Entwicklungsautomatisierung von Regelungssoftware mittels Softwaregenerator

Diplomarbeit aus dem Gebiet der Elektronik

vorgelegt von
Ivo Boblan
Wassermannstrasse 90
12489 Berlin
Matrikel Nr.: 132796
Fachbereich Elektrotechnik

Betreuender Assistent IPK: Dipl.-Inf. T. Fries
Betreuender Assistent IPK: Dipl.-Ing. D. Surdilovic
Betreuender Hochschullehrer TU: Prof. Dr.-Ing. D. Naunin

Berlin, Oktober 1996

Inhaltsverzeichnis

1 Einführung

1.1 Einleitung

Die Komplexität großer Steuerungssysteme wächst kontinuierlich an. Um den Entwicklungsprozeß solcher Systeme überschaubarer zu machen, teilt man sie in kleine Teilmodule. Diese Teilaufgaben kann man dann in separaten Entwicklungsumgebungen einfacher und effizienter lösen. Solche speziellen Entwicklungsumgebungen bieten teilspezifische Techniken und Ausdrucksmöglichkeiten, um die jeweilige Aufgabe in genügendem Maße zu lösen. Alle Teillösungen müssen dann zu einem Gesamtsystem integriert werden. Die unabhängig voneinander entwickelten Teillösungen arbeiten durch bereitgestellte Kommunikationsmechanismen zusammen.

Durch automatische Softwaregenerierung vereinfacht sich der Arbeitsschritt der Integration zu einem Gesamtsystem erheblich. Unabhängig vom eigentlichen Einsatzbereich definieren adäquate Entwicklungsumgebungen die einzelnen Teilprobleme. Generatoren übersetzen die Lösungen anschließend in einen vom Einsatzbereich abhängigen Zielcode und generieren zusätzlich Schnittstellen. Über diese Schnittstellen können dann die einzelnen Module in das Gesamtsystem integriert werden.

Die vorliegende Arbeit befaßt sich mit dem Entwicklungsprozeß von der Regelkreisbeschreibung bis zum C++ Quelltextprogramm. Es soll die Teilaufgabe der Entwicklung von Reglersoftware durch den Einsatz eines Generators vereinfacht werden. Das Ziel ist es, eine vollständige und spezifikationskonforme Reglersoftware zu entwickeln, um damit die Umsetzung regelspezifischer Aspekte in lauffähige Programme zu erleichtern. Die Arbeit stellt den Entwurf und die Implementierung eines Übersetzers für die Beschreibungssprache CLDL vor. CLDL ist eine Beschreibungssprache für Regelkreise, die zu diesem Zweck entwickelt wurde. Weitere Entwicklungskriterien sind die Einfachheit der Regelkreisbeschreibung, die leichte Erweiterbarkeit des Übersetzers und die Erstellung eines effektiven C++ Codes.

Dieser Übersetzer vereinfacht den Prozeß der Steuerungsentwicklung mittels automatisch generierter Reglersoftwaremodule. Er ermöglicht dem Entwickler, sich auf die eigentliche Problemlösung zu konzentrieren. Die direkte Implementierung einer Reglersoftware steht somit nicht im Mittelpunkt seiner Entwicklung. Er soll sich vielmehr auf den Entwurf des Reglers und die spätere Validierung der Steuerung konzentrieren können.

Der Teil der Berechnung und Implementierung von Reglersoftware einer Steuerung ist zeitintensiv und fehleranfällig. Um aber trotzdem effektiv entwickeln zu können, setzt man Codegeneratoren zur Erstellung einer korrekten Software ein. Der verwendete Compilergenerator Gentle ermöglicht die effiziente Implementierung des Übersetzers.
Am Fraunhofer-Institut für Produktionsanlagen und Konstruktionstechnik (IPK) wird derzeit eine experimentelle Robotersteuerung mit dem Namen Sokrates entwickelt. Diese Diplomarbeit ist durch ihre generierte Reglersoftware in das Sokrates-Projekt involviert.

1.2 Steuerungsentwicklungsprozeß

Der Entwicklungsprozeß einer Steuerung besteht aus einzelnen Phasen. Diese stehen in einer mehr oder weniger engen Beziehung zu einander. Um eine hohe Qualität und Effektivität eines Steuerungsentwicklungsprozesses zu erreichen, sind eine Menge von verschiedenen Entwicklungsumgebungen notwendig.
Nachfolgend ist die Entwicklung eines Reglermoduls im groben dargestellt. Abbildung 1.1 zeigt den Ablauf der Entwicklungsschritte unter Berücksichtigung gegebener Anforderungen. Es können Anforderungen gestellt werden an

- die Komplexität der zu berücksichtigten Eigenschaften bei der Modellbildung der zu regelnden Strecke (z.B. Reibungsmodell),
- das Verhalten des geschlossenen Regelkreises (z.B. Lagefehler),
- das Verhalten ausschließlich der Reglersoftware im Zusammenspiel mit der Strecke in einer extra Testumgebung (Unterschied zwischen Theorie und Praxis) und
- das Verhalten der Reglersoftware im Zusammenspiel mit der kompletten Steuerung (Realzeitfähigkeit).

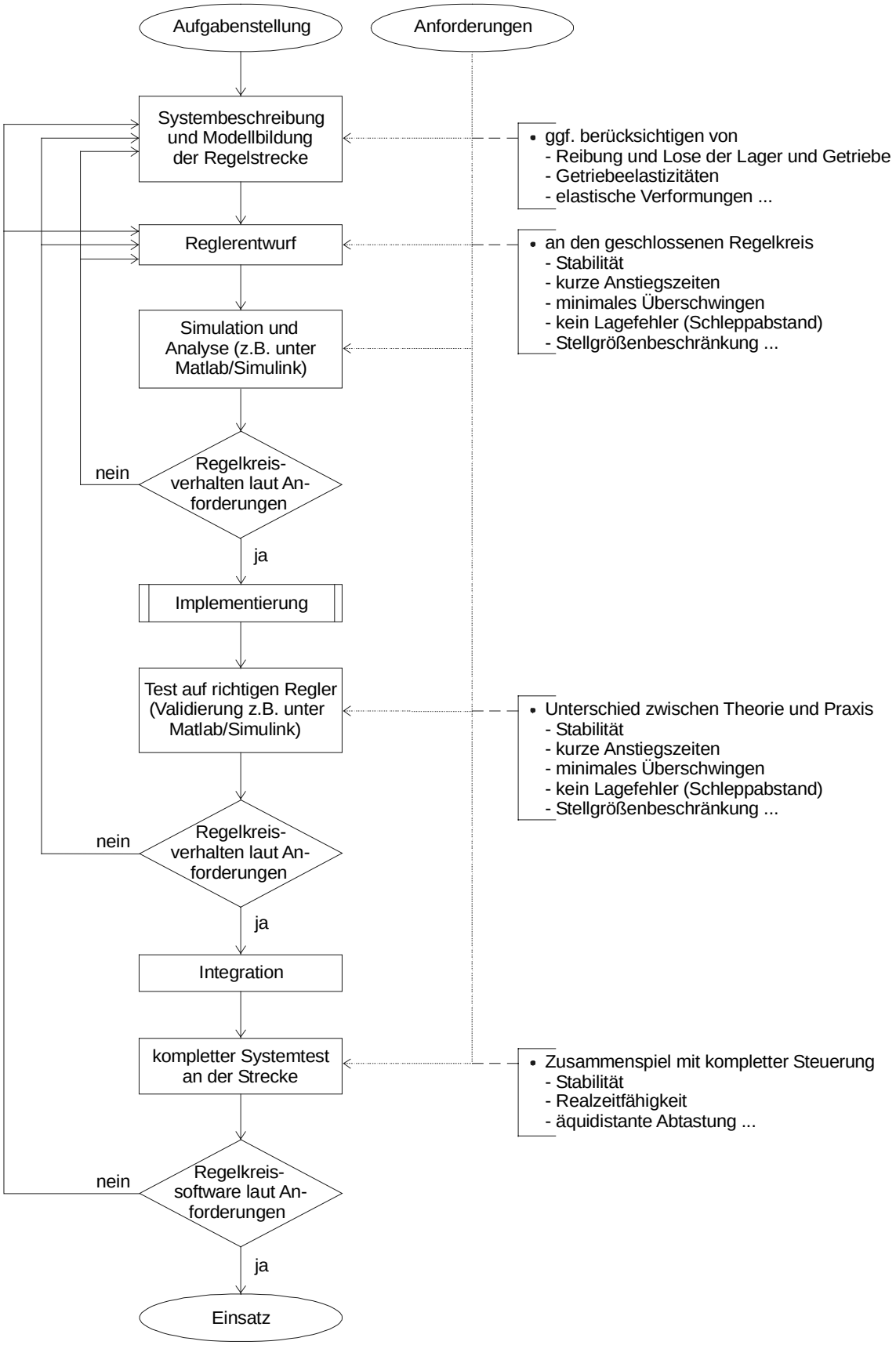


Abb. 1.1: Prozeßablaufplan zur Reglermodulentwicklung

Der dargestellte Entwicklungsablaufplan beinhaltet einen Unterablauf mit dem Namen *Implementierung*. Diesem Unterablauf kommt eine besondere Bedeutung zu, da er sich in einen konventionellen und einen Entwicklungsablauf mit Generierung unterteilen läßt.
Die konventionelle Entwicklungsvariante stellt den Prozeß der Implementierung von der Berechnung der Regelkreisgleichung bis zur programmiertechnischen Umsetzung in Software dar. Diese Art der Entwicklung eines Reglermoduls ist fehleranfällig, kosten- und zeitintensiv. Es steht nun dieser Methode eine neuere und effektivere Möglichkeit der Reglerentwicklung gegenüber. Ein generierter Übersetzers übernimmt die Umsetzung von der Reglerbeschreibung nach C++. Einzelne Phasen der Übersetzung können bei Umstrukturierungen oder Erweiterungen des Generators weiter verwendet werden. Weiterhin liefert ein Softwaregenerator einen Zielcode in beliebiger Anzahl und mit immer der selben einmal entwickelten Qualität. Dadurch wird die Wiederverwendbarkeit und Zuverlässigkeit von Softwarekomponenten wesentlich erhöht. Dieser Übersetzer steht im Mittelpunkt der Arbeit. Die Abbildung 1.2 zeigt den Unterschied beider Reglermodulimplementierungsvarianten.

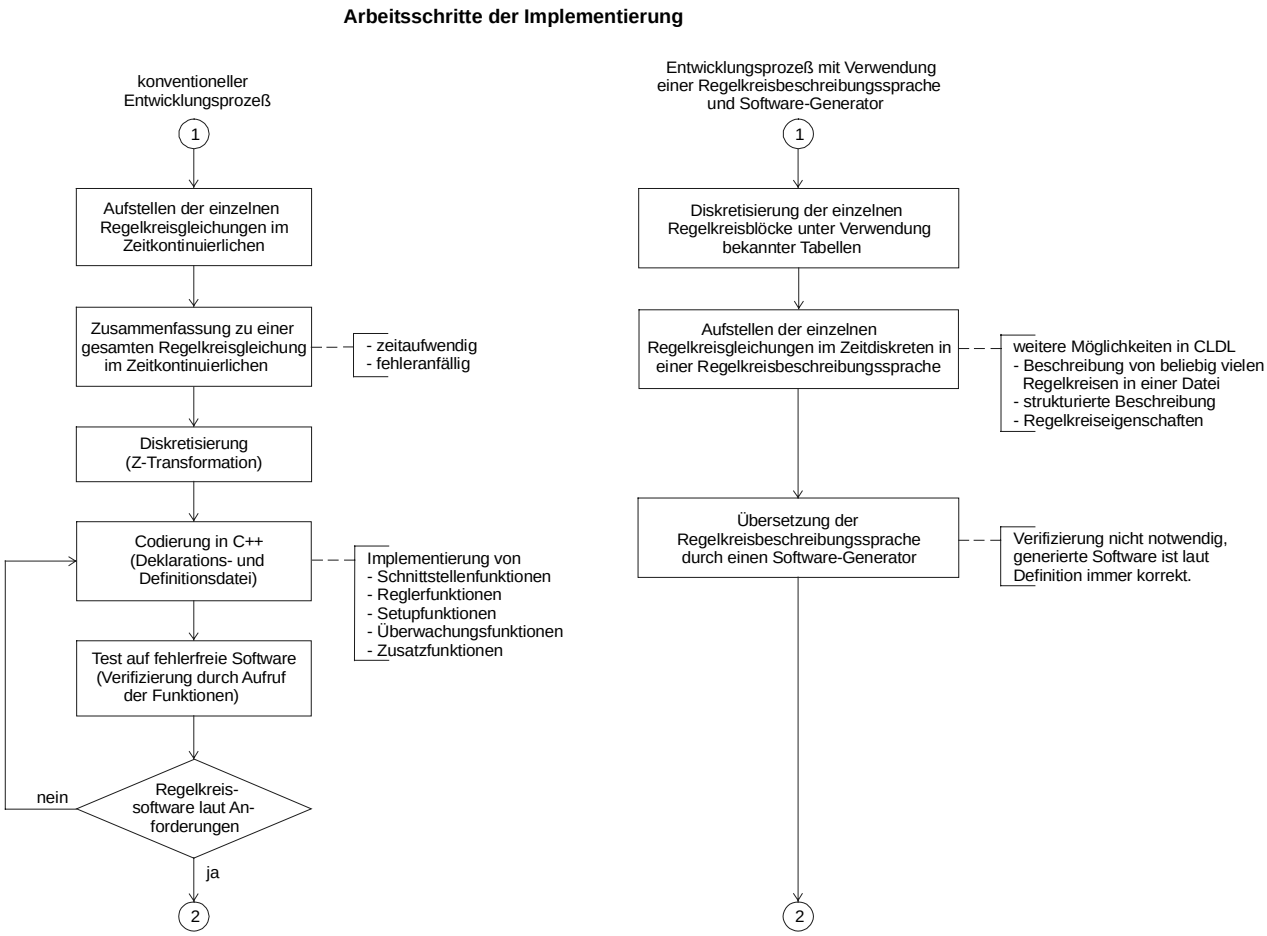


Abb. 1.2: Arbeitsschritte der Implementierung

2 Vergleich bestehender Regelkreisbeschreibungssprachen und Simulationsumgebungen

2.1 Regelkreisbeschreibungssprachen

Heutige Automatisierungssysteme sind durch eine hierarchische Struktur gekennzeichnet. Sie sind in mehrere Kompetenzfelder unterteilt, um klare Schnittstellen zu Systemkomponenten zu schaffen. Schnittstellen zielen auf eine strikte Aufgabenteilung einzelner Module ab. Ein wesentliches Entwicklungsziel der Regelungs- und Informationstechnik ist die Erhöhung der Autonomie derartiger Automatisierungseinrichtungen. Damit rückt die Modellbildung als Voraussetzung zur Lösung von Prozeßführung, Diagnose und Überwachung in den Blickpunkt. Kenntnisse über Ursache-Wirkungs-Zusammenhänge etwa in Form von physikalischen Gesetzen einerseits sowie die Auswertung von Beobachtungen und Erfahrungen (Messen von Signalen) andererseits ermöglichen es, Modelle aufzubauen. Die analytische Modellbildung basiert auf der Verfügbarkeit eines mathematischen Modells des technischen Prozesses bzw. des dynamischen Systems. Dieses Modell wird etwa in parametrischer Form durch Differentialgleichungen, d.h. mittels einer formalisierten Sprache, beschrieben.
Eine Beschreibungssprache im allgemeinen verfolgt das Ziel Zusammenhänge aus Natur, Gesellschaft oder Technik in einfacher Art und Weise formal zu beschreiben. Sie ist durch die Forderung nach Flexibilität und einfacher Bedienbarkeit geprägt. Die Sprachelemente sollten möglichst intuitiv verwendet werden können. Weiterhin sollte sie sich an bereits vorhandene Sprachen anlehnen, damit sie leichter erlernt werden kann und einen breiten Einsatz in Industrie und Wirtschaft findet.
Heute sind für fast jedes Problem und jeden Computer (vom PC bis zum Großrechner) mehrere universelle Simulationssprachen verfügbar. Der Entwickler kann für seine Zwecke die geeignetste auswählen.
Regelkreisbeschreibungssprachen beschreiben die Kopplung der Modellzustandsgrößen in einem Programm. Der Aufbau solcher Programme kann block-, gleichungs- oder modulorientiert sein. Bestehende Sprachen zur Regelkreisbeschreibung unterscheiden sich in

- ihrer formalen Syntaxstruktur,
- ihrem Beschreibungsumfang,

- ihren Schnittstellen zu Simulationsumgebungen,
- ihren Mechanismen, mit denen fremde Funktionen eingebunden werden können und
- ihren Beschreibungsformen.

Systeme können in verschiedenen Beschreibungsformen dargestellt werden. Man beschreibt Systeme

- im zeitkontinuierlichen Bereich,
- im zeitdiskteten Bereich,
- im Frequenzbereich,
- durch stochastische oder
- nichtlineare Beschreibungsformen oder
- mit Hilfe partieller Differentialgleichungen.

Regelkreisbeschreibungssprachen sollen den weniger erfahrenen Anwender bei der Beschreibung einer Regelkreisstruktur wirksam unterstützen, und der routinierte Anwender wird von immer wiederkehrenden Programmieraufgaben befreit.
Viele Regelkreisbeschreibungssprachen sind heute eng mit Simulationssystemen verbunden. Durch meist graphische Eingabe der Modellstrukturen wird die Beschreibung stark vereinfacht. Diese Programmpakete mit regelungstechnischem Schwerpunkt verwenden zur Erzeugung bzw. Überprüfung ihrer Ergebnisse einen simulationstechnischen Programmteil. Meist sind diese Simulationskomponenten jedoch auf die Bedürfnisse der einzelnen Systeme zugeschnitten und auf spezielle Strukturen und Eingangsgrößen beschränkt.
Ein weiterer wichtiger Aspekt ist die Kommunikation zwischen Prozeß, Simulator und Benutzer. Dieses Zusammenspiel bildet den Kern in höchst anspruchsvollen Simulationseinrichtungen, wie Trainingssimulatoren für Flugzeugpiloten, Schulungssimulatoren für Prozeßbediener und großen Entwicklungssimulatoren im Bereich der Kraftfahrzeugindustrie. Somit ist es wichtig, Schnittstellen zur Prozeßinteraktion bei zukünftigen Simulationssprachen wie Systemen zur Verfügung zu stellen.

2.2 Vorstellung einiger Simulationsbeschreibungssprachen

Im folgenden werden einige Beschreibungssprachen vorgestellt. Ein kompletter Überblick kann nicht gegeben werden, da z.Z. an die 90 Softwareprodukte existieren [2]. Einen aktuellen

Stand über Simulationsprodukte findet man im Internet (World Wide Web) und in verschiedenen Publikationen [1][2][3][4][5].

2.2.1 ACSL

ACSL (Advanced Continuous Simulation Language) ist eine weitverbreitete Simulationssprache zur Simulation und Analyse hauptsächlich kontinuierlicher Systeme, die dem CSSL Standard (Continuous System Simulation Language) 1986 für kontinuierliche Simulationssprachen genügt. ACSL steht seit 1975 auf unterschiedlichen Plattformen zur Verfügung. Dadurch erlangt ACSL einen hohen Verbreitungsgrad. Weitere Gründe liegen in der Standardisierung und einer hohen Softwarestabilität.
ACSL ist eine kompilierende, gleichungsorientierte Simulationssprache. Die Modelle können in Form von Systemen von Differentialgleichungen beschrieben und in ACSL Syntax formuliert werden. Diese Beschreibung wird dann von einem Compiler verarbeitet. Ein Runtime-Interpreter erlaubt ein interaktives Arbeiten mit dem übersetzten Modell.
Das nun folgende Beispiel zeigt, wie einfach ein Modell in ACSL beschrieben werden kann. Eine der wesentlichen Eigenschaften von ACSL ist das automatische Sortieren der Modellbeschreibung. Dies ermöglicht ein schnelles und modulares Entwerfen.

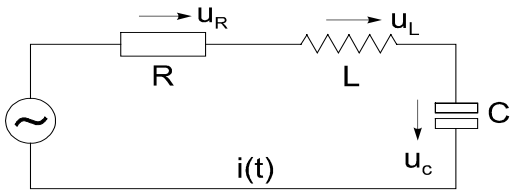


Abb. 2.1: Elektrischer Schwingkreis

Die Abbildung 2.1 zeigt einen elektrischen Schwingkreis, aus dem sofort folgende Beziehungen abgeleitet werden können.

$$
\begin{aligned}
u_L &= u_0 - u_C - u_R \\
\frac{di}{dt} &= \frac{u_L}{L} \\
u_R &= iR \\
\frac{du_C}{dt} &= \frac{i}{C}
\end{aligned}
\qquad (2.1)
$$

Diese Beziehungen kann man direkt in eine ACSL Modellbeschreibung umsetzen.

```
PROGRAM  SCHWINGKREIS
      INITIAL
            CONSTANT  L  =  1.5E-6 ,  R  =  1,  C  =  50E-12 ,  US0  =  10 ,  PI  =  3.1215
            CONSTANT  TEND = 8.E-6 ,  CINT = 1E-8
            F  =  1 / (2 * PI * SQRT( L * C ) )
            OM  =  2 * PI * F
      END
      DYNAMIC
            DERIVATIVE
                  U0  =  US0 * COS( OM * T )
                  UL  =  U0 - UC - UR
                  I  =  INTEG( UL /L , 0 )
                  UR  =  R * I
                  UC  =  INTEG( I / C , 0 )
            END
            TERMT( T , GT , TEND )
      END
END                                                                    (2.2)
```

Das Schlüsselwort "*SECTION*" unterteilt den Aufbau der Beschreibung. Alle echt dynamischen Zusammenhänge sind in der "*DERIVATIVE SECTION*" beschrieben; "*DYNAMIC SECTION*" beinhaltet nur die Endbedingung für die Simulation, in impliziter Form allerdings die gesamte Ablaufsteuerung für einen Simulationslauf; in der "*INITIAL SECTION* " können Konstanten festgelegt und statische Zusammenhänge beschrieben werden. Das wichtigste ACSL Statement zur Beschreibung der Dynamik ist das "*INTEG*"-Statement, das sozusagen die Integration zu einer Grundrechenart macht. "*TERMT*" stellt die Abbruchbedingung dar.
ACSL unterstützt weiterhin die Modellbildung mit regelungstechnischen Übertragungsfunktionen. Diese selbst sind allerdings als Systemmakros implementiert; ein Precompiler wandelt

sie in Differentialgleichungen um. Für diskrete Regler steht die "*DISKRETE SECTION*" zur Verfügung [1][6][7].

2.2.2 Dymola

Dymola (Dynamic Modelling Language) ist eine objektorientierte Sprache. Das dazugehörende gleichnamige Programmpaket ermöglicht das Modellieren von großen dynamischen Systemen. Eine Spezifikation, geschrieben in Dymola, wird von der Entwicklungsumgebung in ein System bzw. Teilsystem transformiert, welches direkt für eine CSSL Sprache (Continuous System Simulation Language) verwendbar ist.
Die Modelle werden hierarchisch in Teilmodelle zerlegt. Das Wiederverwenden von bekanntem Modellwissen wird unterstützt durch Modellklassenbibliotheken und durch Vererbungsmechanismen. Es können Differential- und algebraische Gleichungen beschrieben werden. Hybrid-Modelle aus kontinuierlichen und diskontinuierlichen Gleichungen werden mittels Routinen der numerischen Integration übersetzt. Weiterhin wird das Finden des Minimalsystems sowie das Lösen von algebraischen Schleifen unterstützt. Die Modellbeschreibungssprache ist geeignet zum Beschreiben von mechanischen, elektrischen, thermodynamischen und chemischen Systemen. Dymola verfügt über die Fähigkeit, mittels symbolischer Formelmanipulation die Teilprobleme

- Explizieren von Gleichungen,
- Lösung von linear abhängigen Gleichungen,
- Vereinfachung des Gleichungssystems und
- Eliminierung von trivialen Beziehungen

zu lösen und stellt daher einen wertvollen Preprozessor für CSSL Systeme (Continuous System Simulation Language) dar. Auch elektrische Netzwerke, Bondgraphen oder andere Objektwelten können rasch und sicher in Zustandsraummodelle verwandelt werden.
Die nächste Abbildung zeigt ein einfaches Modell einer Gleichstrommaschine.

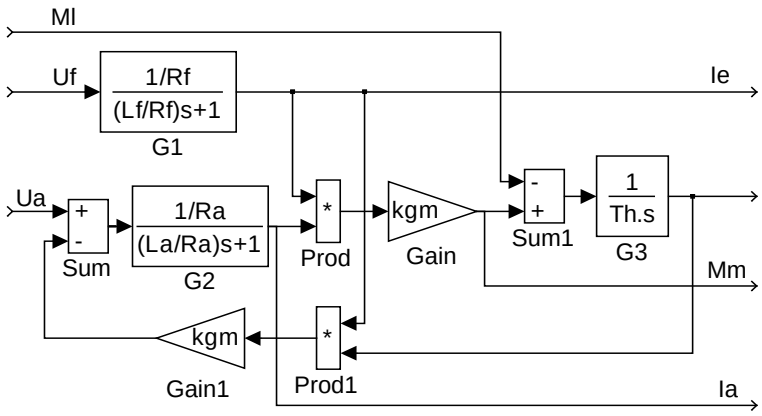


Abb. 2.2: Blockorientierte Darstellung des dynamischen Modells einer Gleichstrommaschine

Die physikalischen Grundgleichungen der Maschine lauten:

$$
\begin{aligned}
U_i &= k_{gm} I_e \omega \\
M_m &= k_{gm} I_e I_a \\
U_a &= R_a I_a + L_a \frac{dI_a}{dt} + U_i \\
U_f &= R_f I_e + L_f \frac{dI_e}{dt} \\
\Theta \frac{d\omega}{dt} &= M_m - M_l - B_m \omega
\end{aligned}
\qquad (2.3)
$$

Aus diesen beiden Daten kann direkt eine Klasse "*GM*" in Dymola definiert werden.

```
Model  class  GM
       cut  eAnker  ( Ua / Ia )
       cut  eErregung  ( Uf / Ie )
       cut  mAnker  ( Omega / Ml )
       parameter  Ra , La , Rf , Lf , Theta , Bm , Kgm
       local  Ui , Mm
       Ui  =  kgm * Ie * Omega                    { EMK }
       Mm  =  kgm * Ie * Ia                       { inneres mech. Drehmoment }
       Ua  =  Ra * Ia + La * der( Ia ) + Ui       { Ankerstromkreis - Maschine }
       Uf  =  Rf * Ie + Lf * der( Ie )            { Erregerstromkreis - Maschine }
       Theta * der( Omega )  =  Mm - Ml - Bm * Omega   { Drallsatz für Anker }
end                                                                    (2.4)
```

Die Schnittstellen des Objektes sind durch das Schlüsselwort "*cut*" eingeleitet. Nachdem man Parameter ("*parameter*") und lokale Variablen ("*local*") deklariert hat, können die physikalischen Gleichungen direkt mit Hilfe des "*der()*"-Operators (Ableitung nach der Zeit) ohne Rücksicht jeglicher Kausalität angegeben werden. Kommentare sind in das Klammerpaar "*{}*" eingeschlossen.
Dymola kann mit verschiedenen anderen Simulationspaketen kooperieren. Gleichungen können in die Formate von ACSL (Advanced Continuous Simulation Language), Desire, Simnon, SIMULINK, C und FORTRAN konvertiert werden. Dymola ist auf PC/Windows, UNIX, VAX/VMS und Macintosh verfügbar [8][9][10][11].

2.2.3 ProDynSA

ProDynSA (Prolog Dynamic Systems Analysis) ist ein lineares, blockdiagrammorientiertes Definitions- und Manipulationswerkzeug zur Diagrammtransformation und zur Simulation von dynamischen Systemen. Es ermöglicht Graphikmanipulationen wie Knoteneliminationen, Kaskaden- und Parallelkombinationen von Blöcken, das Entfernen von Rückführungsschleifen und graphische Umstrukturierungen von Verzweigungs- und Summierpunkten. Mit ProDynSA kann man Basisdatenstrukturen zur Systemrepräsentation beschreiben. ProDySA basiert auf einem Standard Prolog Interpreter, der alle Operationen ausführt. Anhand eines kleinen Beispiels sind nachfolgend einige Beschreibungselemente von ProDynSA erläutert.

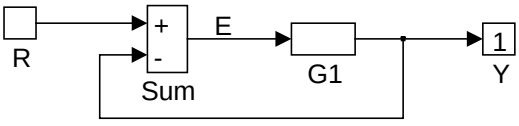


Abb. 2.3: Einfaches Beispiel eines Blockschaltbildes

Das Blockschaltbild in Abbildung 2.3 könnte man in ProDynSA wie folgt beschreiben.

```
connected(´R´, ´E´, 1)
connected(´E´, ´Y´, ´G1´)
connected(´Y´, ´E´, -1)
summer(´E´)
```

source(´R´)
sink(´Y´) (2.5)

ProDynSA bedient sich zur Beschreibung eines Systems der Blockdiagrammstruktur. Ein Blockdiagramm liefert eine symbolische Repräsentation von allen Grundsystemgleichungen in geschlossener Form. Es bildet die quantitative Basis für die Analyse und die Grundlage der Entwicklung von Modellen für analoge und digitale Computersimulationen.
Die "*connect*"-Regeln beschreiben Verbindungen zwischen einzelnen Positionen im Strukturbild. Sie sind von sehr einfacher Gestalt und stellen doch ein sehr mächtiges Beschreibungskonstrukt dar. Für jedes Signal, das am Ausgang eines Summierpunktes anliegt, wird die "*summer*"-Regel verwendet. Als Eingangs- ("*source*") bzw. Ausgangsgröße ("*sink*") stehen zwei weitere Sprachelemente zur Verfügung [12].

2.2.4 Simnon

Simnon ist ein kommandogesteuertes interaktives Systemsimulationsprogramm für nichtlineare Differential- und Differenzengleichungen. Es wird zum Lösen von Regel- und Simulationsaufgaben seit 1974 in der Wissenschaft, Lehre und Industrie verwendet. Die einfache Modellsprache und die erweiterte Fehlerbehandlung favorisieren Simnon zum Einsatz in der Lehre.
Das zu untersuchende Model kann in einer simnoneigenen Sprache oder in FORTRAN beschrieben werden. Der Compiler ist in der Entwicklungsumgebung integriert und arbeitet parallel mit dem Editor. Simnon erlaubt die Beschreibung von Teilsystemen, die durch Eingabe- und Ausgabepunkte aneinander gekoppelt werden können. Dies ermöglicht das Einbinden von Bibliotheksfunktionen. Modellbeschreibungen werden in eine Datei geschrieben und können dann mittels Kommandozeilenparameter in der Simulationsumgebung von Simnon aufgerufen werden. Man kann zeitkontinuierliche wie auch zeitdiskrete Teilsysteme beschreiben. Gleichungen stellt man wie in Algol-60 dar. Man hat die Möglichkeit, FORTRAN oder PASCAL Routinen einzubinden. Alle Schlüsselwörter, wie "*INPUT*", "*OUTPUT*", "*TIME*", "*STATE*", "*DER*" und "*NEW*", werden in Großbuchstaben geschrieben. Zur Darstellung der Ableitung einer Zustandsvariablen in zeitkontinuierlichen Systemen verwendet man "*DER*" und das Schlüsselwort "*NEW*" aktualisiert Variablen. Für Simnon existieren Pakete für Standardfunktionen wie "*ABS*", "*COS*", "*MAX*", "*DELAY*" u.a. .

Das nachfolgende Beispiel im zeitkontinuierlichen Bereich veranschaulicht die Syntaxstruktur der Simnonbeschreibungssprache.

$$\begin{aligned} \dot{x} &= a(y - x) \\ \dot{y} &= bx - y - xy \\ \dot{z} &= xy - cz \end{aligned} \tag{2.6}$$

```
CONTINUOUS SYSTEM test
      STATE x y z                 ´Zustandsvariablen
      DER dx dy dz                ´Ableitungen der Zustandsvariablen
      dx = a*( y - x )            ´Variablenzuweisung
      dy = b * x - y - x * z
      dz = x * y - c * z
      a : 10                      ´Parameterinitialisierung
      b : 28
      c : 2.667
      x : - 8                     ´Variableninitialisierung
      y : - 8
      z : 24
END                                                          (2.7)
```

Die Simnonbeschreibungssprache ist gut strukturiert und fast intuitiv anzuwenden.
Simnon ist lauffähig auf PC/DOS, VAX/VMS und UNIX. Seit 1993 ist Simnon auch als PC/Windows Version verfügbar [13].

2.2.5 SIMULINK

SIMULINK ist ein Programm zur Simulation dynamischer Systeme. Die Basis für SIMULINK bildet Matlab. Modelle können mittels graphischer Benutzeroberfläche definiert und analysiert werden. Es existieren eine Reihe von Beschreibungskomponenten, die mittels graphischer Fenster mit der Maus in die eigene Anwendung kopiert werden können. Ebenfalls mit der Maus verbindet man sehr einfach die einzelnen graphischen Teile der Systembeschreibung. Der Anwender kreiert und modifiziert so ein Modell und kann das Verhalten abspeichern. Verschiedene Programmerweiterungen sind vorhanden und erhöhen die Funktio-

nalität von SIMULINK. Der Verlauf einer Simulation kann unmittelbar beobachtet oder später im Arbeitsraum von Matlab weiterverarbeitet werden.
Die graphische Darstellung eines Beispiels in SIMULINK könnte folgendes Aussehen haben.

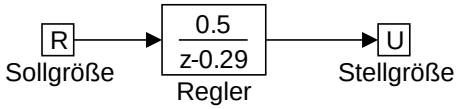


Abb. 2.4: Übertragungsfunktion im Z-Bereich mit Anschlüssen

In der Beschreibungssprache von SIMULINK sieht die Abbildung 2.4 wie folgt aus.

```
function [ ret , x0 , str , ts , xts ]  =  dipl_in7 ( t , x , u , flag ) ;
sys  =  mfilename ;
new_system( sys )
simver( 1.3 )
if ( 0  ==  ( nargin  +  nargout ) )
    set_param( sys , 'Location' , [238,153,623,265] )
    open_system( sys )
end ;
set_param( sys , 'algorithm' , 'RK-45' )
set_param( sys, 'Start time' , '0.0' )
set_param( sys , 'Stop time' , '10' )
set_param( sys , 'Min step size' , '0.0001' )
set_param( sys , 'Max step size' , '10' )
set_param( sys , 'Relative error' , '1e-3' )
set_param( sys , 'Return vars', '' )
set_param( sys , 'Real-time algorithm' , 'rk5' )

add_block( 'built-in/Inport' , [sys,'/','Sollgröße'] )
set_param( [sys,'/','Sollgröße'] , 'Font Name' , 'Arial' , 'Port' , 'R' , 'position' , [40,30,60,50] )

add_block( 'built-in/Discrete Transfer Fcn' , [sys,'/','Regler'] )
set_param( [sys,'/','Regler'] , 'Font Name' , 'Arial' , 'Numerator' , '[0.5]' , 'Denominator' , '[1 -0.29]' ,...
        'position' , [115,18,180,62] )

add_block( 'built-in/Outport' , [sys,'/','Stellgröße'] )
set_param( [sys,'/','Stellgröße'] , 'Font Name' , 'Arial' , 'Port' , 'U' , 'position' , [250,30,270,50] )
```

```
add_line( sys , [65,40;110,40] )
add_line( sys , [185,40;245,40] )

drawnow
if ( nargin | nargout )
        if ( nargin > 3 )
                if ( flag == 0 )
                        eval( [ '[ret,x0,str,ts,xts] = ', sys , '(t,x,u,flag);' ] )
                else
                        eval( [ 'ret = ', sys , '(t,x,u,flag);' ] )
                end
        else
                [ret,x0,str,ts,xts] = feval( sys ) ;
        end
else
        drawnow
end
```
(2.8)

SIMULINK zielt eindeutig auf das Beschreiben von Modellen mit graphischen Mitteln ab. Eine Modellbeschreibung per Hand durchzuführen bedarf einer längeren Einarbeitungsphase, ist fehleranfällig und zeitaufwendig. Komplexere Systembeschreibungen werden leicht unübersichtlich und schwer nachvollziehbar.

Seit einiger Zeit ist ein Erweiterungswerkzeug auf dem Markt, mit dem C Code von einer Modellbeschreibung erzeugt werden kann. Dieser C Code ist sehr komplex und enthält keine Funktionen zur Überwachung von Systemzuständen und zur Variablenveränderung.

Matlab/SIMULINK stellt ein mächtiges Werkzeug zur Analyse und Simulation von komplexen Systemen dar. Es ist im Hochschulbereich und in der Lehre sehr verbreitet.

SIMULINK in Verbindung mit Matlab benötigt MS Windows 3.0 oder höher [14].

2.3 Funktionalitäten von Simulationssystemen im Überblick

Die folgende Tabelle zeigt Funktionalitäten von Simulationssystemen im Überblick. Man unterscheidet sie anhand ihres Sprachumfanges und ihrer Beschreibungsmöglichkeiten.

	kontinuierlich	**diskret**	**linear**	**nichtlinear**	**gleichungsorientiert**	**Zustandsraum**	**Übertragungsfunktionen**	**Modellierung**	**Simulation und Analyse**	**Godegenerierung**	**Quellen**
ACSL	XX			X	XX		X	XX	XX		[1,3,6,7,15]
CC	XX	XX	XX	X		XX	XX		XX		[16]
Ctrl-C	XX	XX	XX	XX		XX	XX		XX		[17]
CSSL-IV	XX						X	XX	XX		[15]
Dymola	XX	XX	XX	X	XX			XX			[8,9,10,11]
EASY5	XX	XX	XX	XX	XX		X	XX	XX		[15,18]
ProDynSA			XX						XX		[12]
Simnon	XX	XX	X	XX	XX				XX		[13]
SIMULINK	XX	XX	XX	XX		XX	XX	XX	XX	XX	[14]
CLDL		XX	XX	X	XX		XX	XX		XX	

Tabelle 2.1: Zusammenstellung einiger Simulationsprodukte

Man findet in [5] eine weitere Zusammenfassung von Produkten mit dem Schwerpunkt des computerunterstützten Entwurfs von Regelsystemen auf PCs in Deutschland.

3 Entwurf der Regelkreisbeschreibungssprache CLDL

3.1 Motivation zur Entwicklung einer neuen Regelkreisbeschreibungssprache

Die Regelkreisbeschreibungssprache CLDL (Control Loop Description Language) soll die Beschreibung von Blockdiagrammstrukturen im zeitdiskreten Bereich ermöglichen. Der Sprachumfang ist relativ klein und überschaubar gehalten. Dies gewährleistet eine schnelle Einarbeitungszeit. CLDL muß für spätere Generierungszwecke maschinell verarbeitbar sein. Sie zielt auf einen beschränkten Einsatz mit gleichzeitiger Erweiterbarkeit in diesem Bereich. Durch die Spezialisierung auf einen bestimmten Bereich können praktische Aspekte im Sprachumfang berücksichtigt werden. Weiterhin soll sie sich an bestehende gebräuchliche Beschreibungssprachen wie etwa Matlab/SIMULINK anlehnen.
Diese Regelkreisbeschreibungssprache im zeitdiskreten Bereich sollte die Fähigkeit besitzen, Eingangs- und Ausgangsmeßgrößen mittels Übertragungsfunktionen und Summierstellen zu verbinden.

3.2 Sprachbeschreibung von CLDL

Die Sprachsyntax der Beschreibungssprache CLDL (Control Loop Description Language) lehnt sich an den von Matlab/SIMULINK an. Es wurden aber darüber hinaus noch zusätzliche Beschreibungselemente notwendig, um ein komplettes Regelkreisstrukturbild im zeitdiskteten Bereich zu beschreiben.
Zu einem Regelkreisblockschaltbild können der Regler (Filter) selbst, Vorfilter für die Führungsgröße (Sollgröße), Meßfilter der Regelgröße (Istgröße) und beliebige Zwischenfilter zur Umwandlung, Filterung oder Sättigung gehören. Die folgende Abbildung verdeutlicht das Einsatzgebiet der Regelkreisbeschreibungssprache CLDL.

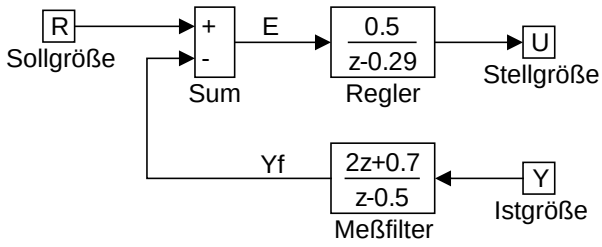


Abb. 3.1: Regelkreisblockschaltbild

Die Soll- und Istgröße sind die Eingangsgrößen des Regelkreises in der Abbildung 3.1 . Die Stellgröße ist die Ausgangsgröße des Regelkreises und gleichzeitig die Eingangsgröße der Strecke, die natürlich nicht beschrieben wird.
Der Schwerpunkt der weiteren Arbeit liegt in der Erweiterung der Sprachbeschreibung von CLDL und der anschließenden Entwicklung eines Übersetzers, der den beschriebenen Regelkreis in eine lauffähige C++ Header- und Quelltextdatei übersetzt.

3.2.1 Programmstruktur

Die Sprachbeschreibung der Grammatik von CLDL basiert auf dem Standard des Erweiterten Backus Naur Formalismus (EBNF). Für die Definition des Syntax von CLDL werden folgende Symbole verwendet.

Metasymbol	Beschreibung
=	definiert als
|	alternativ
.	Ende der Definition
[x]	keines oder eins von x
{x}	keines, eins oder mehrere von x
(x|y)	selektiert x oder y
´´xyz´´	Terminalsymbol
Bezeichner	Nichtterminalsymbol

Tabelle 3.1: Symboltabelle von CLDL in der EBNF

Die Programmstruktur von CLDL im großen sieht in der EBNF wie folgt aus.

```
Program =
        GlobalDefs
        Regelkreise.
GlobalDefs = { GlobalDef }.
Regelkreise = Regelkreis { Regelkreis }.
Regelkreis =
        ´´CONTROL´´ Bezeichner ´´;´´
        Regelkreiskörper
        ´´END´´ ´´;´´.
Regelkreiskörper = { Statement }.                                    (3.1)
```

Ein Programm in CLDL besteht aus globalen Definitionen ("*GlobalDefs*") und Regelkreisen ("*Regelkreise*"). "*GlobalDefs*" bestehen aus keiner, einer oder aus mehreren globalen Definitionen ("*GlobalDef*"). "*Regelkreise*" wiederum bestehen aus einem oder mehreren einzelnen Regelkreisen ("*Regelkreis*"). Der Regelkreis wird im weiteren auch Regelkreisblock oder nur Block genannt. Jeder Block beginnt mit dem Schlüsselwort "*CONTROL*", gefolgt von einem Blocknamen ("*Bezeichner*"). Das Ende jedes Blockes wird durch das Schlüsselwort "*END*" festgelegt. Alle zu einem Block gehörenden Anweisungen ("*Regelkreiskörper*") stehen zwischen diesen Schlüsselwörtern. Nach jedem "*CONTROL*" gefolgt von einem Blocknamen und nach jedem "*END*" steht ein Semikolon. Der "*Regelkreiskörper*" besteht wiederum aus keinem, einem oder mehreren Statements ("*Statement*"). Nach jedem "*Statement*" (Anweisung) folgt ebenfalls ein Semikolon.

Kommentare werden durch die Zeichenkombinationen "/*" und "*/" geklammert. Ein Kommentar darf überall dort stehen, wo auch ein Leerzeichen stehen kann. Er kann sich über mehrere Zeilen erstrecken und verschachtelt sein.

Man unterscheidet zwischen globalen und lokalen Anweisungen. Globale Anweisungen stehen außerhalb eines Blockes ("*GlobalDefs*") und sind in der gesamten Quelltextdatei bekannt. Lokale Anweisungen (eine Liste von "*Statement*") stehen innerhalb eines Blocks und sind auch nur in dieser Ebene bekannt.

Innerhalb einer CLDL Spezifikationsdatei ist es möglich, mehrere Regelkreise zu beschreiben, die dann durch den jeweiligen Blocknamen unterschieden werden. Das macht Sinn, wenn man einen fünf-achsigen Roboterarm beschreiben möchte, der aus fünf von einander unabhängigen

Gelenken besteht. Mit einer Datei "roboterarm" und fünf einzelnen Blöcken mit den Bezeichnern "gelenk1" bis "gelenk5" wird eine gute Strukturierung der Beschreibung erreicht.
Die komplette Syntaxbeschreibung von CLDL in der Backus Naur Form kann man im Anhang A finden.

3.2.2 Datentypen

Unter einem Datentyp versteht man eine Menge von Werten und eine Menge von Operationen, die auf diese Werte angewendet werden können.
Eine Variable (Bezeichner) ist die symbolische Repräsentation einer Systemgröße im Regelkreisblockschaltbild. Sie wird durch ihre Benutzung in einer Anweisung vereinbart. Jeder Name einer Variablen kann aus Buchstaben, dem Zeichen "_" (Unterstreichungszeichen) und Ziffern bestehen. Das erste Zeichen muß ein Buchstabe sein, wobei "_" als Buchstabe gilt. Es wird zwischen Groß- und Kleinschreibung unterschieden. In der EBNF sieht das wie folgt aus.

```
Bezeichner =
        Buchstabe { Buchstabe | Zahl }.
Buchstabe =
        ´´a´´ | ´´b´´ |... | ´´z´´ | ´´A´´ | ´´B´´ |... | ´´Z´´ | ´´_´´.
Zahl =
        ´´0´´ | ´´1´´ | ´´2´´ | ´´3´´ | ´´4´´ | ´´5´´ | ´´6´´ | ´´7´´ | ´´8´´ | ´´9´´.
```
(3.2)

Die Beschreibungssprache erlaubt die Verwendung von Integer- und Realkonstanten. Integerkonstanten können nur als Dezimalkonstanten (Basis 10) vorkommen. Realkonstanten bestehen aus einem ganzzahligen Anteil, einem Dezimalpunkt und einem gebrochenen Anteil.

```
Integer =
        Zahl { Zahl }.
Real =
        Integer |
        Integer ´´.´´ Integer |
        ´´.´´ Integer.
```
(3.3)

Um in einem Regelkreis Filter beschreiben zu können, wird ein neuer Datentyp eingeführt. Der Typ Filter besteht aus einem Zähler- und einem Nennerpolynom. Beide Polynome werden durch die Zeichen "[" und "]" geklammert. Das Zähler- und das Nennerpolynom wird von einander durch ein "/" getrennt. Ein Polynom kann von beliebiger Ordnung sein. Beginnend mit dem Glied höchster Ordnung wird jeder Koeffizient durch ein Komma vom nächsten getrennt. Ist ein Glied nicht vorhanden, muß eine "0" angegeben werden. Koeffizienten eines Gliedes können wiederum Dezimal- oder Realkonstanten sein (EBNF bitte im Anhang A vergleichen). Die Polynomdarstellung ist in Anlehnung an Matlab/SIMULINK gewählt worden.

Beispiele:

Dezimalkonstante:	5
Realkonstante:	2.17
Filterkonstante:	[1 , 0.25]/[1 , 0 , 3.37 , 8] (3.4)

3.2.3 Ausdrücke und Operatoren

Ausdrücke bestehen aus Operanden und Operatoren. Operanden können Variablen und Konstanten sein. Die Auswertung jedes Ausdrucks liefert einen Wert, der sich aus der Verknüpfung von Operanden durch Operatoren ergibt.
Die Wertzuweisung stellt die übliche Methode dar, einer Variablen einen bestimmten Wert zuzuordnen.

```
x = 0
y = x * 4
z = x - y                                                    (3.5)
```

Der rechts vom Operator "=" stehende Ausdruck wird berechnet und sein Wert der Variablen auf der linken Seite zugewiesen. Mehrfachzuweisungen sind bei der Regelkreisbeschreibungssprache CLDL nicht sinnvoll und werden deshalb auch nicht unterstützt.
Als elementare arithmetische Operationen werden der unäre Operator "-" und die binären Operatoren "+", "-" und "*" unterstützt. Die binären Operatoren "+" und "-" werden in der Beschreibungssprache verwendet, um Systemgrößen im Blockschaltbild zu summieren und zu subtrahieren. Der binäre Operator "*" hingegen darf nur zum Verbinden von Filter und Sy-

stemgröße verwendet werden. Andernfalls würde man versuchen, einen nichtlinearen Zusammenhang zu beschreiben, der im aktuellen Stand der Beschreibungssprache nicht unterstützt wird.

3.2.4 Schlüsselwörter

Die bereits bekannten Schlüsselwörter "*CONTROL*" und "*END*" dienen zur Abgrenzung eines Regelkreisblocks. Schlüsselwörter bestehen grundsätzlich aus Großbuchstaben. Funktionen werden durch Schlüsselwörter repräsentiert.
In CLDL werden zwei Funktionen unterstützt, die die Kommunikation eines Regelkreises mit anderen Teilen einer Steuerung sicherstellen. Diese werden durch die Schlüsselwörter "*IN()*" und "*OUT()*" repräsentiert und stellen die Schnittstellen vom Strukturbild zur Außenwelt dar. Mit "*IN(Variablenname)*" werden alle Eingangsmeßgrößen und mit "*OUT(Variablenname)*" die Ausgangsmeßgröße beschrieben. Ein Regelkreisblockschaltbild kann aus mehreren Eingangsmeßgrößen (Istgrößen, Sollgröße) aber nur aus einer Ausgangsmeßgröße (Stellgröße) bestehen.
Eine Funktion zur Amplitudenbeschränkung von Systemgrößen verhindert, daß die Ausgangsgröße (Stellgröße) die angeschlossene Hardware übersteuert. Mit dem Schlüsselwort "*LIMIT(untere Schranke, obere Schranke)*" kann man eine untere und eine obere Schranke für eine Meßgröße festlegen. Werden diese Schranken unter- bzw. Überschritten, werden die jeweiligen Maximalwerte weitergegeben.
An dieser Stelle sind noch weitere Funktionen zur effektiveren Begrenzung oder zum Beschreiben weiterer Regelkomponenten denkbar. Der momentane Sprachumfang reicht aber für einfache Regelkreise in den meisten Fällen aus.

3.3 CLDL Programmbeispiele

3.3.1 Einfacher Eingrößenregelkreis

Das folgende Strukturbild beschreibt einen einfachen Eingrößenregelkreis.

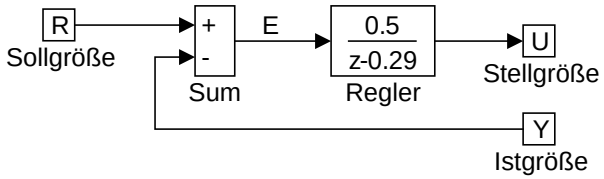


Abb. 3.2: Einfacher Eingrößenregelkreis

In der Regelkreisbeschreibungssprache CLDL kann man den in Abbildung 3.2 gezeigten Eingrößenregelkreis wie folgt darstellen.

```
CONTROL  Motor ;
        a1  =  0.5 ;
        b1  =  1 ;
        b2  =  - 0.29 ;
        Regler  =  [ a1 ] / [ b1 , b2 ] ;

        IN( R ) ;
        IN( Y ) ;
        E  =  R - Y ;
        U  =  Regler * E ;
        OUT( U ) ;
END ;                                                    (3.6)
```

Der Blockname "*Motor*" wird nach der Generierung zum Klassennamen der C++ Datei. Diese Regelkreisbeschreibung enthält nur lokale Anweisungen. Durch die Zuweisung der Filterkoeffizienten an spezielle Variablen, wird eine spätere Neuzuweisung der einzelnen Koeffizienten möglich. Die Anweisungen im "*CONTROL*"-Block können in beliebiger Reihenfolge und an beliebiger Stelle im Block stehen. Will man allerdings ein gut strukturiertes und leserliches Programm schreiben, sollte man Variablenzuweisungen an den Anfang eines Blockes

stellen. Das Schlüsselwort "*IN(Var)*" beschreibt eine Eingangsmeßgröße und weist ihr den Variablennamen "*Var*" zu. In unserem Beispiel sind das die Variablen "*R*" und "*Y*", die im weiteren Programmablauf als Variablen zur Verfügung stehen. Es wird dann der Regelfehler "*E*" gebildet und über den Filter ("*Regler*") die Stellgröße "*U*" für die Strecke berechnet. Damit die Stellgröße "*U*" (Ausgangsmeßgröße) außerhalb der Beschreibung bekannt wird, muß das Schlüsselwort "*OUT(U)*" verwendet werden.

3.3.2 Eingrößenregelkreis mit Vor- und Meßfilter

Ein zweites etwas größeres Beispiel soll die Mächtigkeit der Regelkreisbeschreibungssprache CLDL verdeutlichen.

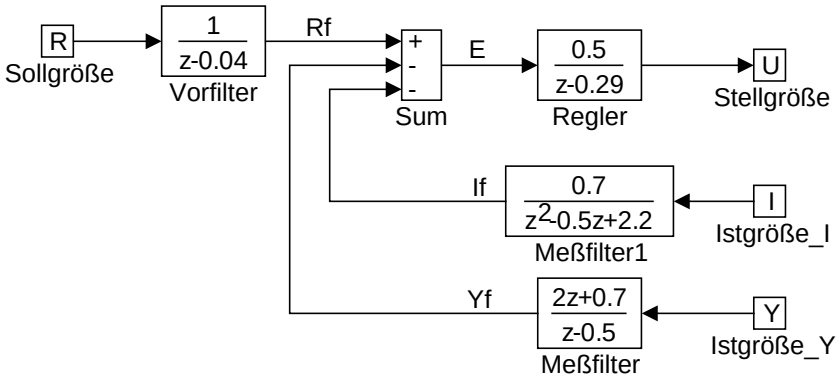


Abb. 3.3: Komplexerer Eingrößenregelkreis

Das Beispiel aus Abbildung 3.3 zeigt einen Eingrößenregelkreis mit mehreren Meßfiltern. Das hat seine Berechtigung, da stark verrauschte Meßgrößen gefiltert werden müssen, um eine gute Regelung zu erreichen. Meßfilter sind im allgemeinen zum Regler gehörend zu betrachten, wenn sie per Software realisiert werden sollen. Das nachfolgende Programm beschreibt diesen Regelkreis in CLDL.

```
CONTROL  Motor1 ;
        a1  =  1 ;
        b1  =  1 ;
        b2  =  - 0.04 ;
        Vorfilter  =  [ a1 ] / [ b1 , b2 ] ;
        c1  =  0.5 ;
```

```
        d1  =  1 ;
        d2  =  - 0.29 ;
        Regler  =  [ c1 ] / [ d1 , d2 ] ;
        e1  =  0.7 ;
        f1  =  1 ;
        f2  =  - 0.5 ;
        f3  =  2.2 ;
        Meßfilter1  =  [ e1 ] / [ f1 , f2 , f3 ] ;
        g1  =  2 ;
        g2  =  0.7 ;
        h1  =  1 ;
        h2  =  - 0.5 ;
        Meßfilter  =  [ g1 , g2 ] / [ h1 , h2 ] ;

        IN( R ) ;
        IN( I ) ;
        IN( Y ) ;
        Rf  =  Vorfilter * R ;
        If  =  Meßfilter1 * I ;
        Yf  =  Meßfilter * Y ;
        E  =  Rf - If - Yf ;
        U  =  Regler * E ;
        U = LIMIT( -12 , 12 ) ;
        OUT( U ) ;
END ;                                                    (3.7)
```

Die Hauptarbeit beim Beschreiben eines größeren Regelkreises ist die Zuweisung der Filterkoeffizienten. Wie im obigen Beispiel zu sehen ist, wird den Variablen "*a*", "*b1*", "*d1*", "*f1*" und "*h1*" der selbe Wert "*1*" zugewiesen. Es ist ratsam, verschiedene Variablen zu verwenden, um bei einer späteren Setup-Funktion auch wirklich nur den gewünschten Wert zu verändern. Die Setup-Funktion ermöglicht ein späteres Verändern von Koeffizienten im laufenden Betrieb der Reglersoftware. Nach dem Zuweisen der Filterkoeffizienten werden die Eingangsmeßgrößen des Systems über einen jeweiligen Filter an den Summierpunkt geführt. Es ist dann allgemein der Regelfehler "*E*" zu bilden. Der Regelfehler ist mit dem eigentlichen Regler direkt verbunden, dessen Ausgang (Ausgangsmeßgröße) extern an die Strecke angeschaltet werden muß.

Die verwendete "*LIMIT*"-Funktion gewährleistet eine Amplitudenbeschränkung von ± 12 (z.B. Volt) um nachfolgend angeschlossene Hardwarekomponenten nicht zu überlasten.

4 Generierungsvorgang der Regelkreissystemsoftware

4.1 Softwaregenerierung

Der Begriff Softwaregenerierung wird in diesem Beitrag als die Berechnung und Erzeugung von Quellsoftware mittels eines Softwarewerkzeuges aufgefaßt. Das Softwarewerkzeug, als Generator bezeichnet, kann ein speziell entwickelter Übersetzer oder ein durch ein Makroprogramm gesteuerter Makroprozessor sein. Generierung ist bei einem sich in den Strukturen wiederholenden Quellprogramm am effizientesten anwendbar. Die Anwendung der Generierung fördert die Entwicklung ausgereifter und regulärer Strukturen.

Ein Entwickler betrachtet bei konventioneller Programmierung einen gegebenen Aufgabenfall und versucht diesen durch entsprechende Algorithmen und Funktionen zu lösen. Ausnahmefälle in der Aufgabenstruktur löst er durch entsprechende Ausnahmefallbehandlungen. Diese Implementierungen werden nachträglich in großer Anzahl vorgenommen, was zu einer logisch zerfaserten Programmstruktur führen kann.

Anders hingegen verhält sich der Fall, wenn Quellsoftware mittels eines Übersetzers generiert werden soll. Hier bedeutet jede Ausnahmebehandlung einen erhöhten Programmieraufwand an Codierungsprädikaten. Bei generierten Funktionen mit allgemeingültigem Lösungscharakter hingegen sinkt der Aufwand für die zu definierende Codierungsprädikate in hohem Maße. Sich wiederholende Programmstrukturen können, unabhängig davon wie oft sie gebraucht werden, durch eine konstante Menge von Codierungsregeln erzeugt werden. Dieser Umstand bewirkt, daß der Entwickler länger über sich wiederholende Programm- und allgemeingültige Funktionsmuster nachdenkt, um seinen Programmieraufwand bei der Generatorentwicklung zu minimieren. Deshalb besitzen generierte Programme mit hoher Wahrscheinlichkeit einen orthogonalen Charakter. Levi bezeichnet in diesem Zusammenhang einen Entwickler, der die Generierung anwendet, als kartesischen Programmierer [19].

Durch Softwaregenerierung werden einzelne Entwicklungsschritte ganz oder teilweise eingespart. Monotone und langwierige Arbeiten entfallen in bestimmten Aufgabenbereichen. Der eingesetzte Generator führt diese Arbeiten in hoher Qualität aus. Der Generierungsvorgang ist durch hohe Effizienz, Kostenersparnis und Fehlerunanfälligkeit gekennzeichnet.

4.2 Benutzung des CLDL Generators

Bei der Entwicklung von Steuerungssoftware spielen Reglermodule eine immer größere Rolle. Das Ziel ist es, eine vollständige und spezifikationskonforme Reglersoftware zu entwikkeln, um damit regelspezifische Aspekte in lauffähige Programme leichter umsetzen zu können. Weiterhin soll eine Methode zur Teilautomatisierung des Softwareerstellungsprozesses verwendet werden. Ein dafür entwickelter Übersetzer transformiert die formalen Textstrukturen des Regelkreisbeschreibungsdokumentes direkt in einen C++ Quellcode. Dieser Zusammenhang ist in der folgenden Abbildung 4.1 dargestellt.

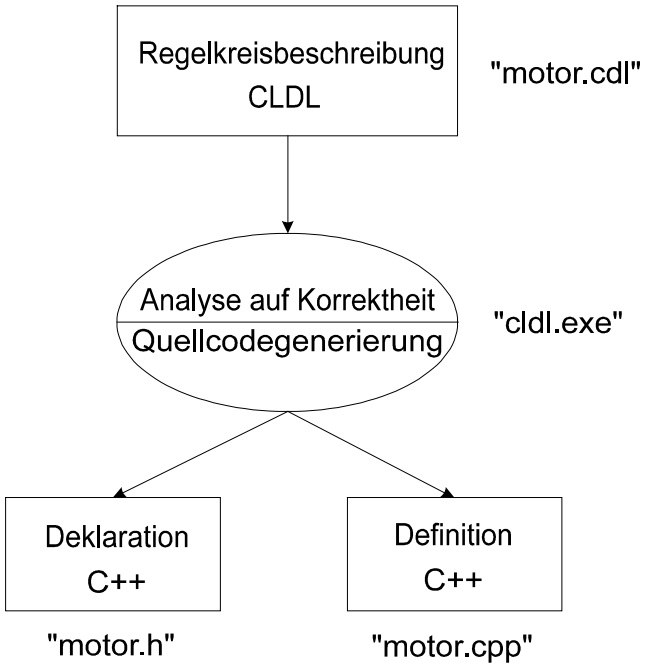


Abb. 4.1: Benutzung des CLDL Reglergenerators

CLDL ist mit wenigen Sprachelementen aufgebaut und daher einfach zu benutzen. Fehler in der Definition von Regelkreisstrukturen sollen zudem möglichst früh und vollständig vom CLDL Compiler erkannt werden. Schon während der Übersetzungszeit sollte die Übereinstimmung zwischen Definition und Aufruf von Reglerfunktionen vom CLDL Compiler überprüfbar sein.

Die Reglersoftware wird in C++ Klassen bestehend aus Funktionsdeklarationen und den dazu gehörenden Klassendefinitionen entwickelt. Der Quellcode der generierten Reglersoftware besitzt die folgenden, für den Softwareentwickler zugeschnittenen Eigenschaften.

- Alle Service-, Parameter- und Variablennamen sind direkt aus der Regelkreisspezifikation abgeleitet. Ein Entwickler, der die Regelkreisspezifikation kennt, kann so leicht den Bezug zur Reglersoftware herstellen. Die Software läßt sich somit leicht verstehen und benutzen.
- Die Software besitzt ein einheitliches Layout, was ebenfalls zur Lesbarkeit beiträgt.

4.3 Erzeugung des CLDL Generators

Für die Entwicklung des CLDL Übersetzers auf dem PC werden vier Werkzeuge verwendet, die eine entscheidende Unterstützung bieten.
Hierarchisch als erstes Glied in der Kette der Generierungswerkzeuge sind der Scannergenerator Flex und der Parsergenerator Bison zu nennen [20]. Mit deren Hilfe können die Komponenten des Übersetzers zur lexikalischen und syntaktischen Analyse als C Quelltext generiert werden. Die Abtrennung der lexikalischen Analyse führt zur Effizienzsteigerung des Übersetzungsprozesses. Bei Änderungen, verursacht durch den Wechsel von Zeichensätzen auf verschiedenen Plattformen, verändert sich nur die Beschreibung des Scanners, die der zeitlich nachfolgenden Übersetzungsvorgänge hingegen nicht.
Zum anderen kommt der Compilergenerator Gentle [21][22][23][24] zum Einsatz, der in der Entwicklungshierarchie über Bison und Flex steht. Das bedeutet, daß der Übersetzer hauptsächlich im Kalkül von Gentle spezifiziert ist. Aus dieser Gentle Spezifikation werden dann der lexikalische Spezifikationsteil für Flex, der gesamte Parser für Bison und die übersetzungsspezifischen Prozeduren für den C Compiler abgeleitet. Reflex als weitere Ergänzung zu Gentle erstellt die vollständige lexikalische Spezifikation aus einigen erzeugten Gentle Dateien und den zusätzlich von Hand geschriebenen Teilen.
Die Abbildung 4.2 verdeutlicht das Zusammenspiel der Compilerwerkzeuge und gibt einen Überblick über generierte Zusatzdateien .

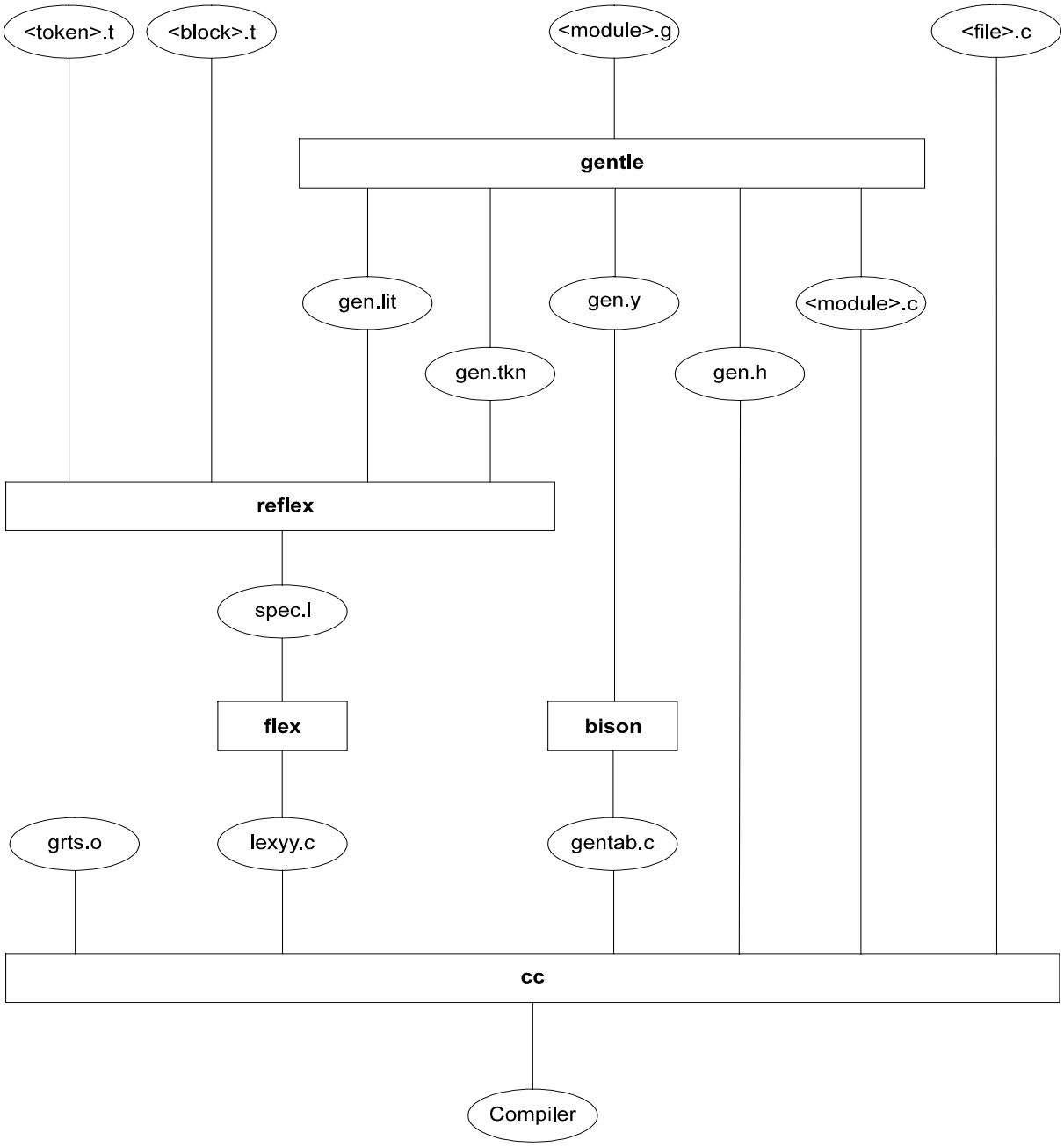


Abb. 4.2: Generierung eines Compilers [21]

4.3.1 Der Scannergenerator Flex

Zur Spezifikation der lexikalischen Einheiten, der sogenannten Token, wird der Scannergenerator Flex benutzt. Der von Flex generierte Scanner faßt die aus der Quelldatei kommenden Zeichen des Eingabestroms zu Token zusammen. Er filtert nicht verwertbare Textabschnitte wie Kommentare, Leerzeichen oder Tabulatoren heraus. Weiterhin kann es notwendig wer-

den, textuelle Repräsentationen eines Wertes in eine interne Darstellung umzuwandeln, damit der Übersetzer später mit diesen Werten rechnen kann. Token und Werte leitet der Scanner an den Parser weiter.
Zur Laufzeit des generierten Scanners wird nach jedem erkannten Token eine Aktion aufgerufen. Diese Aktionen sind in der Sprache des Scannergenerators in Form von C Anweisungen formuliert. Dadurch wird es möglich, externe in C geschriebene Datenstrukturen einzubinden, um eine Kommunikation mit dem Parser und den von Gentle generierten Softwareteilen sicherzustellen.

4.3.2 Der Parsergenerator Bison

Die vom generierten Scanner erfaßte Folge von Token dient als Eingabe für den generierten Parser. Der generierte Parser wird aus einer Spezifikation automatisch vom Parsergenerator Bison erzeugt. Die Spezifikation setzt sich aus einer Menge von Produktionsregeln und den dazugehörigen Aktionen zusammen. Diese Produktionsregeln sind stark an die Backus Naur Form angelehnt. Sie ermöglicht die syntaktische Beschreibung der Quellsprache (Grammatik). Eine Produktionsregel besteht aus einem Nichtterminal auf der linken Seite und einer Folge von Token und Nichtterminalen auf der rechten Seite.

Statement = Bezeichner ´´=´´ Expression ´´;´´ . (4.1)

Die Gesamtheit aller Produktionsregeln repräsentieren die Grammatik der Quellsprache. Jede Regel kann eine Aktion auslösen, wenn sie vom Parser richtig erkannt wird. Eine solche Aktion wird durch eine Folge von C Anweisungen spezifiziert.
Die gesamte Grammatik ist eine Art Regelhierarchie von den einzelnen Token auf der untersten Ebene bis zum Startsymbol, dem Anfang der Grammatik, auf der obersten Ebene. Der generierte Parser arbeitet nach dem Bottom-Up Verfahren ('von den Blättern zur Wurzel'). Dieses Verfahren erkennt zuerst die tiefer liegenden Nichtterminalen einer Zeichenfolge und reduziert diese schrittweise anhand der Produktionsregeln, so daß die letzte Reduktion auf das Startsymbol der Grammatik führt. Wenn der Übersetzer die Grammatik durchläuft, werden die Aktionen der einzelnen Produktionsregeln aufgerufen und halten den Weg fest [25][26].

4.3.3 Der Compilergenerator Gentle

Der Compilergenerator Gentle beinhaltet zum einen die Funktionalität von Bison und zum anderen die Beschreibungsmittel eines streng getypten logischen Prädikatenkalküls mit eingeschränktem Backtracking. Durch die Beschreibung der Grammatik in diesem Kalkül ist eine flexible Spezifikation des Parsers mit den entsprechenden reduktionsbezogenen Aktionen durchführbar.
Gentle beinhaltet kein eigenes Parsergenerierungsverfahren. Es wird eine im Gentlekalkül formulierte Grammatikspezifikation in eine Bisonspezifikation übersetzt, so daß Bison die eigentliche Parsergenerierung vornimmt. Die Aktionen der Grammatikregeln von Gentle werden ebenfalls in C Anweisungsfolgen umgesetzt und tauchen in der generierten Bisonspezifikation auf.

4.3.3.1 Typen

Gentle erlaubt mit seinem Typkonzept die Beschreibung nicht zyklischer Datenstrukturen wie z.B. Terme in Form von Listen und Bäumen. Ein Typ in Gentle wird durch das Schlüsselwort "*'type'* " repräsentiert und besteht aus einer Folge von alternativen Funktoren, wobei jeder Funktor selbst eine Liste von Argumenten besitzen kann.

```
'type' LISTE = leer
             | liste( INT , LISTE )                    (4.2)
```

In (4.2) setzt sich der Typ "*LISTE*" aus den Funktoren "*leer*" und "*liste(INT, LISTE)*" zusammen. Funktoren lassen die Konstruktion beliebig großer Terme zu, wie z.B. liste(0, liste(1, liste(2, liste(3, leer)))). Dieser Typ repräsentiert eine Liste von Integerzahlen.
Derartige Terme werden zur Laufzeit des Übersetzers im Arbeitsspeicher aufgebaut. Die Argumente der Funktoren können wieder beliebig andere Typen sein, so daß sich komplexe Datenstrukturen zusammenbauen lassen.

4.3.3.2 Prädikate

Prädikate ermöglichen die Formulierung von Algorithmen, die auf durch Typen beschriebene Daten operieren. Sie besitzen formale Ein- bzw. Ausgabeparameter, deren Struktur durch Typen festgelegt wird.

```
'action ' Sum( LISTE -> INT )
        'rule ' Sum( leer -> 0 ) :
        'rule ' Sum( liste( X , Restliste ) -> X + Y ) :    Sum( Restliste -> Y )
```
(4.3)

In diesem Beispiel (4.3) wird das Prädikat "*Sum*" definiert. Der Eingabeparameter ist vom Typ "*LISTE*" und der Ausgabeparameter vom Typ "*INT*". Der Typ "*LISTE*" wird im Speicher als eine einfache verkettete Liste repräsentiert. Das Prädikat "*Sum*" durchläuft jedes Element der Liste durch den rekursiven Aufruf von sich selbst, repräsentiert durch die zweite Regel. Trifft es auf ein "*leer*" des Typs "*LISTE*", erste Regel, ist das Prädikat am Ende der Liste angekommen. Von der Wurzel an, also vom letzten Element der Liste, wird bei jedem Rückschritt die Produktionsregel "*X+Y*" ausgeführt, und so das Ergebnis weiter nach oben gereicht. Man spricht hier auch von der Traversierung der zu grundeliegenden Struktur.
Weitere Prädikate in Gentle sind "*'nonterminal'*" und "*'condition'*".

- "*'nonterminal'*" dient zur Spezifikation einer Grammatikregel. Die Ausführung dieses Prädikates wird vom Parser übernommen.
- "*'condition'*" liefert die Möglichkeit des Scheiterns eines Prädikates für eine Fallunterscheidung.

Bei der Ausführung eines Prädikates wird Patternmatching durchgeführt. Patternmatching ist das Vergleichen der Struktur eines aktuellen Eingabeterms mit dem Pattern, daß im Prädikat definiert wurde. Dieser Matchingprozeß kann scheitern und zum Abbruch der Generierung führen [26].

4.4 Architektur des entwickelten Übersetzers

4.4.1 Phasen der Übersetzung

Die Übersetzung einer CLDL Beschreibung erfolgt in mehreren Phasen. Zuerst wird das Spezifikationsprogramm, das aus einer vorgegebenen kontextfreien Grammatik von CLDL ableitbar ist, in eine abstrakte Syntax transformiert. Diese abstrakte Syntax enthält nur noch die wesentlichen Informationen für die spätere Übersetzung. Die abstrakte Syntax bildet die Grundlage für die Überprüfung der statischen Semantik und die Erzeugung des abstrakten Codes.
Die Übersetzung der abstrakten Syntax in den abstrakten Code bildet die zweite Phase des Übersetzungsvorganges. Der abstrakte Code kann als Zwischensprache aufgefaßt werden. Die Umsetzung in eine Zwischensprache ist notwendig, um mathematische Berechnungen oder grammatikalische Umstrukturierungen vorzunehmen.
Die letzte Phase bildet der Codegenerator, der aus der abstrakten Syntax einen C++ Code generiert. Dieser C++ Code setzt sich aus einem Deklarationsteil (Headerdatei) und aus einem Definitionsteil (Quelltextdatei) zusammen. Diese beiden C++ Dateien können dann in verschiedene Programmarchitekturen eingebunden werden.
Die folgende Tabelle verdeutlicht diesen Sachverhalt.

Paßnummer	Transformation	Ergebnis
1	Quellprogramm -> Abstrakte Syntax	Das ursprüngliche Programm liegt jetzt als Baumstruktur vor. Sie spiegelt die Grammatik von CLDL wieder und ist frei von syntaktischen Ausprägungen.
2	Abstrakte Syntax -> Abstrakter Code	Das transformierte und semantisch korrekte Programm liegt in einer Baumstruktur der Grammatik der zu generierenden Reglersoftware vor.
3	Abstrakter Code -> C++ Code	Der fertige rechnerunabhängige C++ Code liegt in Form zweier Dateien (Deklarations- und Definitionsdatei) zur Benutzung bereit.

Tabelle 4.1: Die drei Phasen/Pässe der Übersetzung

Man spricht auch in diesem Zusammenhang von einem drei-pässigen Übersetzer. Im allgemeinen gilt, daß die Ausgabe eines Passes als Eingabe für den nächsten Paß dient. Dies wird solange fortgesetzt bis der fertige C++ Code vorliegt. Wieviel Pässe ein Übersetzer haben sollte, hängt vom Modularisierungskonzept des Entwicklers ab. Ein Übersetzer kann beliebig viele Pässe (Phasen) haben.
Die Möglichkeit, den Übersetzungsvorgang in mehrere Phasen aufzuteilen, bietet folgende Vorteile:

- Verschiedene syntaktische Ausprägungen sind auf der Ebene des abstrakten Syntax gleich handhabbar. Somit muß eine Erweiterung der Grammatik der Quellsprache nicht unbedingt eine Erweiterung der nachfolgenden Phasen nach sich ziehen. Die Wiederverwendbarkeit von Programmteilen wird dadurch erhöht.
- Durch Austauschen des Codegenerators kann in eine andere Zielsprache übersetzt werden.
- Durch die Erhöhung des Modularisierungsgrades wird die Lesbarkeit und die Übersichtlichkeit der Software erhöht. Dies garantiert eine kurze Einarbeitungszeit und einfache Erweiterbarkeit der einzelnen Phasen, um Entwicklungskosten zu sparen.

4.4.1.1 Die abstrakte Syntax

In der ersten Phase der Übersetzung wird die CLDL Spezifikationsdatei vom Scanner und Parser in eine abstrakte Programmstruktur übersetzt. Diese abstrakte Programmstruktur liegt in einer eindeutigen Baumrepräsentation für Überprüfungen und weitere Übersetzungen bereit. In dieser Repräsentation spielen syntaktische Ausprägungen wie Kommata und Schlüsselwörter keine Rolle mehr. Syntaktische Fehler sind bereinigt und die Überprüfung der statischen Semantik erfolgt erst in der nächsten Phase der Übersetzung. Die abstrakte Syntax ist nun frei von Mehrdeutigkeiten und es braucht keine Rücksicht mehr auf das Parsierungsverfahren genommen werden. Somit sind weitere Übersetzungsschemata leichter lesbar und einfacher formulierbar.
Bei kleineren kontextfreien Grammatiken ist es möglich, den nächsten Paß einzusparen, ohne eine übersichtliche Programmstruktur zu verlieren. Man müßten dann syntaktische und semantische Fehler sowie eventuelle Umwandlungen von Links- nach Rechtsrekursion bereits während der Parsierung abarbeiten.

Die folgenden Regeln der CLDL Syntax in EBNF sollen exemplarisch einen Teil der abstrakten Repräsentation vorstellen. Die komplette Syntax ist dem Anhang A zu entnehmen.

```
Regelkreise =
        Regelkreis { Regelreis }.
Regelkreis =
        ´´CONTROL´´ Bezeichner ´´;´´
        Regelkreiskörper
        ´´END´´ ´´;´´.
Regelkreiskörper =
        { Statement }.
Statement =
        Bezeichner ´´=´´ Expression ´´;´´.
```
(4.4)

Aus diesen Regeln läßt sich systematisch die abstrakte Notation des CLDL Übersetzers ableiten.

```
'type' REGELKREISE = rk( REGELKREIS )
                     rkliste( REGELKREIS , REGELKREISE )
'type' REGELKREIS = regelkreis( IDENT , STMTS )
'type' STMTS = nil
               stmts( STMT , STMTS )
'type' STMT = stmt( IDENT , TERM )
'type' IDENT
```
(4.5)

Ein Programm in CLDL kann sich aus mehreren Regelkreisen zusammensetzen. Laut der EBNF besteht die Gesamtheit der Regelkreise aus einem oder aus mehreren Regelkreisen. Jeder Regelkreis besteht aus einem Typ "*IDENT*" (Identifier oder Bezeichner) und aus dem Typ "*STMTS*" (Statements). Der Bezeichner beinhaltet den Namen des Regelkreises und Statements steht für eine Folge von Anweisungen. Hier ist deutlich der Unterschied zur Syntax von CLDL zu erkennen, da die Schlüsselwörter zur Blockdarstellung (Regelkreisblock) hier nicht mehr vorhanden sind.

So entsteht aus der Syntax von CLDL eine abstrakte Form, die eine Analogie zwischen EBNF und abstrakter Notation erkennen läßt.

Die Umsetzung des CLDL Programms in die abstrakte Syntax wird vom generierten Parser übernommen.

4.4.1.2 Der abstrakte Code

In der zweiten Phase der Übersetzung wird aus der abstrakten Syntax des Quellprogramms eine abstrakte Repräsentation der C++ Programme der zu generierenden Reglersoftware erzeugt. Der Übersetzer behandelt nach einander jeden beschriebenen Regelkreis. Dies geschieht unter Einbeziehung der globalen und jeweils lokalen Definitionen.
Jeder Regelkreis durchläuft grob folgende Untersuchungen.

- Es wird nach Limitierungen von Größen (Meß- und Zustandsgrößen) im Regelkreis gesucht und extra im Baum unter "*LIMITS*" vermerkt.
- Die Regelkreisstruktur ist von der Stellgröße (Ausgangsmeßgröße) zurück bis zu den Eingangsmeßgrößen (Führungs- und Meßgrößen) zu expandieren. Es werden alle Blöcke, die in den globalen wie lokalen Anweisungen stehen, zusammengefaßt und durch Z-Transformation in eine Differenzengleichung vom Typ "*DGL*" umgewandelt. Diese Differenzengleichung liegt im Speicher als Liste von Termen für den Codegenerator bereit.
- Die fertige Differenzengleichung ist nach Monitorvariablen vom Typ "*MONITORS*" zu durchsuchen. Diese enthalten dann später die aktuellen Werte im Regelkreisblockschaltbild. Sie dienen dem Zweck der Überwachung, Fehlererkennung und Statistik im laufenden Betrieb der Reglersoftware.
- Weiter muß man die Differenzengleichung nach Ringpuffervariablen vom Typ "*RINGBUFS*" durchsuchen. Sie entstehen durch das Verwenden von Übertragungsfunktionen (Regler und Meßfilter) und deren Transformation in den zeitdiskreten Zustandsraum.
- Die globalen und lokalen Anweisungen müssen nach Übertragungsfunktionen durchsucht werden. Das ist notwendig, um eine C++ Funktion zu generieren, die die Möglichkeit bietet, ganze Filter bei laufender Reglersoftware zu verändern. Diese Übertragungsfunktionen sind in einem extra Typ "*STRUCS*" abgespeichert, da sie in der Baumstruktur des abstrakten Codes nicht mehr explizit auftreten.
- Als letztes ist in allen globalen und lokalen Anweisungen nach Filterkoeffizienten und Konstanten zu suchen. Dadurch wird es in einer weiteren generierten Funktion möglich, nur einzelne Koeffizienten oder Konstanten zu verändern. Der Typ "*SETVARS*" steht hier zum Speichern bereit.

Alle hier aufgeführten Typen sind in die Baumstruktur des abstrakten Codes eingegliedert. Ein Teil dieser Repräsentation hat folgendes Aussehen.

```
'type' CPROGRAM  =  cprogram( HEADER , CODE )
'type' HEADER  =  header( CLASS )
                  headers( CLASS , HEADER )
'type' CLASS  =  class( IDENT , MONITORS , SETVARS , STRUCTS , RINGBUFS )
'type' CODE  =  code( DGL , LIMITS )
                codes( DGL , LIMITS , CODE )
'type' DGL  =  dgl( TERM , TERM )                                    (4.6)
```

Ein C++ Programm setzt sich zusammen aus einem Header- (Deklarationsdatei) und einem Codeteil (Definitionsdatei). Die vereinfachte generierte C++ Deklarationsdatei besteht aus einer Klassendefinition, repräsentiert durch den Typ "*CLASS*", oder aus mehreren Klassendefinitionen. Jedem Regelkreis ist eine Klasse zugeordnet. Jede Klasse setzt sich aus den bereits oben beschriebenen Typen zusammen.
Die Definitionsdatei kann wiederum aus mehreren Codeteilen je nach Anzahl der beschriebenen Regelkreise bestehen. Die kompletten Informationen stehen in den beiden Typen "*DGL*" und "*LIMITS*", aus denen dann im dritten Paß der Code generiert wird.

4.4.2 Implementierung

In diesem Kapitel steht die implementierungstechnische Seite des CLDL Übersetzers im Vordergrund. Es soll nicht der komplette Übersetzer illustriert werden, sondern auf wesentliche und typische Aspekte der einzelnen Phasen eingegangen werden. Es sind also nur beispielhafte kleine Teile der Implementierung vorgestellt.
Als Grundlage aller Entwicklung steht der 32 Bit Watcom C/C++ Compiler 10.6 auf dem PC zur Verfügung. Er ermöglicht eine zuverlässige Übersetzung der von Bison und Gentle generierten C Dateien. Gentle erzeugt einen 32 Bit Code, der dadurch aber nicht der 640K Restriktion unterliegt. Auf dieser Ebene der Portabilität kann der CLDL Übersetzer mittels seiner C Quelldateien auf andere Plattformen übertragen werden.

Analog zu den vorherigen Kapiteln sind typische Phasen der Übersetzung aufgeführt. Es werden einige Übersetzungsschemata in der Terminologie des Compilergenerators Gentle angegeben und erklärt.

4.4.2.1 Übersetzung vom Quelltext in die abstrakte Syntax

Die folgenden Beispiele für Grammatikregeln beziehen sich auf das Kapitel 4.4.1.1 "Die abstrakte Syntax", um den Übersetzungsvorgang besser nachvollziehen zu können. Da Gentle die Spezifikation einer Grammatik nicht in EBNF zuläßt, muß man die Umsetzung in die kontextfreie Form (BNF) per Hand erledigt. Das kann mittels einfacher Regeln geschehen und stellt daher kein Problem dar.

```
'nonterm' Regelkreise( -> REGELKREISE )
        'rule' Regelkreise( -> rk( EinRK ) ) :
                Regelkreis( -> EinRK )
        'rule' Regelkreise( -> rkliste( EinRK , RKreise ) ) :
                Regelkreis( -> EinRK )
                Regelkreise( -> RKreise )
'nonterm' Regelkreis( -> REGELKREIS )
        'rule' Regelkreis( -> regelkreis( Ident , Stmts ) ) :
               "CONTROL" I_IDENT( -> Ident ) ";"
                RKBody( -> Stmts )
               "END" ";"
'nonterm' RKBody( -> STMTS )
        'rule' RKBody( -> nil ) :
        'rule' RKBody( -> stmts( Stmt , Stmts ) ) :
                Statement( -> Stmt )
                RKBody( -> Stmts )
'nonterm' Statement( -> STMT )
        'rule' Statement( -> stmt( Ident , Term ) ) :
                I_IDENT( -> Ident ) "=" Term1( -> Term ) ";"
        'rule' Statement( -> stmt( Ident , limit( Ident , Low , High ) ) ) :
                I_IDENT( -> Ident ) "=" "LIMIT" "(" I_INT( -> Low ) "," I_INT( -> High ) ")" ";"
        'rule' Statement( -> stmt( Ident , func( Func ) ) ) :
                Function( -> Ident , Func )
```
(4.7)

Das Schlüsselwort "*'nonterm'* " kennzeichnet die Spezifikation einer Grammatikregel in Gentle. Dieses Prädikat findet im ersten Paß des Übersetzers seinen Einsatz, um die abstrakte Syntax der Quellsprache zu spezifizieren.
Mit dem Nichtterminal "*Regelkreise*" kann man eine Liste von Regelkreisen beschreiben. Da der Parser nach dem Bottom-Up Verfahren arbeitet, beginnt er an einem Blatt und versucht, die passenden Regeln zur Wurzel hin zu ersetzen. Sind zwei Regelkreise in CLDL beschrieben, merkt er am Ende des ersten Regelkreises, daß er die zweite "*'rule'* " im Nichtterminal "*Regelkreise*" wählen muß, um weitere Regelkreise ersetzen zu können. Da die Struktur jedes einzelnen Regelkreises in der Quelldatei immer gleich bleibt, kann eine Vielzahl von Regelkreisen durch Rekursion beschrieben werden. Rekursion stellt eine Vereinfachung des Beschreibungsaufwandes dar.
Das Nichtterminal "*Regelkreis*" beschreibt genau einen Regelkreis in CLDL. Er muß das Schlüsselwort "*CONTROL*", ein Bezeichner, eine Menge von Anweisungen und das Schlüsselwort "*END*" enthalten. Der Bezeichner enthält den Namen des Regelkreises, der später im generierten C++ Code als Klassenname wieder auftaucht.
"*Statement*" beschreibt eine Menge von Anweisungen. Das kann sein, ein Bezeichner gefolgt von einem "=" gefolgt von einem Nichtterminal "*Term1*" oder ein Bezeichner gefolgt von einem "=" gefolgt von dem Schlüsselwort "*LIMIT*" mit einer unteren und oberen Schranke oder ein Nichtterminal "*Function*".
So ist es möglich, durch extrahieren der Nichtterminale einen abstrakten Syntaxbaum von CLDL zu erstellen. Will man später die Grammatik erweitern, kann man einfach im Nichtterminal "*Statement*" weitere Möglichkeiten einer Anweisung hinzufügen.

4.4.2.2 Übersetzung der abstrakten Syntax in den abstrakten Code

Im zweiten Paß erfolgt die Umsetzung vom abstrakten Syntaxbaum in den abstrakten Code der Zielsprache. Es sind im folgenden Auszug die Kernschritte der Umsetzung aufgeführt. Die Transformation eines Regelkreises erfolgt in sieben Teilschritten, wie schon im Kapitel 4.4.1.2 kurz erläutert wurde.

```
'action' TransformOneRK( STMTS,REGELKREIS -> CLASS,DGL,LIMITS )
        'rule' TransformOneRK( GlobalDefs , regelkreis( Rkname , Stmts ) ->
                                class( Cname , Monitors , Setvars , Structs , Ringbufs ) , Regler , Limits ) :
```

```
let( Rkname -> Cname )
LookForLimits( GlobalDefs , Stmts -> NewGlobalDefs , NewStmts , Limits )
IsLookForOutMgr( NewStmts -> OutId )
ExpandRegelkreis( NewGlobalDefs , NewStmts , OutId -> Regler )
LookForMonitors( Regler -> Monitors )
LookForRingbufs( Monitors , Regler -> Ringbufs )
FilterToStruct( NewGlobalDefs , NewStmts -> Structs )
LookForSetvars( NewGlobalDefs , NewStmts -> Setvars )
```
(4.8)

Das Prädikat "*TransformOneRK*" hat als Eingangsparameter die globalen Definitionen "*GlobalDefs*" und den Namen "*Rkname*" mit den lokalen Definitionen "*Stmts*" des jeweiligen Regelkreises. Als Ausgangsparameter liegen dann vor, die Klasse "*class(...)*", die Differenzengleichung "*Regler*" und Begrenzungen "*Limits*". Jede Klasse besteht aus den Teilen Klassenname, Monitorvariablen, Koeffizientenvariablen, Übertragungsfunktionen (Filter) und Ringpuffervariablen. Nacheinander sind alle Schritte im Prädikat kurz erklärt.

- Der Regelkreisname "*Rkname*" wird gleichzeitig zum Klassenname der C++ Deklaration, da eine Klasse in der Zielsprache genau einem Regelkreisblock in der Quellsprache entspricht.
- "*LookForLimits*" wird dazu verwendet, aus den globalen und lokalen Anweisungslisten vom Typ "*TERM*" den Funktor "*limit*" zu finden, und in eine Liste vom Typ "*LIMITS*" zu schreiben. Diese Anweisungslisten sind dann frei vom Funktor "*limit*".
- "*IsLookForOutMgr*" ist ein Conditionprädikat welches eine Fallunterscheidung ermöglicht. Hier wird nach einer Ausgangsmeßgröße gesucht. Wird keine gefunden, liefert der Parser eine Fehlermeldung.
- Das Prädikat "*ExpandRegelkreis*" bildet den Kern der Übersetzung. Mittels der globalen und lokalen Anweisungen wird die Differenzengleichung der Stellgröße "*OutId*" in den Typ "*DGL*" (Regler) geschrieben.
- "*LookForMonitors*" sucht in der gebildeten Differenzengleichung nach Monitorvariablen. Sie werden in eine Liste vom Typ " *MONITORS*" kopiert und bilden die Grundlage des Codegenerators zur Erzeugung der Überwachungsfunktion.
- Mit "*LookForRingbufs*" wird in der Differenzengleichung nach Filtervariablen gesucht und in eine Liste vom Typ "*RINGBUFS*" kopiert. Durch das Verwenden von Übertragungsfunktionen werden Werte eines Zustandes auch aus der Vergangenheit benötigt, um einen

Regelalgorithmus laut Spezifikation zu erhalten. Diese Variablen müssen dann vom Codegenerator gesondert ausgegeben werden.

- "*FilterToStruct*" sucht aus den globalen und lokalen Anweisungen die Filterdefinitionen heraus und schreibt sie in eine Liste vom Typ "*STRUCTS*". Anhand dieser Liste bildet der Codegenerator eine Setup-Funktion für Übertragungsfunktionen. Es muß hier eine extra Liste gebildet werden, da die Filter (Übertragungsfunktionen) nicht mehr explizit in der Differenzengleichung ("*DGL*") enthalten sind.
- Als letztes sucht auch "*LookForSetvars*" nach Konstanten und Filterkoeffizienten in den globalen und lokalen Anweisungen und schreibt sie in eine Liste vom Typ "*SETVARS*". Auch hier bildet der Codegenerator daraus eine Setup-Funktion für den laufenden Betrieb des Regelalgorithmus.

In diesem Paß sind alle die Umsetzungen vorzunehmen, die unabhängig von der Quell- (CLDL Spezifikation) und Zielsprache (C++ Code) laut Aufgabenstellung sind. Aus Sicht der Wiederverwendbarkeit ist diese Vorgehensweise unabdingbar um zukünftige Erweiterungen oder Veränderungen vornehmen zu können. Es kann somit z.B. der Codegenerator nach C++ durch einen Codegenerator nach FORTRAN ausgetauscht werden.

4.4.2.3 Übersetzung des abstrakten Codes nach C++

Dieser Paß des Übersetzungsvorganges wird allgemein als Codegenerator bezeichnet. Er setzt den abstrakten Code in die Zielsprache C++ um. Er erstellt also die Deklarations- und die Definitionsdatei. Es folgen einmal das Prädikat "*WriteOutput*", welches die grobe Struktur des Codegenerators enthält, und das Prädikat "*WHeaderRingbufs*", daß die Filtervariablen deklariert.

```
'action' WriteOutput( CPROGRAM )
    'rule' WriteOutput( cprogram( Header , Code ) ) :
        OutFNameH -> OutFNameH
        OpenOutput( OutFNameH )
        WHeaderHead( )
        WHeaders( Header )
        CloseOutput( )
        OutFNameC -> OutFNameC
        OpenOutput( OutFNameC )
```

```
            WCodeHead( OutFNameH )
            WCodes( Header , Code )
            CloseOutput( )

'action'  WHeaderRingbufs( RINGBUFS )
      'rule'  WHeaderRingbufs( nil ) :
      'rule'  WHeaderRingbufs( ringbufs(Id , Int , Bufs ) ) :
            Put("    double yy_")  PutIdent( Id )  Put("[")  PutI( Int+1 )  Put("];")  Nl
            WHeaderRingbufs( Bufs )                                                    (4.9)
```

Das Prädikat "*WriteOutput*" hat als Eingangsparameter den Typ "*CPROGRAM*" der sich aus einem Funktor "*cprogram*" und den Typen "*HEADER*" und "*CODE*" zusammensetzt.
Als erstes wird die globale Variable "*OutFNameH*", welche den Namen und die Extension der Quelldatei enthält, lokal gespeichert, und die Deklarationsdatei (Headerdatei) der Zielsprache C++ geöffnet. Diese Datei wird dann mit relevanten Daten (Klassendeklarationen) des abstrakten Codes gefüllt. Anschließend wird die Headerdatei geschlossen und liegt im Speicher bereit.
Als zweites wird der Name und die Extension der C++ Definitionsdatei (Codedatei) in "*OutFNameC*" lokal gespeichert. Nach dem Öffnen der Definitionsdatei werden alle Daten des abstrakten Codes nach C++ übersetzt. Es wird hier sowohl der Typ "*HEADER*" als auch der Typ "*CODE*" benötigt. Nach dem Schließen liegt auch diese Datei im Speicher bereit.
Das Actionprädikat zeigt deutlich die Herangehensweise beim Generieren der Zielsprache. Es wird der Typ "*RINGBUFS*" übergeben, der folgendes Aussehen haben könnte.

```
ringbufs( "Istgröße" , 2 , ( "Sollgröße" , 2 , ( "Stellgröße" , 1 , nil ) ) )     (4.10)
```

Als erstes wird geprüft, ob die Liste von Filtervariablen leer ist. Wenn nicht, wird die Liste gesplittet: in den Bezeichner "*Id*", in seine Verschiebung in die Vergangenheit "*Int*" und in eine Restliste. Mit "*Put*" kann man in Gentle die Zielsprache in eine Datei schreiben. Durch den rekursiven Aufruf mit der Restliste als Übergabeparameter dringt man immer tiefer in die Liste vom Typ "*RINGBUFS*" ein. "*Nil*" steht für das Ende der Liste und leitet das Rückspringen in der Liste und das Beenden des Prädikates ein.

5 Struktur der generierten Regelkreissystemsoftware

Der CLDL Übersetzer generiert aus dem CLDL Spezifikationsprogramm eine C++ Deklarationsdatei (Header) und eine C++ Definitionsdatei (Code). Beide Dateien tragen den Namen der Quelldatei. Die Headerdatei hat die Extension "h" und die Codedatei "cpp". Anhand dieser Extension erkennt der verwendete C/C++ Compiler, daß es sich um eine C++ Datei handelt. Im weiteren wird auf das nachfolgende Beispiel in Abbildung 5.1 Bezug genommen.

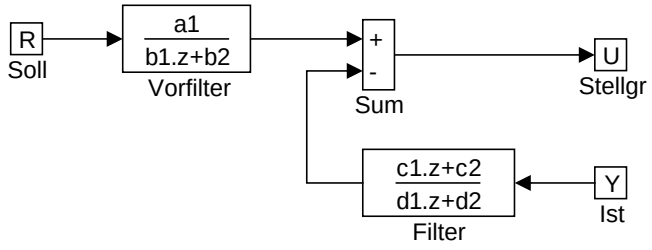


Abb. 5.1: Einfaches Regelkreisstrukturbild

Im Anhang B ist eine komplette C++ Deklarations- und Definitionsdatei dazu abgebildet.

5.1 Struktur der Deklarationsdatei

Die Headerdatei stellt die Schnittstellenspezifikation einer Klasse dar. Zunächst wird in der Deklarationsdatei eine Variable z.B. *__CLDL-Control_H* definiert. Der Name der Variablen kann dem Namen der Quelldatei entsprechen. Durch sie kann der Preprozessor eines Compilers erkennen, ob die Headerdatei schon eingebunden ist.
Jede generierte C++ Headerdatei besteht mindestens aus einer Klassendeklaration. Jede generierte Klasse besteht aus einem öffentlichen Teil (public) und aus einem versteckten oder gekapselten Teil (private). Unter public stehen alle Deklarationen, die nach außen hin sichtbar sind. Private steht für den nach außen hin nicht sichtbaren Teil einer Klasse.

5.1.1 öffentlicher Teil

In diesem Teil werden Typen deklariert, und der Konstruktor und einzelne Memberfunktionen vereinbart. In jeder Klasse werden immer die selben Typen und Memberfunktionen deklariert. Sie unterscheiden sich von Regelkreis zu Regelkreis nur in ihrem Inhalt. Es werden folgende Typdefinitionen durchgeführt, in denen vollständige Namenanalogie zum CLDL Quellprogramm herrscht.

```
Typedef enum {
    Soll, Ist, Stellgr
} yy_monitor;
```

Der Aufzählungstyp "*yy_monitor*" beinhaltet alle Größen im beschriebenen Regelkreis, die auf der linken Seite (LValue) einer Gleichung in der Regelkreisbeschreibungssprache stehen. Dieser Typ dient zur Referenzierung der einzelnen Zustandsgrößen. Verwendung findet "*yy_monitor*" in den Memberfunktionen "*Write*" und "*Monitor*".

```
typedef enum {
    a1, b1, b2, c1, c2, d1, d2
} yy_setvalue;
```

Dieser Aufzählungstyp "*yy_setvalue*" vereinigt alle Filterkoeffizienten und Konstanten der CLDL Datei. Er ermöglicht die geschlossene Übergabe an Funktionen zur späteren Unterscheidung seiner Elemente. Verwendet wird "*yy_setvalue*" in der Memberfunktion "*SetValue*".

```
typedef enum {
    VorFilter, Filter
} yy_filter;
```

Mit "*yy_filter*" kann zwischen verschiedenen Übertragungsfunktionen unterschieden werden. Er dient der Unterscheidung und Zusammenfassung der Filterdefinitionen. Dieser Typ wird in der Memberfunktion "*SetFilter*" benötigt.

```
typedef struct {
    double nominator[2];
    double denominator[2];
} yy_filter_struct;
```

Die Struktur "*yy_filter_struct*" beinhaltet die Komponenten Zähler ("*nominator*") und Nenner ("*denominator*"). Die Größe jedes Feldes (Polynoms) korrespondiert mit der jeweils maximalen Größe der Polynome aller Übertragungsfunktionen im CLDL Quellprogramm. Somit können mit dieser Struktur alle Filterkoeffizienten jeder Übertragungsfunktion erfaßt werden. Sie wird in der Memberfunktion "*SetFilter*" verwendet.

Ist bei einem Aufzählungstyp die Bezeichnerliste leer, erscheint dieser Typ auch nicht in der Zielcodedatei. Wenn man z.B. einen einfachen P-Regler beschreibt, benötigt man den Typ "*yy_filter_struct*" nicht, da keine Übertragungsfunktionen Verwendung finden.
Der Konstruktor der jeweils generierten Klasse besitzt den Namen dieser Klasse.
Danach folgen fünf Memberfunktionen, die das Arbeiten mit dieser Klasse ermöglichen. Sie können an beliebiger Stelle in der Steuerungssoftware aufgerufen werden. Die Funktionen realisieren auch indirekt den Zugriff auf den gekapselten Teil der Klasse.

- "*Write*" als erste Memberfunktion bildet die Eingabeschnittstelle zur Reglersoftware. Diese Funktion beschreibt die Variablen der Eingangsmeßgrößen. Es wird ein Element des Aufzählungstyps "*yy_monitor*" und der jeweilige Wert vom Typ "*double*" übergeben.
- Mit "*SetValue*" ist die Möglichkeit gegeben, bei laufender Software Filterkoeffizienten oder Konstanten zu verändern. Es muß ein Element des Aufzählungstyps "*yy_setvalue*" und der neue Wert vom Typ "*double*" übergeben werden.
- Die Funktion "*SetFilter*" realisiert ebenfalls einen Zugriff auf die laufende Reglersoftware. Es muß der Name des Filters und der Zeiger auf die Struktur vom Typ "*yy_filter_struct*", der die neuen Filterkoeffizienten enthält, übergeben werden.
- Der eigentliche Regelalgorithmus verbirgt sich hinter der Funktion "*Control*". Sie benötigt keine Eingabeparameter, da sie alle nötigen Informationen aus der Klasse selbst erhält. "*Control*" hat einen Rückgabewert vom Typ "*double*" und liefert die Stellgröße (Ausgangsgröße) des Reglers. Dieser Wert liegt zur Weiterverarbeitung für die Strecke bereit.
- "*Monitor*" liefert den aktuellen Wert vom Typ "*double*" des übergebenen Eingabeparameters zurück. Dieser Parameter ist ein Element vom Typ "*yy_monitor*".

5.1.2 gekapselter Teil

Im privaten Teil einer Headerdeklaration stehen alle von außen nicht zugänglichen Konstrukte einer Klasse. Das wirkt sich auf Sicherheit und Fehlertoleranz einer Software positiv aus. Es ist somit ein versehentlich falscher Zugriff auf Variablen im Vorfeld ausgeschlossen. Der gekapselte Teil spaltet sich in zwei Gruppen auf. Die erste Gruppe beinhaltet Konstanten- und Ringpufferfelddeklarationen und die zweite Gruppe deklariert zwei intern verwendete Memberfunktionen. Die Felderdeklarationen der ersten Gruppe sind in Existenz und Größe vom

CLDL Quellprogramm abhängig. Die Memberfunktionen der zweiten Gruppe hingegen existieren in jeder generierten Reglerklasse.
Es folgen einige mögliche Beispiele der ersten Gruppe.

```
double  yy_koeffs[7];

double  yy_Soll[2];
double  yy_Ist[2];
double  yy_Fehler[2];
double  yy_Stellgr[1];                                    (5.1)
```

Das Feld "*yy_koeffs*" enthält alle Filterkoeffizienten und Konstanten im Regelkreis, welche verifiziert werden können durch die Position im Aufzählungstyp "*yy_setvalue*". Der Name des Feldes ist in jeder generierten Reglerklasse gleich.
Die nachfolgenden Felder stellen die Ringpuffervariablen dar. Ihre Größe ist abhängig vom Grad der Polynome der verwendeten Übertragungsfunktionen und von der beschriebenen Regelkreisstruktur. Der Namensteil nach dem Unterstreichungszeichen korrespondiert mit dem jeweiligen Namen im CLDL Quellprogramm. Auch die Anzahl der Ringpuffervariablen ist bestimmt durch ihre Existenz in der Regelkreisbeschreibung.
In der zweiten Gruppe stehen folgende klasseninterne Funktionen, die dem Benutzer verborgen bleiben.

- "*InitBuffer*" initialisiert alle Ringpuffervariablen mit Null.
- Die Funktion " *RotBuffer*" schaltet die aktuelle Position im Ringpufferfeld weiter (Schrittweiterschaltung). Ihr übergibt man ein Element vom Typ "*yy_monitor*", um den richtigen Ringbuffer zu rotieren.

Beide Funktionen finden Verwendung in verschiedenen Memberfunktionen des öffentlichen Teils.

5.2 Struktur der Definitionsdatei

In der Definitionsdatei sind alle Memberfunktionen einer Klasse definiert. Am Anfang jeder Definitionsdatei ist mittels einer "*Include*"-Anweisung die eigene Deklarationsdatei eingebunden, um die Strukturen und Funktionen bekannt zu machen.

Nachfolgend sind alle Memberfunktionen der Reglerklasse "*Motor*" vorgestellt und kurz erläutert. Das komplette Listing dieses Beispiels findet man im Anhang B.

"*Motor :: Motor( )*"

Der Konstruktor jeder Klasse in C++ besitzt immer den selben Namen wie die Klasse selbst. In ihm werden im allgemeinen Initialisierungen und Voreinstellungen vorgenommen. Der Konstruktor lädt, die in der CLDL Beschreibung gesetzten Werte für Filterkoeffizienten und Konstanten, in das Feld "*yy_koeffs*". Die Reihenfolge der Elemente ist abhängig von der Beschreibung im CLDL Quellprogramm. Am Ende des Konstruktors werden alle Ringpuffervariablen mit "*0*" initialisiert. Zur Initialisierung findet eine extra Funktion "*InitBuffer*" Verwendung. Ein Teil des Konstruktors könnte folgendes Aussehen haben.

```
yy_koeffs[ d1]  =  1 ;
yy_koeffs[ d2]  =  0.196 ;
. . .
InitBuffer( ) ;                                                    (5.2)
```

"*void Motor :: Write(yy_monitor in_quantity , double value)*"

Die Funktion "*Write*" bildet die Eingabeschnittstelle des beschriebenen Regelkreises. Durch sie beschreibt man die aktuelle Position der Ringpuffervariablen mit neuen Werten. Die Variable "*in_quantity*" verifiziert den Ringbuffer und "*value*" enthält den neuen Wert. Mit der Mehrwegauswahl "*switch*" von C/C++ kann man die Aufgabe wie folgt bewältigen.

```
switch( in_quantity ) {
  case  Ist  :
    RotBuffer( Ist ) ;
    yy_y[ 0 ]  =  value ;
    break ;
. . .
  default :
    break ;
                                                                   (5.3)
```

Die Funktion "*RotBuffer*" rotiert das Feld und gibt die erste (aktuelle) Position für die nachfolgende Wertzuweisung frei. Mit "*default*" fängt man alle nicht beschriebenen Meßgrößen oder Fehleingaben für "*in_quantity*" ab. Es dürfen nur die Eingangsgrößen eines Regelkreises mit Werten geladen werden.

"*void Motor :: SetValue(yy_setvalue position , double value)*"

Um Filterkoeffizienten und Konstanten im laufenden Betrieb der Reglersoftware verändern zu können, ist die Funktion "*SetValue*" zur Verfügung gestellt. Sie realisiert das einzelne Verändern von Koeffizienten und veränderbaren Werten im beschriebenen Regelkreis. Danach muß eine Neuinitialisierung der Ringpuffervariablen vorgenommen werden, um eventuelle Instabilitäten zu vermeiden. Die komplette Memberfunktion hat folgendes Aussehen.

```
void Motor :: SetValue( yy_setvalue position , double value )
{
 yy_koeffs[ position ] = value ;
 InitBuffer( ) ;
}                                                                    (5.4)
```

"*position*" selektiert in "*yy_koeffs*" die gewünschte Position und "*value*" enthält den neuen Wert. Die Funktion "*InitBuffer*" ist weiter unten näher erläutert.

"*void Motor :: SetFilter(yy_filter filter , yy_filter_struct* filter_adr)*"

Auch "*SetFilter*" ermöglicht das Verändern von Werten bei laufender Software. Sie funktioniert in der gleichen Art und Weise wie "*SetValue*", wie der folgende Ausschnitt zeigt.

```
switch( filter ) {
 case VorFilter :
   yy_koeffs[ a1] = filter_adr -> nominator[ 0] ;
   yy_koeffs[ b1] = filter_adr -> denominator[ 0] ;
   yy_koeffs[ b2] = filter_adr -> denominator[ 1] ;
. . .
}
```

```
InitBuffer( ) ;
```
(5.5)

Mit "*switch*" wählt man auch hier die Übertragungsfunktion (Filter) laut der Variablen "*filter*" aus. "*filter_adr*" enthält den Zeiger auf die neue Filterstruktur vom Typ "*yy_filter_struct*". Es werden dann die neuen Filterkoeffizienten in das Feld "*yy_koeffs*" geladen. Anschließend ist auch hier "*InitBuffer*" aufzurufen, um unerwünschte nichtlineare Effekte zu vermeiden.

"*double Motor :: Control( )*"

Die Kernfunktion jeder Reglerklasse stellt "*Control*" dar. Sie beinhaltet den eigentlichen Regelalgorithmus in Form einer Differenzengleichung. "*Control*" liefert als Rückgabewert die aktuelle Stellgröße. Die Stellgröße (Ausgangsgröße) bildet den Reglerausgang und gleichzeitig den Streckeneingang. Stark vereinfacht, könnte die Funktion folgendes Aussehen haben.

```
RotBuffer( Stell ) ;
yy_Stell[ 0]  =
 yy_g[ 0] -
 ( yy_f[ 0] ) ;
if ( yy_Stell[ 0]  <  - 10 ) { yy_Stell[ 0]  =  - 10 ; }
if ( yy_Stell[ 0]  >  10 ) { yy_Stell[ 0]  =  10 ; }
return( yy_Stell[ 0] ) ;
```
(5.6)

Zuerst ist der aktuellste Wert des Ringpuffers "*yy_Stell*" einmal in die Vergangenheit zu schieben. Dadurch wird Platz für den neuen Wert im Puffer geschaffen. Der letzte Wert wird überschrieben, da er nicht mehr benötigt wird. Die "*if*"-Anweisung wird nur generiert, wenn im CLDL Quellprogramm die "*LIMITS*"-Funktion Verwendung findet. Zum Schluß liefert die Funktion die neue Stellgröße (Ausgangsgröße des Reglers) zurück.

"*double Motor :: Monitor(yy_monitor monitor)*"

Die Funktion "*Monitor*" liefert immer den momentan aktuellsten Wert eines Elements vom Typ "*yy_monitor*" zurück. Sie kann zu Überwachungszwecken bei laufender Systemsoftware eingesetzt werden.

```
switch( monitor ) {
  case  Ist  :  return( yy_Ist[ 0] ) ;
  case  Stell  :  return( yy_Stell[ 0] ) ;
. . .
}
```
(5.7)

Der kurze Ausschnitt zeigt, wie man wieder mit "*switch*" die Elemente eines Aufzählungstyps verarbeiten kann. Mit der Funktion "*Monitor*" wird ein Mittel zur Diagnose und Fehlerüberwachung zur Verfügung gestellt.

"*void Motor :: InitBuffer( )*"

"*InitBuffer*" initialisiert die Ringpuffervariablen mit dem Wert "*0*". Sie findet Verwendung, wo überall durch Wertzuweisung Instabilitäten auftreten können. Das sind die Memberfunktionen "*Motor*", "*SetValue*" und "*SetFilter*". Eine Initialisierung könnte wie folgt aussehen.

```
yy_Ist[ 1]  =  yy_Ist[ 0]  =  0 ;
yy_Soll[ 1]  =  yy_Soll[ 0]  =  0 ;
yy_Stell[ 0]  =  0 ;
```
(5.8)

Es werden hier Mehrfachzuweisungen verwendet, um einen effektiveren Code zu erhalten.

"*void Motor :: RotBuffer(yy_monitor buffer)*"

Mit "*RotBuffer*" als letzte Funktion in einer Reglerklasse werden die Elemente im Feld um eine Position 'nach hinten' oder besser in die Vergangenheit verschoben. Der letzte Wert im Puffer (Feld) wird nicht mehr benötigt und geht verloren. "*RotBuffer*" findet Verwendung in den Funktionen "*Write*" und "*Control*". Ein Teil des Funktionsrumpfes sieht für unser Beispiel aus Abbildung 5.1 folgendermaßen aus.

```
switch( buffer ) {
  case  Ist  :
    yy_Ist[ 1]  =  yy_Ist[ 0] ;
    break ;
. . .
  case  Stell  :
    break ;
}
```
(5.9)

Die Variable "*buffer*" vom Typ "*yy_monitor*" unterscheidet die einzelnen Meßgrößen im Regelkreis. Hat ein Ringpufferfeld nur ein Element, braucht nicht rotiert zu werden.

Bei der Implementierung wurde auf einfache und klare Strukturen Wert gelegt. Durch den Generierungsvorgang werden einmal erstellte Strukturen immer wieder neu in der selben Qualität mit unterschiedlichen Daten gefüllt. Das ist im hohen Grade effizient und fehlervermeidend. In den einzelnen Memberfunktionen ist bei der Auswahl der C/C++ Anweisungen auf Geschwindigkeit, Codegröße und Effizienz Wert gelegt worden. Dies ist wichtig, da Reglersoftware strengen Realzeitanforderungen unterliegt.

6 Verifikation und Validierung der generierten Regelkreissystemsoftware

In diesem Abschnitt sind einige Beispiele der Reglermodulentwicklung aufgeführt. Diese zeigen, daß der Generator korrekt arbeitende Reglersoftware erzeugt.
Der Arbeitsablauf beim Testen des Übersetzers kann wie folgt zusammengefaßt werden. Die kursiv geschriebenen Dateinamen sind anwenderspezifisch und alle großgeschriebenen Wörter sind Eigennamen.

1. Regelkreisbeschreibung in eine Datei mit der Extension CDL schreiben.
2. Den Übersetzer CLDL.EXE mit der Datei *Motor*.CDL aus Punkt 1 als Kommandozeilenparameter starten.

- Es werden die Dateien *Motor*.H und *Motor*.CPP generiert. Beide Dateien kann man nun in eine Steuerungssoftware integrieren.

3. Die Datei CPPMEX.BAT [14] von Matlab ist mit dem Schnittstellenmodul zu Matlab/SIMULINK *CPP_Motor*.CPP als ersten und der generierten Definitionsdatei *Motor*.CPP als zweiten Kommandozeilenparameter aufzurufen.

- Es wird eine Datei mit der Extension MEX erzeugt.

4. In SIMULINK kann nun die entstandene MEX-Funktion (S-Funktion) aus Punkt 3 als Simulinkblock eingebunden werden [14].

Die Datei CPPMEX.BAT für alle Beispiele ist dem Anhang D zu entnehmen.
Alle folgenden Reglerentwürfe sollen einem Pendel mit Reibung ein gutes oder ausreichendes Regelkreisverhalten geben. Es wird gleichzeitig gezeigt, daß einige Reglerstrukturen ungeeignet zur Regelung dieser Strecke sind. Die allgemeine kontinuierliche Übertragungsfunktion des Pendels im S-Bereich lautet.

$$G_{Pendel}(s) = \frac{\omega_n{}^2}{s^2 + 2\zeta\omega_n s + \omega_n{}^2} \tag{6.1}$$

Es wird $\omega_n{}^2 = 404$ und $2\zeta\omega = 4$ gesetzt.

6.1 Einfache Regelkreisstruktur

Bei einem Eingrößenregelkreis ohne weitere Meßfilter ist die Führungsgröße (Soll) mit der Regelgröße (Ist) zu subtrahiert, also der Fehler zu bilden. Die Fehlergröße ist der Eingang des Reglerblocks. Der Ausgang des Reglers wird als Stellgröße (Stell) bezeichnet und bildet den Eingang der Strecke.

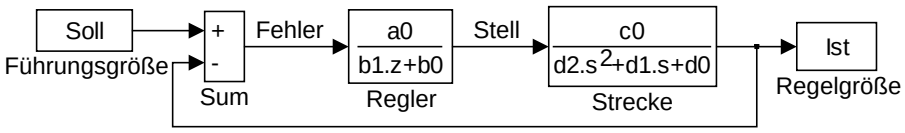


Abb. 6.1: Allgemeine Eingrößenregelkreisstruktur

6.1.1 P-Regler

Der einfachste Regler zum Regeln einer Strecke ist ein P-Glied. Das Proportionalglied kann als idealer Verstärker angesehen werden und hat die Übertragungsfunktion G(s) = V.
In CLDL sieht der P Regler folgendermaßen aus.

```
CONTROL  P_Regler ;
 V = 1.5 ;

 IN( Soll ) ;
 IN( Ist ) ;
 Fehler = Soll - Ist ;
 Stell = V * Fehler ;
 OUT( Stell ) ;
END ;                                                    (6.2)
```

Der Regelkreis ist in Abbildung 6.2 im Vergleich mit der generierten "*P_Regler*"-Klasse dargestellt. Der Simulinkblock "*cpp_p*" stellt eine S-Funktion dar. In ihr wird eine gleichnamige Datei mit der Extension MEX aufgerufen. Sie ist durch Compilieren der Schnittstellendatei zu SIMULINK und der generierten "*P_Regler*"-Klasse entstanden. Jede S-Funktion hat nur einen Eingang und einen Ausgang. Bei mehreren Ein- und/oder Ausgängen sind diese per Multiple-

xer und/oder Demultiplexer zu vektorisieren. Zusätzlich werden verschiedene aktuelle Werte in den Arbeitsraum von Matlab ("*To Workspace*") geschrieben, um später die genauen Werte miteinander vergleichen zu können.

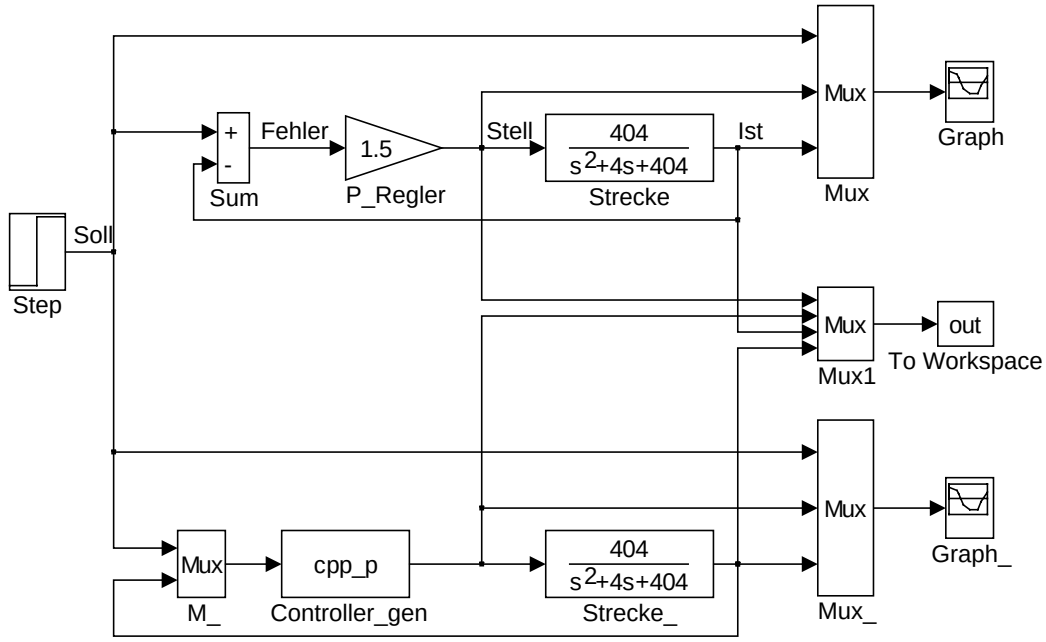


Abb. 6.2: P-Glied im direkten Vergleich mit der dazugehörenden generierten Reglerklasse

Beide Systemausgänge zeigen auf einen Eingangssprung folgendes Verhalten.

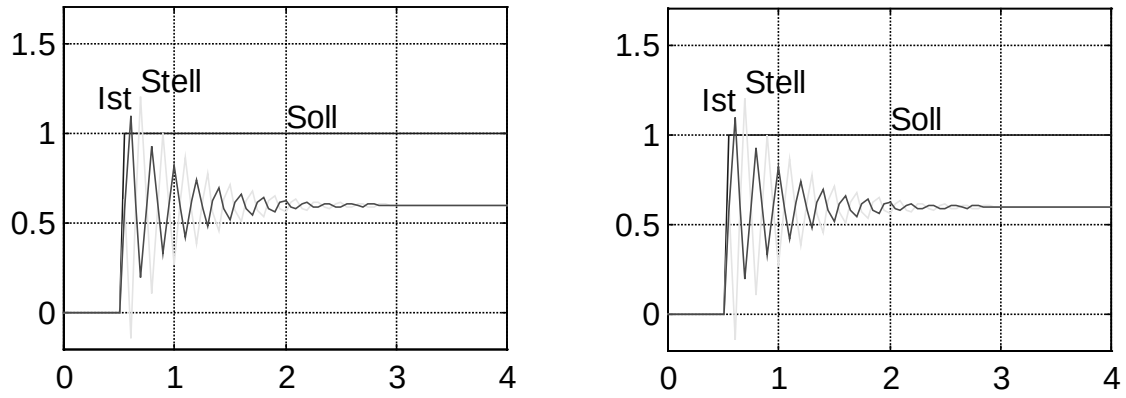


Abb. 6.3: Soll-, Ist- und Stellgröße der Regelkreisstrukturen mit P-Glied

In der Abbildung 6.3 ist zu erkennen, daß der P-Regler nicht zu einem guten Regelergebnis führt. Der Stellgrößenverlauf ist dem Regelgrößenverlauf (Ist) entgegengerichtet und hat eine

leicht größere Amplitude. Man sieht, daß das Pendel nach ca. 3 Sekunden eingeschwungen ist, und sich ein bleibender Lagefehler einstellt. Beide Regelkreise zeigen identisches Verhalten der Ist- und Stellgröße.
Der folgende Auszug der genauen Werte untermauert die Gleichwertigkeit der beider Methoden.

	Stellgröße		Istgröße	
	Simulinkblock	generierter Regler	Simulinkblock	generierter Regler
1	0.0000	0.0000	0.0000	0.0000
2	0.6388	0.6388	0.5741	0.5741
3	-0.1387	-0.1387	1.0925	1.0925
4	0.5895	0.5895	0.6070	0.6070
5	1.2033	1.2033	0.1978	0.1978
6	0.5915	0.5915	0.6057	0.6057
7	0.1073	0.1073	0.9285	0.9285
8	0.6206	0.6206	0.5863	0.5863
9	1.0021	1.0021	0.3319	0.3319
10	0.5720	0.5720	0.6187	0.6187

Tabelle 6.1: Aktuelle Werte der Systemgrößen bei einem P-Regler

6.1.2 PI-Regler

Als zweiter Regler wird ein PI-Regler vorgestellt. Das PI-Glied hat im S-Bereich die Übertragungsfunktion $G(s) = K(1 + \frac{1}{T_n s})$. Die Dimensionierung des Reglers ist anhand des Wurzelortes erfolgt. Die Transformation vom S-Bereich in den Z-Bereich wurde von der Matlabfunktion "*c2dm*" übernommen. Es hat sich eine Abtastzeit von 50ms als ausreichend erwiesen. In der Beschreibungssprache CLDL sieht dieser PI-Regler im zeitdiskreten Bereich folgendermaßen aus.

```
CONTROL  PI_Regler ;
  a1 = 0.012 ;
  a2 = 0.0272 ;
  a3 = 0.0028 ;
```

```
b1 = 1 ;
b2 = -1.9778 ;
b3 = 1.7965 ;
b4 = -0.8187 ;
PI_Glied = [ a1 , a2 , a3 ] / [ b1 , b2 , b3 , b4 ] ;

IN( Soll ) ;
IN( Ist ) ;
Fehler = Soll - Ist ;
Stell = PI_Glied * Fehler ;
OUT( Stell ) ;
END ;                                                    (6.3)
```

Die generierte "*PI_Regler*"-Klasse ist wieder mit dem modifizierten Schnittstellenmodul zu Matlab/SIMULINK in eine MEX-Datei zu übersetzen, um sie als S-Funktion in SIMULINK einbinden zu können.

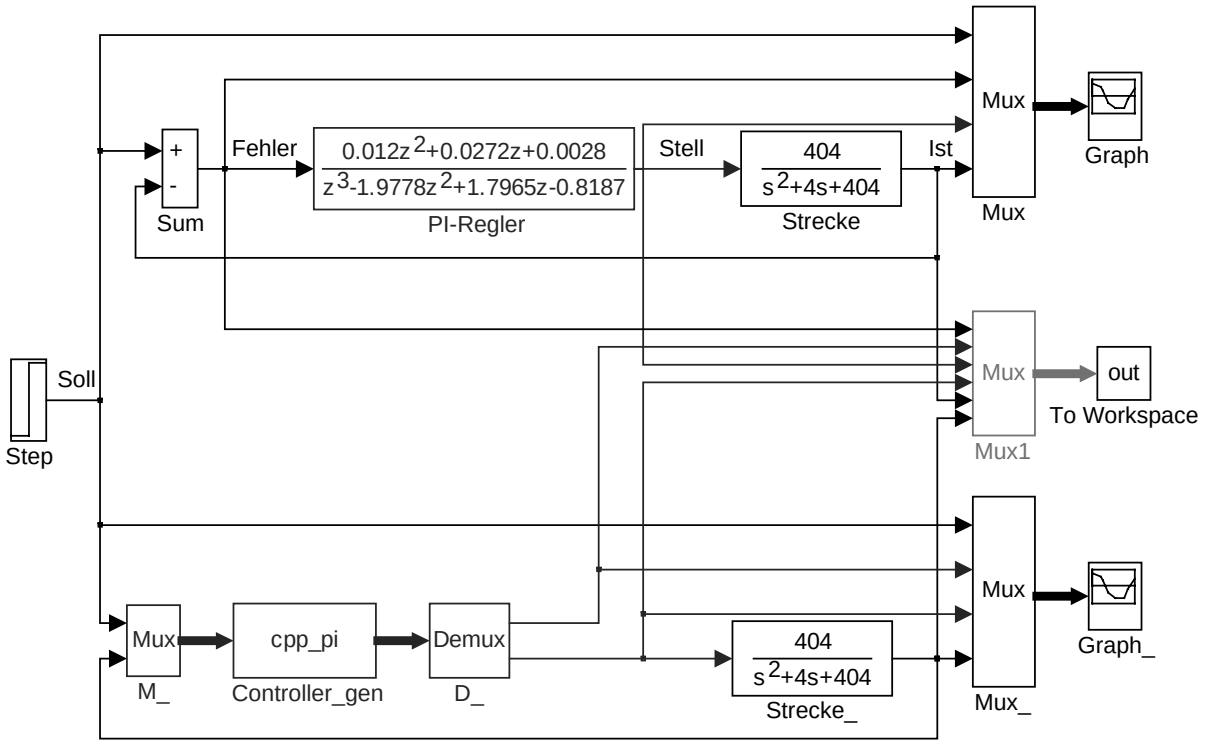
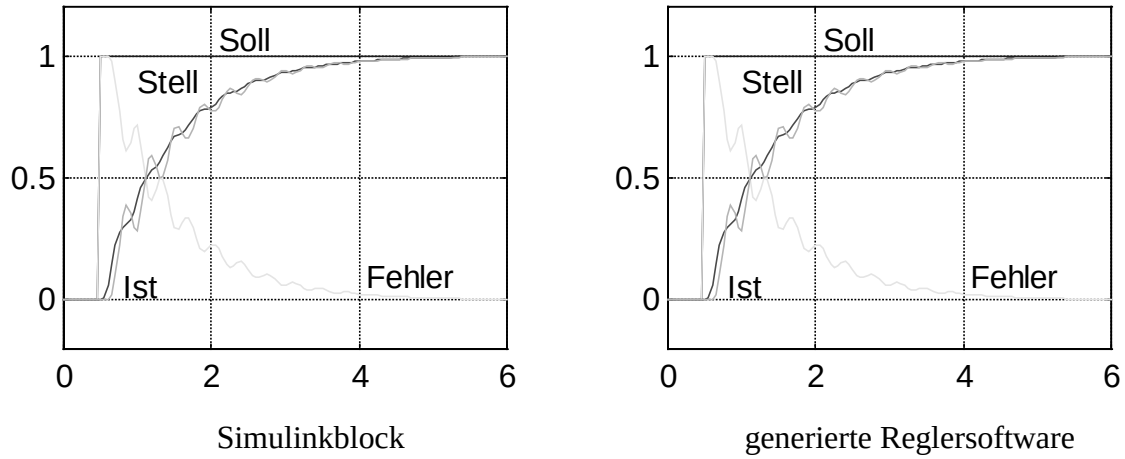


Abb. 6.4: PI-Regler im direkten Vergleich mit der dazugehörenden generierten Reglerklasse



Simulinkblock generierte Reglersoftware

Abb. 6.5: Soll-, Ist-, Fehler- und Stellgröße der Regelkreisstrukturen mit PI-Glied

Der verwendete PI-Regler zeigt hier schon ein wesentlich besseres Ausgangsverhalten als der P-Regler. Ein Lagefehler wird durch den I-Anteil ausgeregelt und der stationäre Endwert nach ca 5s erreicht. Zu bemängeln ist hier das schwingende Einlaufverhalten der Istgröße (Führungsgröße). Eine PI-Reglerstruktur führt somit auch nicht zu einem einwandfreien Regelverhalten.

Der direkte Vergleich der aktuellen Werte zeigt erneut deutlich die Richtigkeit der generierten Reglersoftware.

	Fehlergröße		Stellgröße		Istgröße	
	Simulinkblock	generierter Regler	Simulinkblock	generierter Regler	Simulinkblock	generierter Regler
1	0	0	0	0	0	0
2	1.0000	1.0000	0	0	0	0
3	1.0000	1.0000	0.0120	0.0120	0	0
4	1.0000	1.0000	0.0629	0.0629	0	0
5	0.9764	0.9764	0.1449	0.1449	0.0236	0.0236
6	0.9019	0.9019	0.2251	0.2251	0.0981	0. 0981
7	0.7773	0.7773	0.2766	0.2766	0.2227	0.2227
8	0.6594	0.6594	0.2978	0.2978	0.3406	0.3406
9	0.6105	0.6105	0.3081	0.3081	0.3895	0.3895
10	0.6441	0.6441	0.3281	0.3281	0.3559	0.3559

Tabelle 6.2: Aktuelle Werte der Systemgrößen Fehler-, Stell- und Istgröße mit PI-Regler

6.2 Adaptive Regelkreisstruktur

Bei einer adaptiven Regelkreisstruktur führt man Führungs- und Regelgröße erst über einen Filter und subtrahiert dann. Im Vorfilter als auch im Filter tritt der gleiche Nenner auf. "*Rho*" steht für den Verstärkungsfaktor des Reglers.

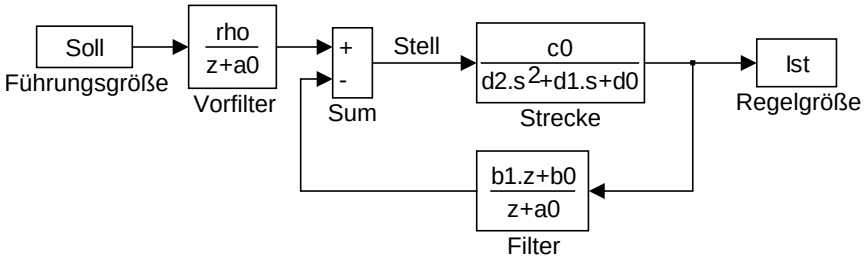


Abb. 6.6: Adaptive Regelkreisstruktur

Beide nun folgenden Regler sind mit Hilfe der Diophantischen (Entwurfs-) Gleichung [27]

$$\Delta_B \Delta_R = Z_S Z_C + \Delta_S \Delta_C \tag{6.4}$$

entworfen worden. Die einzelnen Symbole und Zeichen bedeuten;

- " Δ " steht für den Nenner und
- "*Z*" steht für den Zähler einer Übertragungsfunktion,
- "*B*" wie Beobachter und
- "*S*" wie Strecke,
- "*R*" gilt als Synonym für den geschlossenen Regelkreis und
- "*C*" repräsentiert den Regler.

Wenn man nun ein System 2. Ordnung regeln möchte, muß laut [27] ein Regler 1. Ordnung gewählt werden, da dieser sich positiver auf die spätere Stellgröße auswirkt. Weiter wählt man einen zeitoptimalen Beobachter, der alle Eigenwerte bei Null hat, und auch 1. Ordnung sein muß. Somit lautet der allgemeine Regleransatz

$$G(c) = \frac{b_1 z + b_0}{z + a_0} \ . \tag{6.5}$$

Es werden nun im weiteren zwei verschiedene diskrete dynamische Zustandsregler mit zeitoptimalem Beobachter vorgestellt.

6.2.1 VZ2-Verhalten

Es wird für das geschlossene System VZ2-Verhalten verlangt, d.h. $\Delta_R = (z - 0.5)^2$. Man setzt alle erforderlichen Teile in die Entwurfsgleichung (6.4) ein und löst nach den unbekannten Reglerparametern auf. Diese relativ aufwendige Arbeit kann wieder von Matlab erledigt werden. Die Streckenübertragungsfunktion muß in den Z-Bereich transformiert werden, da die Diophantische Gleichung im zeitdiskreten Zustandsbereich verwendet wird.
Die Reglerstruktur kann in CLDL wie folgt formuliert werden.

```
CONTROL  VZ2_Regler ;
 rho = 0.35 ;
 a1 = 1 ;
 a0 = 0.196 ;
 VorFilter = [ rho ] / [ a1 , a0 ] ;
 b1 = -0.512 ;
 b0 = -0.336 ;
 Filter = [ b1 , b0 ] / [ a1 , a0 ] ;

 IN( Soll ) ;
 IN( Ist ) ;
 g = VorFilter * Soll ;
 f = Filter * Ist ;
 Stell = g - f ;
 OUT( Stell ) ;
END ;
```
(6.6)

Die generierte Deklarations- und Definitionsdatei für dieses Beispiel ist im Anhang B, die Schnittstellendatei zu Matlab/SIMULINK im Anhang C und die Batchdatei zum Erstellen der MEX-Datei im Anhang D zu finden.

Nachdem die generierte Deklarations- und Definitionsdatei in das MEX-Format umgewandelt ist, kann man sie in SIMULINK als S-Funktionblock einbinden.

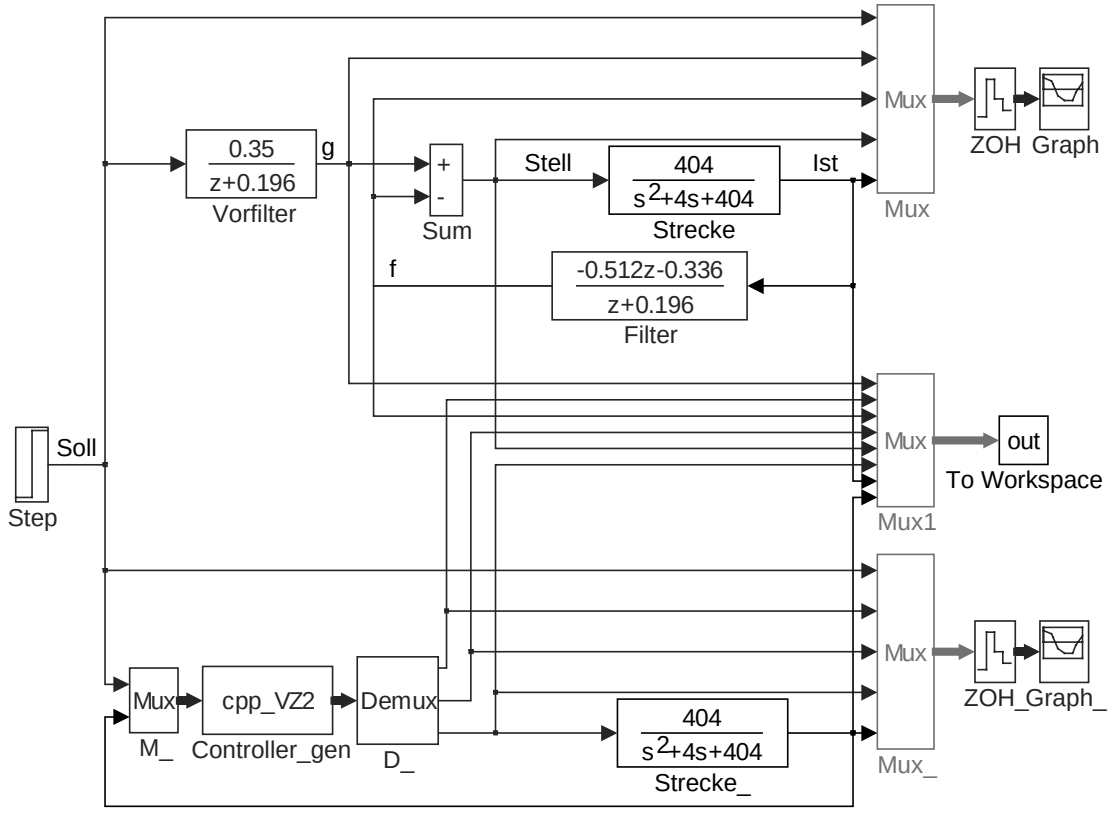


Abb. 6.7: VZ2-Verhalten im direkten Vergleich mit dem dazugehörenden generierten Regler

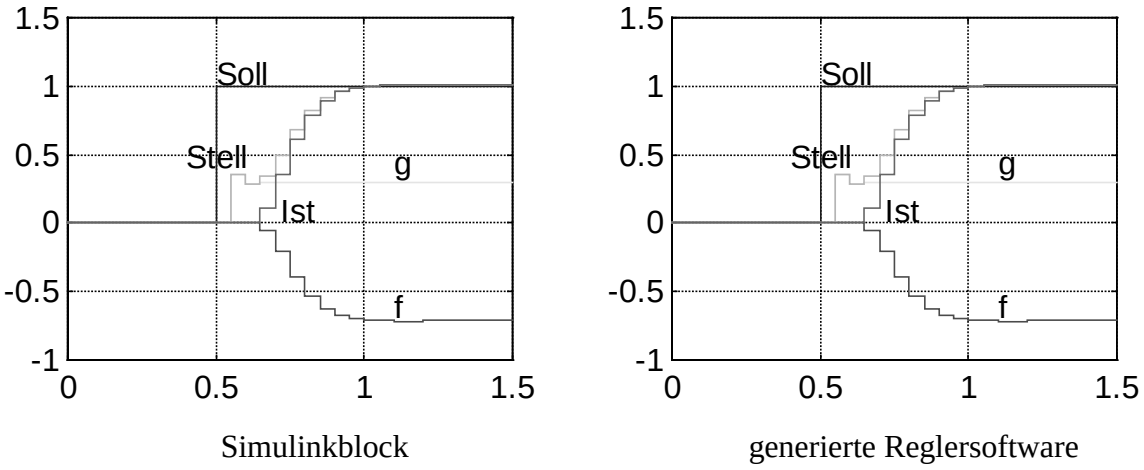


Abb. 6.8: Soll-, Ist-, Stell- und Zwischengrößen der Regelkreisstrukturen mit VZ2-Verhalten

Zu sehen ist, daß auch bei einer anderen Regelkreisstruktur beide Systemausgänge gleiches Verhalten zeigen. Die Regelgröße (Ist) zeigt VZ2-Verhalten, so wie laut Aufgabenstellung verlangt. Die zeitlichen Verschiebungen der Systemgrößen zur Führungsgröße (Soll) ist bei diesem Reglertyp besonders gut zu erkennen. Sie entstehen durch die Verzögerungsglieder erster und zweiter Ordnung. Ein nachgeschaltetes Abtast-Halte-Glied bewirkt den treppenartigen Verlauf der einzelnen Größen in Abbildung 6.8. Wie auch die Graphen, zeigen die aktuellen Werte der Monitorvariablen die Richtigkeit und die Genauigkeit der generierten Reglersoftware.

	g		f		Stellgröße		Istgröße	
	Simulink-block	generierter Regler	Simulink-block	generierter Regler	Simulink-block	generierter Regler	Simulink-block	generierter Regler
1	0	0	0	0	0	0	0	0
2	0.3500	0.3500	0	0	0.3500	0.3500	0	0
3	0.2814	0.2814	0	0	0.2814	0.2814	0	0
4	0.2948	0.2948	-0.0540	-0.0540	0.3488	0.3488	0.1055	0.1055
5	0.2922	0.2922	-0.2086	-0.2086	0.5008	0.5008	0.3589	0.3589
6	0.2927	0.2927	-0.3942	-0.3942	0.6870	0.6870	0.6143	0.6143
7	0.2926	0.2926	-0.5340	-0.5340	0.8266	0.8266	0.7907	0.7907
8	0.2926	0.2926	-0.6205	-0.6205	0.9132	0.9132	0.8975	0.8975
9	0.2926	0.2926	-0.6699	-0.6699	0.9625	0.9625	0.9569	0.9569
10	0.2926	0.2926	-0.6958	-0.6958	0.9885	0.9885	0.9875	0.9875

Tabelle 6.3: Aktuelle Werte der Systemgrößen vom Regelkreis mit VZ2-Verhalten

6.2.2 Dead-Beat-Verhalten

Das letzte Beispiel versucht dem Pendelmodell Dead-Beat-Verhalten aufzuzwingen. Das heißt, der stationäre Endwert muß in zwei Schritten erreicht werden. Ein Dead-Beat-Regler ist schlecht zu realisieren, da er eine sehr hohe Stellgröße liefert, die die angeschlossene Strecke oft nicht verkraftet. Für theoretische Betrachtungen eignet er sich aber besonders gut, da die Schrittzahl bis zum Erreichen des stationären Endwertes leicht zählbar ist.
Soll der geschlossene Regelkreis Dead-Beat-Verhalten zeigen, müssen alle Eigenwerte im Z-Bereich im Ursprung der Z-Ebene, also bei Null liegen. Somit ist $\Delta_R = z^2$; der zeitoptimale

Beobachter lautet $\Delta_B = z$ und ist wie der Regler von 1. Ordnung. Für den Regleransatz wird wieder die allgemeine Form (6.5) verwendet.

In der Beschreibungssprache CLDL kann dieser Regler so formuliert werden.

```
CONTROL  DB_Regler ;
 rho = 1.19 ;
 a1 = 1 ;
 a0 = 0.624 ;
 VorFilter = [ rho ] / [ a1 , a0 ] ;
 b1 = 0.828 ;
 b0 = -1.257 ;
 Filter = [ b1 , b0 ] / [ a1 , a0 ] ;

 IN( Soll ) ;
 IN( Ist ) ;
 g = VorFilter * Soll ;
 f = Filter * Ist ;
 Stell = g - f ;
 OUT( Stell ) ;
END ;
```
(6.7)

Mit CLDL.EXE sind die C++ Dateien zu generieren. Anschließend erzeugt wieder CPPMEX.BAT eine MEX-Datei für SIMULINK.

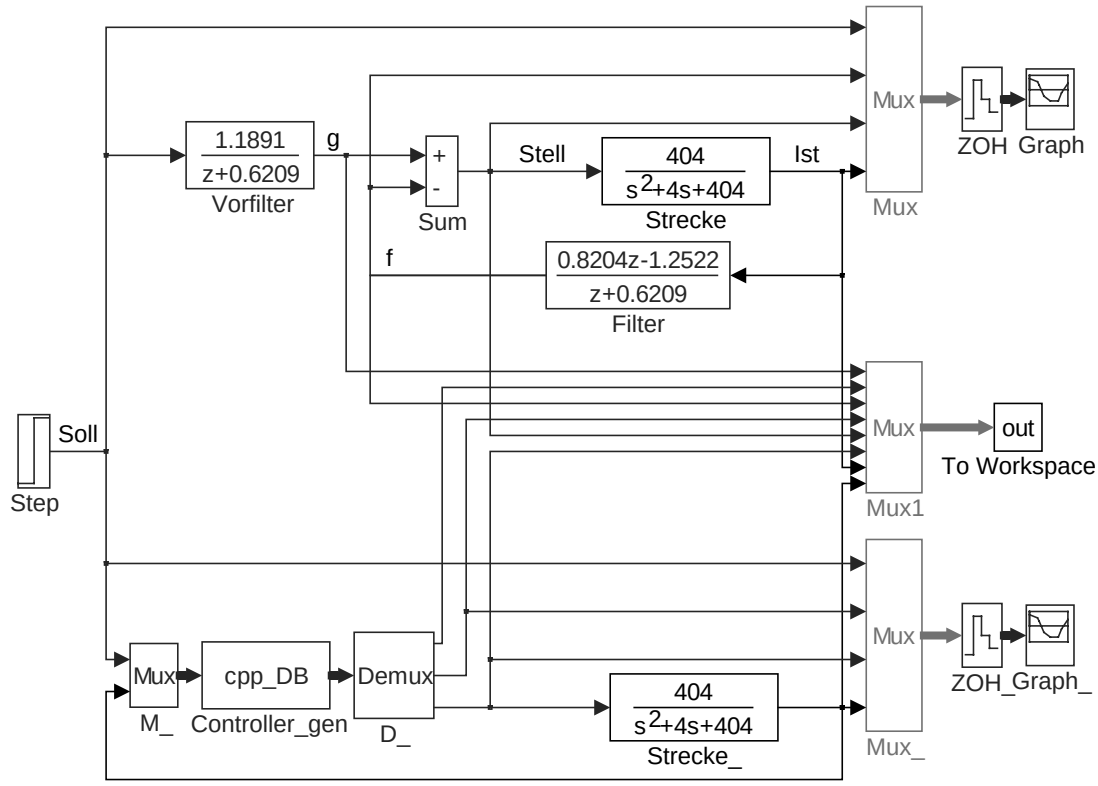


Abb. 6.9: Dead-Beat-Verhalten im Vergleich mit dem dazugehörenden generierten Regler

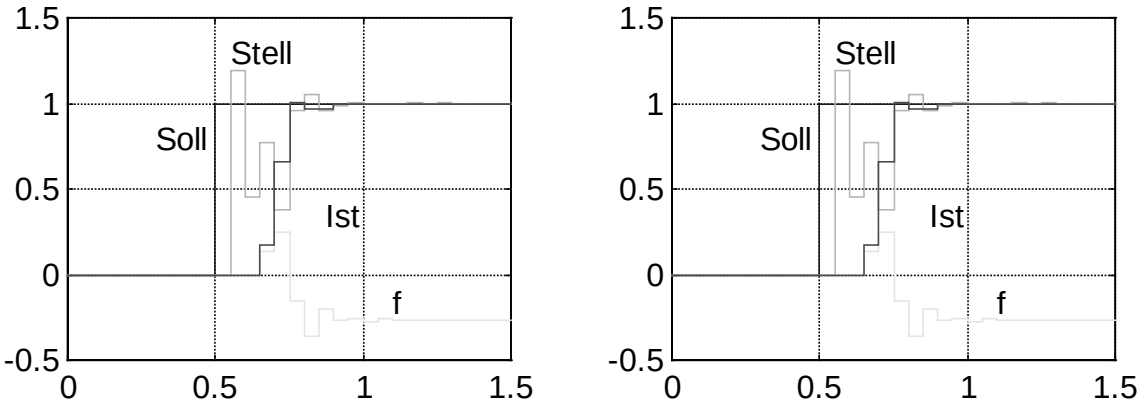


Abb. 6.10: Systemgrößen der Regelkreisstrukturen mit Dead-Beat-Verhalten

Durch das Nachschalten eines Haltegliedes 1. Ordnung sind die äquidistanten Schrittfolgen der unterschiedlichen Systemgrößen klar zu erkennen. Man erkennt auch, daß die Regelgröße (Ist) nicht genau in zwei Schritten ihren stationären Endwert erreicht. Da die Abweichungen in beiden Kurven identisch sind, kann es nicht an einer fehlerhaften generierten Reglersoft-

ware liegen. Vielmehr ist wohl der Reglerentwurf selbst Ursache für das nicht optimale Verhalten im Bezug auf die Vorgaben.
Der nachfolgende Auszug der aktuellen Werte der Systemgrößen verdeutlicht die Übereinstimmung der generierten Reglersoftware mit der Regelkreissimulation unter Matlab/SIMULINK.

	g		f		Stellgröße		Istgröße	
	Simulink-block	generierter Regler	Simulink-block	generierter Regler	Simulink-block	generierter Regler	Simulink-block	generierter Regler
1	0	0	0	0	0	0	0	0
2	1.1891	1.1891	0	0	1.1891	1.1891	0	
3	0.4508	0.4508	0	0	0.4508	0.4508	0	
4	0.9092	0.9092	0.1386	0.1386	0.7706	0.7706	0.1690	0.1690
5	0.6246	0.6246	0.2449	0.2449	0.3796	0.3796	0.6614	0.6614
6	0.8013	0.8013	-0.1576	-0.1576	0.9589	0.9589	1.0028	1.0028
7	0.6916	0.6916	-0.3613	-0.3613	1.0529	1.0529	0.9710	0.9710
8	0.7597	0.7597	-0.1987	-0.1987	0.9584	0.9584	0.9665	0.9665
9	0.7174	0.7174	-0.2703	-0.2703	0.9877	0.9877	0.9953	0.9953
10	0.7437	0.7437	-0.2587	-0.2587	1.0024	1.0024	0.9992	0.9992

Tabelle 6.4: Aktuelle Werte der Systemgrößen vom Regelkreis mit Dead-Beat-Verhalten

6.3 Einsatz in einer Robotersteuerung

Um die generierten Reglermodule auch in einer kompletten Steuerung zu testen, sind sie in eine Experimentalsteuerung mit dem Namen Sokrates integriert worden. Sokrates ist eine PC-basierte Steuerung unter Windows 95 mit generierten Softwarekomponenten.
Sie läuft auf einem INTEL Pentium 586 mit 100 MHz und einer Abtastzeit von 1 ms. Die Softwarearchitektur ist der nachfolgenden Abbildung 6.11 zu entnehmen.

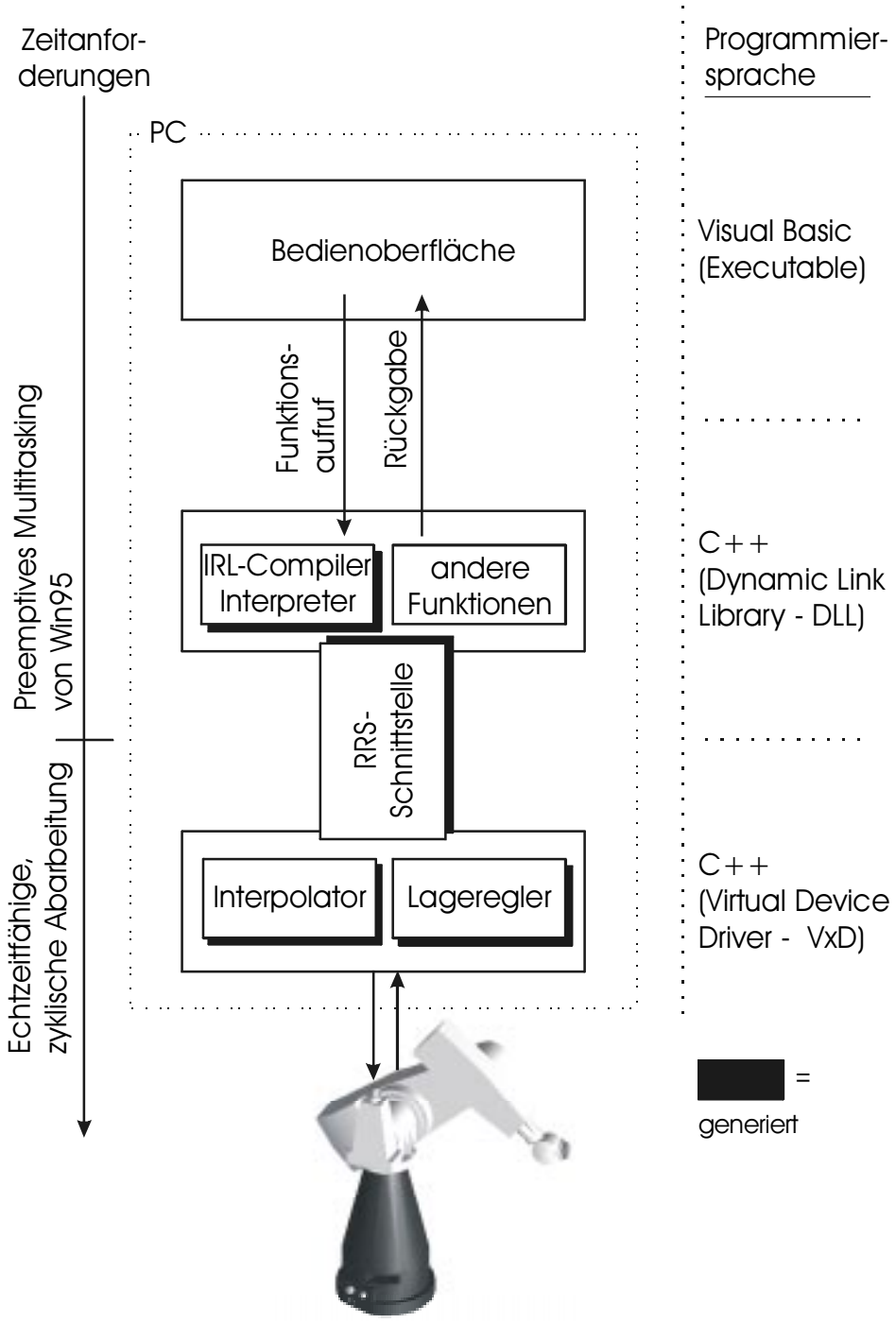


Abb. 6.11: Softwarearchitektur der PC-basierten Robotersteuerung Sokrates [28]

Die vom CLDL Übersetzer generierten Softwaredateien sind ohne Veränderungen und Anpassungen als Lagereglermodul in die Steuerung integriert worden. Die Steuerung ist in C++ entwickelt und ermöglicht so ein schnelles Einbinden von C++ Dateien. Um eine Vielzahl an Teilmodulen leicht integrieren zu können, müssen klare Schnittstellen geschaffen werden.

Eine Schnittstellenspezifikation gewährleistet ein schnelles und reibungsloses Einbinden verschiedener generierter Softwareteile.

Soll der Lageregler verändert werden, ist die generierte Reglerklasse neu zu erstellen. Man beschreibt den neuen Regler in CLDL und übersetzt ihn in die Zielsprache C++. Da alle Schnittstellen zu diesem Modul automatisch generiert werden, ist nur die alte durch die neue Reglerklasse zu ersetzen. Die Steuerung arbeitet nun ohne weitere Anpassung der Software mit dem neuen Regler zusammen.

Der folgende Programmauszug zeigt die Schnittstelle, an der die Kommunikation mit der generierten Reglerklasse erfolgt.

```
class  Motor  VZ2_Regler_Achse1;
class  Motor  VZ2_Regler_Achse2;
class  Motor  VZ2_Regler_Achse3;

void Interrupthandler( )
{
 //************************
 // L A G E R E G E L U N G
 //************************

 // Achse1
 Achse1.LageIstAlt.soft  =  Achse1.LageIst.soft ;
 HolePositionAchse1( ) ;
 VZ2_Regler_Achse1.Write( VZ2_Regler_Achse1.Ist , Achse1.LageIst.soft ) ;
 VZ2_Regler_Achse1.Write( VZ2_Regler_Achse1.Soll ,Achse1.LageSoll.soft ) ;
 PWMAchse1( VZ2_Regler_Achse1.Control( ) ) ;

 // Achse2
 Achse2.LageIstAlt.soft  =  Achse2.LageIst.soft ;
 HolePositionAchse2( ) ;
 VZ2_Regler_Achse2.Write( VZ2_Regler_Achse2.Ist , Achse2.LageIst.soft ) ;
 VZ2_Regler_Achse2.Write( VZ2_Regler_Achse2.Soll , Achse2.LageSoll.soft ) ;
 PWMAchse2( VZ2_Regler_Achse2.Control( ) ) ;

 // Achse3
 Achse3.LageIstAlt.soft  =  Achse3.LageIst.soft ;
 HolePositionAchse3( ) ;
 VZ2_Regler_Achse3.Write( VZ2_Regler_Achse3.Ist , Achse3.LageIst.soft ) ;
```

```
  VZ2_Regler_Achse3.Write( VZ2_Regler_Achse3.Soll , Achse3.LageSoll.soft ) ;
  PWMAchse3( VZ2_Regler_Achse3.Control( ) ) ;

. . .
}
```
(6.8)

Es wird für jede Achse eine Variable vereinbart, über die die generierte Reglerklasse angesprochen wird. Durch den Zugriff über den Klassenamen verhindert man ein falsches oder fehlerhaftes Verwenden von Variablen.
In diesem Beispiel (6.8) werden alle drei Achsen eines Mitsubishi Roboters Melfa RV-M1 mit dem selben Regler gesteuert. Der Roboter fährt jede PTP (Point To Point) Bewegung ohne bleibenden Lagefehler an.
Anhand dieses Beispiels aus der Praxis, zeigt sich deutlich, daß generierte Software harte Realzeitanforderungen erfüllen kann. Dies wird besonders durch effektiv arbeitende Softwarestrukturen unterstützt.

7 Zusammenfassung und Ausblick

Der bis zu diesem Zeitpunkt fertig gestellte Übersetzer umfaßt nur einen kleinen Teil des Sprachumfanges, den ein kommerzielles System haben kann. Es sollte hier der Beweis erbracht werden, daß es möglich ist, durch einfache Regelkreisbeschreibung mittels eines Übersetzers C++ Code zu erzeugen. Es wird also der aufwendige und fehleranfällige Teil der Berechnung eines Regelalgorithmus vom Generator übernommen.

Der Übersetzer ist so entwickelt worden, daß es im nachhinein relativ leicht möglich ist, Spracherweiterungen vorzunehmen. Die Pässigkeit ermöglicht weiterhin das komplette Austauschen des Codegenerators und des Teils, der für das Einlesen der CLDL Quellsprache des Übersetzers verantwortlich ist. Damit wird die Wiederverwendbarkeit von Softwareteilen wesentlich erhöht.

Der verwendete prototypische Compilergenerator Gentle erwies sich als zuverlässiges Entwicklungswerkzeug. Das Prädikatenkalkül von Gentle ist mächtig genug, alle erforderlichen Übersetzungen zuzulassen und zu unterstützen. Es mußte daher nur selten auf selbst geschriebene C Funktionen zurückgegriffen werden.

Durch diese Arbeit wird das Entwicklungskonzept einer Steuerung, basierend auf generierter Software, wirkungsvoll unterstützt. Sie beweist die Möglichkeit der automatischen Umsetzung von der Beschreibungssprache zum realzeitfähigen, korrekten und ausgereiften C++ Code. Dieses Reglersoftwaremodul beinhaltet den Regelalgorithmus und verschiedene Servicefunktionen zum Überwachen und Verändern von Regelkreiskomponenten im laufenden Betrieb der Steuerung. Die generierte Reglersoftware wurde unter Matlab/SIMULINK und an einem Mitsubishi Roboter vom Typ Melfa RV-M1 getestet.

Für den professionellen Einsatz des CLDL Übersetzer sind Erweiterungen der Beschreibungsmöglichkeiten notwendig. Man könnte sich an dieser Stelle weitere nichtlineare Funktionen sowie Eingabeschnittstellen für externe C/C++ Funktionen vorstellen. Aber auch die Realisierung von Schnittstellen für bestehende professionelle Programmpakete wie Matlab/SIMULINK rechtfertigen Erweiterungen der Beschreibungssprache und des Übersetzers.

8 Literaturverzeichnis

[1] K. H. Fasol, K. Diekmann: Fachberichte Simulation, Simulation in der Regelungstechnik, Springer-Verlag, Band 12, ISBN 3-540-52942-X, 1990

[2] H.-P. Franke:VDI Berichte 855, Automatisierungstechnik ´90, Tagung Baden-Baden 18. und 19. September 1990, S 175 ff

[3] http://www.radix.net/%7Ecrbnblu/products.html

[4] Chr. Schmid: Techniques and Tools of CADCS, IFAC Computer Aided Disign in Control Systems, Beijing, PRC, 1988, S 91 ff

[5] R. Schumann, Burgwedel: CAE von Regelsystemen mit IBM-kompatiblen Personal-Computern, atp 34 (1992) 8, S 467 ff

[6] ACSL Reference Manual. Mitchell & Gauthier Ass., Concord, Mass. Auflage 1986, ergänzte Auflage 1989

[7] http://www.eos.ncsu.edu/software/software_index/acsl.html
http://aslan.math.hmc.edu/codee/solvers/acsl.html
http://eurosim.tuwien.ac.at/acsl/acslrt_2.html

[8] http://www.Dynasim.se/dymola.html

[9] J. W. Goldynia:, Objektorientierte Modellierung dynamischer Systeme - zusätzlicher Aufwand oder Vereinfachung, ÖVE, 30.4.1993, e&i 110.Jg. (1993). H. 7/8, S 429-437

[10] H. Elmqvist: Object-Oriented Modeling and Automatic Formula Manipulation in Dymola, Scandinavian Simulation Society, June 9-11, 1993, Kongsberg, Norway

[11] H. Elmqvist, M. Otter: Methods for tearing systems of equations in object-oriented modeling, ESM'94, European Simulation Multiconference, Barcelona, Spain, June 1-3, 1994

[12] P. W. Grant, C. P. Jobling, C. Rezvani: ProDynSA - package of tools for linear dynamic systems analysis written in prolog, University of Wales, UK

[13] S. Berger: Demoversionsbeschreibung, SSPA Systems, PO Box 24002, S-40022 Göteborg Sweden, email simnon@sspa.se

[14] J. Hicklin, A. Grace, J. Kinchen, R. Mauceri: SIMULINK A Program for Simulating Dynamic Systems, MATHWORKS Inc., User's Guide, March 1992

[15] http://www.cray.com/PUBLIC/APPS/DAS/DAS7.html

[16] M. A. Johnson, M. J. Grimble: Using Program CC for control system design, Measurement + Control, Volume 25, June 1992, S 144-148

[17] N. M. Khraishi, W. C. Moore: Ctrl-C Suite and Robust Design of an F100 Engine Controller, Measurement + Control, Volume 25, September 1992, S 203-206

[18] M. R. Katebi, T. Lee, M. Rouse: EASY5 application in process modelling, optimization and control, Measurement + Control, Vol 25, September 1992, S 207-212

[19] Leon S. Levy: Taming the Tiger, Software Engineering and Software Economics, Springer-Verlag, ISBN 0-378-96468-1

[20] Tony Mason, Doug Brown: Lex & Yacc, O'Reilly & Associates Inc., 1991

[21] Jürgen Vollmer: The Compiler Construction System Gentle, Arbeitsbericht der GMD Nr. 508, 1992

[22] F. W. Schröer: Gentle 2.0 Language Specification, GMD Forschungsinstitut, 1993

[23] F. W. Schröer: An Introduction to the Gentle Compiler Description Language, 1995

[24] http://www.first.gmd.de/gentle/

[25] N. Nayeri, K. Voßloh, B. Zimmermann: Compiler-Generierung I, Syntaxanalyse, SS 88, Technische Universität Berlin, Fachbereich Informatik, Institut für Angewandte Informatik, LE Programmiersprachen und Compiler

[26] Ch. Wolf: Entwicklung eines IRL-Übersetzers, Studienarbeit auf dem Gebiet der Automatisierungstechnik, Fachbereich 13 der TU Berlin, Juli 94

[27] J. Böcker, I. Hartmann, Ch. Zwanzig: Nichtlineare und adaptive Regelungssysteme, Springer-Verlag, Berlin 1986, ISBN 3-540-16930-X

[28] N.N.: Sokrates - Ein PC-basiertes Steuerungsentwicklungssystem mit generierter Systemsoftware, Informationsblatt, IPK 1996

Anhang A

EBNF Syntaxregeln für die Regelkreisbeschreibungssprache CLDL:

Program =
GlobalDefs
Regelkreise.
GlobalDefs =
{ GlobalDef }.
GlobalDef =
Bezeichner ´´=´´ Konstante ´´;´´.
Konstante =
Literal |
´´[´´ Polynom ´´]´´ ´´/´´ ´´[´´ Polynom ´´]´´.
Regelkreise =
Regelkreis { Regelreis }.
Regelkreis =
´´CONTROL´´ Bezeichner ´´;´´
Regelkreiskörper
´´END´´ ´´;´´.
Regelkreiskörper =
{ Statement }.
Statement =
Bezeichner ´´=´´ Expression ´´;´´.
Expression =
Funktion |
Term1 |
´´LIMIT´´ ´´(´´ Literal ´´,´´ Literal´´)´´.
Funktion =
´´IN´´ ´´(´´ Bezeichner ´´)´´ |
´´OUT´´ ´´(´´ Bezeichner ´´)´´.
Term1 =
Term2 |
Term1 ´´+´´ Term2 |
Term1 ´´-´´ Term2.
Term2 =
Term3 |
Term2 ´´´´ Term3.*

Term3 =

Konstante |

Bezeichner |

´´-´´ Term3 |

´´+´´ Term3 |

´´(´´ Term1 ´´)´´.

Polynom =

Koeffizient { ´´,´´ Koeffizient }.

Koeffizient =

Literal |

Bezeichner.

Bezeichner =

Buchstabe { Buchstabe | Zahl }.

Literal =

Integer |

Real.

Integer =

Zahl { Zahl }.

Real =

Integer |

Integer ´´.´´ Integer |

´´.´´ Integer.

Buchstabe =

´´a´´ / ´´b´´ / ´´c´´ / ´´d´´ / ´´e´´ / ´´f´´ / ´´g´´ / ´´h´´ / ´´i´´ / ´´j´´ /

´´k´´ / ´´l´´ / ´´m´´ / ´´n´´ / ´´o´´ / ´´p´´ / ´´q´´ / ´´r´´ / ´´s´´ / ´´t´´ /

´´u´´ / ´´v´´ / ´´w´´ / ´´x´´ / ´´y´´ / ´´z´´ /

´´A´´ / ´´B´´ / ´´C´´ / ´´D´´ / ´´E´´ / ´´F´´ / ´´G´´ / ´´H´´ / ´´I´´ / ´´J´´ /

´´K´´ / ´´L´´ / ´´M´´ / ´´N´´ / ´´O´´ / ´´P´´ / ´´Q´´ / ´´R´´ / ´´S´´ / ´´T´´ /

´´U´´ / ´´V´´ / ´´W´´ / ´´X´´ / ´´Y´´ / ´´Z´´ /

´´_´´.

Zahl =

´´0´´ / ´´1´´ / ´´2´´ / ´´3´´ / ´´4´´ / ´´5´´ / ´´6´´ / ´´7´´ / ´´8´´ / ´´9´´.

Kommentar =

´´/´´ { KommentarText | Kommentar } ´´}´´.*

KommentarText =

{ Alle-Character-erlaubt }.

Anhang B

C++ Deklarations- und Definitionsdatei für das Beispiel aus Kapitel 6.2.1 (VZ2-Verhalten):

```
/*************************************************/
/*      Declaration:  control  modul  <VZ2-Beha.h>       */
/*      ==============================      */
/*     Vs.96.3-0   IBo                                   */
/*     Usage:                                            */
/*     - to be included in control modules               */
/*     - for class declaration                           */
/*************************************************/

# ifndef  __CLDL-Control_H
  # define  __CLDL-Control_H
# endif

class  Motor
{
  public :
    typedef  enum {
       Soll , Ist , f , g , Stellgr
    } yy_monitor ;
    typedef enum{
      a1 , b1 , b2 , c1 , c2 , d1 , d2
    } yy_setvalue ;
    typedef  enum {
      VorFilter , Filter
    } yy_filter ;
    typedef  struct {
      double  nominator[ 2 ] ;
      double  denominator[ 2 ] ;
    } yy_filter_struct ;
    Motor( ) ;
    void  Write( yy_monitor , double ) ;
    void  SetValue( yy_setvalue , double ) ;
    void  SetFilter( yy_filter , yy_filter_struct* ) ;
    double  Control( ) ;
```

```
   double  Monitor( yy_monitor ) ;
  private :
   double  yy_koeffs[ 7 ] ;
   double  yy_Ist[ 2 ] ;
   double  yy_f[ 2 ] ;
   double  yy_Soll[ 2 ] ;
   double  yy_g[ 2 ] ;
   double  yy_Stell[ 1 ] ;
   void  InitBuffer( ) ;
   void  RotBuffer( yy_monitor ) ;
} ;

/*********************************************/
/*      Definition:  control  modul  <VZ2-Beha.cpp>      */
/*     ============================      */
/*    Vs.96.3-0   IBo                                    */
/*    Usage:                                             */
/*    - to be linked in Control-Executable               */
/*********************************************/

# include  " VZ2-Beha.h "

Motor :: Motor( )
{
  yy_koeffs[ a1]  =  0.35 ;
  yy_koeffs[ b1]  =  1 ;
  yy_koeffs[ b2]  =  0.196 ;
  yy_koeffs[ c1]  =  - 0.512 ;
  yy_koeffs[ c2]  =  - 0.336 ;
  yy_koeffs[ d1]  =  1 ;
  yy_koeffs[ d2]  =  0.196 ;
  InitBuffer( ) ;
}

void  Motor :: Write( yy_monitor  in_quantity , double  value )
{
  switch( in_quantity ) {
    case  Ist  :
      RotBuffer( Ist ) ;
      yy_Ist[ 0 ]  =  value ;
```

```
      break ;
    case  Soll  :
      RotBuffer( Soll ) ;
      yy_Soll[ 0 ]  =  value ;
      break ;
    default :
      break ;
  }
}

void  Motor :: SetValue( yy_setvalue  position , double  value )
{
  yy_koeffs[ position ]  =  value ;
  InitBuffer( ) ;
}

void  Motor :: SetFilter( yy_filter  filter , yy_filter_struct*  filter_adr )
{
  switch( filter ) {
    case  VorFilter  :
      yy_koeffs[ a1]  =  filter_adr -> nominator[ 0] ;
      yy_koeffs[ b1]  =  filter_adr -> denominator[ 0] ;
      yy_koeffs[ b2]  =  filter_adr -> denominator[ 1] ;
    case  Filter  :
      yy_koeffs[ c1]  =  filter_adr -> nominator[ 0] ;
      yy_koeffs[ c2]  =  filter_adr -> nominator[ 1] ;
      yy_koeffs[ d1]  =  filter_adr -> denominator[ 0] ;
      yy_koeffs[ d2]  =  filter_adr -> denominator[ 1] ;
  }
  InitBuffer( ) ;
}

double  Motor :: Control( )
{
  RotBuffer( g ) ;
  yy_g[ 0]  =
    ( yy_koeffs[ a1] ) / ( yy_koeffs[ b1] ) * ( yy_Soll[ 1] ) +
    ( - ( ( yy_koeffs[ b2] ) / ( yy_koeffs[ b1] ) * ( yy_g[ 1] )
    ) ) ;
  RotBuffer( f ) ;
```

```
 yy_f[ 0]  =
   ( yy_koeffs[ c1] ) / ( yy_koeffs[ d1] ) * ( yy_Ist[ 0] ) +
   ( ( yy_koeffs[ c2] ) / ( yy_koeffs[ d1] ) * ( yy_Ist[ 1] ) +
   ( - ( ( yy_koeffs[ d2 ] ) / ( yy_koeffs[ d1] ) * ( yy_f[ 1] )
   ) ) ) ;
 RotBuffer( Stell ) ;
 yy_Stell[ 0]  =
   yy_g[ 0] -
   ( yy_f[ 0] ) ;
 if ( yy_Stell[ 0]  <  - 10 ) { yy_Stell[ 0]  =  - 10 ; }
 if ( yy_Stell[ 0]  >  10 ) { yy_Stell[ 0]  =  10 ; }
 return( yy_Stell[ 0] ) ;
}

double  Motor :: Monitor( yy_monitor  monitor )
{
 switch( monitor ) {
   case  y  :  return( yy_Ist[ 0] ) ;
   case  f  :  return( yy_f[ 0] ) ;
   case  r  :  return( yy_Soll[ 0] ) ;
   case  g  :  return( yy_g[ 0] ) ;
   case  u  :  return( yy_Stell[ 0] ) ;
   default :  return( - 9999.9999 ) ;
 }
}

void  Motor :: InitBuffer( )
{
 yy_Ist[ 1]  =  yy_Ist[ 0]  =  0 ;
 yy_f[ 1]  =  yy_f[ 0]  =  0 ;
 yy_Soll[ 1]  =  yy_Soll[ 0]  =  0 ;
 yy_g[ 1]  =  yy_g[ 0]  =  0 ;
 yy_Stell[ 0]  =  0 ;
}

void  Motor :: RotBuffer( yy_monitor  buffer )
{
 switch( buffer ) {
   case  Ist  :
     yy_Ist[ 1]  =  yy_Ist[ 0] ;
```

```
      break ;
    case  f  :
      yy_f[ 1]  =  yy_f[ 0] ;
      break ;
    case  Soll  :
      yy_Soll[ 1]  =  yy_Soll[ 0] ;
      break ;
    case  g :
      yy_g[ 1]  =  yy_g[ 0] ;
      break ;
    case  Stell  :
      break ;
  }
}
```

Anhang C

Schnittstellendatei zu Matlab/SIMULINK für das Beispiel aus Kapitel 6.2.1 (VZ2-Verhalten):

```
/*
Alle Funktionen sind vorgegeben in SIMULINK's Dynamic System Simulation Software, Release Notes Version
1.3 und müssen nur noch entsprechend gefüllt werden.
*/

#undef  S_FUNCTION_NAME
#define  S_FUNCTION_NAME  cpp_VZ2.cpp

extern "C" {
  #include  "simstruc.h"
  #include  <math.h>
  #include  "mex.h"
}
#include  "VZ2-Beha.h"

#define  NUMBER_OF_ARGS     (1)                 /* Anzahl der Argumente */
#define  SAMPLE_TIME_ARG    ssGetArg(S,0)       /* wie z.B. Abtastzeit */

#define  SOLL               (0)                 /* Positionen im Eingangsvektor */
#define IST                 (1)

#define STELL               (0)                 /* Positionen im Ausgangsvektor */
#define Vorfilter           (1)
#define Filter              (2)

#define KLASSE              (0)

#define  NUMBER_OF_IWORKS   (0)
#define  NUMBER_OF_RWORKS   (0)
#define  NUMBER_OF_PWORKS   (1)

/*********************************************************************/

static  void  mdlInitializeSizes(  SimStruct *S )
```

```
{
  ssSetNumContStates( S , 0 ) ;
  ssSetNumDiscStates( S , 0 ) ;
  ssSetNumOutputs( S , 3 ) ;
  ssSetNumInputs( S , 2 ) ;
  ssSetDirectFeedThrough( S , 1 ) ;
  ssSetNumSampleTimes( S , 1 ) ;
  ssSetNumInputArgs( S , NUMBER_OF_ARGS ) ;
  ssSetNumIWork( S , NUMBER_OF_IWORKS ) ;
  ssSetNumRWork( S , NUMBER_OF_RWORKS ) ;
  ssSetNumPWork( S , NUMBER_OF_PWORKS ) ;
}

/*********************************************************************/

static  void  mdlInitializeSampleTimes( SimStruct *S )
{
  ssSetSampleTimeEvent( S , 0 , mxGetPr( SAMPLE_TIME_ARG )[ 0 ] ) ;
  ssSetOffsetTimeEvent( S , 0 , 0 ) ;
}

/*********************************************************************/

static  void  mdlInitializeConditions( double *x0 , SimStruct *S )
{
  Motor *Reg  =  new  Motor ;
  ssSetPWorkValue( S , KLASSE , Reg ) ;
}

/*********************************************************************/

extern  "C"  {
  static  void  mdlOutputs( double *y , double *x , double *u , SimStruct *S , int tid )
  {
    Motor *Reg  =  ( Motor * ) ssGetPWorkValue( S , KLASSE ) ;

    Reg->Write( Reg->r , u[ SOLL ] ) ;
    Reg->Write( Reg->y , u[ IST ] ) ;
    y[ STELL ]  =  Reg->Control( ) ;
    y[VORFILTER] = Reg->Monitor( Reg->g);
```

```
   y[FILTER] = Reg->Monitor( Reg->f);
  }
}

/*************************************************************************/

static  void  mdlUpdate( double *x , double *u , SimStruct *S , int tid)
{
}

/*************************************************************************/

static  void  mdlDerivatives( double *dx , double *x , double *u , SimStruct *S , int  tid )
{
}

/*************************************************************************/

static  void  mdlTerminate( SimStruct *S )
{
}

/*************************************************************************/

extern  "C"  {
  #include  "simulink.c"                          /* Mex-file interface */
}
```

Anhang D

Batchdatei zum Erstellen von MEX-Dateien für Matlab/SIMULINK:

```
set MAT_ROOT = D:\MATLAB
set COMP_ROOT = C:\WC100
set INCLUDE = %MAT_ROOT% \simulink\include ; %INCLUDE%
set WAT_LINK = %COMP_ROOT% \bin\wlink.exe
set PATH = %COMP_ROOT% \bin ; %PATH%
set WATCOM = %COMP_ROOT%

rem Erstellen der Linkerdatei
echo option caseexact                                   > wcmex.rsp
echo option map                                         > wcmex.rsp
%MAT_ROOT% \bin\basefnam " name %1 "                    >> wcmex.rsp
echo .mex                                               >> wcmex.rsp
echo format pharlap rex                                 >> wcmex.rsp

%MAT_ROOT% \bin\is_ext %1 .c
if not errorlevel 1 goto COMPWAT

:COMPWAT
set INCLUDE_PATH = -I %COMP_ROOT% \h -I %MAT_ROOT% \extern\include
%COMP_ROOT% \binb\wpp386 -7 -3s -za -D MATLAB_MEX_FILE @INCLUDE_PATH %1
%COMP_ROOT% \binb\wpp386 -7 -3s -za -D MATLAB_MEX_FILE @INCLUDE_PATH %2

%MAT_ROOT% \bin\basefnam " file %1 "                    >> wcmex.rsp
echo .obj                                               >> wcmex.rsp
%MAT_ROOT% \bin\basefnam "file %2 "                     >> wcmex.rsp
echo .obj                                               >> wcmex.rsp
echo library %MAT_ROOT% \extern\lib\libmexwc.lib        >> wcmex.rsp
echo library %COMP_ROOT% \lib386\math387s.lib           >> wcmex.rsp
echo library %COMP_ROOT% \lib386\dos\clib3s.lib         >> wcmex.rsp
echo library %COMP_ROOT% \lib386\plib3s.lib             >> wcmex.rsp

%WAT_LINK% @wcmex.rsp
```