Praktikum Simulationstechnik

Grundlagen zur Vorbereitung

Dieses Dokument wird Studenten, die am Hochschulpraktikum „Simulationstechnik“ des Lehrstuhls für Automatisierung und Informationstechnik teilnehmen, als Download zur Verfügung gestellt. Es dient zur Vorbereitung für das Blockpraktikum und ermöglicht einen selbstständigen Einstieg in das Arbeiten mit MATLAB, Simulink und Stateflow.
Dieser Umdruck wurde mit Hilfe von Studenten erstellt und ersetzt nicht den Besuch des Hochschulpraktikums.

Dieser Umdruck stellt keinesfalls einen Ersatz für einschlägige Lehrbücher dar.

München, September 2014

Version 20140923

Inhalt

1. Praktikum Simulationstechnik – Organisatorisches

1.1 Ziel des Praktikums

Ziel des Praktikums Simulationstechnik ist die Vermittlung von praktischen Erfahrungen bei der Modellierung und Simulation technischer Produkte und Prozesse. Dazu wird mit Hilfe des Simulationswerkzeugs MATLAB/Simulink und der auf Zustandsautomaten basierenden Toolbox „Stateflow“ zunächst eine kleine automatisierungstechnische Anlage simuliert, bei der auch Optimierungsansätze vermittelt werden. Die daraus gewonnenen Erkenntnisse werden für die Simulation einer größeren Anlage genutzt, die sowohl aus Steuerungssoftware als auch aus Hardware (Aktorik und Sensorik) und Mechanik (Fließband, Greifer, etc.) besteht.

Es werden schrittweise die kontinuierlichen Simulationsanteile Hardware und Mechanik in Simulink modelliert, sowie die ereignisorientierte Steuerung mit der Toolbox „Stateflow“ nachgebildet. Die erstellten Einzelmodelle werden anschließend zu einem hybriden System verknüpft.

1.2 Aufbau des Praktikums

Tag 1	Grundlagen zur Simulation mit MATLAB und Simulink und zur ereignisorientierten Simulation mit Stateflow
Tag 2	Simulation einer Kaffeemaschine
Tag 3	Modellieren der HW- und Mechanikkomponenten der Abfüllanlage (Simulink), Modellierung des Prozessguts (kontinuierliche Simulation)
Tag 4	Modellieren der Anlagensteuerung (Stateflow)
Tag 5	Modellbildung, Optimierung und Abschlusstest

Fallen Feiertage in den Zeitraum des Praktikums, wird die entsprechende Zahl an Tagen an die Praktikumswoche angehängt.

1.3 MATLAB als Download

Die TUM hat mit MathWorks einen Rahmenvertrag für den Bezug von MATLAB abgeschlossen. Dies ermöglicht MATLAB, Simulink und alle Toolboxen mit einer Studentenlizenz zu benutzen.

Download und Lizenz

- Der Einstieg zum Bezug dieser Produkte erfolgt über folgendes Portal: http://matlab.rbg.tum.de
- Weitere Informationen zu Bezug und Aktivierung der Produkte erhalten Sie nach dem Login mit Ihrer TUM-Kennung auf dem genannten Portal.
- Die aktuellen Versionen von MATLAB, Simulink und den Toolboxen sind über das MathWorks License Center http://www.mathworks.com erhältlich. Eine Download-Anleitung gibt Hilfestellung für das Herunterladen der Software.

1.4 Weiterführende Literatur

Die folgenden Grundlagen zu MATLAB, Simulink und Stateflow sind speziell auf die Fragestellungen und Aufgaben, die während des Praktikums bearbeitet werden sollen, ausgerichtet. Weiterführende Fachliteratur und Onlinehilfe sind unter folgendem Link zu finden:

http://www.mathworks.de/support/books/

2. Einführung

2.1 Was ist Modellbildung und Simulation

Mit Modellbildung und Simulation wird der Problemlösungsprozess von der Realität auf ein abstrahiertes Abbild der Wirklichkeit verlagert und somit unterstützt. Es wird ein existierendes oder gedachtes System virtuell nachgebildet, um damit experimentieren und schließlich Aussagen über die Wirklichkeit zu erhalten. Simulationen erlauben es, Erkenntnisse über Systeme zu gewinnen, die in der Realität nicht oder nur mit wesentlich höherem Aufwand experimentierbar sind (zu groß: z. B. Galaxien, zu teuer: z. B. Raumfahrttechnik, reales System nicht vorhanden: z. B. zu entwickelndes Produkt) bzw. deren zu untersuchendes Verhalten in der Zukunft liegt (z. B. Wettervorhersage).

Ein Beispiel für eine Simulation ist in Abbildung 1 zu sehen. Der zu entwickelnde Roboter wurde simuliert, um beispielsweise Erreichbarkeits- und Kollisionsvermeidungskontrollen durchzuführen.

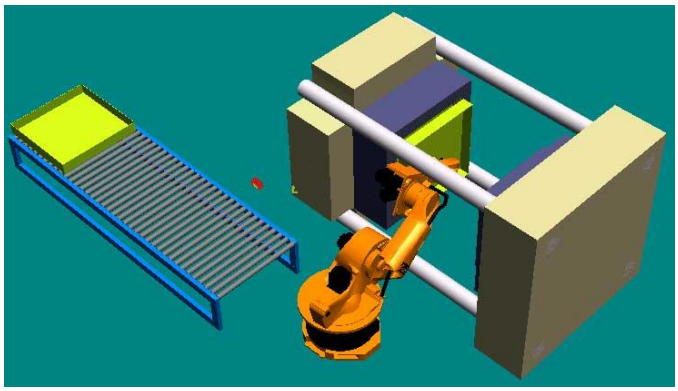


Abbildung 1: 3D-Simulation eines Industrieroboters

2.2 Vorgehensweise der Modellbildung und Simulation

Vor der eigentlichen Durchführung von Simulationsexperimenten muss das zu untersuchende System in einem Simulationssystem nachgebildet werden. Im Folgenden sind die Hauptschritte der Modellbildung und Simulation kurz erläutert:

Problemspezifikation: Wie bei der Entwicklung eines Produkts muss zu Beginn genau spezifiziert werden, was das eigentliche Ziel (Modellierungszweck) der Simulation ist. In der zu erstellenden Anforderungsspezifikation ist eindeutig zu beschreiben, welche Probleme mit Hilfe der Simulation gelöst bzw. welche Aussagen gewonnen werden sollen. Dabei sind neben dem Modellierungszweck Randbedingungen, wie z. B. die notwendige Genauigkeit der Simulation und Zeitbedingungen, die durch das reale System vorgegeben sind, anzugeben.

Modellbildung (Modellieren): In der Phase der Modellbildung wird das zu untersuchende **System analysiert** und ein **konzeptionelles Modell** erstellt. Dabei ist unter Modell allgemein eine vereinfachte Nachbildung eines existierenden oder gedachten Systems in einem anderen begrifflichen oder gegenständlichen System zu verstehen. Die Nachbildung des Systems erfolgt vor dem Hintergrund des Modellierungszwecks. Im Gegensatz zu einem Simulationsmodell, welches bereits ein ablauffähiges Modell darstellt, dient das konzeptionelle Modell dazu, im Vorfeld der Implementierung des Modells die innere Struktur und das Verhalten des nachzubildenden Systems zu beschreiben, um somit einen Einblick in die Funktionsweise des Systems zu erhalten. Die Phase der Modellbildung ist vergleichbar mit der **Entwurfsphase** (Grob- und Feinentwurf) bei der Produkt- bzw. Softwareentwicklung.

Simulation: In der Phase der Simulation wird zum einen das konzeptionelle Modell in ein Simulationsmodell überführt (Implementierung), zum anderen werden an dieser Stelle die eigentlichen Experimente durchgeführt. Die Implementierung der Simulationsmodelle kann mit konventionellen Programmiersprachen („C“), mit simulationsspezifischen Sprachen („MATLAB“) oder mit objektorientierten Simulationsentwicklungsumgebungen („Simulink“, „Dymola“) erfolgen.

Gesamtauswertung und Präsentation: Nach dem erfolgreichen Experimentieren sind die gewonnen Ergebnisse auszuwerten und dem Modellierungszweck entsprechend darzustellen.

In Abbildung 2 ist der Zusammenhang zwischen der nachzubildenden Realität, dem konzeptionellen Modell und dem Simulationsmodell dargestellt.

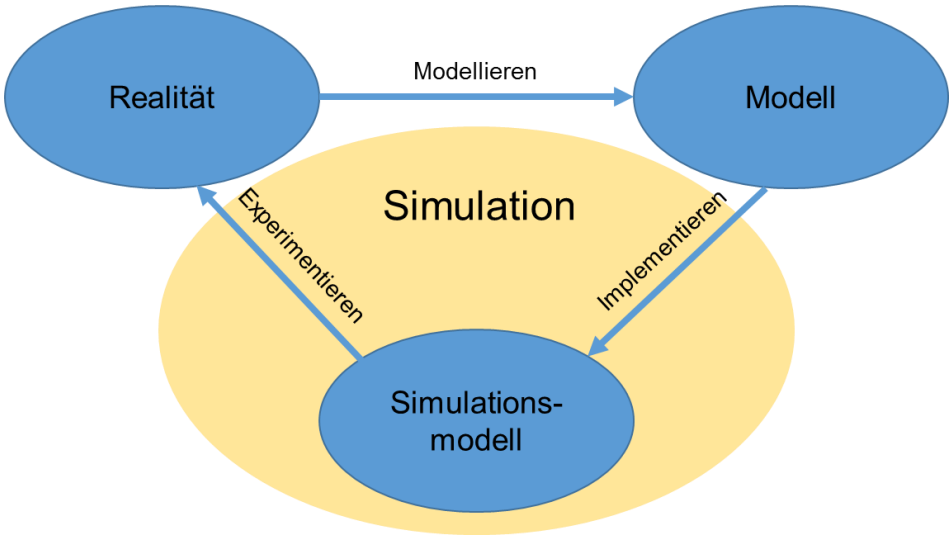


Abbildung 2: Gesamtansicht Modellbildung und Simulation

2.3 Simulationstechnologien

Abhängig vom Ziel und Zweck einer Simulationsstudie (Modellierungszweck) sind die entsprechenden Simulationssysteme unterschiedlich aufgebaut. Im Folgenden werden vier verschiedene Arten von Simulationen, sogenannte Simulationstechnologien, vorgestellt.

2.3.1 Ablaufsimulation

Durch Ablaufsimulationen werden Abläufe von Prozessen (technische, wirtschaftliche, …) nachgebildet. Dabei sind vor allem die Reihenfolge sowie die Zeitanforderungen der Abläufe im Fokus der Betrachtung. Ziel der Ablaufsimulation ist in erster Linie die Funktionsauslegung und -optimierung, der Funktionsnachweis (→ Funktionstests) und Risikominimierung technischer Prozesse.

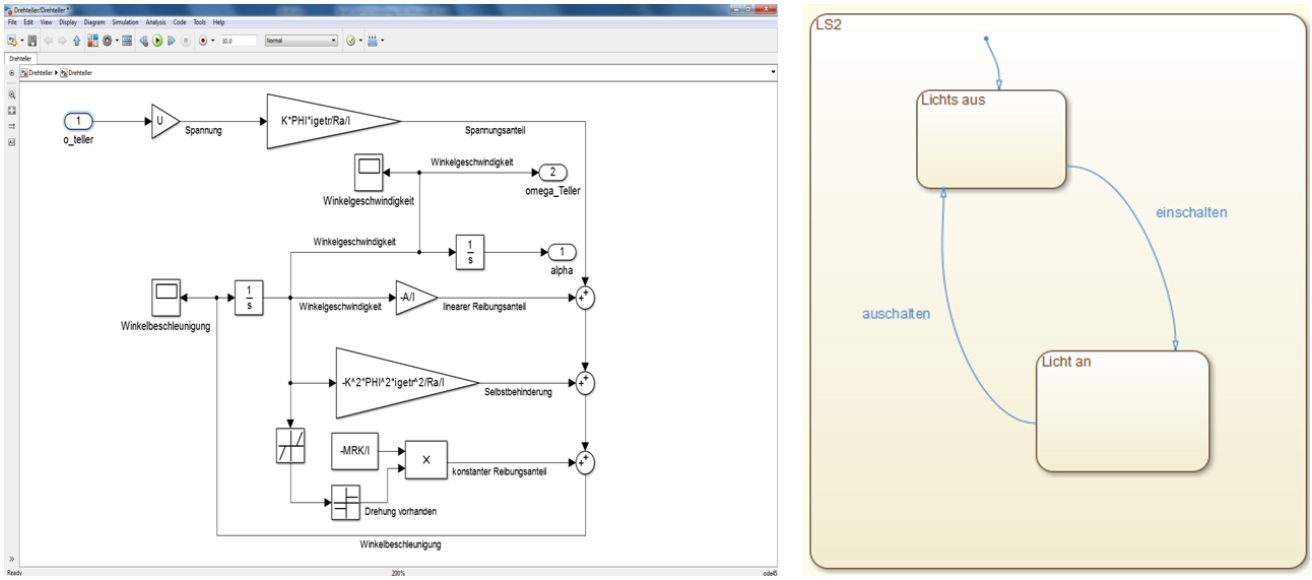
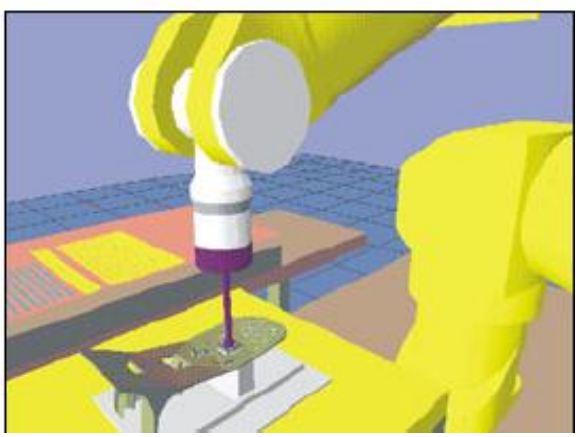


Abbildung 3: Beispiel einer Ablaufsimulation in Simulink / Stateflow

2.3.2 3D-Kinematiksimulation

Die 3D-Kinematiksimulation dient der Analyse und Optimierung von Komponenten eines Produktionssystems. Ziel ist der Test und die Optimierung von geometrischen Auslegungen und Bewegungsabläufen, sowie die Detektierung und Vermeidung von Kollisionen. Die Hauptanwendungsgebiete sind in der Produktentwicklung und speziell auch im Digital Mock-Up zu sehen.



2.3.3 FEM

FEM-Simulation eignet sich zur Nachbildung physikalischer Werkstoffeigenschaften. Dadurch ist es beispielsweise möglich, die mechanische Beanspruchung oder das Schwingungsverhalten einzelner Bauteile bzw. ganzer Baugruppen am Rechner zu analysieren. Das Simulationstool wird als etabliertes Berechnungswerkzeug im Rahmen der Produktentwicklung in der Regel für Werkstoffuntersuchungen, Festigkeitsnachweise und geometrische Auslegungen von Mechanikkomponenten eingesetzt.

Neben mechanischen können auch thermische oder strömungsmechanische Einflüsse und Belastungen nachgebildet werden.

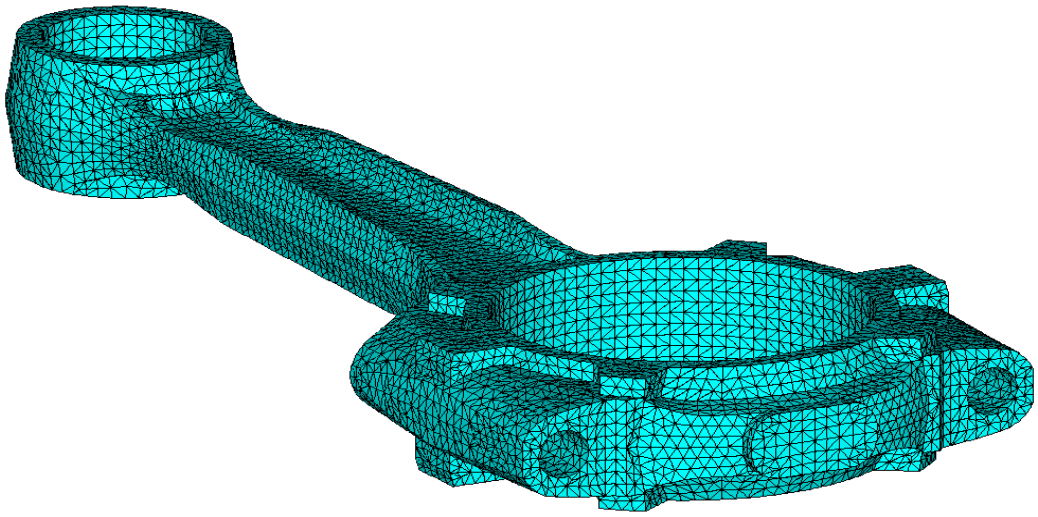


Abbildung 4: FEM-Modell eines Pleuels

2.3.4 Mehrkörpersimulation

Mehrkörpersimulation wird eingesetzt, um das dynamische Verhalten, d. h. die Starrkörperbewegungen sowie das Schwingungsverhalten von technischen Systemen zu analysieren.

Typische Anwendungsgebiete sind:

- Schwingungs-Stabilitätsuntersuchungen in Mehrkörpersystemen
- Analyse und Optimierung von Starrkörperbewegungen
- Berechnung elastischer Bauteilverformungen in Folge der Bewegungsdynamik der Komponente

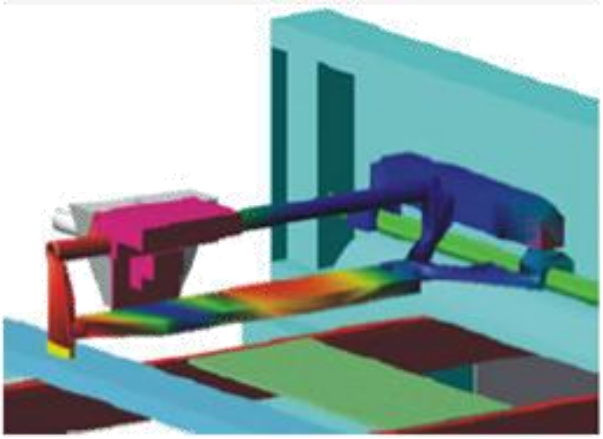


Abbildung 5: Beispiel für Mehrkörpersimulation

2.4 Klassifizierung von Simulation

Dynamische Simulationen, also Simulationen deren Zustand sich im Laufe der Simulationszeit ändert, können danach klassifiziert werden, ob sich ihre Zustandsvariablen und Simulationszeit diskret oder kontinuierlich ändern:

- **Simulationszeit**
  - *Zeitkontinuierlich*: beliebige Zeitpunkte werden simuliert (unendlich viele Zeitpunkte in einem Intervall)
  - *Zeitdiskret*: nur dezidierte Zeitpunkte werden simuliert (endlich viele Zeitpunkte in einem Intervall)
- **Zustandsvariablen**
  - *Zustandskontinuierlich*: Zustandsvariablen können beliebige Werte annehmen → unendliche Anzahl an Zuständen
  - *Zustandsdiskret*: nur endliche Anzahl an Systemzuständen werden simuliert

Diskrete Zustände

Werden von einem System nur bestimmte Zustände (diskrete Zustände) betrachtet, spricht man von einer ereignisorientierten Simulation. Bei einer ereignisorientierten Simulation hätte ein Motor beispielsweise die möglichen Zustände „an“ und „aus“. Sämtliche Zwischenschritte zwischen diesen beiden Zuständen (stetige Geschwindigkeitszu- bzw. -abnahme) werden vernachlässigt.

Kontinuierliche Zustände

Systeme mit unendlich vielen Zuständen (kontinuierliche Zustände) werden nochmals unterteilt. Kann in einer Simulation das nachgebildete System seine Zustände zu jeder beliebigen Zeit ändern, d. h. wird die Simulationszeit als kontinuierlich angenommen, so spricht man von kontinuierlicher (numerischer) Simulation (z. B. ein Motor, bei dem sich kontinuierlich über die Zeit der Geschwindigkeitsverlauf ändert). Wird stattdessen angenommen, dass sich die Zustände in einer Simulation nur zu bestimmten Zeiten ändern können, wird von einer zeitdiskreten Simulation gesprochen.

Hybride Simulationen

Werden ereignisorientierte Simulationen, numerische Simulationen und zeitdiskrete Simulationen miteinander kombiniert, so spricht man von hybriden Simulationen.

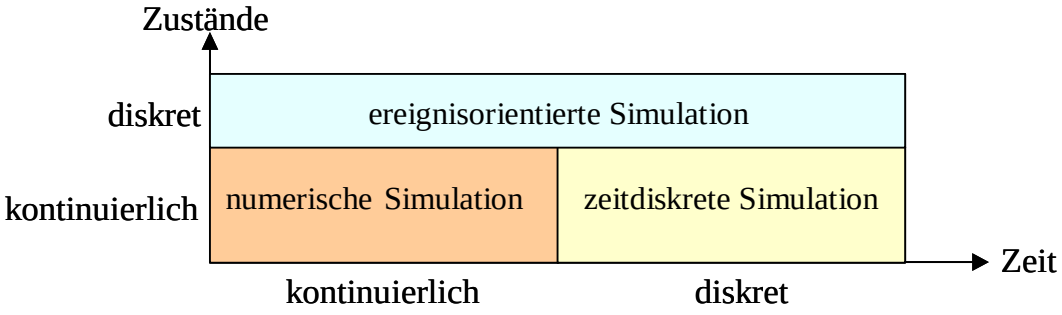


Abbildung 6: Klassifizierung von Simulationen

3. MATLAB

„MATLAB ist ein umfangreiches Softwarepaket für numerische Mathematik. Wie der Name MATLAB, abgeleitet von MATrix LABoratory, schon zeigt, liegt seine besondere Stärke in der Vektor- und Matrizenrechnung. Unterteilt in ein Basismodul und zahlreiche Erweiterungspakete, so genannte Toolboxen, stellt MATLAB für unterschiedlichste Anwendungsgebiete ein geeignetes Werkzeug zur simulativen Lösung der auftretenden Problemstellungen dar.“ [A. Angermann, MATLAB-Simulink-Stateflow].

3.1 Grundlagen zu MATLAB

Im Fokus des Praktikums steht die Modellierung mit Simulink und Stateflow. Trotzdem sollen im Folgenden kurz die Grundlagen zu MATLAB und dessen Benutzung vermittelt werden.

Command Window

Im Command Window können direkt alle Befehle aufgerufen oder Variablen deklariert werden.

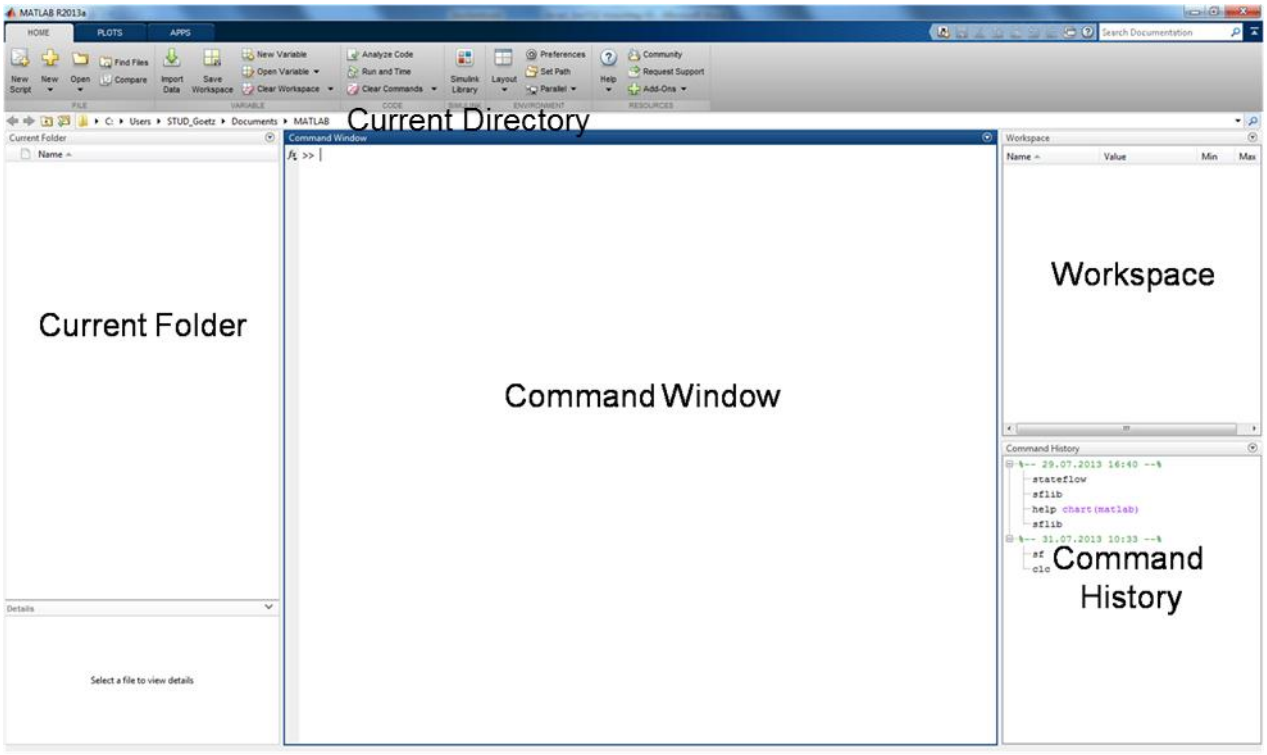


Abbildung 7: MATLAB Desktop

Unter „Current Directory“ sollte während des Praktikums stets das eigene Datenverzeichnis (Z:\) stehen. „Current Directory“ gibt den Pfad an, unter dem MATLAB alle erstellten Funktionen und Modelle sucht, die ausgeführt werden sollten.

Workspace

Im Workspace werden alle Variablen hinterlegt. Es gibt einen globalen Workspace und lokale Workspaces.

3.1.1 Basis-Funktionalität

Variablen

Ähnlich wie in der Programmiersprache C können in MATLAB Variablen zum (Zwischen-) Speichern von Werten verwendet werden.

Die Variablen werden nach einer üblichen Zuweisung eines Wertes (über Eingabe im Command Window) direkt im MATLAB-Workspace deklariert.

```
>> a = 28
```

Hier kann man unterscheiden zwischen globalen Variablen (abgelegt im globalen Workspace) und lokale Variablen (abgelegt im lokalen Workspace). Globale Variablen können direkt über die Eingabe im Command Window in den globalen Workspace abgelegt werden. Variablen, die innerhalb einer Funktion verwendet werden (einschließlich der Ein- und Ausgabevariablen), sind üblicherweise lokale Variablen.

- Einige Variablennamen, wie pi (Kreiszahl π), i bzw. j (imaginäre Einheit) und inf (unendlich) sind in MATLAB reserviert und können entsprechend verwendet werden.
- Das Semikolon am Ende der Zeile unterdrückt die Ausgabe der eben eingegebenen Zuweisung.

Beispiel mit Semikolon:

```
>> a = 10 + 5 / 5;

>>
```

Beispiel ohne Semikolon:

```
>> a = 10 + 5 / 5

a =

    11
```

- Die Eingabe des Variablennamens ohne Zuweisung gibt den aktuellen Wert der Variable aus.
- In MATLAB gelten die gewöhnlichen Rechenoperationen (“+“, „-„, “*“, “/“, “()“).

```
>> a = ((5 + 4) * 2 - 8) / 5;
```

Vektoren und Matrizen

Ein Vektor bzw. eine Matrix wird direkt zwischen eckigen Klammern eingegeben.

- Die Elemente einer Zeile werden durch Komma oder Leerzeichen getrennt und eine neue Zeile wird durch Strichpunkt oder Zeilenumbruch gekennzeichnet.
- Vektor oder eine Matrix durch die Eingabe von „ ’“ nach dem Variablennamen zu transformieren.
- Durch ein Semikolon „;“ kann eine neue Zeile beim Initialisieren erzeugt werden.
- Mit dem „%“-Zeichen wird ein Kommentar eingeleitet. Alles, was in der Befehlszeile nach dem „%“-Zeichen steht, wird als Kommentar interpretiert.

Im Folgenden sind einige Beispiele zum Befüllen und Auslesen von Vektoren und Matrizen aufgelistet:

Vektor initialisieren

```
>> Zeilenvektor = [1 2 3]

Zeilenvektor =

     1     2     3
```

```
>> Spaltenvektor = [1 ; 2 ; 3]

Spaltenvektor =
     1
     2
     3
```

Vektor transponieren

```
>> Vektor_trans = Zeilenvektor’ %Das Hochkomma erstellt die Transponierte

Vektor_trans =
     1
     2
     3
```

Matrix initialisieren

Die Initialisierung einer Matrix kann sowohl durch manuelle Zuordnung erfolgen oder über interne MATLAB Funktionen:

- `eye(zeilen)` % Erzeugen einer Einheitsmatrix (quadratisch)
- `ones(zeilen, spalten)` % Initialisierung einer Matrix mit „1“
- `zeros(zeilen, spalten)` % Initialisierung einer Matrix mit „0“
- `rand(zeilen, spalten)` % Initialisierung einer Matrix mit Zufallswerten zwischen 0 und 1
- `randn(zeilen, spalten)` % Initialisierung einer Matrix mit normal-verteilten Zufallswerten

```
>> Matrix = [Zeilenvektor;4 5 6]

Matrix =
     1     2     3
     4     5     6
     1     2     3
>> Matrix = ones(3,2)

Matrix =
     1     1
     1     1
     1     1
```

```
>> Einheitsmatrix = eye(3)

Einheitsmatrix =
     1     0     0
     0     1     0
     0     0     1
>> Matrix = zeros(2,3)

Matrix =
     0     0     0
     0     0     0
```

Elemente auslesen

```
>> erstes_Element = a(1)   % Index 1 für das erste Element (nicht 0!!)

erstes_Element =

     1

>> letztes_Element = Matrix(end)

letztes_Element =

     6

>> b = Matrix(3,2)         % Element der dritten Zeile und zweiten Spalte

b =

     1
```

Doppelpunkt-Operator

Der Doppelpunkt Operator ist einer der wichtigsten Operatoren in MATLAB und findet in vielen Fällen Verwendung. Mit seiner Hilfe können spezielle Zeilenvektoren erzeugt werden, die z. B. bei der Indizierung in `for`-Schleifen oder beim Plotten verwendet werden. Dabei wird ausgehend von einer Zahl solange eine Einheit addiert und in dem Vektor gespeichert, bis ein vorgegebenes Ende erreicht oder überschritten wurde. Wird kein Inkrement angegeben so wird automatisch eine Schrittweite von 1 verwendet. Die allgemeine Syntax ist

anfang:ende oder anfang:inkrement:ende

Matrix initialisieren

```
>> Matrix = [1:3;4:6;6:3:12]

Matrix =

     1     2     3

     4     5     6

     6     9    12
```

Zeile auslesen

```
>> ZweiteZeile = Matrix(2,:)     %Alle Elemente der 2. Zeile

ZweiteZeile =

     4     5     6
```

Bestimmte Bereiche auslesen

```
>> Matrix2 = Matrix(2:end,2:3)

Matrix2 =

     5     6

     9    12
```

for-Schleife

Durchläuft die Variable i von anfang bis ende durch und erhöht ihn in jedem Schritt um den Wert inkrement. Dabei werden alle Anweisungen zwischen `for` und `end` sooft ausgeführt, bis i=ende.

```
>> for i = 1:0.1:1.3
   3*i
   end

Ausgabe:

     1    1.1000    1.2000    1.3000
```

3.1.2 Hilfefunktion

Für das Praktikum ist die Hilfefunktion essentiell. Auftretende Fragen sollen immer erst durch die Verwendung von `help` oder `lookfor` zu lösen versucht werden.

Der Befehl

`help` FUNKTIONSNAME

bietet im Eingabefenster Hilfe zu Funktionen, deren Namen bekannt ist. Die Suche nach Hilfstexten über das Command Window ist dadurch nur empfehlenswert, wenn der Befehl bekannt ist. Falls man für ein Problem eine passende Funktion sucht, kann man mit folgendem Befehl eine Suche beginnen:

lookfor SUCHBEGRIFF

Noch einfacher ist es, über die Taskleiste „Help“ -> Dokumentation (Shortcut F1) die komplette MATLAB-Dokumentation nach einem bestimmten Suchbegriff (Englisch!) zu durchsuchen. Dies sowohl für MATLAB und die im Praktikum verwendeten Toolboxen Simulink und Stateflow.

Beispiel:

```
>> help sin
sin - Sine of argument in radians
    This MATLAB function returns the sine of the elements of X.
    Y = sin(X)
    Reference page for sin
    See also asin, asind, sind, sinh
    Other functions named sin
        fixedpoint/sin, symbolic/sin
>> help          %Aufruf aller Hauptkategorien
>> help elfun    %Aufruf elementarer mathematischer Funktionen
```

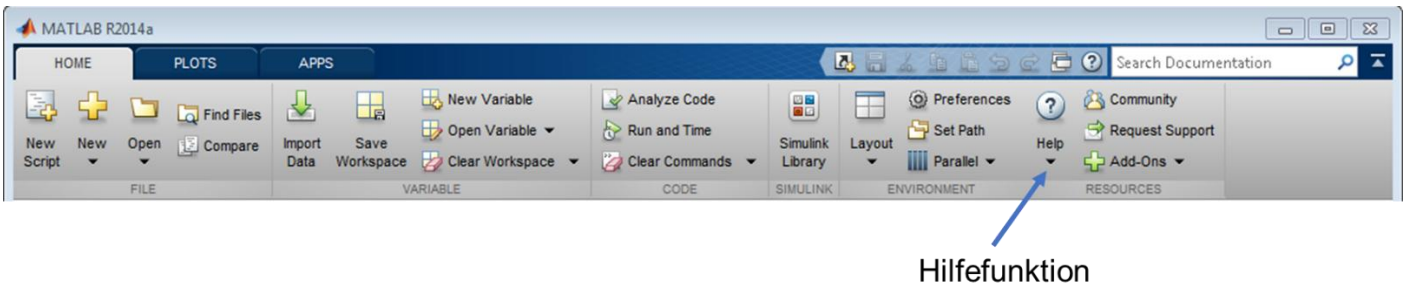


Abbildung 8: Hilfefunktion aufrufen

```
>> lookfor eigenvalues

eigshow           - Graphical demonstration of eigenvalues and singular values.

expmdemo3         - Matrix exponential via eigenvalues and eigenvectors.

condeig           - Condition number with respect to eigenvalues.

eig               - Eigenvalues and eigenvectors.

ordeig            - Eigenvalues of quasitriangular matrices.
```

3.1.3 m-Files: Skripte und Funktionen

In MATLAB erstellte Programme liegen als sogenannte m-files (Dateiendung .m) vor. Es existieren zwei Arten von Programm- oder m-Files: Skripte (Script) und Funktionen (Function).

Allgemeines:

- Aufgerufen wird ein m-File wie jede andere Funktion im Command-Window oder in einem beliebigen anderen m-File.
- Ein neues m-File kann man unter New→Script für Skripte oder New→Function für Funktionen erzeugen. Ein Editor öffnet sich anschließend automatisch. Beim Speichern wird die Endung „.m“ automatisch angehängt.
- Ein m-File kann im Command-Window gestartet werden, indem man dort den Namen des m-Files eingibt (ohne die Endung „.m“). Außerdem kann der m-File im Current Directory durch Rechtsklick→Run gestartet werden.
- Der Dateiname darf keine Umlaute und Sonderzeichen außer Unterstrich erhalten.

Skripte

Ein Skript ist eine externe Datei, die eine Folge von Anweisungen enthält. MATLAB arbeitet dieses Skript genauso ab, wie wenn diese Anweisungen direkt nacheinander „von Hand“ im Command Window eingegeben würden.

- **Programmzeilen:** Meistens wird man auf jeder Programmzeile nur einen Befehl notieren. Ein **Strichpunkt** nach dem Befehl verhindert, dass die erzeugten Werte im Command Window ausgegeben werden.
- **Kommentare** werden mit dem **Prozentzeichen %** gekennzeichnet. Alles, was auf einer Programmzeile nach diesem Zeichen erscheint, wird von MATLAB nicht interpretiert.

Beispiel:

```
%Parameterdatei für die Simulationsanlage

%Anlage

MaxFlaschen = 25;     %Anzahl der Flaschen, die max. durchlaufen dürfen

%Drehteller

PHI = 0.001;          % Hauptfluss im Elektromotor

U = 24;               % [V] angelegte Spannung am Elektromotor

Ra = 40;              % [Ohm] Ankerwiderstand des Elektromotors

I = 0.5;              % [kgm²] Massenträgheit des Tellers

K = 8;                % Wicklungsanzahl

MRK = 1;              % Konstante Reibungsmoment

A = 5;                % Konstante für geschwindigkeitsabhängige Reibung

iGetr = 1080;         % Getriebeübersetzung
```

Funktionen

Selbst programmierte Funktions-m-Files werden auf die gleiche Weise angewandt, wie die von MATLAB bereitgestellten Funktionen, z. B. cos (Cosinusfunktion), d. h. sie werden mit ihrem Namen (= Name des m-Files) und entsprechenden Argumenten aufgerufen:

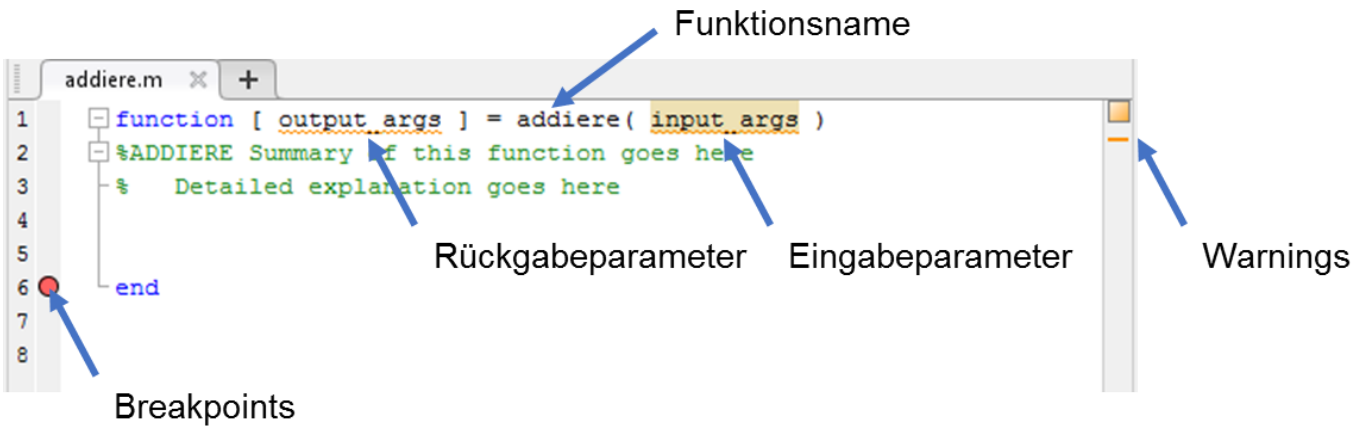


Abbildung 9: Aufbau von Funktionen

Eine Funktion zeichnet sich dadurch aus, dass ihr (meistens) Werte übergeben werden, die sie verarbeiten soll, und dass sie die Resultate an den „Aufrufer“ (aus dem Command Window oder aus einem anderen Skript bzw. Funktion) zurückgibt.

- Der **Name der Funktion** muss mit dem Dateinamen übereinstimmen, unter dem die Funktion als m-File abgespeichert ist.
- **Eingabeparameter** (in1, in2,…) legen fest, welche Werte der Funktion übergeben werden
- **Rückgabeparameter** (out1, out2,…) spezifizieren die Werte, durch die die Funktion die Resultate der Berechnung zurückgeben kann.
- Neben selbst erstellten Funktions-m-Files gibt es schon in **MATLAB implementierte Funktionen** wie mathematische Funktionen (*help elfun*) oder die zahlreichen Toolboxen.
- Neben der Scrollbar findet man ein **farbiges Quadrat.** Dies kann die Farben „grün“ (=alles in bester Ordnung), „orange“ (=Warning. Programm hat keine Fehler, kann aber noch optimiert werden) oder „rot“ (=Syntaxfehler).
- **Breakpoints** können gesetzt werden, um ein besseres Debuggen zu ermöglichen, um z. B. eine Funktion Schritt für Schritt bei Fehlersuche zu debuggen.

Beispiel für eine Funktion:

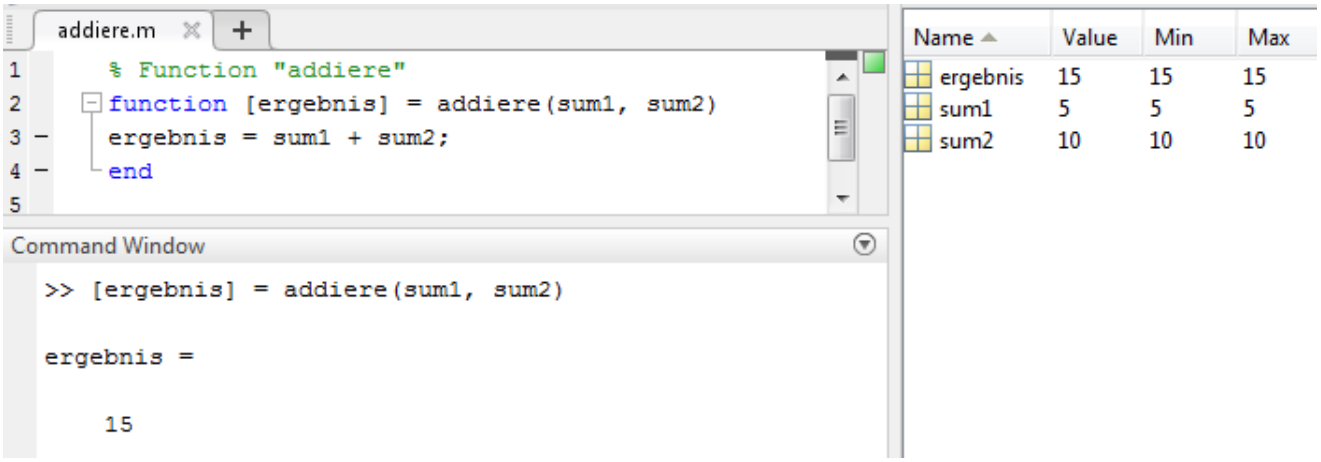


Abbildung 10: Beispiel für eine Funktion

4. Simulink

Simulink ist eine graphische Oberfläche zur Modellierung von physikalischen Systemen mittels Signalflussgraphen. Dabei ist Simulink ein Erweiterungspaket, eine sog. Toolbox von MATLAB.

- Gestartet wird Simulink durch die **Eingabe von „simulink“** in das MATLAB Command-Window oder durch klicken auf das **Simulinksymbol („Simulink Library“)** in der MATLAB Toolbar.
- Die Basis der Signalflussgraphen sind die Funktionsbausteine. Diese sind durch Ein- und Ausgänge, Namen und Blocksymbolen gekennzeichnet.
- Durch einen Doppelklick auf einen Funktionsbaustein lassen sich alle nötigen Parameter einstellen.
- Alle Funktionsbausteine findet man in der „Simulink Library“, die sich beim Start von Simulink automatisch öffnet.

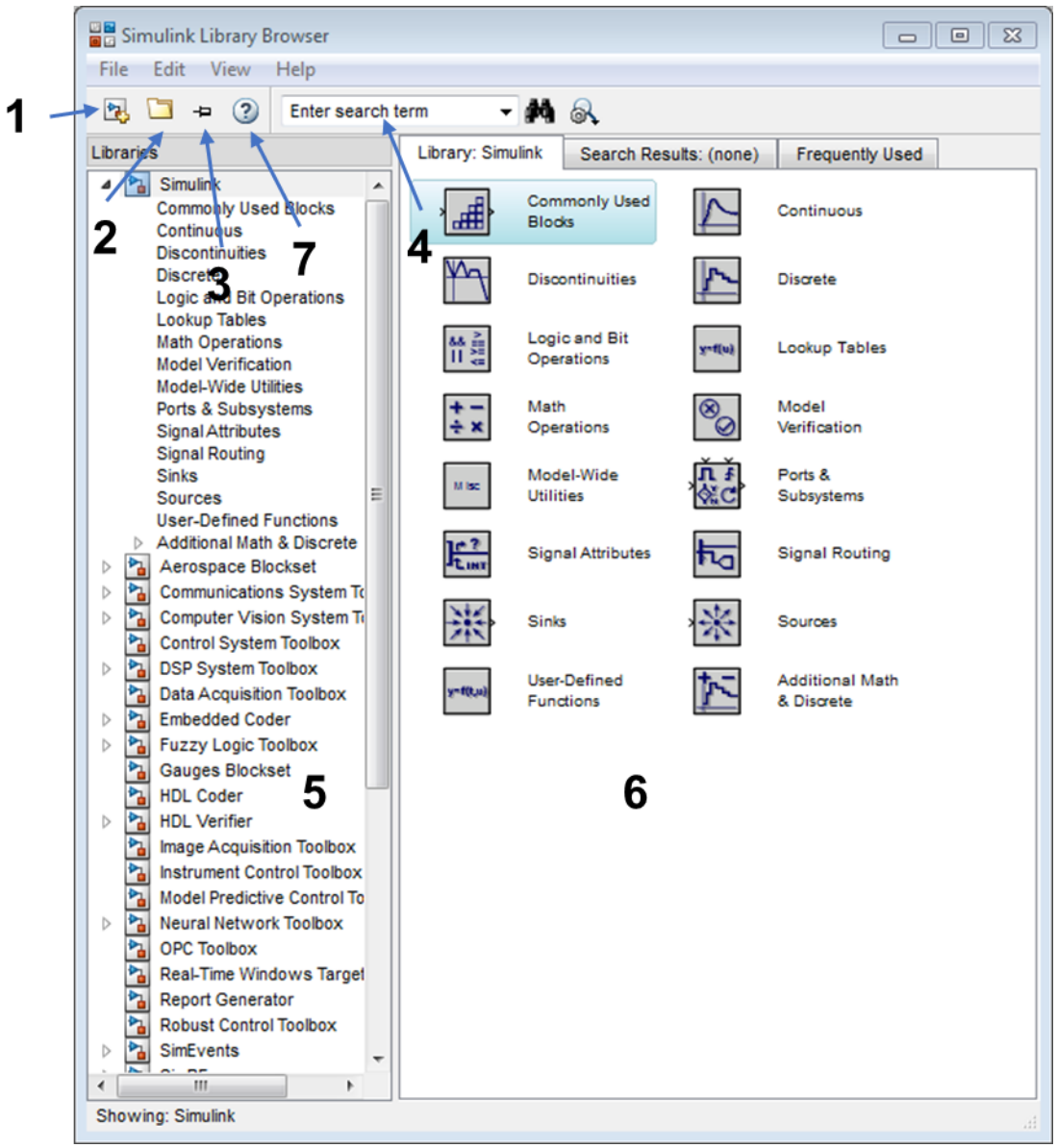


Abbildung 11: Der Library Browser

1. Durch Klick auf dieses Symbol kann ein **neues Simulinkmodell** erstellt werden.
2. Durch Klick auf dieses Symbol kann ein **vorhandenes Simulinkmodell** geöffnet werden.
3. Durch Klick auf dieses Symbol bleibt die Library **immer im Vordergrund** (immer sichtbar vor allen anderen Fenstern).
4. In der **Suchleiste** können Blöcke gesucht werden, deren Name bekannt ist.
5. In diesen Bereich kann man die Toolboxen oder ein Unterverzeichnis auswählen (vgl. Windows Explorer: **Verzeichnisbaum**).
6. In diesem Bereich kann man die einzelnen Blöcke auswählen (vgl. Windows Explorer: Dateifenster). Diese Blöcke lassen sich dann einfach über **Drag'n'Drop** in das Modell ziehen oder über **Strg + I** in das aktuelle Simulinkmodell hineinkopieren.
7. Durch Klick auf das Symbol wird die MATLAB Simulink **„Help" Funktion** aufgerufen. Dort findet man eine detaillierte Beschreibung der Blöcke inklusive einem Beispiel.

   Alternative: Rechtsklick auf Block → Help for the „Blockname" block

   Beispiel:

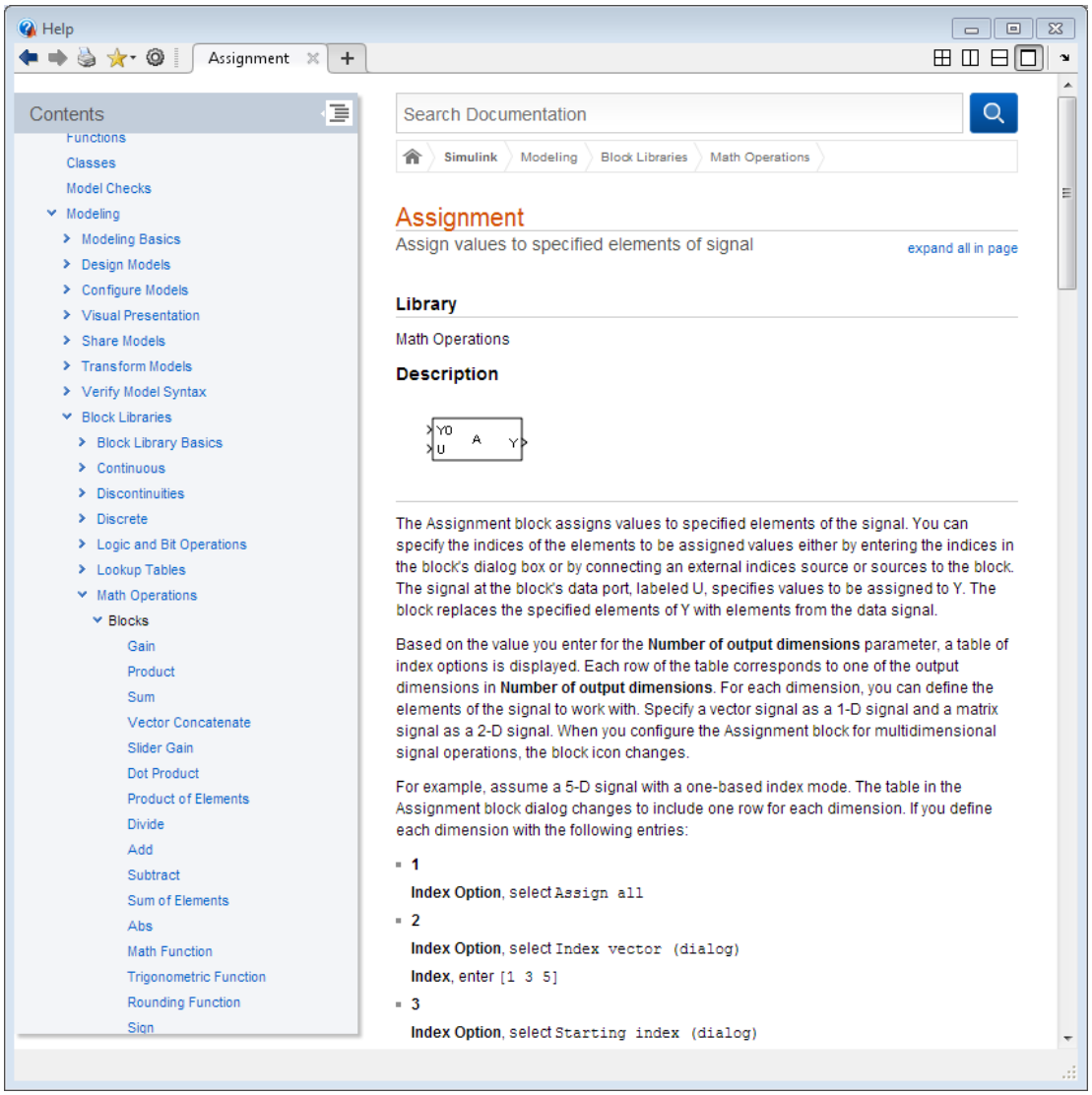


Abbildung 12: Hilfefunktion für Block „Assignment"

4.1 Beschreibung von Simulink

Als nächstes werden die Bestandteile und die Funktionsweise von Simulink erläutert.

4.1.1 Simulink Bibliotheken

Die Simulink Bibliotheken beinhalten einzelne, elementare Bausteine zum Bilden von Systemen. Im Folgenden werden die wichtigsten Bibliotheken vorgestellt.

- Alle Blöcke können mit „Strg+R“ gedreht bzw. mit „Strg+I“ gespiegelt werden.
- Duplizieren lässt sich ein Block, indem man die rechte Maustaste gedrückt vom Ursprungsblock wegzieht
- Das Parametermenü jedes einzelnen Blocks kann durch Doppelklick auf den Block geöffnet werden.
- **Rechtsklick → Help öffnet eine Hilfedatei mit wichtigen Hinweisen zu den einzelnen Blöcken.**

Sources

Sources (Quellen) dienen zum Erzeugen von Signalen und zum Einlesen von Signalen aus dem Workspace oder aus Dateien.

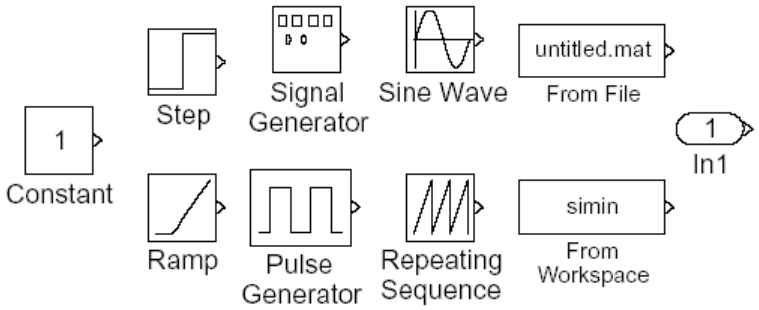
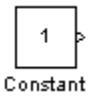


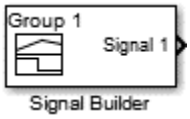
Abbildung 13: Beispiele für Sources

Dieser Block gibt die aktuelle **absolute** Simulationszeit aus.



Mit Hilfe dieses Blockes ist es möglich, eine Konstante in das Modell als Signal zu geben. Wie unter MATLAB können auch Vektoren und Matrizen ausgegeben werden.

Dieser Block erzeugt ein pulsweitenmoduliertes Rechtecksignal.



Mit diesem Block können beliebige Signale erzeugt werden, weswegen er **hervorragend zum Testen eines Modells geeignet** ist.

Dieser Block kann Sinussignale ausgeben.

Mit diesen Block kann man ein Sprungsignal erzeugen, d. h. der Wert springt zu einer bestimmten Zeit einmalig auf einen anderen.

Beispiel Pulse Generator

Der Block „Pulse Generator“ erzeugt eine periodische Rechteck-Funktion mit einstellbarer Periode, Amplitude, Startzeit, etc. Der Block eignet sich hervorragend zum Testen von selbst erstellten Modellen.

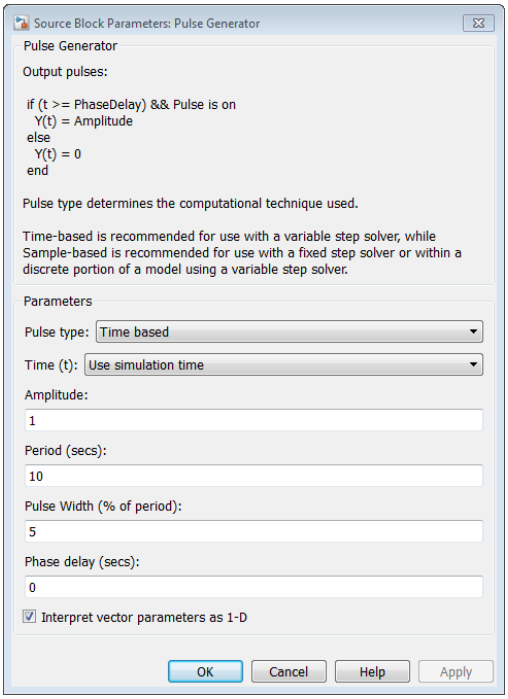


Abbildung 14: Parameter des Blockes „Pulse Generator"

Die Grundeinstellung für Amplitude, Periode und Pulsbreite für die folgenden Beispiele sind geändert worden und sollten zum Testen in ihrem Modell auch geändert werden zu folgenden Einstellungen:

- Amplitude: 1
- Periode: 2
- Pulsebreite: 50

Die Änderung der Einstellung führt zu folgendem Scope:

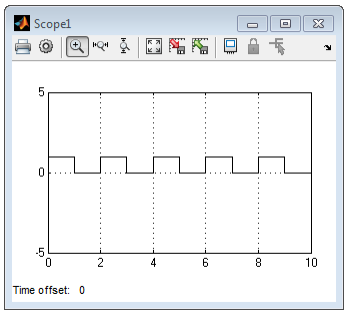


Abbildung 15: Scope „Pulse Generator"

Beispiel Signal Builder

Im Block „Signal Builder“ können Signale graphisch erzeugt werden.

- Nach einem Doppelklick auf den Block können unter dem Menüpunkt „Signal“ neue Signalausgänge hinzugefügt werden.
- Mit Hilfe der Umschalttaste lassen sich in bestehende Signale neue Punkte einfügen.

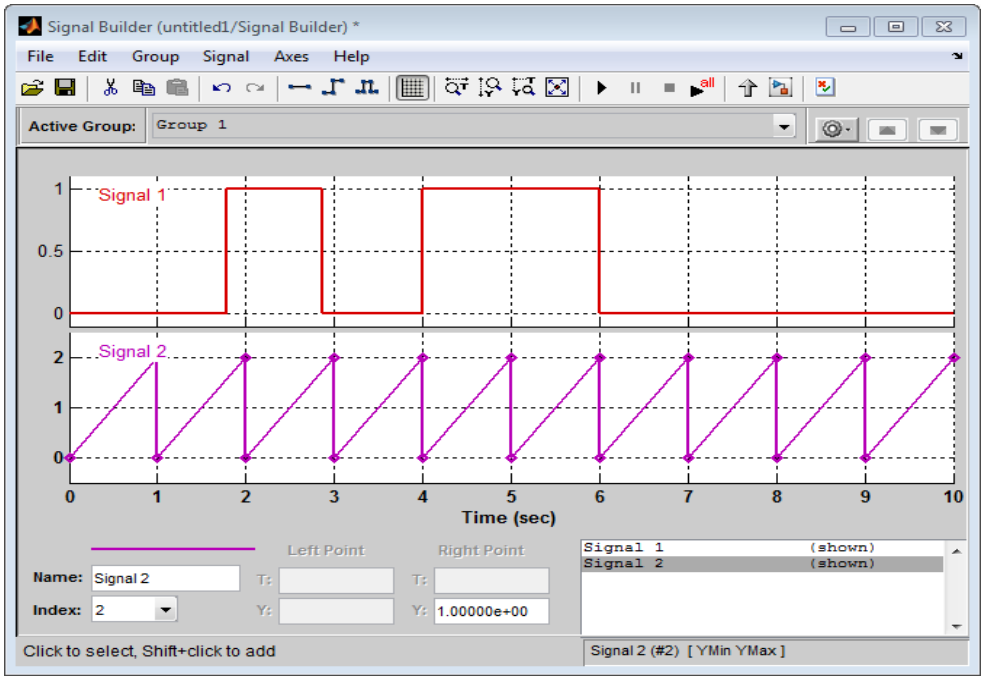


Abbildung 16: Aufbau des Blockes "Signal Builder"

Sinks

Sinks dienen der Anzeige von Signalen und der Ausgabe von Signalen in den Workspace oder einer Datei.

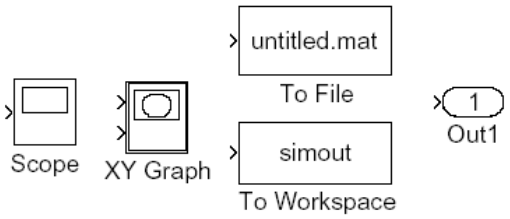
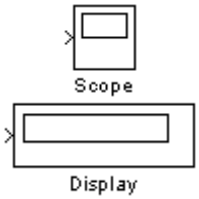


Abbildung 17: Beispiele für Sinks



Scope ist ein virtueller Oszillograph. Mit ihm lassen sich (vektorielle) Signale graphisch darstellen.

Auf dem Display können die aktuellen Werte eines Signals (nicht aber deren Verlauf) abgelesen werden.

Beispiel:

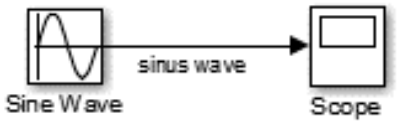


Abbildung 18: Einfaches Beispiel zu einem Scope

- Durch einen Doppelklick auf das Scope-Symbol erhält man nach einem Simulationsdurchlauf die nachfolgende Abbildung.
- Durch Klick auf das „Zahnrad“ öffnet sich das Parameterfenster des Scopes.
- „Number of axes“ stellt die Zahl der Eingänge des Blockes dar.
- Durch Beschriftung der Eingänge in den Scope (im Simulinkmodell) erscheinen diese über dem entsprechenden Graphen in der Darstellung
- Das Häkchen bei History → „Limit data points to last“ entfernen, um längere Simulationszeiten darstellen zu können.

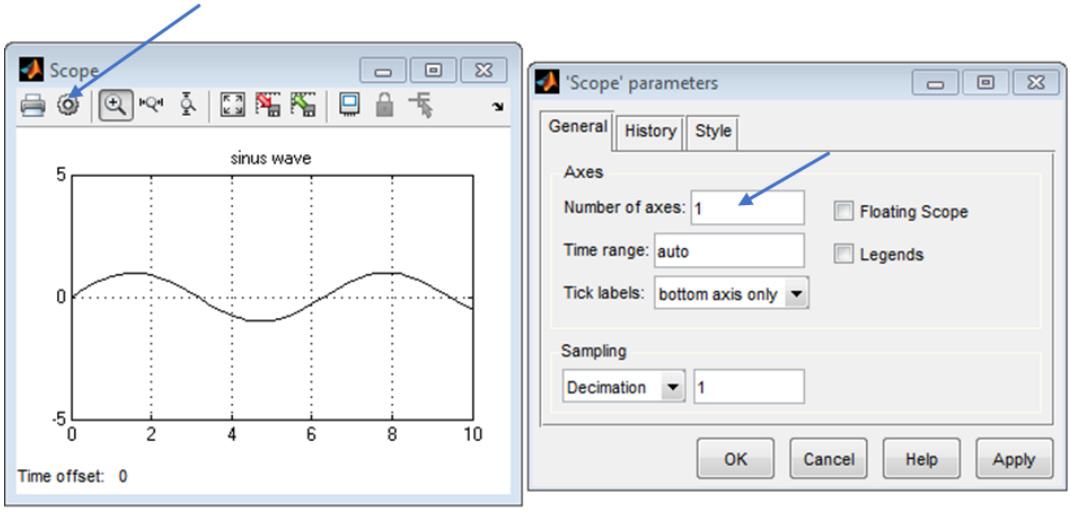


Abbildung 19: Darstellung des Eingangssignals durch ein Scope

Math Operations, Logical Operations

Mittels der Math Operations lässt sich ein Signal mathematisch und logisch verändern oder mit anderen vergleichen.

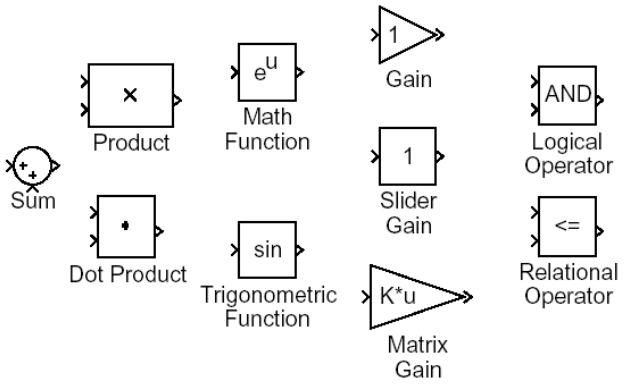
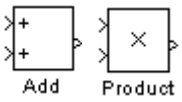
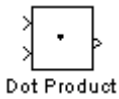


Abbildung 20: Beispiele für Math Operations

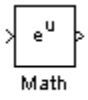


Diese Blöcke dient zum aAdieren/Subtrahieren, bzw. Multiplizieren/Dividieren zweier Signale. Mehrere Eingänge können durch Hinzufügen von Operatoren in den Blockeigenschaften erzeugt werden.

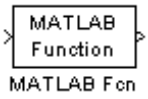


Dieser Block erzeugt das Skalarprodukt zweier Vektoren.

Mit diesem Block lassen sich Signale verstärken oder dämpfen (Dies entspricht der Multiplikation des Eingangswertes mit einer Konstanten).

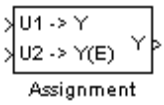


Ein Block um verschiedene Funktionen auf das Eingangssignal anzuwenden. Die Funktion kann auch hier in den Blockeigenschaften ausgewählt werden.



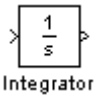
Mit diesem Block können MATLAB-Funktionen auf ein Signal angewendet werden. Das Signal wird dabei immer mit *u* bezeichnet.

Dieser Block kann die verschiedenen trigonometrischen Funktionen ausführen.



Mit Hilfe dieses Blocks kann ein einzelner Wert innerhalb eines Vektors verändert werden.

Dies entspricht der Anweisung `Y(E) = X`.



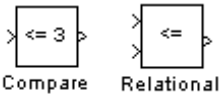
Dieser Block bildet das Integral über ein Eingangssignal.

Mit Hilfe der Einstellungen kann das Integral auf einen bestimmten Wertebereich limitiert werden, sowie einen „Resetport“ erhalten, mit dem das Integral auf seinen Startwert („Initial condition“) zurückgesetzt werden kann.

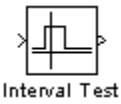
Für eine physikalische Eigenschaft eines Objekts immer nur ein Integral verwenden!

Eine Ableitung (Differentiation) des Eingangssignals wird mit dem Derivate-Block durchgeführt.
Beispiel: Berechnung des Geschwindigkeitsverlaufs aus dem Verlauf der zurückgelegten Wegstrecke.

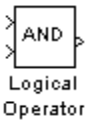
Mit Hilfe dieses Blocks kann die Summe der Elemente eines Vektors gebildet werden.



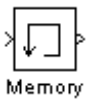
Zwei Blöcke, um Vergleiche mit Signalen durchzuführen. Dabei stellt der obere Eingang die linke und der untere die rechte Seite der Ungleichung dar.



Testet, ob das Signal zwischen zwei Konstanten ist.

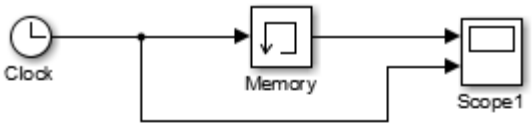


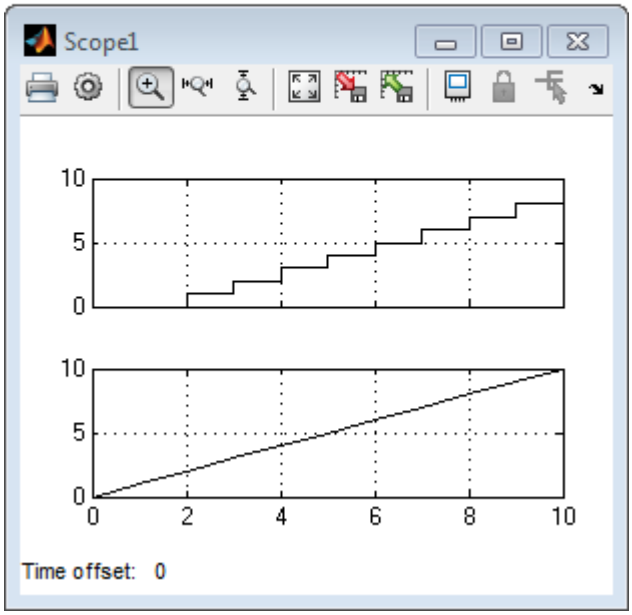
Dieser Block kann diverse logische Verknüpfungen realisieren. Der Verknüpfungstyp kann in den Blockeigenschaften eingestellt werden.



Dieser Block speichert den Signalwert und gibt ihn im nächsten Simulationsschritt wieder aus (das Signal wird um einen Simulationsschritt verzögert).

Einfaches Beispiel mit einer Simulationsschrittweite **von einer Sekunde (Fixed Step: 1 sec, Standardeinstellung 0.01 sec!!!)**





Hinweis:

Der Ausgabedatentyp logischer Operationen ist „bool“, das heißt eine nachfolgende Multiplikation mit *zehn* bleibt *null* oder *eins* (`10 · true = 1`)!

Folglich ist in einem solchen Fall eine Datentypkonvertierung nötig.

Signal Routing

In Abbildung 21 sind alle Funktionsblöcke zur Datenspeicherung und Datenmanagement sowie zur Verknüpfung und Auswahl von Signalen dargestellt.

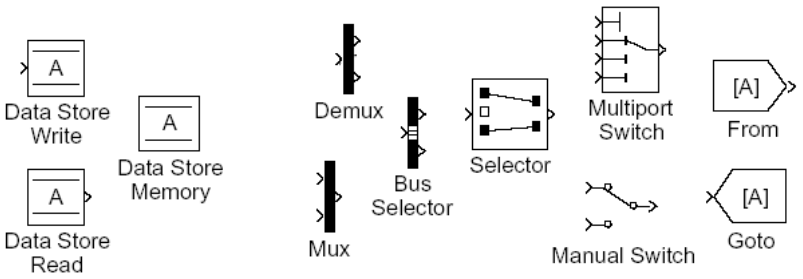


Abbildung 21: Beispiele für Signal Routing

Wählt aus einer Matrix oder einem Vektor bestimmte Elemente aus. Diese Auswahl kann intern oder auch extern indiziert werden.

Dieser Block bildet aus seinen Eingängen einen Vektor, dabei können die Eingänge selbst auch Vektoren sein.

Extrahiert aus einem Vektor dessen Elemente (Umkehrung zum Mux).

Mux

Mux kann verwendet werden um mehrere Eingänge in einem Graphen im Scope anzeigen zu lassen

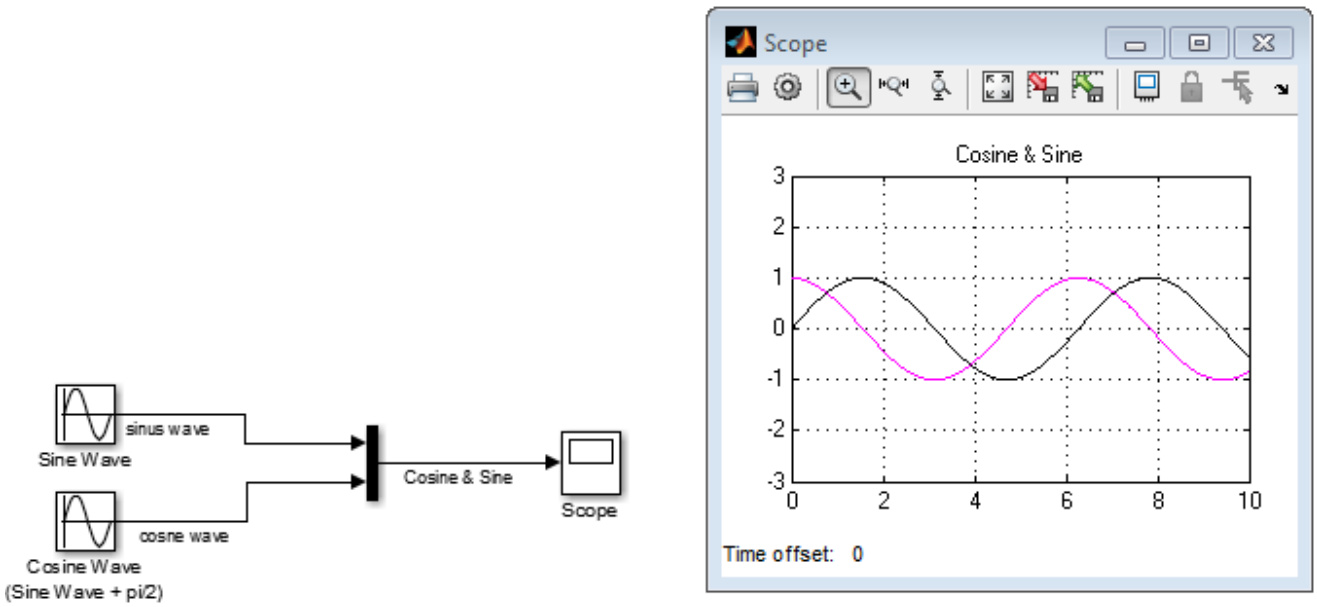


Abbildung 22: Kosinus - und Sinus in einem Graphen des Scopes

Signal Attributes

In Abbildung 23 sind alle Funktionsblöcke zum Ermitteln von Signaleigenschaften oder zum Ändern von Signaleigenschaften dargestellt.

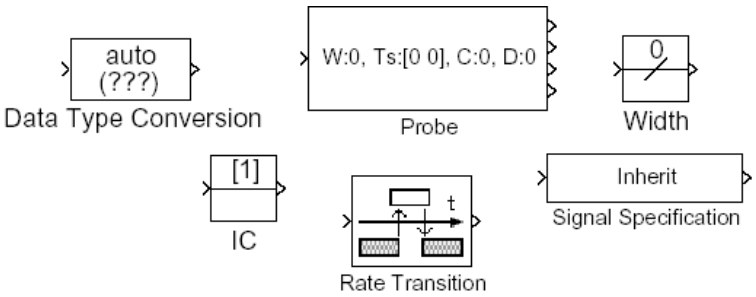
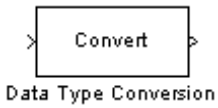


Abbildung 23: Beispiele für Signal Attributes



Führt einen Datentyp in einen anderen über. Die Konvertierung findet häufig Anwendung bei logischen Operatoren.

Subsysteme

Subsysteme dienen zum übersichtlichen Gestalten komplexer Modelle. Durch sie kann man aus mehreren Funktionsblöcken und deren Verknüpfungen einen Block gestalten, der das gleiche Verhalten besitzt.

- Dadurch ist ein Aufbau in hierarchischen Strukturen möglich (Systemgedanken).
- Um Schnittstellen nach außen zu erhalten, sind In- und Out-Ports im Subsystem anzulegen.
- Man kann jedoch auch automatisch ein Subsystem erzeugen lassen, indem man alle Blöcke und Verbindungen markiert, die in das Subsystem enthalten soll, auf ein markiertes Element mit der rechten Maustaste klickt und aus dem Kontextmenü Create Subsystem auswählt.
- Bereits bestehende Subsysteme können durch einen Doppelklick geöffnet werden.

<u>Normale Subsysteme</u>

Dienen zur besseren Übersicht und zur einfacheren Handhabung einzelner Teilsysteme. Diese Systeme laufen mit der gleichen Simulationsschrittweite wie das übergeordnete System ab (Abbildung 24).

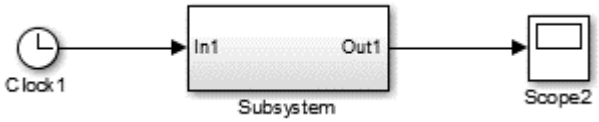


Abbildung 24: Beispiel eines Modells mit einem Subsystem

Durch einen Doppelklick auf das Subsystem kann dieses geöffnet werden. Die Eingangsports werden mit „In“ die Ausgangsports durch „Out“ gekennzeichnet (Abbildung 25).

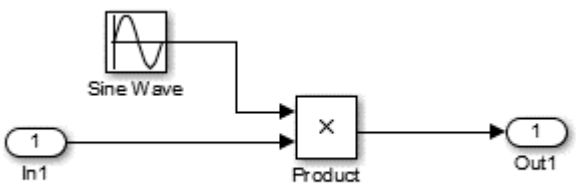


Abbildung 25: Beispiel für den Inhalt eines Subsystems

Getriggerte Subsysteme

Getriggerte Subsysteme verfügen zusätzlich über einen Triggereingang. Dieser überwacht das anliegende Signal auf auftretende Flanken. Dieser Triggereingang löst ein **Event** aus, das einen **einmaligen Durchlauf** des Systems bewirkt.

Dieses Event kann sowohl bei einer steigenden Flanke (*rising edge*), einer fallenden Flanke (*falling edge*) oder durch beide (*both*) ausgelöst werden (Abbildung 26 und Abbildung 27). Die Einstellung hierfür wird in den Blockparametern des „Trigger“-Blocks vorgenommen (Abbildung 27). Durch Doppelklick auf den Block gelangt man zum Parameterfenster (Abbildung 28).

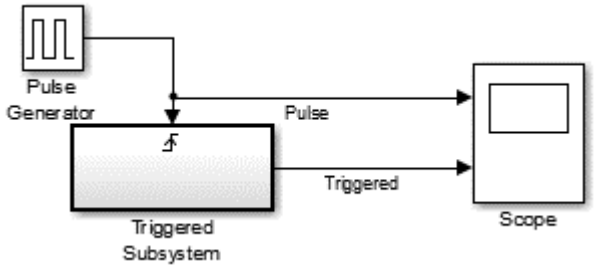


Abbildung 26: Beispiel eines Modells mit getriggertem Subsystem

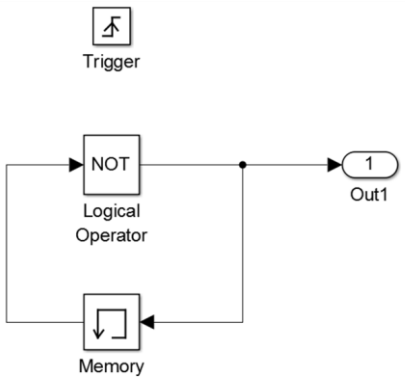


Abbildung 27: Beispiel für den Inhalt eines getriggerten Subsystems

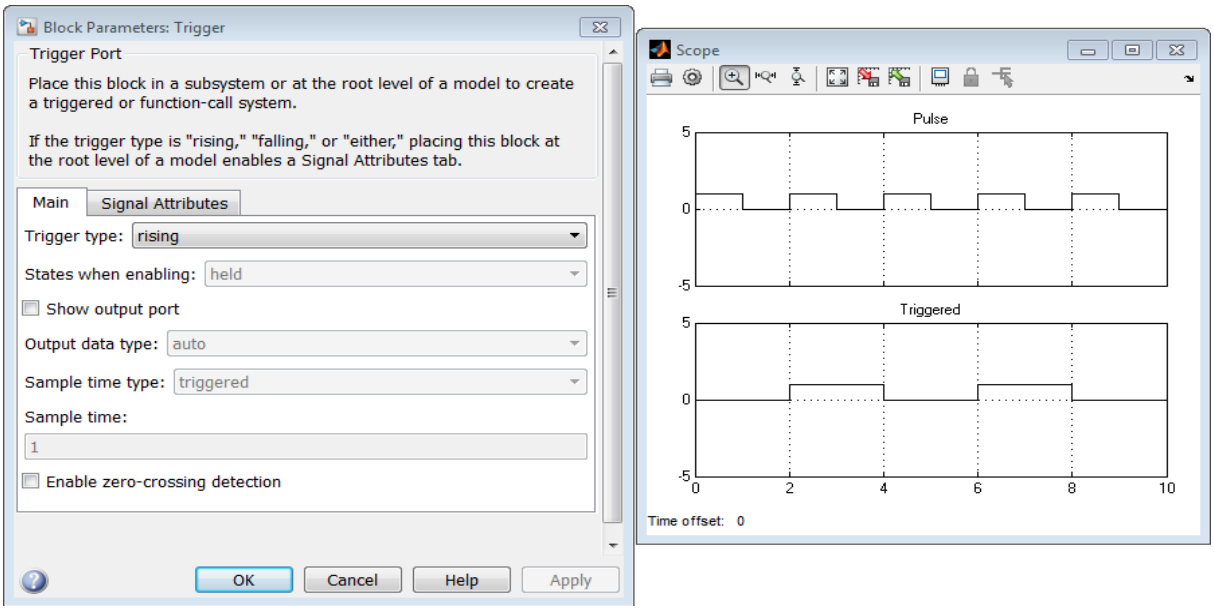


Abbildung 28: Einstellungsdialog des Triggers und Ausgabe des Beispiels

Enabled Subsysteme

Ein Enabled Subsystem besitzt zusätzlich einen Enabler-Port. Das Subsystem wird **solange** ausgeführt, wie das **Enable-Signal True** ist (>0) (Abbildung 29 und Abbildung 30).

Solange ein Eingangssignal anliegt, wird die Funktion im Enabled Subsystem entsprechend des eingestellten Simulationstakts wiederholt.

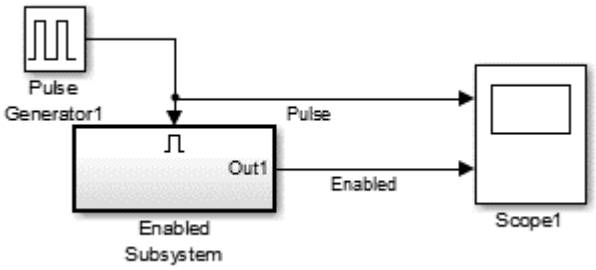


Abbildung 29: Beispiel eines Modells mit Enabled Subystem

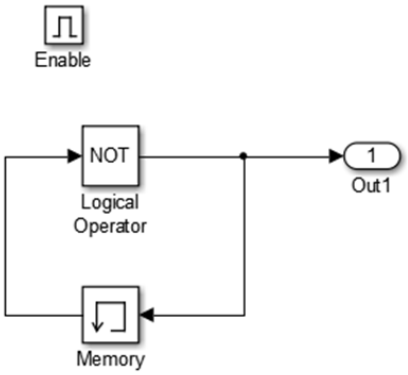


Abbildung 30: Beispiel für den Inhalt eines Enabled Subsystems

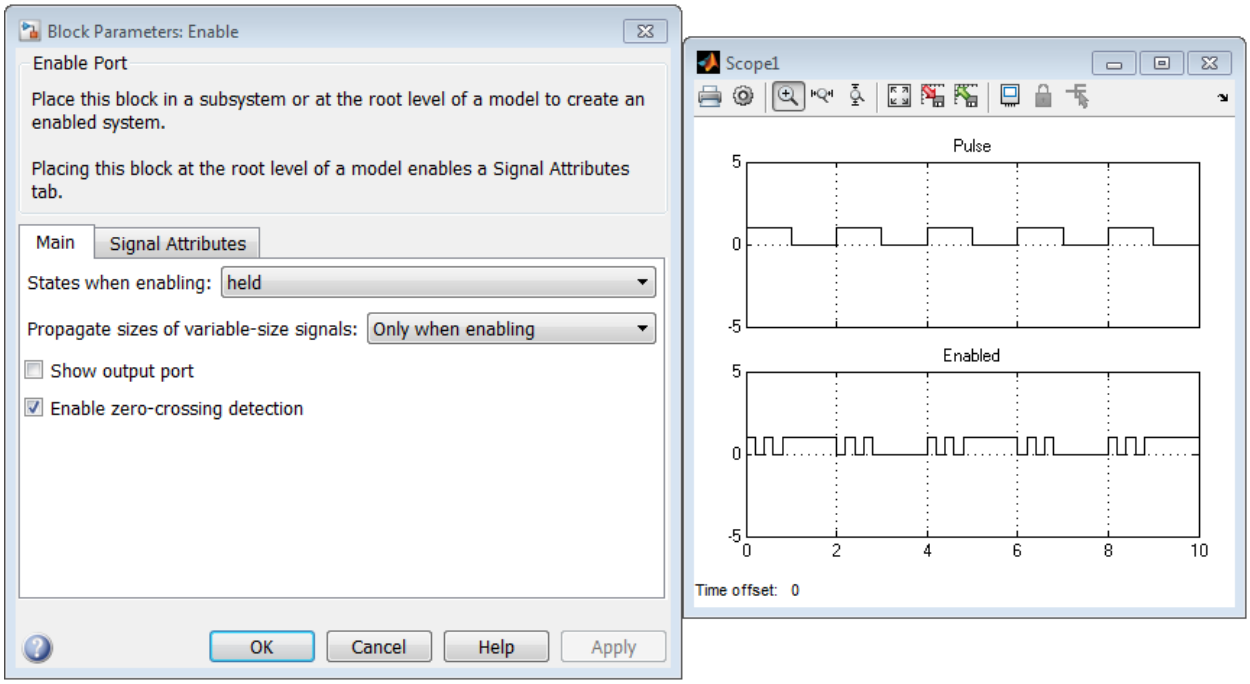


Abbildung 31: Einstellungsdialog und Ausgabe des Beispiels

4.1.2 Das Modellfenster

Abbildung 32 zeigt ein Beispiel eines fertigen Simulink-Modells

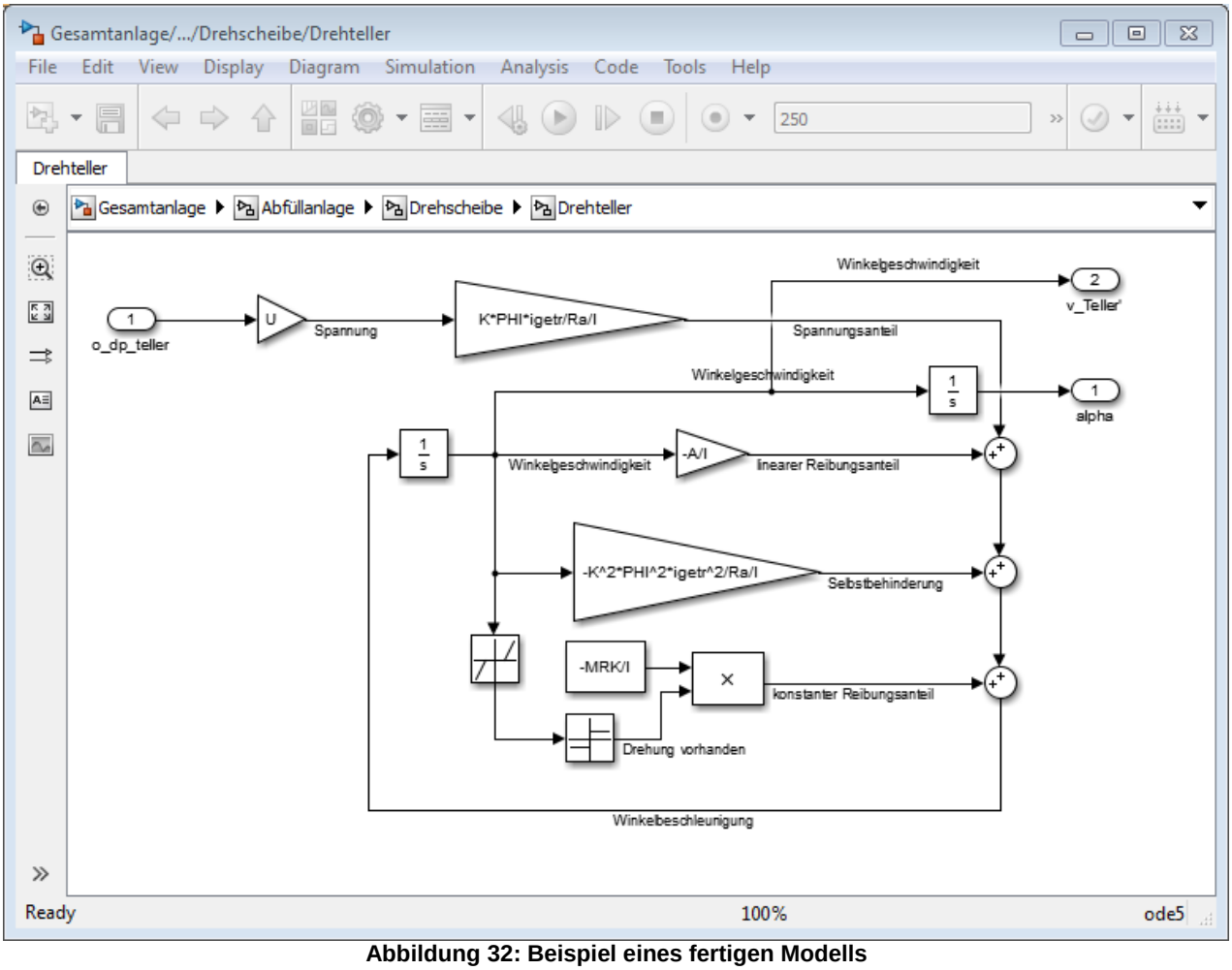


Abbildung 32: Beispiel eines fertigen Modells

Durch einen Klick auf dieses Symbol kann das Modell gespeichert werden.

Durch einen Klick auf dieses Symbol kann die Simulation gestartet werden. Nach dem Start verwandelt sich das Symbol in ein Pausezeichen, welches das Anhalten der Simulation ermöglicht.

Durch einen Klick auf dieses Symbol kann man die Simulation abbrechen (nur während der Simulationsausführung möglich).

Durch einen Klick auf dieses Symbol wird der Library Browser geöffnet.

Durch einen Klick auf dieses Symbol wird der Modellbrowser geöffnet. Dieser stellt eine Art Verzeichnisbaum des Modells in dem die Subsysteme Ordnern entsprechen dar.

Durch einen Klick auf das Symbol wird ein neues Simulink Modell geöffnet.

- Im Menüpunkt „Display“ → „Signal & Ports“ können verschiedene Eigenschaften von Signalen, wie den Datentyp („Port Data Types“) oder die Größe („Signal Dimensions“) direkt im Modell angezeigt werden. Es empfiehlt sich diese anzeigen zu lassen, da dies häufige Fehlerquellen sind.

4.1.3 Simulationsparameter

Mittels der Simulationsparameter lassen sich alle für die Simulation wichtigen Eingaben tätigen. Hierunter fallen die Startzeit und das Ende der Simulation, die Auswahl des Algorithmus zur numerischen Integration, die Genauigkeit, die Schrittweiten und das Verhalten bei Fehlern.

Die Einstellung der Simulationsparameter erfolgt im Modellfenster unter dem Menüpunkt „Simulation“ → „Modell Configuration Parameters“ oder über die Beschleunigertastenkombination Strg+E (Abbildung 33).

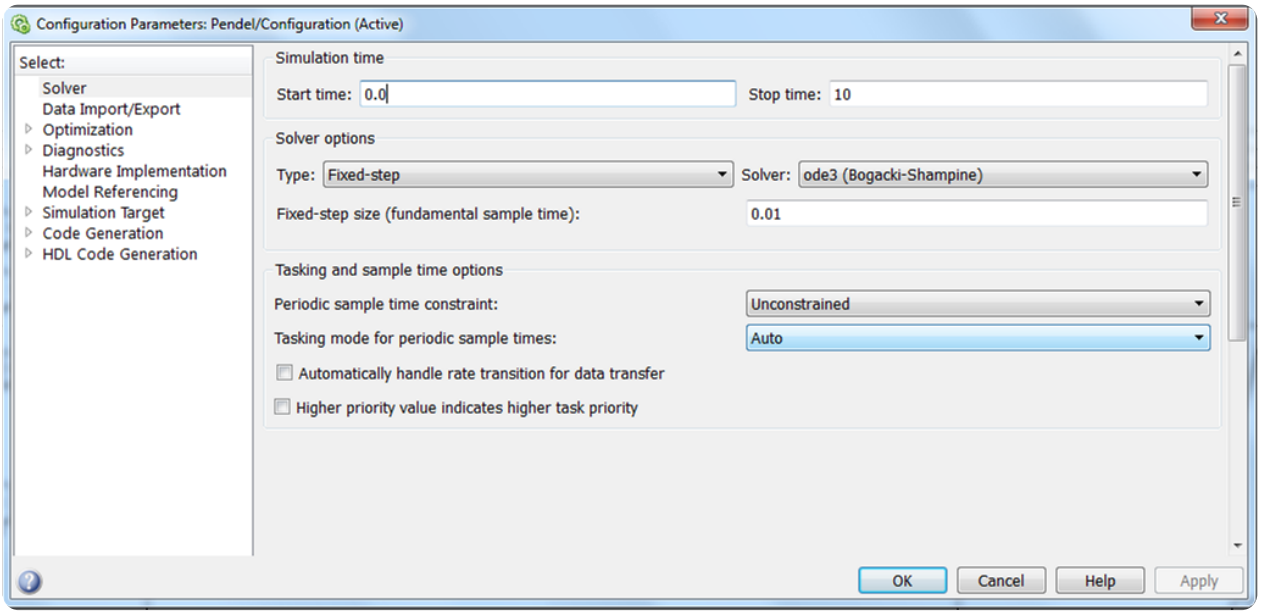


Abbildung 33: Der Simulationsparameterdialog

Simulation time: Einstellung der Start und Endzeit der Simulation

Solver options: **Type:** **Fixed-step**

Solver: **ode3 (Bogacki-Shampine)**

Fixed-step size: **0.01**

Hier muss während des Praktikums immer „Fixed-step“ und eine maximale Schrittweite von 0.01[s] eingestellt werden, da es ansonsten zu Fehlern in der Modellberechnung kommen kann.

Hintergrund:

Das Problem in der kontinuierlichen Simulation besteht darin, dass ein Rechner nur diskret arbeiten kann. Um ein möglichst gutes Ergebnis zu erhalten, kann die Schrittweite verkleinert und der Fehler verringert werden. Dadurch erhöht sich Rechenaufwand und -zeit (Fixed-step). Eine weitere Möglichkeit besteht darin, die Schrittweite an kritischen Punkten zu verkleinern und an unkritischen Punkten zu vergrößern (Variable-step). Dadurch erhält man ein zufriedenstellendes Ergebnis ohne eine Erhöhung der Rechenzeit in Kauf nehmen zu müssen. Um die kritischen Punkte zu erhalten, ist bei Variable-step eine Fehlerüberwachung und eine Zero-crossing control durchzuführen. Die Genauigkeit kann man auch durch Ändern der Ansatzfunktion erhalten (z. B. kubisch statt quadratisch). Dies geschieht in Simulink durch die Auswahl der verschiedenen Solver. Mit der Ansatzfunktion wird die zu berechnende Funktion in einem kleinen Teil angenähert. Je höher der Grad einer Ansatzfunktion ist, desto besser kann sie sich potentiell an die Ursprungsfunktion anschmiegen. Es sind aber dadurch auch mehr Rechenoperationen nötig.

4.1.4 Diagnosefenster

Sollten vor und während der Simulation Fehler auftreten, werden diese im Diagnosefenster ausgegeben. Die Fehler werden durch deren Art, Ursache, Ort und Komponente beschrieben (Abbildung 34).

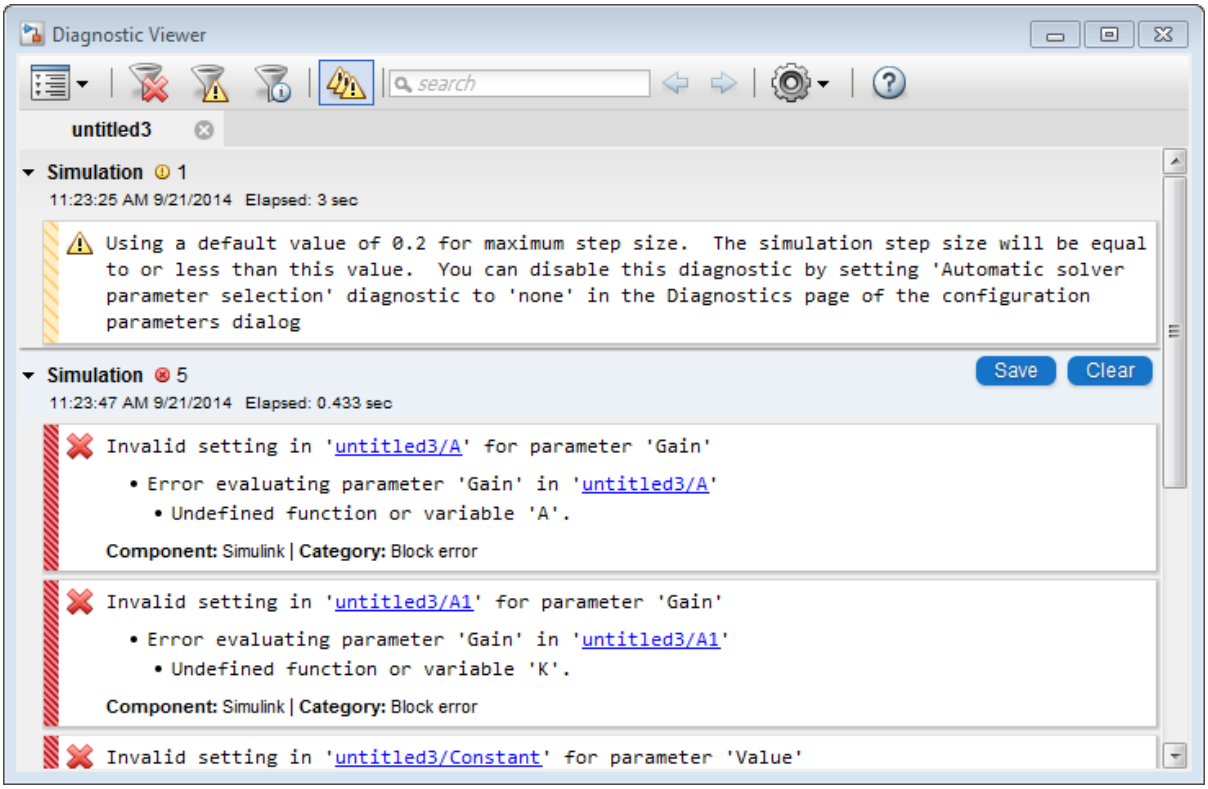


Abbildung 34: Das Diagnosefenster

- Man findet hier eine Auflistung aller aufgetretenen Fehler und Warnungen.
- Bei jedem Fehler oder jeder Warnung findet man eine detaillierte Beschreibung des ausgewählten Fehlers mit Link auf den Fehler verursachenden Block.
- Solange das Diagnosefenster geöffnet ist, sind Blöcke, die Fehler verursachen, innerhalb des Modells farblich markiert. Die blauen Textteile in der Fehlerbeschreibung sind anklickbar und führen direkt zum fehlerhaften Block.
- Typische Fehler
    - Fehlerhafter Datentyp, oder eine
    - Falsche Signalgröße
    - Syntaxfehler

Hinweis:

Es ist wichtig, die Fehler oder Warnung zu lesen und interpretieren zu können. Fehlersuche und -behebung sind ein fundamentaler Bestandteile beim Arbeiten mit MATLAB, Simulink und Stateflow!

4.2 Vorgehen zur Erstellung eines kontinuierlichen Systems

Anhand der folgenden Beispiele wird das prinzipielle Vorgehen zum Erstellen eines Modells dargestellt. Es soll hier nicht der schnellste, sondern ein möglichst allgemeingültiger Weg aufgezeigt werden.

4.2.1 Federpendel

In diesem Kapitel soll ein Federpendel modelliert werden. Es handelt sich dabei um eine ungedämpfte, freie Schwingung mit normierter Einheitsfrequenz $\omega_0 = 1$. Am Anfang ist das Pendel in Ruhe und um x = 1 ausgelenkt.

Aufstellen des mathematischen Modells

Mittels verschiedener Lösungsmethoden (Lagrange, Newton-Euler,...) kann man die Differentialgleichung des Systems erhalten. In unserem Fall lautet diese

$\ddot{x}(t) + {\omega_0}^2 \cdot x(t) = 0$ Anfangsbedingungen: $\ddot{x}(t=0) = 0; x(t=0) = 1$

Umschreiben zur Verwendung in Simulink

Um die Differentialgleichung in Simulink zu modellieren, wird diese nach der höchsten Ableitung aufgelöst:

$\ddot{x}(t) = -{\omega_0}^2 \cdot x(t)$

Übertragung zu Simulink

Die Übertragung des mathematischen Modells in Form von Differentialgleichungen erfolgt in 3 Schritten:

1. Zunächst erfolgt eine zweifache Integration der Beschleunigung $\ddot{x}(t)$ mit Hilfe zweier Integrator-Blöcke, an deren Ende man die Strecke $x(t)$ erhält.

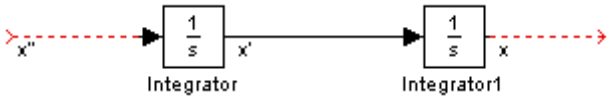


Abbildung 35: Blockschaltbild zu Schritt 1

2. Es wird nun sukzessive die rechte Gleichungsseite der umgeformten Differentialgleichung mit Hilfe von Integratoren, Additionsstellen etc. aufgebaut. Im vorliegenden Falle wird der Ausgang des zweiten Integrators $x(t)$ im Block Gain mit $-{\omega_0}^2$ multipliziert.

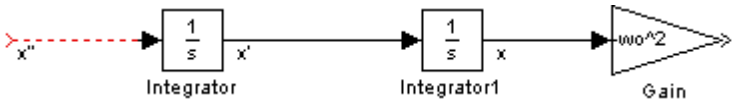


Abbildung 36: Blockschaltbild zu Schritt 2

3. Zum Schluss erfolgt die Rückkopplung. Da die Verstärkung (Block Gain) der zweifachen Differentialgleichung (Integrator) wieder die Beschleunigung $\ddot{x}(t)$ darstellt, wird der Gainausgang mit dem Integratoreingang verbunden.

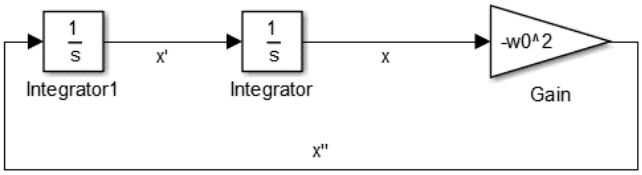


Abbildung 37: Blockschaltbild zu Schritt 3

Randbedingungen einbringen Nun sind noch die Anfangsbedingungen der beiden Integratoren zu setzen. Ausgang des letzten Integrators ist der Weg x. Also wird dieser durch Doppelklicken auf den zweiten Integrator und Eintrag des Wertes 1 in das Feld „Initial condition“ mit der Anfangsbedingung initialisiert. Der erste Integrator liefert die Geschwindigkeit. Dieser Block wird mit der Anfangsgeschwindigkeit 0 initialisiert.

4.2.2 Drehtellerbewegung

Der Drehteller ist eine Scheibe, auf der die Flaschen abgestellt werden können. Durch die Drehbewegung des Tellers können Flaschen auf einer Kreisbahn transportiert werden.

Ein- und Ausgangsgrößen

Vor der Modellierung eines Systems sind die Ein- und Ausgangsgrößen des zu modellierenden Systems zu bestimmen.

- Der Drehteller verfügt über die Eingangsgröße o_teller (entspricht dem logischen Ausgangssignal der Ansteuerung (hier „Pulse Generator“), das angibt, ob die Stromzufuhr zum Motor an- oder ausgeschaltet ist).
- Die Ausgangsgrößen sind Drehgeschwindigkeit $\dot{\omega}$ und den Drehwinkel ω.

Aufstellen der Bewegungsgleichung

Im vorliegenden Beispiel stellt sich diese Wirkkette wie folgt dar: Das Eingangssignal schaltet ein Relais (Verstärker). Die Relaisspannung wird an einen Elektromotor angelegt. Dieser wandelt die elektrische Energie in ein Moment um. Das Moment treibt den Drehteller an. Die Drehtellerbewegung ergibt, über die Zeit integriert, den aktuellen Drehwinkel α, der die Ausgangsgröße darstellt. Die Bewegungsgleichungen werden über das Momentengleichgewicht von Teller und Elektromotor ermittelt.

Momentengleichgewicht des Tellers:

$$M_{el} \cdot i = I \cdot \dot{\omega} + M_R = I \cdot \dot{\omega} + \mathrm{sgn}(\omega) \cdot M_{RK} + A \cdot \omega$$

M_{el}	Drehmoment des Elektromotors
i	Gesamtübersetzung
I	Trägheitsmoment des Drehtellers
$\omega = \dot{\alpha}$	Drehgeschwindigkeit
$\dot{\omega}$	Drehbeschleunigung
M_R	Reibmoment (=MRK + A ω)
M_{RK}	konst. Reibmoment
A	geschwindigkeitsabhängige Reibung
sgn	Signumfunktion $\mathrm{sgn}(x) = \frac{\lvert x \rvert}{x}$ *für alle* $x \neq 0$, *sonst* $\mathrm{sgn}(x=0) = 0$

Momentengleichung eines Gleichstrommotors:

$$M_{el} = \frac{K \cdot \Phi}{R_a} \cdot \left(U - K \cdot \Phi \cdot \Omega\right)$$

K	Wicklungsanzahl
Φ	Hauptfluss im Elektromotor

R_a Ankerwiderstand des Elektromotors

U angelegte Spannung

Ω Drehzahl des Elektromotors

Koppelbedingung: $\Omega = \omega \cdot i$

Übertragen der Bewegungsgleichung nach Simulink

Um die Bewegungsgleichung nach Simulink zu übertragen, wird diese nach der höchsten, abgeleiteten Größe aufgelöst:

$$\dot{\omega} = \frac{M_{el} \cdot i - \mathrm{sgn}(\omega) \cdot M_{RK} - A \cdot \omega}{I} = \frac{\frac{K \cdot \Phi \cdot i}{R_a} \cdot (U - K \cdot \Phi \cdot \omega \cdot i) - \mathrm{sgn}(\omega) \cdot M_{RK} - A \cdot \omega}{I}$$

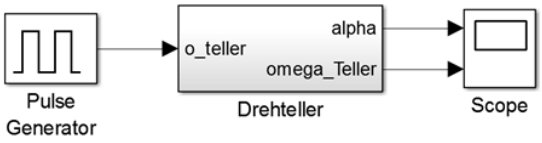


Abbildung 38: Aufbau des Drehtellers

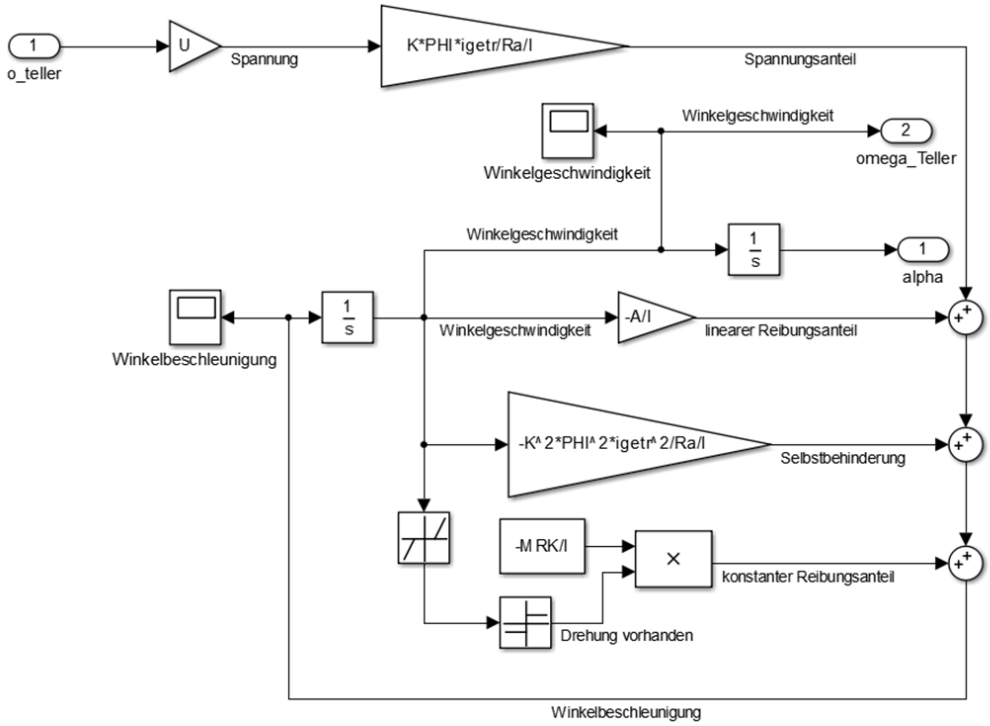


Abbildung 39: Aufbau des Drehtellers

Vor der Signumfunktion wurde ein Tot**zonen**glied (nicht ein Tot**zeit**glied) eingefügt.

Randbedingungen festlegen

Da sich die Anlage zu Beginn weder dreht noch irgendwelche Kräfte wirken, sind die Drehbeschleunigung, die Drehgeschwindigkeit und der Drehwinkel initial mit Null anzusetzen.

5. Stateflow

In diesem Teil des Versuchs werden Systeme behandelt, deren dynamischer Verlauf nicht zeit-, sondern **ereignisgesteuert** ist. Die möglichen Zustände, in denen sich das System befinden kann, sind **diskrete** Zustände, wie z. B. ein- oder ausgeschaltet. Zum **Wechseln zwischen den Zuständen** sind **Ereignisse** notwendig. Dies stellt einen großen Unterschied zu den *kontinuierlichen* Systemen da, die Sie gerade eben im Simulink-Teil kennengelernt haben.

Die auftretenden Ereignisse sind meist Eingangssignale des Systems, z. B. ein **Sensorsignal**. Es können jedoch ebenso Zustände aus einem anderen Programmteil abgefragt werden (Beispiel: Schalte Warnlampe an, wenn in der Steuerung der Zustand „Fehler“ aktiv ist.). Zur Modellierung dieser Art von Systemen stehen neben den universellen textbasierten Programmiersprachen auch graphische Modellierungstechniken zur Verfügung, z. B. Petri-Netz-Modelle oder Zustandsautomaten.

Für die Modellierung zustandsbasierter Systeme, wie sie beispielsweise auch Steuerungen darstellen, bietet sich die Verwendung von *Stateflow* an. Die Grundlagen von Stateflow werden auf den folgenden Seiten näher erläutert.

5.1 Grundlagen

In diesem Abschnitt wird die Implementierung von Zustandsautomaten mittels Stateflow beschrieben. Aufgrund des sehr großen Funktionsumfanges dieser Software wird nur auf die für dieses Praktikum unmittelbar relevanten Bestandteile eingegangen. Weiterführende Informationen sind geeigneter Fachliteratur bzw. der Dokumentation in der Hilfefunktion zu entnehmen.

Stateflow ist eine zusätzliche Bibliothek von Simulink, deshalb können Stateflow-Charts nur **innerhalb eines Simulink-Modells** ausgeführt werden. Die Stateflow-Library kann durch Anklicken des entsprechenden Symbols im Simulink-Library Browser oder durch Eingabe von einem der folgenden Befehle im MATLAB-Command-Window geöffnet werden.

```
>> sf

>> stateflow
```

Die Stateflow-Library (Abbildung 40) enthält verschiedene Blöcke, für dieses Praktikum ist jedoch nur der Block „Chart“ relevant. Dieser kann durch Drag‘n‘Drop in ein bestehendes Simulink-Modell gezogen werden. Bei Doppelklick auf das leere Chart öffnet sich der Editor von Stateflow (Abbildung 41).

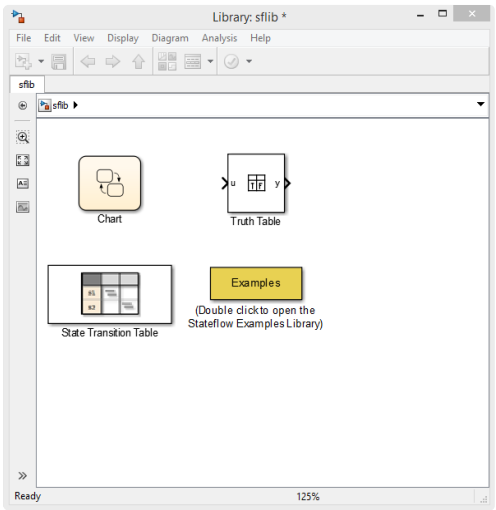


Abbildung 40: Stateflow-Bibliothek

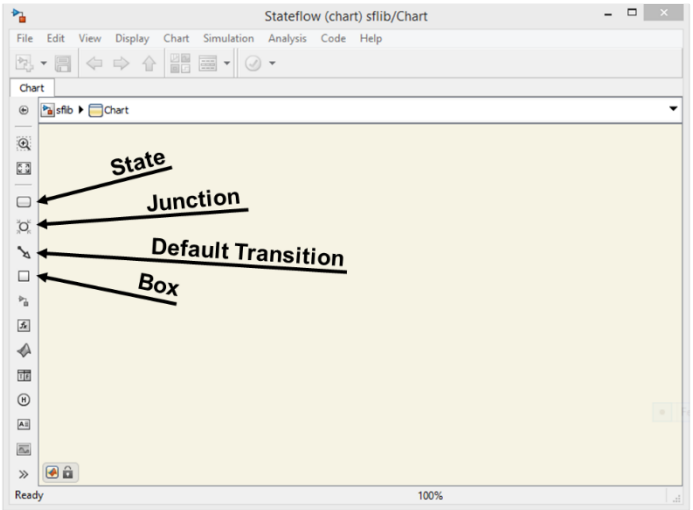


Abbildung 41: Stateflow-Chart

5.1.1 Elemente eines Zustandsautomaten

In den folgenden Abschnitten werden die in Stateflow verfügbaren Elemente beschrieben. Abbildung 42 gibt einen Überblick über den möglichen Aufbau eines Zustandsautomaten in Stateflow. Beachten Sie, dass es sich hierbei um ein Beispiel ohne Funktion handelt.

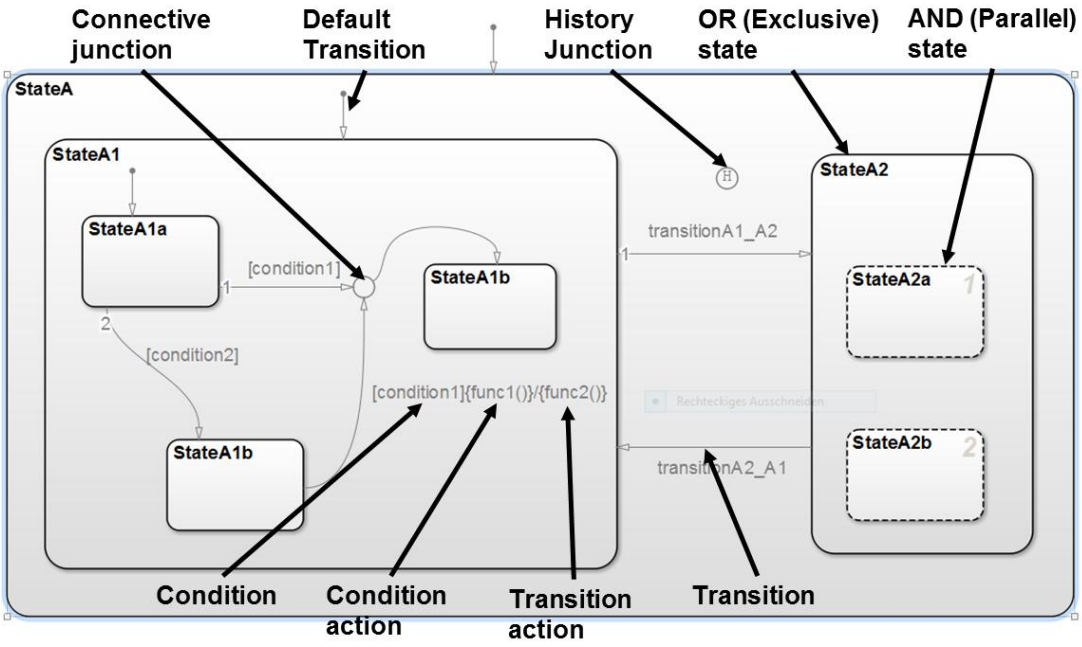


Abbildung 42: Elemente eines Charts in Stateflow

Zustände (states)

Ein „State“ stellt einen möglichen Zustand dar, in welchem sich ein dynamisches System befinden kann. Ein „Chart“ kann beliebig viele Zustände enthalten. Ein Zustand kann wiederum einen Zustandsautomaten – also weitere States – beinhalten, wodurch eine **Hierarchisierung** und **bessere Übersichtlichkeit** des Zustandsautomaten erreicht werden kann.

Ein Zustand kann durch Anklicken des entsprechenden Symbols der Werkzeugleiste im Chart platziert werden (siehe Abbildung 41). Jeder Zustand muss benannt werden. Dazu wird das Symbol „?“ mit den gewünschten Namen ersetzt. Es werden zwei Arten von Zuständen unterschieden, die im Folgenden erläutert werden (**für die grafische Darstellung siehe auch Abbildung 42**).

Zustandsarten	**(OR) Exclusive states**	**(AND) Parallel states**
Beschreibung	Nur ein einziger Zustand in der gleichen Hierarchieebene ist zu einem Zeitpunkt aktiv. Es muss pro Hierarchieebene immer einen aktiven Zustand geben. Der bei Beginn der Simulation aktive State muss mit einer Default-Transition gekennzeichnet werden. (mehr dazu auf S. 37)	Alle Zustände der Hierarchieebene sind gleichzeitig aktiv.
Darstellung	durchgezogene Rechtecke	gestrichelte Rechtecke
Einstellung	Klicken Sie mit der rechten Maustaste auf den markierten Zustand. Unter *Decomposition* wählen Sie die gewünschte Art von Zustand (Standardeinstellung: „OR State“). **Achtung**: Die Gliederung in exklusive oder parallele Zustände bezieht sich auf die im gewählten State beinhalteten Zustände, nicht auf den markierten Zustand!	

Tabelle 1: Parallele und exklusive Zustände

Ein komplettes Label, also die Beschriftung eines Zustands, enthält den Namen sowie die auszuführenden Aktionen. Durch **Anklicken der Beschriftung im jeweiligen State lässt sich das Label eingeben** (auch möglich mit der rechten Maustaste auf den markierten Zustand klicken und dann unter „Properties…“).

Abbildung 43 zeigt das Label eines Zustands und das zu diesem State zugehörige Properties-Fenster.

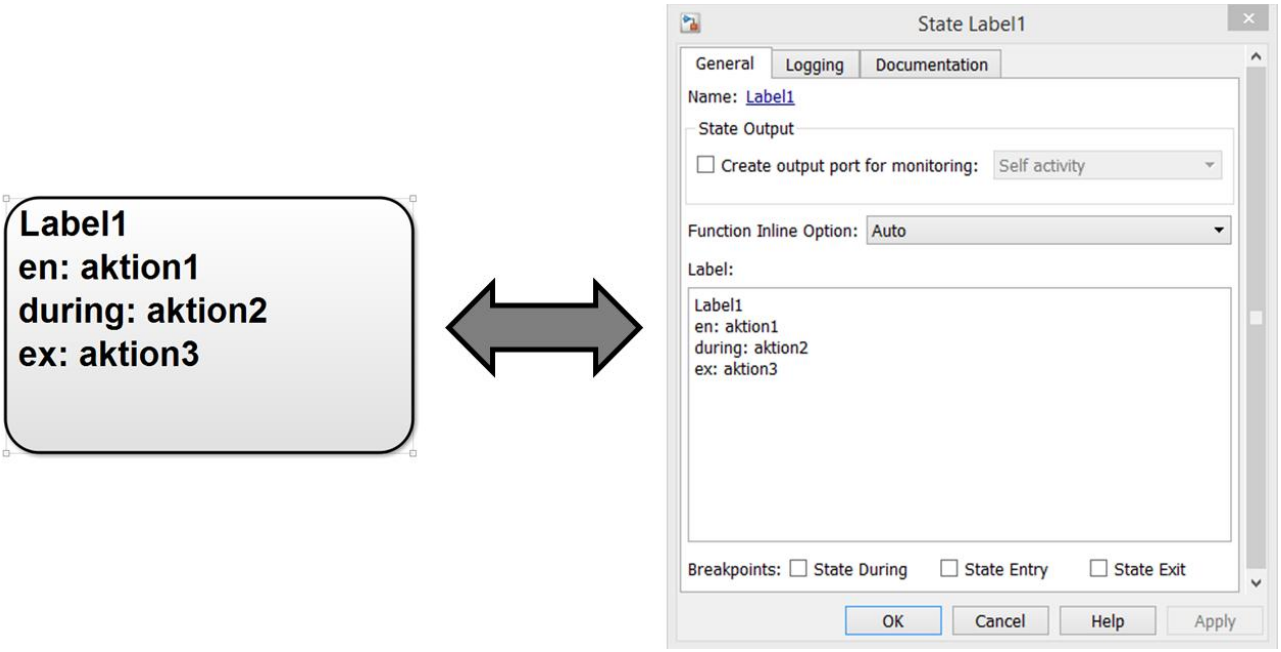


Abbildung 43: Zustand und Properties-Fenster

Hinweis: Der Name eines States muss im Modell einmalig sein und darf weder Leerzeichen noch Umlaute und Sonderzeichen enthalten. Alles andere führt zum Abbruch der Simulation. Um den Inhalt eines States dennoch klar zu bezeichnen, bietet sich ein Name mit Unterstrichen an.

Beispiel für zulässige und aussagekräftige Benennung eines States: *tuer_auf*

Die Aktionen werden in der sogenannten *„Action Language“* implementiert. **Wenn ein Zustand aktiv wird, werden alle im Label enthaltenen Aktionen (z. B. ändern des Wertes einer Variable) ausgeführt**. Diese Aktionen werden bei der Aktivierung (*entry*) (Standardverhalten), während der Aktivität (*during*) oder beim Verlassen des Zustands (*exit*) ausgeführt. (weitere Informationen zur Action Language folgen im Kapitel 5.1.5 auf Seite 43).

Transitionen (transitions)

Eine Transition stellt einen **Übergang zwischen zwei States** im Stateflow-Chart dar. Eine Transition wird durch **Anklicken des Randes** der ersten States mit der linken Maustaste und **Ziehen** zum gewünschten State erzeugt. Die Transition erscheint nun als Pfeil (Abbildung 44) zwischen den beiden States und kann auch nur in **eine Richtung** durchlaufen werden. Wird die Transition in Abbildung 44 aktiv, springt der Chart von *State1* in *State2* und setzt die Variable *on* gleich 1.

Abbildung 44: Beispiel einer Transition mit komplettem Label

Über einer neu gezogenen Transition erscheint ein Fragezeichen. Dieses kann angeklickt und durch ein „Transition Label“ ersetzt werden. Eine Transition mit Label wird nur dann durchlaufen, wenn **alle im Label genannten Events und Bedingungen erfüllt** werden. Im Umkehrschluss wird eine **Transition ohne Label immer sofort aktiv**.

In Abbildung 44 ist eine Transition mit allen möglichen Typen von Labels dargestellt. Es ist selbstverständlich möglich auch nur eine Art von Label oder mehrere Labels des gleichen Typs zu verwenden. Die verschiedenen Arten von Labels werden im Folgenden kurz vorgestellt.

Event	Nur wenn das genannten Event (auch mehrere Events sind möglich) auftritt, wird die evtl. vorhandene Bedingung (Condition) auf ihre Gültigkeit geprüft. Mehrere Events können durch Verknüpfung mit logischen Operatoren kombiniert werden. Die Abarbeitung von Events erfolgt in der zeitlichen Reihenfolge des Empfangs.
Condition	Boolscher Ausdruck, z. B. [i_Temperature>20], der zu *true* werden muss, damit die Transition ausgeführt wird. Wenn keine Bedingung im Label steht, wird immer der Wert *true* für die Auswertung angenommen.
Condition Action	Die Aktion wird ausgeführt, sobald die Bedingung und das Event der jeweiligen Transition erfüllt werden. Gibt es keine Bedingung und wird auf kein Event gewartet, wird die Transition sofort aktiv und die Condition Action sofort ausgeführt.
Transition Action	Die Aktion wird erst ausgeführt, wenn ein gültiges Ziel der Transition erkannt wird und alle zum Erreichen notwendigen Events und Bedingungen, falls angegeben, erfüllt werden. Falls eine Transition aus mehreren Segmenten besteht, muss der gesamte Pfad der Transition gültig sein und der State am Ende des Pfades aktiv geworden sein.

Die Deklaration von Events sowie der für die logischen Abfragen in den Conditions nötigen Variablen erfolgt im Data Dictionary. Dies wird in Kapitel 5.1.4 genauer erläutert.

Default transition

Ist eine Hierarchieebene als Exclusive (OR) definiert, stellt sich für Stateflow die Frage, *welcher* der beinhalteten States **als erstes aktiv** ist. Dieser Sachverhalt wird deutlicher bei Betrachtung von Abbildung 45. Im State „On“ gibt es zwei weitere States „Lamp1“ und „Lamp2“. Mit Hilfe des Events *„switch“* lässt sich zwischen den beiden Zuständen wechseln. Hierfür muss Stateflow jedoch wissen, in welchem der beiden States es sich **am Anfang der Simulation** befindet. Diese Aufgabe übernimmt die Default transition, die grafisch als Pfeil mit einem ausgefüllten Punkt erscheint.

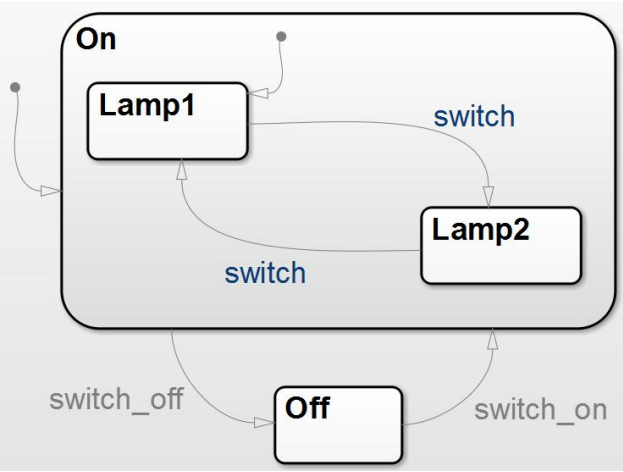


Abbildung 45: Default Transition ohne Label

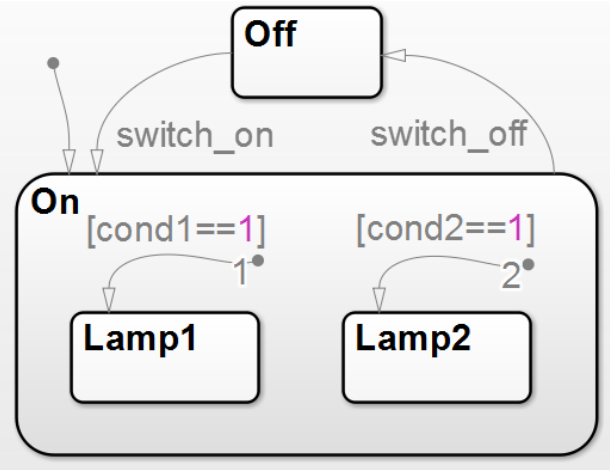


Abbildung 46: Default Transition mit Label

Die Default transition wird immer dann aktiv, wenn ihr Superstate, also die **übergeordnete Hierachieebene** aufgerufen wird. In Abbildung 45 gibt es zwei Default transitions. Die erste Default transition zeigt auf den State „On“ und befindet sich hiermit in der **höchsten Hierarchieebene**. Dies bedeutet sie wird nur **ein einziges Mal beim Start der Simulation** aufgerufen. Danach ist auf dieser Ebene immer entweder „On“ oder „Off“ aktiv. Dadurch ist der **Zustand dieser Ebene eindeutig definiert**. Die zweite Default transition befindet sich **innerhalb des States „On“**. Sie wirkt daher immer dann, **wenn dieser State aktiv** wird.

Hieraus ergibt sich, dass bei einer Exclusive-(OR)-Ausführung immer genau eine aktive Default transition pro Hierarchieebene geben muss.

In Abbildung 46 gibt es innerhalb des States „On“ zwei Default transitions. Wir wissen bereits, dass hiervon **immer exakt eine** aktiv sein muss. Dies ist immer dann der Fall, wenn das Label auf einer Default Transition erfüllt ist. Ist das Label nicht erfüllt, wird diese Default transition von Stateflow so gehandhabt, **als würde sie nicht existieren**. Es darf daher auch auf keinen Fall passieren, dass gar keine Default transition aktiv ist, da in diesem Fall **kein Startpunkt** gefunden wird und die Simulation **abgebrochen** wird.

Auf Grund der hohen Fehleranfälligkeit sollte daher eine Programmierung mit mehreren Default transitions auf einer Ebene vermieden werden.

Verbindungspunkte (Junctions)

Ein Verbindungspunkt wird als ein Kreis dargestellt und kann nach Anklicken des entsprechenden Symbols der Werkzeugleiste in den Chart eingefügt werden. Er stellt eine **Entscheidungsmöglichkeit zwischen mehreren Pfaden einer Transition** bzw. eine **Zusammenführung von mehreren Pfaden einer Transition** dar. Die Transition wird nur dann gültig, wenn ein kompletter Pfad vom Quellzustand zum Zielzustand gültig ist. Zudem kann immer nur ein Pfad gültig sein.

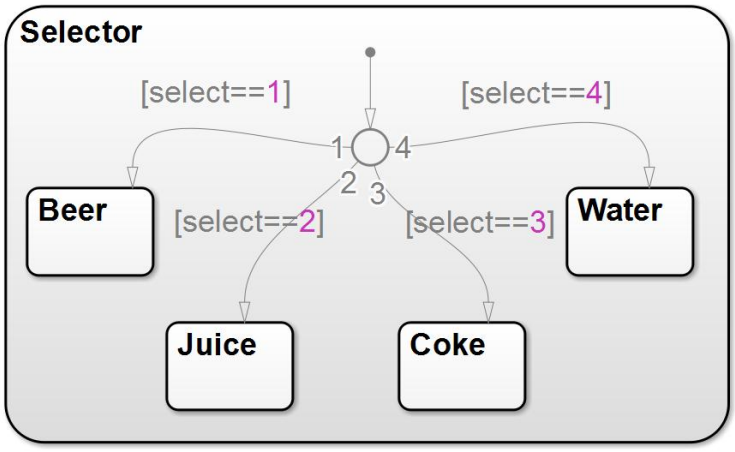


Abbildung 47: Verzweigung mit Connective Junction

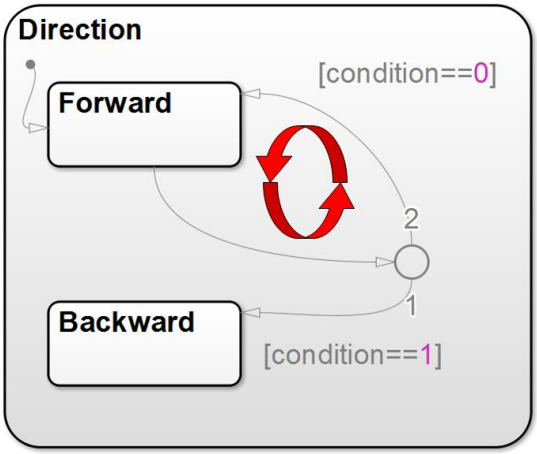


Abbildung 48: Self Loop Transition

Abbildung 47 stellt ein Beispiel dar, in dem eine Default transition zu einem Verbindungspunkt führt. Dieses Beispiel entspricht also einer aus der textbasierten Programmierung bekannten if-Abfrage.

Abbildung 48 zeigt den Fall einer **„self-loop-transition“**. Eine self-loop-transition ist eine Transition, bei der Quell- und Zielobjekt identisch sind. Dies sollte man **unter allen Umständen vermeiden**, da die Gefahr besteht, dass die so entstandene Schleife in jedem Simulationsschritt durchlaufen wird, was zu hoher Prozessorlast und damit einer **sehr starken Verlangsamung der Simulation** führt.

5.1.2 Hierarchisierung und Beherrschung von Komplexität

Bei der Simulation komplexer Systeme treten eine Vielzahl verschiedener Zustände auf, die zu unübersichtlichen und damit fehleranfälligen Modellen führen. Aus diesem Grund bietet Stateflow verschiedene Möglichkeiten zur *Hierarchisierung*, wodurch sich ein großes, komplexes Modell in **kleinere und leicht zu durchschauende Teile** strukturieren lässt.

Superstates

In den vorhergehenden Beispielen wurde schon eine Art der Hierarchisierung vorgestellt, die Superstates. Ein **Superstate ist ein Zustand, der wiederum einen Zustandsautomaten (also weitere durch Transitionen verknüpfte States) enthält**. Wird ein Superstate aktiv, dann wird die entsprechende Default transition des Zustandsautomaten innerhalb des Superstates ausgeführt. Während der Aktivität des Superstates ist einer der zugehörigen Unterzustände immer aktiv. **Superstates können sowohl exklusiv als auch parallel eingeordnet werden.** Die Einführung von Superstates in einen Chart führt zu weiteren Elementen, die im Folgenden erklärt werden (siehe auch Abbildung 49).

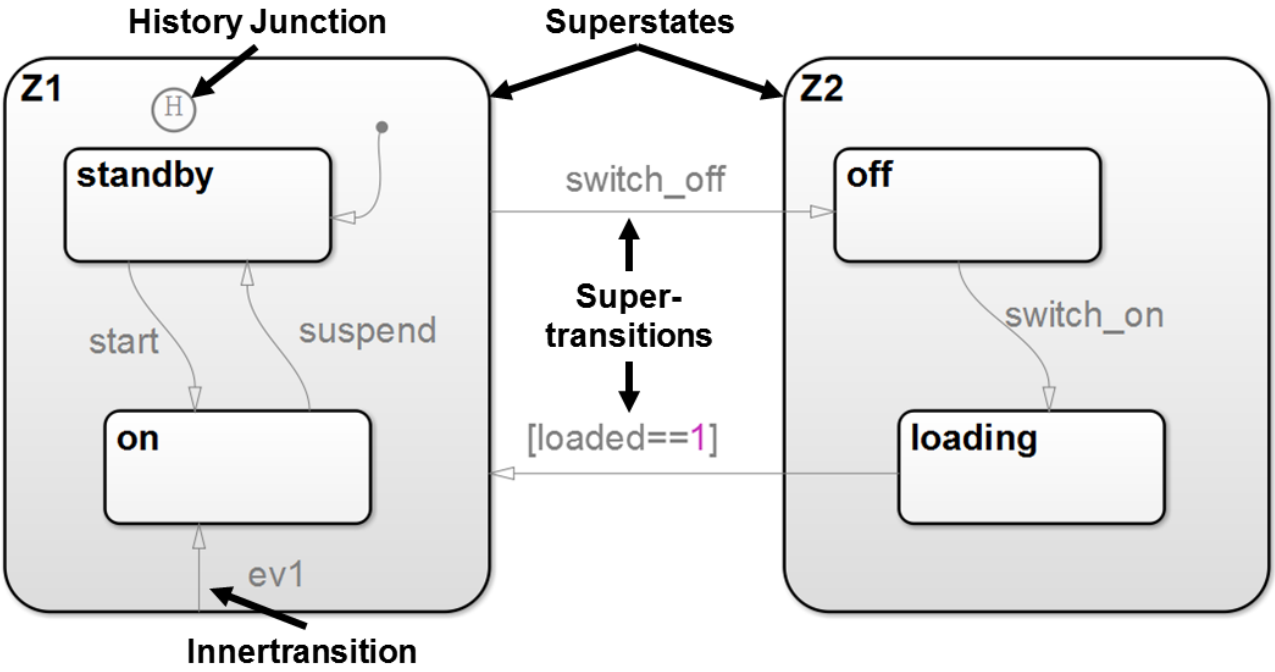


Abbildung 49: Superstates, Super- & Innertransitions und History Junction

Supertransitions: Supertransitions sind **hierarchieübergreifende Transitionen**. Das bedeutet, dass Quell- und Zielobjekt nicht zu der gleichen Hierarchieebene gehören. Dadurch kann man Superstates direkt von einem Unterzustand ohne Bedingung verlassen.

Innertransitions: Das Quellobjekt dieser Transitionen ist immer ein Superstate und das Zielobjekt ist einer der zugehörigen Unterzustände. Damit kann z. B. nach Auftritt des Events *ev1* der Zustand „on“ aktiviert werden, unabhängig davon, welcher State in Superstate Z1 vorher aktiv war..

History Junctions: Sie können durch Anklicken des entsprechenden Symbols in der Werkzeugleiste erzeugt werden und müssen **innerhalb der Grenzen eines Superstates** platziert werden. Befindet sich eine History Junction in einem Superstate, wird der **letzte aktive State bei Verlassen des Superstates** gespeichert. Bei erneutem Aktivieren des States **wird das Programm an der gespeicherten Stelle fortgesetzt**. Die History Junction ersetzt also die Default transition (diese wird jedoch immer noch für die *erste Ausführung* des States benötigt)

Groups: In einem Chart kann der Inhalt von Superstates gruppiert werden. Dies erfolgt durch den Rechtsklick auf einen markierten Zustand. Anschließend wird „Make Contents“ → “Grouped“ gewählt. Eine Group hat ausschließlich Auswirkungen auf das graphische Verhalten der enthaltenen Elemente. Während bei nicht-gruppierten Superstates die Elemente einzeln verschoben und ihre Größe verändert werden können, können Groups und ihre Größe nur als Ganzes verschoben bzw. geändert werden. Eine Group ist im Gegensatz zu einem nicht gruppieren Superstate grau hinterlegt.

Subcharts

In einem Chart können Superstates zu Subcharts umgewandelt werden. Dies erfolgt durch den Rechtsklick auf einen markierten Zustand. Anschließend wird „Make Contents“ → “Subcharted“ gewählt. Der Inhalt des States ist nun verborgen und kann durch einen Doppelklick aufgerufen werden. Dies hat **keinerlei Auswirkungen auf den Programmablauf** und dient lediglich der **Übersichtlichkeit**.

Ein Beispiel ist in Abbildung 50 zu sehen. Hierbei wurde der Zustand „Z2“ aus Abbildung 49 auf „Subcharted“ gestellt. Diese Vorgehensweise macht vor allem Sinn bei sehr großen Modellen, die ansonsten mehrere Bildschirme einnehmen würden.

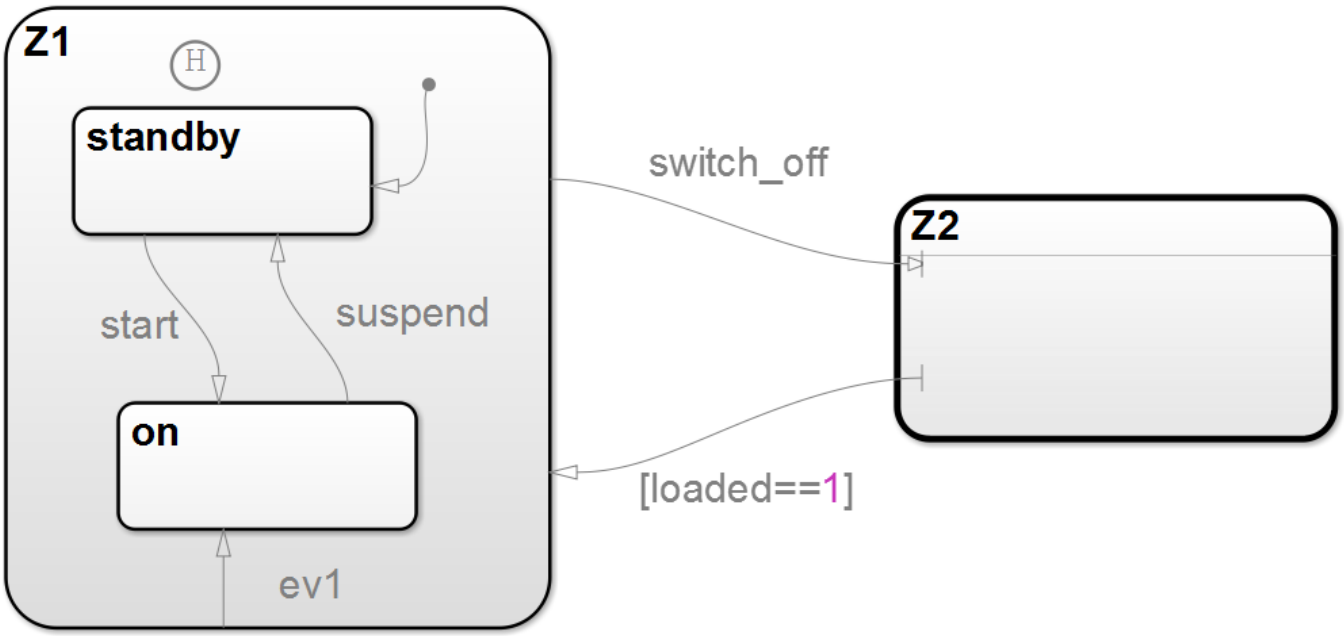


Abbildung 50: Subcharts

Boxes

Boxes verhalten sich analog zu Superstates. Sie stellen allerdings keinen eigenen Zustand dar, sondern erscheinen lediglich als frei platzierbares Rechteck ohne weitere Auswirkungen auf die Ausführung ihrer Inhalte. Auch sie lassen sich in einen Subchart verwandeln und dienen ausschließlich der Übersichtlichkeit.

Zur Erstellung einer Box wird auf das entsprechende Symbol in der Werkzeugleiste geklickt (siehe Abbildung 41 auf Seite 34) und die Box anschließend auf die richtige Größe gezogen.

5.1.3 Eigenschaften von Charts

Ein Chart wird beim Auftreten von Events aktiviert. Events können innerhalb des Charts oder auch extern erzeugt werden. Durch die Aktivierungsmethode (= Update method) wird festgelegt, wie Simulink die Charts in einem Modell steuert. Simulink bietet vier Arten von Update methods: Triggered, Inherited, Discrete und Continuous, die im Menü File → Model Properties → Chart properties entsprechend ausgewählt werden können (siehe Abbildung 51). Die Standardeinstellung ist „Inherited“.

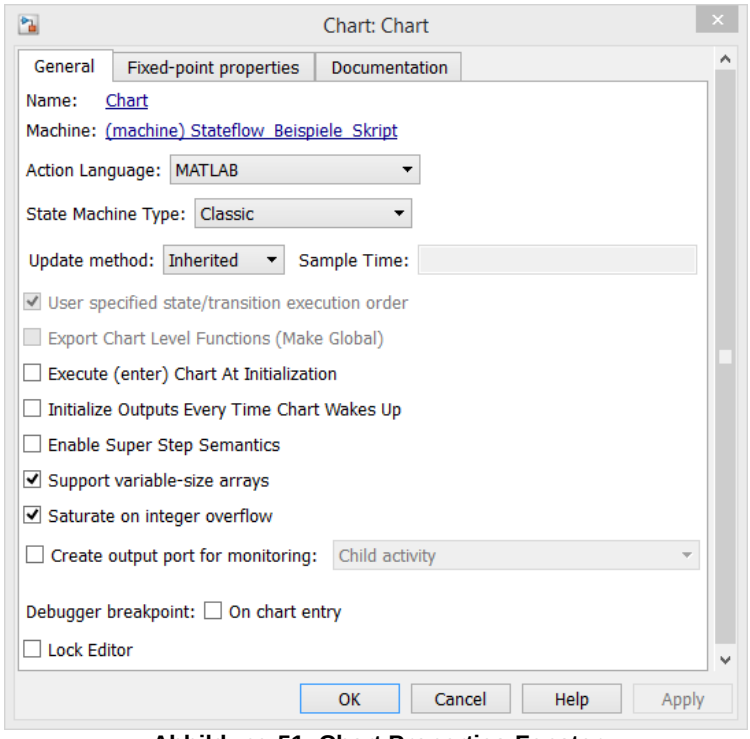


Abbildung 51: Chart Properties-Fenster

Inherited: Hier wird der Chart durch die Abtastrate seiner Eingangssignale aus Simulink gesteuert. Das heißt, jedes Mal wenn mindestens eines der Eingangssignale aktualisiert wird, werden die Bedingungen auf den Transitionen überprüft. Die Signale müssen dementsprechend im Data Dictionary als data (Input from Simulink) deklariert werden (Kapitel 5.1.4).

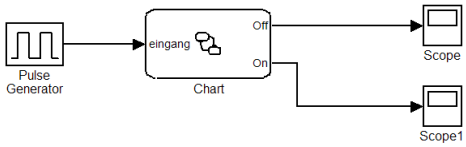


Abbildung 52: Chart mit inherited Update Method

Triggered: Ein triggered Chart wird nur dann aktiviert, wenn eine entsprechende Flanke des Trigger-Signals erkannt wird. Eine Flankenerkennung ist analog zur Erkennung eines Nulldurchgangs. Ändert sich ein Signal von Null zu einem positiven Wert, wird eine steigende Flanke erkannt. Ändert sich ein Signal von einem positiven Wert zu Null, wird eine fallende Flanke erkannt. Das Input-Trigger-Signal muss in der Data Dictionary als event deklariert werden (näheres zu Data Dictionary in Kapitel 5.1.4)

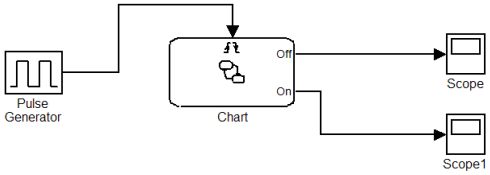


Abbildung 53: Triggered Chart

Discrete: Der Chart wird mit einer angegebenen und festen Rate (sample time) periodisch aktiviert. Diese Rate kann sich von der Integrationsschrittweite des Simulinkmodells und den Eingängen unterscheiden. Ein derartig aktiviertes Chart muss über keine Ein- oder Ausgänge zu Simulink verfügen.

Continuous: Der Chart wird mit der kontinuierlichen Simulationsschrittweite des Simulink-Solvers gesteuert. Die Aktivierung erfolgt auch zu Zwischenzeitpunkten, falls diese wegen der angegebenen Genauigkeit des Modells (refine factor) benötigt werden.

5.1.4 Data Dictionary und Model Explorer

Der *„Data Dictionary“* enthält **alle im Chart benutzten Variablen und Events** und ist **Teil des *„Model Explorers“***. Der Model-Explorer von Stateflow wird durch das Menü View → Model Explorer geöffnet bzw. durch Anklicken des entsprechenden Symbols in der Werkzeugleiste. Hier lassen sich sowohl die Hierarchisierung des Charts als auch die zu jeder Hierarchieebene gehörenden Variablen und Events betrachten.

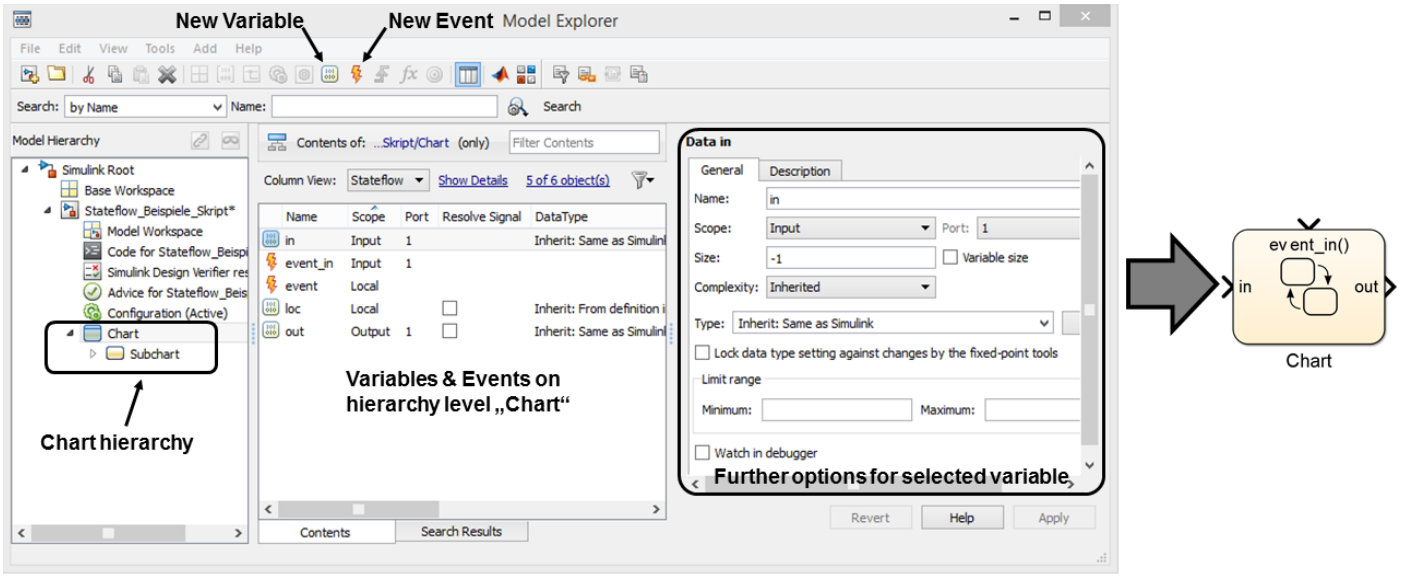


Abbildung 54: Stateflow Model-Explorer

Das **Vorgehen bei der Definition neuer Variablen und Events** ist in Abbildung 54 gezeigt und besteht aus folgenden Schritten:

1. Wähle im linken Fenster „Model Hierarchy“ die gewünschte Ebene, auf der die neue Variable definiert werden soll.
   Der Zugriff auf Variablen und Events erfolgt immer nur **innerhalb der eigenen und der unterlagerten Hierarchieebenen**. Sollen Variablen und Events global **im gesamten Chart** benutzt werden, müssen sie daher auf der **obersten Ebene** definiert werden.
2. Klicke in der oberen Leiste auf die Symbole für *„Add Data“* bzw. *„Add Event“*.
3. Benenne die Variable bzw. das Event und lege unter *„Scope“* fest, ob diese als Input, Output oder Lokal definiert sein soll.

Hinweis: Lokale *Events/Data* können in allen Hierarchieebenen definiert werden. **Ein- und Ausgänge nach Simulink müssen immer auf der obersten Ebene definiert sein**. Wird eine Variable oder ein Event als In-/Output definiert, entsteht am Chart ein entsprechender Port (Abbildung 54).

Wird eine Variable bzw. Event im mittleren Fenster **durch Anklicken markiert**, erscheinen im rechten Fenster weitere Optionen. Weitere für dieses Praktikum relevante Einstellungen (neben den bereits genannten) sind:

- **Size:** Legt fest, ob die **Variable ein Skalar oder ein Vektor ist** (und dessen Größe). Die Elemente des Vektors werden in eckigen Klammern angesprochen (z. B. input[2]).
- **Type:** Legt den Datentyp fest (also Double, Integer, Bool usw.). Bei der Standardeinstellung „Inherit“ wird der Datentyp aus Simulink übernommen.

Neben diesen Grundeinstellungen bietet der Model Explorer viele weitere fortgeschrittene Funktionen, die für dieses Praktikum nicht relevant sind und daher an dieser Stelle nicht behandelt werden. Bei Interesse sei auf die Dokumentation in der Hilfefunktion verwiesen.

5.1.5 Action Language

Der Begriff ***„Action Language“*** bezeichnet sämtlichen Syntax, der **zur Formulierung von Transitions- und Zustands-Labels** in Stateflow zur Verfügung steht. Dies können beispielsweise logische Ausdrücke, Aufrufe von Funktionen oder Formulierungen von Aktionen sein.

Einen kleinen Teil der Action Language haben wir bereits in Abbildung 44 auf Seite 36 kennengelernt. Hier wurde bei der Erfüllung einer definierten Bedingung der erste State verlassen, auf der Transition zwei Funktionen ausgeführt und schließlich bei Aufruf des zweiten States eine Variable gesetzt.

Einige (aber bei weitem nicht alle) Ausdrücke der Action Language werden im Folgenden aufgeführt.

Für dieses Praktikum relevante Ausdrücke sind grau hinterlegt. Selbstverständlich ist jedoch auch jegliche andere Syntax erlaubt. (Sofern sie das Richtige macht)

Schlüsselwörter und Elemente in Action Language

Schlüsselwort (Abkürzung)	Bedeutung
entry	Aktion bei Eintritt in den State (z. B. `en: variable_in = 1;`)
during	Fortlaufende Aktion während Simulation sich im State befindet.
exit	Aktion bei Verlassen des States (z. B. `ex: variable_out = 1;`)
on event_name	Aktion, wenn das Event „event_name“ auftritt.
chan**g**e(data_name)	Erzeugen eines lokalen Events, wenn sich der Wert von `data_name` ändert bzw. bei der Aktivierung/Deaktivierung des Zustands `state_name`
entry(state_name)	
exit(state_name)	
send(event_name,state_name)	Sendet das Event `event_name` an den Zustand `state_name`. Das Event kann auf der vom State wegführenden Transition abgefragt werden.

Tabelle 4: Schlüsselwörter der Action Language

Logische und mathematische Operatoren

Eignen sich vor allem für WENN-DANN-Abfragen mittels Transitionen. (z. B. WENN Temperatur > 100 °C, DANN springe in State „Wasser kocht“). **Verschachtelungen** sind mittels **runden Klammern ( )** möglich.

Operator	Beschreibung
a * b	Multiplikation
a / b	Division
a %% b	Restwert Division (Modulus), besser ist hier aber „ml.mod“
a + b	**Addition**
a - b	**Subtraktion**
a > b	Größer-Vergleich
a < b	Kleiner-Vergleich
a >= b	Größer-Gleich-Vergleich
a <= b	Kleiner-Gleich-Vergleich
a == b	Gleichheit
a ~= b	Ungleichheit

Tabelle 6: Logische und mathematische Operatoren

Aufruf von MATLAB-Funktionen und Zugriff auf den Workspace

Aus Stateflow lässt sich auch auf im MATLAB-Workspace definierte Variablen zugreifen. Des Weiteren können auch MATLAB-Funktionen ausgeführt werden. Der MATLAB-Workspace (für Variablen und Funktionen) muss mit einem vorangestelltem „ml.“ adressiert werden. Hierzu einige Beispiele:

Anweisung	Stateflow-Variablen	Workspace-Variablen
a=ml.x	a	x
a=ml(‘sin(%g)’,b) oder a=ml.sin(b)	a, b	
ml.y=ml.sin(ml.x)		x, y
a=(b+ml.x)*ml.y	a, b	x, y
v[1]=ml.vector[1]	V	vector
a=ml.mod(b, c)	a, b, c	

Tabelle 8: Zugriff auf den MATLAB-Workspace und Aufruf von MATLAB-Funktionen

Anmerkung: „ml.mod(b, c)“ entspricht „b %% c“, mit dem Unterschied, dass erstere Funktion mit Gleitpunktwerten rechnet und somit deutlich genauer arbeitet.

Wie immer an dieser Stelle auch der Hinweis auf die Hilfefunktion, in der viele weitere Befehle und Einsatzmöglichkeiten zu finden sind.

5.1.6 Variablen und Events

Folgende grundsätzliche Dinge müssen beachtet werden:

- **Sensorsignale** können nur **abgefragt** und **Aktorsignale** nur **gesetzt** werden.
- Eine Komponente muss den Zustand der anderen Komponenten berücksichtigen. So darf z. B. ein Auto erst dann losfahren, wenn die Ampel grün wird. Dies wird in Stateflow mittels Bedingungen auf Transitionen abgefragt.

Namenskonvention für den Aufgabenteil dieses Praktikums

Namen von **Sensorsignalen in der Form (i_...)**. **Aktorsignale in der Form (o_...)**. Dies dient der Übersichtlichkeit und ist Voraussetzung für eine Abnahme der Aufgaben.

Beispiel 1:

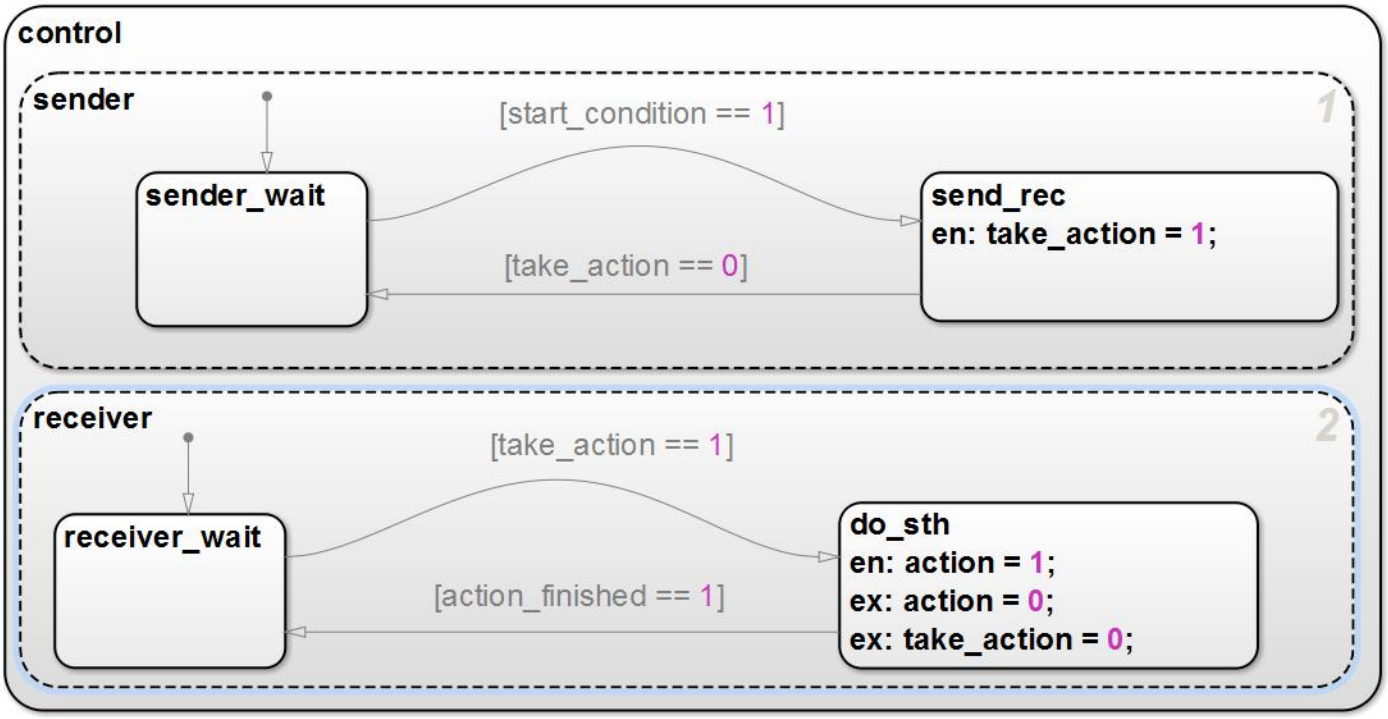


Abbildung 55: Beispiel zur Verwendung einer Kommunikationsvariablen

Der Empfänger (im State „empfaenger_befehl1“) **ist so lange im Wartezustand, bis er durch den Sender über die Variable *„take_action“* den Auftrag zur Ausführung einer Aktion bekommt**. In diesem Beispiel ist das die Aktivierung von *„action“*. Über die Variable *„action_finished“* wird dem Empfänger mitgeteilt, wann seine Aktion beendet ist. Als **Exit-Bedingung** müssen dann sowohl ***„action“*** als *auch* ***„take_action“*** zurückgesetzt werden. **Sobald *„take_action“* zurückgesetzt ist, weiß der Sender, dass der Empfänger seine Funktion ordnungsgemäß ausgeführt hat** und kehrt in den Ausgangszustand zurück.

Wird diese Methode verwendet benötigt man **eine Kommunikationsvariable pro Komponente**. Andernfalls ist die Gefahr zu groß, dass aus Versehen mehrere Komponenten gleichzeitig aktiviert werden.

Die Variable wird **immer im Sender gesetzt** und vom **Empfänger immer nach Beendigung der Aktion zurückgesetzt**.

Eine andere Möglichkeit der Kommunikation zwischen Sender und Empfänger sind Events. In diesem Fall entfällt die Notwendigkeit des Zurücksetzens der Kommunikationsvariablen (im letzten Beispiel war das „befehl1“).

Beispiel 2:

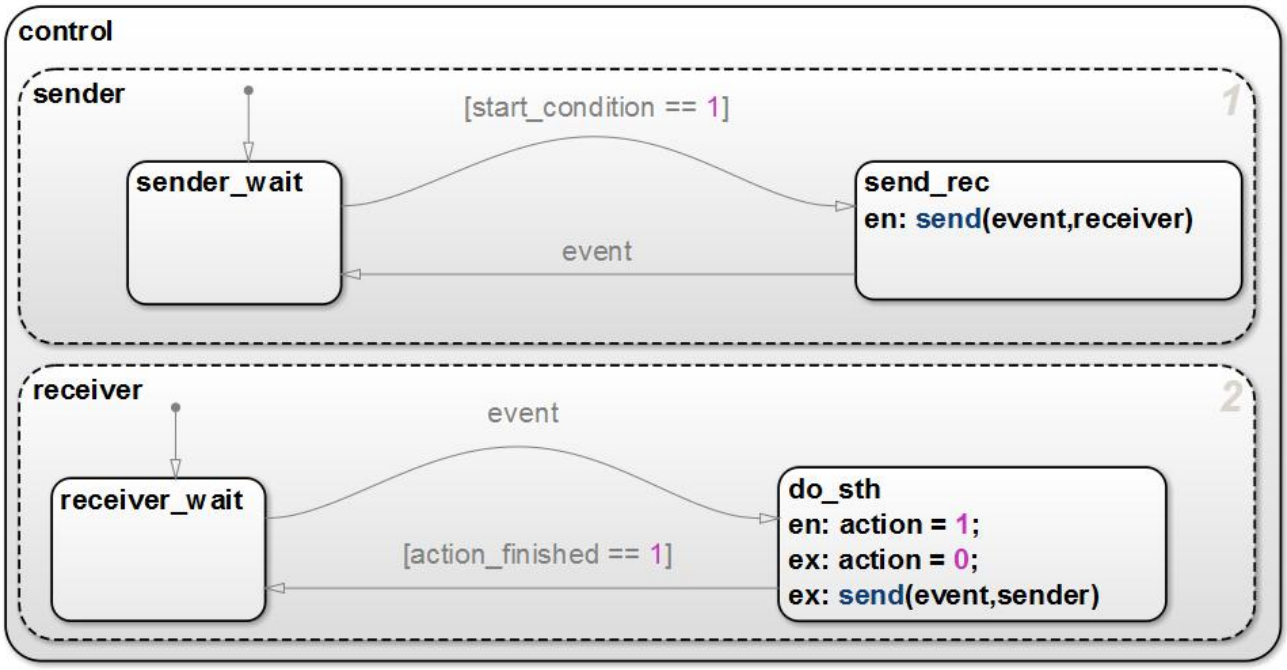


Abbildung 56: Beispiel zur Verwendung von Events

Hier **sendet der Sender das Event „event“ an den Empfänger**. Nachdem dieser seine Aktion beendet hat, teilt er dies dem Empfänger über das Event mit. Der Vorteil dieser Methode ist, dass das **Event gezielt an einen einzigen Empfänger** gesendet werden kann. Man muss sich also keine Gedanken über die versehentliche Aktivierung anderer Komponenten machen. **Zudem kann ein Event für mehrere Komponenten verwendet werden.**

Beispiel 3:

Folgende Abbildung zeigt das **Grundprinzip, wie mechanische Aktoren bedient werden**. Nachdem der Aktor (hier Schieber) in Bewegung gesetzt wurde, wird auf den Endschalter (Sensorsignal „extended“) gewartet. Ist dieser erreicht, wird der Aktor wieder eingefahren und **nach Beendigung des Vorgangs** ein Event an den Sender geschickt, damit dieser weiß, dass die Aktion erfolgreich beendet wurde.

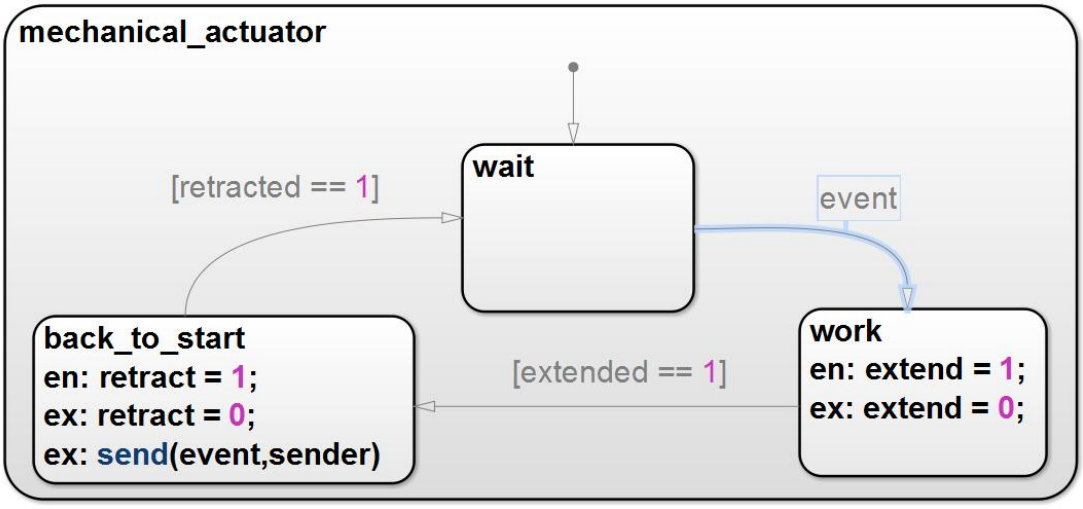


Abbildung 57: Steuerung eines Aktors

5.1.7 Beispiel zur allgemeinen Vorgehensweise

Eine zu simulierende Heizung hat **zwei Zustände** und soll über einen **Temperatursensor** gesteuert werden. Unterhalb von 16°C soll sich die Heizung einschalten, oberhalb von 20°C aus. Wie könnte die Temperatursteuerung für diese Heizung aussehen?

Dies ist nur ein Beispiel zum Verständnis, keine Praktikumsaufgabe!

1. Chart in das existierende Simulink-Modell der Heizung einfügen und mittels Doppelklick öffnen.
2. Überlegen, **welche Variablen benötigt werden und diese im Data Dictionary anlegen** (siehe Kapitel 5.1.4 auf Seite 42). In diesem Beispiel ist das die *Temperatur*, die über einen *externen Sensor* geliefert wird, sowie das eigentliche *Steuersignal für die Heizung (an-aus)*. Folglich ist hier eine Variable, die als Eingang deklariert werden muss, nötig (Temperatur), sowie ein Ausgang (Steuersignal).
3. Überlegen, **welche Zustände das System einnehmen kann** und diese Zustände (States) einfügen. Anschließend eindeutige und sinnvolle Namen für die Zustände vergeben. Hier gibt es nur zwei mögliche Zustände: „Heizung an“ und „Heizung aus“.
4. Überlegen, unter **welchen Bedingungen von einem Zustand in einen anderen Zustand gewechselt wird**. In diesem Beispiel wird zu „Heizung_an“ gewechselt, wenn die Temperatur kleiner als 16°C ist und zum Zustand „Heizung_aus“, wenn die Temperatur oberhalb von 20°C ist.
5. Transitionen mit den in Schritt 4 überlegten Bedingungen anlegen. Ebenso überlegen, **in welchem Zustand sich die Steuerung am Anfang befinden soll** und entsprechend die **Default transition anlegen**. In diesem Beispiel ist es egal, in welchem Zustand die Simulation beginnt, da es keinen logischen Startzustand gibt.
6. **Anweisungen in den Zuständen anlegen** (die erste Zeile eines Zustandes ist dessen Name, alle folgenden Zeilen sind Anweisungszeilen (Zuweisungen)). Im Zustand „Heizung_an“ wird die Heizung angeschaltet (en: heizung=1), im Zustand „Heizung_aus“ wird die Heizung abgeschaltet (en: Heizung=0).
7. **Testen!!!** (z. B. durch Betrachtung der Ein- und Ausgangssignale in einem Scope)

5.1.8 Programmbeispiel

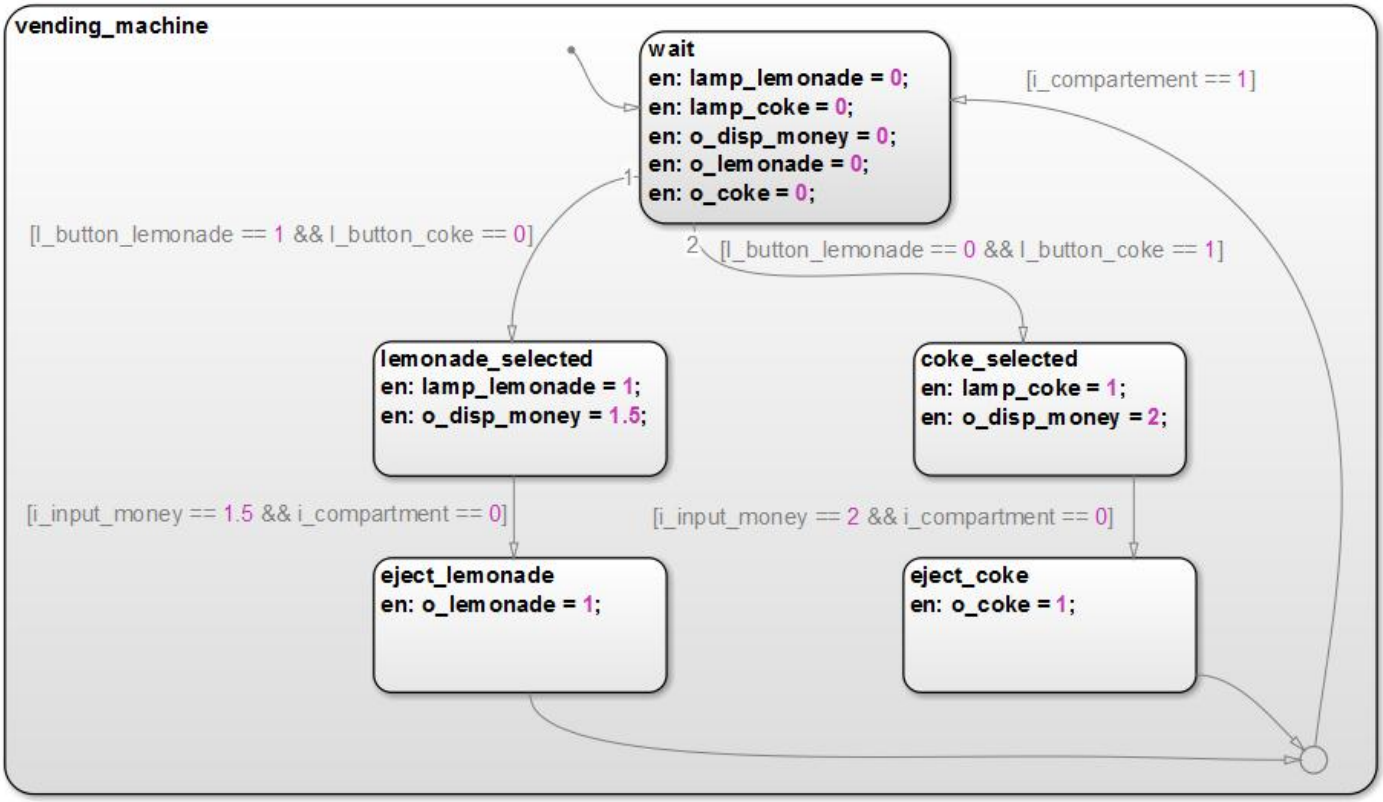


Abbildung 58: Programmbeispiel Getränkeautomat

Abbildung 58 zeigt die Programmierung der Steuerung eines (sehr rudimentären) Getränkeautomaten. Die Steuerung läuft in folgenden Schritten ab.

Dies ist nur ein Beispiel zum Verständnis, keine Praktikumsaufgabe!

1. Im ersten Schritt wird der Ausgangszustand hergestellt: Alle Ausgänge werden auf einen definierten Zustand gesetzt.
2. Anschließend wartet der Automat auf eine Tasteneingabe für Cola oder Limo. Hierzu werden laufend die Bedingungen auf den Transitionen geprüft.
3. Entsprechend der Eingabe des Benutzers (simuliert durch die Sensorsignale der Wahltasten) wird eine Lampe angeschaltet und der passende Geldbetrag auf dem Display angezeigt.
4. Wenn der Kunde den Betrag bezahlt hat und sich im Auswurf keine Flasche befindet, wird das passende Fach geöffnet.
5. Ist die Flasche ausgegeben (erkennbar durch den Flaschensensor im Ausgabefach), kehrt der Automat in seinen Ausgangszustand zurück und der Programmzyklus beginnt von neuem.

5.1.9 Checkliste: Häufige Fehler

1. **Keine bzw. mehrere gültige Default transitions** innerhalb einer Hierarchieebene.
2. **Transition kreuzt Grenze des Superstates**. Hierdurch gilt der State als Verlassen und es kommt zu unvorhersehbarem Verhalten.

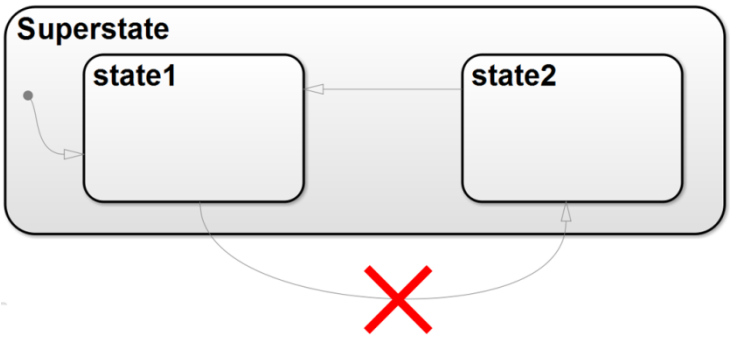


Abbildung 59: Transition verlässt State

3. **Sonderzeichen, Umlaute oder Leerzeichen** in jeglichen Benennungen (States, Variablen, Events).
4. **Sackgassen**, die zu einem Programmstillstand führen. In untenstehendem Beispiel kann das Wasser nie wieder abgeschaltet werden. Programme sollten daher immer als Kreisprozess angeordnet sein.

Abbildung 60: Sackgasse

5. **Mehrere, gleichzeitig erfüllte Transitionen** ausgehend von einem State. Stateflow kann sich nicht entscheiden und stoppt das Programm.

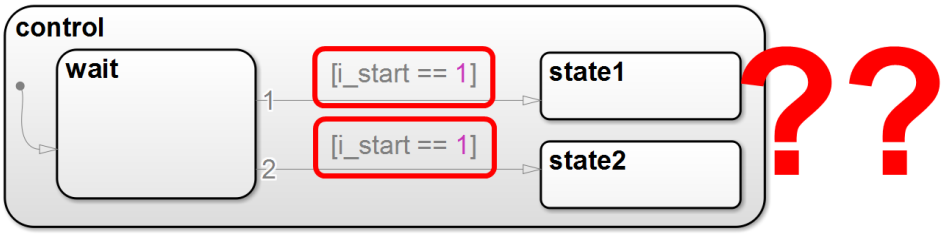


Abbildung 61: Zwei gleichzeitig gültige Transitionen

6. **Vergleich von Double Werten mit „==“**. Auf Grund numerischer Ungenauigkeiten wird diese Bedingung nie erfüllt. Versuchen Sie das Problem so zu formulieren, dass „>=“ oder „<=“ benutzt werden kann.
7. **Variablen werden nicht zurückgesetzt**. Lassen Sie Variablen nicht länger als notwendig gesetzt.
8. **Mehrere Zustände mit gleichen Namen**. Jeder State muss einen im Chart einmaligen Namen haben.
9. **`[1 < x < 10]` funktioniert nicht**. Verwenden Sie `[x > 1 && x < 10]`.
10. **Rechnen mit Integer-Werten**. Bei Verwendung von Integer-Werten ergibt sich: *`8/5 = 1`* ! Für ein korrektes Ergebnis müssen 8 und 5 als Double definiert werden.