Abbildung muss jedoch nicht deterministisch sein, d. h. das Modell kann bzgl. eines nicht beobachtbaren Zustands mehrdeutig sein.

2.3 Definition der Modellklassen

Auf Basis der in Kapitel 2.2 definierten Zustandsräume können stochastische, zeitkontinuierliche, ereignisdiskrete oder auch zeitlose Modelle formuliert werden. Im Hinblick auf Systeme aus heterogenen Sensoren und Aktoren mit elektronischer Signalverarbeitung werden Modellierungsmöglichkeiten für die folgenden sehr unterschiedlichen Anwendungsgebiete benötigt:

- Zur Beschreibung von räumlich ausgedehnten Modellen werden Modellierungskonzepte benötigt, welche die Lösung von partiellen Differentialgleichungen ermöglichen. Solche Modelle werden z. B. zur Beschreibung von Drucksensoren benötigt. Im Kontext der im letzten Kapitel vorgestellten Zustandsraumdarstellung werden dazu insbesondere zeitkausale, ortsabhängige Zustandsgrößen benötigt.

- Für die Beschreibung von zeitkontinuierlichen Modellen mit konzentrierten Grundelementen genügen einfachere Modellierungskonzepte für die effizientere Simulationsalgorithmen zur Verfügung stehen. Ein typisches Beispiel für diese Modellklasse ist die Modellierung analoger elektrischer Schaltungen für SPICE-basierte Simulatoren. In dieser Modellklasse werden Beschreibungsmöglichkeiten für gewöhnliche Differentialgleichungssysteme mit vielen, zeitkausalen, aber nicht ortsabhängigen Zustandsgrößen benötigt.

- Durch die industrielle Verfügbarkeit von Synthesewerkzeugen für synchrone digitale Schaltungen ist die Bedeutung der ereignisdiskreten Modelle stark gestiegen. Eine Vielzahl an Modellierungsansätzen, die auf ereignisdiskreten Zustandsgrößen mit gerichtetem Signalfluss basieren, wurde definiert und analysiert. Bei diesen Zustandsgrößen ändert sich der Zustand an einzelnen eindeutig definierbaren Zeitpunkten, wobei eine eindeutige Ursache/Wirkung-Beziehung zwischen den Zustandsgrößen besteht.

- Modellansätze aus dem Hardware-/Software-Codesign motivieren letztendlich eine noch abstraktere Modellierung, in der lediglich zeitkausale Zusammenhänge abgebildet werden. Die Dauer von Zustandsübergängen wird hierbei nicht betrachtet, so dass die für diese Modellklasse benötigten Zustandsgrößen einzelne Ereignisse in festgelegter Reihenfolge - aber ohne einen globalen Zeitstempel - enthalten.

Zur Klassifizierung der Modellierungsansätze werden nun domänenübergreifende Modellklassen definiert. Dabei wird zum einen der Bezug zu etablierten Modellierungskonzepten hergestellt, und zum anderen werden die Unterschiede im Zeitmodell, sowie der Mächtigkeit der Beschreibungskonzepte formal deklariert.

2.3.1 Modelle der α-Klasse

In Modellen der α-Klasse wird der Systemzustand mit einer endlichen Anzahl wertdiskreter Zustandsgrößen $\mathbf{z}$ beschrieben. Deren Elemente besitzen einen *Tag* $t(z_i)$, der eine zeitliche Ordnung zwischen den Elementen dieser Zustandsgröße definiert. Das *Tag*-System modelliert jedoch keine globale Zeit, denn die Menge, welche alle *Tags* aller Zustandsgrößen umfasst, ist im allgemeinen *nicht vollständig zeitlich geordnet*. Ebenso ist zwischen den *Tags keine Abstandsmetrik* definiert. Mit anderen Worten: zur Modellbeschreibung wird ein lokales Zeitkausalitätsprinzip zu Grunde gelegt, in dem der zeitliche Abstand zwischen zwei Zuständen irrelevant ist.

Alle Zustandsgrößen $\mathbf{z} \in \mathbf{Z} \in \mathscr{V}$ sind ortsunabhängig:

$$\mathbf{z} \in \mathscr{U}_{\mathbf{z}} \times \mathscr{T}_{\mathbf{z}} \times \{\mathbf{x}_0\} \tag{2.44}$$

Die Zustandsgrößen müssen funktionale Zustandsgrößen und im Sinne von Def. 2.12 ereignisdiskret sein. Der Wertebereich einer Zustandsgröße $\mathscr{U}_{\mathbf{z}}$ kann aus jeder Menge bestehen, die mindestens ein Element hat. Die Menge der *Tags* $\mathscr{T}_{\mathbf{z}}$ besteht aus Symbolen, für die gemäß Def. 2.1 eine zeitliche Ordnung definiert ist. Für *Tags* unterschiedlicher Zustandsgrößen muss jedoch keine zeitliche Ordnung definiert sein.

Definition 2.27: Ein Tupel $\mathbf{Z}$ aus solchen Zustandsgrößen heißt *partiell geordnet*.

In einem partiell geordneten Tupel $\mathbf{Z}$ existieren $t(z_1) \in \mathscr{T}_{\mathbf{z}_1}$, $t(z_2) \in \mathscr{T}_{\mathbf{z}_2}$, so dass

$$(t(z_1) \neq t(z_2)) \wedge (t(z_1) \not< t(z_2)) \wedge (t(z_2) \not< t(z_1)) \tag{2.45}$$

Modelle der α-Klasse unterscheiden sich in zwei Hauptgruppen: sequenz-orientierte Modelle und *Tag*-orientierte Modelle.

2.3.1.1 Sequenz-orientierte Modelle

Ein sequenz-orientiertes Modell der α-Klasse verarbeitet alle Ereignisse der Zustandsgrößen in $\mathscr{S}$ sequentiell, d. h. der *Tag* der Ereignisse dient zwar zur Definition einer Reihenfolge der Ereignisse in $\mathbf{z}$, der Wert des *Tags* wird jedoch im Modell nicht verwendet.

Definition 2.28: Das nach *Tags* geordnete Tupel der Werte aller Ereignisse einer funktionalen Zustandsgröße $\mathbf{z}$ sei $\mathrm{Seq}(\mathbf{z})$:

$$\mathrm{Seq}(\mathbf{z}) = (u_i | (u_i, t_i, \mathbf{x}_0) = \mathbf{z}(t_i) \wedge t_i < t_{i+1}) \tag{2.46}$$

Für die Beschreibung von sequenz-orientieren Modellen ist es hilfreich Zusammenhänge zwischen den Werten des i-ten Elements einer Zustandsgröße und des j-ten Elements einer anderen Zustandsgröße notieren zu können. Die dafür benötigte zeitkausale Ordnung der Werte der Zustandsgrößen ist über $\mathrm{Seq}(\mathbf{z}_1)$ und $\mathrm{Seq}(\mathbf{z}_2)$ definiert, für den Zugriff auf ein Element dieser Sequenzen wird die $\mathrm{Seq}(\mathbf{z}, i)$ eingeführt.

Definition 2.29: Der Zugriff auf das i-te Element von $\mathrm{Seq}(\mathbf{z})$ sei über $\mathrm{Seq}(\mathbf{z}, i)$ möglich.

$$\mathrm{Seq}(\mathbf{z}) = (\mathrm{Seq}(\mathbf{z}, 1), \mathrm{Seq}(\mathbf{z}, 2), \ldots) \tag{2.47}$$

Anhand einer kurzen Modellbeschreibung wird der Nutzen dieser Definition veranschaulicht: Es wird ein Zähler modelliert, dessen Eingang durch die Zustandsgröße $\mathbf{z}_1$ und dessen Ausgang durch die Zustandsgröße $\mathbf{z}_2$ beschrieben ist. Die Eingangsgröße $\mathbf{z}_1$ besteht aus einer funktionalen, ereignisdiskreten, nicht ortsabhängigen Zustandsgröße deren Werte aus der Menge $\{-1; +1\}$ stammen. Der Wert der Zustandsgröße $\mathbf{z}_2$ soll je nach Wert von $\mathbf{z}_1$ inkrementiert oder dekrementiert werden. Dementsprechend wird $\mathbf{z}_2$ ebenfalls eine funktionale, ereignisdiskrete und nicht ortsabhängige Zustandsgröße sein. Das Modell kann wie folgt notiert werden:

$$\begin{aligned} \mathbf{Z} &= (\mathbf{z}_1, \mathbf{z}_2) \\ \mathscr{U}_{\mathbf{z}_1} &= \{-1; +1\} & \mathscr{U}_{\mathbf{z}_2} &= \mathscr{N} \\ \mathscr{X}_{\mathbf{z}_1} &= \{\emptyset\} & \mathscr{X}_{\mathbf{z}_2} &= \{\emptyset\} \end{aligned}$$

$$\mathbf{S} = (\mathbf{z}_1) = \mathbf{Z}_{\{1\}} := (\{+1, -1, +1, +1, -1, \ldots\}) \tag{2.48}$$

$$\begin{aligned} \mathscr{M} = \{\mathbf{Z} | &(\mathrm{Seq}(\mathbf{z}_2, 1) = 0) \wedge \\ &(\mathrm{Seq}(\mathbf{z}_2, i+1) = \mathrm{Seq}(\mathbf{z}_2, i) + \mathrm{Seq}(\mathbf{z}_1, i) \quad \forall i > 0)\} \end{aligned} \tag{2.49}$$

Durch Gl. (2.49) wird $\mathbf{z}_2$ nicht eindeutig festgelegt, da die Zeitpunkte der Ereignisse nicht definiert sind. Die Modellbeschreibung enthält nur eine Aussage über die Werte der Zustandsgröße und die Reihenfolge, in der diese auftreten. Daher gibt es viele $\mathbf{z}_2$, die das Modell erfüllen: Sie sind *sequenz-äquivalent*.

Definition 2.30: Zwei funktionale Zustandsgrößen $\mathbf{z}_1, \mathbf{z}_2$ sind *sequenz-äquivalent*, wenn $\mathrm{Seq}(\mathbf{z}_1) = \mathrm{Seq}(\mathbf{z}_2)$.

Definition 2.31: Zwei Tupel aus Zustandsgrößen $\mathbf{Z}_1, \mathbf{Z}_2$ sind *sequenz-äquivalent*, wenn $|\mathbf{Z}_1| = |\mathbf{Z}_2|$ ist und alle in $\mathbf{Z}_1$ und $\mathbf{Z}_2$ enthaltenen Zustandsgrößen funktional und paarweise sequenz-äquivalent sind:

$$\begin{aligned} &\mathrm{Seq}((\mathbf{Z}_1)) = \mathrm{Seq}((\mathbf{Z}_2)) \Leftrightarrow \\ &\qquad \Big(\mathrm{Seq}((\mathbf{Z}_1)_{\{i\}}) = \mathrm{Seq}((\mathbf{Z}_2)_{\{i\}}) \quad \forall i \leq |\mathbf{Z}_1|\Big) \end{aligned} \tag{2.50}$$

Definition 2.32: Ein Modell $\mathscr{M}$ ist *sequenz-deterministisch* bezüglich $\mathscr{S}$, wenn alle $\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{V}$ sequenz-äquivalent sind.

Definition 2.33: Ein Modell ist eine sequenz-eindeutige Abbildung $\mathscr{S} \rightarrow \mathscr{O}$, wenn für alle $\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{V}$ gilt:

$$\text{aus } \mathrm{Seq}((\mathbf{Z}_1)_{I_S}) = \mathrm{Seq}((\mathbf{Z}_2)_{I_S}) \tag{2.51}$$

$$\text{folgt } \mathrm{Seq}((\mathbf{Z}_1)_{I_O}) = \mathrm{Seq}((\mathbf{Z}_2)_{I_O}) \tag{2.52}$$

Definition 2.34: Ein Tupel $\mathbf{A}$ ist ein *Prefix* von $\mathbf{B}$, wenn

$$|\mathbf{A}| \leq |\mathbf{B}| \tag{2.53}$$

$$\text{und } a_i = b_i \quad \forall i \leq |\mathbf{A}| \tag{2.54}$$

$$\text{mit } \mathbf{A} = (a_i | i \in \mathbb{N}^+ \wedge i \leq k_1) \tag{2.55}$$

$$\mathbf{B} = (b_i | i \in \mathbb{N}^+ \wedge i \leq k_2) \tag{2.56}$$

Definition 2.35: Das Zeichen '$\sqsubseteq$' wird verwendet um eine Prefix-Sortierung zu notieren. Wenn $\mathbf{A}$ ein Prefix von $\mathbf{B}$ ist, gilt:

$$\mathbf{A} \sqsubseteq \mathbf{B} \tag{2.57}$$

Definition 2.36: Ein sequenz-kausales Modell ist eine sequenz-eindeutige Abbildung $\mathscr{S} \rightarrow \mathscr{O}$ in der für alle $\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{V}$ gilt:

$$\text{wenn} \qquad \mathrm{Seq}((\mathbf{Z}_1)_{I_S}) \sqsubseteq \mathrm{Seq}((\mathbf{Z}_2)_{I_S}) \tag{2.58}$$

$$\text{folgt daraus:} \qquad \mathrm{Seq}((\mathbf{Z}_1)_{I_O}) \sqsubseteq \mathrm{Seq}((\mathbf{Z}_2)_{I_O}) \tag{2.59}$$

Sequenz-orientierte Modelle der α-Klasse müssen sequenz-kausal sein.

2.3.1.2 *Tag*-orientierte Modelle

In *Tag*-orientierten Modellen der α-Klasse werden Relationen zwischen Zustandsgrößen aus $\mathscr{S}$ und freien Zustandsgrößen für Ereignisse mit identischen *Tags* definiert:

$$\begin{aligned} &\text{Mit } I_S = \{1\}, I_O = \{2\}, \mathbf{Z} = (\mathbf{z}_1, \mathbf{z}_2): \\ &\quad \mathscr{S} \rightarrow \mathscr{O} \Leftrightarrow \{\mathbf{z}_1(t) | \mathbf{Z} \in \mathscr{S} \wedge t \in (\mathscr{T}_{z_1} \cap \mathscr{T}_{z_2})\} \rightarrow \mathbf{z}_2 \end{aligned} \tag{2.60}$$

Ereignisse, deren *Tags* nicht zu $\mathscr{T}_{z_2}$ gehören, sind für $\mathbf{z}_2$ nicht sichtbar. Die für die Bestimmung von $\mathbf{z}_2$ zur Verfügung stehende Datenmenge wird somit verringert. Diese Form der Modellbildung ist unter anderem zur Beschreibung von verteilten Systemen geeignet. In diesen wird mit lokalen Subsystemen auf lokalen Zustandsgrößen gearbeitet, wobei nur zu festgelegten Zeitpunkten ein Datenaustausch zwischen den Teilsystemen stattfindet. Das ermöglicht eine effizientere Simulation von großen Systemen mit vielen solchen Teilsystemen als die anderen Modellklassen.

2.3.1.3 Ursache-/Wirkungsparadigma für *Tag*-orientierte Modelle der α-Klasse

Tag-orientierte Modelle der α-Klasse müssen streng zeitkausale und eindeutige Abbildungen $\mathscr{S} \rightarrow \mathscr{O}$ sein.

Definition 2.37: Eine Abbildung ist *streng zeitkausal*, wenn für alle $\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{V}$ gilt:

$$\forall k \in I_O :$$

$$(\mathbf{Z}_1)_{\{k\}}(t) = (\mathbf{Z}_2)_{\{k\}}(t) \quad \forall t \leq \min\left\{\left(\bigcup_{i \in I_S} \mathscr{A}_i\right) \cap \mathscr{T}_{z_k}\right\} \tag{2.61}$$

$$\text{mit } \mathscr{A}_i = \{t | t \in \mathscr{T}_{z_i} \wedge (\mathbf{Z}_1)_{\{i\}}(t) \neq (\mathbf{Z}_2)_{\{i\}}(t)\} \tag{2.62}$$

In streng zeitkausalen Modellen können für alle Zustandsgrößen aus $\mathscr{O}$ explizite Zustandsübergangsfunktionen bestimmt werden, deren Wert durch frühere $\mathbf{Z}(t)$ bestimmt wird.

Definition 2.38: Die Zustandsgrößen aus streng zeitkausalen Modelle werden im Folgenden auch *Signale* genannt.

2.3.2 Modelle der β-Klasse

In Modellen der β-Klasse wird der Systemzustand, so wie bei den Modellen der α-Klasse, durch eine endliche Anzahl ortsunabhängiger Zustandsgrößen festgelegt. Alle Zustandsgrößen müssen funktionale Zustandsgrößen sein und die Modelle der β-Klasse müssen eine eindeutige Abbildung $\mathscr{S} \rightarrow \mathscr{O}$ beschreiben.

Im Unterschied zur α-Klasse sind die *Tags* der Elemente aller Zustandsgrößen aus einer gemeinsamen Obermenge, so dass für die Modellbildung eine *globale Zeit* zur Verfügung steht.

Definition 2.39: Jede Zustandsgröße eines Modells der β-Klasse muss zeitkontinuierlich oder ereignisdiskret sein. Andere Zustandsgrößen sind nicht erlaubt.

Definition 2.40: Ein Modell der β-Klasse ist ein *ereignisdiskretes Modell*, wenn alle Zustandsgrößen des Modells ereignisdiskret sind. Andernfalls ist es ein zeitkontinuierliches Modell.

Definition 2.41: Für jede Zustandsgröße $\mathbf{z}_i$ eines Modells der β-Klasse gilt:

$$\mathscr{T}_{\mathbf{z}_i} \subseteq \mathbb{R} \qquad \mathbb{T} = \bigcup_{\forall i} \mathscr{T}_{\mathbf{z}_i} \tag{2.63}$$

$$\mathbf{z}_i \in \mathscr{U}_{\mathbf{z}_i} \times \mathscr{T}_{\mathbf{z}_i} \times \{\mathbf{x}_i\} \tag{2.64}$$

Jede Menge $\mathscr{T}_{\mathbf{z}_i}$ muss eine Teilmenge der reellen Zahlen sein. Diese Randbedingung erweitert die Beschreibungsmöglichkeiten gegenüber Modellen der α-Klasse deutlich. Die *Tags* der Elemente *aller* Zustandsgrößen sind paarweise zeitlich geordnet[5]: Man spricht von einem *vollständig zeitlich geordneten Tag*system. Der Zustandsraum von Modellen der β-Klasse ist somit ebenfalls vollständig zeitlich geordnet (s. Def. 2.17).

Zusätzlich kann der zeitliche Abstand zwischen zwei beliebigen z_i, z_j zur Modellbeschreibung verwendet werden: Um ein Maß für die Übereinstimmung zweier Zustandsgrößen zu haben, wird die Cantor-Metrik benutzt. Ist $d(\mathbf{z}_1,\mathbf{z}_2) = 0$, so sind die Zustandsgrößen identisch, andernfalls ist $d(\mathbf{z}_1,\mathbf{z}_2)$ umso größer, desto *früher* sich die Zustandsgrößen unterscheiden.

Definition 2.42: Cantor-Metrik:

$$d(\mathbf{z}_1,\mathbf{z}_2) = \max\{e^{-t} | t \in \mathbb{T} \subseteq \mathbb{R} \wedge \mathbf{z}_1(t) \neq \mathbf{z}_2(t)\} \tag{2.65}$$

Eine Metrik muss die folgenden vier Bedingungen erfüllen:

$$d(\mathbf{z}_1,\mathbf{z}_2) = d(\mathbf{z}_2,\mathbf{z}_1) \tag{2.66}$$

$$d(\mathbf{z}_1,\mathbf{z}_2) \geq 0 \tag{2.67}$$

$$d(\mathbf{z}_1,\mathbf{z}_2) = 0 \quad \Leftrightarrow \quad \mathbf{z}_1 = \mathbf{z}_2 \tag{2.68}$$

$$d(\mathbf{z}_1,\mathbf{z}_2) + d(\mathbf{z}_2,\mathbf{z}_3) \geq d(\mathbf{z}_1,\mathbf{z}_3) \tag{2.69}$$

Damit ist die Cantor-Metrik tatsächlich eine Metrik.

In den vollständig zeitlich geordneten Zustandsräumen mit $\mathbb{T} \subseteq \mathbb{R}$ kann diese Metrik auch für Tupel $\mathbf{Z}$ definiert werden:

Definition 2.43:

$$d(\mathbf{Z}_i,\mathbf{Z}_j) = \max\{e^{-t} | t \in \mathbb{T} \subseteq \mathbb{R} \wedge \mathbf{Z}_i(t) \neq \mathbf{Z}_j(t)\} \tag{2.70}$$

$$\mathbf{Z}_k(t) = ((\mathbf{Z}_k)_{\{1\}}(t), (\mathbf{Z}_k)_{\{2\}}(t), \ldots) \tag{2.71}$$

Mit Hilfe dieser Metrik wird die Eigenschaft der *Zeitkausalität* für Modelle in vollständig zeitlich geordneten Zustandsräumen, deren $\mathbb{T} \subseteq \mathbb{R}$ ist, definiert.

Definition 2.44: Eine eindeutige Abbildung $\mathscr{S} \rightarrow \mathscr{O}$ ist *zeitkausal*, wenn für alle $\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{V}$ gilt:

$$d((\mathbf{Z}_1)_{I_O}, (\mathbf{Z}_2)_{I_O}) \leq d((\mathbf{Z}_1)_{I_S}, (\mathbf{Z}_2)_{I_S}) \tag{2.72}$$

Def. 2.44 sagt aus, dass sich zwei mögliche Observablen-Tupel nicht früher unterscheiden können, als ihre Eingangs-Tupel.

[5] *Paarweise zeitlich geordnet* heißt, dass für jedes beliebige Paar aus der Menge der *Tags* eine zeitliche Ordnung existiert.

Definition 2.45: Eine eindeutige Abbildung $\mathscr{S} \rightarrow \mathscr{O}$ ist *streng zeitkausal*, wenn für alle $\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{V}$ gilt:

$$d((\mathbf{Z}_1)_{I_O}, (\mathbf{Z}_2)_{I_O}) < d((\mathbf{Z}_1)_{I_S}, (\mathbf{Z}_2)_{I_S}) \tag{2.73}$$

Definition 2.46: Alle Modelle der β-Klasse müssen *streng zeitkausal* sein, d. h. jedes $\mathbf{z}_i(t_0)$ mit $i \notin I_S$, wird durch die Menge aller $\mathbf{Z}(t)$ mit $t < t_0$ eindeutig bestimmt.

Wie in der α-Klasse sind die Zustandsgrößen in Modellen der β-Klasse *Signale*, in dem Sinn, das für jede Zustandsgröße eine explizite Zustandsübergangsfunktion formuliert werden kann, die den Wert der Zustandsgröße festlegt.

Definition 2.47: Ein delta-kausales Modell ist ein streng zeitkausales Modell, in dem für alle $\mathbf{Z}_1, \mathbf{Z}_2 \in \mathscr{V}$ ein $\delta < 1$ existiert, so dass gilt:

$$d((\mathbf{Z}_1)_{I_A}, (\mathbf{Z}_2)_{I_A}) < \delta \cdot d((\mathbf{Z}_1)_{I_S}, (\mathbf{Z}_2)_{I_S}) \qquad \delta \in \mathbb{R}^+ \tag{2.74}$$

Def. 2.47 sagt aus, dass sich eine Änderung der Eingangsstimuli frühestens nach einer Verzögerung von $\Delta = ln(\delta^{-1})$ auf andere Zustandsgrößen auswirkt. Diese Eigenschaft garantiert, dass Zustandsgrößen, die sich selbst über eine Rückkopplung beeinflussen, ein wohldefiniertes Verhalten aufweisen. Wenn in einem nicht delta-kausalen Modell eine Rückkopplung enthalten ist, kann ein nicht auflösbarer Konflikt entstehen. Um das zu verhindern, muss bei der Modellbildung darauf geachtet werden, dass die Rückkopplung zu einem stabilen Zustand führt, während bei delta-kausalen Modellen eine Simulierbarkeit in jedem Fall garantiert werden kann.

Die Wertebereiche der Zustandsgrößen $\mathscr{U}_{\mathbf{z}}$ können aus jeder Menge bestehen, die mindestens ein Element hat.

Bei ereignisdiskreten Modellen ist der Systemzustand stückweise konstant. Für diese Modelle wird unter Simulation die exakte Bestimmung des Systemzustands als Funktion der Zeit verstanden. Bei zeitkontinuierlichen Modellen der β-Klasse versteht man unter Simulation die Berechnung des Systemzustands für chronologisch geordnete Zeitpunkte.

2.3.3 Modelle der γ-Klasse

Modelle der γ-Klasse müssen eine eindeutige Abbildung $\mathscr{S} \rightarrow \mathscr{O}$ beschreiben. Alle Zustandsgrößen $\mathbf{z}_i$ aus $\mathbf{Z} \in \mathscr{V}$ müssen ortsunabhängige, funktionale Zustandsgrößen sein. Die Zustandsgrößen sind zeitkontinuierlich und stückweise stetig. Für jede Zustandsgröße $\mathbf{z}_i$ gilt:

$$\mathscr{U}_{\mathbf{z}_i} \subseteq \mathbb{R} \tag{2.75}$$

$$\mathscr{T}_{\mathbf{z}_i} \subseteq \mathbb{R} \qquad \mathbb{T} = \bigcup_{\forall i} \mathscr{T}_{\mathbf{z}_i} \tag{2.76}$$

$$\mathbf{z}_i \in \mathscr{U}_{\mathbf{z}_i} \times \mathscr{T}_{\mathbf{z}_i} \times \{\mathbf{x}_i\} \tag{2.77}$$

Die Mengen $\mathscr{T}_{\mathbf{Z}_i}$ müssen *stückweise kontinuierliche* Mengen sein. Wie man an Gl. (2.76) sieht, besitzen Modelle der γ-Klasse eine globale Zeit und ihre Zustandsräume sind vollständig zeitlich geordnet.

Im Unterschied zur β-Klasse gibt es in der γ-Klasse keine eindeutige Ursache/Wirkung-Beziehung zwischen den Zustandsgrößen; Modelle der γ-Klasse müssen lediglich zeitkausal sein. Zum Nachweis der Zeitkausalität kann Def. 2.44 verwendet werden.

Definition 2.48: Man unterscheidet zwischen *strukturvarianten* und *strukturinvarianten* Modellen. Bei den strukturinvarianten Modelle ist $\mathbb{T}$ eine kontinuierliche Menge und es gilt:

$$\mathscr{T}_{\mathbf{Z}_i} = \mathbb{T} \qquad \forall \mathscr{T}_{\mathbf{Z}_i} \tag{2.78}$$

Bei den strukturvarianten Modellen existieren Zustandsgrößen, deren $\mathscr{T}_{\mathbf{Z}_i}$ sich unterscheiden.

Mit solchen Zustandsübergangsfunktionen können implizite gewöhnliche Differentialgleichungssysteme beschrieben werden:

$$0 = f\left[Z_{I_S}(t), Z(t), \dot{Z}(t)\right] \tag{2.79}$$

Ein Anfangszustand und die Anregungen werden dann mit Hilfe der Menge $\mathscr{S}$ festgelegt. Die Menge $\mathscr{M}$ ist die Lösungsmenge der arithmetische Funktion f. Ein weiterer Spezialfall der Modelle der γ-Klasse sind zeitkontinuierliche Modelle, die ohne Differentialgleichungen beschreibbar sind. Diese gehören ebenfalls zu den Modellen der γ-Klasse, wenn die Zustandsgrößen dieser Modelle nicht streng zeitkausal sind.

Bei den Modellen der γ-Klasse ist eine Unterteilung in *konservative Modelle* und *nicht konservative Modelle* hilfreich.

2.3.3.1 Konservative Modelle der γ-Klasse

In konservativen Modellen wird der Systemzustand durch eine Anzahl von Flussgrößen und dazugehöriger Potentialgrößen beschrieben. Für diese Größen gelten Erhaltungssätze. Die Modellierung mit Potential- und Flussgrößen kann die Beschreibung von physikalischen Systemen erleichtern, welche aus Komponenten enthalten, die mehrfach benötigt werden. Der Zwang alle Zustandsgrößen auf Potential- und Flussgrößen abzubilden kann die Modellierung eines konzeptionellen Prototypen erschweren. Er ermöglicht jedoch eine implementierungsnahe Modellbildung. Konservative Modelle sind gut geeignet für eine iterative Modellverfeinerung, da Zusammenhänge zwischen Modell und physikalischer Implementierung einfacher ausgedrückt werden können. Zusätzlich ermöglicht diese Art der Modellbildung implizite Konsistenzprüfungen, mit denen manche Modellbildungsfehler schon vor der Simulation erkannt werden können.

Die jeweils paarweise genutzten Potential- und Flussgrößen $(\mathbf{p}_k, \mathbf{f}_k)$ werden paarweise in das Tupel $\mathbf{Z}$ aus Zustandsgrößen aufgenommen, so dass gilt:

$$\begin{aligned}\mathbf{Z} = (\mathbf{z}_i | ((\forall i = 2k-1 : \mathbf{z}_i = \mathbf{p}_k) \wedge \\ (\forall i = 2k : \mathbf{z}_i = \mathbf{f}_k) | (k \in \mathbb{N}^+) \wedge (k \leq N/2)))\end{aligned} \tag{2.80}$$

Ein konservatives Modell ist in der Regel aus mehreren Teilmodellen aufgebaut, die in zwei Mengen unterteilt werden können. Die Menge der Verbindungsmodelle (Connector) $\mathscr{C}$ und die Menge der Komponentenmodelle (Behavior) $\mathscr{B}$.

$$\mathscr{C} = \{\mathscr{M}_{C,1}, \mathscr{M}_{C,2}, \ldots\} \tag{2.81}$$

$$\mathscr{B} = \{\mathscr{M}_{B,1}, \mathscr{M}_{B,2}, \ldots\} \tag{2.82}$$

Die Komponentenmodelle $\mathscr{M}_{B,i}$ sind über zueinander disjunkte Mengen $I_{B,i}$ definiert, d. h. ihr Verhalten wird ausschließlich über die Menge der Konnektoren eingeschränkt.

$$\mathscr{M}_{B,i} \subseteq \mathscr{Z}_{I_{B,i}} \tag{2.83}$$

$$(2k-1) \in I_{B,i} \Leftrightarrow (2k) \in I_{B,i} \qquad \forall i, k \tag{2.84}$$

$$|I_{B,i} \cap I_{B,j}| = 0 \qquad \forall i \neq j \tag{2.85}$$

$$\bigcup_{\forall i} I_{B,i} = I_{\text{ges}} \tag{2.86}$$

Der Zweck der Konnektoren $\mathscr{M}_{C,i}$ besteht in der Verknüpfung der Komponentenmodelle unter Einhaltung von Erhaltungssätzen. So kann das Prinzip der *Energieerhaltung* in den Konnektoren dadurch gewährleistet werden, dass das Integral über alle Flussgrößen eines Konnektors für beliebige Zeitintervalle 0 ist.[6] Das Verhalten der Konnektoren ist dadurch vorgegeben: Alle Potentialgrößen, die in $I_{C,i}$ liegen, müssen gleich sein. Die Summe der Werte der Flussgrößen in $I_{C,i}$ muss 0 ergeben und alle Zustandsgrößen dürfen nur an einen Konnektor angeschlossen werden.

$$\mathscr{M}_{C,i} \subseteq \mathscr{Z}_{I_{C,i}} \tag{2.87}$$

$$\mathbf{z}_{2k-1} = \mathbf{z}_{2j-1} \qquad \forall (2k-1), (2j-1) \in I_{C,i} \tag{2.88}$$

$$\sum_{\forall 2k \in I_{C,i}} \mathbf{z}_{2k} = 0 \tag{2.89}$$

$$(2k-1) \in I_{C,i} \Leftrightarrow (2k) \in I_{C,i} \qquad \forall i, k \tag{2.90}$$

$$|I_{C,i} \cap I_{C,j}| = 0 \qquad \forall i \neq j \tag{2.91}$$

$$\bigcup_{\forall i} I_{B,i} = I_{\text{ges}} \tag{2.92}$$

[6] Alternativ könnte auch eine Verallgemeinerung des Maschenstromverfahrens zur Sicherstellung der Erhaltungssätze verwendet werden.

Randbedingungen und Stimuli sind in konservativen Modellen in der Menge der Komponentenmodelle enthalten:

$$\mathscr{S} \in \mathscr{B} \tag{2.93}$$

Das Verhalten des Modells wird durch $\mathscr{V}$ bestimmt, mit

$$\mathscr{V} = \left(\bigcap_{\forall i} \mathscr{M}_{B,i} \right) \cap \left(\bigcap_{\forall i} \mathscr{M}_{C,i} \right) \tag{2.94}$$

Bezeichnet man die Anzahl der Zustandsgrößen in $\mathbf{Z}$ mit N, so werden durch die Menge der Konnektoren $\frac{N}{2}$ Gleichungen implizit spezifiziert (s. Gl. (2.88), Gl. (2.89)). Damit das konservative Modell deterministisch wird, müssen durch die Komponentenmodellen jeweils $\frac{|I_{B,i}|}{2}$ Gleichungen beschrieben werden.

2.3.4 Modelle der δ-Klasse

Modelle der δ-Klasse unterscheiden sich von den Modellen der anderen Modellklassen durch die Verwendung von ortsabhängigen Zustandsgrößen. Die Modelle müssen eine eindeutige Abbildung $\mathscr{S} \rightarrow \mathscr{O}$ beschreiben; alle Zustandsgrößen $\mathbf{z}_i \in \mathbf{Z} \in \mathscr{V}$ müssen funktionale Zustandsgrößen sein. Für jede Zustandsgröße $\mathbf{z}_i$ gilt:

$$\mathscr{U}_{\mathbf{z}_i} \subseteq \mathbb{R} \tag{2.95}$$

$$\mathscr{T}_{\mathbf{z}_i} \subseteq \mathbb{R} \qquad \mathbb{T} = \bigcup_{\forall i} \mathscr{T}_{\mathbf{z}_i} \tag{2.96}$$

$$\mathscr{X}_{\mathbf{z}_i} \in \mathbb{R}^n \qquad n \geq 1 \tag{2.97}$$

$$\mathbf{z}_i \in \mathscr{U}_{\mathbf{z}_i} \times \mathscr{T}_{\mathbf{z}_i} \times \mathscr{X}_{\mathbf{z}_i} \tag{2.98}$$

Die Menge $\mathscr{T}_{\mathbf{z}_i}$ und $\mathscr{X}_{\mathbf{z}_i}$ müssen stückweise kontinuierliche Mengen sein. Die Zustandsgrößen müssen jeweils zeitkontinuierlich und stückweise stetig sein und dürfen ortsabhängig sein.

Wie bei den Modellen der γ-Klasse unterscheidet man in strukturvariante und strukturinvariante Modelle gemäß Def. 2.48.

Wie bei den Modellen der β-Klasse und der γ-Klasse existiert eine globale Zeit im Sinne von Def. 2.17.

Mit $\mathbf{z}_i(t) = \emptyset$ für alle $t \notin \mathscr{T}_{\mathbf{z}_i}$ kann - wie für die Modelle der γ-Klasse- für das Tupel aus den Zustandsgrößen $\mathbf{Z}$ eine zeitliche Ordnung definiert werden:

Definition 2.49:

$$\mathbf{Z}(t_1) \preccurlyeq \mathbf{Z}(t_2) \quad \text{falls } t_1 \leq t_2 \qquad \forall t_1, t_2 \in \mathbb{T} \tag{2.99}$$

Der Zustandsraum $\mathscr{Z}$ eines Modells der δ-Klasse ist somit *vollständig zeitlich geordnet* und alle Modelle der δ-Klasse sind deshalb ebenfalls vollständig zeitlich geordnet. Es gilt zusätzlich die Cantor-Metrik aus Def. 2.43 und die Definition von Zeitkausalität aus Def. 2.44.

Definition 2.50: Modelle der δ-Klasse müssen zeitkausal sein.

Die δ-Klasse kann zur Beschreibung von Komponenten genutzt werden, für deren Modellierung partielle Differentialgleichungen benötigt werden. Potentialgrößen können über eine Zustandsgröße ($\mathbf{z}$) modelliert werden. Flussgrößen können durch ein Tupel $\mathbf{Z}_I$ aus Zustandsgrößen modelliert werden.

Mit Hilfe der Menge $\mathscr{S}$ kann der Anfangszustand sowie externe Stimuli definiert werden. Die Menge $\mathscr{M}$ ist die Lösungsmenge der partiellen Differentialgleichungen, die durch Zustandsübergangsrelationen repräsentiert werden.

In einer Implementierung wird zur Berechnung des Systemzustands der Zustandsraum durch den Simulator diskretisiert. Das entstehende diskrete System wird für die folgende *Simulation* verwendet. Die Berechnung des Systemzustands zu chronologisch geordneten Zeitpunkten wird Simulation genannt. Die Diskretisierung (Meshing) wird getrennt von der Modellbeschreibung vorgenommen. Dabei werden die Diskretierungspunkte (halb-) automatisch gesetzt, wobei der Abstand zwischen den Punkten durch eine benutzerdefinierte Metrik beeinflusst werden kann.

2.3.5 Simulationseigenschaften der Modellklassen

Tab. 2.1 fasst die charakteristischen Eigenschaften der Modellklassen zusammen. Der Simulationsaufwand für Modelle der δ-Klasse ist sehr hoch; die Abschätzung des Quantisierungsfehlers ist schwierig und vom Simulationssystem abhängig. Dafür können Problemstellungen mit räumlicher Ausdehnung und starken Wechselwirkungen, wie z. B. Feldberechnungen, einfach beschrieben werden.

Durch die raumdiskrete Modellierung ist der Simulationsaufwand bei Modellen der γ-Klasse im Vergleich zu den Modellen der δ-Klasse geringer. Wie bei diesen Modellen versteht man unter Simulation die Berechnung des Systemzustands zu chronologisch geordneten Zeitpunkten. Die Wahl der Zeitpunkte, für die der Systemzustand berechnet wird, ist nicht Bestandteil der Modellbeschreibung. Diese Zeitpunkte werden automatisch gewählt, wobei eine endliche Anzahl von Zeitpunkten vorgegeben werden kann.

Im Gegensatz zu Modellen der γ-Klasse sind die Zustandsgrößen der β-Klasse gerichtete Größen. *'Bidirektionale Wechselwirkungen'* sind dadurch nicht modellierbar, dafür können deutlich effizientere Simulationsalgorithmen eingesetzt werden als in der γ-Klasse oder der δ-Klasse.

Durch den Verzicht auf eine globale Zeit ist die Simulation von Modellen der α-Klasse noch effizienter möglich als bei Modellen der β-Klasse. Die Zustandsgrößen sind wert-

	Zustandsgrößen	Modelle
α-Klasse	funktional nicht ortsabhängig wertdiskret	sequenzkausal o. streng zeitkausal partiell geordnet
β-Klasse	funktional nicht ortsabhängig ereignisdiskret o. zeitkontinuierlich	streng zeitkausal globale Zeit vollständig zeitl. geordnet
γ-Klasse	funktional nicht ortsabhängig zeitkontinuierlich stückweise stetig	zeitkausal globale Zeit vollständig zeitl. geordnet
δ-Klasse	funktional z.T. ortsabhängig zeitkontinuierlich stückweise stetig	zeitkausal globale Zeit vollständig zeitl. geordnet

Tabelle 2.1: Zusammenfassung der charakteristischen Eigenschaften der Modellklassen.

und ereignisdiskret und lediglich partiell geordnet. Während einer Simulation wird die Abfolge der Ereignisse bestimmt. Ein zeitlicher Abstand zwischen zwei Ereignissen wird nicht festgelegt.

2.4 Erzeugung einer kohärenten zeitlichen Ordnung

Bei der Zeitbereichssimulation wird im Simulationssystem eine kohärente zeitliche Ordnung von Systemzuständen berechnet. Je nach Abstraktionsebene bieten sich dafür unterschiedliche Ansätze an. Für eine Mixed-Level Simulation ist jedoch eine gemeinsame globale Synchronisation notwendig, um Daten zum passenden Zeitpunkt auszutauschen. Im Folgenden werden für die vorgestellten Modellklassen adäquate Zeitmodelle vorgestellt. Um einerseits die Unterschiede zu verdeutlichen und andererseits, um eine problemspezifische Wahl der globalen Synchronisationsmethode zu motivieren.

Wie in Kapitel 2.3 definiert, wird in Modellen der α-Klasse keine globale Zeitfunktion verwendet. Stattdessen basiert die Simulation auf Folgenindizes oder einer abzählbaren Menge von *Tags*.

Es muss ein Konzept zur Realisierung der Eigenschaft der partiellen zeitlichen Ordnung gefunden werden, das sich im Allgemeinen nicht mit einer eindimensionalen Zeitgröße definieren lässt.

M. Raynal und M. Singhal [76] haben sich mit der Implementierung von *Tags* für die Simulation verteilter Systeme befasst. Die von ihnen betrachteten Systeme bestehen aus einer Menge von asynchronen Prozessen, die durch FIFO-Nachrichtenkanäle verbunden sind. Die Prozesse können sich ausschließlich über Nachrichten austauschen, wobei die Prozessausführung und die Nachrichtenübertragung asynchron erfolgen: Die Prozessausführung kann zu einem beliebigen Zeitpunkt durchgeführt werden und ein Prozess muss nach dem Versenden einer Nachricht nicht warten,bis die Nachricht empfangen wurde. Dieses verteilte System gehört zur Klasse der Tag-orientierten α-Klasse, wobei zusätzlich eine Zuordnung von Zustandsgrößen zu Prozessen durchgeführt wird.[7] Die Zustandsgrößen des Prozesses n seien im Tupel $\mathbf{Z}_{I_n}$ zusammengefasst, somit sind jeweils die Ereignisse der Zustandsgrößen aus $\mathbf{Z}_{I_n}$ untereinander vollständig geordnet, während eine Menge von Ereignissen aus $\mathbf{Z}_{I_{ges}}$ des Gesamtmodells eine partiell geordnete Menge sein kann. Im Folgenden wird gezeigt, dass in einem Beispiel mit k Prozessen, die *Tags* t_α der Zustände durch einen k-dimensionalen Vektor aus nicht negativen Integer-Zahlen gebildet werden können. Die i-te Komponente des Vektors t_α wird mit $t_{\alpha\,(i)}$ bezeichnet.

$$\text{Es sei } t_\alpha := (t_{\alpha\,(i)} | i \in [1..k]) \qquad t_{\alpha\,(i)} \in \mathbb{N}_0^+ \tag{2.100}$$

In den Zustandsgrößen aus $\mathbf{Z}_{I_n}$ beschreibt die n-te Komponente des *Tags* einen lokalen Zeitstempel, der die Ereignisse dieser Zustandsgrößen vollständig zeitlich ordnet. Die anderen Komponenten werden bei der Verarbeitung von Ereignissen der Zustandsgrößen aus anderen Prozessen bestimmt.

Die Erzeugung neuer Ereignisse kann zwei Ursachen haben: Zum einen die Verarbeitung von Nachrichten (Ereignisse) anderer Prozesse, zum anderen der darauf folgende prozessinterne Ablauf. In beiden Fällen muss für das neue Ereignis ein *Tag* berechnet werden:

Der aktuelle *Tag* des Prozesses n, d. h. der *Tag* des letzten in diesem Prozess erzeugten Ereignisses, wird mit t_{P_n} bezeichnet. Bei der Verarbeitung des Ereignisses z_1 wird der *Tag* $\tilde{t}_{P_n}$ für die daraus resultierenden Ereignisse wie folgt berechnet:

$$t(z_1) = (t(z_1)_{(i)} | i \in [1..k])$$

$$\tilde{t}_{P_n} := (\tilde{t}_{P_n\,(i)} | i \in [1..k]) \tag{2.101}$$

$$\tilde{t}_{P_n\,(i)} := \max(t_{P_n\,(i)}, t(z_1)_{(i)}) \qquad \forall i \neq n \tag{2.102}$$

$$\tilde{t}_{P_n\,(n)} := t_{P_n\,(n)} + 1 \tag{2.103}$$

[7] In einem Modell, bei dem diese Zuordnung nicht gegeben ist, kann diese erzwungen werden, in dem für jede Zustandsgröße ein Prozess deklariert wird.

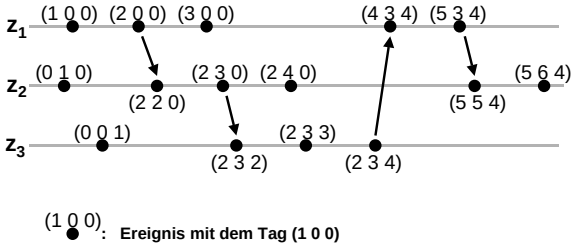


Abbildung 2.1: Beispiel für die Entwicklung der *Tags* in einem Modell der α-Klasse

Werden in einem Prozess eigenständig Ereignisse erzeugt, so vereinfacht sich die Berechnung des *Tags* für diese Ereignisse:

$$\tilde{t}_{P_n} := (\tilde{t}_{P_n(i)} | i \in [1..k]) \tag{2.104}$$

$$\tilde{t}_{P_n(i)} := t_{P_n(i)} \qquad \forall i \neq n \tag{2.105}$$

$$\tilde{t}_{P_n(n)} := t_{P_n(n)} + 1 \tag{2.106}$$

Abb. 2.1 zeigt an einem Beispiel mit drei Zustandsgrößen die *Tags* einiger Ereignisse. Die Zustandsgrößen sind jeweils eigenen Prozessen zugeordnet. Die Pfeile deuten zeitkausale Abhängigkeiten zwischen Ereignissen an. Die *Tags* sind mit obigen Formeln berechnet. Für diese *Tags* müssen nun die Relationen $\leq$ und $<$ definiert werden:

Definition 2.51: Für zwei *Tags* $t_\alpha, \tilde{t}_\alpha$ gilt $t_\alpha \leq \tilde{t}_\alpha$, genau dann wenn:

$$t_{\alpha(i)} \leq \tilde{t}_{\alpha(i)} \quad \forall i \tag{2.107}$$

Definition 2.52: Für zwei *Tags* $t_\alpha, \tilde{t}_\alpha$ gilt $t_\alpha < \tilde{t}_\alpha$, genau dann wenn:

$$t_\alpha \leq \tilde{t}_\alpha \quad \text{und ein i existiert, so dass} \quad t_{\alpha(i)} < \tilde{t}_{\alpha(i)} \tag{2.108}$$

Mit diesen Definitionen kann die partielle Ordnung in dem Beispiel aus Abb. 2.1 überprüft werden. So ist beispielsweise für das dritte Ereignis von $\mathbf{z}_1$ und das dritte Ereignis von $\mathbf{z}_2$ keine zeitliche Ordnung definiert, während das dritte Ereignis von $\mathbf{z}_3$ *nach* dem dritten Ereignis von $\mathbf{z}_2$ stattfindet:

$$(300) \not\leq (230) \quad \text{und} \quad (230) \not\leq (300)$$
$$(230) < (233)$$

Die Ergebnisse lassen sich auf alle Modelle übertragen, die als Modelle der α-Klasse interpretiert werden können.

Implementiert man die *Tags* auf heutigen 32-Bit Computersystemen als Vektoren mit *'natural integers'*, so können mindestens $2^{31} - 1$, also über 2 Mrd. Simulationszyklen zeitlich geordnet verwaltet werden - eine für die meisten Zwecke großzügig dimensionierte Menge.

Bei den *ereignisdiskreten Modellen* sind die Zeitpunkte, an denen sich der Systemzustand ändert, ebenfalls abzählbar. Dadurch ist auch bei ereignisdiskreten Modellen der β-Klasse eine *vollständige*, korrekte Protokollierung des Systemzustands möglich. Im Gegensatz zu den Modellen der α-Klasse sind die Modelle der β-Klasse jedoch vollständig zeitlich geordnet. Zusätzlich zur zeitlichen Ordnung ist eine Protokollierung der Zeitverzögerungen notwendig, so dass die *Tags* nun echte Zeitstempel werden.

Anhand des Zeitstempels (t_{ev}) werden die Momentaufnahmen des Systemzustandes ($\mathbf{Z}(t_{ev})$) unterschieden. Dabei muss für jede Teilmenge

$$\mathscr{Z}_{1-n} = \{\mathbf{Z}(t_{ev,1}), \mathbf{Z}(t_{ev,2}), \ldots \mathbf{Z}(t_{ev,n})\}$$

eine eindeutige zeitliche Abfolge anhand des Zeitstempels feststellbar sein. Zusätzlich muss der Zeitstempel für jede beliebige Vereinigung zweier Teilmengen $\mathscr{Z}_{k_1-k_3}, \mathscr{Z}_{k_2-k_4}$ eine eindeutige zeitliche Ordnung erzeugen. Deshalb sollte der Zeitstempel ein reeles Zeitmaß sein, damit beliebig schnell aufeinanderfolgende Ereignisse zeitlich geordnet werden können. Da der Zeitstempel für die EDV gestützte Verarbeitung jedoch in einem diskretisierten Zahlenformat dargestellt werden muss, wird eine geeignete Implementierung gesucht, welche die Eigenschaften der ereignisdiskreten Modelle der β-Klasse erhält.

Die im IEEE Standard 1076 (VHDL) gewählte Implementierung basiert auf der Definition eines minimalen Zeitschritts - der Zeitstempel ist ein natürliches Vielfaches dieses Inkrements. Das Modellierungskonzept wird hiermit durch eine zusätzliche Randbedingung ergänzt: Der zeitliche Abstand zwischen zwei Ereignissen muss ein ganzzahliges Vielfaches des minimalen Zeitschritts (Δ_{min}) sein:

$$t_{ev,n} = k_n \cdot \Delta_{min} \tag{2.109}$$

Dies ermöglicht eine Vereinigung des Zeitstempels mit dem Folgenindex der Ereignisverwaltung. Der minimale Zeitschritt kann abhängig von den zu simulierenden Modellen bestimmt werden, wobei sicherzustellen ist, dass die streng kausale zeitliche Ordnung der Systemzustandsaufnahmen erhalten bleibt.

Es ist auch möglich den Zeitstempel als Fließkommazahl $t_{\mathbb{R}}$ zu definieren. Dies verursacht jedoch unerwünschte Nebeneffekte für eine Reihe typischer ereignisdiskreter Modelle wie im Folgenden gezeigt wird:

Synchrone digitale Schaltungen sind ein Hauptanwendungsgebiet für die ereignisdiskrete Simulation. Diese Schaltungen zeichnen sich durch periodisch wiederkehrende Ereignisse aus (direkt - oder indirekt - basierend auf einem periodischen Taktsignal). In der Regel werden Fließkomazahlenformate mit fester Mantissenbreite (p) definiert, wodurch sich bei wachsender Simulationszeit das kleinste darstellbare Zeitinkrement (Δ_{min}) ändert:

$$t_{\mathbb{R}} = (-1)^s \cdot f \cdot 2^e \qquad f \in [1.0; 2.0 - 2.0^{-p}] \qquad e \in [-E_{max}+1; E_{max}] \tag{2.110}$$

$$\Delta_{min}(t_{\mathbb{R},n}) = t_{\mathbb{R},m} - t_{\mathbb{R},n} \qquad \forall t_{\mathbb{R},m} > t_{\mathbb{R},n} \wedge t_{\mathbb{R},m} \rightarrow t_{\mathbb{R},n} \tag{2.111}$$

$$\rightsquigarrow \Delta_{min}(t_{\mathbb{R},n}) = 2^{-p} \cdot t_{\mathbb{R},n} \tag{2.112}$$

$$\rightsquigarrow \Delta_{min}(t_{\mathbb{R},i}) = \frac{1}{2}\Delta_{min}(t_{\mathbb{R},j}) \qquad \forall t_{\mathbb{R},j} = 2 \cdot t_{\mathbb{R},i} \tag{2.113}$$

Durch diese sich ändernde zeitliche Auflösung wird das Ereignis-Scheduling beeinflusst, eine im Allgemeinen unerwünschte Eigenschaft.

Bei den *zeitkontinuierlichen Modellen* kann der Systemzustand prinzipbedingt nicht vollständig erfasst werden. Das gilt sowohl für die Modelle der γ-Klasse und der δ-Klasse, als auch für die *streng zeitkausalen* zeitkontinuierlichen Modelle der β-Klasse.

Berücksichtigt man, dass viele etablierte Simulationssysteme für zeitkontinierliche Modelle Berechnungsverfahren mit automatischer Schrittweitenadaption verwenden, so wird eine Implementierung des Zeitstempels mit Fließkommazahlen attraktiv. Sie ermöglicht es relative Fehler in der Berechnung des Systemzustands auf Iterationsintervalle, an Stelle von absoluten Zeitintervallen, zu begrenzen. Somit wird der Simulator unabhängig von den Zeitkonstanten eines Modells - lediglich die Steifheit der daraus abgeleiteten Matrizen beeinflusst den Simulationsfehler.

Um eine Mixed-Level Simulation zu ermöglichen müssen offensichtlich Schnittstellen zum Datenaustausch generiert werden. Andererseits muss eine gemeinsame kohärente zeitliche Ordnung erzeugt werden, deren Konzeption im Folgenden erläutert wird.

Ein Konzept zur Realisierung der Mixed-Level Simulation ist die *Master-/Slave Konfiguration* bei der der/die Slave(s) im Einzelschrittmodus ausgeführt wird.

Eine Variante dieses Konzeptes wird im IEEE Standard 1076.1 (VHDL-AMS) zur Simulation von zeitkontinuierlichen Modellen mit ereignisdiskreten Modellen beschrieben. In dieser Anwendung (siehe Abb. 2.2) werden die ereignisdiskreten Modelle im Einzelschrittmodus abgearbeitet und die zeitkontinierlichen Modelle werden quasi parallel dazu verarbeitet.

In VHDL-AMS ist für den 'analogen Solver' ein Fließkommazahlenformat für den Zeitstempel gewählt. Die Mindestanforderung schreibt dabei das IEEE-Fließkommazahlenformat 'single precision' (SP) mit einer Mantissenbreite von 23 Bit vor. Der 'digitale

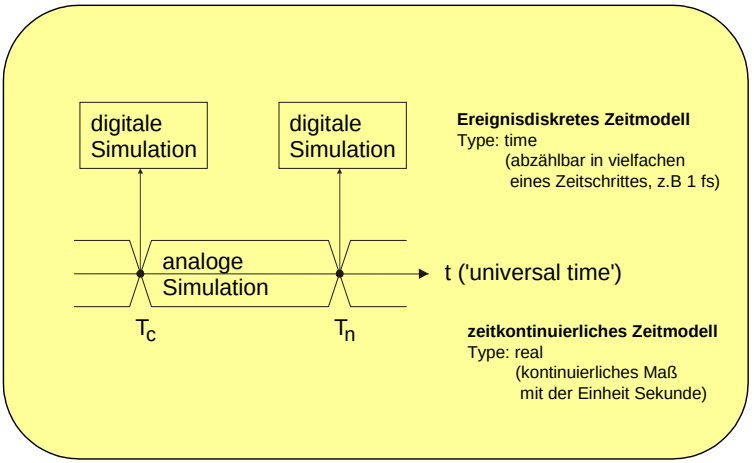


Abbildung 2.2: Zeitliche Abfolge einer VHDL-AMS Simulation

Solver' nutzt eine Kombination aus einem minimalen Zeitschritt, z.B. '1 ps' und einem Multiplikator aus dem Bereich der natürlichen Zahlen. Die Mindestanforderung für diesen Zeitstempel ist ein Integer mit 31 Bit. Da bei der Simulation von Mixed-Signal Schaltungen eine eindeutige kausale Ordnung der Ereignisse erhalten bleiben soll, muss der Konvertierungsfehler bei einer Konvertierung der 'digitalen Zeit' in das analoge Format und zurück klein genug sein. Die Obergrenze für solche Konvertierungsfehler wird durch den Abstand zweier aufeinanderfolgender Ereignisse vorgegeben. Wenn die Zeitstempel zweier streng kausal geordneter Ereignisse im analogen Zeitformat nicht mehr unterschieden werden können, ist die maximale Simulationsdauer überschritten. Somit ist die Simulationsdauer im ungünstigsten Fall auf 2^{23} Zeiteinheiten beschränkt:

Definition 2.53: Es sei $r_{i+\varepsilon}$ die kleinste darstellbare Zahl, die größer ist als r_i.

Die kleinste positive Zahl des SP-Datenformats[8] nach IEEE Norm 754-1985 ist

$$\begin{aligned} r_1 &= 1 \cdot 2^{-23} \cdot 2^{-126} = 2^{-149} \\ \rightsquigarrow r_{1+\varepsilon} &= 2 \cdot 2^{-23} \cdot 2^{-126} = 2^{-148} \end{aligned} \tag{2.114}$$

Damit ergibt sich ein minimales Inkrement von

$$\Delta_{min}(r_1) = 2^{-149} \tag{2.115}$$

[8] SP: single precision

Dieses minimale Inkrement gilt für alle denormalisierten Zahlen. Für nicht denormalisierte Zahlen ist das minimale Inkrement, wie in Gl. (2.113) gezeigt, variabel.

$$r_j = 2 \cdot r_i \rightsquigarrow \Delta_{min}(r_j) = 2 \cdot \Delta_{min}(r_i) \tag{2.116}$$

Anwendungsbeispiel:

Die Zeitbasis der zeitkontinuierlichen Zeitdarstellung sei $1s$. Dann ist für SP-Datentypen am Anfang der Simulation $\Delta_{min}(0s) = 2^{-149} \approx 1,4 \cdot 10^{-30} fs$. Für den Simulationszeitpunkt $1s$ gilt jedoch $\Delta_{min}(1s) = 2^{-23} \approx 119ns$. Im double precision (DP) Datenformat ergibt ist $\Delta_{min}(1s) = 2^{-52} \approx 0,222 fs$. Unter der Annahme, dass eine Simulation eines gemischt analog/digitalen Systems für eine Sekunde benötigt wird, ist der im VHDL-AMS Standard geforderte SP-Datentyp ungeeignet, da moderne CMOS Technologien Schaltzeiten haben, die wesentlich kleiner als 119 ns sind.

Im Umkehrschluss kann ebenso die maximale Simulationsdauer bestimmt werden. Die Schaltzeitpunkte der digitalen Logik sollen mit einem Fehler kleiner 1% protokolliert werden, die verwendete CMOS Technologie habe eine Durchschaltverzögerung von $1ns$, das benötigte minimale Zeitinkrement ist somit $\Delta_{min}(t_{max}) = 0,01ns$.

Daraus ergibt sich für das SP-Datenformat ein t_{max} von $83,9\mu s$; für Fließkommazahlen im DP-Datenformat ist $t_{max} = 750min$.

Ebenso kann die analoge Teilschaltung die Simulationsdauer beschränken. Unter der Annahme, dass zur Sicherstellung der Konvergenz des verwendeten Integrationsverfahrens das Zeitinkrement im Verlauf der Simulation mehrfach auf 0,01 ns reduziert werden muss, ergibt sich für das SP-Datenformat eine maximale Simulationsdauer von $83,9\mu s$, das sind lediglich 839 Takte einer synchronen Teilschaltung mit 10 MHz Taktsignal.

Ein Ansatz zur Umgehung dieser Problematik ist ein anderes Zeitmodell, z. B. ein Fließkommazahlenformat, dessen Bezugspunkt die *simulative Gegenwart* darstellt. In einer solchen Zeitordnung ist das minimale Zeitinkrement stets gleich und lediglich für weit in der Vergangenheit liegende Zustandswerte entsteht ein zu obiger Problematik äquivalentes zeitliches Auflösungsproblem.

Ein anderer Ansatz, der auch eine Modellbildung in VHDL-AMS Simulationssystemen ermöglicht, liegt in der *Modellabstraktion*. Hierbei wird für kritische Komponenten ein Ersatzmodell erstellt, für dessen Berechnung größere Zeitinkremente verwendet werden können. Wenn man eine verminderte Genauigkeit oder einen eingeschränkten Arbeitsbereich in Kauf nimmt, kann ein solches Ersatzmodell ebenso die Simulationsgeschwindigkeit erhöhen, da im Simulator aufgrund der größeren Schrittweite weniger Berechnungsaufwand entsteht. Der Ansatz der vereinfachten Ersatzmodelle führt nun wieder zurück zu der Idee der *Gesamtsystemsimulation mit High-Level Beschreibungen der Systembestandteilen*.

Kapitel 3

Modellierungskonzepte

Bei der Entwicklung heterogener Systeme werden vielfach Verhaltensmodelle eingesetzt, die Teilkomponenten des Systems durch heuristische Näherungen beschreiben. Für die Klassifizierung dieser Verhaltensmodelle wurden Modellklassen vorgestellt, die sich in der Ausdrucksmächtigkeit und dem Zeitmodell unterscheiden.

Modelle der α-Klasse benötigen eine Infrastruktur, die die Aktivierung der Prozesse und den diskreten Informationsaustausch übernimmt. Für Modelle der β-Klasse müssen zusätzlich eine globale Zeit und Vorgehensweisen zur Festlegung der zeitlichen Abfolge (Schedulingmechanismen) zur Verfügung stehen. Modelle der γ-Klasse benötigen einen Simulationskern, der differential-algebraische Gleichungssysteme lösen kann. Die Modelle der δ-Klasse müssen vor der Simulation zusätzlich räumlich diskretisiert werden.

Diese Modellklassen wurden im vorherigen Kapitel durch Einschränkungen in der Ausdrucksmächtigkeit definiert. In den folgenden Abschnitten werden etablierte domänenspezifische Modellierungskonzepte und deren Einbettung in das domänenunabhängige System aus Modellklassen vorgestellt. Einige Modellierungskonzepte sind durch den Grad der Modellabstraktion voneinander abgegrenzt, andere sind im Hinblick auf das Entwicklungsstadium des Entwurfsobjekts geordnet. Meistens haben sich Mischformen etabliert, bei denen aus dem Entwicklungsstadium und der Ausprägungen der Beschreibungsmöglichkeiten ein System aus Abstraktionsebenen definiert wurde.

Aus den domänenspezifischen Modellierungskonzepten werden für den Entwurf heterogener Systeme geeignete Abstraktionebenen hergeleitet und, soweit sinnvoll, entsprechend der üblichen Nomenklatur benannt. Abschließend wird die für die praktischen Beispiele verwendete Umsetzung der Modellierungskonzepte in die Modellierungssprachen VHDL-AMS, SPICE und C erläutert.

3.1 Domänenspezifische Modellierungskonzepte

3.1.1 Modellbildung zur Simulation und Synthese digitaler Schaltungen

Der Entwicklungsprozess der meisten digitalen Schaltungen basiert heutzutage auf textuellen Beschreibungen des Schaltungsverhaltens in VHDL oder Verilog. Diese Modelle werden mit Synthese-Programmen weiterverarbeitet, um eine Netzliste aus einfachen

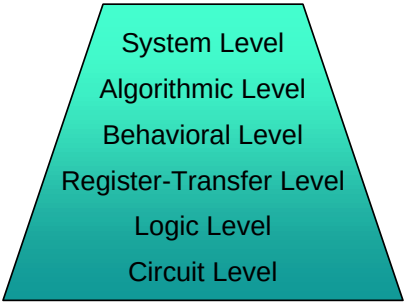


Abbildung 3.1: Abstraktionsebenen für den Entwurf digitaler Schaltungen in der etablierten englischen Nomenklatur.

Schaltungsbausteinen zu erzeugen, die dann für das Layout weiterverwendet werden kann. Passend zu diesem Entwicklungsablauf wurden Abstraktionsebenen definiert, die nun vorgestellt werden.

Die Entwicklung der Grundelemente, aus denen die Synthese-Bibliotheken gebildet werden, ist abhängig von der Prozesstechnologie. Im Standardzellenentwurf und den daran angelehnten Entwurfsabläufen werden diese Grundelemente in die Synthese-Bibliotheken mit ihrem Zeitverhalten, einigen elektrischen Eigenschaften und ihrem Platzbedarf aufgenommen. Durch diese Elemente werden grundlegende kombinatorische Logik-Funktionen und Speicher-Elemente implementiert. Bei der Erstellung der Elemente wird der Aufbau der Grundelemente aus der in der Halbleitertechnologie verfügbaren physikalischen Elementen (z. B. Transistoren, Dioden, Widerständen oder Kondensatoren) festgelegt. Zusätzlich werden die physikalischen Elemente dimensioniert, mit festen Abständen zueinander plaziert und verdrahtet. Die Entwicklung dieser Bauelemente findet auf der Abstraktionsebene mit dem Namen *Circuit Level* (siehe [2]) oder *Transistor Level* (z. B. [23]) statt. Die Ergebnisse dieser Arbeit bilden die Grundelemente für die Abstraktionsebene *Logic Level*.

Auf der Abstraktionsebene *Logic Level* (auch *Gate Level* genannt), werden digitale Schaltungen durch boolesche Gleichungen und binäre Datenspeicher modelliert. Die Modellbildung kann auch durch eine Strukturbeschreibung, d. h. eine Netzliste mit kombinatorischen Zellen und Flip-Flops, gebildet werden (siehe Abb. 3.2). Diese Abstraktionsebene wird heutzutage hauptsächlich zum Datenaustausch von Logik-Synthese-Programmen zu Layoutprogrammen genutzt. Die meisten Synthese-Programme können *Logic Level* Beschreibungen in VHDL oder Verilog exportieren, so dass eine ereignisbasierte Simulation dieser Modelle möglich ist. Für die Weiterverarbeitung in Layout-Programmen genügen jedoch weniger ausdrucksstarke Beschreibungssprachen. Ein Beispiel für eine gut dokumentierte und weitverbreitete Beschreibungssprache zum

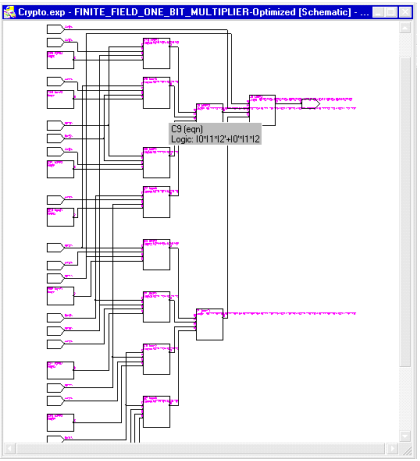


Abbildung 3.2: Ausschnitt aus einem *Logic Level* Strukturmodell für einen Xilinx FPGA. Die boolesche Logik (z. B. C9) kann jeweils in ein Grundelement (LUT) abgebildet werden.

Austausch von Daten zwischen Logik-Synthese- und Layout-Programmen ist der EDIF Sprachstandard.

Speziell an die Bedürfnisse von Entwurfs-Ingenieuren und Synthese-Programmen angepasste Abstraktionsebenen werden verwendet, um digitale Schaltungen in einem höheren Abstraktionsgrad zu modellieren. Aufgrund der Tatsache, dass die Synthesewerkzeuge stetig mächtiger werden, gibt es eine fortlaufende Erweiterung und Weiterentwicklung dieser Abstraktionsebenen.

Die Abstraktionebene *Register-Transfer Level* (RTL) ist mittlerweile eine gute Wahl für die Modellierung von digitalen Schaltungen, für die nicht festgelegt ist, mit welchem Synthese-Programm sie weiterverarbeitet werden. 1999 wurde der IEEE Standard 1076.6 [38] verabschiedet, der einen einheitlichen VHDL-Sprachumfang festlegt, den ein VHDL-Synthese-Programm erfüllen muss. Die Modelle der Abstraktionebene RTL sind ein wichtiger Vertreter ereignisdiskreter Modelle. Auf der Abstraktionsebene RTL wird mit Funktionsblöcken wie Registern, Speichern, ALUs und getakteter Kontrollogik modelliert (siehe Beschreibung 3.1). In den letzten 10 Jahren wurden einige Untermengen von RTL definiert und genutzt (*Logic Level* [22], *Gate Level* [64, 86]), aber aufgrund der wachsenden Schaltungsgrößen und der guten Unterstützung von RTL haben diese Untermengen keine praktische Bedeutung mehr.

Durch die Verfügbarkeit großer FPGAs und durch kürzer werdende Produktzyklen bei den Konsumgütern, werden zunehmend schnelle 'Algorithm-To-Chip' Entwurfsabläufe

```
-------------------------------------------------------------------------------
-- Title       : Multibit FIFO-Shift-Register
-------------------------------------------------------------------------------
-- Description: A multibit FIFO Shift Register with arbitrary
--              word sizes.
--              Default configuration:
--                Wordlength = 12
--                parallel Load loads 3 words
--                parallel 'Read' has access to 2 words
--              Usage: Load happens in every 3rd clockcycle,
--                     Read in every 2nd clockcycle
-------------------------------------------------------------------------------
entity datasplit is
  generic (
    DataInLength : integer := 36;
    DataOutLength : integer := 24;
    BufferSize : integer := 60;
    ShiftFactor : integer := 12);
  port (
    signal clock       : in std_logic;
    signal ClkEnable : in std_logic;
    signal LoadEnable : in std_logic;
    signal DataIn      : in std_logic_vector(DataInLength-1 downto 0);
    signal DataOut     : in std_logic_vector(DataOutLength-1 downto 0));
end entity datasplit;

architecture rtl of datasplit is
signal Buffer : std_logic_vector(BufferSize-1 downto 0);
begin  -- architecture rtl
  main: process (clock) is
    variable i : integer;
  begin  -- process main
    if clock'event and clock = '1' then  -- rising clock edge
      if ClkEnable = '1' then
        for i in 0 to (Buffer'High-DataIn'length) loop
          Buffer(i) <= Buffer(i+ShiftFactor);
        end loop;  -- i
        for i in Buffer'High-DataIn'High to Buffer'High loop
          if (LoadEnable = '1')  then
            Buffer(i) <= DataIn(DataIn'High-Buffer'High+i);
          elsif (i+ShiftFactor > Buffer'high) then
            Buffer(i) <= '0';
          else
            Buffer(i) <= Buffer(i+ShiftFactor);
          end if;
        end loop;  -- i
      end if;
    end if;
  end process main;
  DataOut <= Buffer(DataOut'range);
end architecture rtl;
```

Beschreibung 3.1: RTL Beschreibung eines Schieberegisters mit konfigurierbarer Wortbreite.

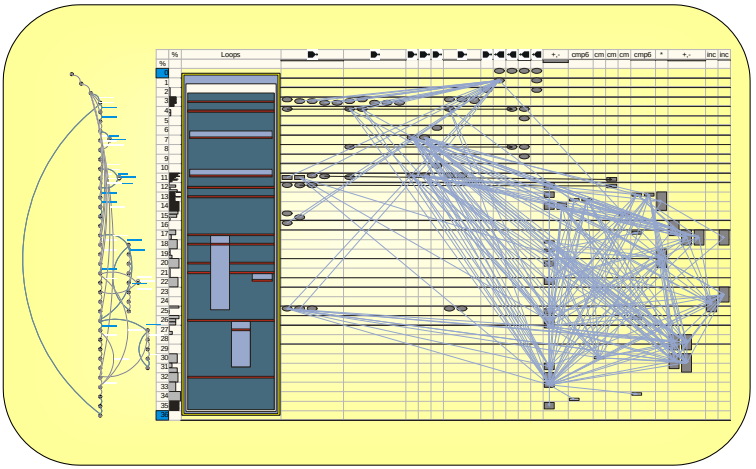


Abbildung 3.3: Ergebnis des *Resource Allocation and Scheduling* Algorithmus des Behavioral Compiler der Firma Synopsys: In den Spalten, in denen mehrfach Aktivität markiert ist, wird ein Funktionsblock mehrfach genutzt. In den Spalten rechts sind die *Multi-Cycle* Datenpfade zu sehen. In diesem Beispiel benötigen sie 2 Takte.

benötigt. Ebenso ist in manchen Anwendungsgebieten die Entwicklungszeit wichtiger als die Qualität[1] der synthetisierten Schaltung. Dementsprechend wurden in vielen Versuchen Synthese-Programme für Modelle höherer Abstraktionsebenen entwickelt, mit dem Nebeneffekt, dass sich eine Abstraktionsebene namens *Behavioral Level* gebildet hat, die jeweils dem aktuellen Stand der Entwicklung der dazugehörigen Synthese-Programmen entspricht. Ein grundlegender Unterschied zu RTL ist, dass das Synthese-Programm mehrfach benötigte Funktionen durch *Resource Allocation and Scheduling* Algorithmen auf mehrfach genutzte Komponenten abbilden kann (siehe Beschreibung B.2). Die Modellbildung auf *Behavioral Level* ist durch Restriktionen der Synthese-Programme eingeschränkt. Unabhängig von den Synthese-Programmen wurde der *Algorithmic Level* definiert, wobei zu erwarten ist, dass durch die Weiterentwicklung der Synthese-Programme die Begriffe *Behavioral Level* und *Algorithmic Level* gleichwertig, bzw. austauschbar werden.

Die Abstraktionsebene *Algorithmic Level* wurde im Hinblick auf Mehrtakt-Operationen (Multi-Cycle Operations) definiert. Während in RTL jeder Zustandsübergang auf einem Taktzyklus basiert, wird im *Algorithmic Level* ein Kausalitätsprinzip verwendet: Es wird angenommen, dass die für die Berechnung eines Algorithmus benötigte Anzahl

[1] Mit Qualität ist hierbei weniger die Funktionsfähigkeit des Endproduktes gemeint, sondern vielmehr Qualitätsmerkmale wie Stromverbrauch, Platzbedarf und Lebensdauer.

an Taktzyklen nicht entscheidend ist. Um eine Kommunikation zwischen verschiedenen Komponenten zu ermöglichen, werden Kommunikationsprotokolle verwendet. Das benötigte Synthese-Programm erledigt automatisch das Ressourcen-Management (Bereitstellung und Zuteilung), sowie die Synthese der Daten- und Kontrollpfade. Dabei können Datenpfade, deren Latenz mehrere Taktperioden lang ist, durch 'Pipelining' mehrfach genutzt werden (siehe Abb. 3.3). Um das Syntheseergebnis zu optimieren, werden zusätzliche Nebenbedingungen berücksichtigt. Durch die Definition von Latenz, Durchsatz oder maximalem Platzbedarf können aus einer gemeinsamen algorithmischen Schaltungsbeschreibung unterschiedliche, produktspezifisch optimierte Syntheseergebnisse erzeugt werden. Die algorithmischen Schaltungsbeschreibungen werden überschaubarer und leichter zu testen, da sie weniger Implementierungsdetails enthalten, und sie können öfter wiederverwendet werden als RTL-Beschreibungen. Andererseits können projektbezogen optimierte RTL-Beschreibungen deutlich kleiner und schneller sein.

Eine Abstraktionsebene mit dem Namen *System Level* ist vorgesehen, um Systemspezifikationen zu ermöglichen. Frühe Definitionen (z. B. [22]) sind sehr ähnlich zum *Algorithmic Level*. In den letzten Jahren hat sich eine neue Definition der Abstraktionsebene durchgesetzt [40], in der Anforderungen aus dem Hardware-/Software-Codesign berücksichtigt werden. *System Level* Beschreibungen basieren auf verteilten Kontrolleinheiten und 'Multi-Threaded' Berechnungen. Ein Modell besteht aus einer Menge hierarchischer, parallel aktiver, kommunizierender Prozesse. Der Zustand der Prozesse ändert sich während einer Simulation durch Kommunikationsvorgänge, wobei die Zeitdauer zwischen den Zustandsübergängen nicht betrachtet wird. Modelle dieser Abstraktionsebene sind (noch) nicht synthetisierbar.

3.1.1.1 Abbildung auf die Modellklassen des Meta-Modells

Für die Dimensionierung und Validierung der Modelle des *Circuit Level* werden zeitkontinuierliche Simulationssysteme benötigt. Durch die Beschreibung der Modelle mit Grundelementen wie Kondensatoren, Widerständen und Transistoren entstehen Modelle, die parametrisiert, jedoch nicht ortsabhängig sind. Diese Modelle müssen bei wiederholten Simulationen stets dasselbe Verhalten zeigen; es werden also deterministische Modelle benötigt. Wie man leicht sieht, sind für diese Aufgabe nur zeitkausale Modelle geeignet, obwohl durch die verwendeten Grundelemente keine streng zeitkausalen Modelle entstehen. Somit können nur Modelle der γ-Klasse für diese Abstraktionsebene verwendet werden. In der Regel sind die *Circuit Level* Beschreibungen ausschließlich strukturinvariante, konservative Modelle der γ-Klasse.

Die Grundelemente der Modelle des *Logic Level* haben feste Ein- und Ausgänge. Verhaltensbeschreibungen in Form von booleschen Gleichungen werden in expliziter Form angegeben. Es ist somit ein gerichteter Signalfluss vorhanden, der zu einem streng zeitkausalen Simulationsmodell führt. Die Zustandsgrößen sind ereignisdiskrete Signale

und nicht ortsabhängig. Die Simulationsmodelle können sowohl ereignisdiskrete Modelle der β-Klasse, als auch Modelle der α-Klasse sein. *Logic Level* Modelle, in denen auch Gatterlaufzeiten modelliert sind, gehören zu den Modellen der β-Klasse und müssen mit einer globalen Simulationszeit ausgeführt werden. Die Modelle, bei denen die Verzögerungszeiten in den kombinatorischen Zellen vernachlässigt werden, gehören zu den Tag-orientierten Modellen der α-Klasse.

Register Transfer Level Modelle sind streng zeitkausale, ereignisdiskrete Modelle mit nicht ortsabhängigen Signalen als Zustandsgrößen. Modelle dieser Abstraktionsebene sind typische Vertreter der ereignisdiskreten Modelle der β-Klasse.

Die Modelle des *Behavioral Level* können größtenteils als ereignisdiskrete β-Klassen Modelle betrachtet werden, obwohl sie bei der Synthese als Modelle der α-Klasse betrachtet werden müssen. In der Regel bestehen die in VHDL notierten Modelle des *Behavioral Level* aus mehreren sequentiellen Prozessen, in denen eine Datenverarbeitung in einer Endlosschleife stattfindet. Diese Datenverarbeitung kann mehrfach bis zur nächsten Taktflanke unterbrochen werden, so dass ein ereignisdiskretes Modell mit globaler Zeit entsteht. Bei der Synthese werden die sequentiellen Anweisungen in den Prozessen jedoch nicht zwingend in der im Modell angegebenen Reihenfolge implementiert. Das Synthese-Programm stellt lediglich sicher, dass die von außen beobachtbaren Signale zum richtigen Zeitpunkt gesetzt werden. Syntheseergebnis und die zugrundeliegende Modellbeschreibung haben somit nur ein partiell geordnetes Grundmodell gemeinsam - ein Tag-orientiertes α-Klassen Modell.

Die Beschreibungen des *Algorithmic Level* können durch Tag-orientierten Modelle der α-Klasse dargestellt werden. Die Zustandsgrößen sind ereignisdiskret und nicht ortsabhängig. Die Modelle sind streng zeitkausal, aber der Abstand zwischen zwei Ereignissen ist nicht festgelegt. Mit Hilfe von Kommunikationsprotokollen wird sowohl einen Datenaustausch zwischen Modellen des *Algorithmic Level*, als auch zu Modellen der β-Klasse, ermöglicht.

Für die Abstraktionsebene *System Level* eignen sich ebenfalls Modelle der α-Klasse. Sie ermöglichen mit den Sequenz-orientierten Modellen Beschreibungen in der Form von Kahn's Prozess Netzwerken [70] oder mit den Tag-orientierten Modellen Beschreibungen wie CSP [30] oder CSS [70]. Eine tiefergehende Diskussion solcher *System Level* Beschreibungen ist in [60] zu finden.

3.1.2 Modellbildung zur Simulation analoger Schaltungen

Der Entwurf analoger Schaltungen ist meistens ein *Bottom-Up* Ansatz und sehr arbeitsintensiv. Um sicherzustellen, dass sowohl Signale mit kleinen Signalamplituden[2], als

[2] Die zu verarbeitenden Ströme in einem analogen CMOS-IC, können um viele Zehnerpotenzen verschieden sein. Das Nutzsignal eines photoelektrischen CMOS-Sensors beträgt wenige pA, während die in einigen ICs integrierte Stromquelle für eine externe LED mehrere mA Dauerstrom liefern muss. (mA/pA = 10^9)

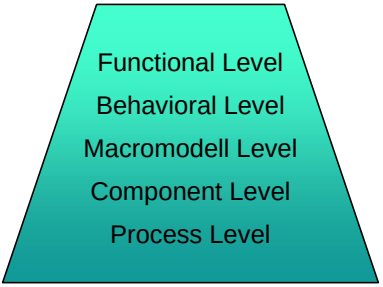


Abbildung 3.4: Abstraktionsebenen für den Entwurf analoger Schaltungen

auch schnelle transiente Signale untersucht werden können, wird i. d. R. ein zeitkontinuierliches Simulationsparadigma genutzt. Der Simulationskern benötigt ausgefeilte Algorithmen zur Bestimmung der Berechungszeitpunkte. Durch variable Zeitschrittweiten und eine Rollback-Funktion bei Konvergenz-Problemen[3] kann trotz strenger Genauigkeitsanforderungen eine akzeptable Rechenzeit erreicht werden. Simulationsmodelle einer hohen Abstraktionsebene können die Rechenzeit um mehrere Größenordnungen reduzieren. Diese Modelle erzeugen jedoch i. d. R. mehr Konvergenzprobleme als hardwarenah beschriebene Modelle. Die Vorschläge zur Unterteilung des Entwurfsraums in Abstraktionsebenen basieren auf den zur Verfügung stehenden Simulationssystemen und unterscheiden sich beispielsweise aufgrund von implementierungsabhängigen Randbedingungen. Die Definitionen, die in diesem Kapitel vorgestellt werden, sind nicht simulatorspezifisch, basieren jedoch auf Ideen und Demonstratoren, die mit HDL-A, MAST AHDL oder VHDL-AMS umgesetzt wurden [64, 89, 33].

Für den Entwurf integrierter analoger Schaltungen werden Simulationsmodelle benötigt, welche die Eigenschaften des Fertigungsprozesses genau nachbilden und sich leicht anpassen lassen, wenn sich Prozessparameter ändern. Für die Erstellung dieser Simulationsmodelle wird die Abstraktionsebene *Process Level* verwendet. *Process Level* Beschreibungen analoger Schaltungen sind Modelle der δ-Klasse. Sie basieren auf nichtlinearen partiellen Differentialgleichungen und benötigen spezialisierte Simulationsprogramme mit hoher Genauigkeit und guten Konvergenzeigenschaften. Diese Abstraktionsebene wird nur für Grundelemente verwendet, da die Simulation dieser Modelle sehr zeitaufwendig ist. Halbleiterhersteller nutzen solche Simulationen um ihre Herstellungsprozesse zu analysieren. Da man aus den Modellen Rückschlüsse auf die Herstellungsprozesse ziehen kann, sind die Modelle nicht öffentlich erhältlich. Die Ergebnisse der *Process Level* Simulationen werden verwendet um *Component Level* Modelle zu erstellen.

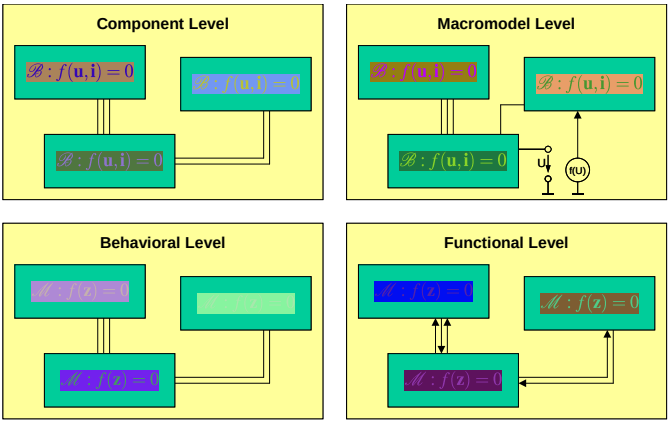
[3] Bei einem Rollback werden die zuletzt berechneten Simulationsergebnisse verworfen und neu berechnet. Dabei werden für die Neuberechnung i. d. R. andere Zeitschrittweiten oder andere Lösungsalgorithmen verwendet.

Die Grundelemente von Modellen der Abstraktionsebene *Component Level* sind passive und aktive Bauelemente wie Widerstände, Kondensatoren oder Transistoren. Die Simulationsmodelle bestehen aus einer Menge verbundener Grundelemente. Alle Verbindungen müssen den Kirchhoffschen Gesetzen genügen. Die Modelle sind dementsprechend konservative Modelle der γ-Klasse. Diese Abstraktionebene ist auch unter dem Namen *Primitive Level* bekannt, da sie der Ausgangspunkt für viele analoge Schaltungsentwicklungen ist.

Die Abstraktionsebene *Macromodel Level* ist vom *Component Level* abgeleitet. Als Grundelemente sind im *Macromodel Level* jedoch nur einige idealisierte Grundelemente (z. B. Widerstände, Kondensatoren, abhängige und unabhängige Strom- und Spannungsquellen) erlaubt. Komplexere Bauelemente müssen durch Schaltungsblöcke, die ein ähnliches Verhalten zeigen, nachgebildet werden. Die entstehenden Modelle sind dementsprechend ebenfalls konservative Modelle der γ-Klasse. Die Einschränkung der Grundelemente auf wenige und idealisierte Komponenten ermöglicht einfachere Simulationsprogramme und wurde von SPICE-ähnlichen Simulationsprogrammen genutzt. Heutzutage sind die meisten SPICE Derivate um die wichtigsten Grundelemente aus dem *Component Level* ergänzt, so dass der *Macromodel Level* an Bedeutung verliert.

Der *Behavioral Level* ist eine Abstraktionsebene, in der analoge Schaltungen durch Verknüpfung von Verhaltensmodellen gebildet werden. Die Verhaltensmodelle können durch eine beliebige Kombination aus differentialalgebraischen Gleichungssystemen und Zustandsmaschinen gebildet werden. Sie gehören zur γ-Klasse, sind jedoch nicht auf die konservative Untermenge beschränkt. Die Verbindungen zwischen den Verhaltensmodellen müssen wie auf den niedrigeren Abstraktionsebenen den Kirchhoffschen Gesetzen genügen. Für diese Abstraktionsebene wurde in [33] eine sinnvolle Unterteilung in einen *Algorithmic Level* und einen *Procedural Level* vorgestellt. Hierbei wird besonderen Wert auf einen 'Meet-in-the-Middle' Entwurfsablauf gelegt: Schaltungsbeschreibungen, für deren durch Verhaltensmodelle beschriebene Komponenten eine bibliotheksgestützte, automatisierte Implementierung möglich ist, sind auf *Procedural Level*, während Beschreibungen mit anderen Verhaltensmodellen 'nur' auf *Algorithmic Level* sind. Diese Abstraktionsebene ist abstrakter, da die Implementierung der Modelle noch nicht festgelegt ist.

Um die Simulation umfangreicher Schaltungen zu erleichtern wurde die Abstraktionsebene *Functional Level* definiert. Modelle dieser Abstraktionsebene werden aus den selben Bestandteilen aufgebaut wie Modelle des *Behavioral Level*, aber die Modelle werden nicht auf Genauigkeit, sondern auf Simulationsgeschwindigkeit optimiert. Ein Verhaltensmodell dieser Abstraktionsebene gehört zur γ-Klasse. Die Kommunikation zwischen den Einzelmodellen erfolgt jedoch durch Kommunikationsschnittstellen, die einen wohldefinierten Signalfluss besitzen. D. h. wie in Modellen der β-Klasse wird zwischen Signalquellen und Signalsenken unterschieden. Dementsprechend genügen Verbindungen auf dieser Abstraktionsebene den Kirchhoffschen Gesetzen nicht, so dass spezielle Kommunikationsschnittstellen benötigt werden, wenn Modelle niedrigerer Abstraktionsebenen mitsimuliert werden sollen.



$\mathscr{B}$: konservatives Komponentenmodell, $\mathscr{M}$: beliebiges Komponenten-Modell

Abbildung 3.5: Modellierung der Abstraktionsebenen analoger Schaltungen im Meta-Modell

3.1.2.1 Abbildung auf die Modellklassen des Meta-Modells

Die Zustandsgrößen in Modellen zur Simulation analoger Schaltungen sind zeitkontinuierlich. Die Modelle selbst sind zeitkausal, im allgemeinen jedoch nicht streng zeitkausal.

Zur Beschreibung der nichtlinearen partiellen Differentialgleichungen der *Process Level* Modelle werden ortsabhängige Zustandsgrößen benötigt, die durch ein Modell der δ-Klasse beschrieben werden können. Für die anderen Abstraktionsebenen sind parametrisierbare Zustandsgrößen ausreichend, so dass sich hierfür Modelle der γ-Klasse anbieten. Jede dieser Abstraktionsebenen hat jedoch Besonderheiten, die in Abb. 3.5 veranschaulicht werden.

Component Level sind vollständig konservative Modelle, während die abhängigen Strom- und Spannungsquellen des *Macromodel Level* die Modellierung eines gerichteten Signalflusses ermöglicht, d. h. es ist möglich ein zeitkontinuierliches Teilmodell der β-Klasse in das ansonsten konservative γ-Klassen Modell einzubetten. Verhaltensbeschreibungen des *Behavioral Level* sind mit Modellen der γ-Klasse beschreibbar, sind jedoch nicht auf die Untermenge der konservativen Modelle beschränkt. Lediglich die Verknüpfung von Teilkomponenten muss mit konservativen Zustandsgrößen erfolgen. In Beschreibungen des *Functional Level* erfolgt die Verknüpfung von Teilkomponenten wie in Modellen der β-Klasse mit Signalen, d. h. es wird zwischen Signalquellen und Signalsenken unterschieden, während die Teilkomponenten selbst durch γ-Klassen Modelle beschrieben werden können.

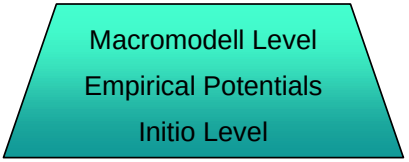


Abbildung 3.6: Abstraktionsebenen für den Entwurf mikromechanischer Komponenten [41]

3.1.3 Modellbildung in der Mikrosystemtechnik

> „Micromechanical systems, or MEMS, are: integrated micro devices or systems combining electrical and mechanical components, fabricated using integrated circuit (IC) compatible batch-processing techniques and varied in size from micrometers to millimeters. These systems merge computation with sensing and actuation to change the way we perceive and control the physical world.“[4]

Der systematische Entwurf von mikromechanischen Systemen (MEMS) ist eine große Herausforderung, für die Know-How in elektrischen und mechanischen Gebieten benötigt wird. In diesem Abschnitt werden die Anforderungen für die Untersuchung der Materialeigenschaften betrachtet, wie sie von M. Kasper [41] definiert wurden.

Die Tatsache, dass die mechanischen Teile stark miniaturisiert sind, erzeugt für den Entwurf und die Simulation solcher Bausteine neue Anforderungen: Die physikalischen Eigenschaften von Sensoren und Aktoren werden stark durch Oberflächeneigenschaften beeinflusst. Es werden Simulationsmodelle benötigt, die Effekte aufgrund von atomaren Wechselwirkungen beschreiben können.

Die niedrigste Abstraktionsebene wird *Initio Level* genannt. Modelle dieser Abstraktionsebene beschreiben atomare Wechselwirkungen mit Hilfe von fundamentalen physikalischen Gesetzen. Mit solchen Modellen können z. B. die Auswirkungen von atomaren Oberflächendefekten auf die elektrische und thermische Leitfähigkeit untersucht werden. Aufgrund des hohen Simulationsaufwands können nur aus wenigen Atomen bestehende Bruchstücke eines Sensors gleichzeitig simuliert werden. Dabei ist zu berücksichtigen, das solche Problemstellungen nicht mit gewöhnlichen FEM-Ansätzen[5] untersucht werden können, da die Materialeigenschaften nicht gleichförmig sind. Statt dessen müssen auf die Position der Einzelatome bezogene Potentialfelder verwendet werden [77].

[4] Quelle: MCNC, http://www.mcnc.org/
[5] FEM: Finite Elements Methods

Um größere Teile zu simulieren, werden Modelle auf der Abstraktionsebene *Empirical Potentials Level* verwendet. Diese Modelle basieren auf verallgemeinerten Potential- und Flussgrößen, mit denen Wechselwirkungen im atomaren Verbund beschrieben werden. Solche Modelle ermöglichen z. B. die simulative Optimierung der Geometrie von widerstandsbehafteten Anschlusskontakten eines piezoresistiven Sensors. Um diese Modelle zu berechnen, müssen partielle Differentialgleichungsysteme gelöst werden. Dazu wird i. d. R. die numerische Feldtheorie genutzt, d. h. das Modell wird mit Hilfe von *Finite Elemente* Methoden diskretisiert und numerisch approximiert.

Einige Bauelemente aus der MEMS-Technologie (z. B. GHz Silizium-Resonatoren mit Materialdicken in der Größenordnung von 100 Atomlagen [16]) sind so klein, dass Mischmodelle benötigt werden, in denen *Initio Level* Modelle in gewöhnliche *Empirical Potentials Level* Modelle eingebettet werden [77].

Die Abstraktionebene *Macromodel Level* ist über den geometrie-orientierten Abstraktionsebenen angeordnet. Auf dieser Abstraktionsebene wird nur das Verhalten der mikromechanischen Komponenten nachgebildet, wobei Simulationsdaten aus den niedrigeren Abstraktionsebenen zur Kalibrierung verwendet werden können. Diese Modelle werden hauptsächlich verwendet um Stimuli für die dazugehörigen elektrischen Komponenten zu erzeugen. Um diese Anwendung zu ermöglichen, werden die Modelle meist in der Syntax einer analogen elektrischen Simulationssprache (siehe Kapitel 3.1.2) notiert.

3.1.3.1 Abbildung auf die Modellklassen des Meta-Modells

Eine Untersuchung der physikalischen Eigenschaften von Werkstoffen und der geometrischen Beschaffenheit von mechanischen Komponenten erfordert raumkontinuierliche Simulationsmodelle. Die Modelle der Abstraktionsebenen *Initio Level* und *Empirical Potentials* können deshalb innerhalb des Meta-Modells nur mit Modellen der δ-Klasse beschrieben werden. Die Definition der δ-Klasse beschreibt lediglich fundamentale Grundeigenschaften der Modelle: Sie müssen vollständig zeitlich geordnet und zeitkausal sein. Die Menge der Raumpunkte muss eine stückweise kontinuierliche Menge sein. Die Zustandsgrößen selbst müssen funktional sein, dürfen jedoch ortsabhängig sein. Diese weitgefasste Festlegung ermöglicht für beide obengenannte Abstraktionsebenen eine Modellierung innerhalb des Meta-Modells, wobei außer Zweifel steht, dass innerhalb der δ-Klasse eine weitere Klassifizierung von Modellen mit unterschiedlichen Eigenschaften möglich ist.

Die Abstraktionsebene *Macromodel Level* dient wie oben angesprochen, zur Interaktion mit dazugehörigen elektrischen Komponenten und kann i. d. R. mit Modellen der γ-Klasse beschrieben werden.

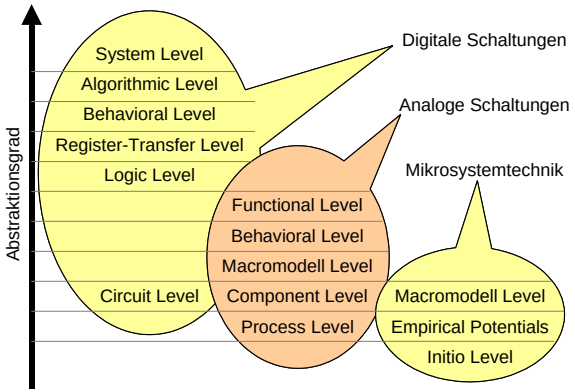


Abbildung 3.7: Zusammenstellung der domänenspezifischen Abstraktionsebenen im Gesamtsystemkontext

3.2 Modellbildung für heterogene Systeme

Die im letzten Kapitel vorgestellten Abstraktionsebenen sind auf die domänenspezifischen Entwicklungsabläufe zugeschnitten. Sie können nicht ohne Erweiterungen für die Entwicklung heterogener Systeme benutzt werden. Die vorgestellten Gliederungen in Abstraktionsebenen wurden zum Teil mit ähnlichen Begriffen benannt, obwohl die darin beschriebenen Modelle unterschiedliche Anforderungen an ein Simulationssystem stellen. Abb. 3.7 zeigt einen Gesamtüberblick über diese domänenspezifischen Abstraktionsebenen.

In diesem Kapitel wird nun ein neues, kompakteres System aus Abstraktionsebenen definiert. Dieses System bietet domänenübergreifende Beschreibungsmöglichkeiten und ist für den Entwurf vieler heterogener Systeme geeignet. Diese in Abb. 3.8 dargestellten Abstraktionsebenen wurden im Hinblick auf Systeme aus heterogenen Sensoren und Aktoren mit elektronischer Signalverarbeitung erstellt. Dabei wurde die Zeitbereich-Simulation als ein zentraler Bestandteil des Entwurfsprozesses angesehen. Die Abstraktionsebenen dienen zur Klassifizierung von *Verhaltensmodellen*. Die Abstraktionsebenen sind domänenübergreifend und unterscheiden sich bezüglich des Zeitmodells, sowie der Mächtigkeit der Beschreibungskonzepte. Sie werden in vier Kategorien eingeteilt, für welche die Begriffe *Algorithmische Ebene*, *Funktionale Ebene*, *Gleichungsbasierte Ebene* und *PDE-Ebene* eingeführt werden.

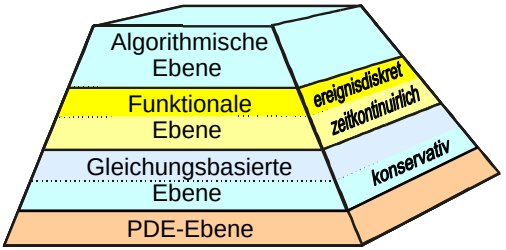


Abbildung 3.8: Abstraktionsebenen für die Modellierung heterogener Systeme

3.2.1 Algorithmische Modellierung

Algorithmische Modelle sind eine Umsetzung des Modellierungskonzepts der α-Klasse. Sie werden im allgemeinen durch strukturierte Modelle aus mehreren Teilkomponenten beschrieben. Diese Teilkomponenten besitzen Eingangs- und Ausgangsschnittstellen, über welche sie zu einem Gesamtmodell zusammengeschaltet werden.

Ein einfaches Modellierungskonzept für algorithmische Modelle ist das Konzept der *'Cycle based Simulation'*, das unter anderem im VHDL Simulator Scirocco [83] verwendet wird. Die Modelle bestehen aus einem Netzwerk synchroner Prozesse. Jeder Prozess wird pro Simulationszyklus einmal aktiviert. Bei der Aktivierung werden die Eingangsdaten gelesen und verarbeitet. Die berechneten Ausgangsdaten werden in einem Zustandspeicher gespeichert und den angeschlossenen Prozessen im folgenden Simulationszyklus zur Verfügung gestellt.

Konsistente synchrone Datenflussgraphen (SDF) [85] können ebenfalls in algorithmische Modelle umgesetzt werden. Jeder Knoten des Graphen ist ein Prozess, der pro Simulationszyklus einmal ausgeführt werden kann. Die Kanten des Graphen, welche die Ein- und Ausgänge der Prozesse verbinden, sind als FIFO Speicher realisiert. Bei der Ausführung der Prozesse werden von den Eingängen 'Marken' eingelesen, verarbeitet und an den Ausgängen die Resultate als 'Marken' für den nächsten Simulationszyklus erzeugt, wobei die Anzahl der konsumierten und produzierten 'Marken' unterschiedlich sein kann. Wenn für einen Prozess die benötigen Eingangsdaten nicht vorhanden sind, wird seine Ausführung ausgesetzt. Um einen Datenüberlauf auf den Kanten zu vermeiden, ist die Ausführungsrate der Generatorknoten geeignet zu wählen.

Das Modellierungskonzept der *'Process Network Domain'* [59] erzeugt ebenfalls algorithmische Modelle. Ein Modell wird als ein Netzwerk aus asynchronen Prozessen beschrieben, die durch FIFO Speicher miteinander kommunizieren. Ein Prozess, der von einem leeren Kanal lesen will, wird unterbrochen, bis eine Nachricht verfügbar ist. Zusätzlich können die Prozesse sich selbst durch einen 'delay' Befehl stillegen. Wenn alle Prozesse unterbrochen oder stillgelegt sind, wird die Simulationszeit ein Inkrement weitergeschaltet und die Prozesse werden wieder aktiviert.

Ein Hauptanwendungsgebiet für algorithmische Modelle ist die Beschreibung von signalverarbeitenden Systemen. Algorithmische Modellbeschreibungen können direkt in digitale Schaltungen umgesetzt werden. Genauso effizient lässt sich aus solchen Modellen Software für eingebettete Systeme mit Mikroprozessor erstellen. Für die Optimierung und Validierung nicht elektrischer Komponenten liegt die Bedeutung algorithmischer Modelle in der Bereitstellung von Stimuli.

3.2.2 Funktionale Modellierung

Die funktionalen Modelle unterscheiden sich von den algorithmischen Modellen durch das Zeitmodell. Die funktionalen Modelle gehören zu den Modellen der β-Klasse. Sie nutzen eine globale Zeit, die für Modellierungszwecke verwendet werden kann.

Das ereignisbasierte Modellierungskonzept ist das bekannteste Konzept für funktionale Modelle. Ereignisbasierte Modelle werden aus Prozessen und unidirektionalen Signalen, mit denen die Ein- und Ausgänge der Prozesse verbunden werden, gebildet. Die Prozesse werden durch Ereignisse auf ihren Eingangskanälen aktiviert und können auf ihren Ausgängen Ereignisse erzeugen, die mit wählbarer Verzögerung weitergeleitet werden. Damit die Prozesse in der richtigen Reihenfolge aktiviert werden, gibt es eine zentrale Liste, in der alle zukünftigen Ereignisse chronologisch geordnet notiert werden. Ist die Ausführung der Prozesse abgeschlossen, wird die globale Zeit auf den Zeitpunkt des frühsten Ereignisses in der Liste gesetzt und die entsprechenden Prozesse aktiviert. Alle Prozesse, die zur gleichen Zeit aktiviert werden, werden parallel ausgeführt.

Das ereignisbasierte Modellierungskonzept wird zur Simulation von VHDL oder Verilog Beschreibungen eingesetzt. Dabei werden durch die einzelnen Prozesse sowohl verzögerungsbehaftete boolesche Funktionen, als auch flankengesteuerte Kontrollogik modelliert.

Eine Anwendung für zeitkontinuierliche funktionale Modelle ist die Beschreibung von Signalquellen oder die Modellierung von Sensoren, die in gleichungsbasierte Modelle eingebettet werden. In dieser Applikation werden gerichtete Signalgrößen genutzt, um sicherzustellen, dass der Sensor nicht invers betrieben wird.

Hauptanwendungsgebiet für funktionale Modelle ist die Transientenanalyse ereignisdiskreter Systeme. Ereignisdiskrete Systeme können exakt simuliert werden und auf funktionale Fehler oder Hazards untersucht werden. Für integrierte Schaltungen werden vielfach funktionale Modelle erstellt, die mit Hilfe von Synthesewerkzeugen weiterverarbeitet werden. Für die Optimierung und Validierung von heterogenen Systemen sind die zeitkontinuierlichen funktionalen Modelle für die Modellierung von Steuer- und Regelkreisen von Bedeutung.

3.2.3 Gleichungsbasierte Modellierung

Die gleichungsbasierten Modelle sind eine Umsetzung der γ-Klasse. Konservative gleichungsbasierte Modelle sind in der Elektrotechnik seit vielen Jahren ein bewährtes Hilfsmittel zur Validierung und Optimierung von analogen Schaltungen. Das Modellierungskonzept der Netzwerksimulation, auf dem die Simulatoren der SPICE Familie aufbauen, ist allgemein bekannt. Modelle werden durch Verdrahtung von Elementarbausteinen wie Widerstände, Kondensatoren, Spulen, etc., zusammengesetzt. Für die Verbindungsknoten gelten die Kirchhoffschen Erhaltungssätze, die automatisch in Gleichungen umgesetzt werden. Das Verhalten der Elementarbausteine ist im Simulatorkern hinterlegt.

Mit den vorgegebenen Elementarbausteinen lassen sich zwar verschiedenste mathematische Gleichungssysteme erzeugen. Diese Modelle sind jedoch schwer zu verstehen, wenn sie Komponenten aus anderen Domänen beschreiben sollen. Analoge Beschreibungssprachen bieten eine besser verständliche Alternative. In diesen Beschreibungssprachen werden Modelle ebenfalls durch Verdrahtung von Entwurfsobjekten gebildet, wobei deren Verhalten mit mathematischen Gleichungen definiert werden kann.

Moderne Beschreibungssprachen für zeitkontinuierliche Modelle wie HDL-A [66], MAST [84] oder VHDL-AMS [39] ermöglichen, zusätzlich zur Beschreibung gleichungsbasierter Modelle, die Beschreibung funktionaler Teilmodelle.

Im Entwurfsablauf heterogener Systeme hat sich die Modellierung mit gleichungsbasierten Modellen als ein erster Schritt zur Systemsimulation etabliert. So wurden z.B. optische und magnetische Sensoren [61, 31] als gleichungsbasiertes Verhaltensmodell beschrieben, um ein System aus Sensoren und Elektronik gemeinsam simulieren zu können.

In dieser Abstraktionsebene können auch ortsabhängige Problemstellungen modelliert werden, wenn diese Problemstellungen durch *örtlich konzentrierte* Zustandsgrößen beschrieben werden können. So ist beispielsweise die Position eines Gewichts während einer Pendelbewegung in einem gleichungsbasierten Verhaltensmodell beschreibbar, da hierfür Zustandsgrößen definiert werden können, deren Wert die Position als Funktion der Zeit beschreibt.[6]

3.2.4 PDE-Modellierung

PDE-Modelle sind Modelle der δ-Klasse. Sie ermöglichen die Beschreibung räumlich verteilter Problemstellungen. Ein wichtiger Bestandteil dieser Modelle sind partielle Differentialgleichungen *Partial Differential Equations, PDE*.

[6] Diese Zustandsgrößen sind nach Def. 2.4 *nicht* ortsabhängig.

Simulationssysteme für PDE-Modelle sind zahlreich und meist auf spezielle Anwendungen zugeschnitten. Die Modellierungskonzepte sind jedoch ähnlich. Die Modellbildung ist zweiphasig: In der ersten Phase wird eine geometrische Beschreibung der zu simulierenden Körper erstellt. In der zweiten Phase werden die Anregungen festgelegt. Dies können Kräfte, Potentialfelder oder Flussgrößen sein.

Die zu simulierenden partiellen Differentialgleichungen sind in vielen der spezialisierten Simulatoren Bestandteil der Lösungsalgorithmen und können nur über Materialparameter variiert werden.

Die Simulation ist ebenfalls zweiphasig. Zuerst wird die räumliche Diskretisierung durchgeführt und dadurch ein gleichungsbasiertes Ersatzmodell erzeugt. Dieses wird anschließend in der Zeitdomäne simuliert. Mit Hilfe der Simulationsergebnisse kann die räumliche Diskretisierung überprüft, bzw. korrigiert werden, so dass durch Wiederholung des Zyklus aus Diskretisierung und Zeitsimulation die Genauigkeit der Simulationsergebnisse verbessert werden kann.

PDE-Modelle eignen sich nur zur Beschreibung von ausgewählten Teilkomponenten heterogener Systeme. Eine Simulatorkopplung für PDE-Modelle ist aufwendig [42], da an den I/O-Schnittstellen Informationen über 2D-/3D-Felder übertragen werden müssen. Die Bedeutung der PDE-Modelle liegt in der Validierung und Optimierung physikalischer Eigenschaften von Sensoren und Aktoren, die in den höheren Abstraktionsebenen nicht ohne weiteres dargestellt werden können.

3.3 Transformation zwischen Abstraktionsebenen

Wie im letzten Kapitel hervorgehoben, werden mit der Wahl des Modellierungskonzepts die Anforderungen an ein Simulationssystem festgelegt. Wenn Modelle mit unterschiedlichen Modellierungskonzepten gemeinsam simuliert werden sollen, müssen an den Modellschnittstellen zusätzliche Randbedingungen eingehalten werden.

Modellierungskonzepte der selben Modellklasse besitzen kompatible Datenmodelle, so dass durch Modelltransformationen oder Simulatorkopplung gemeinsame Modellausführungen möglich sind. Für Modelle der α-Klasse oder der β-Klasse muss lediglich ein 'Fenster' zum Datenaustausch ermöglicht werden.

Da für die gemeinsame Simulation zweier Modelle der γ-Klasse die Lösung des gemeinsamen impliziten Gleichungssystems benötigt wird, ist die Aufteilung auf zwei Simulatoren mit deutlich höherem Aufwand verbunden. Um den Zustand der gemeinsamen Zustandsvariablen Z zu ermitteln, wird ein *Meta-Simulator* benötigt, der durch geeignete Synchronisationsverfahren den Datenaustausch ermöglicht. Dabei ist das Optimierungsziel die Minimierung des Fehlers $|Z_{\mathrm{Sim}_1} - Z_{\mathrm{Sim}_2}|$. Der Informationsfluss zwischen den gekoppelten Simulatoren ist dementsprechend verrauscht. Zusätzliche Fehler entstehen durch die in der Regel unterschiedlichen zeitlichen Diskretisierungsschrittweiten der Simulatoren.

Bei der Kopplung von Simulatoren der δ-Klasse ist diese Problematik noch ausgeprägter. Die Zustandsgrößen liegen als diskretisiertes n-dimensionales Feld vor. Da der angekoppelte Simulator im allgemeinen eine andere Diskretisierung verwendet, werden zusätzliche Rauschsignale eingestreut. Desweiteren ist die Übermittlung des diskreten n-dimensionalen Feldes mit einem hohen Datentransfervolumen verbunden. Dadurch sinkt die Simulationsgeschwindigkeit.

3.3.1 Mixed-Level Simulation

Bei der Entwicklung komplexer Systeme ändern sich die Anforderungen an die Systemmodellierung: In der Projektphase werden Machbarkeitsstudien durchgeführt, für die sich die α-Klasse anbietet. Für die Entwicklung von Subsystemen wie Sensoren und deren Ansteuerelektronik werden Modelle der γ-Klasse und der δ-Klasse verwendet. Die Entwicklung der signalverarbeitenden Komponenten basiert hingegen auf Modellen der β-Klasse und der α-Klasse. Ein Gesamtsystemmodell ermöglicht hierbei die Validierung und Optimierung der signalverarbeitenden Komponenten bei realistischen Anregungen. Für eine solche Gesamtsystemsimulation ist es sinnvoll, abstrakte Verhaltensmodelle für Sensoren und Aktoren zu erstellen.

Alternativ bieten sich Validierungssysteme an, die Modelle aus mehreren Modellklassen gemeinsam simulieren können. Die Simulation von Modellen unterschiedlicher Modellklassen wird *Mixed-Level* Simulation genannt. VHDL-AMS basierte Simulationssysteme gehören zu dieser Kategorie. Sie können Verhaltensmodelle der β-Klasse und der γ-Klasse simulieren. Das Simulationssystem Ptolemy II [59] gehört ebenfalls zu dieser Kategorie. In diesem System können Modelle der α-Klasse und Modelle der β-Klasse gemeinsam ausgeführt werden.

Um Modelle unterschiedlicher Modellklassen gemeinsam zu simulieren, müssen die abstrakteren Verhaltensmodelle durch ein intelligentes Interface angekoppelt werden. In diesem Interface werden die Komponentenmerkmale nachgebildet, die auf der höheren Modellklasse nicht ausdrückbar sind.

3.4 Umsetzung auf vorhandene Simulationsumgebungen

3.4.1 Modellbildung mit VHDL-AMS

VHDL-AMS [39] ist eine Weiterentwicklung des IEEE Standards 1076 von 1993. Diese neue Simulationssprache beinhaltet das aus dem Entwurf digitaler Schaltungen bekannte VHDL und ergänzt es mit Beschreibungsmöglichkeiten für zeitkontinuierliche Modelle. Dadurch wird es möglich drei Modellklassen mit verschiedenen Ausführungsparadigmen gemeinsam zu notieren und zu simulieren: Sequentielle zeitfreie Algorithmen

mit ereignisdiskreter Kommunikationsschale, nebenläufige ereignisdiskrete Verhaltensbeschreibungen und zeitkontinuierliche Modelle mit bidirektionalem Signalfluss. Die durch implizite differential-algebraische Gleichungssysteme beschriebenen Modelle in der zuletzt genannten Klasse eigenen sich z. B. zur Beschreibung von analogen Schaltungen. Zur Beschreibung von synthetisierbaren digitalen Schaltungen sind die über explizite Zuweisungen (bzw. Bus-Resolution Functions) beschriebenen Modelle über den ereignisdiskreten Zustandsgrößen sinnvoll, während mit Hilfe von lediglich sequentiell geordneten, zeitkausalen Algorithmen in *Subprograms* komplexe Berechnungen effizient durchgeführt werden können [47]. Ein über *Variablen* definiertes Verhaltensmodell ist ein Element der α-Klasse. Die mit Hilfe von *Signalen* modellierten ereignisdiskreten Modelle sind Elemente der β-Klasse. Modelle mit ungerichteten zeit- und wertkontinuierlichen Zustandsgrößen (*Quantities* und *Terminals*) sind aus der γ-Klasse. Mischmodelle, in denen Elemente verschiedener Klassen kombiniert werden, sind in VHDL-AMS möglich und oft sinnvoll, da sie eine Zusammenfassung des Modellverhaltens in Funktionsgruppen ermöglichen und somit die Modellpflege erleichtern.[7]

Diese Ausdrucksmächtigkeit ermöglicht einen Entwicklungsablauf, der von der Konzeptionsphase bis zur Implementierungsphase auf VHDL-AMS Modellen basiert. Aus den Ergebnissen einer Machbarkeitsstudie werden zunächst Gesamtsystemmodelle erzeugt, die für eine experimentelle Validierung verwendet werden. Dementsprechend sind die Anforderungen an ein Gesamtsystemmodell: Es muss effizient simulierbar sein; die enthaltenen Komponentenmodelle müssen leicht verständlich sein und das durch die Gesamtsystemsimulation nachgebildete Systemverhalten muss eine aussagekräftige Validierung ermöglichen. In Kapitel 5 wird an einem Anwendungsbeispiel gezeigt, wie diese Ziele mit VHDL-AMS Modellen erreicht werden können.

In den folgenden Implementierungsschritten werden einzelne Komponenten des Gesamtsystemmodells weiterentwickelt und genauer beschrieben. Dadurch entstehen Modelle, deren Verhalten besser beschrieben ist, deren Simulation jedoch mehr Rechenzeit in Anspruch nimmt. Dementsprechend ist es in der Regel ineffizient eine Simulation des Gesamtsystems mit vielen detailliert beschriebenen Komponenten durchzuführen. Stattdessen werden *kalibrierte High-Level Modelle* benötigt, die das Verhalten einer Komponente simulationseffizient nachbilden. Auch für diese Modelle ist VHDL-AMS geeignet, sowohl bei analogen und digitalen Komponenten, als auch bei mechanischen Baugruppen, wie beispielsweise Gibson et. al. am Beispiel eines *Cantilever Beam-Capacitor Systems* demonstriert [24].

Diese Pflege der Gesamtsystemmodelle ist ein zusätzlicher und unverzichtbarer Bestandteil des Entwurfsablaufs.

In größeren Projekten, die von mehreren Arbeitsgruppen bearbeitet werden, ist die Integration der Teilkomponenten zu einem Gesamtsystem eine zusätzliche Fehlerquelle,

[7] Für die Kommunikation zwischen Modellen der α-Klasse werden ereignisdiskrete Signale benötigt, so dass zwangsläufig Elemente der β-Klasse enthalten sind. Wenn in einem Modell Signale ausschließlich zu Kommunikationszwecken eingesetzt werden, so wird es im Folgenden als Modell der α-Klasse bezeichnet.

deren Signifikanz nur durch die frühzeitige Gesamtsystemmodellierung an Bedeutung verlieren kann. Ebenso ist es sinnvoll ausführliche Testbenches zu erarbeiten, die eine automatisierte Validierung ermöglichen. Zu diesem Zweck können Komponentenmodelle aus der Gesamtsystemmodellierung als *Referenz* verwendet werden, so dass sich der Aufwand für die Gesamtsystemmodellierung relativiert.

3.4.2 Modellbildung mit C

Die Programmiersprachen C und C++ werden bei der Beschreibung von heterogenen Systemen auch eingesetzt. Es existieren mehrere Spracherweiterungen und Klassenbibliotheken, die eine Modellierung physikalischer Problemstellungen erleichtert. Die Entwicklung von *SystemC*[88] soll die Erstellung von synthesefähiger Modelle digitaler Schaltungen ermöglichen. Die Fortschritte in dieser Richtung sind beachtlich: Die Firma *Modeltech Inc.* erweitert ihren Digitalsimulator ModelSim [71], so dass eine gemeinsame Simulation von VHDL Modellen und SystemC Modellen möglich wird. Die Synopsys Inc. stellt Syntheseprogramme her, die erste SystemC Modelle auf Standardzellen-Bibliotheken abbilden kann. Mit SystemC können somit ereignisdiskrete Modelle der β-Klasse beschrieben und simuliert werden. Aktuelle Forschungsarbeiten beschäftigen sich mit der Erweiterung von SystemC zu *SystemC-AMS*[90]. Im Rahmen dieser Arbeiten wird das SystemC System so erweitert, dass eine Beschreibung von zeitkontinuierlichen Modellen der β-Klasse und eine Beschreibung einiger Modelle der γ-Klasse möglich wird.

Neben diesen Modellierungsaufgaben, die auch mit VHDL-AMS gelöst werden können, bietet sich eine Programmiersprache wie C zur Programmierung von Mikroprozessoren und anderen programmierbaren Komponenten eines heterogenen Systems an. Diese Programme erfüllen i. d. R. die Anforderungen an ein Modell der α-Klasse. Typische Beispiele für solche Modellierungsaufgaben sind auch Stimuligeneratoren und Machbarkeitsstudien. Ein solcher Stimuligenerator kann eingesetzt werden, um aus vorhandenen Datensätzen die zur aktuellen Anwendung passenden Daten zu extrahieren, oder um zusätzliche Datenwerte zu interpolieren.

Für die in Kapitel 5 beschriebene Anwendung wurde ein C Programm eingesetzt, um den Algorithmus zur Signalverarbeitung zu untersuchen. Im Rahmen dieser Machbarkeitsstudie wurde überprüft, ob der Algorithmus mit dem geplanten anwendungsspezifischen Mikroprozessor realisiert werden kann. Dabei wurden Randbedingungen wie die benötigte Rechengenauigkeit, oder die benötigte Taktrate bestimmt.

3.4.3 Modellbildung mit SPICE

Das Simulationssystem SPICE [17] wurde an der University of California in Berkeley entworfen und wird seitdem kontinuierlich weiterentwickelt. Es ermöglicht die Beschreibung und Simulation analoger elektronischer Schaltungen mit Hilfe von nicht-

linearen, zeitinvarianten Bauelementen. Mittlerweile gibt es parallel zu dem Original SPICE Simulator mehrere kommerzielle SPICE-kompatible Netzwerksimulatoren, die im Hinblick auf Simulationsgeschwindigkeit, Konvergenz optimiert, sowie z. T. mit zusätzlichen Grundelementen ausgestattet sind. Die im Rahmen dieser Arbeit gezeigten Simulationsergebnisse wurden mit dem *eldo* Simulator [68] erzeugt, einem SPICE-kompatiblen Simulator der Firma Mentor Graphics.

Diese Netzwerksimulatoren berechnen Maschenströme und Knotenpotentiale in Modellen analoger elektrischer Schaltungen. Die Modelle sind Strukturmodelle, die durch Verknüpfung von parametrisierbaren Elementarbausteinen entstehen. Das Verhalten dieser Grundelemente, die Widerstände, Dioden, Transistoren, etc. beschreiben, ist durch simulatoreigene Modelle vorgegeben. Die Modellentwicklung besteht somit aus der Auswahl geeigneter Grundelemente, ihrer Parametrisierung und ihrer Verdrahtung. Für die Verbindungen gelten die Kirchhoffschen Erhaltungssätze. SPICE Modelle können deshalb i. d. R. durch konservative Modelle der γ-Klasse dargestellt werden.

Durch die Beschränkung der Verhaltensmodelle auf wenige systeminterne Grundelemente ist die Modellierung von komplexen Komponenten durch vereinfachte Modelle erschwert. Andererseits profitiert man bei der Simulation von den hochoptimierten Grundelementen, die z. T. numerische Optimierungen und Konvergenzhilfen enthalten.

Im Entwurfsablauf für analoge integrierte Schaltungen hat sich die SPICE Simulation zur Validierung des Schaltungsverhaltens etabliert. Halbleiterhersteller stellen für ihre Herstellungsprozesse parametrisierte Bauteilbibliotheken zur Verfügung. In diesen Bibliotheken sind i. d. R. mehrere Parametersätze enthalten, um die zu erwartenden Prozessschwankungen abzudecken. So wird eine Simulation der daraus zusammengesetzen Komponentenmodelle sowohl bei typischen, als auch bei besonders guten und relativ ungünstigen Prozessparametern ermöglicht.

Zusätzlich zu den für die Beschreibung elektronischer Schaltungen benötigten Grundelementen gibt es einige Elemente, die zur Beschreibung von abstrakten Verhaltensmodellen genutzt werden können. So kann ein rückkopplungsfreies Modell beispielsweise mit gesteuerten Strom- und Spannungsquellen beschrieben werden. Da die Grundelemente jedoch ausschließlich über die 'konservative Verdrahtung' verknüpft werden können, resultiert eine Abbildung von nicht konservativen Modellen in ein SPICE Modell nur bei sehr trivialen Modellen in einer nachvollziehbaren Beschreibung.

Kapitel 4

Entwurf und Validierung heterogener Systeme

4.1 Systemspezifikation und Test

Eine große Herausforderung beim Entwurf heterogener Systeme ist die Erstellung eines exakten Pflichtenheftes und dessen Verifikation. In der Regel ist es nicht möglich eine umfassende und vollständige Spezifikation vor Beginn der Implementierungsphase zu erhalten. Es müssen Methoden erarbeitet werden, die eine nachträgliche Ergänzung der Spezifikation ermöglichen. Dabei müssen sowohl Inkonsistenzen beseitigt, als auch funktionale und strukturelle Details ergänzt werden können. Zusätzlich zu diesen auch im Softwareentwurf vorhandenen Randbedingungen müssen in heterogenen Systemen in der Regel harte zeitliche Randbedingungen überprüft werden.

In großen Projekten, bei denen mehrere Entwickler zusammenarbeiten, müssen zusätzliche Fehlerquellen berücksichtigt werden, die erst bei der Integration der Einzelarbeiten offensichtlich werden. Typische Fehlerquellen sind die Kommunikationsschnittstellen der einzelnen Komponenten. Diese Problematik lässt sich an einem Beispiel aus der Praxis zeigen:

In einem System zur Positionsbestimmung werden über ein Getriebe mehrere Dauermagnete angesteuert. Die Position dieser Magnete wird mit Hilfe von Magnetfeldsensoren erfasst. Diese Magnetfeldsensoren bestehen aus dem Sensorelement selbst, einer analogen Signalaufbereitung und einem A/D Wandler. Die Ausrichtung des Magnetfelds kann über ein digitales Interface abgefragt werden. Die Sensoren haben zusätzlich einen Fehlerausgang und einen Eingangspin mit dem der Baustein vorübergehend stillgelegt werden kann (Suspend-Modus). Sinnvollerweise wird spezifiziert, dass der Fehlerausgang signalisieren soll, wenn der Baustein keine gültigen Positionsdaten liefern kann. Ebenso wird festgelegt, dass im Suspend-Modus an den Ausgängen die Signaltreiber ausgeschaltet werden, und die Pins hochohmig geschaltet werden.

Diese Funktionalität wurde durch einen Entwickler realisiert, indem er I/O Zellen entwarf, die als digitale Ausgangstreiber und als Tristate-fähige Eingangsschnittstelle genutzt werden können. Im Suspend-Modus werden die Pins des Bausteins lediglich als

Eingänge konfiguriert, da hierbei die Signaltreiber ausgeschaltet werden und die Pins hochohmig werden.

Ein anderer Entwickler ist für die Entwicklung der PCB-Platine und deren Bestückung zuständig. Dessen Pflichtenheft fordert ebenfalls, dass die Fehlersignale der Sensorbausteine stets einen Fehler melden müssen, wenn keine gültigen Positionsdaten zur Verfügung stehen. Darüber hinaus weiß er, dass der Fehlerausgang hochohmig ist, wenn der Baustein stillgelegt ist. Er setzt einen PullUp-Widerstand ein, so dass im Suspend-Modus der Fehler auf 'an' ($5V$) steht. Dieser Entwickler hat somit seine Spezifikation ebenfalls erfüllt.

Die Spezifikation des Sensor-Bausteins ist erfüllt, das gewünschte Ziel wurde jedoch nicht erreicht: Das Gesamtsystem benötigt im Suspend-Modus zuviel Strom.

Der zu hohe Stromverbrauch kommt durch die ungünstige Realisierung des hochohmigen Zustands am Fehlerausgang zustande. Das *Eingangsinterface* hat eine schwache interne Spannungsquelle, die eine Spannung von $2,5V$ liefert, um den hochohmigen Zustand zu erkennen. Durch den externen PullUp-Widerstand fließt nun im Suspend-Modus ein Dauerstrom, der durch den internen Widerstand der Spannungsquelle und den PullUp-Widerstand bestimmt wird.

Dieser Fehler zeigt die Bedeutung von Systemtests, denn die Bestandteile des Systems - für sich genommen - haben die Zielanforderungen erfüllt. Da jede Änderung eine Ursache für neues Fehlverhalten sein kann, ist es notwendig gute *Testbenches* mit hoher Fehlerabdeckung zu haben.

Die Erstellung dieser Testbenches wird vielfach als zusätzliche Arbeitslast angesehen, die gegen Ende des Entwicklungszyklus zu leisten ist. Aller Erfahrung nach führt das dazu, dass die Qualität der Testbenches leidet, wenn die Arbeit an den Komponenten unter Termindruck abgeschlossen werden muss.

Die Erstellung von Testbenches muss jedoch keine zusätzliche Entwicklungszeit kosten. Im Gegenteil: Durch ihre richtige Anwendung kann die Entwicklungszeit reduziert werden. Dabei ist es sinnvoll mehrere Testbenches zu erstellen, um zu vermeiden, dass die Testbench selbst komplex und fehleranfällig wird. In den letzten drei Jahren konnte im Rahmen von mehreren Projekten [53, 44, 43] gezeigt werden, dass eine frühzeitige Validierung der Entwicklungsarbeit zu besseren Ergebnissen führt. Anstelle der üblichen Arbeitseinteilung wird dazu zuerst eine (oder mehrere) Testbenches geschrieben, und erst *im Anschluss daran* die zu entwickelnde Komponente (Abb. 4.1).

Diese Umkehr des Arbeitsflusses bringt mehrere Vorteile mit sich. Sobald ein Pflichtenheft und die Kommunikationsschnittstelle einer Komponente bekannt ist, sollte mit der Erstellung von Testszenarien begonnen werden. Das Entwickeln von Testszenarien hilft, sich eine Vorstellung über die benötigte Funktionalität der Komponente zu verschaffen. In einer Testbench kann ein abstraktes Ersatzmodell der Komponente frühzeitig sinnvoll eingesetzt werden. Anhand des Entwurfs eines Demonstrators für das Projekt *Real*

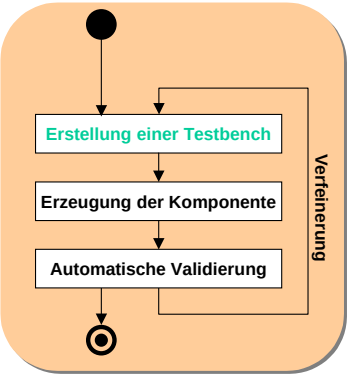


Abbildung 4.1: Test basierter Entwurfsablauf

Time Image Processing based on FPGA Hardware Acceleration wird dieser Arbeitsansatz im Folgenden verdeutlicht. Das implementierte Level-Set Verfahren wird in der Bildverarbeitung unter anderem zur Segmentierung von verrauschten Bilddaten eingesetzt. Die Aufgabenstellung ist durch eine nichtlineare partielle Differentialgleichung gegeben, die - wie in Beispiel 4.1 umrissen - diskretisiert wird.

Zur Validierung des Demonstrators standen Aufnahmen eines menschlichen Gehirns zur Verfügung, in denen ein Tumor segmentiert werden sollte. Somit war es möglich im Vorfeld ein C-Programm zu schreiben, das die gewünschte Funktionalität zeigte. Im nächsten Arbeitsschritt wurde dieses Programm so umstrukturiert, dass es als Testbench genutzt werden konnte. Dazu wurden die Funktionsblöcke, die im FPGA realisiert werden sollten, in eigene *Functions* isoliert. Beschreibung 4.1 zeigt die Iterationsvorschrift, die zur Validierung der FPGA-Ergebnisse verwendet wurde.

Die Ankopplung der FPGA-Karte basierte auf dem Konzept der Simulationsumgebung, die in Kapitel 4.3 vorgestellt wird. Innerhalb einer anwendungsspezifischen I/O Bibliothek wurde eine Funktion definiert, die gegen die Funktion aus Beschreibung 4.1 ausgetauscht werden kann (siehe Abb. 4.2). Diese Ersatzfunktion gibt die Eingangsparameter über den PCI Bus an eine FPGA-Karte zur Datenverarbeitung weiter. Als Rückgabewert wird das Ergebnis der FPGA-Rechenoperation zurückgegeben. Da die dafür benötigte Infrastruktur zur Verfügung steht, ist dieser Arbeitschritt einfach zu bewältigen.

Nachdem nun eine Testbench entstanden war, die eine umfangreiche Validierung der Level-Set Iterationsvorschrift im FPGA ermöglichte, wurde mit der Implementierung und Validierung derselben in VHDL begonnen.

Level-Set Verfahren werden in der Bildverarbeitung unter anderem zur Segmentierung von verrauschten Bilddaten eingesetzt. Die Aufgabenstellung ist durch eine nichtlineare partielle Differentialgleichung gegeben. Gesucht ist die Level-Set Function $\phi : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ mit

$$\begin{aligned} \frac{\partial}{\partial t}\phi + f\|\nabla\phi\| &= 0 \ , \quad \text{in } \mathbb{R}^+ \times \Omega, \\ \phi(0) &= \phi_0 \ , \quad \text{auf } \Omega, \\ \partial_\nu \phi &= 0 \ , \quad \text{auf } \mathbb{R}^+ \times \partial\Omega, \end{aligned} \tag{4.1}$$

wobei f aus den Informationen des Orginalbilds $p : \Omega \to \mathbb{R}$, und der Level-Set Funktion ϕ berechnet wird. Die Diskretisierung von Gl. (4.1) wurde am Institut für angewandte Mathematik der Universität Bonn durchgeführt. Mit Hilfe eines expliziten Eulerverfahrens kann die folgende Iterationsvorschrift gewonnen werden:

$$\bar{\Phi}_\alpha^{k+1} = \bar{\Phi}_\alpha^k - g(D_\alpha^- \bar{\Phi}^k, D_\alpha^+ \bar{\Phi}^k) \tag{4.2}$$

$$D_\alpha^+ \bar{\Phi}^k := \begin{pmatrix} \bar{\Phi}_{\alpha+(1,0)}^k - \bar{\Phi}_\alpha^k \\ \bar{\Phi}_{\alpha+(0,1)}^k - \bar{\Phi}_\alpha^k \end{pmatrix} \tag{4.3}$$

$$D_\alpha^- \bar{\Phi}^k := \begin{pmatrix} \bar{\Phi}_\alpha^k - \bar{\Phi}_{\alpha-(1,0)}^k \\ \bar{\Phi}_\alpha^k - \bar{\Phi}_{\alpha-(0,1)}^k \end{pmatrix} \tag{4.4}$$

$$g(\bar{U}_\alpha, \bar{V}_\alpha) = \bar{F}_\alpha^\oplus \left\| \begin{matrix} \bar{U}_\alpha^\oplus \\ \bar{V}_\alpha^\ominus \end{matrix} \right\| + \bar{F}_\alpha^\ominus \left\| \begin{matrix} \bar{U}_\alpha^\ominus \\ \bar{V}_\alpha^\oplus \end{matrix} \right\| \tag{4.5}$$

$$A^\oplus := \max(A, 0) \tag{4.6}$$

$$A^\ominus := \min(A, 0) \tag{4.7}$$

$$\left\| \begin{matrix} \bar{U} \\ \bar{V} \end{matrix} \right\| \approx c(|\bar{U}|_1 + |\bar{V}|_1) + c\max(|\bar{U}|_\infty, |\bar{V}|_\infty) \tag{4.8}$$

$$c \in [\frac{2}{5}, \frac{1}{2}] \tag{4.9}$$

Beispiel 4.1: Diskretisierung des Level-Set Verfahrens

```
int calcNewPoint(int f, int x10, int x01,
                 int x11, int x21, int x12)
{
  int Dxp, Dxm, Dyp, Dym, c, norm, res;

  c   = x11;
  Dxm = max(0, -sgn(f)*(c-x01));
  Dym = max(0, -sgn(f)*(c-x10));
  Dxp = max(0, sgn(f)*(x21-c));
  Dyp = max(0, sgn(f)*(x12-c));
  norm= Dxm+Dym+Dxp+Dyp+max(max(Dxm,Dym),max(Dxp,Dyp));
  res = f*norm;
  // Addition mit passendem 'Linksshift' von c
  res = (c<<(BITS+2)) +res;
  // round_bits wirft (Bits+2) Bits weg (und rundet evtl. auf)
  res = clampmax(round_bits(res,BITS+2));
  return(res);
}
```

Beschreibung 4.1: Implementierung der Iterationsvorschrift zum Level-Set Verfahren

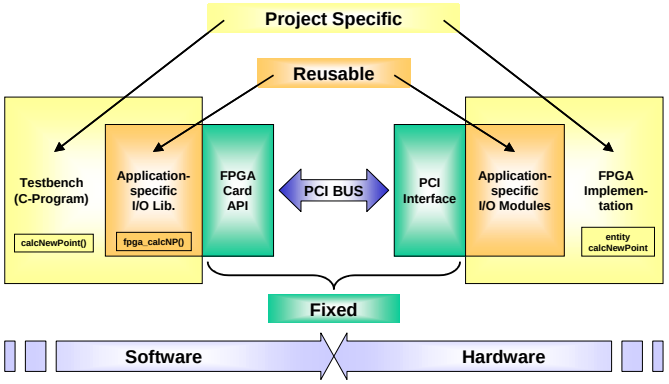


Abbildung 4.2: Partitionierung des Level-Set Demonstrators

Dieses Vorgehen hat sich bewährt. Der Demonstrator wurde innerhalb von 3 Monaten implementiert, wobei für jeden Implementierungsschritt zuerst eine Erweiterung des C-Programms stattfand. Dabei entstanden schon frühzeitig lauffähige 'FPGA-beschleunigte' Level-Set Implementierungen, so dass gegen Ende der Projektphase nicht die Fehlersuche, sondern die Laufzeit-Optimierung im Vordergrund stehen konnte. So konnte das Level-Set Verfahren mit dem endgültigen Demonstrator (etwa 4000 Zeilen VHDL-Code) im Vergleich zur reinen Softwarelösung auf einem Pentium III um den Faktor 20 beschleunigt werden. Eine Testbench sollte automatisiert ausführbar sein und ohne menschliche Interaktion ein Validierungsergebnis produzieren. Es ist sinnvoll etwaige Fehler automatisch festzustellen, wobei es nicht so entscheidend ist, ob die Fehlerursache korrekt erkannt wird. In obigem Beispiel kann ein solcher Automatismus erreicht werden, indem die Referenzberechnung in C parallel zur Berechnung im FPGA durchgeführt wird. Ein Vergleich der Berechnungsergebnisse deckt auch unerwartete Fehler auf: Durch diese Methodik wurde festgestellt, dass in der Testbench ursprünglich anders gerundet wurde als in der FPGA-Implementierung. Eine Testbench muss nicht alle Fehler detektieren. Es können mehrere Testbenches nacheinander ausgeführt werden, wobei der Benutzer nur die fehlerhaften Validierungen überprüfen muss. Eine solche Testsuite kann während der Entwicklungsphase in regelmäßigen Abständen ausgeführt werden um sicherzustellen, dass eine Komponente noch das gewünschte Verhalten zeigt. Diese Tests erleichtern eine *Arbeitsmethodik der kleinen Schritte*, bei der die Komponente inkrementell um zusätzliche Eigenschaften[1] erweitert wird, da Wechselwirkungen frühzeitig offensichtlich werden. Dabei entstehen bei der Weiterentwicklung der Einzelkomponenten zusätzliche Testbenches, da sich die Komplexität der zu testenden Komponenten ändert. Im Level-Set Beispiel wurden nach erfolgreicher Validierung stets größere Teile des Berechnungsverfahrens in den FPGA ausgelagert:

Die Berechnungsfunktion wurde ergänzt durch Cache-Strukturen als Zwischenspeicher für mehrfach benötigte Daten. Diese Cache-Strukturen wurden für die Randpunktbehandlung erweitert. Danach wurde die komplette Datenhaltung auf die FPGA-Karte ausgelagert.

Ein Nachteil dieser Methodik ist, dass selbst eine kleine Änderung an den Komponentenschnittstellen stets Folgearbeiten in den Testbenches erfordert und im Verlauf eines Top-Down Entwurfsablaufs (siehe Abb. 4.3) solche Änderungen wahrscheinlich sind. Durch eine funktionale Partitionierung der Testbench in drei Komponenten kann der Wartungsaufwand in den Testbenches eingegrenzt werden: Wie in Abb. 4.4 gezeigt, wird eine Komponente zur Stimulierzeugung, eine Referenzimplementierung der zu testenden Komponente (Device Under Test, DUT) und ein Kommunikations-Interface für das DUT benötigt. Wenn der zweite Block ebenfalls in anderen Testbenches eingesetzt werden kann, ergibt sich eine Synergie, die eine effektive Wartung der Testbenches ermöglicht.

[1] Darunter sind z. B. Ausnahmeregeln wie die Behandlung der Randpunkte beim Level-Set Verfahren zu verstehen.

Dafür können diese Testbenches während der gesamten Entwicklungsphase und selbst in der Wartungsphase eingesetzt werden, um sicherzustellen, dass die vorhandene Funktionalität nicht durch hinzugefügte Eigenschaften gestört wurde. Das ermöglicht qualitativ hochwertige Entwürfe, selbst wenn gegen Ende der Entwicklungsphase unter Termindruck letzte Änderungen notwendig werden.

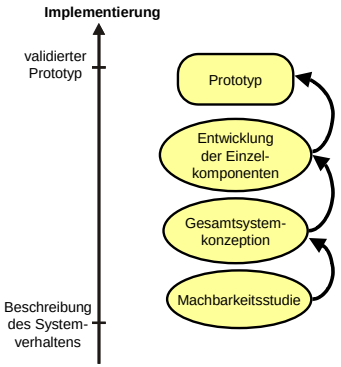


Abbildung 4.3: Implementierungsschritte des Top-Down Entwurfs

Mit einer Testbench können verschiedene Fehlerarten getestet werden. Man unterscheidet die Testbench-Typen nach der Art des Verhaltens, das sie überprüfen. In frühen Entwicklungsstadien muss die algorithmische Korrektheit eines Lösungsansatzes sichergestellt werden. Für diesen Zweck bieten sich α-Testbenches an, die das Verhalten einer Komponente durch Vergleich mit einem Modell der α-Klasse validiert. Nach dem Entwurf des Lösungsansatzes wird das Zeitverhalten des Designs implementiert. Für die Überprüfung der Grenzfrequenzen, Latenzzeiten und Einschwingvorgänge ist eine β-Testbench geeignet, d. h. eine Testbench, die für eine endlichen Anzahl diskreter Messstellen das zeitliche Verhalten überprüft. Für einzelne heterogene Komponenten, z. B. Sensoren oder Aktoren, müssen kontinuierliche ortsabhängige Eigenschaften sichergestellt werden, für deren Überprüfung stark spezialisierte Tests eingesetzt werden. Aufgrund des hohen Aufwands ortskontinuierlicher Simulationen sind solche Tests i. d. R. Speziallösungen, die hier unter dem Oberbegriff der δ-Testbenches zusammengefasst werden.

Während beim Entwurf heterogener Systeme eine Validierung des Systems auf unterschiedlichen Abstraktionsebenen naheliegend erscheint, sind es Entwicklungsingenieure von integrierten elektronischen Schaltungen gewohnt, in ihre α-Testbenches die

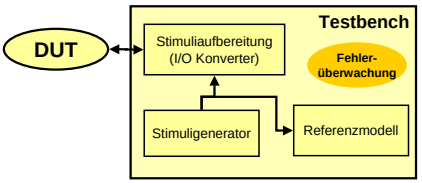


Abbildung 4.4: Funktionale Partitionierung einer Testbench

Validierung des Zeitverhaltens zu integrieren. Diese Vorgehensweise ist jedoch selbst bei rein digitalen Designs nicht zweckdienlich. Die Wiederverwendbarkeit einer ansonsten langlebigen α-Testbench wird dabei durch technologieabhängige Randbedingungen eingeschränkt. Mit der wachsenden Bedeutung von technologieübergreifenden Bibliotheken und entsprechenden Entwurfsumgebungen wächst auch die Bedeutung einer technologieunabhängigen Testbench. Die in einer einzelnen Komponente realisierte Funktionalität nimmt zu, während Halbleiterherstellungsprozesse in kurzen Zyklen weiterentwickelt werden. Die Adaption existierender Entwicklungen auf neue Herstellungsprozesse darf nicht aufgrund einer Aufweichung der Entwurfsmethodik zu unzureichend getesteten Designs führen.

4.2 Entwurfsqualität durch Modellierung mit hohem Abstraktionsgrad

Entwurfsqualität ist ein Oberbegriff, unter dem verschiedene Qualitätskriterien zusammengefasst werden. Ein naheliegendes Maß für Qualität ist die Anzahl der Anwendungsszenarien, in denen ein Fehlverhalten auftritt. Diese Fehlerfälle können mit Hilfe von Testbenches reduziert werden.

Andere Qualitätskriterien können durch eine systematische Systementwicklung sichergestellt werden. Zu diesen Kriterien gehört die Strukturierung eines Systems nach funktionalen Gesichtspunkten, die Wahl der Schnittstellen zwischen den einzelnen Unterblöcken und die Definition und Wiederverwendung von IP-Zellen.[2]

4.2.1 Problematik der Überspezifikation

In jedem strukturierten Entwurfsprozess spielt der Vorgang der Partitionierung eine zentrale Rolle zur Beherrschung der Komplexität großer Systeme. Die informelle Spezifikation des Gesamtsystems muss in eine Struktur aus leichter zu handhabenden Teilsystemen übersetzt werden. Als informelle Spezifikation wird in diesem Zusammenhang beispielsweise ein Pflichtenheft angesehen, dass sowohl Beschreibungen in natürlicher Sprache, Tabellen gewünschter Systemeigenschaften als auch Grafiken, die wichtige funktionale Blöcke des Systems darstellen, enthalten kann. Führt man den Partitionierungsvorgang rekursiv für die entstehenden Teilsysteme weiter aus, so entsteht eine baumartige Hierarchie von immer kleineren Untersystemen, die von Ebene zu Ebene stärker detailiert sind. Die Strukturierung kann dabei nach funktionalen oder physikalischen Gesichtspunkten erfolgen. Sie endet, wenn man bei Blöcken angelangt ist, für die eine Verhaltensbeschreibung direkt aufgestellt werden kann, bzw. bereits vorliegt und die weitere Struktur als bekannt vorauszusetzen ist.

[2] IP-Zelle: Aus dem englischen Intellectual Property Cell abgeleitet.

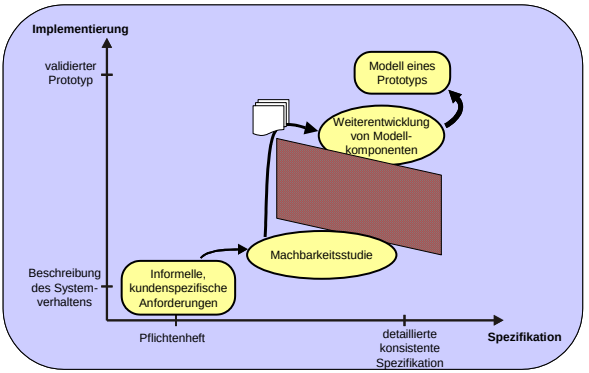


Abbildung 4.5: Problematik üblicher Entwurfsabläufe: Aufgrund von Lücken im Entwurfsablauf können die Ergebnisse der Machbarkeitsstudie nicht direkt weiterverwendet werden.

In frühen Entwicklungsphasen ist dieser Partitionierungsvorgang jedoch problematisch. Oftmals werden inkompatible Simulationsumgebungen genutzt, so dass Modelle aus der Machbarkeitsstudie in späteren Entwicklungsphasen nicht weiterverwendet werden können. Schnittstellen zwischen den Systemkomponenten können evtl. erst in einer späteren Entwurfsphase festgelegt werden, so dass eine Schnittstellendefinition nur eingeschränkt möglich ist. Somit ergibt sich ein Entwurfsablauf, indem insbesondere beim Übergang zwischen den Entwurfsphasen Informationen verfälscht werden können. Abb. 4.5 zeigt ein solches Szenario. Eine simulative Validierung der Entwurfsentscheidungen in den einzelnen Entwurfsphasen ist zwar möglich, eine Lücke im Entwurfsablauf erschwert jedoch die Weiterverwertung früherer Simulationsmodelle.

In der Spezifikationsphase wird eine grobe Partitionierung vorgenommen, die mit Blöcken endet, die durch Machbarkeitsstudien evaluiert werden.

Wenn sich die Simulationsumgebungen nach der Machbarkeitsstudie ändern, z. B. durch den Wechsel von MatLab zu VHDL-AMS, werden die Ergebnisse der Machbarkeitsstudien in zusätzlichen informellen Spezifikationen zusammengefasst. Simulationsmodelle, die eine Entscheidungsgrundlage für diese erweiterten Pflichtenhefte darstellten, stehen jedoch in den folgenden Entwurfsphasen nicht mehr zur Verfügung. Die neuen Pflichtenhefte können überspezifiziert oder sogar widersprüchlich sein, da sie anstelle der ursprünglichen Gesamtsystemspezifikation treten, ist eine nachträgliche Verbesserung der zusätzlichen Spezifikationen erschwert.

Ein durch simulierbare Modelle unterstützter Entwurfsablauf ist somit von Vorteil. Um dieses Ziel zu erreichen, wird ein Entwurfsprozess angestrebt, in dem die simulative Validierung von der Machbarkeitsstudie bis zur Prototypen-Evaluierung ein zentrales Ele-

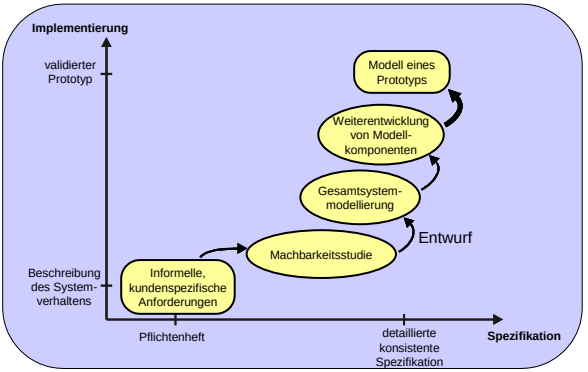


Abbildung 4.6: Durchgehender Entwurfsablauf mit der zusätzlichen Entwurfsphase der Gesamtsystemmodellierung

ment ist. Ein signifikater Vorteil entsteht dabei, wenn die Simulationsmodelle der verschiedenen Entwicklungsphasen so aufgebaut werden, dass sie weiterverwendet werden können.

Die Lücke zwischen den Machbarkeitsstudien und der Entwicklung der Einzelkomponenten wird durch ein Gesamtsystemmodell (Abb. 4.6) geschlossen. Das simulierbare Gesamtsystemmodell ergänzt die informelle Spezifikation des Systems und ermöglicht eine flexiblere Arbeitsweise. Im Idealfall erlaubt der so erweiterte Entwurfsablauf eine frühzeitige Aufdeckung von Widersprüchen und kann auch bei der Beseitigung von zu restriktiven Block-Spezifikationen hilfreich sein.

4.2.2 Gesamtsystemsimulation

Die Gesamtsystemsimulation wird als *experimentelle Entwurfsvalidierung* von der Spezifikationsphase bis zur Realisierung eines Prototypen genutzt.

Deshalb ist es notwendig eine gemeinsame Simulation von Modellen aus unterschiedlichen Entwicklungsstadien zu ermöglichen, bzw. Modelle aus verschiedenen Abstraktionsebenen gemeinsam zu simulieren.

Die gemeinsame Simulation von Modellen unterschiedlicher Modellklassen erfordert ein Simulationssystem, in dem mehrere Simulationskerne gekoppelt werden; wobei bei der Gesamtsystemmodellierung auf eine geeignete Einbettung der abstrakteren Modelle geachtet werden muss.

Für einen transparenten Übergang zwischen den Abstraktionsebenen wird ein „Zwiebelschalenmodell” mit abstrakten Kernen und Schnittstellen-Schichten benötigt. Eine

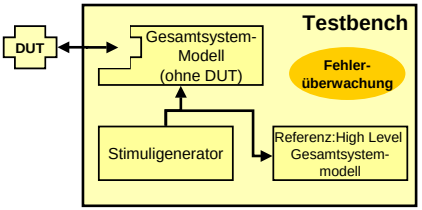


Abbildung 4.7: Aufbau einer Testbench unter Einbeziehung eines vorhandenen Gesamtsystemmodells

Schnittstellen-Schicht konkretisiert die abstrakten Daten des Kerns. Dazu wird ein Teil des Low-Level Modellwissens benötigt, da die entsprechenden Modelleigenschaften im abstraktem Kern nicht dargestellt werden können.

Desweiteren bietet es sich an, ein Gesamtsystemmodell bei der Validierung der einzelnen Teilsysteme zu nutzen. Der Aufbau einer Testbench (siehe Abb. 4.4) kann somit weiter vereinfacht werden: Das Referenzmodell kann evtl. durch eine abstrakteres Modell des DUT ersetzt werden und die Stimuliaufbereitung kann durch Bestandteile eines Gesamtsystemmodells durchgeführt werden (Abb. 4.7).

4.3 Konzeption einer heterogenen Simulationsumgebung

Um Modelle des Gesamtsystems bei der Validierung von einzelnen Komponenten verwenden zu können, wird eine heterogene Simulationsumgebung benötigt, die eine ausreichende Ausführungsgeschwindigkeit zur Verfügung stellt. In dieser Simulationsumgebung sollen einerseits Modelle der α-Klasse, β-Klasse und γ-Klasse automatisiert ausgeführt werden können. Zum anderen soll eine interaktive Simulation ermöglicht werden, bei der dem Benutzer die Simulationsergebnisse mit einer einfach zu interpretierenden grafischen Ausgabe veranschaulicht werden. Um die Validierung von ereignisdiskreten Modellen in der Komplexität üblicher digitaler integrierter Schaltungen zu ermöglichen, wird eine FPGA Karte als *Hardwareemulator* eingesetzt.

Diese Simulationsumgebung ist ein Komponentenmodell, dessen Konzeption eine weitgehend freie Programmierbarkeit ermöglicht. Für das in Kapitel 5 vorgestellte Anwendungsbeispiel existiert eine spezialisierte Implementierung der Simulationsumgebung, die im Folgenden zur Veranschaulichung der Möglichkeiten der Simulationsumgebung genutzt wird.

4.4 Anforderungen

Das Simulationssystem muss für eine Gesamtsystemsimulation und zur Validierung der einzelnen Komponenten während der Komponentenentwicklung geeignet sein. Daraus ergeben sich folgende Anforderungen:

- Für eine Komponente müssen mehrere Beschreibungen existieren können, die wahlweise in unterschiedlichen Modellen eingesetzt werden können.
- Systemmodelle müssen aus Teilmodellen unterschiedlicher Modellklassen zusammengesetzt werden können.
- Die Modellbildung soll mit üblichen Simulations-, bzw. Programmiersprachen erfolgen können.
- Digitale Komponenten sollen in einer Hardwareemulation ausgeführt werden können.
- Eine verteilte Simulation von zeitkausal gekoppelten Teilsystemen soll unterstützt werden. D. h. es soll möglich sein Teilsysteme, die nur über Zustandsgrößen der α-Klasse verbunden sind, auf verschiedenen Computern auszuführen.
- Neben der automatisierten Validierung soll eine frei definierbare grafisches Benutzerschnittstelle zur Visualisierung der Simulationsergebnisse zur Verfügung stehen.

Die Zeitdauer für die Berechnung eines Modells wächst überproportional mit der Komplexität der Modellbeschreibungen, so dass eine Gesamtsystemsimulation nur effektiv eingesetzt werden kann, wenn der Berechnungsaufwand für einige der enthaltenen Komponenten reduziert wird. Aus dieser trivialen Erkenntnis ergibt sich die Notwendigkeit, dass für einen Simulationslauf ein Gesamtsystemmodell aus mehreren, unterschiedlich komplexen Komponentenmodellen mit geringem Aufwand zusammenstellbar sein muss. Neben der Komplexität des Modells ist die Modellklasse (siehe Kapitel 2) entscheidend für den Zeitbedarf einer Simulation: Für ein Modell der γ-Klasse muss ein Simulationsalgorithmus verwendet werden, der implizite Gleichungen und Differentialgleichungen lösen kann, während für Modelle der β-Klasse ein gerichteter Signalfluss vorgegeben ist. Modelle der α-Klasse können mit einem noch einfacheren Ausführungsparadigma simuliert werden, da in diesen Modellen lediglich die Reihenfolge der Berechnungsschritte, aber nicht der genaue Zeitpunkt festgelegt ist.

Die Beschreibung der Modelle soll mit existierenden Simulations-, bzw. Programmiersprachen erfolgen, wobei möglichst weit verbreitete Sprachen eingesetzt werden sollen. So bietet sich für die Modelle der α-Klasse eine auf der Programmiersprache C basierende Beschreibungssprache an. Für Modelle der β-Klasse ist VHDL geeignet und für

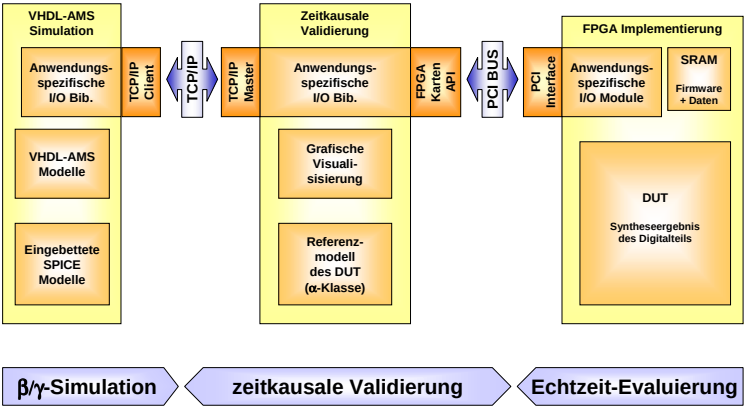


Abbildung 4.8: Konzeption des heterogenen Simulationssystems

Modelle der γ-Klasse ist je nach Anwendungsgebiet eine domänenspezifische Modellierungssprache wie SPICE, oder auch eine domänenübergreifende Modellierungssprache wie VHDL-AMS geeignet.

Um die Validierung von Systemen mit umfangreichen digitalen Komponenten zu beschleunigen, soll die Simulation von digitalen Teilschaltungen in einen FPGA ausgelagert werden können. Dabei ist eine schnelle Anbindung des FPGA an den softwarebasierten Teil der Simulationsumgebung wichtig. Von den in einem PC zur Verfügung stehenden Schnittstellen ist der PCI-Bus mit einer Datenübertragungsrate von bis zu 130 MByte/s am besten für die Aufgabe geeignet, so dass eine PCI-Steckkarte mit FPGA als Hardwareemulator verwendet werden kann.

Um die Bewertung der Simulationsergebnisse zu erleichtern, wird das Simulationssystem um eine programmierbare grafische Benutzerschnittstelle ergänzt, die mit einem üblichen *Rapid Application Development Tool* (RAD Tool) projektbezogen erweitert werden kann. Visual Basic [69] von Microsoft und Delphi [11] von Borland sind weit verbreitete RAD Tools, die diese Aufgabe erfüllen können. In Visual Basic wird die Programmbeschreibung zur Laufzeit interpretiert, so dass die Ausführungsgeschwindigkeit gegenüber Delphi langsamer ist. Programmbeschreibungen in Delphi werden im Vorfeld übersetzt und sind danach direkt ausführbar. Beide RAD Tools sind unter MS Windows lauffähig, wobei es für Delphi ein Linux-Pendant gibt (Kylix [12]), so dass eine Umstellung von Delphi-Beschreibungen auf eine unter Linux ausführbare Beschreibung möglich ist.

Abb. 4.8 zeigt den ausgewählten Lösungsansatz, der die obigen Anforderungen erfüllt. Zur Beschreibung von Modellen der α-Klasse wurde zum einen die Programmiersprache C ausgewählt, zum anderen können kleinere Beschreibungen der α-Klasse mit Hilfe

von *VHDL functions* oder *VHDL procedures* realisiert werden. Zur Beschreibung von Modellen der β-Klasse und der γ-Klasse wird VHDL-AMS eingesetzt. Diese Sprache bietet den großen Vorteil, dass der Übergang zwischen diesen zwei Modellklassen innerhalb eines vorgefertigten kommerziell erhältlichen Simulators stattfindet, so dass ein heterogenes Modell effizient simuliert werden kann. Zusätzlich können Strukturmodelle der γ-Klasse in SPICE-Notation mit in die Simulation aufgenommen werden. Da die Modellierung mit SPICE-Netzlisten beim Entwurf analoger Schaltungen noch immer unverzichtbar ist, ist eine Einbindung dieser Modellierungssprache in die heterogene Simulationsumgebung für einen durchgängigen simulationsgestützten Entwurfsablauf von zentraler Bedeutung.

Die Einbindung von SPICE-Netzlisten in ein übergeordnetes VHDL-AMS Modell wurde mit Hilfe des ADvance MS Simulators [67] realisiert. Dieser Simulationskern wird über eine interne Programmierschnittstelle mit einem *TCP/IP Socket* erweitert, über den zeitkausal Zustandsgrößen der α-Klasse übertragen werden können.

4.5 Die FPGA-Karte

Die Validierung von Systemen, die umfangreiche digitale Komponenten enthalten, kann durch ein Einsatz von spezieller Hardware, die einen Teil der Funktionalität der digitalen Komponenten nachbildet, um mehrere Größenordnungen beschleunigt werden.

Bei der hier verwendeten FPGA Lösung handelt es sich um die PCI-Karte microEnable der Firma Silicon Software GmbH. Diese ist mit mehreren Xilinx FPGAs lieferbar, auf dem eingesetzten Modell wird ein XC4085XLA verwendet. Der FPGA stellt 3136 CLBs zur Verfügung, die ein Design von etwa 55.000 bis 180.000 Gatteräquivalenten Größe ermöglichen. Der Takt des FPGA kann in 1 MHz Schritten eingestellt werden, so dass ein zur Latenz der kombinatorischen Pfade passender Takt gewählt werden kann. Auf der PCI-Karte ist zusätzlich 36x512 kBit SRAM integriert, das direkt an den FPGA angeschlossen ist. Aufgrund der Latenzzeit des SRAM und der Leitungsverzögerungen kann dieser Speicher mit einer maximalen Taktrate von 50 MHz ausgelesen werden. Außerdem ist der FPGA über einen 32 Bit breiten Bus mit einem lokalen PCI-Baustein verbunden, der die Kommunikation zwischen FPGA und dem PCI-Bus koordiniert. Durch diesen Baustein können unterschiedliche Taktraten des FPGAs und des PCI-Buses (33 MHz) realisiert werden.

Die Simulation von VHDL-Strukturbeschreibungen nach der Logiksynthese ist ein vielfaches langsamer, als deren Ausführung in einem FPGA, aber auch die Berechnung eines Modells der α-Klasse kann in der FPGA-Karte vorteilhaft sein: Im Rahmen des Projekts *Real Time Image Processing based on FPGA Hardware Acceleration* der Stiftung CAESAR[3] [44], wurde die Berechnung eines Level Set Verfahrens zur Bildverarbeitung in dieser FPGA-Karte implementiert. Die Referenzimplementierung ist eine

[3] Center of Advanced European Studies and Research, Bonn

Beschreibung des Level Set Verfahrens in der Programmiersprache C. Diese Beschreibung implementiert ein Modell der α-Klasse. In der FPGA-Karte wurde dieses Modell mit den Beschreibungsmöglichkeiten der β-Klasse als synthetisierte VHDL Beschreibung ausgeführt. Die optimierte FPGA Lösung berechnet 2000 Iterationen pro Sekunde, während ein Pentium III Rechner mit dem C-Programm nur 100 Iterationen pro Sekunde berechnen konnte.

Der Xilinx FPGA der PCI-Karte eignet sich gut für das hier implementierte Simulationssystem, da die integrierten FlipFlops über ein *ClockEnable* Signal gezielt in einen Ruhezustand versetzt werden können. Mit diesem Hilfsmittel kann die zu testende Schaltung in einem 'Einzelschrittmodus' ausgeführt werden. Jedes FlipFlop des FPGAs hat einen eigenen ClockEnable Eingang, so dass gezielt bestimmte Teilmodelle aktiviert, bzw. schlafengelegt werden können. Im rechten Drittel von Abb. 4.8 ist der Aufbau der FPGA Schaltung skizziert. Für die Anwendung in dem heterogenen Simulationssystem wird verlangt, dass die Kontrolle über die ClockEnable Signale einem der *Anwendungsspezifischen I/O Module* (AIM) überlassen wird. In dieser Komponente wird festgelegt, wie viele Takte die zu testende Einheit (*Device Under Test* (DUT)) aktiv sein darf, bevor ein Datenabgleich über den PCI-Bus stattfindet. Durch ein solches im FPGA integriertes AIM können unterschiedlichste 'Breakpoints' realisiert werden. Die Ausführung des DUT kann unterbrochen werden, wenn ein Datenwort einen bestimmten Wert annimmt, wenn externe Stimulidaten benötigt werden, oder auch wenn das DUT einen bestimmten Zustand in einem seiner endlichen Automaten erreicht. In Kapitel 5 wird das Anwendungsbeispiel *v*DC vorgestellt, in dem diese Mechanismen zum Test von Dekompressionsalgorithmen und des DUT eingesetzt werden. Abb. 5.6 auf Seite 87 zeigt das dazugehörende grafische Benutzerinterface, mit drei der möglichen Breakpoint-Mechanismen: *Run Once* stoppt, wenn die interne Zustandsmaschine in den 'Sleep'-Zustand wechselt. *Run Until PC* stoppt die Ausführung vor der Ausführung des Befehls an Adresse PC und *Enable Single Step* aktiviert das DUT für jeweils nur einen Takt.

Das integrierte lokale SRAM kann zum Zwischenspeichern von Daten genutzt werden, die über den PCI-Bus übertragen werden müssen. Das ermöglicht einen Burst-Transfer auf dem PCI-Bus, der eine drei- bis vierfach höhere Datenübertragungsrate ermöglicht als Einzeltransfers. Desweiteren kann der Speicher vom DUT zur Speicherung von internen Daten verwendet werden. Mit Hilfe eines AIM können diese Daten ebenfalls über den PCI-Bus übertragen werden, so dass weitreichende Möglichkeiten zur Fehlersuche zur Verfügung stehen. Diese können jedoch nur genutzt werden, wenn ein geeignetes Benutzerinterface (s. Kapitel 5) zur Verfügung steht, mit dem der Benutzer gezielt auf die Daten zugreifen kann.

4.6 Refactoring für Test, Wartbarkeit und Erweiterbarkeit

Abstrakte Verhaltensmodelle, wie die Gesamtsystemmodelle, deren Implementierung nicht mit Hilfe eines vollautomatischen Syntheseverfahrens erzeugt werden kann, müssen von Entwurfsingenieuren manuell weiterentwickelt werden. Bei vielen typischen Komponenten eines heterogenen Systems sind solche Modelle eher die Regel als die Ausnahme. Aufgrund der stetig wachsenden Komplexität der Systeme ist eine Wiederverwendung von existierenden Komponenten, bzw. von existierenden Modellen der Komponenten, sinnvoll. Dieses Ziel kann am besten erreicht werden, wenn das Gesamtsystem systematisch auf dieses Ziel hin partitioniert wird. Für eine effiziente Arbeitsweise müssen die vielfältigen Aufgaben des Entwurfsingenieurs in kleine, überschaubare und testbare Teilaufgaben zerlegt werden. Um dieses Ziel zu erreichen, wird ein Ansatz aus dem Softwareentwurf verwendet:

Kent Beck prägte 1997 in [4] im Zusammenhang mit Fragestellungen aus dem Software-Engineering den Begriff *Refactoring*. Refactoring ist ein Oberbegriff für Arbeitsschritte, in denen ein Softwareprogramm so geändert wird, dass sich die interne Struktur verbessert - ohne dass das externe Verhalten des Programms verändert wird. Es ist eine disziplinierte Vorgehensweise um existierenden Programmcode so zu vereinfachen, dass die Wahrscheinlichkeit dabei Fehler zu erzeugen, minimiert wird. Im Softwareentwurf kann Refactoring zum Beispiel genutzt werden, um für nachträgliche Erweiterungen die Architektur des Programms zu überarbeiten.

> *Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*
>
> (Martin Fowler, [20, S. 53])

In einem textbasierten Entwurfssystem, z. B. mit Modellierungssprachen wie VHDL-AMS, sind Parallelen zum Softwareentwurf offensichtlich: Beim Top-Down Entwurf heterogener Systeme werden abstrakte Modelle iterativ konkretisiert, bis eine realisierbare Beschreibung vorliegt. Die Aufgabe, die interne Struktur eines Modells zu modifizieren, ohne das von außen sichtbare Verhalten zu ändern, ist ein wichtiger Bestandteil der Entwicklungsarbeit.

Aufgrund dieser verwandten Arbeitsvorgänge bietet es sich an, auf der Basis des Software-Refactorings, eine Anleitung für das Refactoring in der Modellbildung heterogener Systeme zu entwickeln. Eine Methodik für dieses Anwendungsgebiet unterscheidet sich jedoch grundsätzlich von den etablierten Refactoring-Ansätzen, da die auf objektorientierte Programme zugeschnittenen Methoden auf die Modellbildung heterogener Systeme nur eingeschränkt übertragen werden können. Zum einen sind Modellierungssprachen, wie z. B. VHDL-AMS, nicht objekt-orientiert, zum anderen müssen zahlreiche Komponenten des heterogenen Systems im Verlauf der Produktentwicklung auf

eine Strukturbeschreibung mit vordefinierten Bibliothekselementen zurückgeführt werden.

Das Optimierungsziel beim Refactoring wird im Hinblick auf das neue Anwendungsgebiet erweitert. Im Softwareentwurf ist Refactoring eine zusätzliche Arbeitsphase mit dem Optimierungsziel das Programm leichter verständlich zu gestalten - und ist damit beispielsweise deutlich abgegrenzt von einer Performanz-Optimierung, bei der Programmteile auf Ausführungsgeschwindigkeit optimiert werden.

Im Entwurfsprozess heterogener Systeme ist die iterative Konkretisierung von Modellen ein zentraler Bestandteil, dessen methodische Umsetzung in überschaubaren und testbaren Arbeitsschritten eng mit dem ursprünglichen Gedanken des Refactorings verbunden ist. Das Refactoring heterogener Systeme soll deshalb beide Zielsetzungen beinhalten. Die daraus resultierende Arbeitsmethodik dient sowohl zur Restrukturierung und Vereinfachung vorhandener Modelle, als auch als Hilfe zur Erstellung von konkretisierten Modellen.

Definition 4.1 (Refactoring heterogener Systeme): Änderungen in der Beschreibung eines Modells, die das gewünschte Verhalten des Gesamtsystems nicht verändern, jedoch in Hinblick auf eine der folgenden Kriterien von Vorteil ist:

- Verständlichkeit der Modellbeschreibung
- Erweiterbarkeit des Modells
- Abbildung der Modellbeschreibung auf konkretere Modelle (Implementierung/Synthese)

Eine Refactoring-Methode stellt jeweils eine überschaubare Arbeitseinheit dar, die auf ein getestetes Modell angewendet wird. Ein Katalog an Refactorings (siehe Kapitel A) hilft dabei, die Umstrukturierungsmöglichkeiten so aufzubereiten, dass deren Anwendung zu einer einfachen Routinearbeit wird.

Eine methodische Anwendung des Refactoring setzt voraus, dass die neu entstandenen Modelle vor der Weiterverwendung einen erneuten Validierungstest durchlaufen. Als Konsequenz entsteht ein Arbeitsfluss, in dem sowohl eine Komponente entwickelt, als auch deren Testbench aktualisiert, bzw. erweitert wird. Der Entwurf eines heterogenen Systems besteht nicht nur aus dem Refactoring eines Gesamtsystemmodells: Die Komponenten dieses Gesamtsystemmodells müssen fortwährend mit neuen Eigenschaften und Fähigkeiten erweitert werden. Das Refactoring ermöglicht hierbei eine Trennung von kreativen Prozessen und der notwendigen Routinearbeit.

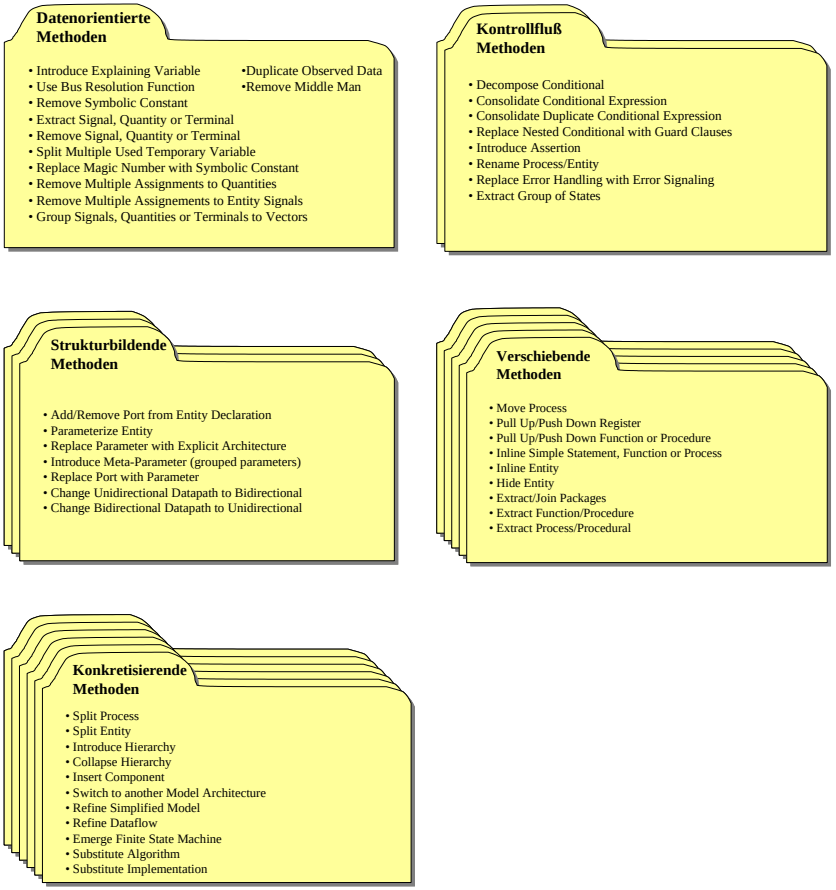


Abbildung 4.9: Übersicht über die Refactoring-Methoden

```
architecture Original of MyStimuli is
  quantity P across P_Flow through Pressure_Sensor;
begin
  if (now < 3.0 us) use
    p == 2.5*time2real(now/3.0 us);
  elsif (now < 28.0 us) use
    p == 2.5;
  elsif (now < 34.0 us) use
    p == 2.5*(1-time2real((now-28.0 us)/6.0 us));
  else
    p == 0.0;
  end use;
end architecture Original;
-------------------------------------------------------------------------------
architecture Refactored of MyStimuli is
  function pressure (aTime : time) return real is
  begin
    if (aTime < 3.0 us) then
      return(2.5*time2real(aTime/3.0 us));
    elsif (aTime < 28.0 us) then
      return(2.5);
    elsif (aTime < 34.0 us) then
      return(2.5*(1-time2real((aTime-28.0 us)/6.0 us)));
    end if;
    return(0.0);
  end function pressure;

  quantity P across P_Flow through Pressure_Sensor;
begin
  p == pressure(now);
end architecture Refactored;
```

Beschreibung 4.1: In obigem Stimuligenerator wurde das *if* ... *use* Konstrukt durch einen Funktionsaufruf ersetzt.

Typische Anwendungsszenarien für Refactoring

Am Beispiel der Modellierungssprache VHDL-AMS werden nun typische Anwendungsszenarien für Modelltransformationen mittels Refactoring-Methoden vorgestellt. Die Refactoring-Methoden werden in fünf Gruppen aufgeteilt, die im Hinblick auf die Anwendungsszenarien gebildet werden (siehe Abb. 4.9).

4.6.1 Datenorientierte Methoden

Diese Gruppe enthält Methoden zur Optimierung und Vereinfachung des Datenflusses innerhalb einer Komponente. Einige Methoden ermöglichen eine kompaktere Notation des Modells, beispielsweise durch die Bildung von Signalvektoren, oder durch die gezielte Suche und Entfernung nach Zustandsgrößen und Deklarationen, die in der

aktuellen Fassung des Komponentenmodells nicht (mehr) verwendet werden. Die Zustandsgrößen werden in VHDL-AMS mit Hilfe von *Signalen*, *Quantities* und *Terminals* modelliert.

Die meisten Methoden dienen jedoch der Erweiterung eines Verhaltensmodells: Einerseits kann ein Modell einfacher zu verstehen sein, wenn sinnvoll benannte Variablen für Zwischenergebnisse oder symbolische Bezeichner für numerische Konstanten eingeführt werden. Andererseits sind mehrfach verwendete Zwischenvariablen oder in mehreren Fallunterscheidungen zu beschreibende Ausgangssignale übliche Fehlerquellen. Bei der Umformung des in Beschreibung 4.1 beschriebenen zeitkontinuierlichen Modells wird durch die Umformung zusätzlich der eindeutige Signalfluss der zeitabhängigen Druckquelle dargestellt.

Diese Refactorings enthalten jedoch auch Methoden für spezielle Problemstellungen beim Entwurf ereignisdiskreter Modelle: In Modellen getakteter digitaler Schaltungen mit mehreren unabhängigen Clockdomänen kann es beim lesenden Zugriff auf Signale der jeweils anderen Clockdomäne Fehler geben, wenn die Signallaufzeiten realitätsnah beschrieben werden. Das Refactoring *Duplicate Observed Data* ermöglicht eine frühzeitige Beseitigung dieser Fehlerquelle. Elektrische Modelle, in denen verschiedene Signalquellen auf einen gemeinsamen Datenbus geschaltet werden, können mit expliziten Multiplexern, mit Tristate-Logik, mit Open-Collector-Logik oder ähnlichem realisiert werden. Die Kodierung eines expliziten Multiplexers ist zwar eine naheliegende Beschreibung; sie muss jedoch für jeden neuen Busteilnehmer erweitert werden. Um dieses Problem zu umgehen, können *Bus-Resolution Functions* eingesetzt werden. Diese ermöglichen eine problemlose Beschreibung alternativer Busmodelle wie den Tristate-Ansatz.

4.6.2 Kontrollflussorientierte Methoden

Diese Refactoring-Methoden dienen zur Verbesserung der Qualität von Verhaltensmodellen mit umfangreichem *Kontrollfluss*. In diesen Modellen sind sie zum Beispiel bei der Aufbereitung komplexer sequentieller Prozesse hilfreich. Diese können vereinfacht werden, indem man Teile des Prozesses in zusätzliche Funktionen oder Prozeduren auslagert. So kann eine boolesche Funktion eine komplexe Bedingung in einer *if*-Abfrage ersetzen. Gemeinsame Ausdrücke in sequentiell geordneten *if*-Konstrukten können in einer übergeordneten Abfragen zusammengefasst werden und umfangreiche bedingt auszuführende Datenpfade können in einer Prozedur beschrieben werden, um den Kontrollfluss übersichtlicher darzustellen. In einigen Fällen ist es auch möglich mehrere *if*-Abfragen zusammenzufassen, beispielsweise wenn eine bestimmte Anweisung als Reaktion auf eines von mehreren Ereignissen ausgeführt werden soll.

Ebenso gehören alltägliche Handlungen, wie das Umbenennen eines Prozesses oder einer Komponente zu diesen Refactorings. Wenn unklar ist, wie oft eine Komponente eingesetzt ist, kann es sehr nützlich sein, wenn ein Test existiert, in dem das gesamte

System, mit allen Instantiierungen dieser Komponente, eingesetzt ist. Dieser Test findet Inkonsistenzen automatisch, so dass der Entwurfsingenieur entlastet wird.

Eine Optimierung der Fehlerbehandlung kann sowohl den Kontrollfluss verbessern, als auch im Hinblick auf eine spätere Synthese von digitalen Teilkomponenten sinnvoll sein. So bietet es sich in einigen Fällen an, eine Fehlerüberprüfungen durch *assert* Anweisungen zu ersetzen. So kann die Simulation für ungültig erklärt werden, anstatt eine aufwendige Korrektur eines nur in der Simulation möglichen Fehlerfalls durchzuführen. Beschreibung 4.2 zeigt am Beispiel eines Komparators das Umformen einer Fehlerbehandlung in eine Fehlermeldung. Der Komparator funktioniert nur korrekt, wenn eine der Eingangsspannungen über 0.7V liegt. Die implizite Fehlerbehandlung im Originalmodell wird durch das *Error* Signal übernommen.

Letztendlich gehört die Zerlegung eines endlichen Automaten in zwei kleinere, überschaubare endliche Automaten ebenfalls in diese Kategorie. Bei der Erweiterung der Komponente um neue Funktionalität wächst oft auch die Anzahl der benötigten Zustände der Komponente an. Lange sequentielle Prozesse sind schwer zu verstehen und dementsprechend anfällig für versteckte Modellierungsfehler. Oftmals kann ein Teil der Funktionalität ausgelagert werden, in dem beispielsweise für ein aufwendiges Protokoll beim Datenaustausch ein zusätzlicher endlicher Automat eingeführt wird.

4.6.3 Strukturorientierte Methoden

Die strukturorientierten Refactoring-Methoden dienen zur Umgestaltung der Struktursicht auf Komponentenmodelle, d. h. mit diesen Methoden sind Umgestaltungen an Komponentendeklarationen möglich, die jedoch das Verhalten der Hauptkomponente nicht verändern dürfen.

Diese Gruppe der Refactorings ermöglicht Änderungen an den Komponentenschnittstellen von Teilkomponenten, indem beispielsweise zusätzliche Signale, Quantities oder Terminals an eine Unterkomponente weitergereicht werden. Eine solche Umgestaltung kann notwendig sein, um eine Erweiterung des Modells vorzubereiten.

In Hinblick auf Wiederverwendbarkeit von Komponenten können Erweiterungen einer Modelldeklaration durch *Generics*, d. h. Parameter, die beispielsweise die Breite der Eingangsvektoren oder die interne Rechengenauigkeit festlegen, sinnvoll sein. Wenn die Anzahl der Parameter steigt, ist oft eine Zusammenfassung einiger Parameter zu einem Meta-Parameter möglich, so dass die Lesbarkeit der Modelle verbessert wird. In dem Anwendungsbeispiel aus Beschreibung 4.3 wird die Anzahl der Parameter in der Komponentendeklaration reduziert, indem die benötigte Funktionalität in ein *Package* ausgelagert wird.

Andererseits kommt es ebenso vor, dass solche Konfigurationsparameter genutzt werden, um zwischen mehreren voneinander unabhängigen Verhaltensbeschreibungen umzuschalten. Beispielsweise um von einem vereinfachten Systemmodell zu einem genauen, hardwarenah beschriebenen Modell umzuschalten. Es ist dann zu überlegen, ob

```
architecture Orginal of Comparator is
quantity deltaU across T_plus to T_minus;
signal Dout : std_logic;
begin
  a2d: process (deltaU'above(-0.1),
                deltaU'above(0.1),
                T_plus'reference'above(0.7),
                T_minus'reference'above(0.7)) is
  begin
    if ((T_plus'reference < 0.7)
        and (T_minus'reference < 0.7)) then
      Dout <='U';
    elsif (deltaU'above(0.1)) then
      Dout <= '1';
    elsif not(deltaU'above(-0.1)) then
      Dout <= '0';
    end if;
  end process a2d;
end architecture Orginal;
--------------------------------------------------------------------------------
architecture Refactored of Comparator is
quantity deltaU across T_plus to T_minus;
signal Dout : std_logic;
begin
  a2d: process (deltaU'above(-0.1),deltaU'above(0.1)) is
  begin
    if (deltaU'above(0.1)) then
      Dout <= '1';
    elsif not(deltaU'above(-0.1)) then
      Dout <= '0';
    end if;
  end process a2d;
  error <= '0' when ((T_plus'reference'above(0.7)
                      or (T_minus'reference'above(0.7)))
           else '1';
end architecture Refactored;
```

Beschreibung 4.2: Das obige Beispiel zeigt die Anwendung des Refactorings *Replace Error Handling with Error Signaling*.

es nicht besser ist, mehrere Verhaltensmodelle zu erstellen, die über den *Architecture*-Namen unterschieden werden.

Die strukturorientierten Refactorings erfordern auch eine Überprüfung des Informationsgehalts der Signale und Terminals zwischen zwei Komponenten. Zwischen nicht synchronisierten Teilkomponenten sind Handshake-Protokolle notwendig. In einigen Fällen kann man jedoch nach einer Adaption der Taktnetze solche Handshake-Protokolle wieder vereinfachen, auch konstante Signale und Quantities[4] können i. d. R. ersetzt werden.

[4]Der Substratanschluss ist in vielen Modellen integrierter elektronischer Schaltungen ein Beispiel für ein Spannungssignal mit konstantem Wert.

```
entity Original_fpu1 is
  generic (
    manWidth : natural := 23;
    expWidth : natural := 8);
  port (
    Clk,Rst,En : in std_logic;
    arg1, arg2 : in  std_logic_vector(manWidth+expWidth downto 0);
    res        : out std_logic_vector(manWidth+expWidth downto 0));
end entity Original_fpu1;
entity Original_fpu_inv is
  generic (
    manWidth : natural := 23;
    expWidth : natural := 8);
  port (
    Clk,Rst,En : in std_logic;
    arg        : in  std_logic_vector(manWidth+expWidth downto 0);
    res        : out std_logic_vector(manWidth+expWidth downto 0));
end entity Original_fpu_inv;
-------------------------------------------------------------------------------
package public is
  use ieee.std_logic_1164.all;
  constant manWidth : natural := 23;
  constant expWidth : natural := 8;
  subtype  Tfloat_sign is std_logic;
  subtype  Tfloat_exp is std_logic_vector(expWidth-1 downto 0);
  subtype  Tfloat_man is std_logic_vector(manWidth-1 downto 0);
  subtype  Tfloat is std_logic_vector(expWidth+manWidth downto 0);
end package public;

use work.public.all;
entity refactored_fpu1 is
  port (
    Clk,Rst,En : in std_logic;
    arg1, arg2 : in  Tfloat;
    res        : out Tfloat);
end entity refactored_fpu1;

use work.public.all;
entity refactored_fpu_inv is
  port (
    Clk,Rst,En : in std_logic;
    arg        : in  Tfloat;
    res        : out Tfloat);
end entity refactored_fpu_inv;
```

Beschreibung 4.3: Mit Hilfe des Datentypen Tfloat wird die Komponentendeklaration vereinfacht.

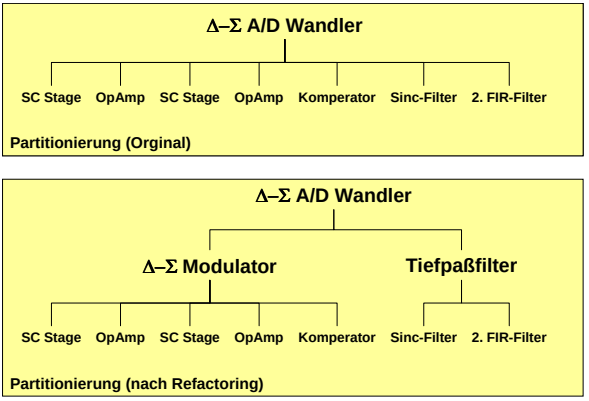


Abbildung 4.10: Partitionierung eines Δ-Σ A/D Wandlers

4.6.4 Verschiebende Methoden

Die verschiebenden Methoden beschreiben Refactorings, in denen ein Teil der Modellbeschreibung umsortiert wird. Prozesse, Funktionen und Speicherblöcke sollten den Komponenten so zugeordnet sein, dass eine nachvollziehbare Partitionierung erkennbar ist. Bei einem iterativen Entwurfsablauf, in dem ein System durch wiederholte Konkretisierung der Teilkomponenten entsteht, ist es hilfreich nach einigen Iterationen die vorhandene Partitionierung zu hinterfragen und etwaige Optimierungen vorzunehmen.

Wenn man die vorgegebene Partitionierung einer Komponente untersucht, kann man mehrere typische Optimierungsansätze finden. Einerseits gibt es umfangreiche, flache Partitionierungen, bei denen die Übersichtlichkeit durch eine hierarchisch gestaffelte Partitionierung verbessert werden kann (siehe Abb. 4.10). Andererseits gibt es Partitionierungen in denen die Teilkomponenten Funktionalität besitzen, deren Implementierung in anderen Teilkomponenten naheliegender wäre. Die Umformung der Teilkomponenten im Kontext der übergeordneten Komponente bietet den großen Vorteil, dass stets auch eine Konsistenzprüfung der Komponentenschnittstellen stattfindet.

Ein Prozess oder eine Komponente, deren Verhaltensbeschreibung trivial ist, wird entfernt. Eine solche Beschreibung kann beispielsweise durch die Verschiebung von Funktionalität in andere Beschreibungen entstehen. Das Verhalten der entfernten Komponente wird in der übergeordneten Komponente nachgebildet.

Zu den verschiebenden Methoden gehört auch die Umstrukturierung der Funktions-Bibliotheken, bei denen Bestandteile verschoben werden können, mehrere Bibliotheken zusammengefasst werden oder auch neue Bibliotheken gebildet werden.

```
entity cap is
  generic (Capacity : real := 1.0e-12);
  port (terminal T1, T2 : electrical);
end entity cap;

architecture ideal of cap is
quantity VCap across ICap through T1 to T2;
begin
  ICap ==  VCap'dot * Capacity;
end architecture ideal;

architecture substrat_coupled of cap is
quantity VCap across ICap through T1 to T2;
quantity VSub across ISub through T2;
begin
  ICap ==  VCap'dot * Capacity;
  ISub ==  VSub'dot * 0.9*Capacity;
end architecture substrat_coupled;
--------------------------------------------------------------------------------
architecture Original of MyEntity is
begin
  cap_1: entity work.cap(ideal)
    generic map (Capacity => 0.8e-12)
    port map (T1 => T1, T2 => T2);
  ...;
end architecture Original;
--------------------------------------------------------------------------------
architecture Refactored of MyEntity is
begin
  cap_1: entity work.cap(substrat_coupled)
    generic map (Capacity => 0.8e-12)
    port map (T1 => T1, T2 => T2);
  ...;
end architecture Refactored;
```

Beschreibung 4.4: In diesem Beispiel wird das Modell einer idealen Kapazität gegen eine Beschreibung eines Kondensators mit Substratkapazität ausgetauscht. Als Substratpotential wurde dabei das systeminterne Referenzpotential verwendet.

4.6.5 Konkretisierende Methoden

Die konkretisierenden Refactorings ermöglichen die methodische Durchführung der Arbeiten, die für die Abbildung von Modellen in konkretere Beschreibungen notwendig sind.

Ein Teil der Methoden ermöglicht die Neubildung einer Partitionierung durch Aufspaltung von Prozessen und Komponenten oder durch Umgruppierung von Unterkomponenten in neue Komponenten. Einige weitere Methoden sind notwendig, um implementierungsnahe Beschreibungen in das Gesamtmodell einzubinden. Dabei sind mehrere Ansätze möglich. Bei Komponenten, für die mehrere Beschreibungen (*Architectures*) existieren, kann die Modellbeschreibung ausgetauscht werden (siehe Beschreibung 4.4).

In anderen Komponenten können implementierungsnahe Teilkomponenten eingefügt werden, so dass die ursprüngliche Komponente größten Teils als Strukturnetzliste notiert werden kann. Zusätzlich sind Optimierungen in den Verhaltensmodellen möglich, indem eine Beschreibung um parasitäre Effekte ergänzt wird.

Desweiteren können mit diesen Refactorings aus einem algorithmischen Verhaltensmodell konkretere Beschreibungen gewonnen werden, indem implizite sequentielle Teilmodelle so umgeformt werden, dass zustandsbasierte endliche Automaten entstehen. Diese endlichen Automaten können dann z. B. mit einem Synthese-Programm weiterverarbeitet werden. Eine implementierungsfreundliche Umformung des Datenflusses kann durch Zerlegung der Berechnungsvorgänge in einfachere Grundoperationen erreicht werden, wobei es letztendlich auch notwendig sein kann, einen Berechnungsalgorithmus durch einen äquivalenten, aber besser zu implementierenden Algorithmus zu ersetzen.

Kapitel 5

Anwendung und Bewertung der Entwurfsmethodik

Die in Kapitel 3 vorgestellten Abstraktionsebenen und die Entwurfsmethodik aus Kapitel 4.1 dienen als Hilfestellung für den Systementwurf. Die Kombination der Refactoring Methoden aus Kapitel 4.6 mit den Entwurfsparadigmen ermöglicht eine kreative Arbeitsweise, die einen übersichtlichen Arbeitsfluss ermöglichen, ohne die Mächtigkeit der Modellierungsmöglichkeiten einzugrenzen. So kann sich der Entwurfsingenieur auf die Lösung konkreter Teilprobleme konzentrieren, anstatt sich in der Implementierung des Gesamtsystems zu verlieren.

Im Rahmen des Projekts *virtual Diving Computer* (*v*DC) wurde am Institut für Integrierte Schaltungen und Systeme der Technischen Universität Darmstadt von 1998 bis 2002 an der Modellierung und Validierung von Dekompressionscomputern gearbeitet [35, 49, 43, 56, 57, 80, 82]. Dieses Projekt wird als Designstudie zur Bewertung der vorgestellten Entwurfsmethodik verwendet. Ein Dekompressionscomputer überwacht den Stickstoffhaushalt des Tauchers in Echtzeit und warnt den Taucher rechtzeitig vor Gesundheitsproblemen durch Stickstoffübersättigung. Zusätzlich wird die Sicherheit des Tauchers erhöht indem geeignete Auftauchprofile vorgeschlagen werden.

5.1 Kurzbeschreibung des Anwendungsbeispiels *v*DC

Ein Dekompressionscomputer sollte möglichst klein, leicht und portabel sein. Idealerweise sollte das Gerät wie eine Armbanduhr am Handgelenk getragen werden können. Dementsprechend ist eine Miniaturisierung des Produktes durch spezialisierte hochintegrierte Komponenten gewünscht.

Der Dekompressionsrechner besteht im wesentlichen aus Sensoren zur Erfassung der Umwelt des Tauchers, einer digitalen Verarbeitungseinheit zur Berechnung des Dekompressionsalgorithmus, sowie Schaltkreisen zur Sensorsignalaufbereitung und Digitalwandlung. Abb. 5.1 zeigt die Strukturierung des *v*DC, sowie eine zur Validierung des Systems verwendete Testbench. In dieser Testbench wird die Funktionalität der Sensoren, der A/D Wandler und der Signalkonditionierung überprüft.

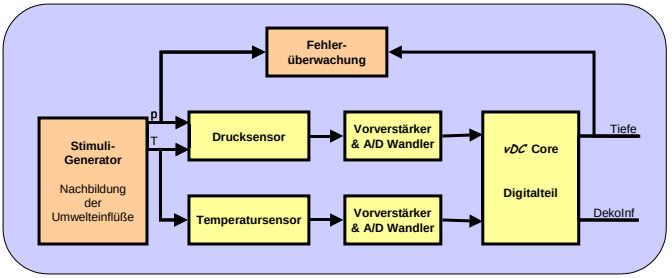


Abbildung 5.1: Partitionierung des Dekompressionsrechners und der Testumgebung

Konkret für das *v*DC Projekt wurden Entwicklungsarbeiten in den folgenden Arbeitsfeldern durchgeführt:

- *Bio-physikalisches System 'Taucher':* Erstellung von Testszenarien; Typische Anwendungsfälle müssen nachgebildet werden, um eine Validierung des Gerätemodells in einem möglichst frühen Stadium zu ermöglichen.
- *Analoge Schaltungsentwicklung:* Auswahl, bzw. Entwicklung der Sensoren und der dazugehörigen Analog/Digital-Wandler
- *Digitale Schaltungsentwicklung:* Entwicklung der digitalen Blöcke; der digitale Teil beinhaltet die Ansteuerungslogik für die Sensoren, einen Mikroprozessor mit spezifischer Fließkomma-Arithmetikeinheit zur Berechnung der Dekompressionsalgorithmen, sowie die Ansteuerlogik für die Ausgabeeinheiten.
- *Digitale Signalverarbeitung:* Entwicklung der Software für die Berechnung der Dekompressionszeiten.

Die Stickstoffsättigung im Körper des Tauchers soll durch den Dekompressionsrechner berechnet und überwacht werden. Dazu wird ein Modell benötigt, mit dem aus dem Umgebungsdruck und der Zusammensetzung der Atemluft die Stickstoffaufsättigung in den einzelnen Bestandteilen des menschlichen Körpers bestimmt werden kann. Die Analyse des bio-physikalischen Systems 'Taucher' ist deshalb ein zentraler Bestandteil der Entwicklung des *v*DC. Die Modellbildung basiert auf medizinischen Daten und einem Schichtmodell des menschlichen Körpers, das von A. A. Bühlmann erarbeitet wurde. In diesem Schichtmodell wird der Stickstoffgehalt in 16 Geweben mit Hilfe von Differentialgleichungen berechnet. Diese werden im Rahmen des *v*DC Projekts diskretisiert und mehrfach implementiert. Zum einen wird ein Referenzmodell für die Verwendung in einer Testbench erzeugt. Zum anderen wird der Algorithmus für die Berechnung mit einem applikationsspezifischen Mikroprozessor aufbereitet.

Neben der Modellierung der Sättigungsvorgänge wird ein geeigneter Stimulus benötigt. Beim Entwurf der Generatoren für die Sensorstimuli wird auf eine einfache, generische Struktur geachtet. Es werden einige triviale Stimuliverläufe benötigt, um die Funktion des Referenzmodells zu überprüfen. Hierfür eignet sich z. B. ein *Rechteckprofil*, d. h. ein Druckprofil, dessen Zeitdiagramm entlang eines Rechtecks verläuft. Ebenso werden algorithmische Modelle erzeugt, die eine realitätsnahe Beschreibung üblicher Tauchgänge, d. h. eine Modellierung auf Basis von Tauchtiefe und Gewässerstruktur, ermöglichen. Zusätzlich ist auch der Import von Tauchprofilen realer Tauchgänge und eine Echtzeitstimulierung durch eine grafische Benutzeroberfläche möglich.

Um die Stimulidaten zu verarbeiten, benötigt man ein Modell der Sensoren zur Druck- und Temperatur-Messung, sowie geeignete A/D Wandler, welche die Umgebungsinformationen erfassen. Um ein kompaktes Gerät zu ermöglichen, wird als Messsensor ein piezoresistiver Drucksensor auf Silizium-Basis gewählt. Solche Sensoren können in einem MEMS Prozess gemeinsam mit einer analogen Signalaufbereitung und einem A/D Wandler auf einem gemeinsamen Substrat hergestellt werden. Der eigentliche Sensor wird von einigen Halbleiterherstellern als fertige Makrozelle zur Verfügung gestellt [93]. Als Architektur für den A/D Wandler wird eine Delta-Sigma Architektur verwendet, mit dem Ziel eine Integration des Delta-Sigma Modulators (Δ-Σ Modulator) mit dem Drucksensor zu ermöglichen. Im Rahmen der Projektarbeit an der Technischen Universität Darmstadt wurde ein existierendes Sensorelement der AMSYS GmbH & Co. KG [1] vermessen und ein Δ-Σ Modulator in einem $0.8\mu m$ Mixed-Signal CMOS Prozess entwickelt, produziert und vermessen. Die Integration des Drucksensors und des Δ-Σ Modulator auf einem gemeinsamen Substrat ist auf Basis der Projektergebnisse möglich, wurde jedoch nicht mehr durchgeführt.

Der für die Verarbeitung der digitalisierten Sensorinformationen benötigte applikationsspezifische Digitalteil ist in VHDL implementiert und stellt neben den üblichen Datenfluss- und Kontrollflussbefehlen eine Fließkomma-Arithmetikeinheit, einen Interrupt-Eingang und Stromsparfunktionalität zur Verfügung.

Die Firmware für diesen Digitalteil ist eine echtzeitfähige Implementierung des diskretisierten bio-physikalischen Systems, die in einer eigens für dieses Projekt definierten Assemblersprache formuliert ist. Zusätzlich zu dieser Implementierung für den applikationsspezifischen Mikroprozessor wurde eine Implementierung in ANSI C durchgeführt, die sowohl unter Microsoft Windows, als auch unter Linux lauffähig ist. Die Implementierung in C ermöglicht mit geringem Aufwand simulative Untersuchungen bezüglich der benötigten Rechengenauigkeit und der maximalen Integrationsschrittweiten, so dass die Anforderungen an den applikationsspezifischen Mikroprozessor experimentell überprüft werden können.

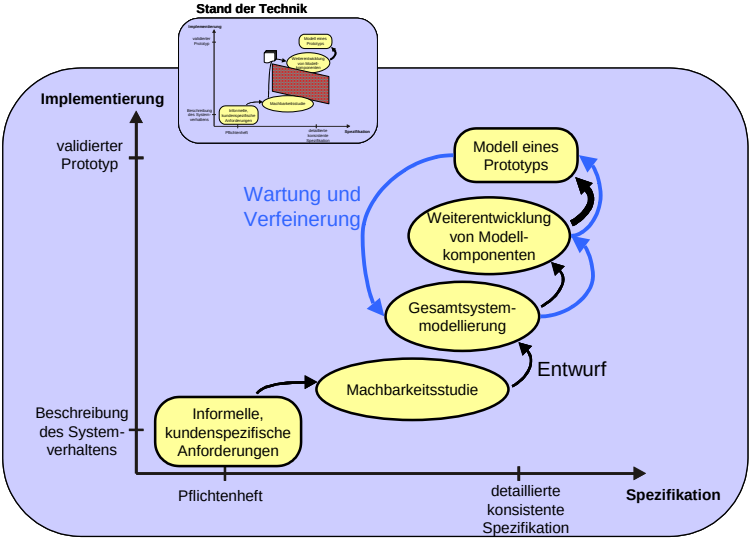


Abbildung 5.2: Entwurfsschritte und deren Nutzen im Hinblick auf Produktspezifikation und Implementierung

5.2 Entwurfsablauf im *v*DC-Projekt

Der *v*DC dient als Demonstrator für die Weiterentwicklung von Methoden zum schnellen Entwurf von qualitativ hochwertigen Prototypen. Durch die wachsende Komplexität heutiger Produkte und den Druck zu stets kürzeren Entwicklungszeiten wird die Validierung am virtuellen Prototypen zunehmend wichtiger [33]. Andererseits sind eingebettete Systeme wie der Dekompressionsrechner so komplex, dass eine Modellierung mit den etablierten Verfahren nur eingeschränkt zur Validierung verwendet werden kann. Um so wichtiger ist ein konsequent auf Qualitätssicherung ausgelegter Arbeitsfluss. Zentrale Bedeutung hat die Definition von Testkriterien in Form von Simulationsbeschreibungen, die eine automatisierte Validierung erlauben.

Der Entwicklungsablauf basiert wie in Kapitel 4 propagiert auf einer konsistenten Modellbildung, bei der die Simulationsmodelle jeweils zur Validierung der folgenden Entwicklungsstufe verwendet werden [35]. Der prinzipielle Entwicklungsablauf ist in Abb. 5.2 skizziert. Ausgehend von medizinischen Untersuchungen werden die Anforderungen an den *v*DC in Form von biophysikalischen Modellen formuliert. Mit diesen Vorgaben erarbeitet der Systemingenieur anschließend im Rahmen der Machbarkeitsstudie eine detailierte Spezifikation. Diese wird durch die Gesamtsystemmodellierung unterstützt. Dafür werden unterschiedliche Realisierungsvarianten evaluiert. Nach einer Bewertung

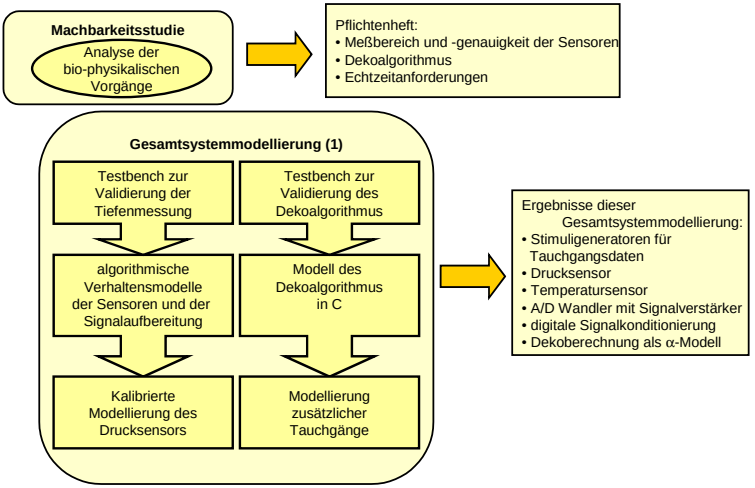


Abbildung 5.3: Arbeitsschritte bei der Gesamtsystemmodellierung des *v*DC

der Lösungsvarianten wird die Aufgabe in möglichst unabhängige Teilaufgaben partitioniert. Abb. 5.3 zeigt diese Arbeitsschritte für den Demonstrator *v*DC. Das erste Gesamtsystemmodell wird mit Hilfe von zwei Testbenches gewonnen. Die Messdatenerfassung wird mit einer Testbench zur Validierung der Tiefenmessung erstellt. Dabei wird die analoge Signalaufbereitung und die A/D Wandlung nur mit einem vereinfachten algorithmischen Modell beschrieben, während die Beschreibung des Drucksensors mit Hilfe von Datenblattangaben äquivalenter Drucksensoren zu kalibrierten Modellen weiterentwickelt werden. Der Dekompressionsalgorithmus wird in ein C-Programm umgesetzt, das mit Hilfe einiger einfacher Tauchgangsprofile validiert wird. Im Anschluss daran werden zusätzliche Tauchgangsprofile erstellt, die für die weiteren Entwicklungsschritte verwendet werden sollen.

Es entsteht so ein Gesamtsystemmodell, dass sowohl den Dekompressionsrechner beschreibt, als auch Stimuligeneratoren für eine Reihe von Tauchgängen beinhaltet. Diese Modelle ergänzen die Spezifikation insofern, als dass weiterentwickelte Komponenten stets ein zu diesem Gesamtsystemmodell äquivalentes Simulationsergebnis liefern müssen. Es wird jedoch im Fehlerfall sowohl eine Änderung der Komponente als auch eine Adaption des Gesamtsystemmodells zugelassen. Zur Simulation des Gesamtsystems wird die in Kapitel 4.5 vorgestellte Simulationsumgebung verwendet, wobei wie in Abb. 5.4 zu sehen ist, die TCP/IP Kopplung zwischen VHDL-AMS Simulator und dem RAD Environment genutzt wird, um das zeitkausale Modell des Dekompressionsalgorithmus und Stimuligeneratoren einzubinden.

Die auf die Gesamtsystemmodellierung folgenden Entwicklungsarbeiten werden i. d. R.

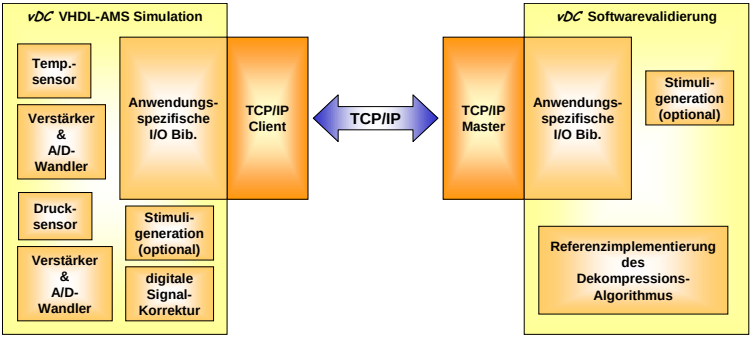


Abbildung 5.4: Infrastruktur zur Simulation des *v*DC-Modells aus der Gesamtsystemmodellierung

durch verschiedene Arbeitsgruppen bearbeitet. Die zu lösenden Aufgaben stammen aus verschiedenen Anwendungsdomänen, so dass eine Abstimmung des simulationsgestützten Entwurfsablaufs mit Hilfe der domänenunabhängigen Abstraktionsebenen aus Kapitel 3.2 sinnvoll ist. Da die Lösungsmöglichkeiten stark vom Funktionsumfang der anderen Komponenten bestimmt werden, ist eine konsistente Spezifikation wichtig. Diese wird durch das Gesamtsystemmodell sichergestellt. Die in Kapitel 2 definierten Modelleigenschaften können hierbei zur Erstellung kompatibler Modelle für die Gesamtsystemsimulation sinnvoll eingesetzt werden. Die Beschreibung des Gesamtsystems als simulierbares Modell ermöglicht mit einem darauf abgestimmten Arbeitsfluss einen effektiven Entwicklungsprozess, der durch Validierungsumgebungen für die benötigten Abstraktionsgrade unterstützt wird.

Während bei der Weiterentwicklung der Sensorsignalaufbereitung und deren Digitalwandlung die Simulationsumgebung aus Abb. 5.4 ausreichend ist, wird für die Validierung des Digitalteils und insbesondere für die Validierung der Firmware des *v*DC eine Erweiterung notwendig. Die Simulation des VHDL-Modells des anwendungsspezifischen Mikroprozessors im Register-Transfer-Level inklusive der dazugehörigen Firmware ist so rechenaufwendig, dass eine aussagekräftige Validierung des Dekompressionsalgorithmus durch diese Simulation nicht möglich wäre. Zur Evaluierung des digitalen Blocks wird deshalb von der in Kapitel 4.5 vorgestellten Validierungsumgebung die Hardwareemulation per FPGA eingesetzt (siehe Abb. 5.5). Durch Einbindung einer FPGA-Karte in den Validierungsablauf wird es möglich eine synthetisierte Implementierung des Digitalteils in Echtzeit zu testen. In diesem Validierungssystem wird die Referenzimplementierung des Dekompressionsalgorithmus in Ansi C verwendet, um die korrekte Funktion des *v*DC zu testen. Die VHDL-AMS Modelle werden in diesem Entwicklungsschritt durch idealisierte zeitkausale Modelle ersetzt. Während die Referenzimplementierung in einem Pentium III-System ausgeführt wird, läuft ein

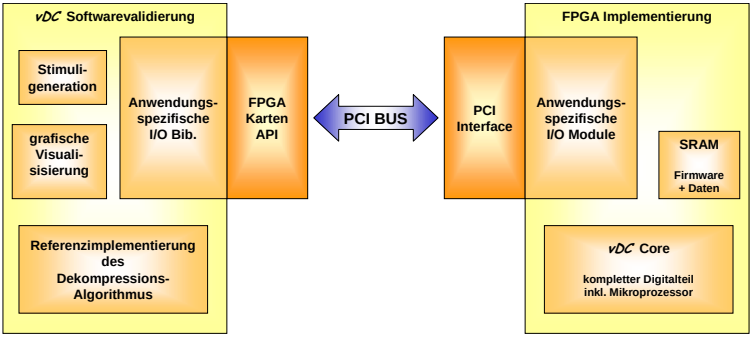


Abbildung 5.5: Infrastruktur zur Validierung der synthetisierten digitalen Komponenten

entsprechendes Assemblerprogramm in dem *v*DC Mikroprozessor, der in einen Xilinx FPGA abgebildet ist. In Abb. 5.6 ist ein Bildschirmfoto der grafischen Oberfläche des Validierungssystems zu sehen, das die für den Taucher relevanten Informationen aus der *v*DC-Emulation, d. h. der Softwareimplementierung, mit denen aus der Hardware-Berechnung vergleicht.

Das Gesamtsystem *v*DC stellt einen virtuellen Prototypen dar, der durch einen vollständig simulationsgestützten Entwurfsablauf entstanden ist. Als Ergebnis sind Softzellen[1] für die digitalen Blöcke entstanden, deren Funktionalität in einem Xilinx FPGA getestet wurde. Außerdem ein Mixed-Signal ASIC, in dem der Δ-Σ Modulator zweiter Ordnung, sowie eine programmierbare Stromquelle für den Drucksensor realisiert sind. Erleich-

[1] Diese Softzellen sind synthetisierbare VHDL Beschreibungen mit validierter Funktionalität.

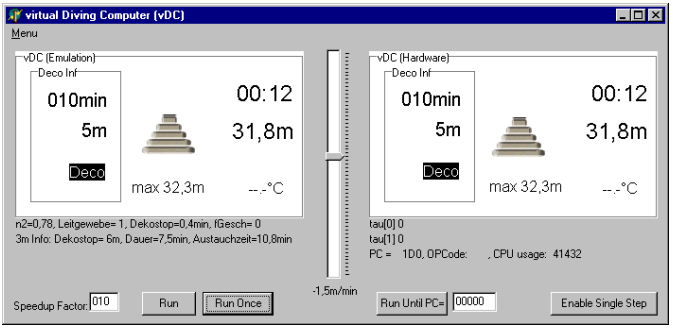


Abbildung 5.6: Bildschirmfoto der Validierungssoftware für den Dekompressionsalgorithmus

tert wurde der Entwurfsprozess durch die Entwicklung des Systems auf Basis eines Gesamtsystemmodells, das zur Stimulierzeugung bei der Validierung der Teilkomponenten weiterverwendet wird. Einige Komponenten des Gesamtsystems werden mehrfach modelliert, um den jeweiligen Anforderungen an Simulationsgeschwindigkeit oder Genauigkeit gerecht zu werden. Bei der Entwicklung des digitalen Teils des *v*DC verkürzte die FGPA basierte Validierungsumgebung die Implementierungs- und Testphase deutlich, da der Austausch von Teilkomponenten des Digitalteils, z. B. der Fließkomma-Arithmetikeinheit, durch Download einer FPGA Programmierdatei ermöglicht wird.

5.3 Modellierung der biologischen Vorgänge

Eine besondere Herausforderung bei der Entwicklung heterogener Systeme ist die optimale Abbildung der Aufgabenstellung in ein technisch realisierbares System. In diesem Kapitel wird die Modellbildung und Implementierung der biomedizinischen Aspekte des *v*DC vorgestellt. Die Modellierung der biophysikalischen Vorgänge im Körper des Tauchers und deren Bewertung unter medizinischen Gesichtspunkten steht am Anfang des Entwurfsprozesses. Motiviert wird dies einerseits methodisch durch das in Kapitel 4.1 vorgestellte Konzept des Test-basierten Entwurfsablaufs - und andererseits aus rein praktischen Überlegungen: Die Anforderungen an den Dekompressionscomputer können erst dann exakt formuliert werden, wenn die benötigte Genauigkeit der Messdaten, sowie die benötigte Rechenleistung im digitalen Teil bestimmt sind.

5.3.1 Einführung in die Problematik des Presslufttauchens

Seit über 100 Jahren ist bekannt, dass bei zu schneller Druckreduktion Verletzungen im organischen Gewebe auftreten. So existiert eine Veröffentlichung von R. Heller et. al. [29] aus dem Jahr 1900, der anhand von Tierexperimenten nachwies, dass direkt nach zu schneller Dekomprimierung im venösen Blutkreislauf Blasen zu finden sind, welche Schmerzen, Bewusstlosigkeit und den Tod auslösen können. Dasselbe gilt auch für Menschen. Der Umgebungsdruck, der auf einen Taucher einwirkt, ist direkt proportional zu der Wassersäule über ihm. Der Umgebungsdruck an der Wasseroberfläche beträgt etwa 100 kPa, eine 1 m hohe Wassersäule erhöht diesen Druck um 10 kPa. Daraus ergibt sich eine Druckverdopplung beim Abtauchen auf 10 m Tiefe, bzw. eine Druckhalbierung beim Auftauchen aus 10 m Tiefe.

Damit der Taucher atmen kann, wird er mit Pressluft versorgt. Durch die präzise Anpassung des Drucks der Atemluft an den Umgebungsdruck erhält der Taucher die Möglichkeit sich uneingeschränkt im Wasser zu bewegen. In der Lunge löst sich durch den Gasdruck ein Teil der Atemluft im Blut des Tauchers und wird zusätzlich zu dem an Hämoglobin gebundenen Sauerstoff in den Körper transportiert. Die Menge des gelösten Gases steht dabei (im Sättigungszustand) in direktem Verhältnis zum Druck des Gases.

Bei Druckerhöhung ist dieser aus der Physik bekannte Zusammenhang medizinisch unbedenklich, bei Druckreduktion, bzw. beim Auftauchen, kann es durch die abnehmende Löslichkeit des Gases im Gewebe zu Bläschenbildung kommen.

Die Hauptbestandteile von Pressluft sind Stickstoff (78%) und Sauerstoff (21%). Eine Übersättigung des Tauchers mit Sauerstoff ist in der Regel unkritisch, da der überschüssige Sauerstoff im Blut durch Anlagerung an Hämoglobin beseitigt werden kann. Das Inertgas Stickstoff kann im Körper jedoch nicht abgebaut werden, so dass überschüssiger Stickstoff vollständig über die Lunge abgeatmet werden muss. Wenn die Stickstoffübersättigung im Körper zu groß ist, bilden sich Stickstoffbläschen, die durch Anlagerung zusätzlicher Stickstoffmoleküle rasch wachsen. Diese führen zu dem auch als Caissonkrankheit bekannten Dekompressionsunfall.

In den folgenden Abschnitten wird der Stickstoffaustausch im menschlichen Körper, sowie dessen Modellierung genauer betrachtet. Die für diese Beschreibungen benötigten Größen sind in Tab. 5.1 zusammengestellt.

5.3.2 Modellierung der Sättigungsvorgänge im menschlichen Körper

Ein Teil des in der Atemluft enthaltenen Stickstoffs wird durch die Lunge in das arterielle Blut übernommen. Der Stickstoffpartialdruck im arteriellen Blut p_{N_2} ist dabei vom Stickstoffanteil der Atemluft, sowie vom Umgebungsdruck p_{Env} abhängig. Der Wasserdampf, der in den Lungenbläschen vom Blut in die Atemluft abgegeben wird, erzeugt einen Gegendruck (etwa 6,27 kPa), der dem Umgebungsdruck entgegenwirkt, so dass p_{N_2} bei normaler Atemluft mit 78% Stickstoffanteil wie in Gl. (5.1) angegeben zu berechnen ist.

$$p_{N_2}(p_{Env}) = 0,78 \cdot (p_{Env} - 6,27\,kPa) \tag{5.1}$$

Das Blut transportiert den gelösten Stickstoff durch den Körper des Tauchers. Dabei diffundiert der Stickstoff in die unterschiedlichen Körperschichten. Der von Prof. J. S. Haldane im Jahr 1908 vorgestellte Ansatz [14] den menschlichen Körper in eine Reihe unabhängiger Schichten zu unterteilen, ist Grundlage vieler heutzutage üblicher Modellierungsansätze. Gut durchblutete Schichten, z. B. Muskeln, werden schnell aufgesättigt, andere, z. B. stark fetthaltige, Schichten werden langsamer aufgesättigt. Unter der Annahme, dass eine organische Schicht über ihre Oberfläche Stickstoff aufnimmt, dieser jedoch innerhalb der Schicht nach vernachlässigbarer Zeit gleichmäßig verteilt ist, wird angenommen, dass sich bei einer Umgebungsdruckänderung die Gewebesättigung entsprechend Gl. (5.2) ändert.

$$\frac{dp_{Gew}(t)}{dt} = k \cdot (p_{N_2}(t) - p_{Gew}(t)) \tag{5.2}$$

Bezeichnung	Erklärung
Gew	Platzhalter für die Gewebesorte $Gew \in \{G1, G2, G3, \ldots G16\}$
τ_{Gew}	Halbwertszeit des Gewebes Gew
a_{Gew}, b_{Gew}	Koeffizienten des ZH-L16 Algorithmus (siehe Gl. (5.5))
$p_{Env}(t)$ $p_{Env}(d)$	Umgebungsdruck zum Zeitpunkt t Umgebungsdruck in der Wassertiefe d
$d(p_{Env})$	Wassertiefe, in der der Umgebungsdruck p_{Env} herrscht
$p_{N_2}(t)$ $p_{N_2}(d)$	Stickstoffpartialdruck des arteriellen Blutes zum Zeitpunkt t Stickstoffpartialdruck des arteriellen Blutes in der Tiefe d
$p_{Gew}(t)$	Stickstoffsättigung des Gewebes Gew zum Zeitpunkt t Die Stickstoffsättigung entspricht dem aktuellen Stickstoffpartialdruck des Gewebes.
$p_{Gew,\max}(p_{Env})$	Maximal zulässige Stickstoffsättigung im Gewebe Gew für den Umgebungsdruck p_{Env}
$p_{Env,\min}(Gew, p_{Gew})$	Maximal tolerierter Umgebungsunterdruck für Gewebe Gew, bei einer Gewebesättigung von p_{Gew}
$p_{Env,\min}(t)$	Geringster Umgebungsunterdruck, dem der Taucher zum Zeitpunkt t ohne gesundheitliche Gefährdung ausgesetzt werden darf
$t_{NZ,Gew}(t)$	Verbleibende Nullzeit für das Gewebe Gew zum Zeitpunkt t, unter der Annahme, dass die aktuelle Tauchtiefe beibehalten wird
$t_{NZ}(t)$	Verbleibende Nullzeit des Tauchers in der aktuellen Tauchtiefe zum Zeitpunkt t

Tabelle 5.1: Übersicht über die für den Dekompressionsalgorithmus benötigten Größen

Gewebesorte	G1	G2	G3	G4	G5	G6
$\tau\,[min]$	4	8	12,5	18,5	27	38,3
Gewebesorte	G7	G8	G9	G10	G11	G12
$\tau\,[min]$	54,3	77	109	146	187	239
Gewebesorte	G13	G14	G15	G16		
$\tau\,[min]$	305	390	498	635		

Tabelle 5.2: Halbwertzeiten der Gewebesorten des ZH-L16 Modells. Die Halbwertszeiten sind in Minuten angegeben.

Mit Hilfe von Experimenten an Ziegen wurden 5 Halbwertszeiten für gefährdete Körperschichten ermittelt. Dabei wurde eine Übersättigung der Körperschichten um 100% als tolerabel angenommen.

Nach einer Vielzahl von Experimenten mit US-Navy Tauchern erweiterten O. D. Yarbrough (1937) [94], J. V. Dwyer (1956) [25] und R. D. Workman (1965) [92] das Modell, indem sie die medizinisch unbedenkliche Übersättigung der Körperschichten genauer untersuchten.

Daraus entstand 1965 der heutzutage übliche Ansatz zur Berechnung der maximal zulässigen Stickstoffsättigung $p_{Gew,\max}$ als Funktion der Gewebeschicht und des Umgebungsdrucks p_{Env} (siehe Gl. (5.3)).

$$p_{Gew,\max}(p_{Env}) = c_{Gew} \cdot p_{Env} + M_{Gew} \tag{5.3}$$

A. A. Bühlmann erarbeitete auf Basis dieser Ansätze die Dekompressionsalgorithmen ZH-L12 (1983), ZH-L16 (1990) und ZH-L8ADT (1995), die für die Umsetzung in ein Computerprogramm aufbereitet sind [7].

Die Entwicklung von Dekompressionsalgorithmen ist bis heute nicht abgeschlossen. Das Divers Alert Network (DAN) sammelt die bei Tauchunfällen gewonnenen Daten, um die Parametrisierung der Modelle zu verbessern. Das zur Zeit aktuellste Tabellenwerk zur Berechnung von Dekompressionszeiten [27] wurde 2000 von M. Hahn veröffentlicht. Zusätzlich zu den auf Unfällen basierenden Daten werden von einigen langjährigen Tauchern Tauchprofile in Bezug auf Ermüdungserscheinungen untersucht. Zusätzliche Dekompressionspausen in größerer Tiefe können eventuell die Müdigkeit nach einem tiefen Tauchgang reduzieren [3].

Für den Entwurf des *v*DC wurde der am ausführlichsten dokumentierte ZH-L16 Algorithmus ausgewählt. In diesem Algorithmus wird der Körper in 16 Gewebesorten unterteilt, die unterschiedliche Halbwertszeiten (τ) besitzen (siehe Tab. 5.2).

Durch Umformung von Gl. (5.3) kann für jedes dieser Gewebe der geringste, medizinisch unbedenkliche Umgebungsdruck $p_{Env,\min}$ wie in Gl. (5.5) angegeben, berechnet werden.

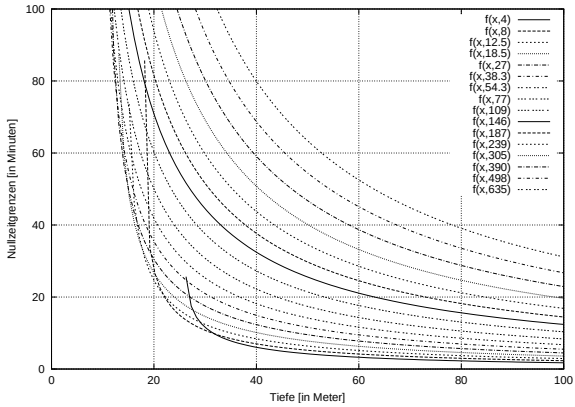


Abbildung 5.7: Nullzeiten für die Gewebeschichten des ZH-L16 Modells

$$p_{Env,\min}(Gew, p_{Gew}) = \frac{p_{Gew} - M_{Gew}}{c_{Gew}} \tag{5.4}$$

$$\rightsquigarrow p_{Env,\min}(Gew, p_{Gew}) = (p_{Gew} - a_{Gew})b_{Gew} \tag{5.5}$$

$$a_{Gew} = 200kPa(\frac{\tau_{Gew}}{60s})^{-\frac{1}{3}}$$

$$b_{Gew} = 1,005 - (\frac{\tau_{Gew}}{60s})^{-\frac{1}{2}}$$

Abb. 5.7 zeigt die innerhalb dieses Modells erlaubten Tauchgänge mit Direktaufstieg zur Wasseroberfläche. Unterhalb der Kurve ist die Stickstoffsättigung für die dazugehörige Gewebeschicht geringer als die an der Wasseroberfläche erlaubte Menge. Bei zunehmender Tauchtiefe nimmt die als medizinisch unbedenklich eingestufte Tauchzeit stark ab.

Ab dem Beginn des Tauchgangs ist mit Gl. (5.2) für die 16 Gewebesorten der Druck p_{Gew} zu berechnen. Beim Auftauchen ist darauf zu achten, dass der Umgebungsdruck nie kleiner als $p_{Env,\min}$ wird.

Mit den Daten aus Abb. 5.7 können nun direkt Tauchprofile definiert werden, mit denen eine Validierung des Dekompressionscomputers möglich ist. Ein Tauchgang, bei dem eine konstante Tauchtiefe eingehalten wird, wird *Rechteck-Tauchgang* genannt. Wie man anhand der Abb. 5.7 leicht sieht, ist ein Rechteck-Tauchgang von 20 Minuten in 20m Tiefe innerhalb der Nullzeit, d. h. dieser Tauchgang führt nicht zu einem Dekompressionsunfall. Ein Rechteck-Tauchgang von 40 Minuten in 20m Tiefe führt jedoch zu Dekompressionsbeschwerden in 5 Gewebesorten. Eine Validierung mit einem

Stimuligenerator, der den 20 minütigen Tauchgang nachbildet, kann als erfolgreich betrachtet werden, wenn das Modell des *v*DC *keine* Dekompressionswarnung berechnet. Die Validierung auf Basis des 40 minütigen Tauchgangs ist dementsprechend erfolgreich verlaufen, wenn die Dekompressionswarnung auftritt.

Abschließend wird das mathematisches Meta-Modell zur Modellierung der Sättigungsvorgänge im Körper des Tauchers formuliert. Die biologischen Vorgänge sind durch gewöhnliche Differentialgleichungen beschrieben, so dass die Beschreibung durch ein Modell der γ-Klasse (siehe Kapitel 2.3.3) erfolgt. Gemäß der Definitionen aus Kapitel 2.2 wird das Modell mit Hilfe von Zustandsgrößen $\mathbf{z}_i$ beschrieben.

Als externer Stimulus wird der Umgebungsdruck vorgegeben. Der Stickstoffpartialdruck im Blut wird mit der Zustandsgröße $\mathbf{z}_2$ beschrieben und kann über das Teilmodell $\mathscr{M}_1$ beschrieben werden. Der Stickstoffpartialdruck in den einzelnen Gewebesorten wird mit den 16 Zustandsgrößen mit den Indizes 3 bis 18 modelliert:

$$\mathscr{S} = \{\mathbf{Z} | \mathbf{z}_1(t) = p_{Env}(t)\} \tag{5.6}$$

$$I_S = \{1\} \tag{5.7}$$

$$\mathscr{M}_1 = \{\mathbf{Z} \,|\, \mathbf{z}_2(t) = 0,78 \cdot (\mathbf{z}_1(t) - 6,27\,kPa)\,\} \tag{5.8}$$

für $k \in \{3,4,5,\ldots,19\}$:

$$\mathscr{M}_{k-1} = \left\{ \mathbf{Z} \,\middle|\, \lim_{t \to t_0} \left(\frac{\mathbf{z}_k(t) - \mathbf{z}_k(t_0)}{t - t_0} \right) = k \cdot (\mathbf{z}_2(t) - \mathbf{z}_k(t)) \right\} \tag{5.9}$$

für $k \in \{19,20,21,\ldots,34\}$:

$$\mathscr{M}_{k-1} = \left\{ \mathbf{Z} \,\middle|\, \mathbf{z}_k(t) = (\mathbf{z}_{(k-16)}(t) - a_k) b_k \right\} \tag{5.10}$$

Die benötigten konstanten Koeffizienten a_k, b_k können mit Gl. (5.5) bestimmt werden.

Eine Gefährdung des Tauchers kann anhand des geringsten tolerierten Umgebungsdrucks der Gewebesorte berechnet werden. Dazu muss ein Vergleich dieser Größe mit dem aktuellen Umgebungsdruck stattfinden. Die Menge der Observablen besteht aus den Zustandsgrößen $\mathbf{z}_{19}$ bis $\mathbf{z}_{34}$ in denen $p_{Env,\min}(Gew)$ gespeichert wird sowie $\mathbf{z}_2$. Zur numerischen Bestimmung der Zustandsgrößen wird ein Simulator benötigt, der für ein vorgegebenes $\mathbf{z}_1$ die Observablen $\mathscr{O}$ berechnen kann:

$$I_O = \{2,19,20,21,\ldots,34\} \tag{5.11}$$

$$\mathscr{V} = \mathscr{S} \bigcap_{k=1..33} \mathscr{M}_k \tag{5.12}$$

$$\mathscr{O} = \{\mathbf{Z}_{I_O} \,|\, \mathscr{Z}(\mathbf{Z}_{I_O}) \in \mathscr{V}\,\} \tag{5.13}$$

5.3.3 Abbildung des Dekoalgorithmus in ein Modell der α-Klasse

Die Stickstoffsättigung der Gewebesorten ist mit dem Modell der γ-Klasse korrekt beschrieben. Dieses Modell ist jedoch für die Implementierung in einem Dekompressionsrechner nicht geeignet, da in diesem Modell implizite Differentialgleichungen gelöst werden müssen. Im Folgenden werden die Ergebnisse der Machbarkeitsstudie beschrieben, mit der ein Implementierungsansatz erarbeitet wurde.

Zur Berechnung der biologischen Vorgänge soll der Digitalteil eines Mixed-Signal ASICs verwendet werden. Mit der Randbedingung, dass der integrierte digitale Prozessor in konstanten und bekannten Zeitabständen den Umgebungsdruck als Messwert erhält, kann das Modell aus dem vorherigen Kapitel in ein Modell der α-Klasse umgewandelt werden. Dieses Modell kann dann in dem Prozessor ausgeführt werden. Zusätzlich zur Berechnung der Gewebesättigung soll der Prozessor dem Taucher Dekompressionsinformationen zur Verfügung stellen. Während der Taucher direkt zur Wasseroberfläche auftauchen darf, soll die Nullzeit angezeigt werden. Die Nullzeit ist die Zeit, die der Taucher in der aktuellen Tiefe noch tauchen darf, bevor $p_{Env,\min}$ größer als der Umgebungsdruck an der Wasseroberfläche wird. Die benötigten Informationen während der dekompressionspflichtigen Phase bestehen aus der Wassertiefe, in der der Druck $p_{Env,\min}$ herrscht und der restlichen Mindestverweildauer unter Wasser.

Ausgangspunkt ist Gl. (5.2) mit der die Zustandsgrößen $\mathbf{z}_3$ bis $\mathbf{z}_{19}$ bestimmt wurden. Die Differentialgleichung muss gelöst werden, so dass eine explizite Gleichung entsteht mit der die Stickstoffsättigung berechnet werden kann. Aus Gl. (5.2) ergibt sich durch Umformung:

$$\frac{d\,p_{Gew}(t)}{dt} + k\,p_{Gew}(t) = k\,p_{N_2}(t) \tag{5.14}$$

Gl. (5.14) ist eine *gewöhnliche lineare Differentialgleichung erster Ordnung* für die eine allgemeine Lösung existiert (siehe [15, S. 420ff]):

$$\begin{aligned} p_{Gew}(t) &= e^{-kt}\left(\int_0^t k\,e^{kt}\,p_{N_2}(t)\,dt + c_1\right) \\ \rightsquigarrow p_{Gew}(t) &= k\,e^{-kt}\left(\int_0^t e^{kt}\,p_{N_2}(t)\,dt + c_1\right) \end{aligned} \tag{5.15}$$

Für die Dekompressionsberechnung ist eine iterative Berechnungsformel ausreichend, die durch Umformung der Integralgleichung entsteht:

$$p_{Gew}(t+\Delta t) = k\,e^{-k(t+\Delta t)} \left(\int_0^{t+\Delta t} e^{kt}\, p_{N_2}(t)\, dt + c_1 \right)$$

$$\rightsquigarrow p_{Gew}(t+\Delta t) = e^{-k\Delta t} \left(p_{Gew}(t) + k\,e^{-kt} \int_t^{t+\Delta t} e^{kt}\, p_{N_2}(t)\, dt \right) \tag{5.16}$$

Das Integral in Gl. (5.16) kann auf verschiedene Art und Weise gelöst werden. Wenn man das Integral mit der Trapezregel diskretisiert, ergibt sich Gl. (5.17).

$$\begin{aligned} p_{Gew}(t+\Delta t) =& e^{-k\Delta t} p_{Gew}(t) \\ &+ \frac{k\Delta t}{2} \left(p_{N_2}(t+\Delta t) + e^{-k\Delta t}\, p_{N_2}(t) \right) \end{aligned} \tag{5.17}$$

Wenn man annimmt, dass sich der Umgebungsdruck im Zeitintervall $[t; t+\Delta t]$ linear ändert, kann das Integral auch analytisch gelöst werden:

$$\text{Annahme: } p_{N_2}(t) = p_{N_2}(t_0) + \dot{p}_{N_2}(t_0)(t-t_0) \qquad \forall t \in [t_0; t+\Delta t] \tag{5.18}$$

$$\begin{aligned} &\frac{p_{Gew}(t_0+\Delta t) - e^{-k\Delta t} p_{Gew}(t_0)}{k\,e^{-k(t_0+\Delta t)}} = \int_{t_0}^{t_0+\Delta t} e^{kt}\, p_{N_2}(t)\, dt = \\ &\int_{t_0}^{t_0+\Delta t} e^{kt} \left(p_{N_2}(t_0) + \dot{p}_{N_2}(t_0)(t-t_0) \right) dt = \\ &\left(p_{N_2}(t_0) - t_0\, \dot{p}_{N_2}(t_0) \right) \int_{t_0}^{t_0+\Delta t} e^{kt}\, dt + \dot{p}_{N_2}(t_0) \int_{t_0}^{t_0+\Delta t} t\, e^{kt}\, dt = \\ &\frac{p_{N_2}(t_0) - t_0\, \dot{p}_{N_2}(t_0)}{k} \left(e^{k(t_0+\Delta t)} - e^{k(t_0)} \right) \\ &\quad + \frac{\dot{p}_{N_2}(t_0)}{k^2} \left(e^{k(t_0+\Delta t)} (k(t_0+\Delta t) - 1) - e^{k(t_0)} (k t_0 - 1) \right) = \\ &\frac{\left(p_{N_2}(t_0) - t_0\, \dot{p}_{N_2}(t_0) \right)(1 - e^{-k\Delta t})}{k\,e^{-k(t_0+\Delta t)}} \\ &\qquad\qquad + \frac{\dot{p}_{N_2}(t_0) \left(\Delta t + (t_0 - \frac{1}{k})(1 - e^{-k\Delta t}) \right)}{k\,e^{-k(t_0+\Delta t)}} \end{aligned}$$

$$\rightsquigarrow p_{Gew}(t_0+\Delta t) = e^{-k\Delta t} p_{Gew}(t_0) + \Delta t\, \dot{p}_{N_2}(t_0) + (1-e^{-k\Delta t})\left(p_{N_2}(t_0) - \frac{\dot{p}_{N_2}(t_0)}{k}\right) \quad (5.19)$$

$$\rightsquigarrow p_{Gew}(t_0+\Delta t) = p_{Gew}(t_0) + \Delta t\, \dot{p}_{N_2}(t_0) + (1-e^{-k\Delta t})\left(p_{N_2}(t_0) - p_{Gew}(t_0) - \frac{\dot{p}_{N_2}(t_0)}{k}\right) \quad (5.20)$$

Falls der Umgebungsdruck im Integrationszeitraum konstant ist, kann Gl. (5.20) weiter vereinfacht werden:

$$\rightsquigarrow p_{Gew}(t_0+\Delta t) = e^{-k\Delta t}\, p_{Gew}(t_0) + \left(1-e^{-k\Delta t}\right) p_{N_2}(t_0) \quad (5.21)$$

Gl. (5.21) entspricht der in der einschlägigen Literatur [5, 7, 79] benutzte Differenzengleichung. Üblicherweise wird anstelle der Exponentialfunktion die Funktion 2^x eingesetzt und anstelle des Koeffizienten k wird eine Halbwertszeit τ definiert. Mit $k = \frac{\ln(2)}{\tau_{Gew}}$ ergibt sich für eine lineare Umgebungsdruckänderung:

$$p_{Gew}(t_0+\Delta t) = p_{Gew}(t_0) + \Delta t\, \dot{p}_{N_2}(t_0) + (1-2^{-\frac{\Delta t}{\tau_{Gew}}})\left(p_{N_2}(t_0) - p_{Gew}(t_0) - \frac{\tau_{Gew}\, \dot{p}_{N_2}(t_0)}{\ln 2}\right) \quad (5.22)$$

Bei konstantem Umgebungsdruck gilt:

$$p_{Gew}(t+\Delta t) = p_{Gew}(t) + (p_{N_2}(t) - p_{Gew}(t))\left(1-2^{-\frac{\Delta t}{\tau}}\right) \quad (5.23)$$

Gl. (5.23) eignet sich gut für ein Modell der β-Klasse, da die Gleichung eine streng zeitkausale Abbildung von p_{N_2} und p_{Gew} auf p_{Gew} ist. Die Modellbeschreibung aus Gl. (5.9) kann nun streng zeitkausal definiert werden.

Für $k \in \{3,4,5,\ldots,19\}$:

$$\mathscr{M}_{k-1} = \Big\{\mathbf{Z}\Big|\mathbf{z}_k(t_2) = \mathbf{z}_k(t_1) + (\mathbf{z}_2(t_1) - \mathbf{z}_k(t_1))\left(1-2^{-\frac{t_2-t_1}{\tau}}\right) \wedge (t_2 > t_1) \wedge (\forall t_3 \in \mathscr{T}_{\mathbf{z}_k} : t_3 > t_1 \Rightarrow t_3 \geq t_2)\Big\} \quad (5.24)$$

Wenn die Anzahl der Zeitstempel in $\mathscr{T}_{\mathbf{z}_2}$ endlich ist, sind diese Teilmodelle ereignisdiskret.

Somit ist auch eine Umformung des obigen Teilmodells in ein sequenz-orientierten Modell der α-Klasse denkbar, wenn der Stickstoffpartialdruck im Blut nur für ausgewählte Zeitpunkte erfasst wird.

In Modellen der α-Klasse kann der Abstand zwischen zwei *Tags* nicht berechnet werden. Wenn der Abstand jedoch im voraus bekannt ist, können die dazugehörigen Zahlenwerte als Teil der Stimulidaten übergeben werden. Für die Implementierung des Dekompressionsrechners bietet sich eine konstante Samplingrate an, der Abstand zwischen 2 Messwerten ist dann Δt.

$$\mathscr{S} = \{\mathbf{Z} | \mathrm{Seq}(\mathbf{z}_1, i) = p_{Env}(i \cdot \Delta t)\} \tag{5.25}$$

$$I_S = \{1\} \tag{5.26}$$

$$\mathscr{M}_1 = \{\mathbf{Z} \,|\mathrm{Seq}(\mathbf{z}_2, i) = 0,78 \cdot (\mathrm{Seq}(\mathbf{z}_1, i) - 6,27\,kPa)\,\} \tag{5.27}$$

Für $k \in \{3, 4, 5, \ldots, 19\}$:

$$\mathscr{M}_{k-1} = \Big\{\mathbf{Z}\Big|\mathrm{Seq}(\mathbf{z}_k, i+1) = \mathrm{Seq}(\mathbf{z}_k, i) + (\mathrm{Seq}(\mathbf{z}_2, i) - \mathrm{Seq}(\mathbf{z}_k, i))\left(1 - 2^{-\frac{\Delta t}{\tau}}\right)\Big\} \tag{5.28}$$

Für $k \in \{19, 20, 21, \ldots, 34\}$:

$$\mathscr{M}_{k-1} = \Big\{\mathbf{Z}\Big|\mathrm{Seq}(\mathbf{z}_k, i) = (\mathrm{Seq}(\mathbf{z}_{(k-16)}, i) - a_k)b_k\Big\} \tag{5.29}$$

Der Taucher soll durch den vDC Informationen bekommen, mit deren Hilfe er sicherstellen kann, dass der geringste tolerierte Umgebungsdruck $p_{Env,\min}$ $(\mathbf{z}_{35})$ größer ist als der aktuelle Umgebungsdruck $(\mathbf{z}_1)$.

$$\mathscr{M}_{34} = \Big\{\mathbf{Z}\Big|\mathrm{Seq}(\mathbf{z}_{35}, i) = \max_{19 \le k \le 34}\{\mathrm{Seq}(z_k, i)\}\Big\} \tag{5.30}$$

Zu überprüfen ist also $\mathrm{Seq}(z_1, i) > \mathrm{Seq}(z_{35}, i)$. Der Taucher kann diese Überprüfung leichter durchführen, wenn die Druckinformationen in Tauchtiefen umgerechnet werden. Tiefenangaben sind beim Tauchen seit langem durch analoge Tiefenmesser etabliert. Die Umrechnung des Umgebungsdrucks in eine Tauchtiefe d ist bei bekanntem Oberflächendruck $p_{Env}(0m)$ mit Gl. (5.31) möglich[2]. Die Wasserdichte beträgt je nach

[2] Um die Modellbildung zu vereinfachen, wird vorausgesetzt, dass der Umgebungsdruck zu Beginn der 'Modellierungszeit' dem Umgebungsdruck an der Wasseroberfläche entspricht.

Salzgehalt $1\frac{kg}{l^3}$ bis $1,03\frac{kg}{l^3}$, multipliziert mit der Erdanziehungskraft $(9,81\frac{N}{kg})$ ergibt sich der Umrechnungsfaktor von etwa $10\frac{N}{l^3} = 10\frac{kPa}{m}$.

$$d(p_{Env}) = \frac{p_{Env} - p_{Env}(0m)}{10kPa} \cdot 1m \tag{5.31}$$

$$\mathscr{M}_{35} = \{\mathbf{Z} \,|\, \mathrm{Seq}(\mathbf{z}_{36}, i) = 10(\mathrm{Seq}(\mathbf{z}_1, i) - \mathrm{Seq}(\mathbf{z}_1, 0))\,\} \tag{5.32}$$

$$\mathscr{M}_{36} = \{\mathbf{Z} \,|\, \mathrm{Seq}(\mathbf{z}_{37}, i) = 10(\mathrm{Seq}(\mathbf{z}_{35}, i) - \mathrm{Seq}(\mathbf{z}_1, 0))\,\} \tag{5.33}$$

$$I_O = \{36, 37\} \tag{5.34}$$

$$\mathscr{V} = \mathscr{S} \bigcap_{k=1..36} \mathscr{M}_k \tag{5.35}$$

$$\mathscr{O} = \{\mathbf{Z}_{I_O} \,|\, \mathscr{Z}(\mathbf{Z}_{I_O}) \in \mathscr{V}\,\} \tag{5.36}$$

Wenn $p_{Env,\min}$ größer ist als der Oberflächendruck, dann ist der Taucher dekopflichtig[3], und die effektivste Dekompressionstiefe ist $d(p_{Env,\min})$.

Ist $p_{Env,\min}$ kleiner als der Umgebungsdruck an der Wasseroberfläche, so ist die dazugehörige Tiefenangabe negativ. Für den Taucher ist es wichtig zu erfahren, dass er direkt zur Wasseroberfläche auftauchen darf, der Wert $d(p_{Env,\min})$ ist für ihn jedoch nicht von Interesse. Deshalb wird in dieser Nullphase genannten Tauchphase statt $d(p_{Env,\min})$ die Zeitdauer ausgegeben werden, die der Taucher noch in der aktuellen Tauchtiefe verweilen darf, ohne beim Auftauchen Zwischenstopps einlegen zu müssen. Diese Zeitdauer wird Nullzeit (t_{NZ}) genannt.

Für jede nicht dekopflichtige Gewebesorte kann eine solche Zeit $t_{NZ,Gew}$ berechnet werden. Dazu muss die folgende Gleichung gelöst werden:

$$p_{Env,\min}(t + t_{NZ,Gew}(t)) = p_{Env}(0m) \tag{5.37}$$

$$\rightsquigarrow p_{Gew}(t + t_{NZ,Gew}(t)) = \frac{p_{Env}(0m)}{b_{Gew}} - a_{Gew} \tag{5.38}$$

Bei gleichbleibender Tauchtiefe, d. h. bei konstantem p_{N_2} gilt Gl. (5.23) auch für große Zeiten $t_{NZ,Gew}$:

[3] Ein Taucher ist dekopflichtig, sobald er nicht direkt zur Wasseroberfläche auftauchen darf. Er muss dann für das Auftauchen Dekompressionspausen einhalten.

$$p_{Gew}(t+t_{NZ,Gew}(t)) = p_{Gew}(t) + (p_{N_2}(t) - p_{Gew}(t))\left(1-2^{-\frac{t_{NZ,Gew}(t)}{\tau_{Gew}}}\right) \quad (5.39)$$

$$\rightsquigarrow t_{NZ,Gew}(t) = \tau_{Gew}\log_2\left(\frac{p_{N_2}(t)-p_{Gew}(t)}{p_{N_2}(t)-p_{Gew}(t+t_{NZ,Gew})}\right) \quad (5.40)$$

$$\rightsquigarrow t_{NZ,Gew}(t) = \tau_{Gew}\log_2\left(\frac{p_{N_2}(t)-p_{Gew}(t)}{p_{N_2}(t)-\frac{p_{Env}(0m)}{b_{Gew}}+a_{Gew}}\right) \quad (5.41)$$

Die Nullzeit des Tauchers wird durch die Gewebesorte mit der niedrigsten Nullzeit bestimmt:

$$t_{NZ}(t) = \min_{\forall Gew}\{t_{NZ,Gew}(t)\} \quad (5.42)$$

Die Nullzeit des Tauchers ist von der aktuellen Gewebesättigung und vom aktuellen Umgebungsdruck abhängig, da die Gewebesorten in großer Tiefe schneller gesättigt werden. Wenn der Stickstoffpartialdruck in der aktuellen Tiefe jedoch niedriger ist als der an der Wasseroberfläche tolerierte Stickstoffpartialdruck, dann ist die Nullzeit unendlich. In Gl. (5.41) ist in diesem Fall das Argument des Logarithmus negativ.

5.3.3.1 Berechnung der Dekompressionsinformationen

Wenn ein direkter Aufstieg zur Wasseroberfläche nicht mehr erlaubt ist, sollen Dekompressionsinformationen (Dekoinformationen) berechnet werden. Als Dekoinformation wird zum Einen der Mindestdruck $p_{Env,\min}$, bzw. die Wassertiefe für den Dekostopp benötigt. Zum Anderen die restliche Mindestverweildauer bis zur Beendigung des Tauchgangs. Die Wassertiefe, in der dekomprimiert wird, wird *Dekostufe* genannt, die Verweildauer auf einer Dekostufe ist der dazugehörige *Dekostopp*.

Die Tiefe des Dekostopps wird mit Gl. (5.31) berechnet und ist in $\mathrm{Seq}(\mathbf{z}_{35})$ gespeichert. Die restliche Mindestverweildauer (=Austauchzeit) kann mit Gl. (5.2) durch Einsetzen von Gl. (5.1) und Gl. (5.5) berechnet werden. Dabei wird vorausgesetzt, dass der Taucher während der Austauchphase stets in der empfohlenen Wassertiefe $d(p_{Env,\min})$ bleibt.

Die Gewebesorte, die den Wert von $p_{Env,\min}$ bestimmt, wird Leitgewebe (LG) genannt. Dieses Leitgewebe ändert sich während eines Tauchgangs i. d. R. mehrfach, deshalb ist die exakte Berechnung der Mindestverweildauer sehr aufwendig. Wenn man statt

Gewebesorte	G1	G2	G3	G4	G5	G6
$\tau[min]$	4	8	12,5	18,5	27	38,3
Dekostopp	0:12	0:24	0:37,5	0:55,5	1:21	1:55
Gewebesorte	G7	G8	G9	G10	G11	G12
$\tau[min]$	54,3	77	109	146	187	239
Dekostopp	2:43	3:51	5:27	7:18	9:21	11:57
Gewebesorte	G13	G14	G15	G16		
$\tau[min]$	305	390	498	635		
Dekostopp	15:15	19:30	24:54	31:45		

Tabelle 5.3: Stufenförmige Dekompression mit Halbwertszeit-abhängiger Iterationsschrittweite: Je nach Gewebesorte des Leitgewebes wird der Dekostopp zwischen 12 Sekunden und 31 Minuten lang gewählt.

des optimalen Austauchprofils ein treppenförmiges Austauchprofil zugrundelegt, kann man die Mindestverweildauer iterativ berechnen, wodurch sich die Rechenschritte stark vereinfachen.

$$\frac{\Delta t}{\tau_{LG}} = \frac{1}{20} \tag{5.43}$$

Wählt man die Iterationsschrittweite (siehe Gl. (5.43)) abhängig von der Halbwertszeit des Leitgewebes, so erhält man ein vielstufiges Austauchprofil, das ab einer Dekompressionstiefe von 14 m mit Auftauchschrittweiten kleiner als 50 cm arbeitet und somit genauer ist, als die durch den Taucher verursachten Mindesttoleranzen. Somit ist das beschriebene treppenförmige Austauchprofil eine gute Abschätzung für die 'bestmögliche realisierbare Dekompression' und die Berechnung des Austauchprofils wird erleichtert, da der Sättigungsdruck in den Gewebesorten mit der Iterationsvorschrift aus Gl. (5.23) berechnet werden kann. Das hier vorgeschlagene Austauchprofil ist handelsüblichen Austauchprofilen mit Dekostufen, die in 3 m Abständen gewählt werden, überlegen, da die Dekompressionstiefe deutlich häufiger angepasst wird. Die Länge der Dekostopps (siehe Tab. 5.3) wird durch das Leitgewebe bestimmt, ist aber nicht von der Gewebesättigung selbst abhängig.

Um unbeabsichtigtes 'Durchschießen' (=Auftauchen) des Tauchers zu vermeiden, wird eine Mindesttiefe für die Dekostufen festgelegt. Sobald die Dekostufe erreicht ist, in der der Druck $p_{Deko,min}$ herrscht, wird die Dekompression auf diesem Niveau fortgeführt bis ein direkter Aufstieg zur Wasseroberfläche möglich ist.

Setzt man Gl. (5.5) in Gl. (5.1) ein, so erhält man eine Gleichung, aus der sich der Stickstoffpartialdruck des Blutes in der aktuellen Dekompressionstiefe berechnen lässt:

$$p_{N_2}(t) = 0,78 b_{LG} p_{LG}(t_0) - c_{LG} \qquad \forall t \in [t_0; t_0 + \Delta t] \tag{5.44}$$

$$c_{LG} = 0,78(a_{LG} b_{LG} + 6,27kPa) \tag{5.45}$$

Zur Berechnung der Austauchzeit wird angenommen, dass der Taucher nach einem Dekostopp direkt auf die neue Dekostufe auftaucht. Somit kann die Berechnung in Gleichung (5.44) solange wiederholt werden, bis der Umgebungsdruck den Wert $p_{Deko,min}$ erreicht ist. In dieser Dekompressionstiefe wird nun dekomprimiert, bis ein direktes Auftauchen zur Oberfläche möglich ist, so dass der Dekostopp auf dieser Dekostufe mehrere Δt lang sein kann. Die Summe der Verweildauer in den einzelnen Dekompressionstiefen ist die Mindestaustauchzeit.

5.3.3.2 Interpretation des zeitkausalen Modells

Es wurde mit der Machbarkeitsstudie gezeigt, dass der benötigte Dekompressionsalgorithmus mit einem Modell der α-Klasse berechnet werden kann. Bei dieser Rechnung entsteht eine Folge von Zuständen, die nun in einem Zeit-Kontext interpretiert werden müssen. Es wurde vorausgesetzt, dass die Stimulidaten in festen gleichbleibenden Abständen generiert werden. Die Stimulidaten besitzen als *Tag* somit einen Zeitstempel. Die Menge der Stimulidaten mit einem *Tag* kleiner als t wird verwendet um den *aktuellen Systemzustand zum Zeitpunkt t* zu definieren: Dieser wird durch den letzten Zustand der Zustandsgrößen bestimmt, die mit den vorgegebenen Stimulidaten berechnet werden kann.

Für die Implementierung des *v*DC wird ein Modell benötigt, dass im Verlauf einer Zeitbereichssimulation jeweils den aktuellen Systemzustand bestimmt und zur Verfügung stellt. Der Systemzustand zu früheren Zeitpunkten muss nicht gespeichert werden. Somit wird für das Modell kein aufwendiger Simulationskern benötigt, das Modell kann auch in einer Programmiersprache wie ANSI C effektiv realisiert werden.

5.3.4 Zusammenfassung

Die Berechung der Dekoinformation ist mit einem sequenz-orientiertes Modell der α-Klasse möglich. Dieses wird im Rahmen der folgenden Gesamtsystemmodellierung implementiert. Aufgrund der stark unterschiedlichen Zeitkonstanten der Gewebesorten, der exponentiellen Sättigungsvorgänge und der iterativen Berechnungsvorschriften ist zu erwarten, dass numerische Fehler nur bei sorgfältiger Wahl der Wertdiskretisierung der Zustandsgrößen vernachlässigbar sind. Die Zustandsgrößen wurden deshalb mit einem Fließkommazahlenformat implementiert.

5.4 Systemspezifikation und Gesamtsystemmodellierung

Für die Modellierung des *v*DC wird ein Simulationssystem benötigt, in dem sowohl analoge und digitale Schaltungen, als auch zeitkontinuierliche Modelle von nicht elektrischen Komponenten beschrieben werden können. Dafür wird ein VHDL-AMS Simulator genutzt. Dieser Simulator erfüllt die Voraussetzungen und bietet weitere Vorteile: VHDL-AMS ist eine standardisierte Simulationssprache, so dass eine Adaption der Modellbeschreibungen an andere VHDL-AMS Simulatoren einfach durchführbar ist[4]. Da VHDL-AMS eine Obermenge von VHDL ist, können vorhandene Komponentenmodelle in VHDL ohne Änderungen verwendet werden. Ebenso kann ein Gesamtsystemmodell in VHDL-AMS für die Validierung bei der Komponentenentwicklung genutzt werden.

Im Rahmen der im Folgenden beschriebenen Arbeitsphase werden die Anforderungen an die Bestandteile des *v*DC festgelegt, sowie durch Simulation eines Gesamtsystemmodells validiert. Als Ausgangspunkt für die Komponentenentwicklung steht dann neben dem erweiterten Pflichtenheft auch eine ausführbare Spezifikation zur Verfügung. Die VHDL-AMS Modelle, die in diesem Kapitel vorgestellt werden, wurden mit dem Simulator Advance MS [67] simuliert und getestet. Da dieser Simulator nicht den vollständigen VHDL-AMS Sprachumfang unterstützt, sind die Modelle dementsprechend formuliert.

Damit mit geringem Aufwand mehrere Lösungsansätze getestet werden können, sollen die Modelle, die in der Machbarkeitsstudie entstehen, einfach erstellt werden können und leicht zu ändern sein. Wenn viele Lösungsideen getestet werden sollen, müssen die Simulationszeiten kurz sein. Um die Modelle als Zielrichtlinie für die weiteren Arbeitsphasen verwenden zu können, ist es wichtig, dass sie so aufbereitet werden, dass sie leicht zu begreifen sind.

Die Aufgabe des *v*DC wurde in Kapitel 5.3 beschrieben. Es soll sowohl die Stickstoffsättigung im Körper des Tauchers berechnet werden, als auch zusätzliche Informationen, die dem Taucher ein sicheres Auftauchen ermöglichen. Das Gerät benötigt dementsprechend einen Drucksensor, sowie eine elektronische Signalverarbeitung. Obwohl die Realisierung des *v*DC als System-On-Chip im Rahmen der hier vorgestellten Projektarbeit nicht möglich war, wurde eine hochintegrierte Implementierung des *v*DC vorbereitet. Die Integration eines Drucksensors mit CMOS Schaltungstechnik in einem Halbleiter ist zur Zeit nur bei wenigen Halbleiter-Herstellern möglich. Abb. 5.8 zeigt einen Referenzentwurf der X-FAB Semiconductor Foundries AG [93], der in einem $1,0\mu m$ Prozess gefertigt ist.

Die iC-Haus GmbH hat sich bereit erklärt die Fertigungskosten für einen Mixed-Signal ASIC für den *v*DC zu übernehmen. Dieser ASIC sollte auf einem Multi-Projekt Wafer

[4] Idealerweise sollten VHDL-AMS Modelle ohne Änderungen auf allen VHDL-AMS Simulatoren lauffähig sein. Manche Simulatoren unterstützen jedoch (noch) nicht den vollständigen VHDL-AMS Sprachumfang, so dass eventuell Anpassungen notwendig sind. Ebenso können je nach Simulator unterschiedliche Konvergenzprobleme auftreten.

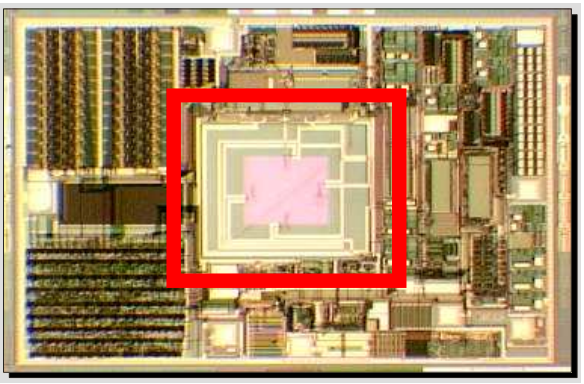


Abbildung 5.8: Chip-Photo einer CMOS Schaltung mit integriertem mikromechanischen Drucksensor, Quelle: X-FAB Semiconductor Foundries AG

in der $0,8\mu m$ Technologie der Firma X-Fab eingeschleust werden[5]. Die Entwurfsregeln (Designrules) für die Schaltung zur Sensorsignalaufbereitung und zur A/D Wandlung der Messergebnisse sind dadurch vorgegeben.

Da die Halbwertszeiten der langsamen Gewebesorten mehrere Stunden betragen, ist eine rein analoge Signalverarbeitung nicht sinnvoll. Statt dessen bietet sich eine Mikrokontroller gestützte Datenverarbeitung an.

Entsprechend des qualitätsbasierten Arbeitsflusses aus Kapitel 4 werden vor der Spezifikation des Dekompressionscomputers erste Testszenarien erarbeitet, die bei der Validierung des Systems genutzt werden können. Zur Überprüfung der Ergebnisse des *v*DC wird eine Referenzimplementierung benötigt, die die biologischen Vorgänge im menschlichen Körper möglichst genau nachbildet. Das in Abschnitt 5.3.3 vorgestellte Berechnungsmodell erfüllt diese Voraussetzungen und soll auf möglichst einfache Weise implementiert werden. Eine Implementierung als C-Programm bietet sich hierfür an. Diese ist in Bezug auf die benötigte Simulationszeit effizient und kann auch in späteren Entwicklungsphasen weiter genutzt werden. Das Modell kann mit einem trivialen Rechteck-Tauchgang Stimuligenerator als eigenständiges Programm ausgeführt werden. Es kann über die C-Schnittstelle des VHDL-AMS Simulators gemeinsam mit einem ereignisdiskreten oder einem zeitkontinuierlichen Modell ausgeführt werden. Außerdem kann es parallel zu einem FPGA Design genutzt werden, um die Ergebnisse des FPGA Designs mit den Daten der Referenzimplementierung zu testen.

Mit Hilfe der Referenzimplementierung kann getestet werden, wie genau die Druckinformation diskretisiert werden muss und in welchen zeitlichen Abständen die Messer-

[5] Die verwendete CMOS Technologie ist der für den MEMS Prozess freigegebenen Technologie sehr ähnlich.

```
library disciplines;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;

entity vdc_sources is
  generic(timescale : real := 1.0);
  port(terminal Pressure_sensor : fluidic;
       terminal Temp_sensor : thermal);
end vdc_sources;
```

Beschreibung 5.1: Entity Deklaration des Stimuligenerators

gebnisse verarbeitet werden müssen, um eine korrekte Nachbildung der zeitkontinuierlich ablaufenden Sättigungsvorgänge des Tauchers zu gewährleisten.

Auf Basis dieser Daten wird der Sensor spezifiziert, wobei als zusätzliche Bedingung eine möglichst kompakte Bauform gesucht wird, um eine Miniaturisierung des *v*DC zu ermöglichen. Die Signalaufbereitung und die A/D Wandlung ergibt sich dann aus den Daten, die mit der Referenzimplementierung gewonnen wurde, sowie dem Ausgangssignal des Sensors.

Die Spezifikation des *applikationsspezifischen Digitalteils* des *v*DC und der Firmware schließt die Spezifikationsphase ab.

Mit Hilfe der Gesamtsystemsimulation wird die gefundene Partitionierung des *v*DC, sowie die Konsistenz der Spezifikation überprüft. Das Gesamtsystemmodell wird dabei so aufbereitet, dass es in der Komponentenentwicklung als Teil der Validierungsumgebung weiterverwendet werden kann.

5.4.1 Erzeugung von Stimulidaten

Für die Validierung des *v*DC werden Testszenarien benötigt. VHDL-AMS bietet mit dem Konzept der *Natures* die Möglichkeit Modelle zu erstellen, deren Zustandsgrößen aus unterschiedlichen Domänen stammen. So kann eine Zustandsgröße vom Typ *Fluidic* definiert werden, deren Potentialgröße eine Druckangabe ist, die z. B. dem Umgebungsdruck in der Tauchtiefe des Tauchers entsprechen kann (s. Beschreibung 5.1). Eine Zustandsgröße vom Typ *Thermal* kann die Temperatur an der Position des Tauchers beschreiben, während eine Zustandsgröße vom Typ *Electrical* z. B. den Stromfluss durch einen Drucksensor bestimmt.

Ein solches 'heterogenes' Modell, das Zustandsgrößen aus verschiedenen Domänen nutzt, kann in Bezug auf die in Kapitel 2.3 vorgestellten Modellklassen homogen sein: Die für die Testszenarien benötigten Stimulidaten sind zeitkontinuierliche, ortsdiskrete Werte. Dementsprechend bietet sich für diesen Zweck ein Modell der γ-Klasse als Beschreibung an.

In dieser frühen Entwicklungsphase ist es sinnvoll, mit den Testszenarien Test-Tauchgänge nachzubilden, mit denen einige prinzipielle Funktionstests durchgeführt werden können. Für eine gründliche Validierung des *v*DC werden jedoch auch Tauchprofile benötigt, die üblichen Tauchgewohnheiten entsprechen. Die Überprüfung von Ausnahmefällen und Extremsituationen wird in späteren Entwicklungsphasen erfolgen.

Der Rechtecktauchgang in 20 m Tiefe ist ein typischer Test-Tauchgang, bei dem der *v*DC nach Überschreitung der Nullzeitgrenze ein Dekompressionsproblem beim Auftauchen feststellen muss.

Als Beispiel für 'übliche Tauchgewohnheiten' wird jeweils ein Süßwassertauchgang im Baggersee und ein Salzwassertauchgang im Meer modelliert. Ein Tauchgang in einem Baggersee in Deutschland ist i. d. R. ein Tauchgang vom Ufer aus. Der Taucher wird also vom Seeufer aus entlang des Seegrundes bis zu einer bestimmten Tiefe tauchen, sich dort einige Zeit aufhalten, um anschließend wieder am Grund entlang zum Ufer zu tauchen. Untersucht man Baggerseen, so stellt man fest, das in vielen Seen in den flachen und wärmeren Regionen des Sees Unterwasserpflanzen wachsen, während in den dunklen tieferen Regionen des Sees der Grund kaum bewachsen ist. Viele Tiere halten sich in den Regionen mit starkem Pflanzenwuchs auf, während nur wenige Fische in den dunklen kalten Regionen zu finden sind. Die Wassertemperatur ist in einem Baggersee stark von der Tiefe abhängig. Ab einer Tiefe von etwa 20 m ist die Temperatur auch im Hochsommer in den meisten Baggerseen deutlich unter 10 °C. Der Beispieltauchgang in Beschreibung 5.2 zeigt einen Tauchgang bis in eine Tiefe von etwa 21 m, nach 18 Minuten taucht der Taucher in die warme Zone des Sees auf, um dann weitere 20 Minuten in den flachen warmen Regionen des Sees zu tauchen.

Wie in Kapitel 5.5.1 motiviert, wird für die Anregung des Dekompressionscomputers neben dem Umgebungsdruck auch die Umgebungstemperatur benötigt. Je nach Tauchgebiet und Jahreszeit existieren zwischen Tauchtiefe und Umgebungstemperatur feste Beziehungen, die insbesondere für Baggerseen leicht zu formulieren sind (siehe Beschreibung 5.2). Im offenen Meer ist die Temperatur aufgrund von Wasserströmungen nicht so stark tiefenabhängig. Dafür kann sich die Temperatur in einer bestimmten Tiefe durch Wasseraustausch ändern. Im offenen Meer reicht die bewachsene Zone tiefer und es gibt mehr Tiere, die sich in großer Tiefe aufhalten, so dass auch tiefere Tauchgänge durchgeführt werden (siehe Beschreibung 5.3). Ab einer Tauchtiefe von 40 m wirkt der Stickstoffpartialdruck der Atemluft narkotisierend, so dass die Konzentrationsfähigkeit des Tauchers nachlässt. Ab einer Tiefe von 80 m steigt zusätzlich das Risiko einer Sauerstoffvergiftung, die meist tödlich endet. Der Einsatzbereich des *v*DC wird dementsprechend beschränkt: Tauchgänge tiefer als 100 m werden nicht für die Validierung benutzt.

Neben diesen künstlich definierten Tauchszenarien ist es auch wünschenswert, Daten von realen Tauchgängen als Stimulidaten verwenden zu können. Zu diesem Zweck wurden Tauchgangsdaten eines Aladin Air Tauchcomputers [87] über ein PC-Interface ausgelesen und mit Hilfe eines Perl-Programms aufbereitet. (siehe Abb. 5.9).

```
library disciplines;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;

library IEEE;
use IEEE.math_real.all;

architecture beh_LakeDive of vdc_sources is
  constant surfacePressure : pressure := 100.0;  -- (100 kPa)
  constant metalimnion     : pressure := 220.0;  -- (220 kPa)
  constant surfaceTemp     : temperature := 21.0;  -- (21 °C)

  function LakeTemp (constant Tsurface, Psurface, Pmetalimnion : real;
                     p : real) return real is

    constant Tc : real := 5.0 / (Pmetalimnion - Psurface);
    constant Tb : real := (Tsurface - 10.0) /
                          (exp(Tc * Psurface) - exp(Tc * Pmetalimnion));
    constant Ta : real := Tsurface - Tb * exp(Tc * Psurface);
    constant Td : real := 4.0;
    constant Tf : real := (Tb * Tc * exp(Tc * metalimnion))  / (10.0 - Td);
    constant Te : real := (10.0 - Td) / exp(Tf * metalimnion);
    -- the 6 constants are calculated based on:
    -- Temp(p=Psurface) == surfaceTemp
    -- Temp(p=Pmetalimnion) == 10.0 °C
    -- Temp(p -> infty) == 4.0 °C
    -- Temp'dot(p=Psurface) * exp(5.0) == Temp'dot(p=Pmetalimnion)
    -- Temp(metalimnion - delta ) == Temp(metalimnion + delta)
    -- Temp'dot(metalimnion - delta ) == Temp'dot(metalimnion + delta)
    variable res : real;
  begin  -- LakeTemp
    if (p < Pmetalimnion) then
      res := Ta + Tb * exp(Tc * p);
    else
      res := Td + Te * exp(Tf * p);
    end if;
    return res;
  end LakeTemp;

  quantity p across  P_flow through  Pressure_sensor;
  quantity Temp across  T_P through  Temp_sensor;
begin
  if (now < 0.1 * timescale) use        -- diving starts at the surface
    p == surfacePressure;
  elsif (now < 3.0 * timescale) use     -- descending along the lake's profile
    p'dot == 30.0 / timescale;
  elsif (now < 16.0 * timescale ) use   -- descending along the lake's profile
    p'dot == 10.0 / timescale;
  elsif (now < 18.0 * timescale ) use   -- ground time: 2 min
    p'dot == 0.0;                       -- (cold, dark, very few fish)
  elsif (p > 140.0) use
    p'dot == - 75.0 / timescale;        -- ascending up to 4m
  elsif (p > 130.0) use
    p'dot == - 0.5 / timescale;        -- diving in the epilimnion
  elsif (p > surfacePressure) use       -- (much fish, crayfish and insects)
    p'dot == - 4.0 / timescale;         -- ascending to the surface
  else
    p == surfacePressure;               -- diving ends at the surface
  end use;
  Temp == LakeTemp(surfaceTemp, surfacePressure, Metalimnion, p);
end beh_LakeDive;
```

Beschreibung 5.2: Umgebungsdruck und Umgebungstemperatur bei einem typischen Baggerseetauchgang.

```
library disciplines;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;

architecture beh_SeaDive of vdc_sources is
  constant startPressure : pressure := 100.0;  -- (100 kPa)
  constant startTemp     : temperature := 25.0;  -- (25 °C)

  quantity P across  P_flow through  Pressure_sensor;
  quantity Temp across  T_P through  Temp_sensor;
  quantity waterPressure : pressure;

begin
  if (now < 3.0 * timescale) use
    waterPressure ==  250.0 / 3.0 * now / timescale;
  elsif (now < 28.0 * timescale ) use
    waterPressure ==  250.0;
  elsif (now < 29.6 * timescale ) use
    waterPressure ==  250.0 - 100.0 * (now - 28.0) / timescale;
  elsif (now < 33.0 * timescale) use
    waterPressure ==  90.0;
  elsif (now < 33.3 * timescale) use
    waterPressure ==  90.0 - 100.0 * (now - 33.0) / timescale;
  elsif (now < 37.0 * timescale) use
    waterPressure ==  60.0;
  elsif (now < 37.3 * timescale) use
    waterPressure ==  60.0 - 100.0 * (now - 37.0) / timescale;
  elsif (now < 45.0 * timescale) use
    waterPressure ==  30.0;
  elsif (now < 45.3 * timescale) use
    waterPressure ==  30.0 - 100.0 * (now - 45.0) / timescale;
  else
    waterPressure == startPressure;
  end use;

  p == startPressure + waterPressure;
  Temp == startTemp - temperature(waterPressure/100.0
                                  * (1.0 + now / (40.0*timescale)));
end beh_SeaDive;
```

Beschreibung 5.3: Umgebungsdruck und Umgebungstemperatur bei einem typischen Meertauchgang.

N. Bhatti hat ein VHDL-AMS Modell implementiert, das diese Daten einliest [6]. Die eingelesenen Werte beschreiben eine Zustandsgröße eines sequenzorientierten Modells der α-Klasse. Mit Hilfe von zusätzlichem Systemwissen wurde ein zeitkontinuierliches Modell der β-Klasse in VHDL-AMS erzeugt. Dazu wurden die Tags des sequenzorientieren Modells auf einen globalen Zeitstempel abbildet und die Zustandsgröße Druck für Zeitpunkte zwischen zwei Stützstellen linear interpoliert.

$$\mathbf{z}(t_i) = \mathrm{Seq}(\mathbf{z}, i) \quad \text{, falls } t_i = i \cdot 20s \tag{5.46}$$

$$\mathbf{z}(t) = \mathbf{z}(t_i) + \frac{t - t_i}{20\,s}(\mathbf{z}(t_{i+1}) - \mathbf{z}(t_i)) \quad \forall t \in [t_i; t_{i+1}] \tag{5.47}$$

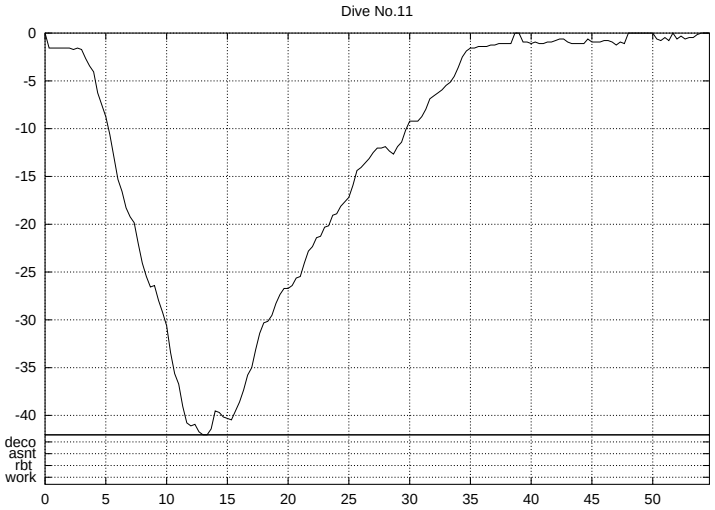


Abbildung 5.9: Tauchprofil eines Meertauchgangs, rekonstruiert aus den Messdaten eines Aladin Tauchcomputers

5.4.2 Spezifikation der Sensoren

Die Zielvorgabe des Pflichtenheftes für die Tiefenmessung ist die Erfassung der Tauchtiefe mit einer Auflösung von 10 cm. Das entspricht einer Druckänderung von 1 kPa. Wenn man die Sättigungsvorgänge im Körper des Tauchers betrachtet (siehe Kapitel 5.3), stellt man fest, dass insbesondere für den Vergleich des Umgebungsdrucks mit $p_{Env,\min}$ und für die Berechnung der Dekompressionsrate eine genaue Erfassung des Umgebungsdrucks in der Nähe der Dekostopps notwendig ist. Für den Messbereich zwischen Oberflächendruck und 10 m Wassertiefe wird deshalb ein Messungenauigkeit von unter 10 cm gefordert. Der Gesamtmessbereich muss so gewählt werden, dass der Taucher bei jedem denkbaren Tauchgang innerhalb dieses Bereichs taucht. Der Verband der deutschen Sporttaucher hat 40 m als maximale Tiefe für Sporttaucher festgelegt. Es existieren jedoch Berichte von Tauchern, die z. B. im Bodensee Tauchgänge in Tiefen von über 80 m überlebt haben. Für den Dekompressionscomputer soll deshalb ein linearer Messbereich für Drücke bis 1100 kPa realisiert werden.[6]

Für die Implementierung des Drucksensors bietet sich ein piezoresistiver Siliziumsensor an. Diese Sensorelemente sind auch als diskrete Sensorelemente verfügbar. Für die Gesamtsystemmodellierung wird ein existierender Sensor mit Gehäuse gesucht. Es gibt

[6] In 100 m Tiefe herrscht ein Druck von etwa 1100 kPa.

	SCC300A	XSX150A	AMS 5310-100
linearer Messbereich $[kPa]$	0 – 2070	0 – 1035	0 – 690
max. Druckbelastung $[kPa]$	3100	3100	1550
Offsetspannung $[mV]$	-30 – 20	-50 – 50	-50 – 50
Druckempfindlichkeit $[\frac{\mu V}{kPa}]$	24 – 58	180 – 325	190 – 360
Ausgangs-Innenwiderstand $[k\Omega]$	5,0 (4,0 - 6,5)	3,0	3,3 (2,7 - 4,0)
Temperaturkoeffizient des Innenwiderstands $[\frac{\%}{°C}]$	0,22		0,28 $\pm$ 5
Temperaturkoeffizient der Druckempfindlichkeit $[\frac{\%}{°C}]$	-0,22	-0,205	-0,22 $\pm$ 5

Tabelle 5.4: Auszüge aus den Datenblättern einiger Drucksensoren: Technische Eigenschaften bei einer Betriebsspannung von 5V

mehrere Hersteller von geeigneten Sensoren, so dass eine Modellierung der Sensoreigenschaften auf Basis dieser Produkte möglich ist. Tab. 5.4 fasst die technischen Eigenschaften des SCC300A [81], des XSX150A [32] und des AMS53100-100 [1] zusammen. Das Sensormodell in Beschreibung 5.4 beschreibt den 2000 kPa Sensor SCC300A.

Die Sensoren arbeiten wie eine Wheatstone-Brücke, in der die Widerstände druckabhängig sind. Legt man eine konstante Eingangsspannung an, so ergibt sich zwischen den Ausgängen $U_{Out,P}$ und $U_{Out,N}$ ein differentielles Spannungssignal, das sich in erster Näherung proportional zum Umgebungsdruck ändert. Die Sensoren haben starke Exemplarstreuungen, d. h. die Druckempfindlichkeit und der Ausgangsspannungsoffset ist bei jedem Sensor unterschiedlich. Diese Streuungen werden durch mechanische Verspannungen des Sensors erzeugt, die bei der Verpackung des Sensorelements in ein Gehäuse auftreten. So beträgt z. B. für die den Sensor AMS5310-100 (s. Tab. 5.4) der Offsetfehler bis zu $\pm 50\,mV$, während die Druckempfindlichkeit des Sensors zwischen $190\,\frac{\mu V}{kPa}$ und $360\,\frac{\mu V}{kPa}$ liegt. Wenn die Druckempfindlichkeit am unteren Grenzwert liegt, entspricht der maximale Offsetfehler einer Verstimmung um 265 kPa, oder *einer Abweichung der gemessenen Wassertiefe um 26,5 m*. Der Offsetfehler und die Empfindlichkeit des Sensors können durch eine Kalibrierung ermittelt werden, so dass eine Korrektur des Messergebnisses durch eine Signalnachbearbeitung im Digitalteil möglich wird. Der Fehler bei der Bestimmung der Wassertiefe soll dadurch auf etwa 0,1 m reduziert werden.

Leider sind die piezoresistiven Siliziumsensoren temperaturempfindlich, das Messergebnis kann durch Temperaturschwankungen um mehr als 8% verfälscht werden (siehe Abb. 5.10). Da der Dekompressionscomputer bei unterschiedlichen Umgebungstemperaturen eingesetzt wird, muss während der Druckmessung ebenfalls die Temperatur erfasst werden. In die Spezifikation wird deshalb die Temperaturerfassung mit aufgenommen.

```
library IEEE;
use ieee.math_real.all;

library disciplines;
use disciplines.electromagnetic_system.all;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;

entity vdc_psens is
  generic (
    k0      : real := 25.0;     -- V/kPa
    UV      : emf  := 5.0;       -- V
    UOffset : emf  := -0.01);    -- V
  port(terminal P_In : fluidic;
       terminal T_In : thermal;
       terminal U_OutP, U_OutN : electrical;
end vdc_psens;

architecture simple of vdc_psens is
  quantity k1, k2 : real;
  quantity P across  P_In;
  quantity Temp across  T_In;
  quantity Uout_V across  Uout_I through  U_OutP to U_OutN;
  quantity U_OutP_V across  U_OutP_I through  U_OutP;
begin
  k1 == (1.0 + 1.32e-4 * exp(7.98e-3 * P)
            + 1.43e-4 * exp(-8.7e-3 * P));
  k2 == (1.0 + 9.30e-4 * exp(-0.083 * Temp)
            + 1.60e-2 * exp(0.0139 * Temp));
  Uout_V == UV * k0 * k1 * k2 * P + UOffset;
  U_OutP_V == (Uout_V + UV) / 2.0;
end simple;
```

Beschreibung 5.4: Modell des Drucksensors SCC300A für die Gesamtsystemsimulation

Die Korrektur des Temperaturdrifts[7] kann entweder im analogen Teil der Schaltung oder nach einer A/D Wandlung im digitalen Teil der Schaltung erfolgen. Mit Hilfe von analogen Signalumformern [65] kann der Einfluss der Temperatur in erster Näherung ausgeglichen werden. Dies erfordert jedoch eine mechanische Kalibrierung jedes einzelnen Gerätes. Deshalb wurde die Korrektur der Messergebnisse in die digitale Schaltung verlegt, mit der Absicht das fertige Gerät bei der Inbetriebnahme mit den digitalen Korrekturkoeffizienten zu versehen. Als weiteren Nutzen ergibt sich aus dieser Entwurfsentscheidung die Möglichkeit den Offsetfehler und die Empfindlichkeit des Sensors kostengünstig digital zu kompensieren.

Zur Temperaturmessung wird ein Sensor benötigt, der für den Temperaturbereich der Anwendung geeignet ist und eine für die Korrektur des Drucksensors ausreichende Genauigkeit hat. Diese Randbedingungen sind leicht zu erfüllen. Der Temperaturbereich

[7] Temperaturdrift: Wertänderung der Messgröße aufgrund einer Temperaturänderung

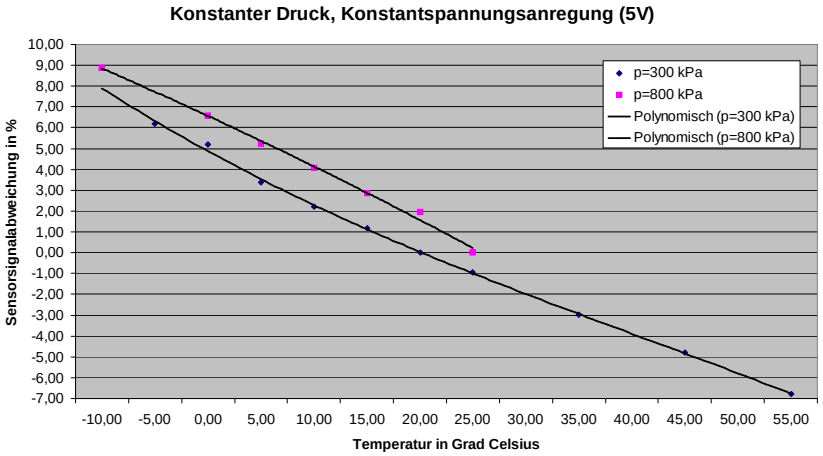


Abbildung 5.10: Temperaturabhängigkeit eines untersuchten Drucksensors bei 300 kPa und 800 kPa Druck

des Wassers liegt zwischen 0 °C und 40 °C, die untere Grenze wird durch den Gefrierpunkt des Wassers bestimmt, die obere Grenze ist durch den menschlichen Taucher vorgegeben. Da der Körper des Tauchers die Körpertemperatur hauptsächlich durch Verdunstung erniedrigt und dieser Mechanismus im Wasser nicht funktioniert, führt eine Wassertemperatur von über 37 °C zu einer Überhitzung des Tauchers. Der Messbereich für den Temperatursensor wird etwas größer gewählt, so dass die Temperaturmessung sofort funktioniert, wenn das an der Wasseroberfläche auf Lufttemperatur akklimatisierte Gerät in Wasser eingetaucht wird. Letztendlich wird ein Temperatursensor benötigt, der im Messbereich von -10°C bis 50°C eine zur Temperatur proportionale Spannung liefert. Der Messfehler soll 1 °C nicht übersteigen.

Dieser Temperatursensor kann mit einer integrierten CMOS Schaltung implementiert werden. In der Machbarkeitstudie wird jedoch ein einfaches, leicht zu verstehendes Modell gesucht, so dass die Modellierung eines diskreten temperaturempfindlichen Widerstandes besser geeignet ist. Das Modell aus Beschreibung 5.5 beschreibt einen Spannungsteiler aus einem temperaturempfindlichen und einem nicht temperaturempfindlichen Widerstand. Der temperaturempfindliche Widerstand ist einem handelsüblichen Platin-Dünnschichtsensor [28] nachempfunden. Der Temperaturkoeffizient von $3,85 \cdot 10^{-3}/°\mathrm{C}$ entspricht der IEC 751 Norm. Dieses Sensorelement ist im Automotive Bereich weit verbreitet, so dass eine gute Verfügbarkeit garantiert ist.

```
library IEEE;
use ieee.math_real.all;
library disciplines;
use disciplines.electromagnetic_system.all;
use disciplines.Thermal_system.all;

entity vdc_tsens is
  generic (
    R0   : real := 1000.0;
    Rfix : real := 1000.0;
    UV   : emf := 5.0;
    Tk   : real := 3.85e-3);
  port(terminal T_In : thermal;
       terminal Uout : electrical);
end vdc_tsens;

architecture simple of vdc_tsens is
  quantity R1 : real;
  quantity Temp across T_In;
  quantity Uout_V across  Uout_I through  Uout;
begin
  R1 == R0 * (1.0 + Tk * Temp);
  Uout_V == UV * R1 / (R1 + Rfix);
end simple;
```

Beschreibung 5.5: Modell eines Temperatursensors, der aus einer Spannungsquelle und einem temperaturempfindlichen und einem konstanten Widerstand aufgebaut ist.

5.4.3 Aufbereitung der analogen Sensorsignale und A/D Wandlung

Bevor die Sensordaten im Mikroprozessor des *v*DC verarbeitet werden können, müssen die analogen Daten mit einem A/D Wandler konvertiert werden. Die Anforderungen an den A/D Wandler ergeben sich zum Einen aus den Sensordaten, zum Anderen aus den Anforderungen des Dekompressionsalgorithmus.

Wie in Tab. 5.4 dargestellt, ist die Empfindlichkeit des Drucksensors großen Produktionsschwankungen unterworfen. Da die digitalisierte Information die Tiefenangabe mit einer Genauigkeit von 0,1 m enthalten soll, muss ein LSB des digitalisierten Ergebnisses einer Druckänderung von 1 kPa entsprechen.

Die kleinste zu konvertierende Spannung ergibt sich aus dem Druck an der Wasseroberfläche an einem Bergsee[8] und dem minimalen Offsetfehler des Sensors, während sich die größte zu konvertierende Spannung aus dem Druck 100 m unter der Meeresoberfläche, bei maximaler Sensorempfindlichkeit und maximalem Offsetfehler, ergibt.

[8] Als höchster Bergsee wird ein See auf 4000 m über dem Meeresspiegel angenommen. Der Luftdruck in dieser Höhe beträgt etwa 60 kPa.

	SCC300A	XSX150A	AMS 5310-100
$\min(U_p)$	-28,5 mV	-39,2 mV	-38,6 mV
$\max(U_p)$	83,8 mV	407,5 mV	446 mV
benötigte dig. Auflösung	4682	2482	2551
benötigte dig. Auflösung in Bit	12,2	11,3	11,3

Tabelle 5.5: Eingangswertebereich der analogen Signalverarbeitung für die verschiedenen Drucksensoren. Die benötigte digitale Auflösung ist für eine Messauflösung von 0,1m berechnet.

Für den Drucksensor AMS5310-100 ergibt sich:

$$\min(U_p) = 60kPa \cdot \frac{190\mu V}{kPa} - 50mV = -38,6mV \tag{5.48}$$

$$\max(U_p) = 1100kPa \cdot \frac{360\mu V}{kPa} + 50mV = 446mV \tag{5.49}$$

Mit der geringsten spezifizierten Druckempfindlichkeit von $190\frac{\mu V}{kPa}$ wird somit ein LSB einer Eingangsspannung von $190\mu V$ entsprechen:

$$p_{dig} = \left\lfloor \frac{U_p}{190\mu V} \right\rfloor + k_1 \tag{5.50}$$

Mit diesen Angaben kann die benötigte Auflösung des digitalisierten Signals berechnet werden:

$$\max(p_{dig}) - \min(p_{dig}) = \left\lfloor \frac{446mV + 38,6mV}{190\mu V} \right\rfloor = 2550 \tag{5.51}$$

Tab. 5.5 zeigt den benötigten Eingangsspannungsbereich und die Anzahl der Diskretisierungsschritte für alle in Kapitel 5.4.2 vorgestellten Drucksensoren.

Ein weiteres wichtiges Kriterium für die Auswahl eines A/D Wandlers ist die benötigte *Samplerate*. In Kapitel 5.3 wurden die Halbwertszeiten der Gewebesorten vorgestellt, deren Sättigung mit den zeitdiskret abgetasteten Sensordaten berechnet werden soll. Das Gewebe, das am schnellsten auf eine Druckänderung reagiert, hat eine Halbwertszeit von 4 Minuten. D. h. in einem Zeitintervall von 0,5 s wird das Gewebe sich um weniger als 0,15 % der Druckdifferenz aufsättigen. Bei einem 'Direktabstieg' von der Wasseroberfläche auf eine Tiefe von 70 m würde sich das schnellste Gewebe in dem Zeitintervall von 0,5 s um etwa 1 % ändern. Die tatsächliche Änderung der Gewebesättigung wird durch eine lineare Interpolation des Druckverlaufs zwischen zwei

Diskretisierungszeitpunkten approximiert, so dass eine Abweichung zwischen berechneter Gewebesättigung und tatsächlicher Gewebesättigung in diesem Beispiel maximal 0,5 % betragen kann. Eine so geringe Abweichung kann akzeptiert werden, so dass eine Samplerate von wenigen Hertz ausreichend erscheint.

Die Anforderungen an die Samplerate des A/D Wandlers sind so gering, dass sie von allen in Frage kommenden A/D Wandlerarchitekturen erreicht werden. Die Auflösung von 12 bis 13 Bit bei einer Spannungsänderung von einigen μV pro LSB ist dagegen ein entscheidendes Spezifikationsmerkmal. Im Sinne der Miniaturisierung des vDC wird ein A/D Wandler bevorzugt, der auf dem selben Substrat wie der Drucksensor implementiert werden kann. Damit bietet sich eine integrierte Schaltung mit MOS Transistoren als A/D Wandler an. Um die Chipfläche klein zu halten, soll ein Δ-Σ A/D Wandler implementiert werden, bei dem Samplerate gegen Genauigkeit getauscht werden kann.

Δ-Σ A/D Wandler sind in den letzten 20 Jahren sehr beliebt geworden, da bei diesen Architekturen durch digitale Nachbearbeitung Ungenauigkeiten im analogen Teil ausgeglichen werden können. Das ermöglicht eine industrielle Produktion mit guter Ausbeute in günstigen IC-Prozessen.

Ein Δ-Σ A/D Wandler besteht im allgemeinen aus drei Blöcken: Dem Δ-Σ Modulator, einem schnellen Decimation-Filter und einem nachgeschalteten weiteren Filter hoher Ordnung (s. Abb. 5.11). Im Allgemeinen ändert sich in jeder Stufe des Wandlers die Datenrate und die Breite der Datenworte. Im DSP-Core werden die Daten entsprechend der enthaltenen Nutzsignalfrequenzen und des Nyquist-Theorems mit einer Datenrate von f_N verarbeitet. Der vorgeschaltete Tiefpass erzeugt diese Daten i. d. R. aus einem 4 bis 8 mal schnelleren Signal.

Vor dem Tiefpassfilter ist ein schneller und platzsparender Decimation Filter, der einen hohen Downsampling-Faktor bei geringer Rauschsignal-Einkopplung ermöglichen muss. Im Decimation-Filter wird das Ausgangssignal des Modulators i. d. R. zwischen 8- und

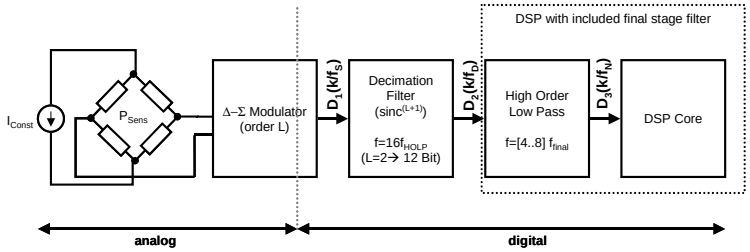


Abbildung 5.11: Generische Architektur eines Δ-Σ A/D Wandlers

64-fach reduziert. Daraus ergibt sich ein Oversampling-Faktor für den Δ-Σ Modulator, der zwischen 32 und 512 liegt. Die analogen Komponenten des Modulators müssen passend zur benötigten Samplingrate dimensioniert werden. So wird beispielsweise für einen Audio-A/D Wandler, der ein Signalspektrum bis 48 kHz erfassen soll, bei 128-fach Oversampling eine Samplingrate (f_S) von etwa 6 MHz benötigt.

Für die Anwendung im *v*DC genügt eine moderate Samplingrate von 1 MHz, um bei einem Oversampling-Faktor von 256 eine Ausgangssamplerate von 4 kHz zu erreichen, was weit über der benötigten Mindestsamplerate liegt.

Der Eingangswertebereich eines Δ-Σ A/D Wandler ist aufgrund von Stabilitätsanforderungen an den Δ-Σ Modulator beschränkt, wodurch ebenfalls eine Beschränkung des Ausgangswertebereichs entsteht. Wenn die Eingangsspannung zu klein oder zu groß ist, kann der Δ-Σ Modulator überladen werden. D. h. an einer Stelle im Modulator kann keine zusätzliche Ladung auf einem Kondensator gespeichert werden, obwohl dies für die korrekte Funktion des Modulators notwendig wäre (*Integrator Overload Noise*). Der Fehler entsteht, weil die enthaltenen Operationsverstärker einen begrenzten Ausgangsspannungsbereich haben. Mit steigender Frequenz des Eingangssignals werden diese Fehler wahrscheinlicher. Bei einer Erhöhung des Oversampling-Faktors wird die Auswirkung dieses Fehlers etwas gedämpft. Um Fehler, bzw. Rauschen durch Überladung des Modulators, systematisch zu verringern, muss man den Eingangsspannungsbereich reduzieren, so dass der Betrag der Feedbackspannung deutlich größer ist als der Betrag der maximalen Eingangsspannung. Das führt jedoch dazu, dass die Tiefpass-gefilterte Bitfolge des Modulatorausgangs nie den theoretischen Maximalwert erreichen kann.

Der Ausgangswertebereich des Δ-Σ A/D Wandler ist zusätzlich aufgrund eines *Code Noise* genannten Rauschens eingeschränkt: Wie in [62] beschrieben, kommt es in Δ-Σ Modulator mit binärem Ausgangssignal zu einem als *Code Noise* bezeichneten Rauschen, das vom Eingangssignal abhängig ist. Das Ausgangssignal des Δ-Σ Modulators ist pulsweitenmoduliert, d. h. die Nutzinformation erhält man durch eine Integration des binären Modulatorausgangssignals y. Für die weiteren Betrachtungen wird ohne Beschränkung der Allgemeinheit ein normiertes Ausgangssignal $Y \in [0;1]$ verwendet:

$$Y(t) = \frac{1}{t - t_0} \int_{t_0}^{t} y(t)dt \tag{5.52}$$

Für die Kodierung des größtmöglichen Y wird $y(t) = 1 \forall t$ benötigt, für die Kodierung des kleinstmöglichen Y wird $y(t) = 0 \forall t$ benötigt. Das Modulatorausgangssignal ist in diesen Fällen eindeutig. Für $Y \approx 0.5$ gibt es jedoch eine Vielzahl von Modulatorausgangssignalfolgen, so dass der Approximationsvorgang nach einer kürzeren Zeitspanne abgeschlossen ist. In [62] wurde gezeigt, dass die Eingangssignalabhängigkeit des Code Noise für $0.2 < Y < 0.8$ kleiner 3 dB ist. Für Eingangssignale außerhalb dieses Bereichs steigt der Rauschpegel exponentiell an.

Um $0.2 < Y < 0.8$ sicherzustellen, muss das analoge Eingangssignal x(t) am Modulator ebenfalls klein genug sein. Mit den Feedbackspannungen $\pm\frac{\Delta_V}{2}$ ergibt sich:

$$-0.6\frac{\Delta_V}{2} < x(t) < 0.6\frac{\Delta_V}{2} \tag{5.53}$$

$$-0.3\Delta_V < x(t) < 0.3\Delta_V \tag{5.54}$$

Der nutzbare Ausgangswertebereich ist somit auf Werte zwischen ungefähr 20% - 80% des maximal möglichen Ausgangswertes beschränkt. Wenn man den Wertebereich um ein Prozent erweitert, liegt bei einem digitalen Codewort mit 12 Bit Länge der Ausgangswertebereich ungefähr im Intervall $[770; 3326]$. Das Intervall ist 2556 Werte breit, so dass die Sensorinformation der Drucksensoren XSX150A und AMS5310-100 in diesem Bereich übertragen werden können. Für den SCC300A wird ein 13 Bit Codewort benötigt, bei dem der Ausgangswertebereich ungefähr im Intervall $[1470; 6720]$ liegt.

Die ausführbare Spezifikation des A/D Wandlers soll eine Referenzimplementierung sein, die einfach und überschaubar ist. Sie kann beispielsweise aus einem idealen A/D Wandler bestehen. Der ideale A/D Wandler hat folgende Eigenschaften:

- Lineares Verhalten, d. h. es treten keine Konvertierungsfehler im digitalen Ergebnis auf.
- Keine Modellierung des Zeitverhaltens, d. h. das Modell kann nur eingesetzt werden, wenn die Konvertierungszeit vernachlässigbar ist.
- Keine interne Zustände, somit werden Korrelationen zwischen zeitlich benachbarten Konvertierungsergebnissen vernachlässigt.
- Stets betriebsbereit, d. h. das Modell hat keine Initialisierungsphasen. Es liefert auf jede Anfrage ein gültiges Ergebnis.

Zur Modellierung eines solchen A/D Wandlers bietet sich die α-Klasse an. Beschreibung 5.6 zeigt eine Implementierung als VHDL-Funktion. Ist dieses Modell gut? Es modelliert keine Einschwingvorgänge, es ist nicht geeignet, um einen schnellen Flash-A/D Wandler von einem langsamen Δ-Σ A/D Wandler zu unterscheiden; es gibt keine Möglichkeit Nichtlinearitäten eines A/D Wandlers zu berücksichtigen. Darüber hinaus ist es im Vergleich zu einem physikalischen Testaufbau zu ungenau - sowohl im zeitlichen Verhalten, als auch im quasistationären Betriebsfall. Phänomene wie Oszillation zwischen zwei digitalen Werten können durch Rauschen auf dem Eingangssignal erzeugt werden; es ist jedoch nicht möglich eine Hysterese einzustellen.

Das Modell ist dennoch sinnvoll: Es ist klein, überschaubar und verständlich. Es ist simulationseffizient, da es rückkopplungsfrei ist und nur wenige Zuweisungen evaluiert werden müssen. Es ist trivial und beinhaltet wenig Fehlerquellen: Es eignet sich somit zur automatischen Validierung, da es mit wenig Aufwand verbunden ist, im Fehlerfall

```
function Convert_AnalogToDigital
  ( U_min : real; U_max : real;
  BitRes  : integer; U_in : real)
  return std_logic_vector is

  variable r     : real;
  variable i     : integer;
  variable u     : unsigned(BitRes-1 downto 0);
begin
  r := (U_in-U_min))/(U_max-U_min);
  i := integer(r*(2.0**BitRes));
  u := conv_unsigned(i, u'length);
  return std_logic_vector(u);
end Convert_AnalogToDigital;
```

Beschreibung 5.6: Algorithmisches Modell eines generischen A/D Wandlers

den Defekt auf das DUT zurückzuführen. Das Modell ist universell einsetzbar, es ist für beliebige Bitbreiten konfigurierbar und modelliert das quasistationäre Verhalten beliebiger A/D Wandler. Es kann zur Stimulierung digitaler Schaltungen eingesetzt werden, ohne die Digitalsimulation durch einen Analogsolver zu bremsen. Dieses Modell wird Ausgangspunkt für die Implementierung des Δ-Σ A/D Wandlers.

Der Eingangsspannungsbereich des Modells wird durch die Eingangsgrößen U_{min} und U_{max} bestimmt. Diese Größen können aus den gewünschten digitalen Codeworten für die kleinste und die größte Eingangsspannung berechnet werden. Die folgende Berechnung erläutert den Algorithmus am Beispiel des AMS5310-100 Sensors:

$$\begin{aligned}
\min(p_{dig}) = 773 &= (\min(U_p) - U_{min})/(U_{max} - U_{min}) \cdot 2^{12} \\
&= (-38,6mV - U_{min})/(U_{max} - U_{min}) \cdot 2^{12} && (5.55)\\
\max(p_{dig}) = 3323 &= (\max(U_p) - U_{min})/(U_{max} - U_{min}) \cdot 2^{12} \\
&= (446mV - U_{min})/(U_{max} - U_{min}) \cdot 2^{12} && (5.56)\\
773(446mV - U_{min}) &= 3323(-38,6mV - U_{min}) && (5.57)\\
U_{min} &\approx -185,5mV && (5.58)\\
U_{max} &\approx 592,9mV && (5.59)
\end{aligned}$$

Das Modell des A/D Wandlers zeigt die gewünschte Funktionalität, wenn man für U_{min} und U_{max} obige Werte einsetzt. Diese Werte müssen für jeden Drucksensor erneut berechnet werden.

Die Berechnung dieser Werte von Hand ist jedoch aufwendig und die Berechnungsvorschrift muss sorgfältig dokumentiert werden. Eine Änderung des Modells um diese

```
function Convert_AnalogToDigital_DSM_org
  ( U_min : real; U_max : real;
  BitRes  : integer; U_in : real)
  return std_logic_vector is
  variable U_low : real;
  variable U_high: real;
begin
  U_range := (U_max-U_min) / (0.78-0.18);
  U_low  := U_min - 0.18*U_range;
  U_high := U_low + U_range;
  return Convert_AnalogToDigital(U_Low, U_High, BitRes, U_in);
end Convert_AnalogToDigital_DSM_org;


function Convert_AnalogToDigital_DSM
  ( U_min : real; U_max : real;
  BitRes  : integer; U_in : real)
  return std_logic_vector is

  variable r     : real;
  variable i     : integer;
  variable u     : unsigned(BitRes-1 downto 0);
  constant U_low : real := U_min-0.18/0.64*(U_max-U_min);
  constant U_high: real := U_max+0.18/0.64*(U_max-U_min);
begin
  r := (U_in-U_low)/(U_high-U_low);
  i := integer(r*(2.0**BitRes));
  u := conv_unsigned(i, u'length);
  return std_logic_vector(u);
end Convert_AnalogToDigital_DSM;
```

Beschreibung 5.7: Algorithmische Beschreibung des Ausgangssignals des Delta-Sigma A/D Wandlers.

komplizierte Berechnung zu vermeiden, ist im Hinblick auf die Wiederverwertbarkeit der Beschreibung sinnvoll. Dazu wird eine erweiterte Funktion (Convert_AnalogToDigital_DSM) definiert, die im Anschluss mit einigen Refactoring-Methoden optimiert wird. Beschreibung 5.7 zeigt die Originalbeschreibung und das Ergebnis des Refactorings.[9]

5.4.4 Spezifikation der digitalen Signalverarbeitung

Der Dekompressionscomputer benötigt einen Digitalteil, in dem der in Kapitel 5.3 vorgestellte Algorithmus ausgeführt werden kann. Zu diesem Zweck soll der Digitalteil einen programmierbaren applikationsspezifischen Mikroprozessor enthalten. Zwei entscheidende Merkmale dieses Mikroprozessors bestehen aus dem Rechenwerk und dem Sprachumfang des Befehlssatzes mit dem der Dekompressionsalgorithmus nachgebildet werden muss.

[9] Rfact. 30 *Inline Function or Process*, Rfact. 9 *Remove Symbolic Constant*, Rfact. 45 *Refine Simplified Model*

Für eine genaue Berechnung der Gewebesättigung der 16 Gewebesorten muss in kurzen Abständen eine Aktualisierung der Gewebesättigung unter Berücksichtigung des aktuellen Umgebungsdrucks stattfinden. Hierfür wurde ein C-Programm erstellt, dass den Dekompressionsalgorithmus berechnet. So dass bei geeigneter Anregung eine Simulation des Tauchgangs mit einem überschaubaren algorithmischen Rechenmodell durchgeführt werden kann.

Wenn man die Sättigungsvorgänge im Körper des Tauchers gemäß Gl. (5.22) und Gl. (5.23) in einer Simulation berechnet, so fällt auf, dass zu kurze Update-Intervalle zu sehr geringen Veränderungen des Gewebedrucks führen. Diese geringen Veränderungen können bei begrenzter Rechengenauigkeit durch Rundungsfehler verfälscht werden. Ein Beispiel verdeutlicht dieses Problem:

Die Nullzeit der Gewebesorte mit der Halbwertszeit $\tau_{G10} = 146\,min$ sei 0, d. h. aufgrund des bisherigen Tauchgangs ist die Gewebesättigung so hoch, dass bei einer weiteren Aufsättigung kein direkter Aufstieg zur Wasseroberfläche mehr möglich ist. Der Taucher befinde sich in einer Wassertiefe von 10 m.

$$\begin{aligned}
&p_{Env,\min}(G10, p_{G10}(t_0)) := p_{Env}(0m) \\
\rightsquigarrow &p_{G10}(t_0) = \frac{p_{Env}(0m)}{b_{G10}} + a_{G10} = 143,44kPa \\
&p_{N_2}(t_0) := p_{N_2}(10m) = 0,78 \cdot (200kPa - 6,27kPa) = 151,11kPa
\end{aligned}$$

Unter der Annahme, das der Taucher in dieser Tiefe verweilt kann Gl. (5.23) als Iterationsformel zur Berechnung der neuen Gewebesättigung eingesetzt werden:

$$\begin{aligned}
p_{G10}(t_0 + \Delta t) - p_{G10}(t_0) &= (p_{N_2}(t_0) - p_{G10}(t_0)) \left(1 - 2^{-\frac{\Delta t}{\tau_{G10}}}\right) \\
&= 7,67kPa \cdot \left(1 - 2^{-\frac{\Delta t}{8760s}}\right)
\end{aligned} \tag{5.60}$$

Δt	$1\mu s$	$1ms$	1 s	10 s
Δp_{G10}	$6 \cdot 10^{-10}kPa$	$6 \cdot 10^{-7}kPa$	$6 \cdot 10^{-4}kPa$	$6 \cdot 10^{-3}kPa$

Tabelle 5.6: Änderung der Gewebesättigung in Abhängigkeit von der Iterationsschrittweite

Wie man in Tab. 5.6 sieht, ist es nicht sinnvoll die Integrationsschrittweite Δt zu minimieren. Wenn man die Gewebesättigung in einer Festkommazahl mit einer Genauigkeit von 23 Bit speichert und die maximale Gewebesättigung dem Stickstoffpartialdruck in 100 m Tiefe entspricht, so ist die Rechengenauigkeit, bzw. das kleinste Δp durch die folgende Gleichung bestimmt:

$$\min(\Delta t) = \frac{0.78 \cdot 1100kPa}{2^{23}} = 10^{-4}kPa$$

Mit einer Integrationsschrittweite von 1 ms würden die Änderungen in der Gewebesättigung durch Rundungsfehler vollständig ausgelöscht werden, so dass das Gewebe nicht dekopflichtig werden würde.

Dieses Problem kann durch die frühzeitige Beschreibung des konzeptionellen Systems als ausführbare Beschreibung schon während der Machbarkeitsstudie erkannt werden. Um dieses Problem zu lösen, wird unter der Annahme, dass die Änderung von p_{Gew} während k Integrationsschritten vernachlässigt werden kann, aus Gl. (5.22) eine Iterationsvorschrift hergeleitet, bei der die Gewebesättigung erst nach $k\Delta t$ aktualisiert wird:

$$\begin{aligned}
t_{i+1} &= t_i + \Delta t \\
\Delta p_{Gew}(t_i) &= p_{Gew}(t_i) - p_{Gew}(t_{i-1}) \\
c_1 &= 1 - 2^{-\frac{\Delta t}{\tau_{Gew}}} \\
c_2 &= \Delta t - \frac{\tau_{Gew}}{\ln 2} c_1 \\
\rightsquigarrow \Delta p_{Gew}(t_{i+1}) &= -c_1 p_{Gew}(t_i) + c_1 p_{N_2}(t_i) + c_2 \dot{p}_{N_2}(t_0)
\end{aligned}$$

Mit $p_{Gew}(t_i) = p_{Gew}(t_0)$ für $i \in [0;k]$:

$$\rightsquigarrow p_{Gew}(t_0 + k\Delta t) = (1 - c_1 k) p_{Gew}(t_0) + c_1 \sum_{i=1}^{k} p_{N_2}(t_i) + c_2 \sum_{i=1}^{k} \dot{p}_{N_2}(t_0) \quad (5.61)$$

Indem man k an die Halbwertszeiten der Gewebesorten anpasst, kann nun ein numerisch stabiles Modell formuliert werden.

Die Zeit Δt wird so gewählt, dass der Druckverlauf während des Tauchgangs korrekt nachvollzogen werden kann. Das ist bis zu einer Schrittweite von 0,5 s sichergestellt. Die Taktrate des Mikroprozessors, der in einem $0,8\mu m$-CMOS Prozess implementiert werden soll, muss einerseits schnell genug sein, damit der komplette Dekompressionsalgorithmus in der Zeit Δt abgearbeitet werden kann. Andererseits soll der Hardwareentwurf des Prozessors nicht durch eine zu große Taktrate unnötig kompliziert werden. Als Kompromiss wurde eine Taktrate von 2 MHz festgelegt. Somit stehen für die Abarbeitung des Dekompressionsalgorithmus 1 Million Takte zur Verfügung, während für die Implementierung der benötigten arithmetischen Operationen eine Latenzzeit von bis zu 500 ns zur Verfügung steht.

Nachdem die ausführbare Beschreibung des Dekompressionsalgorithmus um die Berechnungsvorschrift aus Gl. (5.61) ergänzt ist, kann eine erneute Evaluierung durchgeführt werden. Für das Rechenwerk des *v*DC muss das Zahlenformat der Zustandsgrößen des Modells bestimmt werden. Aufgrund der Struktur des Algorithmus ist ein Fließkommazahlenformat vorteilhaft. Es entlastet zusätzlich den Applikationsingenieur, der die Firmware für den *v*DC erstellen muss: Könnte das Rechenwerk nur mit Integer-Zahlen rechnen, müsste man in der Firmware künstlich Festkommazahlen definieren und bei jedem Berechnungsschritt die Zahlen passend konvertieren. Für das Fließkommazahlenformat müssen nun die Anzahl der gültige Stellen, sowie die Größe des Wertebereichs festgelegt werden. Berechnet man ein Testszenario sowohl mit exakter Zahlendarstellung, als auch mit einem eingeschränkten Fließkommazahlenformat, so ermöglicht ein Vergleich der Ergebnisse eine schnelle Einschätzung, ob das eingeschränkte Zahlenformat geeignet ist. Für das *v*DC Projekt wurden letztendlich *Binary Single Precision Floating Point Numbers*, wie sie im IEEE Standard 754[37] definiert wurden, ausgewählt. Dieser Datentyp ist 32 Bit lang, er hat eine Mantissenlänge von 23 Bit und einen Exponentenlänge von 8 Bit, so dass Zahlenwerte zwischen $-6,8 \cdot 10^{38}$ und $6,8 \cdot 10^{38}$ dargestellt werden können.

Der Sprachumfang des *v*DC-Mikroprozessors (siehe Tab. 5.7) wird ebenfalls mit Hilfe der Beschreibung des Dekompressionsalgorithmus festgelegt. Neben den üblichen Befehlen (Vergleiche, bedingte und unbedingte Sprünge, Unterprogrammaufrufe und Stack-Operationen) benötigt der Mikroprozessor einen Satz Arithmetik-Befehle für die Fließkommaoperationen und einige Arithmetik-Befehle für Integerzahlen, die z. B. als Schleifenvariablen eingesetzt werden. Desweiteren erhält der Mikroprozessor eine 'Sleep'-Anweisung und einen Interrupt-Eingang, mit dem er alle 0,5 s aktiviert wird. Zwei Konvertierungsfunktionen, mit denen die Sensordaten in das Fließkommazahlenformat gewandelt werden, und eine Funktion, mit der die Ausgabedaten auf das Display des *v*DC übertragen werden, runden den Befehlssatz des Dekompressionscomputers ab.

Als Bitlänge der Assemblerbefehle wird 32 Bit festgelegt. Von diesen 32 Bit enthalten die ersten 6 Bit den Operationscode, die nächsten 13 Bit enthalten die Speicheradresse des ersten Operanden und die letzen 13 Bit sind für die Speicheradresse des zweiten Operanden reserviert. Bei Operationen, die weniger als 2 Operanden benötigen, bleiben die letzten Bits ungenutzt.

Für den Digitalteil existiert nun eine informelle, nicht simulierbare Spezifikation, die durch eine Implementierung des Dekompressionsalgorithmus in ANSI C ergänzt wird. Als Demonstrator für die Tragfähigkeit der Wahl des Fließkommazahlenformats und der Assemblersprache, wurde die *ausführbare Spezifikation* des Dekompressionsalgorithmus so partitioniert, dass die Grundoperationen direkt im Mikroprozessor des *v*DC nachgebildet werden können. Die weitere Entwicklung des Digitalteils als Komponente wird in [56, 82, 57, 80] ausführlich beschrieben. Das zu erstellende Modell des Digitalteils muss synthetisierbar sein, so dass in diesem Fall ein fließender Übergang von der Machbarkeitsstudie zur Komponentenentwicklung möglich ist.

Fließkommabefehle	fadd, fsub, fmult, fdiv, fexp2, flog2
Integerbefehle	add, sub, inc, dec, shl, shr
Datenflussbefehle	mov, movX (a=[b]), movY ([a]=b), push, pop
Kontrollflussbefehle	cmp, fcmp, jmpL, jmpE, jmpG, jmpNE, jmp, call, ret
vDC Befehle	jmpD (JumpIfDiving), DisUpd (DisplayUpdate), sleep, savDepth (a=intToFloat(Depth)), savTemp (a=intToFloat(Temperature))

Tabelle 5.7: Befehlssatz des *v*DC-Mikroprozessors

5.5 Komponentenentwicklung

5.5.1 Sensorik

Als Drucksensor kommt ein piezoresistiver Siliziumsensor zum Einsatz, der in einem MEMS Prozess gemeinsam mit dem Delta-Sigma Modulator gefertigt werden kann. In Kapitel 5.4.2 wurden 3 Drucksensoren vorgestellt, die im Rahmen der Machbarkeitsstudie untersucht und für das Gesamtsystem modelliert wurden. Der Sensor SCC300A hat einen geeigneten Messbereich, jedoch eine relativ geringe Druckempfindlichkeit und musste mit mehr als 12-Bit digitalisiert werden. Der interne Aufbau des XSX150A ist unklar, da er mit einem Innenwiderstand von 5 kΩ und einem Ausgangswiderstand von 3 kΩ spezifiziert ist. Der AMS5310-100 hingegen weist die übliche Charakteristik einer reinen piezoresistiven Wheatstone-Brücke auf und hat eine geeignete Druckempfindlichkeit.

Für die weitere Komponentenentwicklung wurde deshalb der AMS5310-100 Sensor als Vorbild ausgewählt.

Um das Sensormodell zu verbessern, wird ein solcher Sensor analysiert und im Temperaturschrank ausgemessen. Der Sensor ist wie der auf dem Chipfoto in Abb. 5.12 abgebildete Drucksensor aufgebaut. Die Materialdicke des Sensorelements ist an einer Stelle durch einen Ätzprozess so reduziert, dass eine Membran entsteht, die sich durch Druckbelastung biegt. Durch ein Referenzvakuum auf der einen Seite des Halbleiters, entsteht ein Absolutdrucksensor, der an der Wasseroberfläche den Luftdruck von etwa 100 kPa mißt. Je nach Größe und Dicke der Membran ändert sich der Messbereich des Sensors. Für den ausgewählten Sensor wird ein linearer Messbereich von 690 kPa garantiert, in der Applikation *v*DC soll der Messbereich jedoch auf 1100 kPa erweitert werden. Das differentielle Ausgangssignal wurde in 2 Testserien gemessen. Die erste Messung erfolgt für Temperaturen von -10 °C bis 60 °C für einen Druck bis 500 kPa (s. Abb. 5.13). Der Messaufbau für den Druckbereich bis 1100 kPa wurde bei einer Temperatur von -10 °C und +25 °C gemessen. Wie Abb. 5.14 zeigt, ist das Messsignal auch im erweiterten Messbereich in erster Näherung linear.

Die Modellbildung im Rahmen der Gesamtsystemmodellierung beruhte auf den Datenblattangaben, die eine Modellierung des Sensorausgangssignals durch eine Funktion der Form $U_{Offset} + f_1(p) \cdot f_2(T)$ nahelegt (siehe Beschreibung 5.8). Mit den Messwerten für einen der gemessenen Sensoren können die Modellparameter bestimmt werden. Dazu wird lediglich die Messreihe bei 25 °C Umgebungstemperatur aus Abb. 5.14 benötigt. Die bestmögliche lineare Approximation der Messwerte wurde in Microsoft Excel berechnet, es ergibt sich eine Druckempfindlichkeit $k_0 = 278,6 \frac{\mu V}{kPa}$ bei $U_{Offset} = -5,54mV$:

```
vdc_psens_1: entity work.vdc_psens(ams5310_simple)
  generic map (k0 => 2.78e-4, UV => 5.0, UOffset => -5.54e-3)
  port map (P_In, T_In, U_OutP, U_OutN);
```

Der Innenwiderstand des Sensors ist ebenfalls temperaturabhängig und könnte eventuell zur Messung der Temperatur verwendet werden. Er wird in Abb. 5.15 für zwei der Sensoren dargestellt. Eine Druckabhängigkeit des Innenwiderstands war nicht messbar. Die Linearität ist sehr gut, je nach Sensor ist jedoch die Geradensteigung etwas unterschiedlich. Für die beiden untersuchten Sensoren ist der Temperaturkoeffizient des Innenwiderstands ($T_{k,R}$) 8,81 $\frac{\Omega}{°C}$, bzw. 7,22 $\frac{\Omega}{°C}$. Die Abweichung der linearen Näherung liegt im Bereich von -5°C bis +55°C unter 1 °C, bei -10°C und +60°C beträgt die Messungenauigkeit etwa 1,5°C. Der Innenwiderstand des AMS5310-100 kann somit zur Temperaturmessung eingesetzt werden, so dass der zusätzliche Temperatursensor im vDC entfallen kann.

Das Modell des Drucksensors muss dafür so geändert werden, dass der Innenwiderstand gemessen werden kann. Zur Verdeutlichung der Refactoring Methodik aus Kapitel 4.6 wird diese Überarbeitung ausführlich vorgestellt.

Das in Beschreibung 5.8 gezeigte Modell basiert auf drei elektrischen Terminals, den beiden Terminals für das Sensorausgangssignal und dem *Reference* Terminal, einem

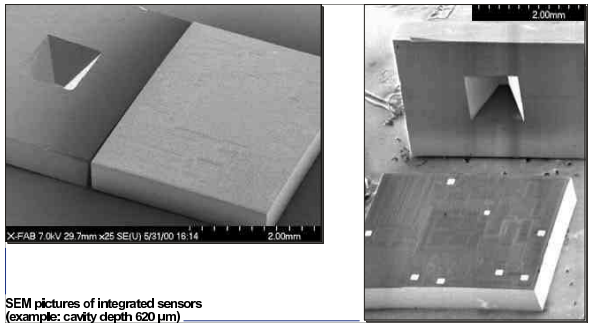


Abbildung 5.12: Aufnahme der Chip-Rückseite einer CMOS Schaltung mit integriertem mikromechanischen Drucksensor, Quelle: X-FAB Semiconductor Foundries AG

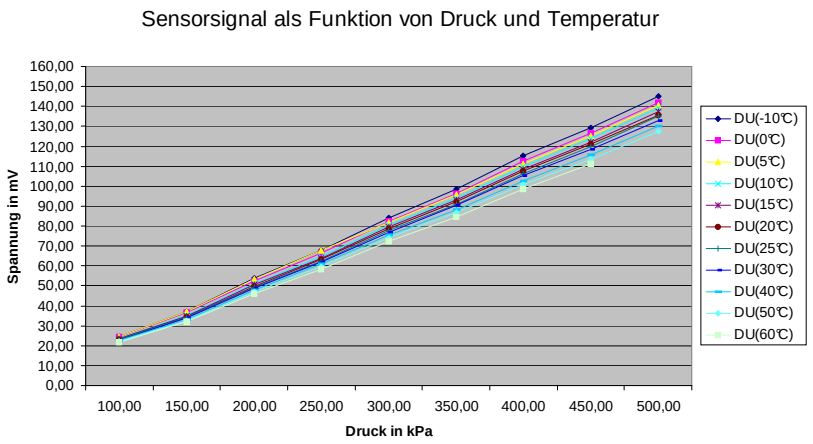


Abbildung 5.13: Sensorsignal eines AMS5310-100 bei Anregung mit einer 5V Spannungsquelle. Die Druckempfindlichkeit nimmt mit zunehmender Temperatur ab.

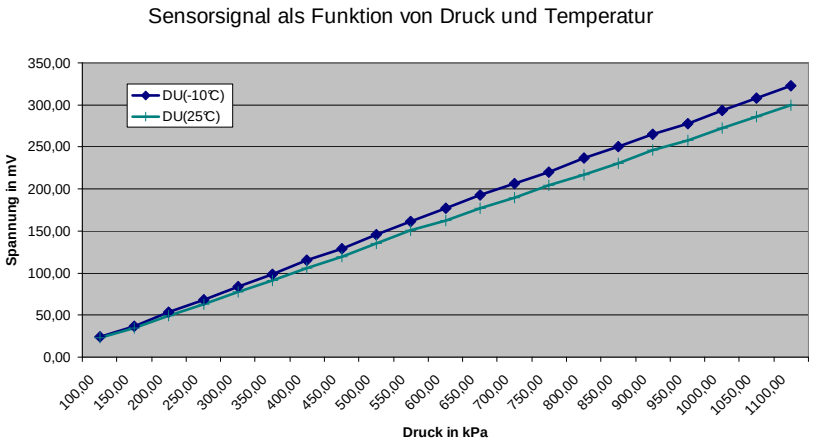


Abbildung 5.14: Sensorsignal eines AMS5310-100 bei Anregung mit einer 5V Spannungsquelle für den Druckmessbereich bis 1100 kPa

```
library IEEE;
use ieee.math_real.all;

library disciplines;
use disciplines.electromagnetic_system.all;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;

-- k0: Druck-Empfindlichkeit bei 25°C Raumtemperatur (in V/kPa)
-- UV: Spannungsversorgung des Sensors (in V)
-- UOffset: Offsetspannung bei p=0kPa, UV=5V, Temp=25°C

entity vdc_psens is
  generic (
    k0      : real;
    UV      : emf := 5.0;
    UOffset : emf := 0.0);
  port(terminal P_In : fluidic;
       terminal T_In : thermal;
       terminal U_OutP, U_OutN : electrical);
end vdc_psens;

architecture ams5310_simple of vdc_psens is
-- typische Werte der Generics:
--   k0=275.0e-6 V/kPa, UV=5V, UOffset=-0.005V
  constant TkP : real := -2.2e-3; -- pro °C
  quantity f1, f2 : real;
  quantity P across  P_In;
  quantity Temp across  T_In;
  quantity Uout_V across  Uout_I through  U_OutP to U_OutN;
  quantity U_OutP_V across  U_OutP_I through  U_OutP;
begin
  f1 == k0 * p;
  f2 == 1.0 + TkP * (Temp-25.0);
  Uout_V == f1 * f2 * UV + UOffset;
  U_OutP_V == (Uout_V + UV) / 2.0;
end ams5310_simple;
```

Beschreibung 5.8: Modell des Drucksensors AMS5310-100 aus der Spezifikationsphase.

globalen Bezugsknoten, über den das Nullpotential festlegt ist. Mit diesem Terminal wird die Spannung U_OutP_V festgelegt. Der Innenwiderstand des Sensors wird modelliert, indem das Modell um ein zusätzliches Terminal erweitert wird, das die Versorgungsspannung zur Verfügung stellt. Beschreibung 5.9 zeigt das Resultat von Rfact. 48 *Refine Dataflow*. Der Parameter UV wird jetzt dazu verwendet, die Spannung am Terminal tVP festzulegen und diese Spannung wird im Rest des Modells anstelle des Parameters UV weiterverwendet.

Diese erste Änderung des Modells ist leicht zu testen: Die Testbench für das Modell aus Beschreibung 5.8 kann weiterverwendet werden. Nun soll der Aufbau des Sensors aus vier Widerständen zur Modellierung genutzt werden. Diese Widerstände sind als Wheatstone-Brücke zwischen tVP, dem *Reference* Terminal und den Ausgangsterminals U_Out_P und U_Out_N geschaltet. Um die Modelltransformation überschaubar zu halten, wird zuerst eine Lösung für einen offsetfreien Sensor gesucht. Mit Rfact. 17 *In-*

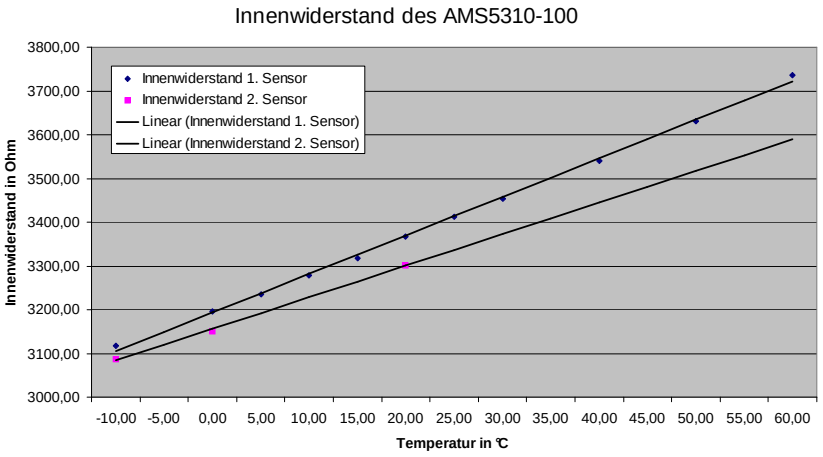


Abbildung 5.15: Innenwiderstand der AMS5310-100 Sensoren

troduce Assertion wird sichergestellt, dass eine Testbench für offsetbehaftete Sensoren als ungültig erkannt wird:

```
assert (UOffset = 0.0)
  report "P-Sensor Offset is not modeled"
  severity error;
```

Das Rfact. 50 *Substitute Implementation* ermöglicht nun eine Umstellung des Drucksensormodells auf ein widerstandsbasierte Modellierung. Das Ergebnis ist in Beschreibung 5.10 zu sehen.

Nach dem erfolgreichen Test dieses Modells wird die Offsetspannung des Sensors ergänzt. Wie in Beschreibung 5.11 zu sehen ist, wird dazu die Spannung verwendet, die in die Widerstandszweige eingeprägt wird. In Beschreibung 5.10 sind die beiden Strompfade I_RPa und I_RNa an tVP angeschlossen. Das Terminal tVP wird nun in zwei Terminals geteilt[10], so dass je ein Strompfad an den Terminals beginnt. An ihnen kann daraufhin das Spannungspotential so modifiziert werden, dass die gewünschte Offsetspannung an den Ausgangs-Terminals entsteht[11].

Mit Beschreibung 5.11 ist nun ein Modell des Drucksensors entstanden, in dem die in Abb. 5.15 gezeigte Temperaturabhängigkeit des Innenwiderstands ergänzt werden kann. Um das testbasierte Entwurfsparadigma aufrecht zu erhalten, wird diese Erweiterung in

[10] Rfact. 48 *Refine Dataflow*
[11] Rfact. 45 *Refine Simplified Model*

```
architecture ams5310_R1 of vdc_psens is
-- typische Werte der Generics:
--   k0=275.0e-6 V/kPa, UV=5V, UOffset=-0.005V
  constant TkP : real := -2.2e-3; -- pro °C
  quantity f1, f2 : real;
  quantity P across  P_In;
  quantity Temp across  T_In;
  terminal tVP : electrical;
  quantity U_VP across I_VP through tVP;
  quantity Uout_V across  Uout_I through  U_OutP to U_OutN;
  quantity U_OutP_V across  U_OutP_I through  U_OutP;
begin
  U_VP == UV;
  f1 == k0 * p;
  f2 == 1.0 + TkP * (Temp-25.0);
  Uout_V == f1 * f2 * U_VP + UOffset;
  U_OutP_V == (Uout_V + U_VP) / 2.0;
end ams5310_R1;
```

Beschreibung 5.9: Refactoring des Drucksensormodells: Erweiterung um ein Terminal für die Versorgungsspannung.

zwei Schritten durchgeführt. Im ersten Schritt wird das Sensormodell um die gewünschte Funktionalität erweitert, ohne die Entity-Deklaration zu verändern. Das ermöglicht die Weiterverwendung der existierenden Testbenches. In einem zweiten Schritt wird dann die Testbench und die Komponenten-Deklaration auf die neuen Bedürfnisse angepasst.

Das in Beschreibung 5.12 gezeigte Modell beschreibt den Drucksensor mit der Temperaturabhängigkeit des Innenwiderstands. Diese kann in einer Testbench als Mittenspannung des Sensorsignals gemessen werden. Um dieses Modell zu erhalten wurden mehrere Refactorings durchgeführt: Das Modell nutzt die beiden zusätzlichen Quantities RTemp und delta zur Speicherung von Zwischenergebnissen[12]. Für die Berechnung des Temperatureinflusses auf den Innenwiderstand wird die Konstante TkR0 eingeführt und die Berechnungsgleichung für RTemp erweitert[13]. Die Quantity delta wird zur Speicherung von $f_1 \cdot f_2$ verwendet. Die Gleichungen zur Berechnung von f1 und f2 wurden in die Gleichung zur Berechnung von delta integriert[14], um im Anschluss daran die Quantity k1 als Zwischengröße zu extrahieren[15].

Anstelle der Temperaturmessung über die Mittenspannung des Sensorausgangssignals wäre es sinnvoll den Innenwiderstand des Sensors direkt zur Temperaturbestimmung zu nutzen. Dazu wird das Terminal tVP in die Komponenten-Deklaration aufgenommen[16]

[12] Rfact. 48 *Refine Dataflow*
[13] Rfact. 48 *Refine Dataflow*
[14] Rfact. 30 *Inline Simple Statement*
[15] Rfact. 1 *Extract Signal, Quantity or Terminal*
[16] Rfact. 21 *Add Port to Entity Declaration*

```
architecture ams5310_R2 of vdc_psens is
-- typische Werte der Generics:
--   k0=275.0e-6 V/kPa, UV=5V
  constant TkP : real := -2.2e-3; -- pro °C
  constant R0 : real := 3300.0;  -- Innenwiderstand in Ohm
  quantity f1, f2 : real;
  quantity P across  P_In;
  quantity Temp across  T_In;
  terminal tVP : electrical;
  quantity U_VP across I_VP through tVP;
  quantity U_RPa across I_RPa through tVP to U_OutP;
  quantity U_RPb across I_RPb through U_OutP;
  quantity U_RNa across I_RNa through tVP to U_OutN;
  quantity U_RNb across I_RNb through U_OutN;
begin
  assert (UOffset = 0.0)
    report "P-Sensor Offset is not modeled"
    severity error;
  U_VP == UV;
  f1 == k0 * p;
  f2 == 1.0 + TkP * (Temp-25.0);
  U_RPa / I_RPa ==  R0 * (1-f1*f2);
  U_RPb / I_RPb ==  R0 * (1+f1*f2);
  U_RNa / I_RNa ==  R0 * (1+f1*f2);
  U_RNb / I_RNb ==  R0 * (1-f1*f2);
end ams5310_R2;
```

Beschreibung 5.10: Refactoring des Drucksensormodells: Konkretisierung durch Modellierung der Ströme und Widerstände in der Wheatstone-Brücke.

und die Spannungsversorgung des Sensors in die Testbench verlagert. Die Komponente wird in AMS5310 umbenannt[17], um Inkonsistenzen mit der vorhandenen Komponente vdc_psens zu vermeiden. Die Temperaturkoeffizienten TkR0 und TkP, sowie der Innenwiderstand R0 werden in die Liste der Komponentenparameter aufgenommen[18], so dass das Modell genauer auf die untersuchten AMS53100-100 Sensoren angepasst werden kann. Der Parameter UV wird entfernt, da die Spannungsversorgung nun extern erfolgt.

Beschreibung 5.13 enthält das erzeugte Modell. Dieses kann auch in vdc_psens eingesetzt werden[19], um einen speziellen AMS5310-100 Sensor im Gesamtsystemmodell der Spezifikationsphase zu verwenden:

```
architecture ams5310_R5 of vdc_psens is
  terminal tVP : electrical;
  quantity U_VP across I_VP through tVP;
begin
  U_VP == 5.0;
  ams5310_1: entity work.ams5310(ams5310_R5)
    generic map (k0 => 281.0e-6, UOffset => -6.3e-3V,
```

[17] Rfact. 18 *Rename Process/Entity*
[18] Rfact. 23 *Parameterize Entity*
[19] Rfact. 42 *Insert Component*

```
architecture ams5310_R3 of vdc_psens is
-- typische Werte der Generics:
--   k0=275.0e-6 V/kPa, UV=5V, UOffset=-0.005V
  constant TkP : real := -2.2e-3; -- pro °C
  constant R0 : real := 3300.0;  -- Innenwiderstand in Ohm
  quantity f1, f2 : real;
  quantity P across  P_In;
  quantity Temp across  T_In;
  terminal tVP1, tVP2 : electrical;
  quantity U_VP1 across I_VP1 through tVP1;
  quantity U_VP2 across I_VP2 through tVP2;
  quantity U_RPa across I_RPa through tVP1 to U_OutP;
  quantity U_RPb across I_RPb through U_OutP;
  quantity U_RNa across I_RNa through tVP2 to U_OutN;
  quantity U_RNb across I_RNb through U_OutN;
begin
  f1 == k0 * p;
  f2 == 1.0 + TkP * (Temp-25.0);
  U_VP1 == UV + UOffset;
  U_VP2 == UV - UOffset;
  U_RPa / I_RPa ==  R0 * (1-f1*f2);
  U_RPb / I_RPb ==  R0 * (1+f1*f2);
  U_RNa / I_RNa ==  R0 * (1+f1*f2);
  U_RNb / I_RNb ==  R0 * (1-f1*f2);
end ams5310_R3;
```

Beschreibung 5.11: Refactoring des Drucksensormodells: Modellierung der Offsetspannung.

```
                R0 => 3410.0, TkR0 => 8.81, TkP => -2.18e-3)
     port map (P_In, T_In, tVP, U_OutP, U_OutN);
end architecture ams5310_R5;
```

Das Drucksensormodell aus Beschreibung 5.13 hat Parameter, deren Werte sich je nach Versorgungsspannung ändern. Die Versorgungsspannung wird in diesem Modell jedoch durch eine externe Beschaltung vorgegeben, so dass die Annahme einer konstanten 5V Spannungsversorgung nicht zwingend ist. Um ein wiederverwendbares, wartungsfreundliches Modell zu erhalten, müssen die beiden Parameter k0 und UOffset, so definiert werden, dass das Modell auch bei Änderungen der Spannungsversorgung sinnvolle Ergebnisse liefert.

Die Druckempfindlichkeit des Sensors wird in Abb. 5.16 dargestellt. Sie wurde für eine Versorgungspannung von 5V aufgezeichnet, so dass als Messbedingung für k0 und den Temperaturkoeffizienten TkP U(tVP)=5 V festgelegt wird. Da das Sensorausgangssignal mit Hilfe von Widerständen gewonnen wird, entsteht ein Modell des Sensors in dem k0 proportional zu U(tVP) wird[20].

Die Offsetspannung des Sensors wird in Abb. 5.17 dargestellt. Ein eindeutiger Tem-

[20] Rfact. 45 *Refine Simplified Model*

```
architecture ams5310_R4 of vdc_psens is
-- typische Werte der Generics:
--   k0=275.0e-6 V/kPa, UV=5V, UOffset=-0.005V
  constant TkR0 : real := 9.24;   -- Tk des Innenwiderstands (Ohm/°C)
  constant TkP : real := -2.2e-3; -- Tk der Druckempfindlichkeit
  constant R0 : real := 3300.0;   -- Innenwiderstand bei 25°C
  quantity P across  P_In;
  quantity Temp across  T_In;
  terminal tVP1, tVP2 : electrical;
  quantity U_VP1 across I_VP1 through tVP1;
  quantity U_VP2 across I_VP2 through tVP2;
  quantity U_RPa across I_RPa through tVP1 to U_OutP;
  quantity U_RPb across I_RPb through U_OutP;
  quantity U_RNa across I_RNa through tVP2 to U_OutN;
  quantity U_RNb across I_RNb through U_OutN;
  quantity RTemp : real;
  quantity k1, delta : real;
begin
  RTemp == R0 +   TkR0 * (Temp-25.0);
  k1    == k0 + k0*TkP * (Temp-25.0);
  delta == k1 * p;
  U_VP1 == UV + UOffset;
  U_VP2 == UV - UOffset;
  U_RPa / I_RPa ==  RTemp * (1-delta);
  U_RPb / I_RPb ==  RTemp * (1+delta);
  U_RNa / I_RNa ==  RTemp * (1+delta);
  U_RNb / I_RNb ==  RTemp * (1-delta);
end ams5310_R4;
```

Beschreibung 5.12: Refactoring des Drucksensormodells: Nachbildung der Temperaturabhängigkeit des Innenwiderstands.

peraturdrift der Offsetspannung ist nicht erkennbar. Die Abweichungen im Diagramm können durch Messungenauigkeiten verursacht sein. Insofern ist es sinnvoll die Offsetspannung lediglich proportional zur Versorgungsspannung zu definieren: Der Parameter UOffset wird als Offsetspannung bei einem Umgebungsdruck von 0 kPa und U(tVP)=5V festgelegt. Fällt das Potential an tVP, so reduziert sich auch die Offsetspannung des Sensors. [21].

Das Modell des Drucksensors aus Beschreibung 5.14 liefert nun auch für Versorgungsspannungen kleiner 5V ein sinnvolles Ergebnis. Es ist deshalb denkbar den Sensor an eine Konstantstromquelle anzuschließen, so dass sich der Temperaturkoeffizient des Innenwiderstands und der Temperaturkoeffizient der Druckabhängigkeit gegeneinander verrechnen. In diesem Anwendungsfall ist die Temperaturabhängigkeit des Sensorausgangssignals deutlich geringer (siehe Abb. 5.18).

[21] Rfact. 45 *Refine Simplified Model*

```
library IEEE;
use ieee.math_real.all;

library disciplines;
use disciplines.electromagnetic_system.all;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;

-- k0: Druck-Empfindlichkeit bei 25°C (in V/kPa)
-- R0: Innenwiderstand bei 25°C (in Ohm)
-- TkR0: Temperaturkoeffizient des Innenwiderstands (in Ohm/°C)
-- TkP: Temperaturkoeffizient der Druckempfindlichkeit (1/°C)
-- UOffset: Offsetspannung in V, bei UV=5V, p=0kPa, Temp=25°C

entity ams5310 is
  generic (
    k0      : real := 275.0e-6;  -- V/kPa
    R0      : real := 3300.0;    -- Ohm
    TkR0    : real := 9.24;      -- Ohm/°C
    TkP     : real := -2.2e-3;   -- pro °C
    UOffset : emf  := 0.0);      -- V
  port(terminal P_In : fluidic;
       terminal T_In : thermal;
       terminal tVP, U_OutP, U_OutN : electrical);
end ams5310;

architecture ams5310_R5 of ams5310 is
-- typische Werte der Generics:
--   k0=275.0e-6 V/kPa, R0=3300 Ohm, TkR0=9,24 Ohm/°C,
--   TkP=-2.2e-3/°C, UOffset=-0,005V
  quantity P across  P_In;
  quantity Temp across  T_In;
  terminal tVP1, tVP2 : electrical;
  quantity U_VP1 across I_VP1 through tVP to tVP1;
  quantity U_VP2 across I_VP2 through tVP to tVP2;
  quantity U_RPa across I_RPa through tVP1 to U_OutP;
  quantity U_RPb across I_RPb through U_OutP;
  quantity U_RNa across I_RNa through tVP2 to U_OutN;
  quantity U_RNb across I_RNb through U_OutN;
  quantity RTemp : real;
  quantity k1, delta : real;
begin
  RTemp == R0 +   TkR0 * (Temp-25.0);
  k1    == k0 + k0*TkP * (Temp-25.0);
  delta == k1 * p;
  U_VP1 == - UOffset;
  U_VP2 == UOffset;
  U_RPa / I_RPa ==  RTemp * (1-delta);
  U_RPb / I_RPb ==  RTemp * (1+delta);
  U_RNa / I_RNa ==  RTemp * (1+delta);
  U_RNb / I_RNb ==  RTemp * (1-delta);
end ams5310_R5;
```

Beschreibung 5.13: Refactoring des Drucksensormodells: Das Resultat ist eine erweiterte Entity-Deklaration.

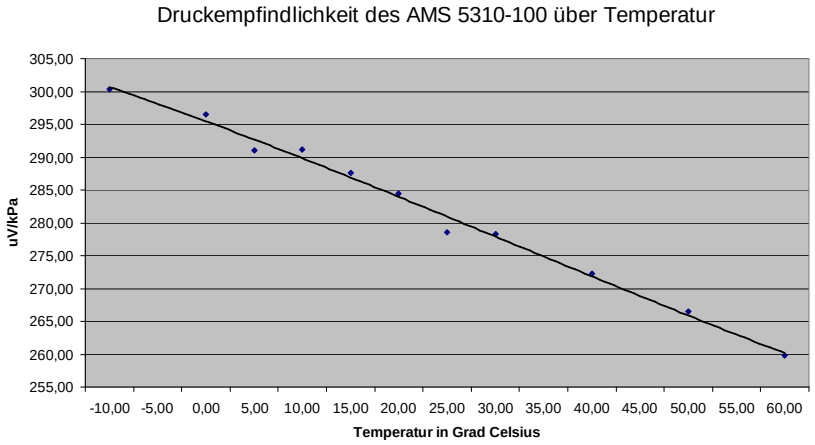


Abbildung 5.16: Druckempfindlichkeit eines AMS5310-100 Sensors bei 5V Versorgungsspannung

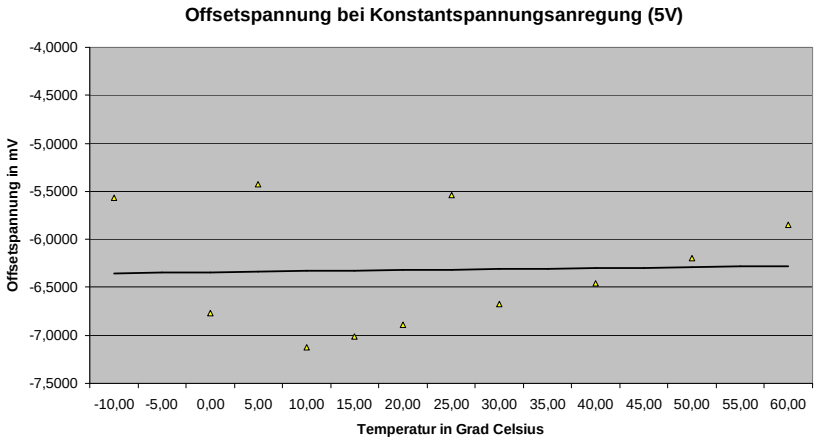


Abbildung 5.17: Offsetspannung eines AMS5310-100 bei Anregung mit einer 5V Spannungsquelle und 0 kPa Umgebungsdruck. Die lineare Trendlinie deutet auf eine temperaturunabhängige Offsetspannung hin

```
-- Geänderte Interpretation der Generics:
-- k0: Druck-Empfindlichkeit in V/kPa, bei U(tVP)=5V, Temp=25°C
-- UOffset: Offsetspannung in V, bei U(tVP)=5V, p=0kPa

architecture ams5310_R6 of ams5310 is
-- typische Werte der Generics:
--   k0=275.0e-6 V/kPa, R0=3300 Ohm, TkR0=9,24 Ohm/°C,
--   TkP=-2.2e-3/°C, UOffset=-5 mV/mA
  quantity P across  P_In;
  quantity Temp across  T_In;
  terminal tVP1, tVP2 : electrical;
  quantity U_VP across tVP;
  quantity U_VP1 across I_VP1 through tVP to tVP1;
  quantity U_VP2 across I_VP2 through tVP to tVP2;
  quantity U_RPa across I_RPa through tVP1 to U_OutP;
  quantity U_RPb across I_RPb through U_OutP;
  quantity U_RNa across I_RNa through tVP2 to U_OutN;
  quantity U_RNb across I_RNb through U_OutN;
  quantity RTemp : real;
  quantity k1, delta : real;
begin
  RTemp == R0 +   TkR0 * (Temp-25.0);
  k1    == k0 + k0*TkP * (Temp-25.0);
  delta == k1 * p * (U_VP)/5.0;
  U_VP1 == - U_VP/5.0 * UOffset;
  U_VP2 == U_VP/5.0 * UOffset;
  U_RPa / I_RPa ==  RTemp * (1-delta);
  U_RPb / I_RPb ==  RTemp * (1+delta);
  U_RNa / I_RNa ==  RTemp * (1+delta);
  U_RNb / I_RNb ==  RTemp * (1-delta);
end ams5310_R6;
```

Beschreibung 5.14: Refactoring des Drucksensormodells: Modellierung der Offsetspannung und der Druckempfindlichkeit in Abhängigkeit von der aktuellen Versorgungsspannung.

5.5.2 Delta-Sigma A/D Wandler

Der Δ-Σ Modulator des vDC A/D Wandlers wurde in einem $0,8\mu m$ CMOS Prozess implementiert, um die Tragfähigkeit der Entwurfsmethodik zu überprüfen. Der A/D Wandler wurde mit einem Meet-In-The-Middle Entwurfsprozess entwickelt, wobei ausgehend von einem Modell des idealen A/D Wandlers durch schrittweise Verfeinerungen ein Verhaltensmodell erarbeitet wurde, in dem einzelne Komponenten durch vorgefertigte Makrozellen ersetzt werden konnten. Diese Makrozellen wurden in Bottom-Up Entwurfsstil aus den im CMOS Prozess zur Verfügung stehenden Grundelementen erzeugt.

Die Bedeutung von Refactoring in der Verhaltensmodellierung wächst mit der Komplexität der beschriebenen Systeme und mit den zusätzlichen Randbedingungen, denen synthetisierbare Modelle genügen müssen. Obwohl für die in diesem Beispiel einge-

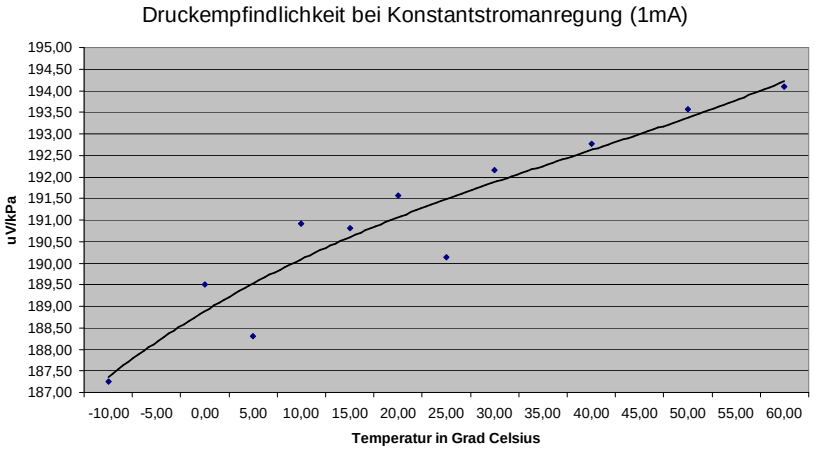


Abbildung 5.18: Druckempfindlichkeit eines AMS5310-100 Sensors bei 1mA Versorgungsstrom

setzen Makrozellen keine generischen Bibliothekszellen vorlagen, wird das Beispiel zeigen, dass die durch Refactoring unterstützte Entwurfsmethodik auch die Extraktion von synthetisierbaren Analogblöcken aus einem komplexen Gesamtsystem erleichtert.

Die untere Funktion in Beschreibung 5.7 aus der Gesamtsystemmodellierung beschreibt das Ausgangssignal eines A/D Wandler mit dem reduzierten Ausgangswertebereich. Diese Funktion kann in einer Architektur-Beschreibung direkt eingesetzt werden, um das Verhalten des Δ-Σ A/D Wandlers zu beschreiben. In Beschreibung 5.15 wird eine

```
architecture alpha_class of genericADC is
quantity U_in across in_P to in_N;
begin
  Ideal_Conversion: process (clock, reset) is
  begin
    if reset = '1' then
      DataOut <= (others => '0');
    elsif clock'event and clock = '1' then
      DataOut <= Convert_AnalogToDigital_DSM(MinVoltage,
                                  MaxVoltage, Res, U_in);
      DataOutValid <= '1';
    end if;
  end process Ideal_Conversion;
end architecture alpha_class;
```

Beschreibung 5.15: Einfache Architektur Beschreibung des Δ-Σ A/D Wandlers RO

```
architecture alpha_within_gamma of genericADC is
quantity U_in across in_P to in_N;

quantity U_slewed,U_delayed : emf;
-- Setting the max. time step:
limit U_slewed,U_delayed : emf with time2real(OutputDelay/50);
-- in Advance MS this has to be done globally in the main .tran card:
-- example: .tran logStep simTime startTime maxTimeStep

begin
-- first order 'gamma class fitting':
  U_slewed == U_in'slew(maxSlewRate);
  U_delayed == U_slewed'Delayed(time2real(OutputDelay));

  Ideal_Conversion: process (clock, reset)
  variable r : real;
  begin
    if reset = '1' then
      DataOut <= (others => '0');
    elsif clock'event and clock = '1' then
      DataOut <=Convert_AnalogToDigital_DSM
                (MinVoltage, MaxVoltage, Res, U_delayed);
    end if;
  end process Ideal_Conversion;
end architecture alpha_within_gamma;
```

Beschreibung 5.16: Einfaches Verhaltensmodell des Δ-Σ A/D Wandlers RO mit Durchlaufzeitverzögerung

solche Architektur-Beschreibung gezeigt.

Der Δ-Σ A/D Wandler besteht aus einem Δ-Σ Modulator, an dessen digitalen Ausgang ein *Decimation Filter* und ein FIR-Filter angeschlossen sind. Das Ausgangssignal des A/D Wandlers wird dementsprechend stets eine signifikante Verzögerung aufweisen. Diese Verzögerung kann in einem einfach approximierten Verhaltensmodell auch durch eine Verzögerung der Eingangssignale nachgebildet werden[22]. Da der Entwurf der digitalen Filter noch nicht abgeschlossen ist, wird für die Verzögerungszeit ein generischer Parameter eingeführt[23]. Das resultierende Verhaltensmodell ist in Beschreibung 5.16 zu sehen.

Dieses Verhaltensmodell kann in einer Testbench automatisch validiert werden, in der man den digitalisierten Wert mit dem Ergebnis der Funktion aus Beschreibung 5.6 vergleicht. Für hinreichend niederfrequente Signale muss gelten:

$$\frac{DataOut}{2^{BitRes}} \approx 0,19 + 0,62 \cdot \frac{u}{2^{BitRes}} \tag{5.62}$$

[22] Mit Hilfe von Rfact. 45 *Refine Simplified Model* wird die Eingangsverzögerung in Beschreibung 5.15 ergänzt.

[23] Rfact. 23 *Parameterize Entity*

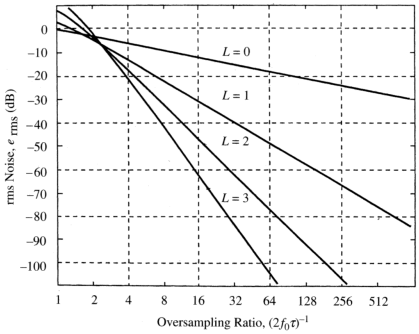


Abbildung 5.19: Quantisierungsrauschen in Abhängigkeit zum OSR und zur Ordnung des Δ-Σ Modulators

Mit Beschreibung 5.16 steht nun ein Referenzmodell für eine erste Testbench zur Entwicklung eines Δ-Σ A/D Wandler zur Verfügung (s. Kapitel 4.1).

Der Δ-Σ A/D Wandler hat aufgrund von *Noise Shaping* und überlagertem *Pattern-Noise* kein weißes Rauschspektrum. Das *Noise Shaping* des Δ-Σ Modulators verschiebt einen Großteil des Quantisierungsrauschens in den hochfrequenten Bereich, der im Digitalteil ausgefiltert wird. Dadurch ergibt sich, wie in Abb. 5.19 gezeigt, pro Verdopplung des Oversampling-Faktors ein deutlich verbesserter SNR (Signal-to-Noise-Ratio) [73, S. 15].

Für einen A/D Wandler mit etwa 75 dB SNR und einer Auflösung von 12-Bit bietet sich ein Δ-Σ Modulator 2. Ordnung an. Das Modulatorrauschen N_0 des idealen Δ-Σ Modulator 2. Ordnung ist durch die folgende Gleichung bestimmt:

$$N_0 = e_{rms}\frac{\pi^2}{\sqrt{5}}OSR^{-5/2} \qquad OSR >> 1 \tag{5.63}$$

$$e_{rms} = \frac{\max(\Delta U_{in})}{2\sqrt{3}} \tag{5.64}$$

$$\rightsquigarrow SNR_{N_0} = 20\log_{10}\frac{\max(\Delta U_{in})}{N_0} \tag{5.65}$$

Dementsprechend wird die Rauschsignalunterdrückung bei einer Verdopplung der Samplingrate um 15 dB, bzw. $\frac{5}{2}$ Bit besser.[24]

Das digitale Signal am Ausgang des Δ-Σ Modulators ist ein hochfrequentes transientes Signal mit niedriger Auflösung. Durch den digitalen *Decimation*- und Tiefpassfilter

[24] In Gl. (5.64) wurde eine binäre Diskretisierung angenommen. Für die Definition von e_{rms} siehe [73, S. 5, Gl. (1.2)].

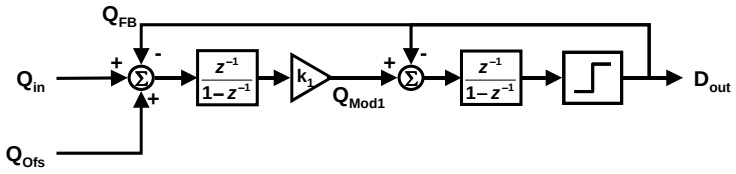


Abbildung 5.20: Architektur des Δ-Σ Modulators 2. Ordnung

wird das hochfrequente Rauschen in diesem Signal unterdrückt, so dass ein niederfrequentes, aber besser aufgelöstes Signal entsteht. Betrachtet man das Frequenzspektrum des Modulator-Ausgangssignals für ein konstantes Eingangssignal, so kann man sehen, dass der Bitstrom sogenannten *Pattern-Noise*[25] aufweist: Bitfolgen, die dem Wert des Eingangssignals entsprechen wiederholen sich. Dementsprechend ist das hochfrequente Rauschen nicht weiß, sondern hat um bestimmte Frequenzen eine deutlich größere Energie [73, S. 8, S. 185]. Die Energiedichte und die Lage des *Pattern-Noise* ist stark abhängig vom Eingangssignal und vom Oversampling-Faktor. Da dieses Rauschen bei Δ-Σ Modulatoren 1. Ordnung besonders ausgeprägt ist, verwendet man in der Regel Modulatoren höherer Ordnung.

Auch im Δ-Σ Modulator 2. Ordnung ist *Pattern-Noise* vorhanden. Das dadurch entstehende Rauschen hat bis zu 20 dB höhere Energiedichte als der durchschnittliche Quantisierungs-Rauschpegel. Um sicherzustellen, dass der *Pattern-Noise* das Rauschsignal nicht dominieren, sollte die Quantisierungs-Rauschunterdrückung um 15 dB stärker sein als der Gesamtrauschpegel [73, S.185].

Zur Validierung der digitalen Filter wird ein Modell des Modulators benötigt, das bei der Simulation wenig Rechenzeit benötigt und trotzdem die zeit- und frequenzabhängigen Eigenschaften des Modulators nachbildet. Da die Lösung von differential-algebraischen Gleichungssystemen wesentlich aufwendiger ist als die Simulation eines ereignisdiskreten Modells, stellt sich die Frage, ob aus den Ergebnissen der Machbarkeitsstudie ein brauchbares β-Klassen-Modell des Modulators extrahiert werden kann. Die Architektur des gewählten Δ-Σ Modulator ist in Abb. 5.20 gezeigt. Sie wurde aus dem in [13] vorgestellten Modulator abgeleitet.

In den meisten Literaturquellen werden zur Berechnung der Modulator-Übertragungsfunktion Spannungspegel verwendet. Da die Definition des Feedbacksignals und der Eingangssignale mit Hilfe von Ladungen jedoch eine flexiblere Spezifikation ermöglichen, wurde in Abb. 5.20 Ladungen als Zustandsgrößen verwendet.

[25] auch *Idle-Tones* genannt

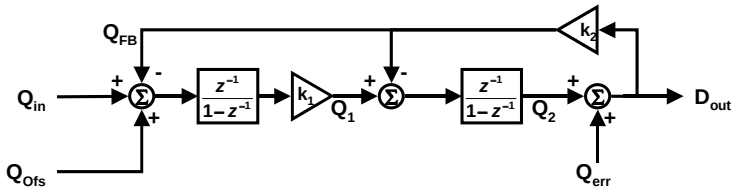


Abbildung 5.21: Signalflussmodell des Δ-Σ Modulators 2. Ordnung

Um die Übertragungsfunktion des Δ-Σ Modulators berechnen zu können, wird der Komparator aus Abb. 5.20 durch eine Summationsstelle mit additivem Fehler Q_{err} ersetzt. Abb. 5.21 zeigt das neue Signalflussmodell, mit dem die Übertragungsfunktionen für das Nutzsignal Q_{in} und das Rauschsignal Q_{err} berechnet werden:

$$Q_1 = k_1 \frac{z^{-1}}{1-z^{-1}} \left(Q_{\text{in}} + Q_{\text{Ofs}} - Q_{\text{FB}}\right) \tag{5.66}$$

$$Q_2 = \frac{z^{-1}}{1-z^{-1}} \left(Q_1 - Q_{\text{FB}}\right) \tag{5.67}$$

$$Q_{\text{FB}} = k_2 \left(Q_{\text{err}} + Q_2\right) \tag{5.68}$$

$$\rightsquigarrow \frac{Q_{\text{FB}}}{k_2} = Q_{\text{err}} + \frac{z^{-1}}{1-z^{-1}} \left(Q_1 - Q_{\text{FB}}\right) \tag{5.69}$$

$$\rightsquigarrow Q_{\text{FB}} \left(\frac{1}{k_2} + \frac{z^{-1}}{1-z^{-1}}\right) = Q_{\text{err}} + \frac{z^{-1}}{1-z^{-1}} Q_1 \tag{5.70}$$

$$\begin{aligned} &\rightsquigarrow \frac{1+(k_2-1)z^{-1}}{k_2(1-z^{-1})} Q_{\text{FB}} \\ &\quad = Q_{\text{err}} + k_1 \left(\frac{z^{-1}}{1-z^{-1}}\right)^2 \left(Q_{\text{in}} + Q_{\text{Ofs}} - Q_{\text{FB}}\right) \end{aligned} \tag{5.71}$$

$$\begin{aligned} &\rightsquigarrow \frac{1+(k_2-2)z^{-1}+(1+k_1-k_2)z^{-2}}{k_2(1-z^{-1})^2} Q_{\text{FB}} \\ &\quad = Q_{\text{err}} + k_1 \left(\frac{z^{-1}}{1-z^{-1}}\right)^2 \left(Q_{\text{in}} + Q_{\text{Ofs}}\right) \end{aligned} \tag{5.72}$$

$$\rightsquigarrow Q_{\text{FB}} = \frac{k_2(1-z^{-1})^2}{1+(k_2-2)z^{-1}+(1+k_1-k_2)z^{-2}} Q_{\text{err}} + \frac{k_1 k_2 z^{-2}}{1+(k_2-2)z^{-1}+(1+k_1-k_2)z^{-2}} \left(Q_{\text{in}} + Q_{\text{Ofs}}\right) \tag{5.73}$$

$$D_{out} = Q_{\text{err}} + Q_2 = \frac{Q_{\text{FB}}}{k_2} \tag{5.74}$$

$$\rightsquigarrow H_{\text{in}}(z) = \frac{k_1 z^{-2}}{1+(k_2-2)z^{-1}+(1+k_1-k_2)z^{-2}} \tag{5.75}$$

$$\rightsquigarrow H_{\text{err}}(z) = \frac{(1-z^{-1})^2}{1+(k_2-2)z^{-1}+(1+k_1-k_2)z^{-2}} \tag{5.76}$$

Die Verstärkungsparameter k_1 und k_2 ergeben sich aufgrund des begrenzten Eingangsspannungsbereichs, der durch die endliche Versorgungspannung des Δ-Σ Modulators und technologische Randparameter vorgegeben ist.

Wenn man den Parameter k_1 groß wählt, wird das Quantisierungsrauschen besser unterdrückt. Andererseits muss $k_1 < 0,6$ sein, um den *Integrator Overload Noise* gering zu halten. Ein guter Kompromiss ist $k_1 = 0,5$, für den der benötigte Ausgangsspannungsbereich der Operationsverstärker über Wahrscheinlichkeitsrechnungen bestimmt werden kann. Das Resultat ist die in Gl. (5.79) verwendete Definition, die aus [13] hergeleitet ist. Dazu wurden in den dort angegebenen Spannungsgleichungen die Kapazitätsverhältnisse eingesetzt.

Man berechnet die Ladung $\max(Q_{int})$ in den Integrationskondensatoren aus

$$k_1 = \frac{1}{2} \tag{5.77}$$

$$\max(Q_{int}) = C_{int}\left(\max(U_{OP,out}) - \min(U_{OP,out})\right) \tag{5.78}$$

$$\rightsquigarrow \Delta := \frac{\max(Q_{int})}{3,4} \tag{5.79}$$

Der Verstärkungsfaktor k_2 ist durch das Verhältnis der Feedbacksignale zum maximalen Eingangssignal festgelegt. Wie in Kapitel 5.4.3 vorgestellt, wird als Feedbackladung $\pm\Delta/2$ verwendet. Für die abgetasteten Eingangssignale gilt:

$$-0,31\Delta \leq Q_{in} \leq 0,31\Delta \tag{5.80}$$

$$\rightsquigarrow k_2 = \frac{5}{3,1} \tag{5.81}$$

Die Verzerrung des Frequenzspektrums des Nutzsignals H_{in} durch den Modulator wird in Abb. 5.22 dargestellt.

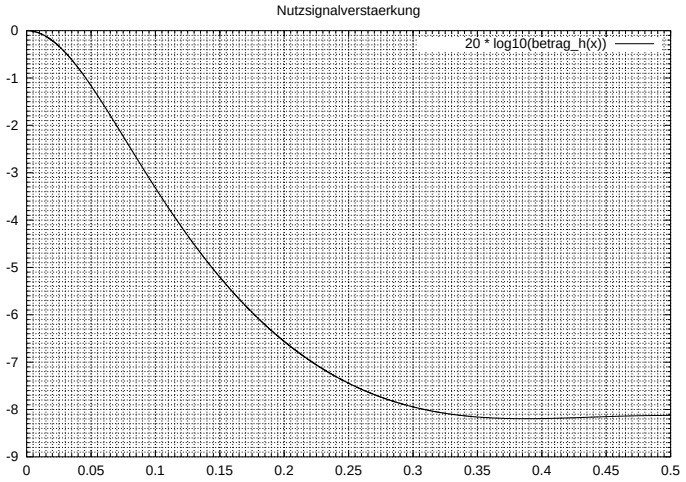


Abbildung 5.22: Signalverzerrung des Nutzsignals (in dB) durch den Modulator im Frequenzbereich 0 Hz bis 0,5 · Samplefrequenz

Abb. 5.23 zeigt den interessanten Ausschnitt des Spektrums: Für Frequenzanteile bis $\frac{1}{256}$ der Samplingfrequenz ist die Amplitudendämpfung unter 0,1%.

Das Rauschsignal H_{err} wird dagegen durch den Δ-Σ Modulator stark verändert. Die charakteristische Verschiebung des Rauschanteils zu den hohen Frequenzen (siehe Abb. 5.24) veranschaulicht das *Noise Shaping*.

Beschreibung 5.17 zeigt eine direkte Umsetzung des Flussdiagramms in ein VHDL Modell, welches das *Noise Shaping* des Modulators inklusive des *Pattern-Noise* nachbildet[26]. Der gezeigte Δ-Σ Modulator erwartet als Eingangsgröße ein Ladung Qin, so dass die Ausgangsspannung des Drucksensors nicht direkt an den Modulator angeschlossen werden kann. Ebenso fehlt ein Modell für den digitalen Tiefpass, der das Ausgangssignal des Δ-Σ Modulators verarbeitet. Ein solcher Tiefpass soll nun ergänzt werden, bevor der Δ-Σ Modulator in das Gesamtsystem eingebettet wird.

Das Modulatorausgangssignal wird in diesem konzeptionellen Tiefpass in drei Stufen gefiltert, wobei nach jedem Filter die Abtastrate reduziert wird. Als Filter kommen jeweils FIR-Filter zum Einsatz, deren Koeffizienten mit der Fenstermethode nach Kaiser [75] berechnet werden. Das gewünschte Downsampling von 256 wird erreicht, indem die Abtastrate nach dem ersten Filter um 16, nach dem zweiten Filter um 8 und nach dem dritten Filter um 2 reduziert wird. In den ersten beiden Stufen kann der Filterent-

[26] Rfact. 48 *Refine Dataflow*

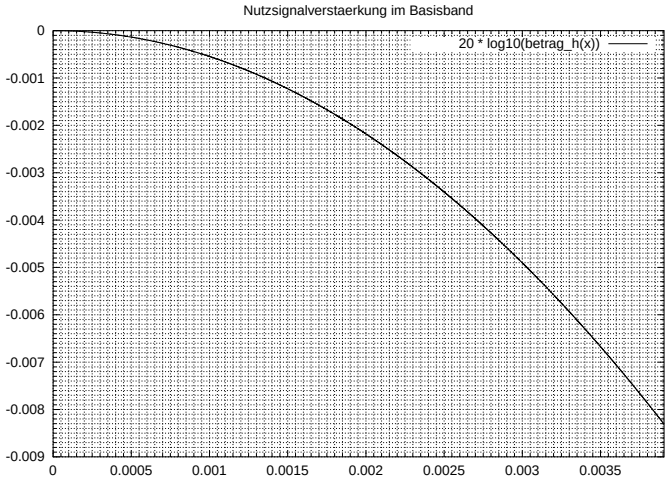


Abbildung 5.23: Signalverzerrung des Nutzsignals (in dB) durch den Modulator im Frequenzbereich 0 Hz bis $\frac{1}{256} \cdot$ Samplefrequenz

wurf vereinfacht werden, indem man *Aliasing* in den Frequenzbereichen zulässt, die in den nachfolgenden Stufen zum Sperrbereich gehören.

Ein VHDL-AMS Modell der Kaiser-Filter für die Gesamtsystemsimulation wird vollautomatisch generiert (s. Beschreibung 5.18). Dazu werden lediglich der Sperr- und der Durchlassbereich, sowie die gewünschte Rauschsignalunterdrückung (s. Tab. 5.8) in einem GnuPlot-Script angegeben.

Filterkenndaten	1. Filter	2. Filter	3. Filter
Eingangssamplerate	f_s	$\frac{1}{16} f_s$	$\frac{1}{128} f_s$
Ausgangssamplerate	$\frac{1}{16} f_s$	$\frac{1}{128} f_s$	$\frac{1}{256} f_s$
Grenzfrequenz des Durchlassbereichs	$\frac{1}{256} f_s$	$\frac{1}{256} f_s$	$\frac{0.75}{256} f_s$
Grenzfrequenz des Sperrbereichs	$\frac{31}{256} f_s$	$\frac{3}{256} f_s$	$\frac{1}{256} f_s$
Rauschsignalunterdrückung	80 dB	80 dB	80 dB
Anzahl der Filterkoeffizienten	88	82	82

Tabelle 5.8: Kenndaten der Kaiser-Filter

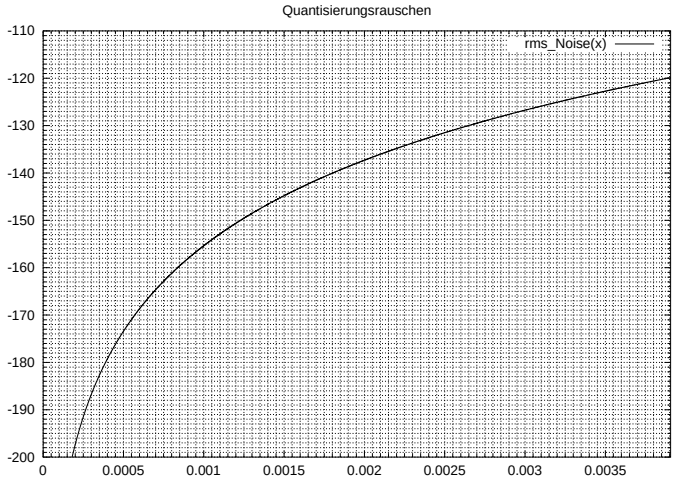


Abbildung 5.24: Theoretisches Minimum des Quantisierungsrauschen (in dB), das bei Verwendung eines idealen Tiefpasses im Ausgangssignal des A/D Wandlers verbleibt. Die x-Koordinate ist die Grenzfrequenz des Tiefpasses aus dem Bereich von 0 bis $\frac{1}{256}$ der Samplefrequenz.

```
library IEEE;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.math_real.all;

entity delta_sigma_modulator_1 is
  generic (
    Q_Feedback : real;
    Q_Offset   : real;
    k1      : real := 0.5);
    port (signal Clock, Reset : in std_logic;
          signal Qin : in real;
          signal DsmRes : out std_logic);
end delta_sigma_modulator_1;

architecture second_order_sample_and_hold of delta_sigma_modulator_1 is
begin
  DSM: process (reset, clock)
    variable  Q1, Q2, Q_FB : real := 0.0;
  begin
    if Reset = '1' then
      Q1 := 0.0;
      Q2 := 0.0;
      Q_FB := Q_Feedback;
      DsmRes <= '0';
    elsif clock'event and clock = '1' then
      Q2 := Q2 + k1* Q1 + Q_FB;
      Q1 := Q1 + Qin - Q_Offset + Q_FB;
      if (Q2 > 0.0) then
        DsmRes <= '1';
        Q_FB := -Q_Feedback;
      else
        DsmRes <= '0';
        Q_FB := Q_Feedback;
      end if;
    end if;
  end process;
end second_order_sample_and_hold;
```

Beschreibung 5.17: Modell des Δ-Σ Modulators in VHDL

```
library IEEE;
  use ieee.math_real.all;

package kaiser_filter_coeff is
  constant kaiser_filter_order : integer := 81;
  constant kaiser_coef : real_vector(0 to 81) := (
                          2.09043252204112e-05,
                          4.0870757316356e-05,
                          6.63873273264839e-05,
                          9.51350785222413e-05,
                          [...]
                          2.09043252204112e-05);
end kaiser_filter_coeff;

package body kaiser_filter_coeff is
end kaiser_filter_coeff;
-------------------------------------------------------------------------------
library IEEE;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use work.kaiser_filter_coeff.all;

entity kaiser_filter is
  generic(BitsPerSample : integer := 12);
  port(
    signal Clock, Reset : in std_logic;
    signal Din    : in  std_logic_vector;
    signal Dout   : out std_logic_vector(BitsPerSample-1 downto 0));
end entity kaiser_filter;

architecture functional of kaiser_filter is
begin  -- architecture functional
  Main: process (Clock, reset)
    variable z_reg : real_vector(kaiser_filter_order downto 0);
    variable r1,r_in,r_out : real;
    variable i : integer;
  begin  -- process Main
    if reset = '1' then                  -- asynchronous reset (active high)
      Dout <= (others => '0');
      for i in z_reg'range loop
        z_reg(i) := 0.0;
      end loop;
    elsif Clock'event and Clock = '1' then  -- rising clock edge
      r_in := real(conv_integer(unsigned(Din))) / (2.0**(Din'length-1));
      i := 1;
      r1 :=  kaiser_coef(0) * r_in;
      while (i <= kaiser_filter_order) loop
        r1 := r1 + kaiser_coef(i) * z_reg(i);
        z_reg(i) := z_reg(i-1);
        i := i+1;
      end loop;
      r_out := r1*(2.0**(BitsPerSample));
      Dout <= std_logic_vector(conv_unsigned(integer(r_out),BitsPerSample));
    end if;
  end process Main;
end architecture functional;
```

Beschreibung 5.18: Automatisch erzeugter Kaiser-Filter

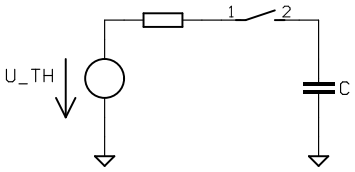


Abbildung 5.25: Modell des thermischen Rauschens einer SC-Komponente aus *Transmission Gate* und Kondensator

5.5.2.1 Thermisches Rauschen

Um den Platzbedarf des Modulators zu minimieren, müssen die im Modulator transportieren Ladungen minimiert werden. Damit kann die Kapazität und der Platzbedarf der Kondensatoren reduziert werden, zusätzlich sinken die Anforderungen an die Ausgangsströme der Operationsverstärker. Die kleinste Größe der Kondensatoren wird durch das thermische Rauschen beim Umladen (Gl. (5.82)) bestimmt [73, S. 353ff]. Dazu sind die SC-Kapazitäten und die Widerstände der Pass-Transistoren im Durchlassfall zu betrachten (s. Abb. 5.25). Der Widerstand in diesem RC-Glied muss so klein sein, dass $\frac{1}{RC} >> f_s$ gilt. Dann kann angenommen werden, dass das RC-Rauschen weiß ist und die Leistung $\frac{kT}{C}$ hat. Bei zwei Abtastungen pro Taktzyklus verdoppelt sich die Rauschleistung und nach der Abtastung mit f_s ist das Rauschen im Frequenzbereich $[0..f_s/2]$ verteilt, somit ergibt sich die Rauschdichte S(f) pro SC-Komponente:

$$S(f) = \frac{4kT}{f_s C} \qquad k = 1,38065 \cdot 10^{-23} J/K \tag{5.82}$$

Die Gesamtrauschdichte durch Umladen von SC-Kapazitäten am Modulatoreingang berechnet sich aus

$$S(f)_{ges} = \frac{4kT}{f_s} \sum_i \frac{1}{C_i} \tag{5.83}$$

Das Gesamtrauschen wird durch Integration der Gesamtrauschdichte über den Frequenzbereich des dezimierten Ausgangssignals berechnet. Da die Gesamtrauschdichte

konstant ist, ergibt sich:

$$e_{ges}^2 = \frac{4kT}{OSR} \sum_i \frac{1}{C_i} \tag{5.84}$$

$$\rightsquigarrow SNR_{ges,th} = 20 \cdot log_{10} \left(\frac{U_{in,\max}}{e_{ges}} \right) \tag{5.85}$$

$$= 20 \cdot \log_{10} \left(U_{in,\max} \right) - 10 \cdot log_{10} \left(\frac{4kT}{OSR} \sum_i \frac{1}{C_i} \right) \tag{5.86}$$

Für eine Implementierung ist es sinnvoll das thermische Rauschen auf ein Viertel des gewünschten Gesamtrauschpegels zu beschränken, wobei lediglich der Eingangsknoten des Δ-Σ Modulators betrachtet werden muss. Bei einem differentiellen Design gewinnt man 6 dB an Signalverstärkung bei 3 dB zusätzlichem Rauschen, so dass der SNR letztendlich um 3 dB besser wird.

Um mit 128-fach Oversampling einen *full-scale SNR* von 75 dB zu erreichen, wird bei einem thermischen Rauschanteil von $\frac{1}{8}$ (=12 dB) und einem differentiellen Design (=3 dB) ein $SNR_{ges,th}$ von 81 dB pro Eingangsknoten benötigt.

Das Spannungssignal des Drucksensors liegt zwischen -38,6 mV und 446 mV. Bei einer maximalen *Die*-Temperatur von 80°C (d. h. 353 K) gilt:

$$\sum_i \frac{1}{C_i} \leq \frac{OSR(U_{in,\max})^2}{4kT} \cdot 10^{-\frac{SNR_{ges,th}}{10}} \approx \frac{1}{96,6fF} \tag{5.87}$$

Aufgrund der benötigten Offsetkorrektur ergibt sich die folgende Summationsformel:

$$Q_{Int}^{i+1} = Q_{Int}^{i} + (Q_{in} - Q_{Ofs}) - Q_{FB} \tag{5.88}$$

Bei einem maximalen Integrator-Ausgangssignal von $3V$, ergibt sich aus Gl. (5.79):

$$Q_{FB} = \pm \frac{3V}{2 \cdot 3,4} C_{int} = \pm \frac{15V}{34} C_{int} \tag{5.89}$$

$$\max(Q_{in} - Q_{Ofs}) = 0,31 \frac{3V}{3,4} C_{int} = \frac{9,3V}{34} C_{int} \tag{5.90}$$

$$Q_{Ofs} = \frac{\max(Q_{in}) + \min(Q_{in})}{2} \tag{5.91}$$

$$\frac{\max(Q_{in})}{\min(Q_{in})} = \frac{446mV}{-38,6mV} \tag{5.92}$$

$$\rightsquigarrow \max(Q_{in}) = 503,48mVC_{int} \tag{5.93}$$

$$\rightsquigarrow Q_{Ofs} = 229,95mVC_{int} \tag{5.94}$$

Mit $\pm 1V$ Konstantspannungsquellen für die Bildung der Offset- und Feedbackladungen, können die benötigten Sample-Kondensatoren berechnet werden:

$$C_{Ofs} = 0,230 C_{Int} \tag{5.95}$$
$$C_{FB} = 0,441 C_{Int} \tag{5.96}$$
$$C_{in} = 1,129 C_{Int} \tag{5.97}$$
$$\sum_i \frac{1}{C_i} = \frac{1}{C_{Int}}(0,8858 + 4,3488 + 2,2667) = \frac{7,50}{C_{Int}} \tag{5.98}$$

Der Rückkopplungszweig über C_{Int} wird nicht geschaltet, deshalb muss dieser Kondensator nicht berücksichtigt werden.

$$\rightsquigarrow C_{Int} \geq 724,8\,fF \tag{5.99}$$
$$\rightsquigarrow C_{Int} := 753\,fF \tag{5.100}$$
$$C_{in} = 850\,fF \tag{5.101}$$
$$C_{Ofs} = 173\,fF \tag{5.102}$$
$$C_{FB} = 332\,fF \tag{5.103}$$

Die dazugehörige Spannungsgleichung lautet:

$$U_{Int}^{i+1} = U_{Int}^{i} + (1,129 U_{in} - 0,23 U_{Ofs}) - 0,441 U_{FB} \tag{5.104}$$

Mit diesen Daten wird das VHDL-AMS Modell des Modulators erweitert (s. Beschreibung 5.19). Durch dieses konkretisierende Refactoring kann beispielsweise ein *Overload*-Fehler durch Überwachung der Variablen Sigma_1 und Sigma_2 detektiert werden und durch Variation der Modellparameter kann das Modell für unterschiedliche *Worst-Case* Simulationen verwendet werden.

5.5.2.2 Übergang zu *Component Level* Modellen

Die nun vorhandenen Modelle des Δ-Σ Modulators basieren auf Verhaltensbeschreibungen. Indem man einzelne Funktionen dieser Modelle durch äquivalente Strukturbeschreibungen aus analogen elektrischen Grundelementen ersetzt, wird ein fließender Übergang vom Gesamtsystemmodell zum *Component Level* Modell möglich.

Die dafür benötigten Strukturmodelle werden im Bottom-Up Entwurfsstil entwickelt. Für die Simulation dieser Modell wird ein Simulator benötigt, der konservative Modelle der γ-Klasse ausführen kann. Die Modelle bestehen aus Grundelementen wie Widerständen, Transistoren und Kapazitäten, die geeignet verbunden werden.

Ob für diese Grundelemente Verhaltensmodelle verwendet werden, für die eine Notation in VHDL-AMS vorliegt, oder statt dessen Verhaltensmodelle aus einer Bibliothek

```
library IEEE;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
library disciplines;
  use disciplines.electromagnetic_system.all;

entity delta_sigma_modulator is
  generic (
    VRefP      : emf;   -- 3.5V
    VRefN      : emf;   -- 1.5V
    C_FB       : real := 0.441;     -- C_FB / C_Integrator
    C_Sample   : real := 1.129;     -- C_sample / C_Integrator
    C_Offset   : real := 0.23;      -- C_Offset / C_Integrator
    C_Mod1     : real := 0.5);      -- C_Mod1 / C_Integrator
    port (terminal In_P : Electrical;
          terminal In_M : Electrical;
          signal Clock, Reset : in std_logic;
          signal DsmRes : out std_logic);
end delta_sigma_modulator;

architecture second_order_sample_and_hold of delta_sigma_modulator is
  quantity u_in across In_P to In_M;
begin
  DSM: process (reset, clock)
    constant Q_Feedback : charge := (VRefP-VRefN) * C_FB;
    constant Q_Offset   : charge := (VRefP-VRefN) * C_Offset;
    variable delta1, delta2, sigma1, sigma2, feedback : real := 0.0;
    variable res : std_logic;
  begin
    if Reset = '1' then
      sigma1 := 0.0;
      sigma2 := 0.0;
      feedback := Q_Feedback;
      DsmRes <= '0';
    elsif clock'event and clock = '1' then
      delta1 := u_in * (2.0 * C_sample) - Q_Offset + Feedback;
      delta2 := sigma1 * C_Mod1 + Feedback;
      sigma1 := sigma1 + delta1;
      sigma2 := sigma2 + delta2;
      if (sigma2 > 0.0) then
        res := '1';
        feedback := -Q_Feedback;
      else
        res := '0';
        feedback := Q_Feedback;
      end if;
      DsmRes <= res;
    end if;
  end process;
end second_order_sample_and_hold;
```

Beschreibung 5.19: Konkretisiertes Modell des Δ-Σ Modulators

des Simulatorherstellers verwendet werden, ist letztendlich nebensächlich. Eine Möglichkeit zur Einbindung von externen Bibliotheken, wie sie Advance MS von Mentor Graphics bietet, erleichtert jedoch den VHDL-AMS basierten Validierungsprozess: Für das Layout des Δ-Σ Modulators wird eine SPICE Netzliste der analogen Schaltungen benötigt. Das Simulationssystem erlaubt nun die Validierung dieser Netzlisten gemein-

```
component Eldo_ro5 is
  port (
    terminal VDDA, AGND : electrical;
    terminal Clk : electrical;
    terminal InP, InN : electrical;
    terminal FB, nFB : electrical);
end component Eldo_ro5;
attribute eldo_device of Eldo_ro5: component is Eldo_subckt;
attribute eldo_subckt_name of Eldo_ro5: component is "ro5__1";
attribute eldo_file_name   of Eldo_ro5: component is "ckt/ro5.cir";
```

Beschreibung 5.20: VHDL-AMS Interface zu einer SPICE-Netzliste

sam mit VHDL-AMS Modellen, indem die Netzlisten als *Component*[27] in ein VHDL-AMS Modell integriert werden.

Durch die Komponentendeklaration aus Beschreibung 5.20 kann der Komparator aus Abb. 5.26 in VHDL-AMS Modellen verwendet werden.

Diese Komponente wird in das Δ-Σ Modulator Modell aus Beschreibung 5.19 eingesetzt (siehe Beschreibung 5.21). Nach der Validierung des Komparators im Gesamtsystem werden weitere *Component Level* Modelle eingesetzt, deren Funktion im Gesamtsystem ebenfalls überprüft wird. Jedes *Component Level* Modell kann für sich im Gesamtsystem validiert und optimiert werden. Zusätzlich können beliebige Kombinationen aus mehreren *Component Level* Modellen gemeinsam simuliert werden. Es entsteht

[27] Rfact. 44 *Replace Simplified Model*

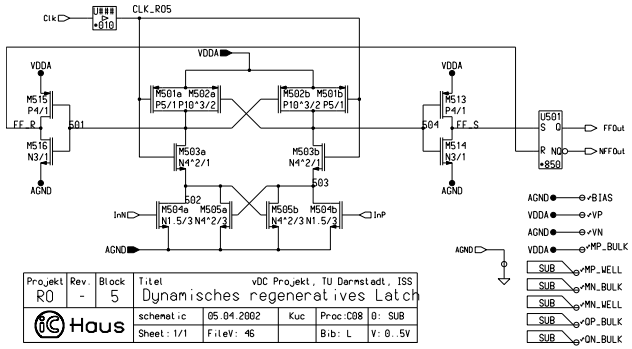


Abbildung 5.26: *Component Level* Modell eines Komparators

ebenso ein Modell des Δ-Σ Modulators, das vollständig aus Strukturbeschreibungen besteht. Das Gesamt-Schaltbild dieses Δ-Σ Modulator Modells ist in Abb. 5.27 skizziert. In Abb. 5.28, Abb. 5.30 und Abb. 5.29 wird der Aufbau der Integrationsblöcke des Modulators dargestellt.

Die Simulationsbeschreibung des Δ-Σ Modulators mit allen *Component Level* Modellen beinhaltet 787 MOS-Feldeffekttransistoren, 425 Dioden, 92 Widerständen, 65 Kapazitäten und 46 bipolare Transistoren. Eine Simulation dieser Beschreibung für eine Millisekunde Simulationszeit benötigt auf einem *Athon XP 1800+* 8,5 Stunden CPU-Zeit . Eine derartige Simulation des Modells aus Beschreibung 5.19 ist auf diesem Rechner nach 1,5 Sekunden beendet. Die Simulation der *Component Level* Modelle ist mehr als 20 000 mal aufwendiger.

Solche Gesamtsystem-Simulationen sind während der Optimierung der einzelnen Komponenten zu langwierig. Durch die Möglichkeit die meisten *Component Level* Modelle durch deren Gesamtsystemmodell zu ersetzen, ist mit der vorgestellten Methodik eine signifikante beschleunigte Simulation möglich, die eine gezielte simulationsbasierte Optimierung ermöglicht.

Die Entwicklung des Δ-Σ Modulators wurde nach 4 Mannmonaten mit der Einschleusung des Layouts abgeschlossen. In Abb. 5.31 wird das Layout des entstandenen Δ-Σ Modulator gezeigt. Das Layout ist $2,1\,mm^2$ gross; der dazugehörige IC wurde im Messlabor exemplarisch getestet. Der Signalrauschabstand für den in Abb. 5.32 gezeigten Stimulus beträgt 60,5 dB. Der Stimulus ist eine 1 kHz Sinusschwingung, das dazugehörige digitale Ausgangssignal des Δ-Σ Modulators wurde für 65 ms aufgezeichnet und anschließend ausgewertet. Die Fouriertransformation des Ausgangssignals des Δ-Σ Modulators ist in Abb. 5.33 zu sehen. Zusätzlich wurde das mit einem digitalen Tiefpassfilter erzeugte Signal bewertet. Die untere Grafik in Abb. 5.32 zeigt den absoluten Fehler zwischen diesem Signal und dem Stimulus, die untere Grafik in Abb. 5.33 zeigt die Fouriertransformation des Signals.

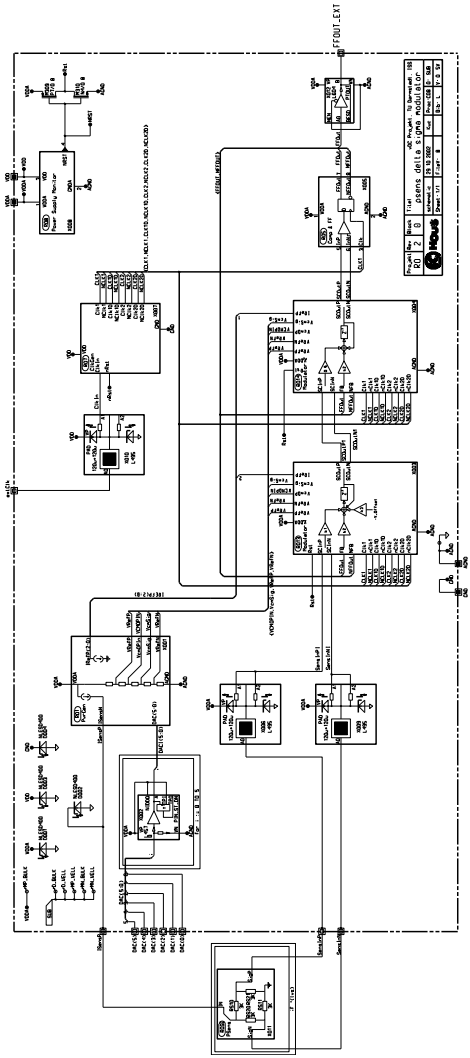


Abbildung 5.27: Gesamtschaltbild des Δ-Σ Modulator (hierarchisches *Component Level Modell*)

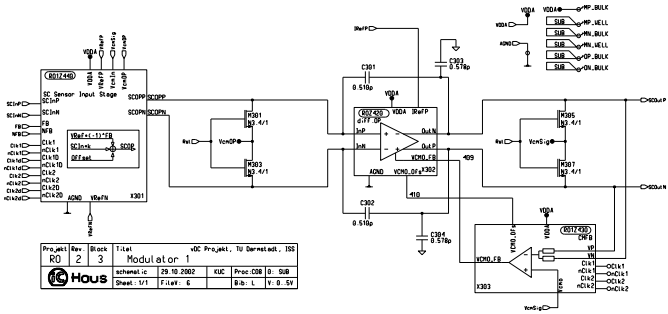


Abbildung 5.28: Schaltbild einer Integrationsstufe

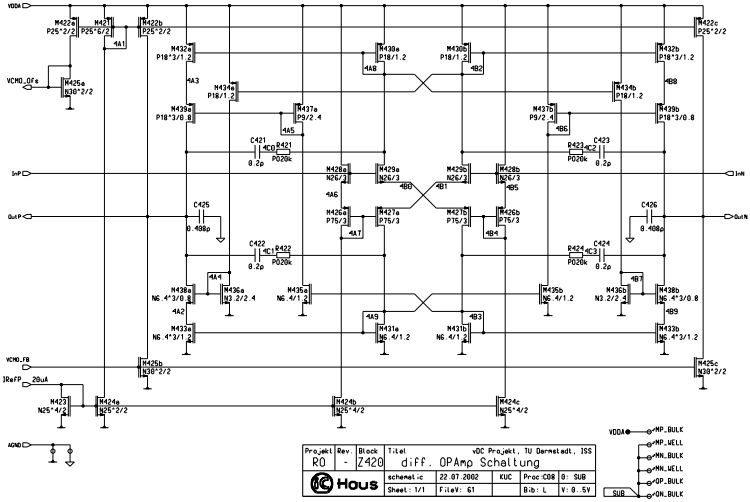


Abbildung 5.29: *Component Level* Modell des Operationsverstärkers

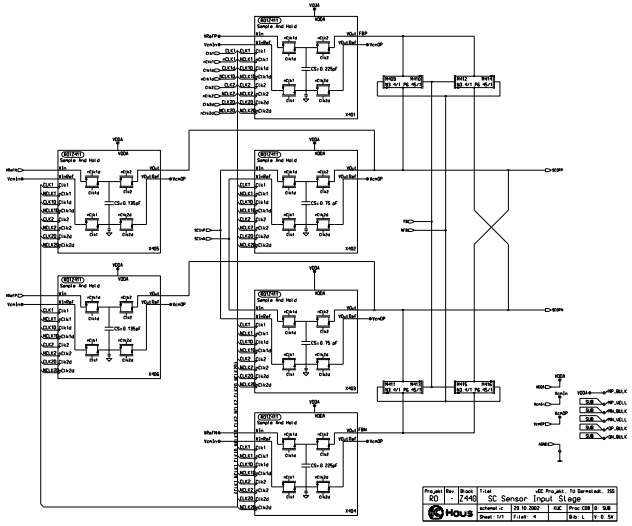


Abbildung 5.30: Aufbau der *Sample and Hold* Stufe

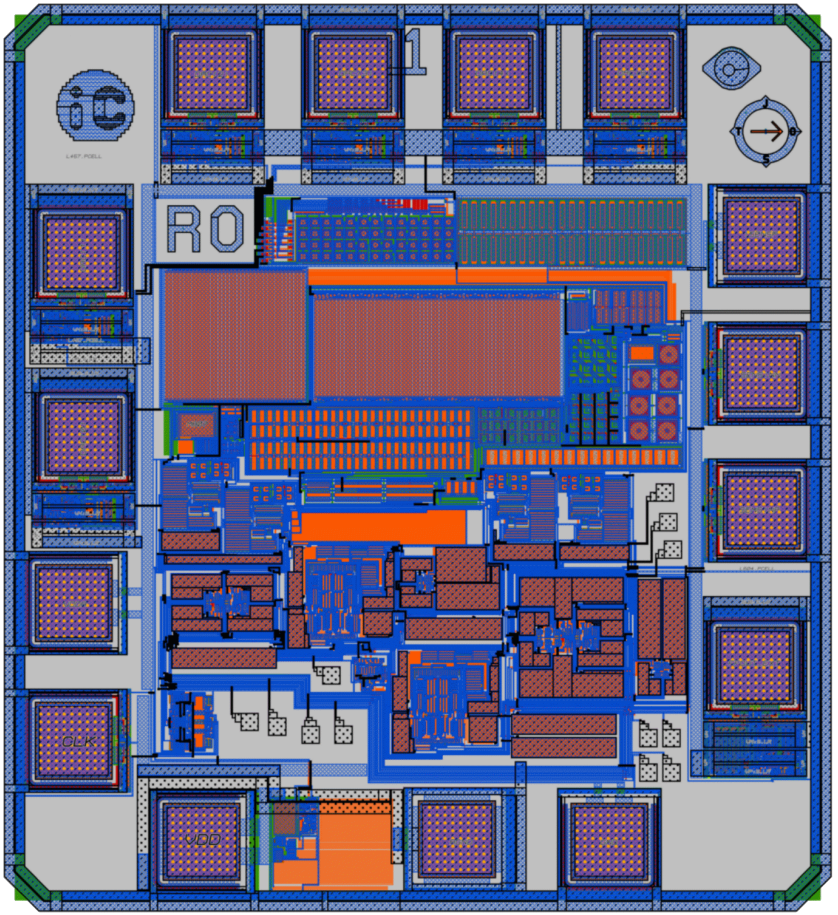


Abbildung 5.31: Layout des Δ-Σ A/D Wandlers

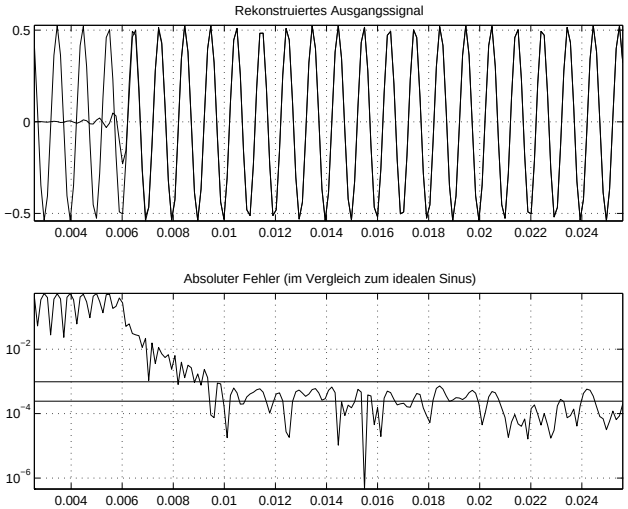


Abbildung 5.32: Auswertung einer Messreihe des realisierten Δ-Σ Modulators

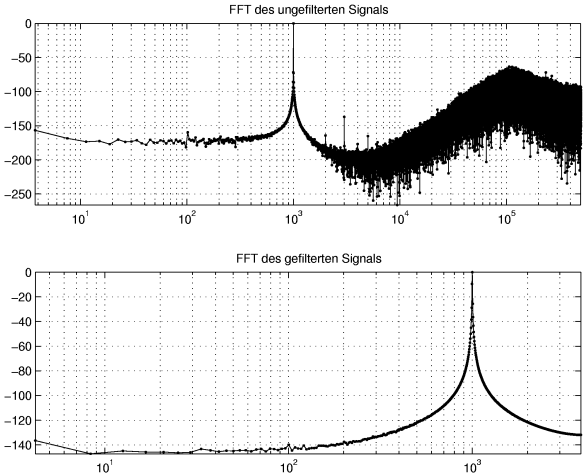


Abbildung 5.33: Fouriertransformation des Ausgangssignalspektrums bei einer 1 kHz Anregung

```
library IEEE;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
library disciplines;
  use disciplines.electromagnetic_system.all;
library MGC_AMS;
  use MGC_AMS.Conversion.all;
library EldoLib;
  use EldoLib.globalPwr.all;
  use work.Eldo_Comp_Package.all;
architecture second_order_sample_and_hold_with_ro5 of delta_sigma_modulator is
  terminal TClk, tSCoutP, tSCoutN, tFB, tNFB : electrical;
  quantity U_SCOut across tSCoutP to tSCoutN;
  quantity U_SCOutP across I_SCOutP through tSCoutP;
  quantity U_SCOutN across I_SCOutN through tSCoutN;
  quantity qClkTmp : real;
  quantity vClk across iClk through TClk;
  quantity U_FB across tFB;
  quantity u_in across In_P to In_M;
  signal res : std_logic;
  signal sigma2 : real;
begin
  DSM: process (reset, clock)
    constant Q_Feedback : charge := (VRefP-VRefN) * C_FB;
    constant Q_Offset   : charge := (VRefP-VRefN) * C_Offset;
    variable delta1, delta2, sigma1, feedback : real := 0.0;
  begin
    if Reset = '1' then
      sigma1 := 0.0;
      sigma2 <= 0.0;
    elsif clock'event and clock = '1' then
      if (Res = '1') then feedback := -Q_Feedback;
      else feedback := Q_Feedback;
      end if;
      delta1 := u_in * (2.0 * C_sample) - Q_Offset + Feedback;
      delta2 := sigma1 * C_Mod1 + Feedback;
      sigma1 := sigma1 + delta1;
      sigma2 <= sigma2 + delta2;
    end if;
  end process;
  CompFF : Eldo_ro5
    port map (
      VDDA => eldolib.globalPwr.tVDDA,
      AGND => eldolib.globalPwr.tGND,
      Clk  => TClk,
      InP  => tSCoutP, InN  => tSCoutN,
      FB   => tFB,     nFB  => tNFB);
  if (clock = '1')  use
    qClkTmp == eldolib.globalPwr.tVDD'reference;
  else
    qClkTmp == eldolib.globalPwr.tGND'reference;
  end use;
  vClk == qClkTmp'slew(7.5e+9, -7.5e+9);
  break on clock;  break when clock'event;
  U_SCOut == sigma2;
  U_SCOutP+U_SCOutN == 5.0;             -- Common Mode Voltage is 0.5*5.0V
  Res <= '1' when U_FB'above(2.0) else '0';
  DsmRes <= Res;
end second_order_sample_and_hold_with_ro5;
```

Beschreibung 5.21: Modell des Δ-Σ Modulators mit externer Komparator-Komponente

Kapitel 6

Zusammenfassung und Ausblick

In der vorliegenden Dissertation wurde ein simulationsbasierter Entwurfsablauf zur Entwicklung heterogener Systeme vorgestellt. Zur Klassifizierung der Simulationsbeschreibungen wurde ein Meta-Modell definiert. Durch eine Abbildung der Komponentenmodelle in dieses Meta-Modell wird ein domänenübergreifender Vergleich ermöglicht. Dieser hilft durch eine mathematisch überprüfbare Einteilung in Modellklassen bei der Zusammenführung von Modellen in ein übergeordnetes Systemmodell. Eine Gegenüberstellung typischer Modellierungsansätze aus den Bereichen des digitalen Systementwurfs, der analogen Schaltungstechnik und der Mikrosystemtechnik wurde genutzt, um ein Konzept zur Modellbildung heterogener Systeme zu motivieren, das einen domänenübergreifenden Entwurfsablauf ermöglicht.

Dabei nimmt der Abstraktionsgrad der eingesetzten Modelle einen hohen Stellenwert ein. Daher ist eine Unterteilung von Modellen in Modellklassen notwendig. Diese Unterteilung basiert auf den Anforderungen an die Berechnungsverfahren. Das Simulationssystem muss passend zu den verwendeten Modellklassen gewählt, bzw. geeignet gestaltet werden. Ebenso ist eine Unterscheidung der Modelle in Hinblick auf den Grad der Abstraktion zwischen der Modellbeschreibung und dem realen Produkt notwendig. Ein heterogenes System, das Sensorik, Signalaufbereitung, digitale Signalverarbeitung und Aktoren enthält, kann vollständig simuliert werden, wenn geeignete Modelle für die Gesamtsystemsimulation erstellt werden. Bei der Entwicklung der Bestandteile des heterogenen Systems werden jedoch genauere Modelle benötigt, welche die Eigenschaften der betrachteten Komponenten mit mehr Details beschreiben. In dieser Arbeit werden sowohl mathematische Kriterien zur Unterscheidung von Modellklassen hergeleitet, als auch ein Arbeitsfluss vorgestellt, in dem durch Variation des Komplexität der Komponentenmodelle ein effektiver Entwurf im Systemkontext ermöglicht wird.

Die vorgestelle Methodik zum Entwurf heterogener Systeme basiert auf einem konsequenten Einsatz von Zeitbereichssimulationen und modellbasierten Validierungsschritten. Dabei nimmt die Gesamtsystemsimulation einen hohen Stellenwert ein. Ein hierfür geeignetes hierarchisches Modell beschreibt die Interaktion der enthaltenen Komponenten, so dass es als Validierungsumgebung für ausgewählte Komponenten eingesetzt werden kann. Außerdem ergänzen die Komponentenmodelle die Spezifikation des Systems, indem durch eine exemplarische Implementierung das Gesamtkonzept veranschaulicht wird.

Die Wartung und Weiterentwicklung der Gesamtsystemmodelle während des Entwurfsprozesses und der fließende Übergang von vereinfachten Modellen zu realitätsnahen Beschreibungen erfordert ein Simulationssystem, in dem Modelle der unterschiedlichen Modellklassen gemeinsam simuliert werden können. Die Simulationsumgebung, die während der hier beschriebenen Doktorarbeit realisiert wurde, ermöglicht die gemeinsame Simulation von Modellen der α-Klasse, der β-Klasse und der γ-Klasse. Das zugrundegelegte Konzept baut auf einem kommerziellen VHDL-AMS Simulator, einer FPGA-Karte mit PCI-Schnittstelle, sowie einem *RAD Tool* auf. Die digitale Signalverarbeitung ist in vielen heterogenen Systemen ein wichtiger Bestandteil, der sehr komplex ist. Die FPGA-Karte ermöglicht eine Simulation der digitalen Komponenten des heterogenen Systems in Echtzeit und, wie am Beispiel eines Echtzeit-Bildverarbeitungs-Projekts dargestellt, sogar eine beschleunigte Ausführung der Aufgabenstellung. Der integrierte VHDL-AMS Simulator wird zur Ausführung von Modellen der Sensorik und der analogen Schaltungen genutzt, wobei die Bedeutung des Refactorings während der Entwicklung dieser Komponenten insbesondere am Beispiel des virtuellen Dekompressionsrechners *v*DC gezeigt wurde.

Anhand dieses Demonstrators wurde gezeigt, wie der Entwicklungsvorgang vereinfacht werden kann, indem man die Arbeitsschritte in einfache überschaubare Arbeitseinheiten zerlegt. Bei diesen wird zwischen kreativen Arbeitseinheiten und Refactorings unterschieden. Die im Anhang ausführlich beschriebenen Refactorings ermöglichen eine unkomplizierte, systematische Umformung der vorhandenen Beschreibungen in optimierte Beschreibungen. Es wurde gezeigt, wie eine Optimierung der Wartbarkeit, eine Erweiterung der Testbarkeit und eine Konkretisierung der Modelle durchgeführt werden kann.

Das Anwendungsbeispiel *v*DC wird verwendet, um den simulationsbasierten Entwurf eines heterogenen Systems in den frühen Entwurfsphasen zu erläutern. Der Dekompressionsrechner ist hierfür aus mehreren Gründen besonders geeignet. Zum einen ist das Gerät ein heterogenes System, in dem eine mechanische Drucksensorik mit einer analogen, elektrischen Signalaufbereitung, einem digitalen Filter und einem softwaregesteuerten Mikroprozessor enthalten ist. Zum anderen wird in diesem Gerät ein Differentialgleichungssystem mit Hilfe eines Modells der α-Klasse berechnet. Die Abbildung des zeit- und wertkontinuierlichen Differentialgleichungssystems in ein Modell der γ-Klasse und die sich daran anschließende Transformation in eine Beschreibung der α-Klasse wurde während der Machbarkeitsstudie durchgeführt. Die Anforderungen an die Messdatenerfassung und die Rechengenauigkeit im Mikroprozessor konnten mit den daraus gewonnenen Erkenntnissen festgelegt werden. Das entstandene Modell kann nun direkt in ein Softwareprogramm abgebildet werden, das im Mikroprozessor des *v*DC ausgeführt wird. Anhand der Transformation selbst wurde eine Anwendung des Meta-Modells demonstriert.

Die Entwicklung der digitalen Komponenten und des Softwareprogramms wurde mit Hilfe von Studenten während Semesterarbeiten und Studienarbeiten durchgeführt. Die Infrastruktur der Simulationsumgebung, die im Rahmen dieser Dissertation entwickelt

wurde, diente dabei zur Validierung. Dabei ermöglichte insbesondere die Integration der FPGA-Karte eine deutliche Verbesserung der Entwurfsqualität. Die Stimuli für das FPGA-Design wurden dabei in einem PC erzeugt und über den PCI-Bus mit bis zu 100 MB pro Sekunde übertragen. Jede digitale Komponente konnte in Echtzeit validiert werden. Das Teilsystem aus *v*DC-*Core* und darauf ausgeführtem Dekompressionsalgorithmus konnte sogar mit einer 60-fachen Beschleunigung ausgeführt werden.

Während bei der Entwicklung digitaler Schaltung ein Top-Down Entwurfsstil mit unterschiedlichen Abstraktionsebenen etabliert ist, wird bei der Entwicklung analoger Schaltungen häufig ein Bottom-Up Entwurfsverfahren angewendet. Die in dieser Arbeit vorgestellte Entwurfsmethodik ermöglicht auch bei solchen Entwicklungsaufgaben eine Verbesserung des Entwurfsablaufs, indem der Bottom-Up Entwurfsstil durch einen Meet-In-The-Middle Ansatz ersetzt wird. Dabei werden die zusätzlichen Modellierungsmöglichkeiten durch den VHDL-AMS Simulator ausgenutzt. Wie anhand des Δ-Σ A/D Wandler gezeigt wurde, ist ein vollständig simulationsgestützter Entwurfsablauf möglich. In diesem wird durch Refactoring und inkrementelle Erweiterung der vorhandenen Beschreibungen ein konzeptionelles Gesamtsystemmodell zu einem *Component Level* Modell weiterentwickelt. Hierbei werden Verhaltensmodelle mehrfach konkretisiert und partitioniert, bis sie durch vorgefertigte Makrozellen ersetzt werden können. Diese Makrozellen wurden im Bottom-Up Designstil für dieses Beispiel erzeugt. Für die während dieser Entwicklung entstandenen *Component Level* Modelle wurde ein Layout in einem Standard-CMOS-Prozess erstellt. Dieses Layout wurde anschließend auf einem *Multi-Project-Wafer* eingeschleust. Der exemplarische Test der daraus entstandenen Testmuster bestätigte den erfolgreichen Entwurf dieser Teilkomponente.

6.1 Ausblick

Die in dieser Arbeit vorgestellte Entwurfsmethodik für heterogene Systeme wurde anhand mehrere Beispiele veranschaulicht. Diese bestanden aus piezoelektrischer Sensorik, analogen und digitalen elektronischen Schaltungen, sowie Firmware-Programmen, die in einem applikationsspezifischen Prozessor verarbeitet wurden. Die Entwurfsmethodik ist jedoch nicht auf diese Anwendungsdomänen beschränkt. Die Durchführung einer Anwendungsstudie im Bereich des Maschinenbaus, der elektrischen Antriebstechnik oder der chemischen Verfahrenstechnik könnte insbesondere in Bezug auf die Refactorings zu zusätzlichen Erkenntnissen und zu einer Erweiterung der vorhandenen Refactoringvorschriften führen.

Das zugrunde liegende mathematische Meta-Modell wurde so definiert, dass eine Beschreibung von räumlich verteilten Problemstellungen mit ermöglicht wird. Die speziellen Anforderungen im Bereich der *Finiten Elemente Simulationen* werden innerhalb der Modelle der δ-Klasse zu einer weiteren Differenzierung führen. Ein Simulationssystem zur Simulation von Modellen der δ-Klasse mit Modellen anderer Modellklassen wird umso sinnvoller, je genauer die Möglichkeiten und Grenzen des Simulationssystems

aufgezeigt werden können. Es erscheint deshalb vielversprechend, zusätzliche Randbedingungen zu definieren, mit denen die Modelle der δ-Klasse entsprechend unterteilt werden können.

Bei der Entwicklung der analogen elektrischen Schaltungen wurden Makrozellen erstellt, die auch in zukünftigen Projekten eingesetzt werden können. Wenn genügend dieser Makrozellen vorhanden sind, ist eine Erweiterung der vorgestellten Entwurfsmethodik möglich, die eine automatisierte Synthese analoger Schaltungen unterstützt.

Letztendlich ist das Projekt vDC während der vorgestellten Dissertation so weit entwickelt worden, dass eine Realisierung des Dekompressionsrechners in einem System-On-Chip Design mit integriertem Drucksensor, Signalaufbereitung, Mikroprozessor, RAM und EEPROM Zellen möglich ist.

Anhang

Anhang A

Katalog der Refactoring-Methoden

A.1 Datenorientierte Methoden

Refactoring 1 (Extract Signal, Quantity or Terminal):
Ist-Zustand: Ein Zwischenergebnis eines Prozesses wird in einem anderen Prozess oder einer anderen Komponente benötigt.

Vorgehensweise: Je nach Art der Berechnung wird das Zwischenergebnis als ereignisdiskrete, zeitkontinuierliche oder gar konservative zeitkontinuierliche Zustandsgröße exportiert. Dazu wird ein *Signal*, eine *Quantity* oder ein *Terminal* definiert, dem der Wert des Zwischenergebnisses zugewiesen wird.

Motivation: Um duplizierten Code zu vermeiden, ist es sinnvoll mehrfach benötigte Berechnungsergebnisse als Zustandsgrößen zur Verfügung zu stellen. Indem man der Zustandsgröße einen geeigneten Namen gibt, kann zusätzlich die Funktion der Berechnung erläutert werden.

Refactoring 2 (Remove Signal, Quantity or Terminal):
Ist-Zustand: Eine Komponente enthält eine Zustandsgröße, die nur innerhalb eines Prozesses genutzt wird.

Vorgehensweise: Die Zustandsgröße wird durch eine Variable ersetzt.

Motivation: In VHDL-AMS werden *Signale*, *Quantities* und *Terminals* benötigt, um Daten über die Grenzen eines Prozesses hinweg auszutauschen. Wenn ein Berechnungsergebnis nur innerhalb des Prozesses benötigt wird, sind Variablen sinnvoller, da Variablen in ihrem Gültigkeitsbereich auf den Prozess beschränkt sind, indem sie definiert werden.

Refactoring 3 (Introduce Explaining Variable):
Ist-Zustand: Das Modell enthält einen komplizierten Ausdruck.

Vorgehensweise: Das Ergebnis oder ein Zwischenergebnis wird einer neuen Variable zugewiesen, deren Name die Bedeutung des Ergebnisses erklärt.

Motivation: Sowohl logische Ausdrücke, als auch Berechnungsvorschriften können sehr komplex und schwer lesbar werden. In solchen Situationen kann eine Zwischenvariable hilfreich sein, um das Modellverhalten in kurzen und gut wartbaren Ausdrücken zu beschreiben.

```
Org_A: process (Clk, Rst) is
  variable float_exp : signed(7 downto 0);
  variable float_man : signed(22 downto 0);
  variable float_denormalized : std_logic;
begin
  if (Rst = '1') then
    res_A <= (others => '0');
  elsif (Clk'event and Clk = '1') then
    float_exp := NumberA(30 downto 23);
    float_man := NumberA(22 downto 0);
    if (float_exp = "10000000" and conv_integer(float_man) /= 0) then
      float_denormalized := '1';
    else float_denormalized := '0';
    end if;
    res_A <= ...;
  end if;
end process Org_A;
Org_B: process (Clk, Rst) is
begin
  if (Rst = '1') then
    res_B <= (others => '0');
  elsif (Clk'event and Clk = '1') then
    if (NumberA(30 downto 23) = "10000000" and
        conv_integer(NumberA(22 downto 0)) /= 0) then
      res_B <= ...;
    else res_B <= ...;
    end if;
  end if;
end process Org_B;
-------------------------------------------------------------------------------
signal NumberA_denormalized : std_logic;
...
Refactored_A: process (Clk, Rst) is
  variable float_exp : signed(7 downto 0);
  variable float_man : signed(22 downto 0);
begin
  if (Rst = '1') then
    res_A <= (others => '0');
  elsif (Clk'event and Clk = '1') then
    float_exp := NumberA(30 downto 23);
    float_man := NumberA(22 downto 0);
    if (float_exp = "10000000" and conv_integer(float_man) /= 0) then
      NumberA_denormalized <= '1';
    else NumberA_denormalized <= '0';
    end if;
    res_A <= ...;
  end if;
end process Refactored_A;
Refactored_B: process (Clk, Rst) is
begin
  if (Rst = '1') then
    res_B <= (others => '0');
  elsif (Clk'event and Clk = '1') then
    if (NumberA_denormalized = '1') then
      res_B <= ...;
    else res_B <= ...;
    end if;
  end if;
end process Refactored_B;
```

Beschreibung A.1: Durch das zusätzliche Signal *NumberA_denormalized* kann im Prozess B ohne Dekodierung auf denormalisierte Zahlen reagiert werden. (Rfact. 1)

```
Original: process (Clk, Rst) is
begin
  if Rst = '1' then
    Count <= (others => '0');
  elsif Clk'event and Clk = '1' then
    Count <= Count + 1;
  end if;
end process Original;
Trigger <= Count(2);
-----------------------------------------------------------------------------
Refactored: process (Clk, Rst) is
variable Count : unsigned(2 downto 0);
begin
  if Rst = '1' then
    Count := (others => '0');
  elsif Clk'event and Clk = '1' then
    Count := Count + 1;
    Trigger <= Count(2);
  end if;
end process Refactored;
```

Beschreibung A.2: Außerhalb des Prozesses wird nur das Signal *Trigger* benötigt. Im Prozess *Refactored* ist der Zählerstand eine Variable und es wird direkt das benötige Signal *Trigger* berechnet. (Rfact. 2)

```
Original: process (x11, x12, x21, x22) is
begin
  if (x11+x12 > x21+x22) then
    Haar <= '0';
  else
    Haar <= '1;
  end if;
  sum <= x11+x12+x21+x22;
end process Original;
-----------------------------------------------------------------------------
Refactored: process (x11, x12, x21, x22) is
variable sum_left, sum_right : unsigned(x11'range);
begin
  sum_left := x11+x12;
  sum_right := x21+x22;
  if (sum_left > sum_right) then
    Haar <= '0';
  else
    Haar <= '1;
  end if;
  sum <= sum_left+sum_right;
end process Refactored;
```

Beschreibung A.3: Das Beispiel ist einer Implementierung des *MPEG-7 Compact Descriptor Algorithm* entnommen. Im Prozess *Refactored* erkennt man durch die hinzugefügten Variablen besser, dass der Prozess den Haar-Koeffizienten durch Vergleich der linken mit der rechten Bildhälfte berechnet. (Rfact. 3)

```
Original: process (x11, x12, x21, x22) is
variable temp : signed(x11'range);
begin
  temp := x11-x12;
  Haar1 <= temp(temp'high);
  temp := x21-x22;
  Haar2 <= temp(temp'high);
end process Original;
-----------------------------------------------------------------------------
Refactored: process (x11, x12, x21, x22) is
variable diff_left, diff_right : signed(x11'range);
begin
  diff_left := x11-x12;
  Haar1 <= diff_left(diff_left'high);
  diff_right := x21-x22;
  Haar2 <= diff_right(diff_right'high);
end process Refactored;
```

Beschreibung A.4: Beispiel für die Einführung dedizierter lokaler Variablen (Rfact. 4)

Refactoring 4 (Split Multiple Used Temporary Variable):
Ist-Zustand: In einem sequentiellen Pfad eines Modells wird einer lokalen Variable mehrfach als Zwischenspeicher für verschiedene Werte genutzt.

Vorgehensweise: Für jeden benötigten Zwischenspeicher wird eine eigene lokale Variable definiert.

Motivation: Viele lokale Variablen dienen zum Speichern von Zwischenergebnissen, die später weiterverwendet werden sollen. Bei vielen solcher Variablen ist es hilfreich für jedes Zwischenergebnis eine eigene Variable zu benutzen, um Verwirrung beim Lesen des Modells zu vermeiden. Bei manchen lokalen Variablen ist es jedoch nicht sinnvoll dieses Refactoring anzuwenden. Schleifenvariablen und Variablen, in denen Teilergebnisse aufsummiert werden, gehören zu der Kategorie der Variablen, bei denen mehrfach Zuweisungen sinnvoll sind.

Refactoring 5 (Use Bus Resolution Functions):
Ist-Zustand: Verschiedene Signalquellen werden auf einen gemeinsamen Datenbus geschaltet und die Instandhaltung des dazugehörigen Multiplexers ist mühsam.

Vorgehensweise: An Stelle des expliziten Multiplexers werden die Signalquellen so erweitert, dass ein direktes Zusammenschalten der Signale über eine Bus Resolution Function zu einem gleichwertigen Signal auf dem Datenbus führt.

Motivation: An Stelle des expliziten Multiplexers kann Tristate-Logik, Open Collector-Logik oder ähnliches eingesetzt werden. Insbesondere wenn die Implementierungsentscheidung erst zu einem späteren Zeitpunkt getroffen werden soll, ist ein Modell mit Bus-Resolution-Function wartungsfreundlicher. Dadurch ist ein von der Anzahl der Signalquellen unabhängiges Konzept gegeben.

```
Original: process (Addr, x1, x2, x3, x4) is
begin
  case Addr is
    when "00" => DataToPCI <= x1;
    when "01" => DataToPCI <= x2;
    when "10" => DataToPCI <= x3;
    when "11" => DataToPCI <= x4;
    when others => DataToPCI <= (others => '0');
  end case;
end process Original;
-------------------------------------------------------------------------------
Refactored: block is
begin
  DataToPCI <= x1 when Addr = "00" else (others => 'Z');
  DataToPCI <= x2 when Addr = "01" else (others => 'Z');
  DataToPCI <= x3 when Addr = "10" else (others => 'Z');
  DataToPCI <= x4 when Addr = "11" else (others => 'Z');
end block Refactored;
```

Beschreibung A.5: Das Beispiel veranschaulicht den Zugriff auf den PCI-Bus einer Xilinx FPGA-Karte. Der Prozess nutzt eine Multiplexer Struktur, während der Block mit Hilfe von Tristate-Treibern auf eine gemeinsame Busleitung schreibt. Wenn die Anzahl der Adressen groß ist, sind Tristate-Treiber vorteilhaft. (Rfact. 5)

Refactoring 6 (Duplicate Observed Data):
Ist-Zustand: In einer synchronen Komponente werden Daten aus einer nicht mit dem selben Takt synchronisierten Komponente benötigt.

Vorgehensweise: Es wird ein synchron getakteter Speicher für die Daten und eine speziell zugeschnittene Komponente, die für die Synchronisierung der Daten sorgt, angelegt.

Motivation: In einigen Fällen kann der lesende Zugriff auf Daten aus einer anderen Clock-Domäne ohne spezielle Zwischenspeicher stattfinden. Dazu müssen jedoch die Taktflanken der Clock-Domänen zu einander synchronisiert sein. Wenn diese Abhängigkeiten der Taktsignale während der Projektentwicklung geändert werden, können Folgefehler durch den Zugriff auf Daten der anderen Clock-Domäne entstehen. Ebenso können durch den Zugriff auf Daten aus asynchronen, ereignisgesteuerten Komponenten schwer wartbare Systeme entstehen, bei denen Änderungen im Zeitverhalten einer Komponente Fehlverhalten in anderen Komponenten zur Folge hat. Dieses Refactoring verhindert solche Folgefehler.

Refactoring 7 (Replace Magic Number with Symbolic Constant):
Ist-Zustand: In einem Modell wird eine numerische Kennung mit festgelegter Bedeutung benutzt.

Vorgehensweise: Es wird eine symbolische Konstante mit dem Wert der numerischen Kennung definiert. Der Name der Konstante wird dem Verwendungszweck entsprechend gewählt.

```
Original: process (Clk, Rst) is
begin
  if Rst = '1' then
      ...;
  elsif Clk'event and Clk = '1' then
    if (unsigned(fexp(MyFloat)) = "10000000") then
      -- denormalized number detected:
      ...;
    end if;
  end if;
end process Original;
-------------------------------------------------------------------------------
Refactored: process (Clk, Rst) is
constant fexp_denormalized : std_logic_vector(7 downto 0) := "10000000";
begin
  if Rst = '1' then
      ...;
  elsif Clk'event and Clk = '1' then
    if (fexp(MyFloat) = fexp_denormalized) then
     ...;
    end if;
  end if;
end process Refactored;
```

Beschreibung A.6: Durch die symbolische Konstante wird der Test auf denormalisierte Fließkommazahlen selbsterklärend. (Rfact. 7)

Motivation: Numerische Kennungen sind eine der ältesten Probleme der Datenverarbeitung. Diese Zahlen haben oft keine offensichtliche Bedeutung, was leicht zu unnötigen Konsistenzproblemen führt, insbesondere wenn die numerischen Kennungen in mehreren Teilmodellen eingesetzt werden. Sollte sich die Kennung ändern, ist das Durchführen dieser Änderung eine sehr fehleranfällige Sisyphusarbeit. Eine symbolische Konstante ist schnell eingefügt und ist für die Lesbarkeit des Modells ein großer Fortschritt.

Refactoring 8 (Change Signal/Quantity/Variable to Constant Value):
Ist-Zustand: Ein *Signal* oder eine *Quantity* soll während der Simulation seinen Wert nicht ändern.

Vorgehensweise: Die Deklaration der Größe wird in eine Konstantendeklaration umgewandelt. Der feste Wert der Größe wird direkt in der Konstantendeklaration angegeben.

Motivation: Die Deklaration einer Größe als *Signal*, *Quantity* oder *Variable* legt nahe, dass die Größe geändert werden kann. Wenn eine Änderung nicht erwünscht ist, sollte die Größe als Konstante definiert werden. Das verdeutlicht, dass die Größe nicht geändert werden soll. Die Beschreibung wird besser verständlich und während der Simulation sind automatisierte Optimierungen möglich.

Refactoring 9 (Remove Symbolic Constant):
Ist-Zustand: Die Verhaltensbeschreibung enthält eine nicht benutzte symbolische Konstante.

```
architecture Original of MyEntity is
  type   tState is (Idle, DoShift, s3);
  signal FSMState : tState;
begin
  process (Clk, rst) is
  begin
    if rst = '1' then
      ...;
    elsif Clk'event and Clk = '1' then
      case FSMState is
        when Idle =>
          if start = '1' then
            FSMState <= DoShift;
          end if;
        when DoShift =>
          ...;
          FSMState <= Idle;
        when others =>
          FSMState <= Idle;
      end case;
    end if;
  end process;
end architecture Original;
--------------------------------------------------------------------------------
architecture Refactored of MyEntity is
  type   tState is (Idle, DoShift);
  signal FSMState : tState;
begin
  process (Clk, rst) is
  begin
    if rst = '1' then
      ...;
    elsif Clk'event and Clk = '1' then
      case FSMState is
        when Idle =>
          if start = '1' then
            FSMState <= DoShift;
          end if;
        when DoShift =>
          ...;
          FSMState <= Idle;
        when others =>
          FSMState <= Idle;
      end case;
    end if;
  end process;
end architecture Refactored;
```

Beschreibung A.7: Der nicht genutzte Bezeichner *s3* wurde aus der Menge der Zustände gelöscht. (Rfact. 9)

Vorgehensweise: Die symbolische Konstante wird entfernt.

Motivation: Ungenutzte symbolische Bezeichner entstehen durch Änderungen oder Erweiterungen einer Modellbeschreibung. Neben dem trivialen Fall der offensichtlich ungenutzten symbolischen Konstanten zählen auch die Elemente eines anwendungsspezifisch definierten Aufzählungstyps zu den symbolischen Konstanten. Solche Aufzählungstypen werden z. B. als Zustandsmenge bei der Beschreibung endlicher Automaten

eingesetzt. Durch Erweiterungen an einem endlichen Automaten kommt es oft zu Erweiterungen der dazugehörigen Zustandsmenge. Nach einer Vereinfachung eines endlichen Automaten sollte die Zustandsmenge überprüft und die nicht mehr benötigten Elemente entfernt werden.

Refactoring 10 (Remove Middle Man):
Ist-Zustand: In einem hierarchischen Modell existiert ein Teilmodell A, über das Daten zwischen zwei anderen Komponenten ausgetauscht werden, ohne das diese Daten in Teilmodell A benötigt werden.

Vorgehensweise: Die beiden anderen Komponenten werden so geändert, dass sie direkt miteinander kommunizieren können. Die Beschreibung von Teilmodell A wird durch Entfernen der überflüssigen Datenverbindungen vereinfacht.

Motivation: Für eine saubere hierarchische Partitionierung eines Gesamtmodells kann es sinnvoll sein, den Datenaustausch zwischen Teilmodellen über ausgewählte I/O Komponenten durchzuführen. Diese Partitionierung kostet jedoch ihren Preis. Bei der Weiterentwicklung des Gesamtmodells von einem konzeptionellen Verhaltensmodell zu einem implementierungsnah beschriebenen virtuellen Prototypen, wächst die Anzahl der durch Strukturverknüpfungen gebildeten Komponenten, in denen weitläufige Datenverbindungen zu unnötig komplexen Port-Deklarationen führen.

Refactoring 11 (Remove Multiple Assignments to Quantities):
Ist-Zustand: Eine Komponentenbeschreibung enthält ein *if* ... *use* Konstrukt, indem einer *Quantity* in Abhängigkeit unterschiedlicher boolescher Bedingungen vorberechnete Werte zugewiesen werden.

Vorgehensweise: An Stelle des *if* ... *use* Konstrukts wird eine äquivalente Funktion definiert, deren Rückgabewert der *Quantity* zugewiesen wird.

Motivation: Dieses Refactoring bezieht sich auf die aktuelle Umsetzung der VHDL-AMS Sprache in dem Produkt AdvanceMS der Firma Mentor Graphics. Die Produktversion aus dem Jahr 2002 kann *Quantities* nur mit Hilfe von impliziten Gleichungen berechnen. Das kann zu Modellen führen, die viele *if* ... *use* Konstrukte enthalten. Die in den Ausführungszweigen enthaltenen impliziten Gleichungen bilden jedoch den Signalfluss einer Wertzuweisung nicht ab. Die Absicht einer *Quantity* einen vorgegebenen Wert zuzuweisen, lässt sich jedoch in einer anwendungsspezifischen Funktion gut darstellen. Als zusätzlicher Vorteil wird die Anzahl der simultanen Gleichungen reduziert und die Simulation effizienter.

Refactoring 12 (Remove Multiple Assignments to Entity Ports):
Ist-Zustand: Aufgrund des internen Zustands kann in einer Komponente das Verhalten an den Ausgängen unterschiedlich berechnet werden. In dem Komponentenmodell wird das Berechnungsergebnis teilweise direkt in den bedingten Zweigen einem Ausgangssignal zugewiesen. Es existieren dadurch an mehreren Stellen Zuweisungen an das selbe Ausgangssignal.

```
architecture Original of MyStimuli is
  quantity P across P_Flow through Pressure_Sensor;
begin
  if (now < 3.0 us) use
    p == 2.5*time2real(now/3.0 us);
  elsif (now < 28.0 us) use
    p == 2.5;
  elsif (now < 34.0 us) use
    p == 2.5*(1-time2real((now-28.0 us)/6.0 us));
  else
    p == 0.0;
  end use;
end architecture Original;
-------------------------------------------------------------------------------
architecture Refactored of MyStimuli is
  function pressure (aTime : time) return real is
  begin
    if (aTime < 3.0 us) then
      return(2.5*time2real(aTime/3.0 us));
    elsif (aTime < 28.0 us) then
      return(2.5);
    elsif (aTime < 34.0 us) then
      return(2.5*(1-time2real((aTime-28.0 us)/6.0 us)));
    end if;
    return(0.0);
  end function pressure;

  quantity P across P_Flow through Pressure_Sensor;
begin
  p == pressure(now);
end architecture Refactored;
```

Beschreibung A.8: In obigem Stimuligenerator wurde das *if* ... *use* Konstrukt durch einen Funktionsaufruf ersetzt. (Rfact. 11)

Vorgehensweise: Das Verhaltensmodell wird um eine interne Hilfsgröße erweitert. Diese Hilfsgröße kann eine *Quantity*, ein *Terminal*, ein *Signal* oder eine *Variable* sein. In den bedingten Zweigen werden die Berechnungsergebnisse der internen Hilfsgröße zugewiesen, die letztendlich mit dem *Entity Signal* verknüpft wird.

Motivation: Fehlerhaftes Verhalten auf *Entity Ports* ist schwer lokalisierbar, da (insbesondere bei *Terminals* und *Quantities*) die Fehlerquelle nicht ohne weiteres auf einen Verursacher eingegrenzt werden kann. Durch Analyse der internen Hilfsgrößen kann die Fehlerursache leichter lokalisiert werden.

```
architecture Original of MyEntity is
  signal mem : std_logic_vector(7 downto 0);
begin
  process (Clk, rst) is
  begin
    if rst = '1' then
      mem <= (0 => '1', others => '0');
      res <= '0';
    elsif Clk'event and Clk = '1' then
      if load = '1' then
          mem <= load_value;
      elsif run = '1' then
          mem <= shl(mem,1);
          if (mem(5 downto 3)) = "000" then
            res <= '1';
          else
            res <= '0';
          end if;
      end if;
    end if;
  end process;
end architecture Original;
--------------------------------------------------------------------------------
architecture Refactored of MyEntity is
  signal mem : std_logic_vector(7 downto 0);
  signal myRes : std_logic;
begin
  process (Clk, rst) is
    variable newValue : std_logic;
  begin
    if rst = '1' then
      mem <= (0 => '1', others => '0');
      myRes <= '0';
    elsif Clk'event and Clk = '1' then
      newValue := myRes;
      if load = '1' then
          mem <= load_value;
      elsif run = '1' then
          mem <= shl(mem,1);
          if (mem(5 downto 3)) = "000" then
            newValue := '1';
          else
            newValue := '0';
          end if;
      end if;
      myRes <= newValue;
    end if;
  end process;
  res <=myRes;
end architecture Refactored;
```

Beschreibung A.9: Das Beispiel zeigt ein Modell, indem *Res* nicht mit jedem Takt neu berechnet wird. Die Verwendung der Variablen *newValue* verdeutlicht dies. (Rfact. 12)

```
architecture Original of MyEntity is
  quantity SensOut across TSensor;
  signal data0, data1, data2, data3 : std_logic;
begin
  adc_1: entity work.adc
    port map (
      Vin  => SensOut,
      res0 => data0,
      res1 => data1,
      res2 => data2,
      res3 => data3);
  ...;
end architecture Original;
-----------------------------------------------------------------------------
architecture Refactored of MyEntity is
  quantity SensOut across TSensor;
  signal data : std_logic_vector(3 downto 0);
begin
  adc_1: entity work.adc
    port map (
      Vin => SensOut,
      res => data);
  ...;
end architecture Refactored;
```

Beschreibung A.10: Die Zusammenfassung von zusammengehörenden Signalen ermöglicht kompaktere Modelle. (Rfact. 13)

Refactoring 13 (Group Signals/Quantities/Terminals to Vectors):
Ist-Zustand: Ein Modell enthält eine Gruppe von einzelnen Zustandsgrößen, die zusammengehören.

Vorgehensweise: Wenn die einzelnen Zustandsgrößen alle vom selben Typ sind, werden sie durch ein *Array* (Vektor) ersetzt, andernfalls durch einen *Record*.

Motivation: Oft gibt es eine Gruppe von Zustandsgrößen, die gemeinsam weitergegeben werden. Verschiedene Komponenten, bzw. verschiedene Prozesse nutzen diese Gruppe von Zustandsgrößen. Solche Gruppen sollten in einer gemeinsamen Datenstruktur zusammengefasst werden. Das reduziert die Anzahl der Zustandsgrößennamen und ermöglicht eine kompaktere Schreibweise und verständlichere Modelle.

A.2 Kontrollflussorientierte Methoden

Refactoring 14 (Decompose Conditional):
Ist-Zustand: Das Verhaltensmodell enthält komplizierte Kontrollogik oder komplexe, nur bedingt auszuführende Datenpfade.

Vorgehensweise: Umfangreiche Bedingungen werden in Funktionen zusammengefasst und komplexe Datenflussbeschreibungen werden durch Prozeduren ersetzt.

```
architecture Original of MyEntity is
begin
  process (Clk, Rst) is
  begin
    if Rst = '1' then
      ...;
    elsif Clk'event and Clk = '1' then
      if (unsigned(fexp(MyFloat)) = 128) then
        -- normalize MyFloat:
        ...;
      end if;
    end if;
  end process;
end architecture Original;
-------------------------------------------------------------------------------
architecture Refactored of MyEntity is
  function denormalized(arg:std_logic_vector)
    return boolean is
  begin
    return (unsigned(fexp(arg)) = 128);
  end function denormalized;
begin
  process (Clk, Rst) is
  begin
    if Rst = '1' then
      ...;
    elsif Clk'event and Clk = '1' then
      if denormalized(MyFloat) then
        ...;
      end if;
    end if;
  end process;
end architecture Refactored;
```

Beschreibung A.11: Durch die Funktion *denormalized* wird das Verhalten des Prozesses verdeutlicht. (Rfact. 14)

Motivation: Einer der häufigsten Gründe für komplizierte Modelle ist eine umfangreiche Kontrollogik mit eingebetteter Datenverarbeitung. Die Größe einer solchen Verhaltensbeschreibung ist ein Faktor, der die Lesbarkeit einschränkt, i. d. R. liegt das Problem jedoch darin, dass sowohl in den Bedingungen, als auch im Datenpfad Informationen enthalten sind, die beschreiben was passiert, jedoch nicht, warum es passiert. Durch die Aufteilung der Verhaltensbeschreibung in mehrere Unterprogrammaufrufe ergibt sich die Möglichkeit mit Hilfe von geeigneten Funktions- und Prozedur-Namen eine Begründung für das Verhalten hervorzuheben, während gleichzeitig die Beschreibung kompakter wird.

Refactoring 15 (Consolidate Conditional Expression):
Ist-Zustand: Das Modell enthält eine Reihe von if-Konstrukten, die das selbe Ergebnis zur Folge haben.

Vorgehensweise: Die if-Konstrukte werden durch ein gemeinsames if-Konstrukt ersetzt, dessen Bedingung aus einer Verknüpfung der ursprünglichen Bedingungen gebildet wird. In der Regel ist es sinnvoll diese neue Bedingung in eine Funktion zu extrahieren.

```
Original: process (Clk, rst) is
begin
  if rst = '1' then
    StartStop <= "00";
  elsif Clk'event and Clk = '1' then
    if (FSMState=Waiting) then
      StartStop <= StartStop + "01";
    elsif (FSMState=StartSend) then
      StartStop <= StartStop + "01";
    elsif (FSMState=Pause) then
      StartStop <= StartStop + "01";
    else
      StartStop <= "10";
    end if;
  end if;
end process Original;
-------------------------------------------------------------------------------
Refactored: process (Clk, rst) is
begin
  if rst = '1' then
    StartStop <= "00";
  elsif Clk'event and Clk = '1' then
    if (FSMState=Waiting)
      or (FSMState=StartSend)
      or (FSMState=Pause) then
      StartStop <= StartStop + "01";
    else
      StartStop <= "10";
    end if;
  end if;
end process Refactored;
```

Beschreibung A.12: Der Prozess *Refactored* zeigt das Ergebnis des Refactorings Rfact. 15.

Motivation: Manchmal sieht man eine Reihe von if-Konstrukten, die jeweils unterschiedliche Tests darstellen, bei denen die resultierende Aktion jedoch gleich ist. In diesen Fällen sollte man die Tests mit kombinatorischen Verknüpfungen zu einem gemeinsamen Test zusammen fassen. Dieses Refactoring ist aus zwei Gründen sinnvoll. Zum einen wird die Überprüfung leicht verständlich, da alle Gründe, die zu der spezifizierten Handlung führen, gruppiert sind. Zum anderen ist das Refactoring hilfreich, da es eine effektive Anwendung des Refactorings *Extract Function/Procedure* auf die Bedingungen ermöglicht.

Refactoring 16 (Consolidate Duplicate Conditional Expression):
Ist-Zustand: In einem mehrfach verzweigten if-Konstrukt ist in allen Zweigen eine gemeinsame Bedingung enthalten.

Vorgehensweise: Die gemeinsame Bedingung wird extrahiert und in ein zusätzliches if-Konstrukt verschoben, welches das restliche Konstrukt umschließt.

Motivation: Wenn man ein Modell liest, das in allen Bedingungen eines if-Konstruktes einen gemeinsamen booleschen Ausdruck enthält, sollte man diesen duplizierten Aus-

```
Original : process (Clk, rst) is
begin
  if rst = '1' then
    FSMState <= Idle;
  elsif Clk'event and Clk = '1' then
    if (BitCounter > 12 and LastByte = '1') then
      FSMState <= EndTransmission;
    elsif (MasterState = Receiving and LastByte = '1') then
      FSMState <= Idle;
    end if;
  end if;
end process Original;
-----------------------------------------------------------------------------
Refactored : process (Clk, rst) is
begin
  if rst = '1' then
    FSMState <= Idle;
  elsif Clk'event and Clk = '1' then
    if (LastByte = '1') then
      if (BitCounter > 12) then
        FSMState <= EndTransmission;
      elsif (MasterState = Receiving) then
        FSMState <= Idle;
      end if;
    end if;
  end if;
end process Refactored;
```

Beschreibung A.13: Die gemeinsame Bedingung *LastByte='1'* wurde in eine äußere if-Bedingung verlagert. (Rfact. 16)

druck entfernen, so dass es leichter zu sehen ist, zwischen welchen Bedingungen in den Zweigen des if-Konstruktes unterschieden wird.

Refactoring 17 (Introduce Assertion):
Ist-Zustand: Ein Teilmodell ist nur sinnvoll modelliert, wenn bestimmte Randbedingungen erfüllt sind. Diese Randbedingungen sind jedoch zum Zeitpunkt der Modellerstellung nicht überprüfbar.

Vorgehensweise: Durch Einführung eines Assert-Statements wird die Überprüfung der Randbedingung zur Laufzeit erzwungen.

Motivation: Oft sind Verhaltensmodelle nur unter bestimmten Bedingungen richtig. Das kann eine triviale Voraussetzung sein, wie z. B. das die Länge eines Bitvektors positiv sein muss. Solche Voraussetzungen werden oft nicht notiert und können zum Teil nur durch Analyse der Gesamtmodelle festgestellt werden. Manchmal sind solche Voraussetzungen als Kommentare im Verhaltensmodell enthalten. Ein Assert-Statement ist jedoch sinnvoller, da es im Fehlerfall automatisch die Fehlerursache meldet.

Refactoring 18 (Rename Process/Entity):
Ist-Zustand: Der Name eines Prozesses oder einer *Entity* verrät nicht den Zweck der Komponente.

```
entity MultiTurn is
  generic (
    ResGes : natural := 12;
    ResSingle : natural := 6);
  port (
    Clk,rst    : in std_logic;
    CntUpDn    : in std_logic_vector(1 downto 0);
    CntSingle  : out std_logic_vector(ResSingle-1 downto 0);
    CntMulti   : out std_logic_vector(ResGes-ResSingle-1 downto 0));
begin
  assert (ResGes>=ResSingle)
    report ("Die Gesamtauflösung kann nicht kleiner sein"
            & " als die Singleturn-Auflösung!")
    severity failure;
end entity MultiTurn;
```

Beschreibung A.14: Der VHDL-Code zeigt nur die *Entity* Deklaration mit *assert*-Statement. Die ursprüngliche Beschreibung war damit bis auf die Überprüfung der Bitlängen identisch. (Rfact. 17)

Vorgehensweise: Ändern des Namens, so dass die Bedeutung des Teilmodells klar wird.

Motivation: Ein wichtiger Bestandteil von guten Modellen ist die Partitionierung komplexer Systeme in überschaubare Komponenten. Dabei ist die Benennung der Komponenten wichtig, um die Funktion der Komponenten zu verdeutlichen. Ein guter Komponentenname zeichnet sich dadurch aus, dass er die Bedeutung der Komponente klarstellt. Natürlich wird man nicht immer im ersten Anlauf den besten Namen für eine Komponente finden, trotzdem ist es wichtig sich um sinnvolle Namen zu bemühen. Wenn sich nach einigen Änderungen an einer Komponente herausstellt, dass der Name nicht mehr aussagekräftig ist, sollte er geändert werden. Das gilt ebenso für andere Aspekte der Komponentendeklaration. Wenn eine Umordnung der Parameter oder Eingangssignale zum Verständnis der Funktionalität der Komponente beiträgt, sollte sie durchgeführt werden.

Refactoring 19 (Replace Error Handling with Error Signaling):
Ist-Zustand: In einem Modell werden Fehlerzustände überprüft, die in den spezifizierten Anwendungsfällen nicht auftreten dürfen. Für diese Fehlerzustände wird Ersatz-Verhalten modelliert.

Vorgehensweise: Anstatt bei einem Fehler einen komplizierten alternativen Datenfluss zu modellieren, wird der Fehler über ein Signal gemeldet und die normale Datenverarbeitung wird fortgesetzt.

Motivation: Die Modellierung von Verhalten für fehlerhaften Einsatz einer Komponente führt zu komplexen Modellen, deren Wartung zusätzlichen Aufwand erfordert. Andererseits ist es für den Anwender des Komponentenmodells sinnvoll, die nicht spezifikationsgerechte Anwendung des Modells zu melden. Für diese Mitteilung an den Anwender ist jedoch ein Fehlersignal ausreichend. Das Setzen eines Fehlersignals ist eine elegante

```
architecture Orginal of Comparator is
quantity deltaU across T_plus to T_minus;
signal Dout : std_logic;
begin
  a2d: process (deltaU'above(-0.1),
                deltaU'above(0.1),
                T_plus'reference'above(0.7),
                T_minus'reference'above(0.7)) is
  begin
    if ((T_plus'reference < 0.7)
        and (T_minus'reference < 0.7)) then
      Dout <='U';
    elsif (deltaU'above(0.1)) then
      Dout <= '1';
    elsif not(deltaU'above(-0.1)) then
      Dout <= '0';
    end if;
  end process a2d;
end architecture Orginal;
--------------------------------------------------------------------------------
architecture Refactored of Comparator is
quantity deltaU across T_plus to T_minus;
signal Dout : std_logic;
begin
  a2d: process (deltaU'above(-0.1),deltaU'above(0.1)) is
  begin
    if (deltaU'above(0.1)) then
      Dout <= '1';
    elsif not(deltaU'above(-0.1)) then
      Dout <= '0';
    end if;
  end process a2d;
  error <= '0' when ((T_plus'reference'above(0.7)
                      or (T_minus'reference'above(0.7)))
           else '1';
end architecture Refactored;
```

Beschreibung A.15: In diesem Beispiel wird ein Schmitt-Trigger Komparator mit einer n-Eingangsstufe beschrieben. Der Komparator funktioniert nur, wenn eine der Eingangsspannungen über 0.7 V liegt. Die Fehlerbehandlung im Originalmodell kann in den angeschlossenen Komponenten zu Folgefehlern führen, während das Modell mit Fehlersignal die Quelle des Fehlers eindeutig meldet. (Rfact. 19)

Möglichkeit die zu erwartende Diskrepanz zwischen Modellverhalten und realistischem Verhalten zu melden, ohne die Simulation abzubrechen.

Refactoring 20 (Extract Group of States):
Ist-Zustand: Ein Prozess enthält einen umfangreichen endlichen Automaten, aus dem eine Gruppe von Zustandsübergängen in einen eigenen endlichen Automaten verschoben werden können.

Vorgehensweise: Für die zu extrahierenden Zustandsübergänge wird ein eigener Aufzählungstyp deklariert. Anschließend werden die zu diesen Zustandsübergängen gehörenden Beschreibungen in einen eigenen Prozess verschoben.

Motivation: Lange sequentielle Prozesse sind schwer zu verstehen und dementsprechend anfällig für versteckte Modellierungsfehler. Durch die Restrukturierung von Prozessen mit langen *Case*-Konstrukten in mehrere Prozesse mit einer überschaubaren Anzahl von Zuständen, wird die Funktionsweise der einzelnen endlichen Automaten deutlicher dargestellt, so dass Fehlerquellen einfacher zu finden sind. Nebenbei führt diese Restrukturierung i. d. R. zu besseren Ergebnissen bei der Synthese synchroner Teilschaltungen.

A.3 Strukturbildende Methoden

Refactoring 21 (Add Port to Entity Declaration):
Ist-Zustand: Eine Komponente benötigt einen zusätzlichen Informationskanal zu einer anderen Komponente.

Vorgehensweise: Die *Entity* Deklaration beider Komponenten wird um den zusätzlichen Informationskanal erweitert. In der Regel wird ein unidirektionaler Datenaustausch diskreter Informationen mit einem *Signal* realisiert, während *Terminals* und *Quantities* für bidirektionale Verbindungen mit kontinuierlichen Größen genutzt werden.

Motivation: Die Motivation für dieses Refactoring ist einfach: Die I/O-Schnittstellen der Komponenten müssen geändert werden, um den Austausch zusätzlicher Informationen zu ermöglichen. Dieses Refactoring wird dementsprechend häufig und selbstverständlich durchgeführt. Allerdings sollte man sich vor der Anwendung überlegen, ob es wirklich notwendig ist. Umfangreiche I/O-Schnittstellen sind ungeschickt, da man sie sich schwer merken kann. Eventuell ist es sinnvoll, im Anschluss an die Erweiterung der I/O-Schnittstelle einige Informationskanäle zu einem Vektor zusammenzufassen.

Refactoring 22 (Remove Port from Entity Declaration):
Ist-Zustand: Ein Informationskanal der *Entity* Deklaration wird nicht benötigt.

Vorgehensweise: Der Informationskanal wird aus der *Entity* Deklaration entfernt und, falls notwendig, als internes *Signal*, *Terminal* oder *Quantity* deklariert.

Motivation: Eine Erweiterung von *Entity* Deklarationen wird bereitwilliger durchgeführt, als das Streichen von Informationskanälen. Meist verursacht ein zusätzlicher Port keine Probleme und man könnte ihn später wieder benötigen. Man kann überflüssige Ports jedoch auch aus einem anderen Blickwinkel bewerten: Eine Komponentenschnittstelle zeigt welche Informationen eine Komponente benötigt, bzw. zur Verfügung stellt. Wenn die Komponente in einem Systemmodell eingesetzt wird, muss man sich Überlegen, wie man die Informationskanäle verdrahtet. Jeder überflüssige Port erzeugt zusätzlichen Arbeitsaufwand. Da das Entfernen eines Ports aus der Komponentenschnittstelle sehr einfach ist, sollte man das bei der Entwicklung neuer Komponenten bereitwillig tun. Bei Komponenten, die schon in vielen Modellen eingesetzt sind, kann eine solche Änderung der Komponentenschnittstelle umfangreiche Nacharbeiten nach sich ziehen,

```
entity adc12 is
  port (
    terminal T_inP, T_inN : electrical;
    signal Clk, Rst : in std_logic;
    signal Dout : out std_logic_vector(11 downto 0));
end entity adc12;
-----------------------------------------------------------------------------
entity genericADC is
  generic (BitRes : natural := 12);
  port (
    terminal T_inP, T_inN : electrical;
    signal Clk, Rst : in std_logic;
    signal Dout : out std_logic_vector(BitRes-1 downto 0));
end entity genericADC;
```

Beschreibung A.16: Aus einem 12-Bit A/D Wandler wurde mit dem Parameter *BitRes* ein A/D Wandler mit wählbarer Auflösung erzeugt. Die Verhaltensbeschreibung muss ebenso angepasst werden. (Rfact. 23)

so dass eventuell ein *Extract Entity* sinnvoller ist, als die Änderung der etablierten Komponentenschnittstelle.

Refactoring 23 (Parameterize Entity):
Ist-Zustand: Es existieren einige Komponenten, deren Verhaltensmodelle sich auf eine gemeinsame Verhaltensbeschreibung zurückführen lassen. Die Komponentenmodelle sind jedoch unabhängig voneinander modelliert.

Vorgehensweise: Eines der Komponentenmodelle wird mit Parametern (Generics) so erweitert, das es bei geeigneter Wahl der Parameter das Verhalten der anderen Komponenten nachbildet.

Motivation: Wenn man mehrere Komponenten findet, die für unterschiedliche Datentypen eine ähnliche Berechnung durchführen, oder deren Verhalten sich nur in einigen Details unterscheidet, sollte man diese mit einem parameterisierten Modell ersetzen. Man kann so eine Menge duplizierten Code beseitigen und erhält zusätzlich eine leichter wiederverwendbare Modellkomponente, da man weitere Variationen durch eine Änderung der Parameterwerte realisieren kann.

Refactoring 24 (Replace Parameter with Explicit Architecture):
Ist-Zustand: In einem Komponentenmodell wird mit einem Parameter zwischen unterschiedlichen Verhaltensbeschreibungen umgeschaltet.

Vorgehensweise: Für jeden Wert des Parameters wird eine eigene Architektur definiert.

Motivation: Dieses Refactoring ist das Gegenstück zu *Parameterize Entity*. Eine parametrisierte Komponente ist sinnvoll, wenn dadurch duplizierter Code vermieden werden kann. Auf der anderen Seite wird durch jeden Parameter die Komponentenschnittstelle

```
entity Original_fpu1 is
  generic (
    manWidth : natural := 23;
    expWidth : natural := 8);
  port (
    Clk,Rst,En : in std_logic;
    arg1, arg2 : in  std_logic_vector(manWidth+expWidth downto 0);
    res        : out std_logic_vector(manWidth+expWidth downto 0));
end entity Original_fpu1;
entity Original_fpu_inv is
  generic (
    manWidth : natural := 23;
    expWidth : natural := 8);
  port (
    Clk,Rst,En : in std_logic;
    arg        : in  std_logic_vector(manWidth+expWidth downto 0);
    res        : out std_logic_vector(manWidth+expWidth downto 0));
end entity Original_fpu_inv;
--------------------------------------------------------------------------------
package public is
  use ieee.std_logic_1164.all;
  constant manWidth : natural := 23;
  constant expWidth : natural := 8;
  subtype  Tfloat_sign is std_logic;
  subtype  Tfloat_exp is std_logic_vector(expWidth-1 downto 0);
  subtype  Tfloat_man is std_logic_vector(manWidth-1 downto 0);
  subtype  Tfloat is std_logic_vector(expWidth+manWidth downto 0);
end package public;

use work.public.all;
entity refactored_fpu1 is
  port (
    Clk,Rst,En : in std_logic;
    arg1, arg2 : in  Tfloat;
    res        : out Tfloat);
end entity refactored_fpu1;

use work.public.all;
entity refactored_fpu_inv is
  port (
    Clk,Rst,En : in std_logic;
    arg        : in  Tfloat;
    res        : out Tfloat);
end entity refactored_fpu_inv;
```

Beschreibung A.17: Mit Hilfe des Datentypen Tfloat wird die Port Deklaration vereinfacht. (Rfact. 25)

komplexer. Wenn für unterschiedliche Werte des Parameters unabhängige Verhaltensbeschreibungen ausgeführt werden, ist es besser mehrere nicht parametrisierte Verhaltensmodelle zu schreiben, die über den *Architecture*-Namen unterschieden werden.

Refactoring 25 (Introduce Meta-Parameter):
Ist-Zustand: Mehrere Komponenten enthalten Parameter, die zusammengehören.

Vorgehensweise: Die Parameter werden zu einem Meta-Parameter zusammengefasst. Dieser Meta-Parameter ist eine Konstante in einem speziellen *Package*. Aus der Konstanten müssen sich die Werte der ursprünglichen Parameter berechnen lassen. Das

Package wird von allen Komponenten, die die ursprünglichen Parameter verwenden, geladen. An Stelle der Parameter in der *Entity* Deklaration werden interne Konstanten definiert, deren Werte mit Hilfe des Meta-Parameters berechnet werden.

Motivation: Oft sieht man *Entity* Deklarationen, die unterschiedlich benannte Parameter enthalten, die voneinander abhängig sind. Das können beispielsweise Parameter sein, welche die Größe von Eingangs- und Ausgangsvektoren festlegen. Betrachtet man ein Systemmodell, in dem mehrere dieser Komponentenmodelle verknüpft werden, so sind oft einige der Parameter der Komponenten aus Parametern der anderen Komponenten berechenbar. Dann können diese Parameter durch einen Meta-Parameter ersetzt werden. Durch konsequente Verwendung solcher Meta-Parameter werden die Komponenten-Deklarationen vereinfacht, was insbesondere bei stark partitionierten Systemmodellen die Lesbarkeit erleichtert.

Refactoring 26 (Replace Port with Parameter):
Ist-Zustand: Ein *Signal*, eine *Quantity* oder ein *Terminal* der Komponentendeklaration ist durch die externe Beschaltung fest vorgegeben.

Vorgehensweise: Die Deklaration der Größe wird durch einen Parameter und eine interne Konstante des passenden Typs ersetzt.

Motivation: Die Deklaration einer Größe als *Signal*, *Quantity* oder *Variable* legt nahe, dass die Größe geändert werden kann. Wenn eine Änderung nicht erwünscht ist, sollte die Größe als Konstante definiert werden. Damit wird klar, dass die Größe nicht geändert werden soll. Das hilft beim Verständnis der Beschreibung und ermöglicht bei der Simulation automatisierte Optimierungen.

Refactoring 27 (Change Unidirectional Datapath to Bidirectional):
Ist-Zustand: Eine synchrone Komponente A verarbeitet Daten einer Komponente B, welche die Daten jedoch nicht mit der Taktrate der Komponente A liefert. Komponente B sendet die Daten ohne Empfangskontrollen.

Vorgehensweise: Die Komponenten werden um Handshake-Signale erweitert, mit denen auf Empfangsbereitschaft und Sendebereitschaft getestet wird.

Motivation: Eine unkontrollierte, unidirektionale Datenübertragung ist zwischen zwei vollsynchronen Komponenten in einem ASIC eine effiziente und problemlose Form der Datenübertragung. Wenn die Komponenten jedoch unterschiedlichen Clock-Domänen angehören und dementsprechend zueinander asynchron laufen, sollten sie über bidirektionale Übertragungsprotokolle synchronisiert werden. In ausgewählten Fällen, z. B. wenn Komponente B viel langsamer getaktet wird als Komponente A, kann eine unidirektionale Datenübertragung funktionieren. Bei nachträglichen Änderungen im Taktverhältnis entstehen jedoch schwer zu findende Fehlerquellen. Ebenso bietet sich ein bidirektionales Handshake-Protokoll zur Konsistenzsicherung zwischen Komponenten, die Multi-Clock Datenpfade enthalten, an.

Refactoring 28 (Change Bidirectional Datapath to Unidirectional):
Ist-Zustand: Zwei Komponenten nutzen ein bidirektionales Handshake-Protokoll, obwohl die empfangende Komponente stets empfangsbereit ist.

Vorgehensweise: Das Signal, mit dem die empfangsbereite Komponente ihre Empfangsbereitschaft anzeigt, wird entfernt. Die sendende Komponente wird so modifiziert, dass die Daten ohne Aufforderung gesendet werden.

Motivation: Handshake-Protokolle sind sinnvoll, aber sie erzeugen zusätzliche Komplexität. Wenn der Aufwand zur Instandhaltung der bidirektionalen Datenpfade vermieden werden kann, ohne die Funktionsweise der Komponenten zu verschleiern, sollte man das tun.

A.4 Verschiebende Methoden

Ist-Zustand: (Für alle Refactorings:) Wenn eine notwendige Änderung nur durchgeführt werden kann, indem an vielen Stellen jeweils eine Änderung gemacht wird, dann ist es an der Zeit zuerst die betroffenen Codestellen 'zu sammeln'.

Refactoring 29 (Move Process):
Ist-Zustand: Eine Komponente enthält einen Prozess, dessen Verhalten entweder mit den sonstigen Vorgängen in der Komponente wenig zu tun hat, oder dessen Beschreibung in einer anderen Komponente sinnvoller erscheint.

Vorgehensweise: Der Prozess wird in eine andere, oder eine neue Komponente verschoben.

Motivation: Prozesse werden verschoben, wenn die Verhaltensbeschreibung einer Komponente zu umfangreich wird oder wenn die Bedeutung einer Komponente nicht klar erkennbar ist. Komponenten, die Prozesse enthalten, die in keinem Zusammenhang zueinander stehen, sind i. d. R. ein Zeichen für eine mangelhafte Partitionierung eines Systems. Durch das Verschieben der Prozesse kann die Partitionierung inkrementell verbessert werden.

Refactoring 30 (Inline Simple Statement, Function or Process):
Ist-Zustand: Die Verhaltensbeschreibung einer Zuweisung, einer Gleichung, einer Funktion oder eines Prozesses ist so trivial, dass dessen Existenz für das Verständnis des Gesamtmodells nicht hilfreich ist.

Vorgehensweise: Die Beschreibung wird entfernt und ihr Verhalten wird durch eine Erweiterung der anderen Beschreibungen nachgebildet.

Motivation: Die Verwendung von kleinen überschaubaren Blöcken ist ein wichtiger Bestandteil von Refactoring. Das Ziel dieser Strukturierung ist eine klare, einfach lesbare Beschreibung. Manchmal werden im Verlauf der Modellentwicklung ehemals komplexe Prozesse so vereinfacht, dass die mehrzeilige Prozessbeschreibung komplexer ist, als

ein nebenläufiges Ersatzmodell. In solchen Fällen sollte man den Prozess ersetzen. Das gilt natürlich gleichermaßen für Funktionen. Ebenso sollte man Prozesse ersetzen, deren Funktionalität ohne Einschränkung der Lesbarkeit in einen anderen Prozess integriert werden kann. Ein weiterer Grund für die Zusammenführung von Prozessen kann ein schlecht partitioniertes Komponentenmodell sein: Um das Modell neu zu strukturieren, werden zuerst die Prozesse in einem großen Prozess zusammengefasst.

Bei Signalzuweisungen und impliziten Gleichungen kann obige Methodik ebenfalls sinnvoll sein, wenn durch eine Zusammenfassung von Signalzuweisungen oder Gleichungen ein kompakteres Modell entsteht.

Dieses Refactoring wird auch eingesetzt, um eine Berechnungsvorschrift anders zu strukturieren. In diesem Fall wird zuerst der vorhandene Datenfluss zusammengefasst, um danach mit *Extract Signal/Quantity/Terminal* und *Introduce Explaining Variable* den Datenfluss neu zu unterteilen.

Refactoring 31 (Pull Up/Push Down Register):
Ist-Zustand: In einem hierarchischen Systemmodell werden Datensätze gespeichert, deren einzelne Einträge in verschiedenen Komponenten verarbeitet werden.

Motivation: Speicher für Datensätze kann in unterschiedlichen Formen realisiert werden. Für einen Speicher mit langen Datensätzen bieten sich RAM-Zellen an, die Platz für mehrere Datensätze zur Verfügung stellen und effizient programmiert werden können. Für in die Datenverarbeitung integrierte Speicher sind Flipflops geeignet, die jeweils nur ein Bit speichern. Bei der Modellierung von Systemen, deren Datensätze in verschiedenen Komponenten verarbeitet werden, ist es oft notwendig in der Konzeptionsphase eine Entscheidung über die Implementierung der Datenspeicher zu treffen. Zentrale Datenspeicher in Gestalt von RAM-Zellen können effizient und platzsparend in Hardware realisiert werden, während lokale Datenspeicher mit Flipflops deutlich mehr Platz beanspruchen. Dafür bieten lokale Datenspeicher meist kürzere Zugriffszeiten und ermöglichen eine Einheit von Daten und Datenverarbeitung und erleichtern damit das Verständnis für das Modell.
Diese Refactoring-Vorschrift dient dazu die Entscheidung für lokalen oder globalen Speicher auch in späteren Entwicklungsstadien zu hinterfragen, bzw. in einzelnen Teilbereichen zu überarbeiten: Um eine effiziente Modellentwicklung zu begünstigen, soll der für die Modellweiterentwicklung zuständige Ingenieur die Entscheidungsfreiheit besitzen, nach Bedarf Speicherstrukturen zu erzeugen, bzw. aufzulösen.

Vorgehensweise: (für *Push Up Register*) Wenn ein Datensatz in mehreren Komponenten lokal gespeichert wird, der Wert des Datensatzes jedoch nur von einer Komponente bestimmt wird, dann bietet sich *Push Up Register* an: An Stelle des lokalen Speichers wird die *Entity* Deklaration um einen Bitvektor passender Breite ergänzt, der an Stelle von Lesezugriffen auf den lokalen Speicher verwendet wird. Der Datenwert wird zentral gespeichert und die Bitvektoren werden mit dem Ausgangssignal des zentralen Speichers verknüpft. Die *Entity*, die den Wert des Datensatzes bestimmt, wird um ein Port-Interface zum Beschreiben des zentralen Speichers ergänzt.

Vorgehensweise: (für *Pull Down Register*) Wenn ein zentraler Datenspeicher Speicherelemente enthält, die nur von einer Komponente genutzt werden, kann an Stelle dieser Speicherelemente in der Komponente lokaler Speicher instantiiert werden. Bei komplexen Speicherstrukturen, kann eine mehrschrittige Umformung sinnvoll sein: Zuerst wird ein Modell beschrieben, das mit dem Original Speicher-Interface auf den lokalen Speicher zugreift. Nach der erfolgreichen Validierung dieses Modells wird das Speicher-Interface den lokalen Bedürfnissen entsprechend angepasst.

Refactoring 32 (Pull Up Function/Procedure):
Ist-Zustand: Eine von mehreren Komponenten genutzte Funktion ist mittels Copy-And-Paste in die Verhaltensbeschreibung der Komponenten eingebunden.

Ist-Zustand: Eine in einer Komponentenbeschreibung eingebettete Funktion soll in einer anderen Komponente eingesetzt werden.

Vorgehensweise: Die Funktionsbeschreibung wird in ein *Package* verschoben. In den Komponentenbeschreibungen steht die Funktion nach Einbinden den *Package* zur Verfügung, so dass die Funktionsdefinition in allen Komponenten entfernt werden kann.

Motivation: Es ist sinnvoll duplizierte Beschreibungen zu beseitigen. Obwohl zwei duplizierte Funktionen an sich problemlos funktionieren, stellen sie potentielle Fehlerquellen dar. Immer wenn Beschreibungen durch Copy-And-Paste wiederverwendet werden, riskiert man, dass eine Verbesserung oder Änderung der Funktionsbeschreibung nicht in allen Instanzen durchgeführt wird, da es in der Regel schwierig ist die Duplikate zu finden. Oft ist *Pull Up Function/Procedure* ein Refactoring, dass sich aus anderen Arbeitsschritten ergibt. Beispielsweise wenn man zwei Funktionen findet, die über Parameter so erweitert werden können, dass sie die selbe Funktionalität bekommen. In diesem Fall wird zuerst die Parameterisierung durchgeführt, anschließend werden die Funktionen in ein *Package* verschoben.

Refactoring 33 (Push Down Function/Procedure):
Ist-Zustand: Eine über ein *Package* deklarierte Funktion oder Prozedur wird nur von einer Komponente genutzt.

Vorgehensweise: Die Funktionsbeschreibung wird in die Präambel der Komponentenbeschreibung verschoben. Im *Package* wird zusätzlich die Funktionsdeklaration gelöscht.

Motivation: Dieses Refactoring ist die inverse Operation zu *Pull Up Function/Procedure*. Sie kann nützlich sein, wenn man eine Komponentenbeschreibung zur Verwendung in einem fremden Systemmodell zur Verfügung stellt, um eine möglichst kompakte Beschreibung zu erhalten. (Die Beschreibung des *Package* entfällt nach Extraktion der Funktion.)

Refactoring 34 (Hide Entity):
Ist-Zustand: Eine Komponente A wird nur gemeinsam mit einer bestimmten anderen Komponente B verwendet.

Vorgehensweise: Die Modellhierarchie wird so geändert, dass Komponente A eine Unterkomponente der anderen Komponente wird. Zusätzlich wird die Komponentendeklaration von Komponente A nur innerhalb der anderen Komponente deklariert, d. h. falls die Komponentendeklaration in einem *Package* enthalten ist, wird sie dort entfernt. Ein Zugriff auf *Ports* der Komponente A ist nun nur noch für Komponente B möglich.

Motivation: Wenn man Refactoring durchführt, müssen viele Entscheidungen bzgl. der hierarchischen Modellstruktur getroffen werden. *Reuse* von existierenden Modellen wird erst durch frei zur Verfügung stehende Komponentendeklarationen möglich. Andererseits können frei zu Verfügung stehende Modelle nicht mehr ohne weiteres geändert werden, da danach alle Modelle, die das Teilmodell verwenden, validiert werden müssen. Für die Modellpflege ist es deshalb sinnvoll, nur lokal verwendete Komponenten ebenso nur in dem dazugehörigen lokalen Kontext zu deklarieren.

Refactoring 35 (Inline Entity):
Ist-Zustand: Eine Komponente hat keine ausreichende Funktionalität.

Vorgehensweise: Die Funktionalität der Komponente wird in eine andere Komponente integriert. Anschließend wird die erweiterte Komponente im Systemmodell passend eingebunden und die überflüssige Komponente gelöscht.

Motivation: Eine Komponente sollte aus einem Systemmodell entfernt werden, wenn die Komponente durch andere Refactorings so in ihrer Funktionalität beschnitten wurde, dass sie für das Verständnis des Systemmodells nicht mehr benötigt wird.
Solche Komponenten können auch als Resultat einer Konzeptionsphase entstehen, wenn während der Konzeptionsphase ein System konzipiert wird, dass auf alle denkbaren Erweiterungen vorbereitet sein soll. Konzepte für eine solche eierlegende Woll-Milchsau sind i. d. R. sehr komplex und meist werden nicht alle Erweiterungen integriert. Mit den Hilfsmitteln des Refactoring ist eine nachträgliche Ergänzung eines Systemmodells einfach möglich, so dass man in der Konzeptionsphase nur das benötigte System konzipieren sollte. Zusätzliche Erweiterungen können in einem einfachen, gut strukturierten Modell leichter integriert werden. Komponenten, die als Platzhalter für Zusatzfunktionalität gedacht waren, sollten entfernt werden, wenn die Zusatzfunktionalität nicht implementiert wird.

Refactoring 36 (Extract Package):
Ist-Zustand: Ein *Package* enthält Elemente, die in einem eigenen *Package* sinnvoller gruppiert werden können. (Oder Elemente aus mehreren *Packages* sollten in einem gemeinsamen *Package* zusammengefasst werden.)

Vorgehensweise: Die Elemente werden in ein eigenes *Package* verschoben, danach wird das Package in die Komponenten, welche die Elemente nutzen, eingebunden.

Motivation: Eine Restrukturierung von *Packages* sollte durchgeführt werden, wenn die existierenden *Packages* zu umfangreich werden, wenn sie Elemente enthalten, die nur selten benötigt werden, oder wenn oft gemeinsam benötigte Elemente über mehrere *Packages* verteilt sind.

Refactoring 37 (Join Packages):
Ist-Zustand: Zwei *Packages* werden in Komponenten eines Systemmodells meist gemeinsam genutzt und enthalten artverwandte Elemente.

Vorgehensweise: Die Elemente der *Packages* werden in ein gemeinsames *Package* verschoben. Anschließend wird dieses *Package* an Stelle der vorherigen *Packages* in den Komponenten eingetragen.

Motivation: Wenn man einige Änderungen an der Modellhierarchie durchgeführt hat und einige neue *Packages* extrahiert hat, kann es passieren, dass einige *Packages* übrig bleiben, die lediglich Reste enthalten, die für sich genommen keine erkennbare Bedeutung haben. Solche *Packages* sollten beseitigt werden. Ein möglicher Lösungsansatz ist nach anderen *Packages* zu suchen, die stets gemeinsam mit den zu beseitigenden *Packages* benutzt werden, um diese zu verschmelzen.

Refactoring 38 (Extract Function/Procedure):
Ist-Zustand: Eine sequentielle Verhaltensbeschreibung enthält ein Stück Code das zusammengefasst werden kann.

Vorgehensweise: Für das Stück Code wird eine Funktion oder eine Prozedur definiert, die an Stelle des Codes in die sequentielle Verhaltensbeschreibung eingesetzt wird.

Motivation: Die Extraktion von Funktionen und Prozeduren ist ein wichtiger Arbeitsschritt, mit dessen Hilfe überlange sequentielle Verhaltensbeschreibungen vereinfacht werden. Kurze Funktionen mit prägnanten Namen bieten einige Vorteile gegenüber langen Beschreibungen, die mit Hilfe von Kommentaren erklärt werden müssen. Zum einen ist die Chance eine Funktion wiederverwenden zu können bei einer einfachen Funktion größer. Zum anderen ermöglicht es komplexe Verhaltensbeschreibungen, deren Beschreibungen selbsterklärend ist, indem anstatt kompliziertem Code lediglich einige Unterprozeduren aufgerufen werden, deren Name den Arbeitsschritt erklärt. Die Benamung der extrahierten Prozeduren ist bei dieser Arbeitsweise entscheidend für die Effizienz des Ansatzes. Letztendlich kann es sinnvoll sein, eine Prozedur zu extrahieren, deren Name länger ist als der extrahierte Code, wenn dadurch das Verhaltensmodell besser erklärt wird.

Refactoring 39 (Extract Process/Procedural):
Ist-Zustand: Eine nebenläufige Verhaltensbeschreibung enthält Funktionalität, die in einem nebenläufigen Kontext effektiver beschrieben werden kann.

Vorgehensweise: Für die Beschreibung wird ein Prozess definiert, der den Teil der nebenläufigen Beschreibung ersetzt.

Motivation: Die Extraktion von Prozessen kann beispielsweise dazu genutzt werden eine Erweiterung des Modells vorzubereiten. So werden einfache nebenläufige Beschreibungen zuerst mit Hilfe eines Prozesses nachgebildet, um danach die Möglichkeiten der Prozessbeschreibungen für die Weiterentwicklung des Modells zu nutzen.

```
architecture Original of MyCalc is
begin
  process (Clk, rst) is
    constant size : integer := Modul'length+1;
    variable
      cntI, incI, mod0,
      s1, s2, res : std_logic_vector(size-1 downto 0);
    variable Carry : std_logic;
  begin
    if rst = '1' then
      Cnt <= (others => '0');
    elsif Clk'event and Clk = '1' then
      cntI   := extend(cnt, size);
      incI   := extend(increment, size);
      mod0 := extend(Modul, size);
      if (OpCode = '1') then
        s1 := cntI + incI;
        s2 := s1 - mod0;
        Carry := not(s2(s2'high));
      else
        s1 := cntI - incI;
        s2 := s1 + mod0;
        Carry := s1(s1'high);
      end if;
      if (Carry = '0') then
        Cnt <= '0' & s1(s1'high-1 downto 0);
      else
        Cnt <= '1' & s2(s2'high-1 downto 0);
      end if;
    end if;
  end process;
end architecture Original;
--------------------------------------------------------------------------------
architecture Refactored of MyCalc is
  function addsub_mod (
    a0     : std_logic_vector;
    OpCode : std_logic;
    a1     : std_logic_vector;
    mod0   : std_logic_vector)
    return std_logic_vector is
    constant size : integer := mod0'length+1;
    variable
      a0i, a1i, mod0i,
      s1, s2, res : std_logic_vector(size-1 downto 0);
    variable Carry : std_logic;
  begin
    a0i   := extend(a0, size);
    a1i   := extend(a1, size);
    mod0i := extend(mod0, size);
    if (OpCode = '1') then
      s1 := a0i + a1i;
      s2 := s1 - mod0i;
      Carry := not(s2(s2'high));
    else
      s1 := a0i - a1i;
      s2 := s1 + mod0i;
      Carry := s1(s1'high);
```

```
    end if;
    if (Carry = '0') then
      res := '0' & s1(s1'high-1 downto 0);
    else
      res := '1' & s2(s2'high-1 downto 0);
    end if;
    return res;
  end function addsub_mod;
begin
  process (Clk, rst) is
  begin
    if rst = '1' then
      Cnt <= (others => '0');
    elsif Clk'event and Clk = '1' then
      Cnt <= addsub_mod(Cnt, OPCode, Increment, Modul);
    end if;
  end process;
end architecture Refactored;
```

Beschreibung A.1: Vereinfachung der Beschreibung durch Extraktion einer Funktion (Rfact. 38)

A.5 Konkretisierende Methoden

Refactoring 40 (Split Process):
Ist-Zustand: Ein Prozess enthält zwei von einander unabhängig durchführbare Berechnungsvorschriften.

Vorgehensweise: Beide Berechnungsvorschriften werden in eigene Prozesse verschoben.

Motivation: Ein Prozess sollte eine Verhaltensbeschreibung mit klar erkennbaren Aufgaben sein. Bei der Modellentwicklung wird nach und nach etwas Verhalten oder einige zusätzliche Signale ergänzt, um den Prozess an die aktuellen Bedürfnisse anzupassen. So entstehen überlange Prozesse, die eine Vielzahl von Signalen berechnen. In diesem Fall ist es sinnvoll die Berechnung einiger Signale in einen eigenen Prozess zu verlagern.

Refactoring 41 (Split Entity):
Ist-Zustand: Die untersuchte Komponente erledigt die Aufgaben von zwei Komponenten.

Vorgehensweise: Man erzeugt eine zusätzliche Komponente und verschiebt die relevanten Ports und Verhaltensbeschreibungen in die neue Komponente. Anschließend werden die Ports der neuen Komponente verknüpft. Eventuell müssen die beiden Komponenten einen neuen Namen erhalten.

Motivation: Es gibt mehrere Gründe eine Komponente durch zwei selbstständige Komponenten zu ersetzen. Der offensichtlichste Grund ist die Beseitigung einer komplexen

Komponente mit einer Vielzahl von *Ports*, deren Beschreibung lang und schwer verständlich ist.
Es kann jedoch auch sinnvoll sein, eine zusätzliche Komponente zu erzeugen, wenn die Komponente mehrfach eingesetzt wird und die Anforderungen an die Komponente aufgrund von unterschiedlichen Optimierungszielen zu hart werden. Man kann diese Situation daran erkennen, dass man in der Komponente mehrfach Änderung durchführen muss, die vorherige Änderungen ad absurdum führen.

Refactoring 42 (Insert Component):
Ist-Zustand: Die untersuchte Komponente A enthält Funktionalität, die durch eine andere Komponente (B) zur Verfügung gestellt wird.

Vorgehensweise: Die Komponente B wird in der Komponente A instantiiert und so eingebunden, das die ursprüngliche Funktionalität der Komponente A erhalten bleibt.

Motivation: Beim Entwurf heterogener Systeme werden meist abstrakte Modelle iterativ konkretisiert und modifiziert, um eine produzierbare Beschreibung zu erhalten. Eine solche Beschreibung ist erreicht, wenn das Systemmodell vollständig in produzierbare Komponenten partitioniert ist. Dieses Ziel lässt sich einfacher erreichen, wenn man eine Bauteil-Bibliothek zur Verfügung hat, deren Elemente automatisiert implementiert werden können. Um diese Komponenten in ein Systemmodell einsetzen zu können, muss das Systemmodell i. d. R. an die Komponentenschnittstelle angepasst werden. Dieser Arbeitsschritt ist ein wichtiger Bestandteil dieses Refactorings.

Refactoring 43 (Substitute Algorithm):
Ist-Zustand: Man will eine Verhaltensbeschreibung durch eine andere ersetzen, die das selbe Verhalten zeigt, deren Funktionsweise jedoch leichter nachvollziehbar ist.

Vorgehensweise: Man ersetzt den Kern des Verhaltensmodells durch die neue Verhaltensbeschreibung.

Motivation: Refactoring dient dazu komplizierte Beschreibungen in kleinere Teile zu zerlegen, aber manchmal kommt man an einen Punkt, an dem eine Beschreibung komplett entfernen werden muss, um sie durch eine geeignetere Beschreibung zu ersetzen. Manchmal muss ein vorhandenes Modell kalibriert werden, und man stellt fest, dass diese Änderung viel leichter durchzuführen ist, wenn zuerst die Modellbeschreibung durch eine andere ersetzt wird. Das Ersetzen von großen, komplexen Algorithmen ist sehr schwierig, deshalb sollte man, bevor dieser Schritt unternommen wird, sicherstellen, dass die zu ersetzende Modellbeschreibung so weit wie möglich eingegrenzt ist.

Refactoring 44 (Switch to another Model Architecture):
Ist-Zustand: Das Verhaltensmodell einer Komponente ist zu stark vereinfacht. Für die Komponente wird jedoch in einer zusätzlichen *Architecture* ein genaueres Verhaltensmodell zur Verfügung gestellt.

```
entity cap is
  generic (Capacity : real := 1.0e-12);
  port (terminal T1, T2 : electrical);
end entity cap;

architecture ideal of cap is
quantity VCap across ICap through T1 to T2;
begin
  ICap == VCap'dot * Capacity;
end architecture ideal;

architecture substrat_coupled of cap is
quantity VCap across ICap through T1 to T2;
quantity VSub across ISub through T2;
begin
  ICap == VCap'dot * Capacity;
  ISub == VSub'dot * 0.9*Capacity;
end architecture substrat_coupled;
-------------------------------------------------------------------------------
architecture Original of MyEntity is
begin
  cap_1: entity work.cap(ideal)
    generic map (Capacity => 0.8e-12)
    port map (T1 => T1, T2 => T2);
  ...;
end architecture Original;
-------------------------------------------------------------------------------
architecture Refactored of MyEntity is
begin
  cap_1: entity work.cap(substrat_coupled)
    generic map (Capacity => 0.8e-12)
    port map (T1 => T1, T2 => T2);
  ...;
end architecture Refactored;
```

Beschreibung A.18: In diesem Beispiel wird das Modell einer idealen Kapazität gegen eine Beschreibung eines Kondensators mit Substratkapazität ausgetauscht. Als Substratpotential wurde dabei das systeminterne Referenzpotential verwendet. (Rfact. 44)

Vorgehensweise: Für die Komponente wird das genauere Verhaltensmodell eingesetzt. Eventuell muss das neue Verhaltensmodell mit den Daten des alten Modells kalibriert werden.

Motivation: Nach einer konzeptionellen Validierung eines Systemmodells muss das Systemmodell auf Realisierbarkeit überprüft werden, dazu müssen in einigen Komponenten zusätzliche parasitäre Effekte berücksichtigt werden. VHDL-AMS ermöglicht es mit dem Konzept der *Architectures* mehrere Beschreibungen einer Komponente parallel zur Verfügung zu stellen, so dass je nach Validierungsanforderungen unterschiedlich genaue Modelle in der Simulation verwendet werden können. In den meisten Fällen ist eine Gesamtsystemvalidierung nur möglich, wenn für einen Teil der Komponenten vereinfachte Modelle zum Einsatz kommen. Aus diesem Grund wird das vereinfachte Modell der Komponente erhalten.

Refactoring 45 (Refine Simplified Model):
Ist-Zustand: Das Verhaltensmodell einer Komponente bildet das reale Verhalten einer Komponente zu ungenau nach.

Vorgehensweise: Die Komponentenbeschreibung wird um die nichtlinearen Effekte erweitert, bis die Genauigkeitsanforderungen erfüllt sind.

Motivation: Für die Validierung eines virtuellen Prototypen sind genaue Verhaltensmodelle notwendig, die neben der prinzipiellen Funktionalität einer Komponente auch deren nicht ideales Verhalten nachbilden. Wenn während der Implementierungsphase entschieden wird, welche realen Komponenten im System zum Einsatz kommen sollen, können die Verhaltensmodelle dementsprechend erweitert werden. An Stelle dieses Refactorings könnte auch ein *Insert Component* oder ein *Replace Simplified Model* angebracht sein. Die Erweiterung eines Verhaltensmodells ist dann sinnvoll, wenn das ursprüngliche Modell nicht mehr benötigt wird und die Wiederverwendung einer Komponentenbeschreibung mit Hilfe von *Insert Component* nicht möglich ist.

Refactoring 46 (Introduce Hierarchy):
Ist-Zustand: Eine Komponente A enthält sehr viele Komponenteninstantiierungen, davon bildet ein Teil eine zusammenhängende Gruppe.

Vorgehensweise: Man erzeugt eine zusätzliche Komponente und verschiebt die zusammenhängenden Komponenteninstantiierungen in die neue Komponente. Diese Komponente wird anstelle der Komponenteninstantiierungen in der Komponente A eingebunden.

Motivation: Durch die Erweiterung der Modellhierarchie wird die Strukturierung des Systemmodells verbessert. Aus der Komponente A wird eine kleinere funktionale Einheit herausgelöst, deren Implementierung und Test dementsprechend einfacher ist.

Refactoring 47 (Collapse Hierarchy):
Ist-Zustand: Eine Komponente A besteht aus wenigen Komponenteninstantiierungen, die wiederum nur aus wenigen Komponentenbeschreibungen bestehen.

Vorgehensweise: Die in der Beschreibung der Komponente A instantiierten Komponenten werden mit Hilfe von *Inline Entity* in die Komponente A aufgenommen.

Motivation: Die Modellhierarchie ist ein Hilfsmittel, das zur Strukturierung eines umfangreichen Systemmodells dient. Diese Strukturierung kann das Verständnis für ein System erleichtern, andererseits ist eine komplexe Hierarchiestruktur ebenfalls schwer zu verstehen. Im Verlauf des Entwurfsprozesses wird eine Erweiterung der Hierarchiestruktur i. d. R. bedenkenlos durchgeführt, ebenso sollte man in regelmäßigen Abständen die Hierarchiestruktur auf sinnvolle Vereinfachungen überprüfen.

Refactoring 48 (Refine Dataflow):
Ist-Zustand: Ein Komponentenmodell ist ein algorithmisches Verhaltensmodell, oder ein konservatives Verhaltensmodell ohne detailierte, explizite Beschreibung des internen Datenflusses.

Vorgehensweise: Der Datenfluss wird iterativ konkretisiert, indem zusätzliche Zwischenergebnisse als *Signale*, *Quantities* oder *Terminals* beschrieben werden.

Motivation: Während für eine Designstudie algorithmische Verhaltensmodelle mit idealisierten, effizient simulierbaren Verhaltensmodellen sinnvoll sind, braucht man in späteren Entwicklungsschritten Verhaltensmodelle, in denen man durch *Insert Component* ausgewählte Funktionalität durch parametrisierbare Komponenten ersetzen kann, deren Implementierung in Hardware automatisiert werden kann. Eine geeignete Vorgehensweise bei der Umformung der algorithmischen Verhaltensmodelle ist es, den Datenfluss einer anvisierten Implementierung nach zu bilden.

Refactoring 49 (Emerge Finite State Machine):
Ist-Zustand: Ein Komponentenmodell ist ein algorithmisches Verhaltensmodell, dessen Kontrollfluss nicht synthesefähig beschrieben ist.

Vorgehensweise: Herausbildung des Kontrollflusses durch Ersetzen impliziter sequentieller Teilmodelle mit zustandsbasierten synthetisierbaren endlichen Automaten.

Motivation: Diese Transformation dient zur Umformung ereignisdiskreter synchroner Komponenten in synthetisierbare Beschreibungen. Da die aktuellen Syntheseprogramme nur eine Untermenge der VHDL-Sprache verarbeiten können, müssen auch Komponenten, die nicht mit AMS-Erweiterungen beschrieben sind, für die Synthese aufbereitet werden. Dabei ist ein wichtiger Arbeitsschritt die Herausbildung des Kontrollflusses.

Refactoring 50 (Substitute Implementation):
Ist-Zustand: Der in einer Komponentenbeschreibung abgebildete Implementierungsansatz erfüllt vorgegebene Qualitätsmerkmale (z. B. Eingangsspannungsbereich, Rauschfärbung) nicht.

Vorgehensweise: Wie bei *Substitute Algorithm* wird die Komponentenbeschreibung durch ein Ersatzmodells mit äquivalentem Klemmenverhalten ersetzt. Das Ersatzmodell, dass den alternativen Implementierungsansatz beschreibt, wird durch Parallelsimulation mit dem Originalmodell kalibriert und validiert.

Motivation: Für die Gesamtsystemsimulation werden effiziente, einfache und leicht erstellbare Verhaltensmodelle benötigt, die für die Implementierungsphase umgeformt werden müssen. Bei dieser Umformung müssen typische Designeigenschaften wie Rauschen, eingeschränkte Spannungsbereiche und ähnliches berücksichtigt werden. Um die Spezifikation von Komponenten einzuhalten, kann dabei eine Umstellung von Implementierungsansätzen notwendig werden.

Anhang B

Beispielhafte Modellbeschreibungen

B.1 VHDL-Modell der Abstraktionsebene Behavioral Level

Als Beispiel eines digitalen Modells der Abstraktionsebene *Behavioral Level* wird ein Teil der FPU vorgestellt, die zum Teil aus studentischen Arbeiten entstanden ist. [56, 82, 57, 80, 63]

```
-------------------------------------------------------------------------------
-- Title       : Floating Point Unit
-- Project     : DP1
-------------------------------------------------------------------------------
-- File        : fpu_kernel_entity.vhd
-- Author      : Wolfram Stumpf, Tobias Kuckuck, Steffen Klupsch
-- Company     : TU Darmstadt
-- Last update : 2000/05/29
-- Platform    : Linux
-------------------------------------------------------------------------------
-- Description : Entity for floating point calculations.
--               This entity can do addition, substraction,
--               multiplication, division and 'power of 2'.
-------------------------------------------------------------------------------
-- entity documentation:
-- Input:
--    clock    : entity is triggered by rising_edge(clock)
--    reset    : synchronous reset
--    op       : FPU Operation (add, sub, mult, div)
--    A_*      : operand 1
--    B_*      : operand 2
--    start    : 1='A_in, B_in and op are set, start calculating Res_*'
-- Output:
--    Res_*    : result of the operation
--    ready    : 1= 'Result is valid'
-------------------------------------------------------------------------------
-- functionality:
-- op= fAdd:   data_out = a_in + b_in
-- op= fSub:   data_out = a_in - b_in
-- op= fMult:  data_out = a_in * b_in
-- op= fDiv:   data_out = a_in / b_in
-- op= fexp2:  dataout  = 2 ^ a_in
-- op= others: dataout = a_in
-------------------------------------------------------------------------------
```

```
library IEEE,FPUlib;
use IEEE.std_logic_1164.all;
use FPUlib.public.all;
use FPUlib.decl.all;

entity fpu_kernel is
  port (
    reset, clock, start : in  std_logic;
    op                  : in  FPU_Opcode;
    A_sign              : in  float_sign;
    A_man               : in  float_man1;
    A_exp               : in  float_exp1;
    B_sign              : in  float_sign;
    B_man               : in  float_man1;
    B_exp               : in  float_exp1;
    Res_sign            : out float_sign;
    Res_man             : out float_man2;
    Res_exp             : out float_exp2;
    ready               : out std_logic
    );
end fpu_kernel;
```

Beschreibung B.1: Entity Deklaration des Modells *fpu_kernel*

Diese FPU wird in dem in Kapitel 5 vorgestellten *v*DC eingesetzt. Die folgende VHDL-Beschreibung kann für beliebige Mantissen- und Exponentenlänge synthetisiert werden und beherrscht die folgenden Rechenoperationen: Addition, Subtraktion, Multiplikation, Division und Exponentation zur Basis 2. Diese Beschreibung basiert auf dem Sprachumfang des *Behavioral Compiler* (Version 2000.05) der Firma Synopsys und wurde auf der in Kapitel 4.5 vorgestellten FPGA-Karte validiert.

```
-------------------------------------------------------------------------------
-- Title       : Floating Point Unit
-- Project     : DP1
-------------------------------------------------------------------------------
-- File        : fpu_kernel_digital_algorithmic.vhd
-- Author      : Wolfram Stumpf, Tobias Kuckuck, Steffen Klupsch
-- Company     : TU Darmstadt
-- Last update : 2000/06/09
-- Platform    : Linux
-------------------------------------------------------------------------------
-- Description : Entity for floating point calculations.
--               This entity can do addition, substraction,
--               multiplication and division.
-------------------------------------------------------------------------------
-- entity documentation:
-- Input:
--    clock    : entity is triggered by rising_edge(clock)
--    reset    : synchronous reset
--    op       : FPU Operation (add, sub, mult, div)
--    A_*      : operand 1
--    B_*      : operand 2
--    start    : 1='A, B and op are set, start calculating Res'
-- Output:
```

```
--    Res_*    : result of the operation
--    ready    : 1= 'Res is valid'
-------------------------------------------------------------------------------
-- functionality:
-- op= fAdd:   Res = a + b
-- op= fSub:   Res = a - b
-- op= fMult:  Res = a * b
-- op= fDiv:   Res = a / b
-- op= fexp2:  Res = 2 ^ a
-- op= others: Res = a
-------------------------------------------------------------------------------


library IEEE,FPUlib,DW02;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use DW02.DW02_components.all;
use FPUlib.public.all;
use FPUlib.decl.all;
use FPUlib.float_const.all;
--synopsys synthesis_off
use ieee.math_real.all;
use FPUlib.fp.all;
--synopsys synthesis_on


architecture digital_algorithmic of fpu_kernel is

  subtype srt_bit_type is integer range -1 to 1;
  subtype f_manProd is unsigned((f_man1'high*2 +1) downto 0);
  subtype f_frt2    is unsigned((f_man1'high+f_exp1'high+1) downto 0);

  attribute dont_unroll : boolean;

begin
  main : process
     variable man1_A, man1_B : f_man1;
     variable exp1_A, exp1_B : f_exp1;
     variable sign_A, sign_B, sign_Res : float_sign;
     variable man2_Res : f_man2;
     variable exp2_Res : f_exp2;
     variable exp2_temp : f_exp2;
     variable man2_temp1,
              man2_temp2 : f_man2;
     variable man3_temp  : f_man3;
     variable man3_add1  : f_man3;
     variable man3_add2  : f_man3;
     variable man3_erg   : f_man3;
     variable man_prod   : f_manProd;
     variable frt2_temp  : f_frt2;


     variable srt_op  : srt_bit_type;
     variable srt_msb, srt_flag : std_logic;
     variable srt_R : signed(2 downto 0);
     variable ok : std_logic;

     attribute dont_unroll of EXP2_LOOP : label is TRUE;
```

```
    attribute dont_unroll of DIV_LOOP : label is TRUE;

  begin  -- process main
    reset_loop : loop
      man1_A := (others => '0');
      exp1_A := (others => '0');
      sign_A := fPlus;
      man1_B := (others => '0');
      exp1_B := (others => '0');
      sign_B := fPlus;
      man2_Res := (others => '0');
      exp2_Res := (others => '0');
      sign_Res := fPlus;
      man2_temp1 := (others => '0');
      man2_temp2 := (others => '0');
      man3_temp := (others => '0');
      man3_add1 := (others => '0');
      man3_add2 := (others => '0');
      man3_erg := (others => '0');
      exp2_temp := (others => '0');

      srt_op := 0;
      srt_msb := '0';
      srt_flag := '0';
      srt_r := (others => '0');

      ok := '0';
      Res_sign <= sign_Res;
      Res_man  <= std_logic_vector(man2_Res);
      Res_exp  <= std_logic_vector(exp2_Res);
      ready <= '0';

-- Wait for rolled 'reset_loop'
      wait until clock'event and clock = '1';
      if (reset = '1') then exit reset_loop; end if;

      work_loop : loop
        Res_sign <= sign_Res;
        Res_man  <= std_logic_vector(man2_Res);
        Res_exp  <= std_logic_vector(exp2_Res);
        wait until clock'event and clock = '1';
        if (reset = '1') then exit reset_loop; end if;
        ready <= '1';

-- Wait between signal assignment and next loop statement:
        wait until clock'event and clock = '1';
        if (reset = '1') then exit reset_loop; end if;

        calc : loop
        exit calc when start /= '1';
        ready <= '0';


        -- OK. We have something to do, let's go:
        man1_A := unsigned(A_man);
        exp1_A := signed(A_exp);
        sign_A := A_sign;
        man1_B := unsigned(B_man);
```

```
        exp1_B := signed(B_exp);
        sign_B := B_sign;

-- Wait for rolled 'calc_loop'
         wait until clock'event and clock = '1';
         if (reset = '1') then exit reset_loop; end if;

        case op is      -- Let's start the calculations:
-------------------------------------------------------------------
          when fADD | fSUB =>
-------------------------------------------------------------------
            -- fSub is transformed to fAdd:
            if op = fSUB then
              sign_B := not(sign_B);
            end if;

            -- A = 0 ?
            if (man1_A(man1_A'high) = '0') then
              man2_Res := (man1_B & '0');
              exp2_Res := (exp1_B(exp1_B'high) & exp1_B);
              sign_Res := sign_B;
              exit calc;
            -- B = 0 ?
            elsif (man1_B(man1_B'high) = '0') then
              man2_Res := (man1_A & '0');
              exp2_Res := (exp1_A(exp1_A'high) & exp1_A);
              sign_Res := sign_A;
              exit calc;
            end if;


            -- A and B need to be expressed with the same exponent:
            exp2_temp := (exp1_A(exp1_A'high) & exp1_A)
                         - (exp1_B(exp1_B'high) & exp1_B);
            -- increasing the number of significant bits
            -- (for correct rounding after addition)
            man2_temp1 := (man1_A & '0');
            man2_temp2 := (man1_B & '0');
            if (exp2_temp > 0) then                -- exp1_A > exp1_B
              exp2_Res := (exp1_A(exp1_A'high) & exp1_A);
              man2_temp2 := SHR(man2_temp2,unsigned(exp2_temp));
            else
              exp2_Res := (exp1_B(exp1_B'high) & exp1_B);
              if (exp2_temp < 0) then              -- exp1_A < exp1_B
                exp2_temp := -exp2_temp;
                man2_temp1 := SHR(man2_temp1,unsigned(exp2_temp));
              end if;
            end if;

            -- adding the mantisse
            if (sign_A = sign_B) then

              man3_add1 := ('0' & man2_temp1);
              man3_add2 := ('0' & man2_temp2);
              man3_erg := man3_add1 + man3_add2;
              sign_Res:= sign_A;

              -- normalization of the result: (increasing the exponent)
```

```
              if (man3_erg(man3_erg'high) = '1') then
                exp2_Res := exp2_Res + 1;
                man2_Res := man3_erg(man3_erg'high downto 1);
              else
                man2_Res := man3_erg(man3_erg'high-1 downto 0);
              end if;

-- wait to compensate the wait in the norm_loop
              wait until clock'event and clock = '1';
              if (reset = '1') then exit reset_loop; end if;

            else
              -- subtracting the mantisse
              if (man2_temp1 >= man2_temp2) then
                man3_add1 := ('0' & man2_temp1);
                man3_add2 := ('0' & man2_temp2);
                man3_erg := man3_add1 - man3_add2;
                man2_Res := man3_erg(man2_res'high downto 0);
                sign_Res := sign_A;
              else
                man3_add1 := ('0' & man2_temp1);
                man3_add2 := ('0' & man2_temp2);
                man3_erg := man3_add2 - man3_add1;
                man2_Res := man3_erg(man2_res'high downto 0);
                sign_Res := sign_B;
              end if;

              if man2_Res /= 0 then
                ok := '0';
                NORM_LOOP: while (ok='0') loop
-- Wait for rolled loop:
                  wait until clock'event and clock = '1';
                  if (reset = '1') then exit reset_loop; end if;

                  if (man2_Res(man2_Res'high)='0') then
                    man2_Res := shl(man2_Res,"1");
                    exp2_Res := exp2_Res - 1;
                  else
                    ok := '1';
                  end if;
                end loop NORM_LOOP;
              else
                -- Ooops, we got the result 0.0:
                man2_Res := man_fZero;
                exp2_Res := exp_fZero;

-- wait to compensate the wait in the norm_loop
                wait until clock'event and clock = '1';
                if (reset = '1') then exit reset_loop; end if;
              end if;
            end if;

-------------------------------------------------------------------
          when fMULT =>
-------------------------------------------------------------------
            sign_Res := sign_A xor sign_B;

            -- A or B = 0 ?
```

```
            if (man1_A(man1_A'high) = '0')
              or (man1_B(man1_B'high) = '0') then
              man2_Res := man_fZero;
              exp2_Res := exp_fZero;
              exit calc;
            end if;

            exp2_Res := (exp1_A(exp1_A'high) & exp1_A)
                          + (exp1_B(exp1_B'high) & exp1_B);

            -- multiplication (using arbitrary stages):
            man_prod := DWF_mult_ns(man1_A, man1_B, clock);
--            man_prod := man1_A * man1_B;

            -- normalization of the result: (increasing the exponent)
            if (man_prod(man_prod'high) = '1') then
              exp2_Res := exp2_Res + 1;
              man2_Res := man_prod(man_prod'high downto
                                   (man_prod'high-man2_Res'high));
            else
              man2_Res := man_prod(man_prod'high-1 downto
                                   (man_prod'high-1-man2_Res'high));
            end if;

-- Wait to compensate the waits in the other when-cases:
            wait until clock'event and clock = '1';
            if (reset ='1') then exit reset_loop; end if;
-------------------------------------------------------------------
          when fDIV | fEXP2 =>
-------------------------------------------------------------------
            if op = fEXP2 then
              -- this algorithm works only if 0 < 2^(A_in) < infty
              -- (range check is needed in the wrapper)
              sign_Res := fPlus;

              frt2_temp(frt2_temp'high
                        downto man1_A'length) := (others => '0');
              frt2_temp(man1_A'high downto 0) := man1_A;

              if (exp1_A <= conv_signed(-man1_A'length,exp1_A'length)) then
                man2_Res := man_fOne;
                exp2_Res := exp_fOne;
                exit calc;
              else
                if exp1_A > 0 then
                  frt2_temp := shl(frt2_temp,unsigned(exp1_A));
                elsif exp1_A < 0 then
                  exp1_A := - exp1_A;
                  frt2_temp := shr(frt2_temp,unsigned(exp1_A));
                end if;
              end if;

              exp1_B := signed(frt2_temp(f_exp1'high+man1_A'high
                                         downto man1_A'high));

              if (frt2_temp(man1_A'high-1) = '1') then
                man1_B := ('1' & unsigned(frt2_man(1)));
              else
```

```
                man1_B := ('1' & unsigned(fOne_man));
              end if;


              EXP2_LOOP: for i in f_man1'high-2 downto 1 loop
-- Wait for the rolled loop:
                wait until clock'event and clock = '1';
                if (reset ='1') then exit reset_loop; end if;

                if (frt2_temp(i) = '1') then
                  man1_A := ('1' & unsigned(frt2_man(float_man'length - i)));
                  -- multiplication (using arbitrary stages):
                  man_prod := DWF_mult_ns(man1_A, man1_B, clock);
--                  man_prod := man1_A * man1_B;

                  -- normalization of the result: (increasing the exponent)
                  -- including rounding towards infty
                  if (man_prod(man_prod'high) = '1') then
                    exp1_B := exp1_B + 1;
                    man1_B := man_prod(man_prod'high downto
                                  (man_prod'high-man1_B'high))
                              + man_prod(man_prod'high-man1_B'high-1);
                  else
                    man1_B := man_prod(man_prod'high-1 downto
                                  (man_prod'high-1-man1_B'high))
                              + man_prod(man_prod'high-man1_B'high-2);
                  end if;
                end if;
              end loop EXP2_LOOP;

              if sign_A = fPlus then
                man2_Res := (man1_B & '0');
                exp2_Res := ('0' & exp1_B);
              else
                man1_A := ('1' & unsigned(fOne_man));
                exp1_A := (others => '0');
              end if;
            else  -- op = fDIV
---fDIV-------------------------------------------------------------
-- Wait to compensate for 'EXP2_LOOP' in the if part:
               wait until clock'event and clock = '1';
               if (reset = '1') then exit reset_loop; end if;

              sign_Res := sign_A xor sign_B;

              if (man1_A(man1_A'high) = '0') then
                man2_Res := man_fZero;
                exp2_Res := exp_fZero;
                exit calc;
              elsif (man1_B(man1_B'high) = '0') then
                -- Oops, B=0 --> result = infty
                man2_Res := man_fInfty;
                exp2_Res := exp_fInfty;
                exit calc;
              end if;
            end if;

            if (op=fDiv) or ((op=fExp2) and (sign_A=fMinus)) then
```

```
            -- fDiv is done using the SRT-algorithm
            srt_op := +1;
            srt_msb  := '0';
            srt_flag := '0';
            man3_temp := (others => '0');
            man3_erg := ('0' & '0' & man1_A);
            man3_add1 := ('0' & '0' & man1_B);
            DIV_LOOP: for i in man3_temp'high downto 0 loop
-- Wait for the rolled loop:
              wait until clock'event and clock = '1';
              if (reset ='1') then exit reset_loop; end if;

              man3_temp := shl(man3_temp,"1");


              case srt_op is
                when -1 => man3_erg := man3_erg + man3_add1;
                           man3_temp := man3_temp - 1;
                when +1 => man3_erg := man3_erg - man3_add1;
                           man3_temp := man3_temp + 1;
                when others => null;
              end case;
              srt_R := signed(man3_erg(man3_erg'high-1
                                       downto man3_erg'high-3));
              if (srt_R = -4) or (srt_flag='1' and srt_msb='0') then
                srt_flag := '1';
                srt_op := -1;
              else
                srt_flag := '0';
                if (srt_R >= 0) then
                  srt_op := 1;
                elsif (srt_R = -1) then
                  srt_op := 0;
                else
                  srt_op := -1;
                end if;
              end if;
              srt_msb := srt_R(srt_R'high);
              man3_erg := shl(man3_erg,"1");
            end loop DIV_LOOP;  -- i

            exp2_Res := (exp1_A(exp1_A'high) & exp1_A)
                        - (exp1_B(exp1_B'high) & exp1_B);

            -- normalization of the result:
            if (man3_temp(man3_temp'high) = '1') then
              man2_Res := man3_temp(man3_temp'high downto 1);
            else
              exp2_Res := exp2_Res - 1;
              man2_Res := man3_temp(man3_temp'high-1 downto 0);
            end if;
          else
-- Wait to compensate for 'DIV_LOOP' in the if part:
            wait until clock'event and clock = '1';
            if (reset = '1') then exit reset_loop; end if;
          end if;

-----------------------------------------------------------------------
```

```
-- NOP
-----------------------------------------------------------------------
          when others =>
-- Wait to compensate the waits in the other when-cases:
            wait until clock'event and clock = '1';
            if (reset ='1') then exit reset_loop; end if;
            exp2_Res := (exp1_A(exp1_A'high) & exp1_A);
            man2_Res := (man1_A & '0');
        end case;
      end loop calc;
    end loop work_loop;
  end loop reset_loop;
end process main;
end digital_algorithmic;
```

Beschreibung B.2: *Behavioral Level* Beschreibung des *fpu_kernel*

Literaturverzeichnis

[1] AMSYS GMBH & CO. KG: *Standard and Low Pressure Transducer AMS5300 and AMS5400*. Datasheet Rev 2.1. An der Fahrt 13, 55124 Mainz, 2001

[2] ARMSTRONG, James R.: *Chip-Level Modeling with VHDL*. Englewood Cliffs : Prentice-Hall, 1989

[3] BAKER, Eric C.: *Clearing Up The Confusion About "Deep Stops"*. – URL: `http://www.gap-software.com/decotheory.html` (10. Mai 2001)

[4] BECK, Kent: Make it Run, Make it Right: Design Through Refactoring. In: *The Smalltalk Report* Vol. 6(4), SIGS Publications, Januar 1997, S. 19–24

[5] BENNETT, Peter (Hrsg.) ; ELLIOT, David (Hrsg.): *The Physiology and Medicine of Diving*. 4. Edition. London : W. B. Saunders Company Ltd., 1993

[6] BHATTI, Nadeem: *VHDL-AMS Routinen zur Generierung von Stimulidaten*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, interner Bericht, Februar 2002. – Betreuer: Steffen Klupsch

[7] BÜHLMANN, Albert A.: *Tauchmedizin*. 4. Auflage. Berlin : Springer-Verlag, 1995

[8] BMBF VERBUNDPROJEKT DEMIS: *Designoptimierung für Mikrosysteme*. – URL: `http://www.iai.fzk.de/demis` (8. Apr. 2002)

[9] BMBF VERBUNDPROJEKT MIMOSYS: *Modellbildung für die Mikrosystemtechnik*. – URL: `http://www.pb.izm.fhg.de/mimosys` (10. Nov. 2003)

[10] BMBF VERBUNDPROJEKT SIMKOS: *Simulation komplexer Systeme unter Einbeziehung intelligenter Komponenten*. – URL: `http://www.eas.iis.fhg.de/sim/projects/simkos` (8. Apr. 2002)

[11] BORLAND SOFTWARE CORP.: *Borland Delphi 6*. 100 Enterprise Way, Scotts Valley, CA, 2001. – URL: `http://www.borland.com/delphi/` (10. Jan. 2002)

[12] BORLAND SOFTWARE CORP.: *Kylix 2*. 100 Enterprise Way, Scotts Valley, CA, 2001. – URL: `http://www.borland.com/kylix/` (10. Jan. 2002)

[13] BOSER, Bernhard E. ; WOOLEY, Bruce A.: The Design of Sigma-Delta Modulation Analog-to-Digital Converters. In: *IEEE Journal of Solid-State Circuits* Vol. 23, Issue 6, IEEE, Dezember 1988, S. 1298–1308

[14] BOYCOTT, A. E. ; DAMANT, G. C. C. ; HALDANE, J. S.: The Prevention of Compressed Air Illness. In: *Journal of Hygiene* Vol. 8, 1908, S. 342–443

[15] BRONSTEJN, Il'ja N. ; SEMENDJAJEW, Konstantin A.: *Taschenbuch der Mathematik*. 25. Auflage. Leipzig : B. G. Teubner Verlagsgesellschaft, 1991

[16] CLELAND, A. N. ; ROUKES, M. L.: Fabrication of High Frequency Nanometer Scale Mechanical Resonators from Bulk Si Crystals. In: *Appl. Phys. Lett.* Vol. 69, 1996, S. 2653–2655

[17] EECS DEPARTMENT OF THE UNIVERSITY OF CALIFORNIA AT BERKELEY: *The Spice Home Page*. – URL: `http://infopad.eecs.berkeley.edu/icdesign/SPICE/` (11. Okt. 2002)

[18] ERNST, Markus ; HENHAPL, Birgit ; KLUPSCH, Steffen ; HUSS, Sorin A.: FPGA based Hardware Acceleration for Elliptic Curve Public Key Cryptosystems. In: *Journal of Systems and Software*, Elsevier Science, accepted to be published

[19] ERNST, Markus ; KLUPSCH, Steffen ; HAUCK, Oliver ; HUSS, Sorin A.: Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems. In: *12th IEEE International Workshop on Rapid System Prototyping*. Monterey, CA : IEEE, Juni 2001

[20] FOWLER, Martin: *Refactoring: Improving the Design of Existing Code*. Reading : Addison Wesley Longman Inc., 1999

[21] FRIEBE, Alexander: *Entwicklung von Strategien zum Entwurf digitaler Blöcke für Mixed-Signal Schaltungen und ihre Anwendung im Power BCD Prozeß*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, Studienarbeit, September 2002. – Betreuer: Steffen Klupsch; Jürgen Schulz (iC-Haus GmbH)

[22] GAJSKI, Daniel u. a.: *High-Level Synthesis: Introduction to Chip and System Design*. London : Kluwer Academic Publishers, 1992

[23] GAJSKI, Daniel u. a.: *Specification and Design of Embedded Systems*. London : Prentice-Hall, 1994

[24] GIBSON, Dennis ; CARTER, Hal ; PURDY, Carla: The Use of Hardware Description Languages in the Development of Microelectromechanical Systems. In: *Analog Integrated Circuits and Signal Processing* Vol. 28. London : Kluwer Academic Publishers, 2001, S. 173–180

[25] GRANGES, M. D.: Standard Air Decompression Table. Washington D.C. : US Navy Naval Gun Factory, 1956. – EDU Report 5-57

[26] GRÜNER, Karsten: *Modellierung und Simulation einer mixed-signal Schaltung am Beispiel eines Batterieladegerätes*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, Studienarbeit, September 1999. – Betreuer: Steffen Klupsch

[27] HAHN, Max: DECO 2000, die neue Dekompressionstabelle. In: *Der Sporttaucher, das offizielle Organ des Verbandes Deutscher Sporttaucher e.V.* Ausgabe 5/2000. Lübeck : Max Schmidt-Römhild KG, 2000

[28] HARAEUS SENSOR-NITE GMBH: *Platin-Temperatursensor in Dünnschichttechnik*. Datasheet. Reinhard-Heraeus Ring 23, 63801 Kleinostheim, 2001

[29] HELLER, R. ; MAGER, W. ; SCHROTTER, H. von: *Luftdruckerkrankungen mit besonderer Berücksichtigung der sogenannte Caissonkrankheit*. Wien : Alfred Holder, 1900

[30] HOARE, C. A. R.: Communicating Sequential Processes. In: *Communications of the ACM* Vol. 21, Issue 8, August 1978

[31] HOFACKER, K. u. a.: Entwurf integrierter optoelektronischer Sensoren. In: *GMM Workshop Methoden und Werkzeuge zum Entwurf von Mikrosystemen*. Berlin : GMM, Dezember 1999

[32] HONEYWELL, SENSING AND CONTROL: *Micromachined Silicon Pressure Sensors*. Datasheet. 11 West Spring Street, Freeport, Illinois 61032, 2001

[33] HUSS, Sorin A.: *Model Engineering in Mixed-Signal Circuit Design*. London : Kluwer Academic Publishers, 2001

[34] HUSS, Sorin A. ; KLUPSCH, Steffen: A New Approach to Mixed-signal Model Refinement by Code Refactoring Methods. In: *Forum on Specification and Design Languages 2003 (FDL'03)*. Frankfurt : European Chips and Systems Iniative (ECSI), September 2003

[35] HUSS, Sorin A. ; KLUPSCH, Steffen ; ROSENBERGER, Ralf: Modellierung gemischt analog/digitaler Schaltungen mit VHDL-AMS. In: *8. GMM-Workshop, Methoden und Werkzeuge zum Entwurf von Mikrosystemen*. Berlin, Dezember 1999

[36] HUSS, Sorin A. ; KLUPSCH, Steffen: Modellbildung und Systemsimulation am Beispiel eines Tiefenmessers. In: *GI-ASIM Fachtagung 'Simulation technischer Systems'*. Aachen, 1999

[37] IEEE-SA STANDARDS BOARD (Hrsg.): *IEEE Standard for Binary Floating Point Arithmetic*. IEEE Standard 754-1985. New York : IEEE, 1985

[38] IEEE-SA STANDARDS BOARD (Hrsg.): *IEEE Standard 1076.6: Standard for VHDL Register Transfer Level Synthesis*. IEEE Standard 1076.6-1999. New York : IEEE, März 1999

[39] IEEE-SA STANDARDS BOARD (Hrsg.): *IEEE Standard VHDL Analog and Mixed-Signal Extensions*. IEEE Standard 1076.1-1999. New York : IEEE, März 1999

[40] JERRAYA, A. A. u. a.: *Behavioral Synthesis and Component Reuse with VHDL*. London : Kluwer Academic Publishers, 1997

[41] KASPER, M.: *Mikrosystementwurf*. Heidelberg : Springer-Verlag, 1999

[42] KLEIN, A. ; GERLACH, G.: Simulation gekoppelter physikalischer Phänomene am Beispiel der Fluid-Struktur-Wechselwirkung einer Mikropumpe. In: *GMM–ITG–GI Workshop Multi-Nature Systems: Optoelektronische, mechatronische und sonstige gemischte Systeme*. Jena : GMM, ITG, GI, Februar 1999, S. 103–112

[43] KLUPSCH, Steffen ; HUSS, Sorin A.: Methodischer Entwurf und Simulation eines Delta-Sigma A/D Wandlers mit VHDL-AMS. In: *ASIM 2002: 16. Symposium Simulationstechnik*. Rostock : SCS European Publishing House, September 2002

[44] KLUPSCH, Steffen ; ERNST, Markus ; HUSS, Sorin A. ; RUMPF, Martin ; STRZODKA, Robert: Real Time Image Processing based on Reconfigurable Hardware Acceleration. In: *IEEE Workshop Heterogeneous Reconfigurable Systems on Chip (SoC): Chances, Applications, Trends*. Hamburg : IEEE, April 2002

[45] KLUPSCH, Steffen ; HUSS, Sorin A.: Implementierung und Validierung eingebetteter Systeme für Multi-Nature Einsatzumgebungen mit virtuellen Techniken. In: *thema FORSCHUNG, Eingebettete Systeme: Entwurf und Anwendung versteckter Computer* Ausgabe 1/2002. Darmstadt : Technische Universität Darmstadt, März 2002, S. 14–20

[46] KLUPSCH, Steffen: Design, Integration and Validation of Heterogeneous Systems. In: *2. IEEE International Symposium on Quality Electronic Design (ISQED 2001)*. San Jose (CA) : IEEE, März 2001

[47] KLUPSCH, Steffen ; HUSS, Sorin A.: Abstraktionsebenen und ihre Anwendung bei der Modellierung und Simulation von heterogenen Systemen. In: *3. GMM-ITG-GI Workshop Multi-Nature Systems: Optoelektronische, mechatronische und sonstige gemischte Systeme*. Hamburg : Verlag TUHH Technologie GmbH, Februar 2001

[48] KLUPSCH, Steffen: Verhaltensorientierte Modellierung mit VHDL-AMS. In: *Elektronik, Fachzeitschrift für industrielle Anwender und Entwickler* Ausgabe 11/2000, November 2000

[49] KLUPSCH, Steffen ; HUSS, Sorin A.: Optimierung der Modellbildung unter Berücksichtigung unterschiedlicher Entwurfskriterien. In: *GMM-ITG-GI Workshop Multi-Nature Systems: Optoelektronische, mechatronische und sonstige gemischte Systeme*. Jena : GMM-ITG-GI, 1999, S. 63–72

[50] KLUPSCH, Steffen ; ERNST, Markus ; BACHMANN, Werner ; HUSS, Sorin A.: Multimedia-System im Praktikum VLSI-Systementwurf. In: *GI-Tagung Informatik und Ausbildung*. Stuttgart, März 1999

[51] KLUPSCH, Steffen ; HUSS, Sorin A.: Modellierung nichtlinearer Systeme unter Berücksichtigung inhärenter Simulatoreigenschaften. In: *2. Workshop Automatisierungstechnische Verfahren für die Medizin*. Darmstadt, Februar 1999, S. 34–35

[52] KLUPSCH, Steffen: *VHDL-AMS in der Gesamtsystemsimulation*. April 2000. – 9. NF-Kolloqium der Fachhochschule Nürnberg

[53] KLUPSCH, Steffen ; ERNST, Markus: *FPGA basierte Hardwareimplementierung von Bildverarbeitungsalgorithmen*. November 2000. – Kolloqium der Angewandten Mathematik, Universität Bonn

[54] KLUPSCH, Steffen ; HUSS, Sorin A.: *Using VHDL-AMS for System Simulation*. Juli 1999. – Mentor Graphics VHDL-AMS Users Meeting

[55] KLUPSCH, Steffen ; HUSS, Sorin A.: *Verhaltensorientierte Modellierung zur Validierung von Mixed-Signal Schaltungen*. November 1999. – VHDL-AMS Seminar

[56] KUCKUCK, Tobias ; STUMPF, Wolfram: *Konzeption und Simulation eines Dekompressionsprozessors*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, Systementwurf-Praktikum, August 1999. – Betreuer: Markus Ernst; Steffen Klupsch

[57] KUCKUCK, Tobias ; STUMPF, Wolfram: *Portierung des Dekompressionsprozessors DP1 auf eine FPGA-Architektur*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, Studienarbeit, Juli 2000. – Betreuer: Markus Ernst; Steffen Klupsch

[58] KUNKEL, R. ; HERRMANN, St.: Modelle von PIN-Photodioden auf Basis experimenteller Daten für Netzwerksimulatoren. In: *GMM Workshop Methoden und Werkzeuge zum Entwurf von Mikrosystemen*. Berlin : GMM, Dezember 1999

[59] LEE, Edward A. u. a.: *Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java*. Memorandum UCB/ERL M99/40. Berkeley, Juni 2000. – URL: `http://ptolemy.eecs.berkeley.edu` (20. Feb. 200)

[60] LEE, Edward A. ; SANGIOVANNI-VINCENTELLI, Alberto: A Framework for Comparing Models of Computation. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* Vol. 17, Issue 12. New York : IEEE, Dezember 1998, S. 1217–1229

[61] LEITIS, K.: Einsatz von Verhaltensmodellen magnetoresistiver Sensoren in der Systemsimulation. In: *GMM Workshop Methoden und Werkzeuge zum Entwurf von Mikrosystemen*. Berlin : GMM, Dezember 1999

[62] LINDQUIST, Claude S.: Code Noise in Delta-Sigma Modulators. In: *Signals, Systems & Computers* Vol. 2. Pacific Grove, CA : IEEE Signal Processing Society, November 1998, S. 1012–1016

[63] MAJER, Mateusz: *Entwurf und Evaluierung von FPU-Grundalgorithmen mit dem Synopsys Module Compiler*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, Studienarbeit, September 2002. – Betreuer: Steffen Klupsch

[64] MARWEDEL, P.: *Synthese und Simulation von VLSI-Systemen: Algorithmen für den rechnergestützten Entwurf hochintegrierter Schaltungen*. München : Hanser Studienbücher, 1993

[65] MAXIM INTEGRATED PRODUCTS: Neue ICs revolutionieren die Sensor-Schnittstelle. In: *Maxim Engineeringjournal* (1998), S. 3–8

[66] MENTOR GRAPHICS CORP.: *Accusim II HDL-A/DEV User's and Reference Manual*. Wilsonville, 1994. – URL: `http://www.mentor.com`

[67] MENTOR GRAPHICS CORP.: *ADvance MS Users Guide*. Wilsonville, 2002. – URL: `http://www.mentor.com`

[68] MENTOR GRAPHICS CORP.: *Eldo User's Manual*. Wilsonville, 2002. – URL: `http://www.mentor.com`

[69] MICROSOFT CORP.: *Microsoft Visual Basic.NET Std. 2002*, 2002. – URL: `http://msdn.microsoft.com/vbasic/`

[70] MILNER, R.: *Communication an Concurrency*. Englewood Cliffs : Prentice-Hall, 1989

[71] MODEL TECHNOLOGY, A MENTOR GRAPHICS CORP. COMPANY: *Release Notes for ModelSim SE 5.8*. Wilsonville, 2003. – URL: `http://www.model.com/support/release_notes/58/RELEASE_NOTES_58.pdf` (25. Nov. 2003)

[72] MOTOROLA INC.: *Motorola Pressure Sensor Macromodel Library*, 1996. – URL: `http://mot2.mot-sps.com/models/bin/sensor2.html` (10. Jun. 1998)

[73] NORSWORTHY, Steven R. (Hrsg.) ; SCHREIER, Richard (Hrsg.) ; TEMES, Gabor C. (Hrsg.): *Delta-Sigma Data Converters: Theory, Design and Simulation*. New York : IEEE, 1997

[74] NÜSSEN, O. ; BECHTOLD, St. ; LAUR, R.: Modellierung eines kapazitive Drucksensors mit Touchdown-Effekt. In: *GMM Workshop Methoden und Werkzeuge zum Entwurf von Mikrosystemen*. Berlin : GMM, Dezember 1999

[75] OPPENHEIM, Alan V. ; SCHAFER, Ronald W.: *Zeitdiskrete Signalverarbeitung*. München : R. Oldenbourg Verlag, 1992

[76] RAYNAL, Michel ; SINGHAL, Mukesh: Logical Time: Capturing Causality in Distributed Systems. In: *IEEE Computer*. New York : IEEE, Februar 1996, S. 49–56

[77] RUDD, Robert E.: The Atomic Limit of Finite Element Modeling in MEMS: Coupling of Length Scales. In: *Analog Integrated Circuits and Signal Processing* Vol. 29, Number 1. London : Kluwer Academic Publishers, 2001, S. 17–26

[78] SCHACHT, R. ; KASPER, M.: Entwurf eines Makromodells zur transienten Simulation von gekoppelten thermisch-elektrischen Problemen am Beispiel eines IGBT. In: *GMM–ITG–GI Workshop Multi-Nature Systems: Optoelektronische, mechatronische und sonstige gemischte Systeme*. Jena : GMM, ITG, GI, Februar 1999, S. 93–101

[79] SCHRÖDER, Kai ; REITH, Steffen: *Sättigungsvorgänge beim Tauchen, das Modell ZH-L16, Funktionsweise von Tauchcomputern*. – URL: `http://www.streit.cc/dive/saett/saett_faq.html` (17. Jan. 03)

[80] SEIFFERT, Björn ; GUALENI, Markus: *vDC: virtual Diving Computer, Systementwurf-Praktikum 2001*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, Systementwurf-Praktikum, November 2001. – Betreuer: Steffen Klupsch

[81] SENSYM INC.: *SCC Series Pressure Sensors*. Datasheet. Sensor Technics, Aubinger Weg 27, 82178 Puchheim, 1998. – URL: `http://content.honeywell.com/sensing/hss/pressure/basic.asp` (8. Aug. 2003)

[82] STUMPF, Wolfram ; KUCKUCK, Tobias: *Modellierung, Synthese und Validierung einer anwendungsspezifischen Fließkommaarithmetik-Einheit für den Dekompressionsrechner DP1*, FG Integrierte Schaltungen und Systeme, Technische Universität Darmstadt, CAE-Praktikum, Dezember 1999. – Betreuer: Markus Ernst; Steffen Klupsch

[83] SYNOPSYS INC.: *Scirocco Reference Manual*. 700 East Middlefield Rd., Mountain View, CA 94043, USA, 1999

[84] SYNOPSYS INC.: *MAST - Analog, Mixed-Technology and Mixed-Signal HDL for Saber*. 700 East Middlefield Rd., Mountain View, CA 94043, USA, 2003. – URL: `http://www.synopsys.com/products/mixedsignal/saber/mast_ds.html` (25. Nov. 2003)

[85] TEICH, J.: *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Heidelberg : Springer-Verlag, 1997

[86] THOMAS, D.E. u. a.: *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. London : Kluwer Academic Publishers, 1990

[87] UWATEC SWITZERLAND: *Aladin Air*. Gebrauchsanleitung. Scubapro/Uwatec Deutschland Tauchausrüstungen GmbH, Rheinvogtstr. 17, 79713 Bad Säckingen-Wallbach, 1996

[88] VA SOFTWARE CORPORATION AND OPEN SYSTEMC INITIAVIVE: *SystemC v2.0.1 Language Reference Manual*. – URL: `http://www.systemc.org/download.php/systemc/13/48/SystemC_2_0_LRM_v1_0.pdf` (25. Nov. 03)

[89] VACHOUX, A. (Hrsg.) ; BERGÉ, J.-M. (Hrsg.) ; LEVIA, O. (Hrsg.) ; ROUILLARD, J. (Hrsg.): *Analog and mixed-signal hardware description languages*. London : Kluwer Academic Publishers, 1997

[90] VACHOUX, Alain ; GRIMM, Christoph ; EINWICH, Karsten: SystemC-AMS Requirements, Design Objectives and Rationale. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*. München : IEEE, März 2003, S. 10388–10393

[91] VDI/VDE-IT: *MST Infobörse Nr. 1-1995: Modellbibliothek für komplexe analoge Bauelemente*. 1996. – URL: `http://www.mstonline.de/publikationen/infoboerse/ib_95_96/95-1/ib-95-1.html` (25. Nov. 2003)

[92] WORKMAN, R. D.: Calculation of Decompression Schedules for Nitrogen-Oxygen and Helium-Oxygen Dives / US Navy Experimental Diving Unit. Washington D.C., 1965. – Research Report 6-65

[93] X-FAB SEMICONDUCTOR FOUNDRIES AG: *1.0 µm CMOS Process with Integrated Pressure Sensor*. Datasheet, Januar 2002. – URL: `http://www.xfab.com/sheets/is-pressure.pdf` (27. Feb. 2002)

[94] YARBROUGH, O. D.: Calculation of Decompression Tables / US Navy Experimental Diving Unit. Washington D.C., 1937. – Research Report

[95] ZEIGLER, Bernhard P. ; PRAEHOFER, Herbert ; KIM, Tag G.: *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. 2. Edition. London : Kluwer Academic Publishers, 2000