Automatisierte Extraktion der Zustandsraummodelle von Bondgraphen in Matlab / Simulink

Entwicklung einer Toolbox zur Emulation von Bondgraphen in Matlab / Simulink mit automatisierter Extraktion der zugehörigen Zustandsraummodelle

Martin Ehlert, B. Eng

26. Juni 2015

Abriss

Ziel der vorliegenden Arbeit ist die Emulation von Bondgraphen in Matlab / Simulink. Hierfür wurde eine Toolbox entwickelt, durch welche es möglich ist Bondgraphen in Simulink zu modellieren und die darauf basierende Zustandsraumdarstellung des jeweiligen Graphen automatisiert zu ermitteln.
Die vorliegende Arbeit beschreibt die Grapherstellung in Simulink mit Hilfe der entwickelten Toolbox und die Realisierung der in Matlab zugrundeliegenden mehrstufigen Lösungsalgorithmen zur analytischen Ermittlung der Zustandsraummodelle sowie die Aufbereitung des Graphen und der internen Elementgleichungen. Des Weiteren werden zusätzlich die implementierten Mechanismen zur Korrektur uneindeutiger Lösungen im Fall von Kausalitätskonflikten und zur Verkettung gleichartiger Knoten erläutert.

Schlagwörter:

- Bondgraphen
- Matlab / Simulink
- Zustandsraumdarstellung
- domänenübergreifende Simulation

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I.

Hauptteil

1. Einführung

1.1. Einleitung

Grundlegendes Ziel des Projektes ist die Erstellung einer Toolbox für Matlab / Simulink, welche die direkte Umsetzung von Bondgraphen in Simulink ermöglicht. Basierend auf dem jeweiligen vom Benutzer in Simulink modellierten Bondgraph wird anschließend mit Hilfe von Matlab das dem Graphen zugehörige Zustandsraummodell ermittelt. Die resultierende Toolbox soll nach ihrer Fertigstellung als nicht kommerzielle und quelloffene Software veröffentlicht werden.

Die vorliegende Arbeit ist in mehrere Segmente unterteilt, wobei im ersten Kapitel auf die allgemeinen Projektziele und den von diesen abgeleiteten Anforderungen an die resultierende Toolbox in Bezug auf den aktuellen Stand der Technik sowie ähnliche bereits existierende Lösungen eingegangen wird.
Das zweite Kapitel liefert einen einführenden Überblick in die Methodik der Bondgraphen und der zugrundeliegenden Elemente. Des Weiteren wird in diesem Teil beispielhaft die Analyse eines Systems mit Bondgraphen sowie ein Vergleich der regulären Modellierungsmethode von System in Simulink gegenüber der Bondgraphenmethode mit Hilfe der Toolbox durchgeführt.
Den Hauptteil der Arbeit bildet das dritte Kapitel, in welchem die internen Mechanismen der Toolbox bei der Modellierung und Verarbeitung der Bondgraphen beschrieben sind. Ansätze für eine Weiterführung des Projektes, Probleme in der aktuellen Version der Toolbox sowie ein Fazit befinden sich im abschließenden vierten Kapitel.
Die vollständigen Quellcodes der aktuellen Version der entwickelten Toolbox sind dieser Arbeit als Anhang in den Abschnitten A bis H angefügt.

Bei Bondgraphen handelt es sich um eine Methode zur Beschreibung von Systemen beliebiger physikalischer Domänen, wobei der Austausch von Leistungen zwischen den einzelnen Elementen die Grundlage der mathematischen Beschreibung darstellt.
Die Interaktion der verallgemeinerten Systembestandteile durch Leistungsübertragung wird mit Hilfe von zwei generalisierten Variablentypen, den Bondvariablen *Effort* und

Flow, beschrieben. Aufgrund dieses allgemeinen Konzeptes sind Bondgraphen in der Lage Systeme über die Grenzen einzelner physikalischer Domänen hinweg und in beliebiger Konstellation dieser Domänen zu beschreiben.
Die Übertragung der Leistung erfolgt von leistungserzeugenden Quellen in Richtung energieverbrauchender oder -speichernder Senkenelemente über serielle und / oder parallele Knotenpunkte sowie Umformelemente, woraus eine relativ schlichten Struktur der Modelle resultiert.

Aus dem modellierten Bondgraph wird die Zustandsraumdarstellung des repräsentierten Systems abgeleitet, was autonom durch ein hierfür entwickeltes Verfahren erfolgt. Die Darstellung im Zustandsraum erfolgt durch die Ermittlung der zugehörigen Zustands- und Ausgangsgleichungen des Graphen. Abschließend werden diese Gleichungen in einen matlabeigenen Zustandsraummodelltyp umgewandelt, wobei sowohl die ursprünglichen Gleichungen als auch das Modell gespeichert werden und anschließend vom Benutzer beliebig weiter verwendet werden können.

Eine weitere Motivation für die Entwicklung dieser Toolbox und zusätzliches Ziel in der Fortführung des Projektes besteht in Ableitung von elementbasierten und domänenspezifischen Toolboxen auf Basis der ursprünglichen Bondgraphen-Toolbox. Diese sollen trotz ihrer jeweiligen Spezialisierung die modellierten Systeme intern als Bondgraphen beschreiben und analog der zugehörige Zustandsraummodell erstellen.

Die resultierende Bondgraphen-Toolbox trägt den Titel “BondLib” und wurde soweit automatisiert, dass vom Benutzer lediglich die Parameter der Quellelemente und der restlichen Element sowie deren Verbindungen untereinander spezifiziert werden müssen. Die vollständige Beschreibung des Bondgraphen durch Ermittlung seines Zustandsraummodells wird von der Toolbox vorgenommen.

Im Verlauf des Projektes wurde ein Lösungsalgorithmus entwickelt, welcher in der Lage ist, Lösungen für Variablen aus einem zusammenhängenden Gleichungssatz zu ermitteln. Basierend auf gegebenen Teillösungen und / oder bereits bekannten Variablen werden aus dem Gleichungssatz die gewünschten Lösungen durch symbolisches Umformen und Substituieren gewonnen.
Der zugrundeliegende Ermittlungsmechanismus könnte mit geringen Modifikationen auch zur Auflösung von Gleichungssätzen in Bezug auf vorgegebene Einzellösungen genutzt werden, da in der aktuellen Toolboxversion auch Systeme ohne Zustandsvariablen bzw. ohne energiespeichernde Elemente dargestellt und berechnet werden können.

1.2. Stand der Technik

Es existieren bereits kommerzielle Lösungen zur Modellierung dynamischer Systeme mit Hilfe von Bondgraphen. Diese Lösungen haben bereits Marktreife erreicht und stehen als eigenständige Programme für die wichtigsten Betriebssysteme zur Verfügung.
Das Ziel des Projektes ist es, die Verwendung von Bondgraphen in der weitverbreiteten Numeriksoftware Matlab / Simulink mit Hilfe einer quelloffenen Toolbox zu ermöglichen. Durch die freie Zugänglichkeit des Quellcodes soll die Weiterentwicklung der Toolbox sowie die Ableitung domänenspezifischer Toolboxen erleichtert werden.

Die Verwendung von Bondgraphen in Simulink ist in der *BG*-Toolbox[3] implementiert, welche auf den Internetseiten von Mathworks unter einer reduzierten BSD-Lizenz veröffentlicht ist. Diese Toolbox wurde bis zu ihrer aktuellen Version 2.1 an der Technischen Universität Dresden von Gert-Helge Geitner entwickelt.
Die Modellierung der Graphen erfolgt mit dieser Toolbox in Simulink, wobei die Kausalität der Bondelemente beziehungsweise die Kausalität der einzelnen Bonds vom Benutzer festgelegt werden muss und kein Korrekturmechanismus für eventuelle hieraus resultierende Kausalitätsfehler existiert.
Es erfolgt keine Extraktion des Zustandsraummodells oder der symbolischen Systemgleichungen zur Weiterverwendung unabhängig von der *BG*-Toolbox.

Eine weitere Implementierung wird in einer Veröffentlichung[2] des Indian Institute of Science beschrieben, stand jedoch nicht als Download oder anderweitig zur Verfügung. Gemäß dieser Veröffentlichung aus dem Jahr 2005 können mit Hilfe der beschriebenen Toolbox Bondgraphen in Simulink modelliert und simuliert werden.
Die Umsetzung erfolgt hierbei durch die Verwendung eines zusätzlichen *Data Space* im ursprünglichen Workspace von Matlab. Bei jedem einzelnen Zeitschritt der Simulation in Simulink werden die Werte aller im modellierten Graph vorkommenden Variablen in diesem Data Space abgelegt. Die Werte der Variablen werden in Matlab entsprechend der Gesetzmäßigkeiten von Bondgraphen verrechnet und die Ergebnisse vor dem nächsten Simulationsschritt in das Simulinkmodell zurückgeschickt, wo die berechneten Ergebnisse die Ausgangswerte für eben diesen nächsten Schritt der Simulation bilden. Der hieraus resultierende hohe Rechenaufwand wird von den Autoren als nachteilig angemerkt.
Zusätzlich zur Simulation ist die Möglichkeit der Extraktion der symbolischen Systemgleichungen der jeweiligen Bondgraphen in der Veröffentlichung aufgeführt.

1.3. Anforderungen

Um eine Lösung für die Verwendung von Bondgraphen in Matlab/Simulink ergänzend zu den bereits existierenden Implementierungen aus Abschnitt 1.2 zu erhalten, wurden die folgenden Anforderungen an die Realisierung der Toolbox gestellt:

- Die Modellierung der Bondgraphen soll graphisch in Simulink unter Verwendung von Simulinkblöcken zur Repräsentation der Bondelemente erfolgen.
- Die Ermittlung des Zustandsraummodells basierend auf dem vom Benutzer modellierten Bondgraph soll außerhalb des Simulinkmodells in Matlab geschehen, um spätere Modifikationen zu erleichtern.
- Die Verwendungen und Bedienung der Toolbox darf nur geringe Anforderungen an den Benutzer bezüglich der Modellierung von Bondgraphen und Aufstellung von Zustandsraummodellen stellen. Gleiches gilt für die repräsentierten physikalischen Systeme, sodass es beispielsweise im Fall von elektrischen Schaltungen für den Benutzer genügt, die Parameter und Verschaltung der Bauelemente zu kennen, um dass Zustandsraummodell der Schaltung mit Hilfe der Toolbox zu ermitteln.
- Durch den Benutzer inkorrekt festgelegte Bondkausalitäten im Graph sollen durch die Toolbox automatisch korrigiert werden, um trotzdem eine Ermittlung des Zustandsraummodells zu ermöglichen.
- Die Grundelemente von Bondgraphen sollen in der Toolbox implementiert sein.
- Das Zustandsraummodell der Graphen soll als matlabeigenen *ss*-Modelltyp sowie dessen zugehörige Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$ und $\underline{D}$ ausgegeben werden, um eine beliebige Weiterverwendung zu gewährleisten und die Implementierung hierauf basierenden Parallelsimulation des Graphen zu ermöglichen.
- Die Toolbox soll unter Verwendung der Grundkonfiguration von Matlab / Simulink vollständig verwendbar sein, sodass keine Installation von zusätzlicher Software oder andere Anpassungen von Matlab / Simulink, bis auf das Einbinden der Toolbox, notwendig sind.
- Die resultierende Toolbox soll nach ihrer Fertigstellung unter passender Lizenz als quelloffene Software veröffentlicht werden.

2. Bondgraphen

2.1. Grundlagen

Bondgraphen sind eine Methode zur domänenunabhängigen Beschreibung von Systemen. Das System wird hierbei durch ein verallgemeinertes Schema repräsentiert, aus welchem die Zustandsraumdarstellung des Systems ermittelt wird.
Diese Darstellung beschreibt das Verhalten des Systems durch Differentialgleichungen erster Ordnung. Grundlage dieser Systembeschreibung ist, dass bei jeder Interaktion zwischen Systemen oder Teilsystemen Arbeit verrichtet wird und somit eine Übertragung von Leistung stattfindet.
Hierdurch bieten Bondgraphen die Möglichkeit domänenübergreifende Systeme und Prozesse mit einer einheitlichen Methode zu erstellen und zu verarbeiten.

Die einzelnen Elemente eines Bondgraphen sind durch die namensgebenden Bonds untereinander verbunden, durch welcher der Leistungsfluss zwischen den jeweiligen Elementen symbolisiert wird. Jeder Bond verfügt über die beiden verallgemeinerten Variablen Effort $e(t)$ und Flow $f(t)$, die entsprechend der vorliegenden physikalischen Domäne verschiedene Größen repräsentieren.

Der englische Begriff "Effort" kann mit "Bestreben" übersetzt werden und stellt in den jeweiligen Domänen die Ursache für die Leistungsübertragung dar, wie es zum Beispiel beim Ausgleich einer Potentialdifferenz der Fall ist. Dementsprechend kann der Flow als die Auswirkung dieser Ursache interpretiert werden. "Flow" wird üblicherweise mit "Fluss" übersetzt, wobei jedoch die alternative Übersetzung mit "Bewegung" den Wirkungscharakter dieser Größe passender beschreibt.
Die Entsprechungen der Bondvariablen Effort und Flow sind in Tabelle 2.1 für die vier wichtigsten physikalischen Domänen aufgelistet.

Die zum jeweiligen Zeitpunkt zwischen zwei Elementen übertragene Leistung berechnet sich als das Produkt von Effort und Flow gemäß den Entsprechungen dieser beiden Bondvariablen in den jeweiligen physikalischen Domänen.

Domäne	Effort $e(t)$ [Einheit]	Flow $f(t)$ [Einheit]
Translation	Kraft $F(t)$ $[N]$	Geschwindigkeit $v(t)$ $[\frac{m}{s}]$
Rotation	Moment $M(t)$ $[Nm]$	Winkelgeschwindigkeit $\omega(t)$ $[\frac{rad}{s}]$
Fluidtechnik	Druck $p(t)$ $[\frac{N}{m^2}]$	Volumenstrom $\dot{V}(t)$ $[\frac{m^3}{s}]$
Elektrotechnik	Spannung $U(t)$ $[V]$	Strom $I(t)$ $[A]$

Tabelle 2.1.: Entsprechungen von Effort und Flow[1, S. 14; Table 2.1]

Als zeitliche Ableitung der Energie beschreibt die Leistung des Weiteren die momentane Änderung der im System enthaltenen Energie:

$$P(t) = e(t)f(t) \tag{2.1}$$
$$= \dot{E}(t) \tag{2.2}$$

Die im Zielelement oder -system eines Bonds enthaltene Energie ergibt sich durch Integration der übertragenen Leistung $P(t)$ gemäß der folgenden Gleichung 2.3. Bei der Variable E_0 handelt es sich in dieser Gleichung um die zum Beginn der Betrachtung bereits im System enthaltene Initialenergie.

$$E(t) = \int^t P(t)dt + E_0 = \int_{t_0}^t e(t)f(t)dt + E_0 \tag{2.3}$$

Zusätzlich zu dem regulären Effort $e(t)$ und dem Flow $f(t)$ sind in Bondgraphen der Impuls $p(t)$ sowie die Verschiebung q(t) von essentieller Bedeutung, da diese die Zustandsvariablen des modellierten Systems beziehungsweise des Bondgraphen darstellen. Die domänenspezifischen Entsprechungen dieser beiden verallgemeinerten Zustandsvariablen können der Übersicht 2.2 entnommen werden.

Domäne	Impuls $p(t)$ [Einheit]	Verschiebung $q(t)$ [Einheit]
Translation	Impuls $p(t)$ $[Ns]$	Verschiebung x $[m]$
Rotation	Drehimpuls $L(t)$ $[Nms]$	Winkel φ $[rad]$
Fluidtechnik	Druckimpuls $p_p(t)$ $[\frac{Ns}{m^2}]$	Volumen V $[m^3]$
Elektrotechnik	magnetischer Fluss Φ $[Vs]$	elektrischer Fluss ψ $[As]$

Tabelle 2.2.: Domänenspezifische Impuls- und Verschiebungsvariablen

Der generalisierte Impuls $p(t)$ wird durch Integration des Effort $e(t)$ berechnet, wobei p_0 in der zugehörigen Gleichung 2.4 der Initialimpuls ist.

$$p(t) = \int_{t_0}^{t} e(t)dt + p_0 \tag{2.4}$$

Die in Bondgraphen verwendete allgemeine Verschiebung $q(t)$ wird aus dem Flow $f(t)$ durch Integration errechnet, woraus die folgende Gleichung 2.5 analog 2.4 resultiert:

$$q(t) = \int_{t_0}^{t} f(t)dt + q_0 \tag{2.5}$$

Effort und Flow lassen sich durch Differenzierung der Gleichungen 2.4 und 2.5 als zeitliche Ableitung des generalisierten Impuls $p(t)$ beziehungsweise der generalisierten Verschiebung $q(t)$ bestimmen. Die Anfangswerte p_0 für den Impuls $p(t)$ und q_0 für die Verschiebung $q(t)$ entfallen durch die Ableitung.

$$\dot{p} = \frac{dp(t)}{dt} = e(t) \tag{2.6}$$

$$\dot{q} = \frac{dq(t)}{dt} = f(t) \tag{2.7}$$

Der Impuls $p(t)$ und die Verschiebung $q(t)$ werden auch als Zustandsvariablen oder Energievariablen bezeichnet, da von ihnen die im jeweiligen Bondelement enthaltene Energie abhängig gemacht werden kann. Hierfür müssen die Gleichungen 2.6 und 2.7 nach dq beziehungsweise dp umgeformt werden:

$$e(t)dt = dp(t) \tag{2.8}$$

$$f(t)dt = dq(t) \tag{2.9}$$

Im Anschluss an diese Umformung können die Gleichungen 2.8 und 2.9 in die ursprüngliche Energiegleichung 2.3 eingesetzt werden. Hierdurch entsteht eine Abhängigkeit der Energie $E(t)$ von der jeweiligen Zustandsvariable.
Es ist zu beachten, dass für die Anfangsenergie E_0 der Wert null angenommen wird[1, S. 16] und somit in den resultiren den Gleichungen 2.10 und 2.11 entfällt.

$$E(t) = \int^t f(t)dp(t) \tag{2.10}$$

$$= \int^t e(t)dq(t) \tag{2.11}$$

Die Energie $E(t)$ kann demzufolge als Funktion des Impulses $p(t)$ beziehungsweise der Verschiebung $q(t)$ beschrieben werden, wobei die jeweilige Zustandsvariable direkt zur Festlegung der Integrationsgrenzen[1, S.16] verwendet wird:

$$E(p) = \int^p f(p)dp \tag{2.12}$$

$$E(q) = \int^q e(q)dq \tag{2.13}$$

Des Weiteren können die zeitlichen Zustandsableitungen $\dot{q}(t)$ und $\dot{p}(t)$ durch Substitution der Gleichungen 2.6 und 2.7 für die regulären Bondvariablen $e(t)$ beziehungsweise $f(t)$ in Gleichung 2.1 zur Beschreibung der Leistung verwendet werden:

$$P(t) = \dot{p}(t)f(t) \tag{2.14}$$

$$= e(t)\dot{q}(t) \tag{2.15}$$

Mit Hilfe der verallgemeinerten Bondvariablen Effort $e(t)$ und Flow $f(t)$ in Kombination mit den ebenfalls verallgemeinerten Zustandsvariablen Impuls $p(t)$ und Verschiebung $q(t)$ können beliebige Systeme im Zustandsraum dargestellt werden.
Die auf Integration und Differentiation basierenden Zusammenhänge zwischen den regulären Bondvariablen und den Zustandsvariablen werden üblicherweise mit Hilfe des so genannten Zustandstetraeder [1, S. 16; Abb. 2.2] schematisch zusammengefasst.

In Abbildung 2.1 ist das modifizierte *Zustandstetraeder* dargestellt, welches die Zusammenhänge zwischen den einzelnen Größen mittels Integration und Diffentiation beschreibt.
Es zu beachten, dass das graphische Schema um die elementbasierten Zusammenhänge zwischen Effort und Verschiebung, Flow und Impuls sowie Flow und Effort gegenüber

der Grundversion erweitert wurde. Diese Verknüpfungen sind durch die in Abschnitt 2.4.1 beschriebenen Bondelemente Resistor R, Capacitor C und Inertia I gegeben.

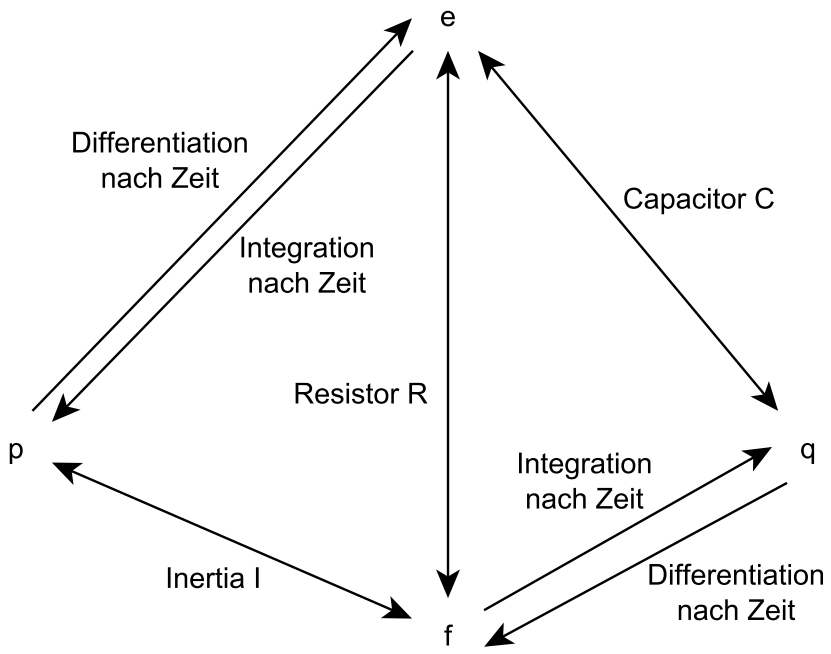


Abbildung 2.1.: Modifiziertes Zustandstetraeder

2.2. Bonds

Die namensgebenden Verbindungen zwischen einzelnen Elementen werden als Bonds bezeichnet und charakterisieren die Leistungsübertragung zwischen diesen Elementen in Bondgraphen.
Es wird zwischen zwei Bondtypen unterschieden, dem *normalen* Bond[1, S. 20] zur Leistungsübertragung und dem *aktiven* Bond [1, S. 22] zu Messzwecken (*engl.: active bond*) mit vernachlässigbarer Leistungsübertragung. Aktive Bonds werden durch einen normalen Pfeil (engl.: *full arrow*) zwischen Elementen und normale Bonds durch einen halben Pfeil (engl.: *half arrow*) gekennzeichnet, wobei die Pfeilrichtung der Richtung der Leistungsübertragung entspricht.
Die Bondvariablen Effort $e(t)$ und Flow $e(t)$ beziehungsweise deren physikalische Bezeichner werden am jeweiligen Bond vermerkt, wobei mit den Zustandsableitungen $\dot{p}(t)$ und $\dot{q}(t)$ in gleicher Weise verfahren wird. Üblicherweise werden im Fall von Bonds mit horizontaler Ausrichtung die Effortvariable oberhalb und die Flowvariable unterhalb des

Pfeils geschrieben, wohingegen sich bei vertikalen Bonds der Effortbezeichner links und der Flowbezeichner rechts vom Bond befinden. [1, S. 20]

In Abbildung 2.2 sind beide Bondtypen an einem Widerstandselement beispielhaft dargestellt. Der Leistungsfluss ist in beiden Fällen auf das Element gerichtet.

(a) Normaler Bond

(b) Aktiver Bond

Abbildung 2.2.: Beispielbonds

2.3. Kausalität

Jeder Bond verfügt zusätzlich über die Eigenschaft der Kausalität. Diese gibt die führende der beiden Bondvariablen an, von welcher die jeweils andere Bondvariable des jeweiligen Bonds abhängt.
Die Kausalität wird durch den so genannten Kausalstrich (engl.: *causal stroke*)[1, S. 25] am Anfang oder Ende eines Bondpfeils markiert. Befindet sich der Kausalstrich auf der Seite des Zielelements, hängt die Flowvariable des Bonds vom eingehenden Effort ab. Im umgekehrten Fall ist die Effortvariable abhängig von der führenden Flowvariable und der Kausalstrich befindet sich am Ausgangspunkt des Bonds. Es ist zu beachten, dass die Kausalität nicht nur für die regulären Bondvariablen $e(t)$ und $f(t)$, sondern auch für die Zustandsableitungen $\dot{p}(t)$ und $\dot{q}(t)$ gilt.

Der bereits bekannte Bond aus Abbildung 2.2a des vorherigen Abschnitts 2.2 ist in Abbildung 2.3 in beiden Kausalitätsvarianten dargestellt.

(a) Effortkausalität
$f(t) = f(e(t))$

(b) Flowkausalität
$e(t) = e(f(t))$

Abbildung 2.3.: Beispielbonds mit Kausalitätsangabe

Die abhängige Bondvariable wird von der Gleichung des jeweiligen Bond beziehungsweise des angeschlossenen Elements in expliziter Form beschrieben, wobei die Gleichung von der jeweils führenden Bondvariable abhängt.
Die beiden möglichen Kausalitätsvarianten werden in der vorliegenden Arbeit als *Effortkausalität f(e)* und *Flowkausalität e(f)* bezeichnet. Die jeweils führende Bondvariable ist hierbei bezeichnend, wodurch im ersten Fall der Flow vom Effort abhängt. Liegt andernfalls Flowkausalität vor, hängt der Effort des Bonds von dessen Flowvariable ab.

Würde ein Bondelement mit angeschlossenem Bond als mathematische oder programmiertechnische Funktion interpretiert werden, gäbe der Kausalstrich die Zuordnung der Effortvariable und der Flowvariable als Eingabeparameter beziehungsweise als Ausgabeparameter dieser Funktion an. Wird diese Interpretation ausgeweitet, muss eine der Bondvariablen in das Element eingegeben und die jeweils andere vom Element ausgegeben werden, wodurch der Effort und Flow aus logischer Sicht gegenläufige Richtungen besitzen müssen.
Der Kausalstrich kann demzufolge zusammen mit dem Bondpfeil als eine logische Richtungsangabe für Effort und Flow gesehen werden. Hierbei ist der Effort immer auf den Kausalstrich gerichtet und somit der gegenläufige Flow dem Kausalstrich entsprechend entgegen gerichtet.
Befindet sich der Kausalstrich auf der Seite des Zielelements, fließt der Effort aus logischer Sicht in das Element hinein und stellt somit die Führungsgröße dar, von welcher der hinausfließende Flow logisch abhängt, wodurch für diesen Bond die Effortkausalität vorliegt. Im Fall der gegenteiligen Flowkausalität hängt der Effort vom führenden Flow ab und der Kausalstrich befindet sich am Ursprung des Bond.
Für die logische Betrachtung kann sinnbildlich festgelegt werden, dass die auf das Zielelement gerichtete Bondvariable die führende Größe darstellt und die jeweils andere von dieser abhängt.

2.4. Elemente

Zur Beschreibung von physikalischen System werden in Bondgraphen verallgemeinerte Elemente verwendet, welche die zugehörigen Grundelemente der jeweiligen Domäne im Bondgraph repräsentieren und deren Verhalten in Bezug auf die gespeicherte Energie oder den Leistungsfluss mathematisch beschreiben.
Bondgraphen setzen sich aus Quellen und Senken sowie Knotenpunkten und Umformern zusammen, wobei mindestens eine Quelle und eine Senke enthalten sein müssen.

Die Grundelemente dieser vier Grundtypen werden in den folgenden Unterabschnitt einführend beschrieben.

Die Gleichungen von Bondelementen liefern den Zusammenhang zwischen den Effort- und Flowvariablen der angeschlossen Bonds. In Abhängigkeit des Elementtyps kann das jeweilige Element einen oder mehrere Eingangsbonds und keinen oder mehrere Ausgangsbonds sowie mindestens eine interne Gleichung besitzen.

Es ist zu beachten, dass Bondelemente grundsätzlich ideale Elemente beziehungsweise Grundeigenschaften des Systems repräsentieren, wodurch diese lediglich mit ihren spezifischen Eigenschaften ohne Berücksichtigung von Nebeneffekten wirken. Die zusätzlichen Effekte von realen Systemen oder Elementen, welche vom idealen Fall abweichen, müssen im jeweiligen Modell durch Kombination von Grundelementen zusätzlich modelliert werden, sofern diese nicht vernachlässigbar sind.
Im Fall eines realen elektrischen Kondensator C wird dessen ideale Speicherfunktion durch ein kapazitives Senkenelement modelliert, jedoch muss zusätzlich der parasitäre Spannungsabfall beziehungsweise Leistungsverlust aufgrund des ohmschen Widerstandes der Zuleitungen durch ein Grundelement mit resistiven Eigenschaften sowie deren induktives Verhalten durch ein inertes Grundelement beschrieben werden.

2.4.1. Senken

Senken bilden die Endpunkte eines Bondgraphen und repräsentieren die physikalischen Eigenschaften des dargestellten Systems in Bezug auf den Verbrauch und die Speicherung von Energie. Der Resistor R ist hierbei das einzige energieverbrauchende Element. Energie kann des Weiteren in potentieller Form vom kapazitiven Element C und in Form von kinetischer Energie vom inerten Element I gespeichert und wieder abgegeben werden. Die drei genannten Grundelemente sind in den folgenden Unterabschnitten 2.4.1.1 bis 2.4.1.3 beschreiben.
Es ist zu beachten, dass die Endpunkte eines Bondgraphen in der Literatur[1] als *1-Port*-Elemente bezeichnet werden und in der vorliegenden Arbeit hierfür jedoch der Begriff Senke verwendet wird.

2.4.1.1. Resistor R

Der Resistor R ist das einzige energieverbrauchende Element und besitzt daher einen einzelnen eingehenden Bond, wobei an dessen Eingangsbond ein starres Verhältnis zwi-

schen Effort e(t) und Flow f(t) herrscht. Das Verhältnis wird in der Elementgleichung des Reistors durch den konstanten Widerstandsfaktor R definiert:

$$R = \frac{e(t)}{f(t)} \tag{2.16}$$

Die Entsprechungen des allgemeinen Resistors in den verschiedenen physikalischen Domänen sowie die Einheit des Parameters sind in Tabelle 2.3 zusammengestellt.

Domäne	Parameter	Einheit	Beispiel
Translation	d_{trans}	$\frac{Ns}{m}$	Dämpfer
Rotation	d_{rot}	Nms	Rotationsdämpfer
Fluidtechnik	c	$\frac{Ns}{m^5}$	hydraulischer Widerstand
Elektrotechnik	R	Ω	Ohmscher Widerstand

Tabelle 2.3.: Domänenspezifische Parameter des Resistors

Die Kausalität des angeschlossenen Bonds kann beliebig zwischen Flow- und Effortkausalität gewählt werden. Im Fall einer Flowkausalität $e(f)$ liegt für den Resistor die folgenden Gleichung vor, in welcher die abhängige Variable durch den Widerstandswert R und den eingehenden Flow f beschrieben wird.

$$e(t) = R * f(t) \tag{2.17}$$

Im Fall der alternativen Effortkausalität hängt der ausgegebene Flow vom Effort ab:

$$f(t) = \frac{1}{R} * e(t) \tag{2.18}$$

Die domänenspezifischen Elementgleichungen des Resistors in Abhängigkeit der beiden möglichen Kausalitäten sind in Tabelle 2.4 nachfolgend dargestellt.

Domäne	Kausalität e(f)	Kausalität f(e)
Translation	$F = d_{trans} * v$	$v = \frac{1}{d_{trans}} * F$
Rotation	$M = d_{rot} * \varphi$	$\varphi = \frac{1}{d_{rot}} * M$
Fluidtechnik	$p = c * \dot{V}$	$\dot{V} = \frac{1}{c} * p$
Elektrotechnik	$U = R * I$	$I = \frac{1}{R} * U$

Tabelle 2.4.: Domänenspezifische Gleichungen des Resistors

2.4.1.2. Capacitor C

Der Capacitor C ist das erste der beiden energiespeichernden Grundelemente. Da es sich beim Capacitor um ein ideales Element handelt, wird die über dessen einzigen Eingangsbond zugeführte Leistung nicht verbraucht, sondern in Form von potentieller Energie von diesem gespeichert. Potentielle Energie wird beispielsweise im Fall der Translation in Federn, in der Rotation in Drehstabfedern sowie in der Fluiddynamik in Form von schwerkraftgetriebenen Hochbehältern gespeichert. In der Elektrotechnik entspricht das Verhalten von Kondensatoren den Speichern für potentielle Energie, jedoch wird hier der Begriff elektrische Energie verwendet.
Die Speicherfähigkeit eines verallgemeinerten Capacitors wird von Compliance C bestimmt, welche sich analog zur elektrischen Kapazitätskonstante C verhält. In der Mechanik wird anstelle der Compliance C üblicherweise die Steifigkeit $k = \frac{1}{C}$ verwendet.
Die Entsprechungen des Capacitors sowie jeweiligen Einheiten der Compliance C entsprechend der verschiedenen physikalischen Domänen sind in Tabelle 2.5 zusammengetragen.

Domäne	Parameter	Einheit	Beispiele
Translation	$k = \frac{1}{C}$	$k = \frac{N}{m}$	Feder
Rotation	$k_{rot} = \frac{1}{C}$	$k_{rot} = \frac{1}{C} = \left[\frac{Nm}{rad}\right]$	Torsionsfeder
Fluidtechnik	C_h	$\frac{m^5}{N}$	Hydrospeicher
Elektrotechnik	C	F	Kondensator

Tabelle 2.5.: Physikalische Entsprechungen des Capacitors

Die im Capacitor gespeicherte Energie hängt sowohl vom Effort $e(t)$, als auch von der Zustandsvariable $q(t)$ ab und kann durch die bereits bekannte Gleichung 2.13 beschrieben werden. Es ist zu beachten, dass der normale Flow $f(t)$ des eingehenden Bonds durch die Zustandsableitung $\dot{q}(t)$ ersetzt wird, wodurch sich die Leistung $P(t)$ dieses Bonds gemäß der ebenfalls bekannten Gleichung 2.15 berechnen lässt.

Der Effort $e(t)$ eines kapazitiven Speichers C kann durch Integration des zugehörigen Flows beschrieben werden, wobei das Integrationsergebnis zusätzlich durch den verallgemeinerten Nachgiebigkeitskoeffizienten C dividiert wird.

$$e(t) = \frac{1}{C} \int f(t)dt \tag{2.19}$$

$$= k \int f(t)dt \tag{2.20}$$

Das Integral des Flow $f(t)$ kann hierbei durch die allgemeine Verschiebung $q(t)$ gemäß Gleichung 2.5 ersetzt werden, woraus die allgemeine Grundgleichung des Capacitors resultiert:

$$e(t) = \frac{1}{C} q(t) \tag{2.21}$$

Diese Form der Grundgleichung wird als Integralform beziehungsweise als Gleichung mit integrativer Kausalität (engl.*: integrative causality*) bezeichnet und stellt für die Zustandsraummodellierung die bevorzugte Erscheinungsform dar, da hierbei der Effort $e(t)$ des Bond direkt von der Zustandsvariable $q(t)$ abhängt.
Die Elementgleichung kann auch in Abhängigkeit des Efforts $e(t)$ dargestellt werden, was als Differentialform (engl.: *differential causality*) bezeichnet wird und lediglich in Konfliktfällen (siehe Abschnitt 3.7.2 auf Seite 73) angewendet wird.

$$q(t) = Ce(t) \tag{2.22}$$

Die resultierenden Elementgleichungen des Capacitors für Effort- und Flowkausalität sind in der folgenden Tabelle 2.6 für die verschieden physikalischen Domänen dargestellt.

Domäne	Kausalität e(q)	Kausalität q(e)
Translation	$F = kx$	$x = \frac{1}{k} F$
Rotation	$M = k_{rot}\varphi$	$\varphi = \frac{1}{k_{rot}} M$
Fluidtechnik	$p = \frac{1}{C_h} V$	$V = C_h p$
Elektrotechnik	$u = \frac{1}{C} \psi$	$\psi = Ce$

Tabelle 2.6.: Domänenspezifische Gleichungen des Capacitors

2.4.1.3. Inertia I

Die Inertia I repräsentiert inerte Energiespeicher und ist das zweite Speicherelement. Wie auch der Capacitor und der Resistor, ist das inerte Element ein ideales Grundelement. Es wird keine Energie verbraucht, sondern lediglich gespeichert und wieder abgegeben. Die gespeicherte Energie und der Leistungsfluss sind durch die bekannten Gleichungen 2.12 und 2.14 gegeben.
Die Entsprechungen der Inertia und des zugehörigen Trägheitsparameter I für die verschiedenen physikalischen Domänen sind in der Tabelle 2.7 auf der nachfolgenden Seite zusammengefasst.

Domäne	Parameter	Einheit	Beispiel
Translation	m	kg	Massenträgheit
Rotation	J	Nms^2	Rotationsträgheit
Fluidtechnik	L_h	$\frac{Ns^2}{m^5}$	hydraulische Induktivität
Elektrotechnik	L	H	elektrische Induktivität

Tabelle 2.7.: Domänenspezifische Parameter der Inertia

Analog zum kapazitiven Speicherelement C besteht ein integraler Zusammenhang zwischen dem Effort und Flow des Eingangsbonds, wobei der Flow $f(t)$ mit dem Integral des Effort $e(t)$ und der reziproken Trägheitskonstante I beschrieben werden kann:

$$f(t) = \frac{1}{I} \int e(t)dt \tag{2.23}$$

Des Weiteren kann das Integral des Effort durch den verallgemeinerten Bondimpuls $p(t)$ gemäß Gleichung 2.4 ausgedrückt werden, was analog zum Capacitor in der Grundgleichung der Inertia mit *integrative causality* resultiert:

$$f(t) = \frac{1}{I} p(t) \tag{2.24}$$

Die *integrative causality* ist wie beim Capacitor die bevorzugte Erscheinungsform der Elementgleichung für die Zustandsraumdarstellung, da diese ausschließlich von der Zustandsvariable $p(t)$ abhängt. Neben dieser existiert die *differential causality* Variante 2.25, welche jedoch nur in Konfliktfällen verwendet wird.

$$p(t) = I f(t) \tag{2.25}$$

Inerte Elemente existieren beispielsweise in der Mechanik in Form massebehafteter und somit trägheitsbehafteter Bauteile sowie in der Elektrotechnik als Elemente mit induktiven Effekten wie Spulen oder normale Leitungen. Die domänenspezifischen Gleichungen für die Entsprechungen der Inertia sind in der folgenden Tabelle 2.8 gelistet.

Domäne	Kausalität f(p)	Kausalität p(f)
Translation	$v = \frac{1}{m} p$	$p = mv$
Rotation	$w = \frac{1}{J} L$	$L = J\omega$
Fluidtechnik	$\dot{V} = \frac{1}{L_h} p_p$	$p_p = L_h \dot{V}$
Elektrotechnik	$i = \frac{1}{L} \Phi$	$\Phi = Li$

Tabelle 2.8.: Physikalische Entsprechungen der Inertia

2.4.2. Quellen

Quellen definieren die Eingänge eines Bondgraphen und können entweder als Flow- oder Effortquellen vorliegen, wobei für diese ähnlich den Senken in der Literatur ebenfalls die zugehörigen englischen Bezeichnungen *effort source* und *flow source* gebräuchlich sind, welche in der vorliegenden Arbeit ebenfalls synonym verwendet werden.
Die jeweilige Quellvariable ist in der Zustandsraumdarstellung des Graphen Bestandteil des Eingangsvektors $\vec{u}$ und kann im endgültigen Zustandsraummodell mit beliebigen mathematischen Funktionen beziehungsweise Verhaltensprofilen belegt werden. Quellen definieren je nach Typ entweder den Effort oder Flow des angeschlossenen Bonds, wobei die jeweils andere Bondvariable nicht durch die Quelle selbst definiert ist, sondern sich aus den Gleichungen des Graphen indirekt ergibt, aber keinen weiteren Einfluss auf die Modellbildung hat und daher häufig vernachlässigt wird. Es ist zu beachten, dass die nicht definierte Variable in der vorliegenden Toolboxversion nicht verwendet wird.
Der Ausgangsbond einer Quelle muss immer eine zu der Quelle passende Kausalität besitzen und die Gleichung des nachfolgenden Elementes von der Quellvariable abhängig sein, da die Quellvariable unabhängig vom Bondgraph und somit immer die führende Variable eines Bond ist. Hierdurch können die an Effortquellen angeschlossenen Elemente lediglich Effortkausalität $f(e)$ und die von Flowquellen nur Flowkausalität $e(f)$ besitzen. Es ist des Weiteren zu beachten, dass in der aktuellen Version der Toolbox die jeweilige Variable einer Quelle mit großen Buchstaben in den Gleichungen bezeichnet wird.
Die Bondsymbole für allgemeine Flow- und Effortquellen sind in der folgenden Abbildung 2.4 mit der jeweiligen Kausalität des Ausgangsbond dargestellt.

(a) Effortsource (b) Flowsource

Abbildung 2.4.: Quellen

In der aktuellen Toolboxversion wird der Typ einer Quelle im Simulinkmodell nicht im zugehörigen Quellelement, sondern durch die Wahl der Kausalität des angeschlossenen Ausangsbonds im nachfolgenden Element festgelegt.

2.4.3. Knoten

In der Regel bestehen Systeme aus mehr Elementen, als lediglich einer Quelle mit einer einzelnen Senke, wodurch eine Möglichkeit zur Verteilung beziehungsweise zur Verzweigung des Leistungsflusses auf verschiedene Teilsysteme des Gesamtsystems in Form von Knotenpunkte zur Modellierung benötigt wird.

Diese Knotenpunkte verfügen über mindestens einen Eingangs- und zwei Ausgangsbonds, wobei der eingespeiste Leistungsfluss der eingehenden Bonds verlustfrei auf die angeschlossenen Ausgangsbonds entsprechend der nachfolgenden Teilsysteme aufgeteilt ist. Die verhältnismäßige Verteilung von der Leistung und somit die Verhältnisse der Effort- und Flowvariablen der angeschlossenen Bonds sind durch die internen Gleichungen eines Knotens definiert.

Knotenpunkte liegen in einer seriellen und in einer parallelen Grundvariante in Bezug auf die Verteilung des eingehenden Leistungsfluss vor, welche direkt mit den Verschaltungen von Bauelementen in der Elektrotechnik verglichen werden können, aber analog in allen weiteren physikalischen Domänen existieren. In der Elektrotechnik wird in Reihenschaltung und Parallelschaltung unterschieden, was ebenfalls für Bondknoten gilt, wobei die Bezeichnungen *1-Knoten* für seriellen und *0-Knoten* für parallele Knoten verwendet werden.

2.4.3.1. Serielle Knoten

Analog zu elektrischen Reihenschaltungen, in welchen alle Bestandteile vom gleichem Strom durchflossen werden, sind die Flowvariablen aller an einem seriellen 1-Knoten angeschlossenen Eingangs- als auch Ausgangsbonds identisch. Hierdurch besitzen an jedem seriellen Knoten mit n Eingangsbonds und m Ausgangsbonds alle Flowvariablen die gleichen Werte gemäß folgender Gleichung:

$$f_{in_1} = f_{in2} = ... = f_{in_n} = f_{out_1} = f_{out_2} = ... = f_{out_m} \tag{2.26}$$

Aufgrund von Gleichung 2.26 müssen die Efforts der an einem seriellen Knoten angeschlossenen Bonds unterschiedlich zueinander beziehungsweise variabel sein. Die Summe der eingehenden Efforts muss an seriellen Knoten immer gleich der Summe der ausgehenden Efforts sein:

$$\sum_{i=1}^{n} e_{in_i} = \sum_{j=1}^{m} e_{out_j} \tag{2.27}$$

Die allgemeine Effortbilanz 2.27 eines seriellen Knoten kann demnach auch nach null umgestellt angegeben werden, was der Darstellung von Spannungen in elektrischen Maschen gemäß der Kirchhoffschen Gesetze ähnelt.

$$0 = \sum_{i=1}^{n} e_{in_i} - \sum_{j=1}^{m} e_{out_j} \tag{2.28}$$

2.4.3.2. Parallele Knoten

Der parallele 0-Knoten besitzt wie serielle 1-Knoten ebenfalls ein interne Identität und eine interne Bilanz, jedoch sind bei diesem Knotentyp die Rollen von Effort und Flow vertauscht, sodass sämtliche angeschlossenen Bonds von einem gemeinsamen Effort getrieben und von variablen Flows durchströmt werden. Dieser Knotentyp wird beispielsweise verwendet, um in der Mechanik von einer einzelnen Kraft gleichzeitig angetriebene Systeme oder in der Elektrotechnik von einer Spannung gespeiste parallele Systeme zu modellieren.
Basierend auf dem vertauschten Verhalten von Effort und Flow liegt an einem parallelen Knoten mit n Eingängen und m Ausgängen die folgende allgemeine Effortidentität vor:

$$e_{in_1} = e_{in2} = ... = e_{in_n} = e_{out_1} = e_{out_2} = ... = e_{out_m} \tag{2.29}$$

Des Weiteren muss sich die Flowbilanz zwischen eingehenden und ausgehenden Bonds von parallelen Knoten analog zur Effortbilanz serieller Knoten zu null ergeben:

$$\sum_{i=1}^{n} f_{in_i} = \sum_{j=1}^{m} f_{out_j} \tag{2.30}$$

Die Flowbilanz kann ebenfalls als zu null umgestellt angegeben werden:

$$0 = \sum_{i=1}^{n} f_{in_i} - \sum_{j=1}^{m} f_{out_j} \tag{2.31}$$

2.4.4. Umformer

Als Umformer werden in der vorliegenden Arbeit Bondelemente bezeichnet, die genau über einen eingehenden sowie einen ausgehenden Bond verfügen und deren Aufgabe beziehungsweise deren Eigenschaft in der Änderung des Verhältnisses der Effort- und Flowvariablen der beiden angeschlossenen Bonds zu einander anzusehen ist.
Es existieren in dieser Elementkategorie zwei Grundtypen: der *Transformer TF* und der *Gyrator GY*. Beide Elemente repräsentieren ideale Übersetzungen zwischen den Efforts und Flows der angeschlossenen Bonds. Da hierbei von den Umformern keine Leistung verbraucht wird und diese über keine Speichereigenschaften verfügt, ist das Produkt von Effort und Flow des eingehenden mit dem des ausgehenden Bonds identisch. Die zugeführte Eingangsleistung $P_{in}(t)$ stimmt stets mit der Ausgangsleitung $P_{out}(t)$ gemäß der folgenden Gleichungen überein:

$$P_{in}(t) = P_{out}(t) \tag{2.32}$$

$$e_{in}(t) * f_{in}(t) = e_{out}(t) * f_{out}(t) \tag{2.33}$$

Da der eingehende gleich dem ausgehenden Leistungsfluss ist, muss der Eingangsbond demnach dieselbe Richtung wie der Ausgangsbond aufweisen.
Die beiden Grundtypen der Umformer sind in der folgenden Abbildung 2.5 mit der üblichen Leistungsrichtung dargestellt, wobei der Transformer über Effortkausalität verfügt und beim Gyrator der Flow die führende Bondvariable ist.

(a) Tranformer TF (Effortkausalität) (b) Gyrator GY (Flowkausalität)

Abbildung 2.5.: Umformer

Umformer werden üblicherweise zur Modellierung von Übersetzungsprozessen innerhalb einer physikalischen Domäne verwendet. Sie können aber auch zur Beschreibung eines Übergangs zwischen Teilsystemen verschiedener Domänen angewendet werden, da ihre internen Übersetzungsfaktoren zur Konvertierung der physikalischen Einheiten von Effort und Flow der beiden angeschlossenen Bonds genutzt werden können.

2.4.4.1. Transformer TF

Der Transformer TF verbindet den Effort $e_{in}(t)$ des eingehenden Bonds mit dem Effort $e_{out}(t)$ des ausgehenden Bonds durch den Faktor m, welcher das Verhältnis der beiden Efforts bestimmt, wodurch die Elementgleichung den Charakter einer Übertragungsfunktion erhält:

$$e_{in}(t) = m * e_{out}(t) \tag{2.34}$$

Ebenfalls beschreibt der Faktor m das Verhältnis der Flowvariablen am Transformer, wobei jedoch Ausgangs- und Eingangsbond vertauscht sind:

$$f_{out}(t) = m * f_{in}(t) \tag{2.35}$$

Der Transformator dient beispielsweise zur Repräsentation von mechanischen Getrieben oder elektrischen Transformatoren, wobei lediglich die Übertragungseigenschaften der idealen Bauteile durch den Block abgebildet werden. Eventuell für das reale Bauteil vorliegende Zusatzeffekte oder Leistungsverluste sind unter Einbezug weiterer Grundelemente zu modellieren.

2.4.4.2. Gyrator GY

Der Gyrator GY besitzt wie der Transformer TF je einen Eingangs- und Ausgangsbond sowie zwei interne Gleichungen. Es erfolgt jedoch eine Verknüpfung des Effort mit dem Flow des jeweils anderen angeschlossenen Bonds.
Der Effort $e_{in}(t)$ des eingehenden Bonds wird durch Multiplikation des ausgehenden Flow $f_{out}(t)$ mit dem internen Übertragungsfaktor r des Gyrators berechnet.

$$e_{in}(t) = r * f_{out}(t) \tag{2.36}$$

Der gleiche Zusammenhang besteht zwischen dem Effort $e_{out}(t)$ des Ausgangsbonds und des Flow $f_{in}(t)$ des Eingangsbonds:

$$e_{out}(t) = r * f_{in}(t) \tag{2.37}$$

Wie auch die restlichen Grundelemente ist der Gyrator ein ideales Element, wodurch es bei der Übertragung der eingehenden Leistung zum nachfolgenden Teilsystem zu keinem Leistungsabfall kommt.

2.5. Beispiel

Die Verwendung von Bondgraphen zur Modellierung physikalischer Systeme soll anhand des folgenden Beispiels eines mechanischen Federschwingers demonstriert werden. Das Beispielsystem ist in untenstehenden Abbildung 2.6 schematisch dargestellt.

Es ist zu beachten, dass nicht alle der in diesem Abschnitt aufgeführten Schritte zur Modellierung eines physikalischen System durch Bondgraphen nötig sind. Zur effizienten Anwendung der Bondgraphenmethodik genügt häufig die Identifikation der einzelnen Bestandteile des zu modellierenden Systems sowie deren Verkettung untereinander, um diese anschließend direkt mit Hilfe der Bondelemente und der generalisierten Grundgleichungen unter Verwendung der elementspezifischen Parameter abzubilden. Hierauf basiend kann nachfolgend das Zustandsraummodell des Systems ermittelt werden.

2.5.1. Darstellung als Bondgraph

Der Federschwinger besteht aus einer Masse m, welche mit einer Feder und einem Dämpfer vertikal an einem starren Punkt im Raum nach unten gerichtet fixiert ist, wodurch die Gewichtskraft F_g die Masse nach unten zieht. Das System soll durch die nach oben gerichtete Eingangskraft F_E angeregt werden.

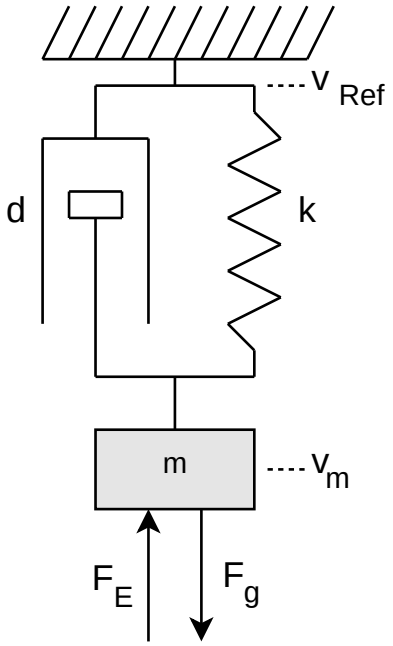
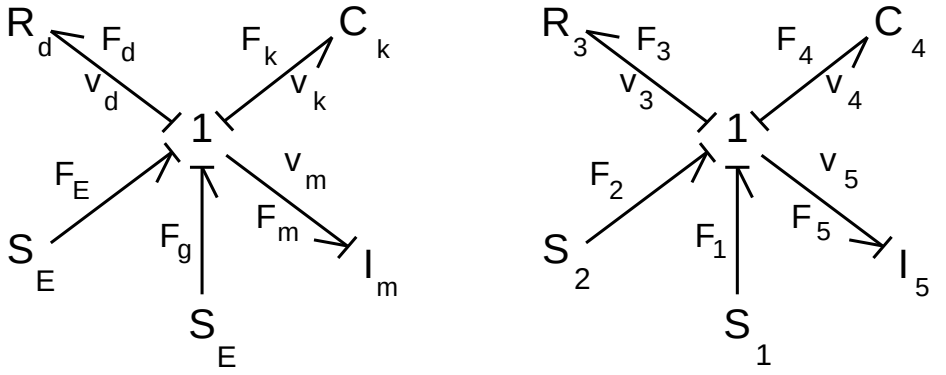


Abbildung 2.6.: Federschwinger

Die Bestandteile des Federschwingers werden als ideal angenommen, wodurch lediglich die typspezifischen Wirkungen dieser Elemente für die Modellierung des Systems relevant sind und die idealen Elemente direkt mit den in Abschnitt 2.4.1 beschriebenen Bondelemente ausgedrückt werden können.
Die Eingangskraft F_E und die Gewichtskraft F_g werden durch jeweils eine Effortquelle repräsentiert. Entsprechend der Tabellen 2.3, 2.5 und 2.7 können die Masse m als Inertia I_m, der Dämpfer d als Resistor R_d und die Feder k als Capacitor C_k dargestellt werden.

Die Erstellung des Bondgraphen kann durch den Umstand, dass die beweglichen Systembestandteilen die gleichen Geschwindigkeiten aufgrund der zwischen ihnen vorhandenen steifen Verbindungen besitzen müssen, vereinfacht durchgeführt werden.

Aufgrund dieser identischen Bewegungsgeschwindigkeiten können die beiden Effortquellen direkt über einen seriellen 1-Knoten zur Repräsentation der identischen Geschwindigkeiten mit dem Resistor, dem Capacitor und der Inertia verbunden werden. Hierdurch ergibt sich der in Abbildung 2.7 mit domänenspezifischen und generalisierten Variablen dargestellte Bondgraph für das System in seiner minimalen Erscheinungsform. Es ist jedoch zu beachten, dass trotz der gleichen Geschwindigkeiten der Systemkomponenten aus formalen Gründen die Geschwindigkeitsvariablen beziehungsweise deren zugehörige Flowvariablen elementspezifisch indiziert werden und die Kausalitäten der Bonds sowie deren Leistungsrichtung gemäß Abschnitt 2.5.2 eingezeichnet sind.



(a) Bondgraph mit spezifischen Variablen

(b) Bondgraph mit generalisierten Variablen

Abbildung 2.7.: Federschwinger in Bondgraphendarstellung

Wird der Bondgraph des Systems auf unverkürztem Weg erstellt, werden hierfür im ersten Schritt die Bewegungsgeschwindigkeiten der einzelnen Systemkomponenten betrachtet werden. Im Fall des Federschwingers bewegt sich der Massepunkt m mit der Geschwindigkeit v_m, wobei bekannterweise die Feder und der Dämpfer sich mit der selben Geschwindigkeit bewegen müssen. Zusätzlich besitzt die Aufhängung des Federschwingers aufgrund der Unbeweglichkeit die Referenzgeschwindigkeit $v_{Ref} = 0$.
Für beide Geschwindigkeiten wird im ersten Schritt der Modellierung jeweils ein serieller Knoten erstellt, an welchen die jeweilige durch das System vorgegebene Geschwindigkeit vorherrscht. Die Bezugsgeschwindigkeit des Systems v_{Ref} wird hierbei durch das Ankoppeln einer Flowsource S_F am zugehörigen Knoten repräsentiert. Die Masse m bewegt sich mit der Geschwindigkeit v_m und wird im Graph als inertes Senkenelement dargestellt und als Inertia I_m direkt an den zugehörigen seriellen Knoten angekoppelt.

Die das System anregende Eingangskraft F_E sowie die Gewichtskraft F_g der Masse werden in Form von zwei Effortquellen an den seriellen Knoten der Masse angehängt. Zwischen diesen beiden seriellen Knoten werden anschließend die Bondelemente der Feder k sowie des Dämpfers d eingebunden. Dies geschieht für beide Elemente jeweils durch Einfügen eines parallelen 0-Knotens zwischen den beiden zuvor erstellten seriellen Knoten für die Geschwindigkeiten v_{Ref} und v_m, wobei an diesen beiden Knoten der Capacitor C_k beziehungsweise der Resistor R_d angekoppelt werden.

Der hieraus resultierende Initialgraph ist in der folgenden Abbildung 2.8 dargestellt.

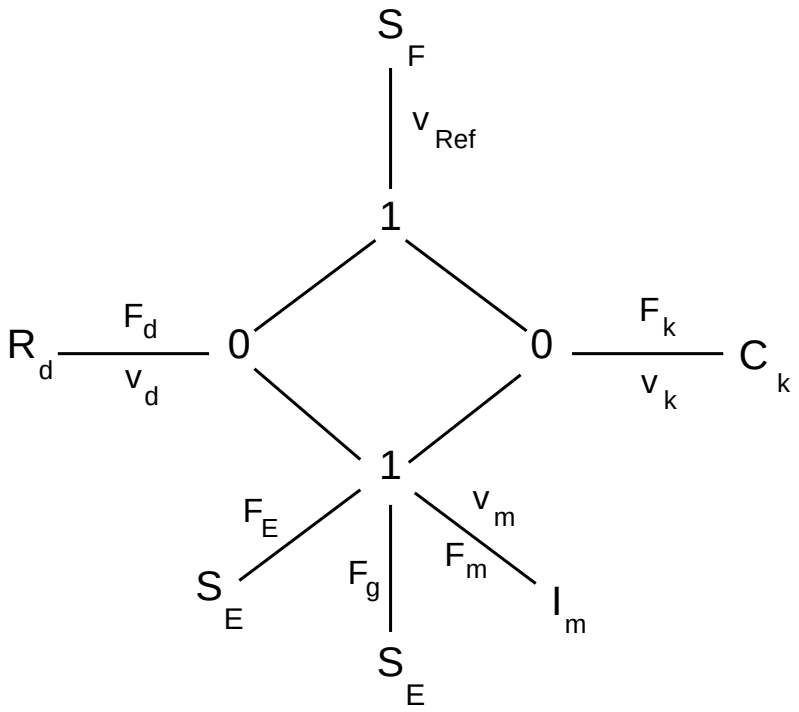


Abbildung 2.8.: Initialgraph

Im Anschluss hieran wird mit der Minimierung des Bondgraphen begonnen, wobei als erster Schritt alle Flow- und / oder Effortquellen entfernt werden, welche den Wert Null besitzen. Der aus dem Entfernen der hiervon betroffenen Flowquelle für v_{Ref} des Beispiels resultierende Graph ist in Abbildung 2.9 dargestellt.

Im zweiten Schritt werden alle Knoten mit lediglich zwei angeschlossenen Bonds entfernt und durch jeweils einen einfachen Bond ersetzt, da aufgrund ihrer internen Gleichungen aus den Abschnitten 2.4.3.1 und 2.4.3.2 nur für drei oder mehr angeschlosse Bonds wirksam sind und andernfalls lediglich die Funktionalität eines einfachen Bonds aufweisen.

Es wird im Beispiel willkürlich mit der Entfernung des seriellen Knotens begonnen, an welchem ursprünglich die bereits entfernte Flowquelle für v_{Ref} angeschlossen war, und anschließend der Prozess für die beiden parallelen 0-Knoten des resistiven Elementes R_d und des kapazitiven Elementes C_k wiederholt.
Die hieraus resultierenden und aufeinander aufbauenden Graphen sind in den nachfolgenden Abbildungen 2.11 und 2.11 sukzessive dargestellt.

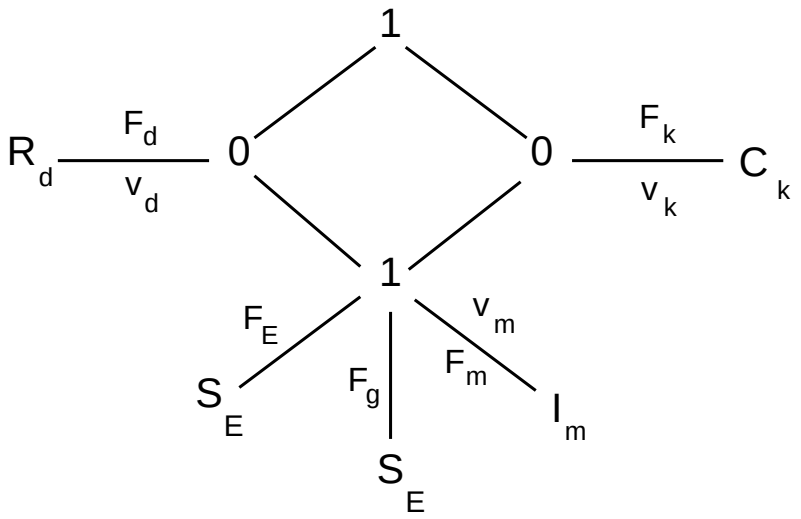


Abbildung 2.9.: Graph nach Entfernung der Flowquelle für v_{Ref}

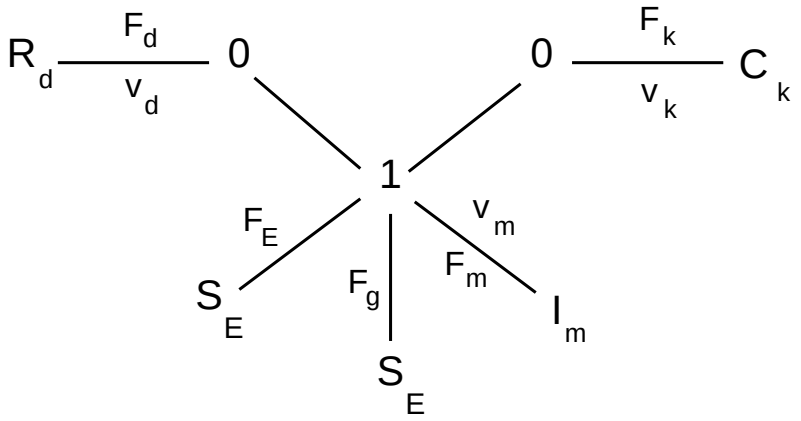


Abbildung 2.10.: Graph nach Entfernung des wirkungslosen 1-Knotens

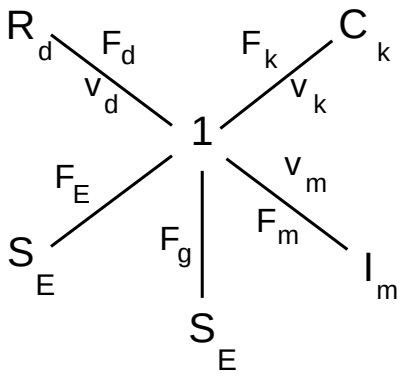


Abbildung 2.11.: Graph nach Entfernung der beiden wirkungslosen 0-Knoten

2.5.2. Grundgleichungen

Der Dämpfer entspricht dem verallgemeinerten Bondelement Resistor und verfügt über die spezifische Grundgleichung 2.38, welche den Zusammenhang zwischen der Dämpfungskraft F_d und der Geschwindigkeit des Federschwingers v_m durch die Dämpfungskonstante d beschreibt, wobei diese bereits in Flowkausalität angegeben ist:

$$F_d = d * v_d = d * v_m \tag{2.38}$$

Werden die spezifischen Variablen F_d und v_d sowie die Dämpfungskonstante d durch die generalisierten Bezeichner der Bondgraphen gemäß Tabelle 2.1 ersetzt, ergibt sich die Grundgleichung des Dämpfers, wie folgt. Es ist zu beachten, dass der resistive Faktor R mit dem Bezeichner der Dämpfungskonstante d als Index ergänzt wurde.

$$e_d = R_d * f_d \tag{2.39}$$

Die domänenspezifische Grundgleichung der Feder ergibt sich aufgrund der Bevorzugung der Flowkausalität zur Verwendung der Integralform, wie folgt. Bei der Auslenkung x_k der mit der Feder verbundenen Masse handelt es sich um die Zustandsvariable der Feder, deren Ableitung $\dot{x}_k$ der Geschwindigkeit v_k entspricht.

$$F_k = k * x_k \tag{2.40}$$

Die Grundgleichung der Feder und somit die Beschreibung der Federkraft F_k in Abhängigkeit der Auslenkung x_k kann ebenfalls in verallgemeinerter Form für kapazitive Elemente dargestellt werden, da die Feder dem Bondelement Capacitor C entspricht.

Die Generalisierung der Grundgleichung erfolgt durch Substitution der zeitveränderlichen Variablen F_k und x_k.
Für die Übertragung der domänenspezifischen Variablen in verallgemeinerte Bondvariablen muss für dieses energiespeichernde Element zusätzlich zu Tabelle 2.1 für die Umwandlung der Zustandsvariable x_k ebenfalls Tabelle 2.2 verwendet werden.
Die Auslenkung x_k der Feder entspricht der generalisierten Verschiebung q_k, wodurch deren Ableitungen $\dot{q}_k$ dem Flow f_k des an den Capacitor C_3 angeschlossen Bonds beziehungsweise der Bewegungsgeschwindigkeit v_k der Feder entsprechen:

$$f_k = \dot{q}_k = v_k = \dot{x}_k \tag{2.41}$$

Es ist zu beachten, dass in der verallgemeinerten Grundgleichung der Feder deren Federkonstante k dem Reziproken der Compliance C entspricht, welche analog R_d zur einfacheren Zuordnung mit dem Index k versehen wurde:

$$e_k = k * q_k \tag{2.42}$$

$$= \frac{1}{C_k} * q_k \tag{2.43}$$

Die Grundgleichung des an der Feder und dem Dämpfer befestigten Körpers liefert den Zusammenhang zwischen der Trägheitskraft F_m mit der Beschleunigung a_m durch die Masse m:

$$F_m = m * a_m \tag{2.44}$$

Für die formelle Generalisierung und Übertragung der domänenspezifischen Grundgleichung 2.44 in das Bondgraphensystem genügt die Substitution der Variablen mit Hilfe der Tabellen 2.1 und 2.2 nicht, da kein Bezeichner von Bondvariablen der Beschleunigung a_m direkt entspricht. Um dennoch eine Umformung zu gewährleisten, wird Gleichung 2.44 nach der Zeit integriert:

$$\int F_m dt = \int m * a_m dt \tag{2.45}$$

$$p_m = m * v_m \tag{2.46}$$

Im Anschluss an die Integration kann die Substitution der Variablen gemäß der Tabellen 2.1 und 2.2 erfolgen, wobei der Bezeichner des Impulses p_m bereits mit dem des generalisierten Bondimpulses p übereinstimmt:

$$p_m = m * f_m \tag{2.47}$$

Aufgrund der bevorzugten integrativen Kausalität für Zustandsgleichungen wird die resultierende Gleichung 2.47 nach dem Flow f_m umgestellt und wird somit von der Zustandsvariablen p_m abhängig gemacht. Es ist zu beachten, dass der Bezeichner der Masse m im Zuge der Umformung durch den Konstantenbezeichner I für inerte Elemente ersetzt wurde, wobei dieser mit dem Bezeichner der physikalischen Masse m indiziert wurde.

$$f_m = \frac{1}{I_m} p_m \tag{2.48}$$

Die Gleichung besitzt hierdurch Effortkausalität, da der Flow f_m explizit beschrieben wird. Die Zustandsableitung $\dot{p}_m$ des in der Gleichung enthaltenen Impuls p_m entspricht dem generalisierten Effort e_m des zugehörigen Bonds und somit der Trägheitskraft F_m:

$$\dot{p}_m = e_m = F_m \tag{2.49}$$

Die Summe der beiden Eingangskräfte des System, die Kraft F_E und F_g, verteilt sich vollständig auf die Summe der Dämpfungskraft F_d, der Federkraft F_k und der Trägheitskraft F_m beziehungsweise der Ableitung $\dot{p}_m$ des Impuls p_m. Hierdurch existiert für das System das folgenden Kräftegleichgewicht 2.50:

$$F_E + F_g = F_d + F_k + \dot{p}_m \tag{2.50}$$

Gleiches gilt für die zugehörigen generalisierten Bondvariablen des Systems, was einer seriellen Verkettung der Elemente und somit einem seriellen 1-Knoten im Bondgraph des vertikalen Federschwingers entspricht. Das Kräftegleichgewicht 2.50 wird im Zuge der Generalisierung durch ein Effortgleichgewicht am seriellen Knoten ausgedrückt:

$$e_E + e_g = e_d + e_k + \dot{p}_m \tag{2.51}$$

Aufgrund der seriellen Verkettung der Elemente bewegen sich diese immer mit der gleichen Geschwindigkeit, wodurch die folgende Identität gilt:

$$v_d = \dot{x}_k = v_m \tag{2.52}$$

Am seriellen Knoten des Bondgraph liegt somit ebenfalls die nachfolgende Identität 2.53 der Flowvariablen vor. Es ist zu beachten, dass die theoretisch vorliegen Flowvariablen der Effortquellen üblicherweise vernachlässigt werden, da diese in Bezug auf die Flowvariablen der anderen Bondelemente des Graphen redundant sind und somit nur die Flowvariablen der am seriellen Knoten angeschlossen Senken in der Identität 2.53 vorkommen:

$$f_d = \dot{q}_k = f_m \tag{2.53}$$

2.5.3. Zustandsraumdarstellung

Im Anschluss an das Aufstellen der Systemgleichung sowie deren Generalisierung erfolgt die Ermittlung der Zustandsraumdarstellung auf Basis dieser Gleichungen.

Für die Darstellung im Zustandsraum muss zuerst der Zustandsvektor $\vec{x}$ des Systems aufgestellt werden. Dieser enthält die Zustandsvariablen der energiespeichernden Systemkomponenten. Im aktuellen Beispiel handelt es sich hierbei um die Auslenkung x_k der Feder und dem Impuls p_m der Masse beziehungsweise deren generalisierte Bondentsprechungen q_k und p_m:

$$\vec{x} = \begin{pmatrix} x_k \\ p_m \end{pmatrix} = \begin{pmatrix} q_k \\ p_m \end{pmatrix} \tag{2.54}$$

Durch Ableitung des Zustandsvektors $\vec{x}$ können die zeitlichen Änderungen der Zustandsvariablen und somit der Ableitungsvektor $\dot{\vec{x}}$ erzeugt werden:

$$\dot{\vec{x}} = \begin{pmatrix} \dot{x}_k \\ \dot{p}_m \end{pmatrix} = \begin{pmatrix} \dot{q}_k \\ \dot{p}_m \end{pmatrix} \tag{2.55}$$

Des Weiteren werden die beiden Eingangskräfte und somit die Quellvariablen der beiden zugehörigen Effortquellen im Eingangsvektor $\vec{u}$ zusammengefasst.
Die Leistungsübertragung der Bonds für die Eingangskraft F_E und die Gewichtskraft F_g sind beide auf den seriellen Knoten gerichtet, obwohl die beide Kräfte eigentlich gegeneinander wirken. Im Bondgraph wird dieser Umstand nicht durch Invertierung der Bonds, sondern durch die Negierung der Wert $-mg$ der Gewichtskraft F_g berücksichtigt:

$$\vec{u} = \begin{pmatrix} e_E \\ e_g \end{pmatrix} = \begin{pmatrix} F_E \\ F_g \end{pmatrix} = \begin{pmatrix} F_E \\ -mg \end{pmatrix} \tag{2.56}$$

Die Elemente des Ausgangsvektors $\vec{y}$ können frei gewählt werden. In diesem Beispiel soll zusätzlich zu den bereits im Ableitungsvektor $\dot{\vec{x}}$ zu ermittelnden Variablen auch die auf die Masse wirkende Kraft F_m und die Bewegungsgeschwindigkeit v_m der Masse ermittelt werden. Um dies zu erreichen, wird der Ausgangsvektor $\vec{y}$ in der folgender Form festgelegt:

$$\vec{y} = \begin{pmatrix} F_m \\ v_m \end{pmatrix} = \begin{pmatrix} \dot{p}_m \\ v_m \end{pmatrix} \tag{2.57}$$

Nach der Erstellung der benötigten Vektoren wird mit der Ermittlung der einzelnen Zustandsableitungen aus dem Vektor $\dot{\vec{x}}$ fortgefahren, was nachfolgend beschrieben ist.

Die Ableitung $\dot{q}_3$ des generalisierten Verschiebung q_3 ist das oberste Element in $\dot{\vec{x}}$ und gleicht durch die Identität 2.53 der Flowvariable f_4. Diese hängt nur von der Zustandsvariable p_4 ab, wodurch $\dot{q}_3$ direkt aufgelöst werden kann:

$$\dot{q}_k = f_m \tag{2.58}$$

$$= \frac{1}{I_m} p_m \tag{2.59}$$

Als zweites Element des Vektor $\dot{\vec{x}}$ wird die Ableitung $\dot{p}_m$ des Impulse p_m der Masse ermittelt. Diese kommt ausschließlich im Kräftegleichgewicht 2.50 beziehungsweise in der zugehörigen Effortbilanz 2.51 vor, welche nach dieser umgestellt wird:

$$\dot{p}_m = e_E + e_g - e_d - e_k \tag{2.60}$$

Die Efforts e_E und e_g sind als Bestandteile des Eingangsvektors $\vec{y}$ Quellvariablen und bedürfen daher als bekannte Variablen keiner weiteren Auflösung. Des Weiteren hängt der Effort e_k gemäß 2.43 lediglich von der Zustandsvariable q_k ab und kann direkt in die Gleichung für $\dot{p}_m$ eingesetzt werden.

$$\dot{p}_4 = e_E + e_g - e_d - \frac{1}{C_k} q_k \tag{2.61}$$

Der noch unaufgelöste Effort e_d kann durch die Gleichung des resistiven Elementes 2.39 beschrieben werden, wobei der enthaltene Flow f_d beziehungsweise die Bewegungsgeschwindigkeit v_d des Dämpfers ebenfalls identisch mit dem Flow f_m der Masse ist:

$$e_d = R_d * f_d = R_d * f_m = \frac{R_d}{I_m} * p_m \tag{2.62}$$

Die resultierende Gleichung für e_d kann anschließend in Gleichung 2.61 eingesetzt werden, wodurch diese Gleichung ausschließlich von Zustandsvariablen und Quellvariablen abhängt:

$$\dot{p}_m = e_E + e_g - \frac{R_d}{I_m} * p_m - \frac{1}{C_k} q_k \tag{2.63}$$

Für die Elemente des Ausgangsvektors $\vec{y}$ sind keine weiteren Ermittlungsschritte nötig, da $\dot{\vec{p}}_m$ durch die zuvor ermittelte Gleichung 2.63 und v_m durch die Zustandsgleichung 2.48 beschrieben sind.

Anschließend können die Zustandsgleichungen des Systems zusammengefasst werden:

$$\begin{pmatrix} \dot{q}_k \\ \dot{p}_m \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{I_m} \\ -\frac{1}{C_k} & -\frac{R_d}{I_m} \end{pmatrix} \begin{pmatrix} q_k \\ p_m \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} e_E \\ e_g \end{pmatrix} \tag{2.64}$$

Die Ausgangsgleichungen des Systems sind mit den Zustandsgleichungen bis auf die Vertauschung der beiden Zeilen identisch:

$$\begin{pmatrix} \dot{p}_m \\ f_m \end{pmatrix} = \begin{pmatrix} -\frac{1}{C_k} & -\frac{R_d}{I_m} \\ 0 & \frac{1}{I_m} \end{pmatrix} \begin{pmatrix} q_k \\ p_m \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} e_E \\ e_g \end{pmatrix} \tag{2.65}$$

2.6. Reguläre Systemmodellierung

Projektziel ist es, die Modellierung von Systemen durch die Verwendung von Bondgraphen in Matlab / Simulink zu vereinfachen und somit eine Alternative zu der regulären Darstellung von Systemen in Simulinkmodellen mittels manuell ermittelter Differentialgleichungen zu schaffen.

Für die Modellierung des in Abschnitt 2.5 als Beispiel gewählten Federschwingers als Bondgraphen mit Hilfe der Toolbox muss lediglich der in Abbildung 2.7 dargestellte Bondgraph implementiert und parametrisiert werden. Die einzelnen Bondelemente werden hierbei durch Simulinkblöcke der Toolbox repräsentiert, wodurch der Graph in Simulink analog Abbildung 2.7 konstruiert werden kann. Die Extraktion des zugehörigen Zustandsraummodells des Systems wird anschließend von der Toolbox durchgeführt.
Hierdurch beschränkt sich der Modellierungsaufwand auf die Identifikation der Bondelemente für das zu modellierenden System und die Erstellung des hieraus resultierenden

Graphen in Simulink. Eine weiterführende Analyse analog den Abschnitten 2.5.2 und 2.5.3 oder eine direkte Identifikation der Systemgleichungen ist nicht notwendig.

Wird der Federschwinger ohne Hilfe der Toolbox auf regulären Weg in Simulink modelliert, muss hierfür das zu modellierende System durch Differentialgleichungen beschrieben werden. Diese Differentialgleichungen müssen zuvor für das jeweilige System manuell hergeleitet und anschließend als Simulinkmodell implementiert werden.
Im Fall des Federschwingers kann hierfür das folgenden Kräftegleichgewicht als Ansatz verwendet werden:

$$F_E - F_g = F_T + F_d + F_k \tag{2.66}$$

Da sich alle Komponenten des Federschwingers mit der Geschwindigkeit der v_m der Masse bewegen und auch deren Auslenkung x besitzen, können die beschleunigungsabhängige Trägheitskraft F_T, die geschwindigkeitsabhängige Dämpfungskraft F_d und die positionsabhängige Federkraft F_k in Abhängigkeit der Auslenkung x sowie derer zeitlichen Ableitungen $\dot{x}$ und $\ddot{x}$ dargestellt werden:

$$F_E - m * g = m * a + d * v + k * x \tag{2.67}$$

$$= m * \ddot{x} + d * \dot{x} + k * x \tag{2.68}$$

Die Differentialgleichung des Systems wird in Simulink üblicherweise mit Hilfe von stufenweiser Integration der zeitlichen Ableitungen gelöst. Um dies zu erreichen, wird die Gleichung nach der Trägheitskraft F_T umgestellt, da diese von der höchsten zeitlichen Ableitung der Auslenkung x abhängt:

$$F_T = F_d + F_k - F_E + F_g \tag{2.69}$$

$$m * \ddot{x} = d * \dot{x} + k * x - F_E + m * g \tag{2.70}$$

Innerhalb des Simulinkmodells werden die Geschwindigkeit $\dot{x}$ und die Auslenkung x aus der Trägheitskraft F_T durch Division durch mit der Masse m und anschließender Integration erzeugt.
Die auf diese Weise berechnete Geschwindigkeit sowie die Auslenkung werden mit Hilfe von Rückführungen innerhalb des Modells zur Berechnung der Dämpfungskraft F_d beziehungsweise der Federkraft F_k genutzt, welche wiederum der Ermittlung der Trägheitskraft F_T dienen.

Das resultierende Simulinkmodell für die Differentialgleichung des Federschwinger ist umseitig in Abbildung 2.12 dargestellt.

Gegenüber der in diesem Abschnitt beschriebenen Implementierung des zu simulierenden Systems auf regulären Weg unter Verwendung der systembeschreibenden Differentialgleichungen entfällt bei der Verwendung der Bondgraphen-Toolbox das vor allem bei größeren System aufwändige manuelle Herleiten der Systemgleichungen. Die einzelnen Eigenschaften der Systemkomponenten sowie deren Struktur muss lediglich identifiziert und als Bondgraph elementbasiert in Simulink konstruiert werden.
Der Aufwand für die Modellierung wird durch die automatisierte Extraktion der Zustandsraummodelle durch die Toolbox weiter reduziert und der Benutzer somit entlastet.

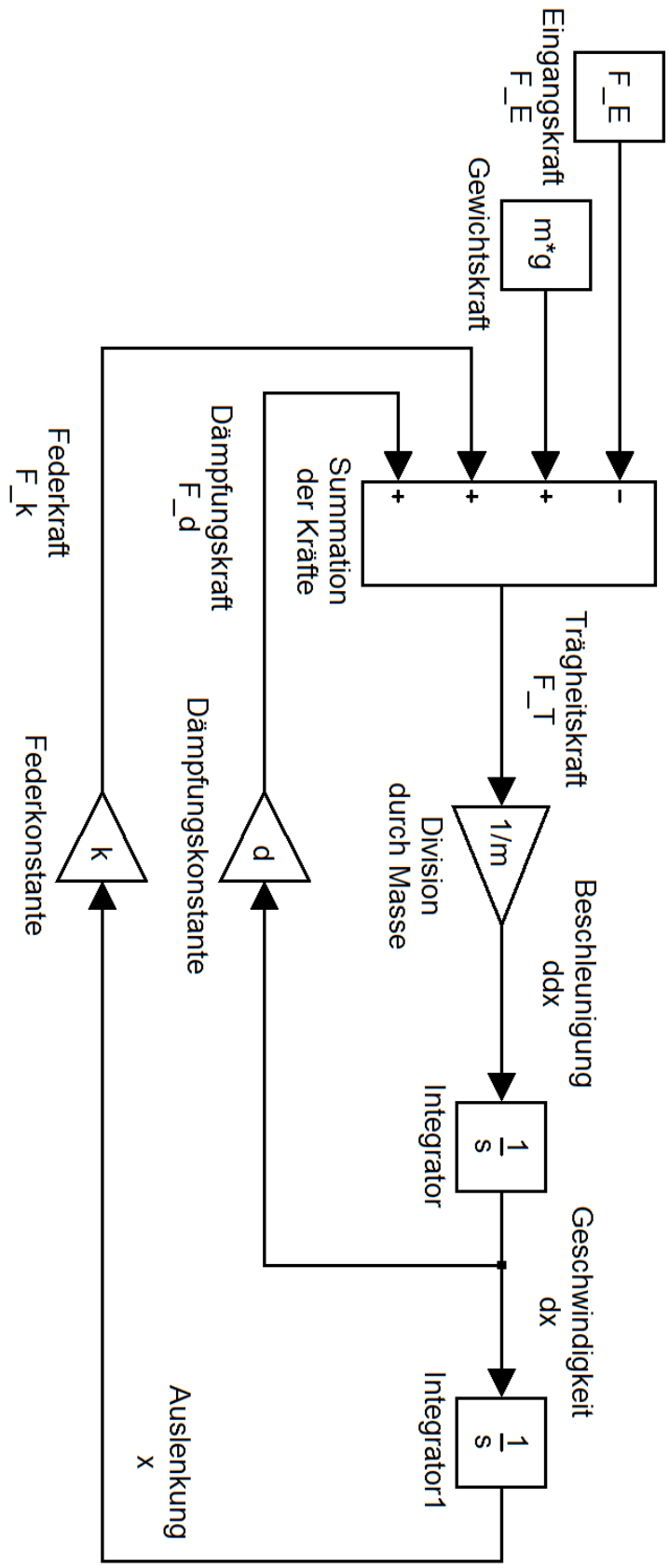


Abbildung 2.12.: Federschwinger als Differentialgleichung in Simulink

3. Realisierung

3.1. Grundlegender Ablauf

Grundidee des Projektes ist es eine Toolbox zu entwickeln, welche eine unkomplizierte Erstellung von Modellen mit Hilfe von Bondgraphen ermöglicht. Der Bondgraph wird hierbei vom Benutzer in der für Bondgraphen üblichen Art in Form von Quellen, Knoten, Senken und Umformer in einem normalen Simulinkmodell graphisch per Drag-and-Drop zusammengefügt. In der aktuellen Version der Toolbox stehen die nachfolgend gelisteten und bereits in Abschnitt 2.4 auf Seite 22 beschriebenen Grundelemente zur Verfügung:

Senken:

- Resistor R: resistiver Verbraucher
- Capacitor C: kapazitiver Energiespeicher
- Inertia I: inerter Energiespeicher

Verteiler

- paralleler 0-Knoten
- serieller 1-Knoten

Umformer

- Gyrator GY
- Transformer TY

Zusätzlich zur strukturellen Erstellung des Bondgraphen muss dieser vom Benutzer parametrisiert werden, wobei sich dieser Prozess lediglich auf das Eintragen der elementspezifischen Werte wie Widerstand, Kapazität, Übertragungsfaktoren und ähnliches beschränkt. Eine genaue Spezifikation des Graphen hinsichtlich der Kausalität der einzelnen Bonds ist nicht zwingend erforderlich, da diese Festlegungen vom Programm

automatisch vorgenommen werden. Die Kausalitäten können trotzdem vom Benutzer manuell festgelegt beziehungsweise modifiziert werden, was bei einer günstigen Wahl den Rechenaufwand gegenüber dem automatischen Vorgehen der Toolbox reduziert.

Die Sequenz der Hauptschritte der aktuellen Toolboxversion ist begleitend zu den folgenden Beschreibungen in Abbildung 3.1 schematisch dargestellt.

Aus dem modellierten Bondgraph wird eine Auflistung der einzelnen Bonds und die Gleichungen der Elemente gemäß ihrer Kausalität generiert. Diese tabellarische Repräsentation des Graphen bildet im weiteren Programmverlauf die Grundlage für Ermittlung der Zustandsraumdarstellung.
Die Bondelemente im Simulinkmodell bestehen jeweils aus einer maskierten *S-Function*. Diese Funktionen werden in Simulink ähnlich normaler Blöcke verwendet und realisieren die Anbindung des jeweiligen Elements an das Hauptprogramm. Beim Simulationsstart wird in den normalen Ablauf von Simulink mit Hilfe der modelleigenen Callback-Funktionen eingegriffen. Während der Initialisierungsphase des Modells wird durch diese Callbacks jedes Element auf einer *globalen Elementeliste* registriert. Diese enthält die Eigenschaften sowie die interne Simulink-Objekt-IDs der Vorgänger und Nachfolger sämtlicher Bondelemente des Graphen, wobei diese Informationen aus den Runtime-Objekte[1] der Simulinkblöcke gewonnen werden.

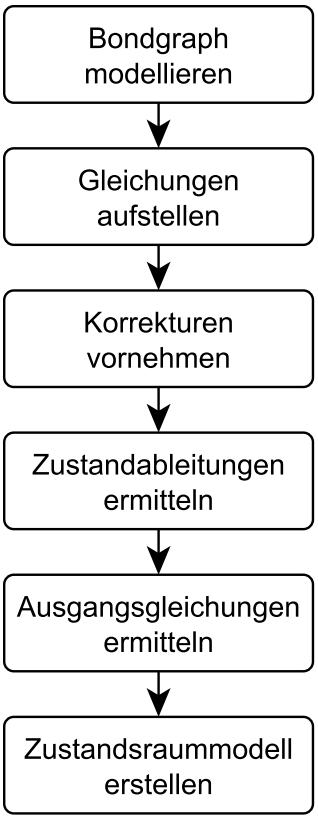


Abbildung 3.1.: Grundablauf

Basierend auf der Elementenliste wird im Anschluss eine Auflistung der einzelnen Bonds und derer auf den Elementen basierenden Eigenschaften erstellt. Aus dieser *globalen Bondliste* werden anschließend sämtliche benötigten Gleichungen und Identitäten konstruiert. Des Weiteren erfolgt anhand der Bondliste die Identifikation der für die Ermittlung der Zustände essentiellen Speicherelemente.

Zusätzlich zu der Elementen- und Bondliste wird eine weitere globale Liste erstellt. Diese wird als *Knotenliste* bezeichnet und beinhaltet die lösungsrelevanten Informationen

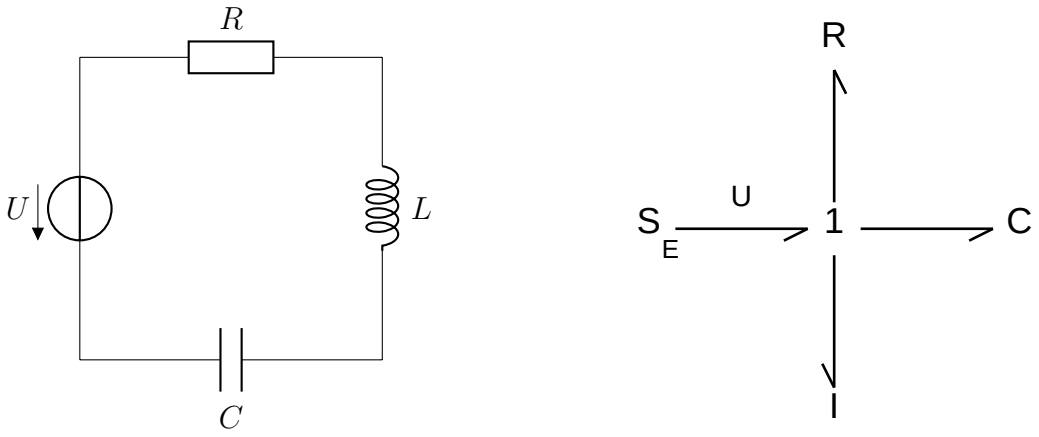
[1] Die internen Runtime-Objekte von Simulink werden in den folgenden Kapiteln mit RTO abgekürzt.

sämtlicher Knoten, wie Bilanzen sowie Effort- beziehungsweise Flowidentitäten in symbolischer Form.
Nach der Erstellung dieser drei Listen werden mit deren Hilfe die einzelnen Zustandsvariablen der Kapazitäten und Induktivitäten sowie deren Ableitungen identifiziert. Die Zustandsvariablen werden in den Zustandsvektor $\vec{x}$ und die zugehörigen Zustandsableitungen in den Vektor $\dot{\vec{x}}$ als symbolische Matlabvariablen eingetragen. Gleiches geschieht mit den Quellen des Graphen, deren Effort- beziehungsweise Flowvariablen dem Eingangsvektor $\vec{u}$ zugewiesen werden. Sofern ein oder mehrere Elemente im Graphen vom Benutzer über deren Maske als "Ausgang" markiert wurden, werden die Effort- und Flowvariablen der betreffenden Eingangsbonds in den Ausgangsvektor $\vec{y}$ eingefügt.

Für die Erstellung des Zustandsraummodells müssen alle Zustandsableitungen des Vektors $\dot{\vec{x}}$ durch Gleichungen beschrieben werden, welche lediglich von Variablen des Eingangsvektors $\vec{u}$ und des Zustandsvektors $\vec{x}$ abhängen dürfen. Um dies zu erreichen wird ein mehrstufiger Lösungsalgorithmus mit Korrekturvorstufe und interner Simulation der Substitutionen unter Zuhilfenahme der genannten Listen angewendet. Der Ablauf dieses Mechanismus sowie die Funktionsprinzipien der beinhalteten Unterstufen werden in Abschnitt 3.9 erläutert.
Im Anschluss an die erfolgreiche Aufstellung der symbolischen Gleichungen der Zustandsableitungen wird das gleiche Lösungsverfahren angewendet, um die Ausgangsgleichungen zur Beschreibung des Vektors $\vec{y}$ in Abhängigkeit von $\vec{u}$ und $\vec{x}$ zu ermitteln, sofern die benötigten Lösungen nicht schon während der Ermittlung von $\dot{\vec{x}}$ aufgestellt wurden.

3.2. Modellierung mit Toolbox

Die Modellierung eines Bondgraphen erfolgt durch das für Simulink typische Drag-and-Drop-Verfahren, bei welchem die einzelnen Bestandteile des Bondgraphen vom Benutzer lediglich in die Arbeitsfläche des Simulinkmodells gezogen und anschließend gemäß der gewünschten Bonds miteinander verbunden werden müssen.
Innerhalb der Erläuterung der einzelnen Programmbestandteile und -abläufe in den folgenden Kapiteln wird unter anderem auf das Beispiel eines elektrischen Reihenschwingkreises sowie des Verfahrens mit diesem und die daraus resultierenden Ergebnisse zurückgegriffen. Der Schaltplan sowie der zugehörige Bondgraph des Schwingkreises ist in Abbildung 3.2 umseitig dargestellt.



(a) Schaltplan (b) Bondgraph

Abbildung 3.2.: RLC-Reihenschwingkreis

Des Weiteren sollen die Element des Schwingkreis folgenden Werte besitzen:

- Widerstand $R = 1000\Omega$
- Induktivität $L = 1mH$
- Kapazität $C = 1\mu F$

Der vom Benutzer in Simulink zusammengesetzte Bondgraph des RLC-Schwingkreises ist in Abbildung 3.3. abgebildet. Die Reihenschaltung von Widerstand R wurde in diesem Modell durch Verkettung von zwei seriellen 1-Knoten umgesetzt. Für die Darstellung in Simulink ist zu beachten, dass es sich bei den Beschriftungen der Blöcken um die Dateinamen der zugrundeliegenden S-Function handelt, da in der aktuellen Version dieser Toolbox keine graphische Maskierung der Simulinkblöcke vorgenommen wird.
Die im Modell als "Quellspannung" bezeichnete Spannungsquelle U des Schwingkreises unterscheidet sich von den restlichen Bondelementen, da es sich bei dieser um einen gewöhnlichen "Constant"-Block von Simulink handelt und der Quelle somit keine S-Function zugrunde liegt. Die Spannungsquelle entspricht im Bondgraph einer Effortquelle. Im Zuge der Aufbereitung des Graphen wird diese Quelle nicht als Bondelemente erkannt, ist aber als Eingang mit dem Bondgraphen verbunden und wird somit als Quelle des Bondgraph beziehungsweise als Eingangsvariable im Vektor $\vec{u}$ für das resultierende Zustandsraummodell interpretiert.

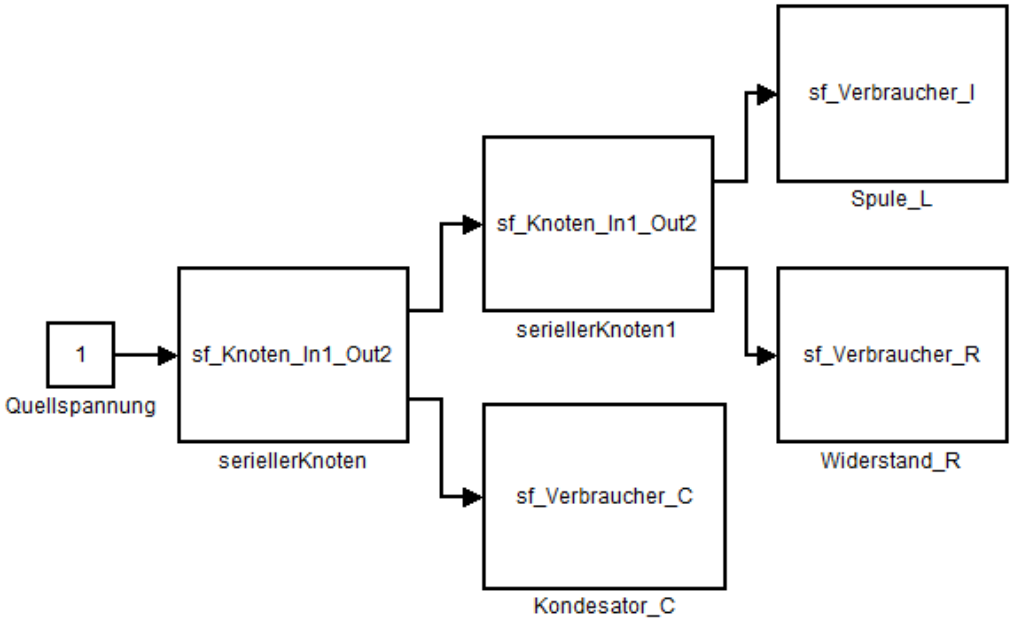


Abbildung 3.3.: RLC-Schwingkreis als Bondgraph in Simulink

Zusätzlich zur Platzierung der Bondelemente und der strukturellen Modellierung des Graphen, müssen die einzelnen Elemente vom Benutzer parametrisiert werden, was über das graphische Userinterface des jeweiligen Elementes geschieht. Grundsätzlich können bei jedem Bondelement die Kausalität und die Richtung des eingehenden Bonds festgelegt werden, wobei für die Kausalität zwischen Flow- und Effortkausalität sowie für die Richtung des Bond zwischen normaler und invertierter Leistungsübertragung gewählt werden kann. Des Weiteren kann in jedem Element der eingehende Bond als Ausgang des späteren Zustandsraummodells festgelegt werden, wodurch sowohl der Effort als auch der Flow des betreffenden Bonds als Variable im weiteren Verlauf des Programms in den Ausgangsvektor $\vec{y}$ aufgenommen werden.
Generell wird die Konfiguration eines Bonds immer in dessen Zielelement vorgenommen. Eine gegebenenfalls eingestellte Richtungsinvertierung der Leistungsübertragung beeinflusst die Rolle von Senke und Quelle des jeweiligen Bonds während der Modellierung nicht, sondern lediglich das Vorzeichen des betreffenden Efforts oder Flows in der Bilanz des vorausgehenden Knotens beziehungsweise der Gleichung des vorausgehenden Übertragungsgliedes. Die Invertierung von Bonds von Quellen ist in der aktuellen Version nicht implementiert und in der Modellierung nicht notwendig, da dies durch die Wahl der jeweiligen Quellvariable im resultierenden Zustandsraummodell realisiert werden kann.
Im Fall von Quellen wird über die Einstellung der Kausalität des Bonds ebenfalls am

nachfolgenden Element vorgenommen und hierdurch der Typ der Quelle festgelegt. Der resultierende Quellentyp in Abhängigkeit der Kausalität kann Tabelle 3.1 entnommen werden. In der vorliegenden Version der Toolbox gelten für Quellen die folgenden Einschränkung:

- Flowquellen dürfen nur an parallelen 0-Knoten angeschlossen werden
- Effortquellen dürfen nur an seriellen 1-Knoten anschlossen werden

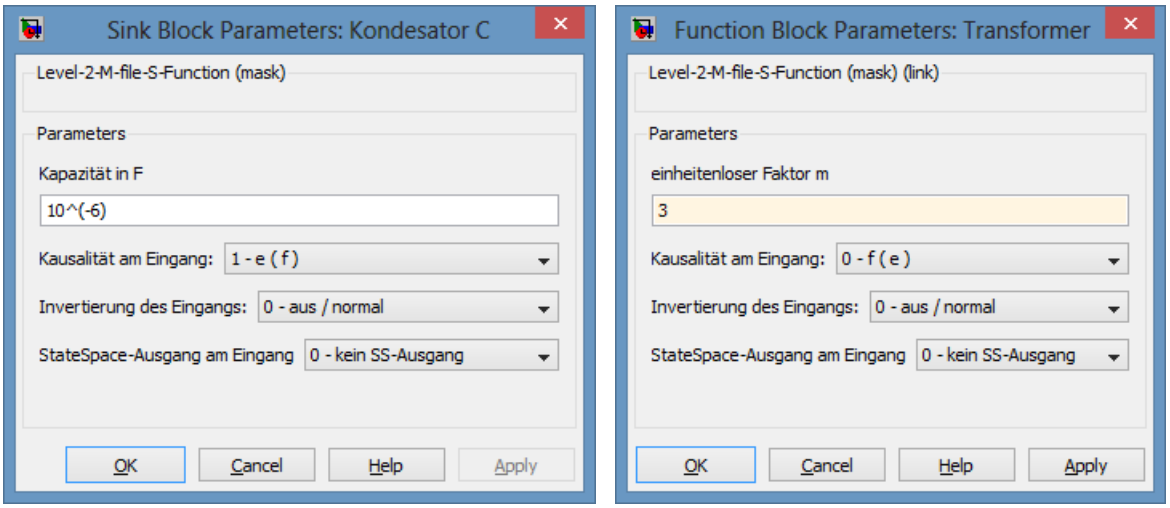
Die Identifikation von Quellen erfolgt in der aktuellen Version nur an den Eingängen von Knoten, wodurch es nicht möglich ist Quellen, direkt an Übertragern, Verbrauchern oder Energiespeichern anzuschließen.

Allg. Bezeichnung	Bondsymbol	Abhängigkeit	GUI-Angabe	Code-Angabe
effort causality	Quelle —>| Senke	f = f(e)	0 - f(e)	'f(e)'
flow causality	Quelle |—> Senke	e = f (f)	1 - e(f)	'e(f)'
Auto	Quelle —> Senke	-	2 - auto	'auto'

Tabelle 3.1.: Übersicht zu verschiedenen Angaben der Kausalität

Bei resistiven Verbrauchern, induktiven und kapazitiven Energiespeichern muss zusätzlich über die GUI jeweils der Widerstand R, die Induktivität I und die Kapazität C vom Benutzer eingestellt werden, andernfalls werden Standartwerte verwendet. Gleiches gilt für die zwei vorhandenen Übertragertypen, Transformer und Gyrator, bei denen jeweils die Faktoren m und r vom angegeben werden müssen. Die Benutzeroberflächen eines kapazitiven Speichers C sowie eines Transformers sind in Abbildung 3.4 auf der nächsten Seite beispielhaft für die restlichen Bondelemente abgebildet.
Da es sich bei Quellen um normale Simulink-Objekte handelt, können deren Werte beziehungsweise deren mathematische Effort- oder Flowfunktion nicht über den Block der Quelle festgelegt werden. Dies ist auch nicht notwendig, da die Variablen von Quellen im Vektor $\vec{u}$ registriert werden und somit die Eingänge des späteren Zustandsraummodells beliebig definiert werden können.

Im Anschluss an die Modellierung und Parametrisierung des Bondgraphen kann vom Benutzer die Simulation des Modells wie in einem gewöhnlichen Simulinkmodell gestartet werden. Über die modellinternen Callback-Funktionen wird hierbei das eigentliche Hauptprogramm der Toolbox aufgerufen und somit die Analyse des Graphen sowie die



(a) Kondensator (b) Transformer

Abbildung 3.4.: Graphische Benutzer Oberflächen von Elementen

Erstellung der Zustandsraumdarstellung eingeleitet wird. Sämtliche Konfigurationen des Simulinkmodells in Bezug auf zeitliche Schrittweiten, verwendete Solver sowie Start- und Stoppzeiten beeinflussen die weiteren Ausführungsstufen der Toolbox nicht und können beliebige Werte annehmen. Simulink wird lediglich als Modellierungsoberfläche verwendet, da sämtliche Prozesse der Toolbox in der Initialisierungsphase des Simulinkmodells durchgeführt werden.

3.3. Aufbereitung des Graphen

Durch den Simulationsstart des Simulinkmodells wird die weitere Ausführung der Toolbox eingeleitet, da während der Ausführung der modelleigenen Callback-Funktionen ebenfalls die internen S-Functions der Bondelementen sowie deren blockinterne Callback-Funktionen ausgelöst werden. Eine Übersicht über die Sequenz der relevanten Callback-Funktionen des Simulinkmodells und der durch diese ausgeführten Callbacks der Bondelemente ist in Abbildung 3.5 auf der nächsten Seite dargestellt.
Die Callback-Sequenz der einzelnen S-Functions wird im Zuge des *InitFcn()*-Callback des Modells ausgelöst. Es ist zu beachten, dass die Callback-Methoden der Elemente von jedem Element individuell abgearbeitet werden. Die Bearbeitungssequenz ist durch Simulink jedoch so abgestimmt, dass alle Elemente jeweils die gleiche Callback-Methode nacheinander ausführen.

Mit Beginn der Simulation wird für jeden im Modell enthaltenen Block ein individuelles RTO erzeugt. Diese RTOs stellen die internen Hauptbestandteile eines Simulationslaufs dar und werden für die Aufbereitung des Bondgraphen genutzt. Bevor die Analyse des Graphen durchgeführt wird, werden im Zuge des *setup()*-Callback in den *UserData*-Felder der jeweiligen RTOs die in Tabelle 3.2 auf der nächsten Seite beschriebenen Subfelder angelegt und jeweiligen objektindividuellen Angaben beschrieben. Tabelle 3.2 enthält hierbei die Originalwerte des Widerstands R aus der verwendeten Beispielsimulation. Es ist zu beachten, dass für das Feld der Kausalität der englische Bezeichner *.Causality* gewählt wurde, da in Matlab keine Umlaute zugelassen sind.

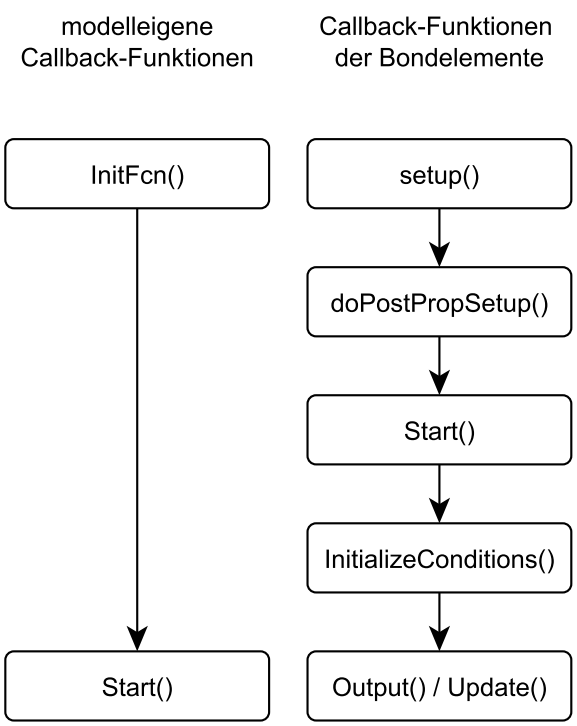


Abbildung 3.5.: Callback-Sequenzen

Im Anschluss an die Konfiguration des UserData-Feldes wird das jeweilige Bondelement in der *globalen Elementeliste* registriert beziehungsweise diese zuvor (neu) erstellt, wenn diese nicht existieren sollte oder nicht dem erwarteten Format entspricht. Hierfür wird die zu diesem Zweck geschaffene Hilfsfunktion *registrieren_ Element()* [2] verwendet. An diese Funktion wird bei Aufruf das RTO des jeweiligen Elementes übergeben, aus welchem die benötigten Informationen für die Elementeliste ausgelesen werden.
Die globale Elementeliste ist hierbei ein Vektor, wobei jeweils ein Vektorelemente eine Substruktur für ein Bondgraphenelement ist und in welcher die für die weitere Bearbeitung nötigen Informationen enthalten sind. Die aus dem Programmlauf resultierende Elementeliste für den RLC-Reihenschwingkreis ist in Tabelle 3.3 auf Seite 55 dargestellt und ist der Beispielsimulation entnommen.

Zusätzlich zur Elementeliste wird in parallelen 0-Knoten und in seriellen 1-Knoten die Erstellung beziehungsweise Erweiterung der *globalen Knotenliste* vorgenommen. Bei der Knotenliste handelt es sich hierbei um eine Struktur, welche für jeden Knoten die grund-

[2] *Quellcode: siehe Anhang B.4.2 auf Seite 199*

Feld	Wert	Beschreibung
.Toolbox	'Bondgraph'	Dient der Zuordnung des Objektes für den Fall, das mehrere ähnlich funktionierende Toolboxen im Modell parallel verwendet werden
.ID	Elementezahl	Wert der globalen Variable ELEMENTEZAHL , welche bei der Erstellung eines jeden Elementes inkrementiert wird und dieses somit eindeutig identifiziert
.Kategorie	'Verbraucher'	Kategorie des Elementes
.Typ	'Resistor'	Typ des Elementes für Detailverarbeitung
.Wert	1000	Wert des Grundparameters des Elementes in SI-Basiseinheit
.Name	'Widerstand R'	Bezeichnung des Elementes in Simulinkmodell
.Invertierung	false	Logischer Parameter zur Angabe der Invertierung des eingehenden Bonds
.Causality	'f(e)'	Angabe der Kausalität des eingehenden Bonds
.Ausgang	false	Logischer Parameter zur Markierung des Effort und Flow des eingehenden Bonds als Ausgangsvariablen des Zustandsraummodells
.Handle	gcbh	*gcbh()* gibt das Handle des aufrufenden RTO zurück und kann direkt als Variable gcbh verwendet werden. Das Handle wird später für die Erstellung der Bondliste benötigt.

Tabelle 3.2.: UserData-Struktur von Widerstand R

legenden Informationen zur Beschreibung von dessen Identitätsgleichungen und Bilanzen enthält. Tabelle 3.4 auf der nächsten Seite bildet die Knotenliste des Beispielgraphen ab, aus welcher ebenfalls die Eigenschaften und Felder der Knotenliste entnommen werden können.

Es ist zu beachten, dass Identitätsgleichungen in der Entwurfs- und Implementierungsphase der Toolbox noch als *Gleichnisse* bezeichnet wurden. Dies wurde zwar im weiteren Verlauf des Projektes fallengelassen, jedoch findet sich diese Bezeichnung weiterhin in den Funktionsnamen der Toolbox und deren Quellcode sowie in den entnommen Beispieldaten des Programms in der vorliegenden Arbeit.

Zu diesem Zeitpunkt der Programmausführung wird die Knotenliste zwar vollständig erstellt, aber noch nicht vollständig beschrieben. Es werden lediglich die Felder *.ID* sowie *.Typ* mit konkreten Werten beschrieben und die restlichen Felder deklariert, wobei

Zeile	ID	Name	Typ	Wert	Handle
1	1	'Kondensator_C'	'Capacitor'	1.0000e-006	4.0010
2	2	'Spule_L'	'Inductor'	1.0000e-003	6.0010
3	3	'Widerstand_R'	'Resistor'	100	7.0011
4	4	'seriellerKnoten'	'Serieller Knoten'	1	8.0011
5	5	'seriellerKnoten1'	'Serieller Knoten'	1	9.0010
6	6	'Quellspannung'	'Eingang'	0	5.0011

Tabelle 3.3.: *Elementeliste* für Beispiel *RLC-Reihenschwingkreis*

diese erst in der späteren Abarbeitung der Hauptroutinen die benötigten Informationen erhalten. Das Verfahren zur Vervollständigung der Knotenliste ist in Abschnitt 3.5 beschrieben.

Eigenschaft	Datentyp	1. Knoten	2. Knoten
ID	double	4	5
Typ	cell	{ 'Serieller Knoten' }	{ 'Serieller Knoten' }
Gleichnisse	cell	{ }	{ }
Inputs	cell	{ }	{ }
Outputs	cell	{ }	{ }

Tabelle 3.4.: unvollständige Knotenliste

Nach Abschluss der *setup()*-Methode der einzelnen Elemente und deren Registrierung auf der Elementeliste wird in diesen jeweils nachfolgend die *doPostPropSetup()*-Callback-Methode ausgeführt, durch welche die benötigten Bondverbindungen auf der *globalen Bondliste* eingetragen werden, was durch Aufruf der Toolboxfunktion *registrieren_Bond()*[3] in dem genannten Callback geschieht. Diese Hilfsfunktion erstellt die Bondliste bei Nichtvorliegen gegebenenfalls initial und trägt den jeweiligen Bond auf dieser ein.
Die Bondliste ist von der Datenstruktur ähnlich der Elementeliste aufgebaut, wobei jede Zeile eine Struktur enthält und einem Bond entspricht. Grundsätzlich verfügt jeder Bond über die nachfolgend beschriebenen Eigenschaften, welche die einzelnen Felder der einzelnen Bondstrukturen in der Bondliste bilden:

.BondID

Das Feld *BondID* beinhaltet die Identifizierungsnummer des jeweiligen Bonds. Diese zur Indizierung des zugehörigen Effort und Flow sowohl in den Gleichungen der Elemente als auch in den Identitätsgleichungen und Bilanzen der Knoten verwendet.

[3] *Quellcode: siehe Anhang B.4.1 auf Seite 197*

.Inversion

Das Feld *Inversion* gibt die Richtung des Leistungsübertragung des Bonds in Form eines logischen Wertes an. Bei Inversion *false* erfolgt die Leistungsübertragung normal von der Quelle zur Senke des jeweiligen Bond. Andernfalls invertiert die Richtung der Übertragung, was sich während der Gleichungserstellung durch Negierung des zugehörigen Effort beziehungsweise Flow in der Bilanz des vorgelagerten Knotens äußert.

.Causality

Das Feld *Causality* gibt die Kausalität des Bond als Zeichenkette an. Die möglichen Werte können Tabelle 3.1 auf Seite 51 entnommen werden. Es ist zu beachten, dass für den Namen des Feldes ebenfalls der englische Begriff für Kausalität gewählt wurde, da Umlaute als Bezeichner in Matlab nicht zulässig sind.

.Gateway

Der logische Wert des Feld *Gateway* gibt an, ob es sich bei dem jeweiligen Bond um eine Schnittstelle zu Simulinkblöcken handelt, welche keine Bondelemente der Toolbox sind. In der aktuellen Version werden hierdurch die Quellen des Graphen identifiziert, wobei dieser Parameter in späteren Versionen ebenfalls zur Ermittlung von Ausgängen des Graphen zu nachfolgenden Simulinkblöcken genutzt werden kann. Für den Bezeichner des Feldes wurde ebenfalls ein englischer Begriff gewählt, um gegebenfalls auftretende Umlautproblematiken zu vermeiden.

.Ausgang

Der logische Wert des Feld *Ausgang* dient zur Markierung des Bonds als Ausgang des Zustandsraummodells. Wird dieser Parameter mit *true* angegeben, werden Effort und Flow des jeweiligen Bond dem Ausgangsvektor $\vec{y}$ hinzugefügt.

.Source

Das Feld *Source* enthält die relevanten Eigenschaften des Quellelementes am Ursprung des Bond. Die Eigenschaften können Tabelle 3.6 auf der nächsten Seite entnommen werden, welche Beispielwerte für Widerstand R aus der Originalsimulation enthält.

.Destination

Das Feld *.Destination* wird analog zu *.Source* für den Endpunkt des Bond verwendet.

Die resultierende Bondliste des Beispielgraph ist in Tabelle 3.5 auf der nächsten Seite dargestellt, wobei zu beachten ist, dass auf eine vollständige Wiedergabe der Inhalte der jeweiligen *.Source*- und *.Destination*-Unterstrukturen verzichtet wird und stattdessen lediglich die Bezeichnug des jeweiligen Quell- und Zielelementes vermerkt ist.

ID	Inversion	Causality	Gateway	Ausgang	Source	Destination
1	false	'e(f)'	false	false	'seriellerKnoten'	'Kondensator_C'
2	false	'f(e)'	false	false	'seriellerKnoten1'	'Spule_L'
3	false	'f(e)'	false	true	'seriellerKnoten1'	'Widerstand_R'
4	false	'f(e)'	true	false	'Quellspannung'	'seriellerKnoten'
5	false	'f(e)'	false	false	'seriellerKnoten'	'seriellerKnoten1'

Tabelle 3.5.: Bondliste der Beispielsimulation

Die Eigenschaftsfelder *Source* und *Destination* eines Bond in der Bondliste entsprechen jeweils dem Eintrag des Ursprungs- und des Zielelements auf der Elementeliste. Die in der Tabelle 3.6 hierfür exemplarisch dargestellten Werte des Widerstands R sind der als Beispiel dienenden Simulation des RLC-Reihenschwingkreises entnommen.

Eigenschaft	Source	Destination
.ID	5	3
.Handle	9.0010	7.0010
.Typ	'Serieller Knoten'	'Resistor'
.Name	'seriellerKnoten1'	'Widerstand_R'
.Wert	1	100

Tabelle 3.6.: *Source*- und *Destination*-Feld von R

Die Funktion *registrieren_Bond()* [4] verwendet die Handles der an den Eingangs- und Ausgangsports angeschlossenen Elemente aus dem RTO des aufrufenden Bondelementes. In Abhängigkeit der Anzahl dieser Eingangs- und Ausgangshandles wird festgelegt, wie viele Elemente mit dem aufrufenden Element verbunden sind und wie viele Bonds dem entsprechend der Bondliste hinzugefügt werden müssen.
Die Eingangshandles werden an die Funktion *registrieren_Inputs()* [5] weitergegeben, welche die durch diese Handles identifizierten RTOs der Vorgängerelemente untersucht. Wird ein vorausgehendes Bondelement festgestellt, werden deren Eigenschaften in die *.Source*-Unterstruktur des Bonds eingetragen.
Sollte es sich beim Vorgängerelement nicht um ein Bondgraphenelement handeln, wird der betreffende Bond zwar der Bondliste hinzugefügt, aber durch Setzen des *Gateway*-Feldes auf *true* als Schnittstelle zu normalen Simulinkblöcken markiert. Der angeschlossene Simulinkblock wird hierbei ebenfalls auf der Elementeliste registriert.

[4] *Quellcode: siehe Anhang B.4.1 auf Seite 197*
[5] *Quellcode: siehe Anhang B.4.3 auf Seite 201*

Mit den Ausgangshandles einen Elementes wird in ähnlich verfahren, wobei die Funktion *registrieren_Outputs()* [6] analog verwendet wird. Es handelt sich bei dieser Funktion um eine reduzierte Version von *registrieren_Inputs()*, da lediglich überprüft wird, ob es sich bei dem nachfolgenden Element um ein normales Simulinkelement handelt. Sollte es sich beim angeschlossen Element am Ausgang um ein Bondelement der Toolbox handeln, erfolgt keine Aktion, da Bonds nur von deren Zielelementen registriert werden. Handelt es sich beim nachfolgenden Element um einen normalen Simulinkblock, müsste der betreffende Bond als ausgehende Schnittstelle des Graphen interpretiert werden, was in der aktuellen Version noch nicht implementiert ist.

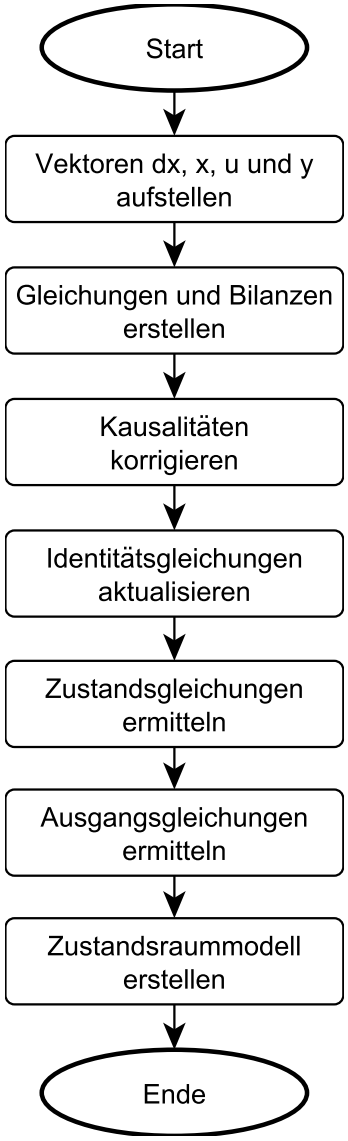


Abbildung 3.6.: Funktionsschema von analysieren()

Während der Registrierung der einzelnen Bonds wird deren Gesamtanzahl in der globalen Variable *registrierteBonds* erfasst, um im weiteren Verlauf die lückenlose Verarbeitung aller Bonds sicherzustellen.

Prinzipiell werden nur die Bonds an den Eingangsports eines Elementes regulär in die Bondliste eingetragen, wodurch sämtliche Bonds zwischen den Elementen des Bondgraphen erfasst werden. Externe Simulinkblöcke werden sowohl an den Ausgängen als auch an den Eingängen des Graphen gesucht, wobei aktuell lediglich die eingehenden Bonds verarbeitet und deren Effort- und / oder Flowvariablen in den Eingangsvektor $\vec{u}$ aufgenommen werden. In nachfolgenden Versionen der Toolbox könnten die Efforts und Flows dieser den regulären Graph verlassenden Bonds automatisch in den Ausgangsvektor $\vec{y}$ der Darstellung im Zustandsraum eingetragen werden, um den Übergang zwischen dem Bondgraph und anderen Simulinksystemen zu ermöglichen.

[6] *Quellcode: siehe Anhang B.4.4 auf Seite 205*

Nach der Erfassung aller Bonds für jedes Bondelement während der Durchführung der *doPostPropSetup()*-Methode wird im nächsten Schritt das *Start()*-Callback in den S-Functions der Elemente ausgeführt. In diesem Callback wird die Funktion *analysieren()*[7] aufgerufen, welche die eigentliche Hauptprozedur der Toolbox enthält. Die wichtigsten Schritte des Hauptprogramms zur Analyse des Graphen und Erstellung des zugehörigen Zustandsraummodells sind im Ablaufplan 3.6 auf der vorherigen Seite schematisch abgebildet.
Die Aufgaben der beschriebenen Callback-Funktionen sind im Folgenden abschließend zusammengefasst:

setup()
In dieser Callback-Funktion wird das RTO des Elementes erstellt und konfiguriert sowie das Bondelement mit Hilfe von *registrieren_Element()*[8] in die globale Elementeliste eingetragen.

doPostPropSetup()
Die eingehenden Bonds des Elements werden unter Verwendung von *registrieren_Bond()*[9] in die globale Bondliste eingefügt.

Start()
Die eigentliche Erstellung des Zustandsraummodells wird durch Aufruf der Hauptfunktion *analysieren()* ausgelöst.

Die Callbacks *InitializeConditions()* sowie *Output()* und *Update()* sind ebenfalls essentielle Bestandteile einer S-Function von Simulink und werden in dieser Version der Toolbox nicht verwendet. Diese könnten jedoch in späteren Versionen zur Umsetzung einer parallelen Simulation des Zustandsraummodells zur Ursprungssimulation genutzt werden, um eine Einbettung der Zustandsraumdarstellung in Simulink zu ermöglichen.

Die weiteren Einzelschritte zur vollständigen Analyse des modellierten Bondgraphen und dessen Übertragung in den Zustandsraum sind in den folgenden Abschnitten beschrieben. Es ist jedoch zu beachten, dass in dieser Arbeit keine detaillierte Erläuterung des Quellcodes erfolgt, sondern nur die Beschreibung der zugrunde liegenden Prinzipien und Abläufe sowie die Beleuchtung einiger ausgewählter Besonderheiten erfolgt.

[7] *Quellcode: siehe Anhang A.2.1 auf Seite 122*
[8] *Quellcode: siehe Anhang B.4.2 auf Seite 199*
[9] *Quellcode: siehe Anhang B.4.1 auf Seite 197*

3.4. Vektoren $\dot{\vec{x}}$, $\vec{x}$, $\vec{y}$ und $\vec{u}$

Die Erstellung der benötigten Vektoren $\dot{\vec{x}}$, $\vec{x}$, $\vec{y}$ und $\vec{u}$ ist der erste essentielle Schritt zur Erstellung der Zustandsraumdarstellung des Graphen und wird während des Aufrufs der Steuerfunktion *analysieren()*[10] ausgelöst. Die Generierung der einzelnen Vektoren ist auf verschiedene Unterfunktionen verteilt und wird nicht unmittelbar in *analysieren()* durchgeführt, wodurch eine höhere Modularisierung der Toolbox erreicht und die Aufgabe von *analysieren()* auf die Steuerung des Ermittlungs- und Aufbereitungsprozess reduziert werden soll.
Die verwendeten Unterfunktionen für die Erstellung der genannten Vektoren sind in Tabelle 3.7 gelistet, während deren Funktionsweisen in den nachfolgenden Unterabschnitten 3.4.1 und 3.4.2 erläutert werden.

Vektor	Symbol	Variable	Funktion
Zustandsvektor	$\vec{x}$	x	*erstelle_Zustaende()*
Zustandsableitung	$\dot{\vec{x}}$	dx	*erstelle_Zustaende()*
Eingangsvektor	$\overrightarrow{u}$	u	*erstelle_Ausgangsvektor()*
Ausgangsvektor	$\vec{y}$	y	*erstelle_Eingangsvektor()*

Tabelle 3.7.: Vektoren $\dot{\vec{x}}$, $\vec{x}$, $\vec{y}$ sowie $\vec{u}$ und deren Erstellungsfunktionen

3.4.1. Zustandsvektoren $\dot{\vec{x}}$ und $\vec{x}$

Der Zustandsvektor $\vec{x}$ besteht aus Zustandsvariablen der energiespeichernden Elemente des Systems, welche bereits in Abschnitt 2.4.1.2 sowie 2.4.1.3 beschrieben wurden.
Um den Zustandsvektor $\vec{x}$ und den zugehörigen Ableitungsvektor $\dot{\vec{x}}$ eines Bondgraphen festzulegen, müssen Effort- beziehungsweise Flowvariablen der enthaltenen kapazitiven Elemente C und / oder inerten Elemente I des Graphen ermittelt werden. Hierfür wird die Unterfunktion *erstelle_Zustaende()*[11] verwendet, welche die Bondliste nach den genannten Elementtypen durchsucht und die Zustandsvariablen der relevanten Elemente in den Vektor $\vec{x}$ sowie die zugehörigen Zustandsableitungen in den Vektor $\dot{\vec{x}}$ einfügt.
Da es sich bei energiespeichernden Elementen ausschließlich um Senken handelt, werden hierfür lediglich die Endpunkte der Bonds in der Bondliste untersucht und überprüft, ob sich bei dem jeweiligen Zielelement um den Typ 'Capacitor' oder 'Inductor' handelt.

[10] *Quellcode: siehe Anhang A.2.1 auf Seite 122*
[11] *Quellcode: siehe Anhang B.1.3 auf Seite 177*

Die hierfür benötigten Informationen werden direkt dem 3.2 auf Seite 54 entsprechenden *.Destination*-Feld der Bondliste entnommen.
Wird ein entsprechendes Element gefunden, wird der Zustandsvektor $\vec{x}$ um diese Zustandsvariable und der Ableitungsvektor $\dot{\vec{x}}$ um die Ableitung dieser Zustandsvariable des betreffenden Elementes gemäß Übersicht 3.8 erweitert.

	allgemeines Symbol		Toolboxvariable	
Typ	Zustand	Ableitung	Zustand	Ableitung
'Capacitor'	q_{BondID}	$\dot{q}_{BondID}$	q*[BondID]*	dq*[BondID]*
'Inductor'	p_{BondID}	$\dot{p}_{BondID}$	p*[BondID]*	dp*[BondID]*

Tabelle 3.8.: Zustände und Zustandsableitung von C und I

Im Fall der Beispielsimulation des RLC-Reihenschwingkreis sind die einzigen enthaltenen Speicherelemente der "Kondensator_C" und die "Spule_L". Die resultierenden generalisierten Zustandsvariablen und Bondvariablen dieser beiden Speicherelemente sind in Tabelle 3.9 gelistet.
Der am Kondensator eingehende Bond besitzt hierbei die BondID 1, wohingegen der eingehende Bond der Spule über die Nummer 2 verfügt.

Element	Effort	Flow
'Kondensator_C'	e1	dq1
'Spule_L'	dp2	f2

Tabelle 3.9.: Variablen von C und L

Die allgemeinen Toolboxvariablen *dq1* und *dp2* für den Kondensatorstrom sowie der Spulenspannung werden direkt in den Ableitungsvektor $\dot{\vec{x}}$ eingetragen, wohin gegen die Integrale dieser Variablen die eigentlichen Zustandsvariablen im Vektor $\vec{x}$ bilden. Hierdurch ergeben sich die in Tabelle 3.10 auf der nächsten Seite hinterlegten Vektoren, wobei zu beachten ist, dass in der mittleren Spalte die mathematische Vektorschreibweise und in der rechten Spalte die Toolbox- beziehungsweise Matlabdarstellung der Vektoren verwendet werden.
Es ist zu beachten, dass die Toolbox ebenfalls die Berechnung von Bondgraphen ohne energiespeichernde Elemente ermöglicht, welche demnach keine kapazitiven oder inerten sondern ausschließlich resistive Senkenelemente enthalten.
Hierdurch besitzt das resultierende Zustandsraummodell des Graphen keine Zustandsvariablen und es entfallen die Zustandsgleichungen $\dot{\vec{x}} = f(\underline{A}, \underline{B}, \vec{x}, \vec{u})$ sowie die Matrizen $\underline{A}$, $\underline{B}$ und $\underline{C}$ als auch die Vektoren $\dot{\vec{x}}$ und $\vec{x}$. Das Zustandsraummodell besteht somit

Vektor	Allgemein	Toolboxdarstellung
Zustandsableitungen	$\dot{\vec{x}} = \begin{pmatrix} \dot{q_1} \\ \dot{p_2} \end{pmatrix}$	dx = [dq1; dp2]
Zustandsvariablen	$\vec{x} = \begin{pmatrix} q_1 \\ p_2 \end{pmatrix}$	x = [q1; p2]

Tabelle 3.10.: Vektoren $\dot{\vec{x}}$ und $\vec{x}$

nur noch aus den vom Eingangsvektor $\vec{u}$ und von der Koeffizientenmatrix $\underline{D}$ abhängigen Ausgangsgleichungen für den Vektor $\vec{y}$.

3.4.2. Aus- und Eingangsvektor $\vec{y}$ und $\vec{u}$

Die Ermittlung des Aus- und Eingangsvektors erfolgt ähnlich der bereits im vorherigen Abschnitt 3.4.1 beschriebenen Erstellung des Zustandsvektors und dessen Ableitung mit Hilfe der Angaben aus der zuvor erstellten Bondliste des Graphen beziehungsweise des Systems. Die Erstellung der Vektoren $\vec{y}$ und $\vec{u}$ geschieht unter Verwendung der in Tabelle 3.7 auf Seite 60 bereits aufgeführten Unterfunktionen.
Ist ein Bond in der Bondliste als Ausgang markiert, werden sowohl die Effortvariable als auch die Flowvariable des Bonds in den Ausgangsvektor $\vec{y}$ eingefügt. In der aktuellen Version der Toolbox ist die Auswahl einzelner Bondvariablen nicht implementiert, wodurch die Effort- und Flowvariable des jeweiligen Bond immer paarweise verwendet werden müssen.
Im Beispiel wurde in der GUI des ohmschen Widerstandes R dieser als Ausgang gesetzt, wodurch der Effort und Flow Variablen des eingehenden Bonds in den Ausgangsvektor aufgenommen werden. Der Ausgangsvektor ergibt sich in mathematischer Schreibweise demnach, wie folgt:

$$\vec{y} = \begin{pmatrix} e_1 \\ f_1 \end{pmatrix} \tag{3.1}$$

Analog wird bei der Erstellung des Eingangsvektor in der Bondliste nach als *Gateway* markierten Bonds gesucht. Derartige Bonds repräsentieren Quellen des Graphen beziehungsweise Eingangsvariablen des Vektors $\vec{u}$ der Zustandsraumdarstellung, wobei die Quellvariable entsprechend der vorliegenden Bondkausalität gewählt wird, was in Abschnitt 2.4.2 genauer beschrieben ist.
Darüber hinaus werden die Eingangsvariablen beziehungsweise Quellvariablen zur Unterscheidung mit Großbuchstaben bezeichnet. Die ursprüngliche Variable und die re-

sultierende für das Zustandsraummodell verwendete Quellvariable der Effort-Quelle des RLC-Beispielgraphen sind in der Übersicht 3.11 aufgeführt. Die Quellvariablen aller Gateway-Bonds werden in dem Eingangsvektor $\vec{u}$ eingefügt, da im Beispiel nur eine Quelle vorliegt. Es existiert daher nur die Variable E_4 im Vektor $\vec{u}$.

Name	Bond	Kausalität	Variable	Quellvariable
'Quellspannung'	4	'f(e)'	e4	E4

Tabelle 3.11.: Variablenbezeichnung der "Quellspannung"

Im weiteren Verlauf der Ermittlung des Zustandsgleichungen und der dabei vorkommenden Substitutionen wird angenommen, dass sich die es sich bei den Elementen des Vektor $\vec{u}$ um bekannte Variablen handelt, da diese durch den Anwender zur Simulation des resultierenden Zustandsraummodells definiert werden müssen.
Es ist zu beachten, dass jeder Bondgraph mindestens über eine Flow- oder Effortquelle verfügen muss, was in der Toolbox durch Erzeugung von Fehlern und Programmabbruch im Fall von nicht verbundenen Eingangsports von Bondelementen erzwungen wird.

3.5. Knotenliste

Im Anschluss an die Erstellung der Vektoren $\vec{y}$, $\vec{u}$ und des Zustandsvektors $\vec{x}$ sowie dessen Ableitung $\dot{\vec{x}}$ erfolgt die Vervollständigung der Knotenliste, wobei die noch fehlenden Informationen für die bisher inhaltslosen Felder *.Gleichnisse, .Inputs* und *.Outputs* ermittelt und ergänzt werden. Die Knotenliste wurde zuvor wie in Abschnitt 3.3 auf Seite 53 erstellt. Die Feldbezeichnungen sowie zugehörige Datentypen der globalen Knotenliste können 3.4 auf Seite 55 entnommen werden. Diese Vervollständigung der Knotenliste ist in die Zusatzfunktion *erstelle_Knotengleichungen()*[12] ausgelagert und wird durch deren Aufruf in der Steuerfunktion *analysieren()*[13] ausgelöst.
Hierfür wird in der globalen Bondliste das Anfangs- sowie Zielelement jedes einzelnen Bonds untersucht, ob es sich bei diesem jeweils um einen Knoten handelt. Wurde an der Quelle und / oder Senke eines Bonds ein auf der Knotenliste vermerkter Knoten gefunden, wird der genaue Typ des Knotens für die weitere Verarbeitung ausgewertet. Das in den nachfolgenden Absätzen beschriebene Bearbeitungs- und Entscheidungssche-

[12] *Quellcode: siehe Anhang B.2.3 auf Seite 189*
[13] *Quellcode: siehe Anhang A.2.1 auf Seite 122*

ma innerhalb der Funktion *erstelle_Knotengleichungen()* ist in Abbildung 3.7 auf der nächsten Seite begleitend dargestellt.
Ist an der Senke des jeweils aktuellen verarbeiteten Bond ein serieller 1-Knoten angeschlossen, wird der Flow des Bond in die *Gleichnis*-Zelle des angeschlossenen Knotens auf der globalen Knotenliste eingefügt und die Effortvariable in die zugehörige Input-Zelle geschrieben. Befindet sich der Knoten an der Quelle des Bond, wird der Effort in die Output-Zelle des Knotens geschrieben.
Die Vorgehensweise ist bei parallelen 0-Knoten analog. Die Rollen von Effort und Flow sind hierbei jedoch vertauscht, wodurch der Effort des jeweiligen Bonds in das Feld der Identitätsgleichung und der Flow in die Input- beziehungsweise Output-Zelle des jeweiligen parallelen Knotens auf der Knotenliste geschrieben werden.
Die hierdurch ergänzte Knotenliste der Beispielsimulation ist in Tabelle 3.12 dargestellt:

Eigenschaft	1. Knoten	2. Knoten
ID	4	5
Typ	{ 'Serieller Knoten' }	{ 'Serieller Knoten' }
Gleichnisse	{ 'dq1' '0' 'f5' }	{ 'f2' 'f3' 'f5' }
Inputs	{ 'e4' }	{ 'e5' }
Outputs	{ 'e1' 'e5' }	{ 'dp2' 'e3' }

Tabelle 3.12.: vollständige Knotenliste des Beispielgraph

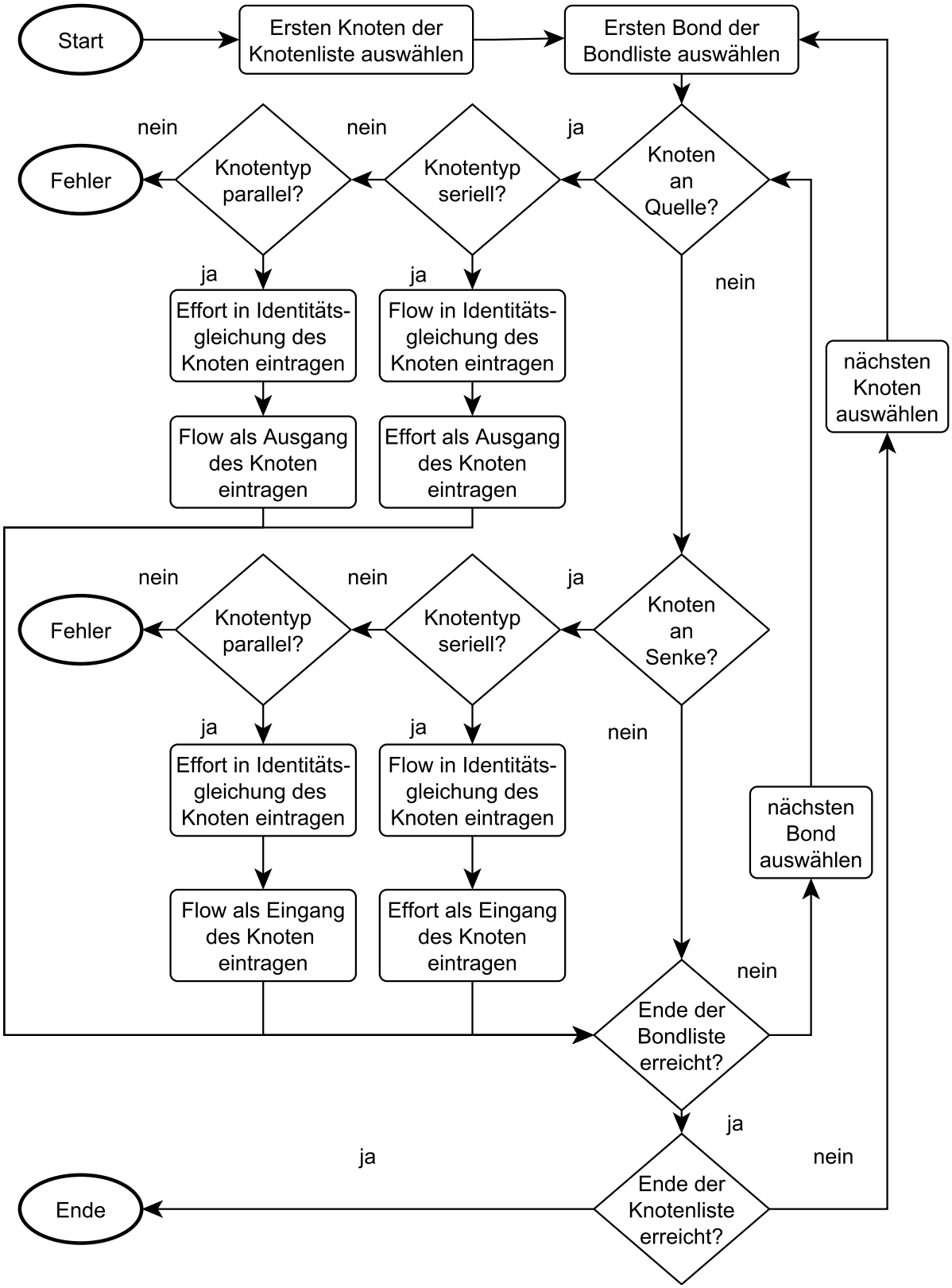


Abbildung 3.7.: Entscheidungsschema für Knotenliste

3.6. Grundgleichungen und Listen

Die Generierung der den Elementen und Knoten zugehörigen Gleichungen ist der nächste essentielle Schritt zur Ermittlung der Zustandsraumdarstellung des Bondgraphen und wird nach der Erstellung der globalen Bondliste sowie der Vervollständigung der ebenfalls globalen Knotenliste durchgeführt. Das Vorhandensein und die Korrektheit dieser beiden Listen ist Voraussetzung für die erfolgreiche Ermittlung der Zustands- und Ausgangsgleichungen des jeweiligen Systems sowie die Basis für die Grundgleichungen der enthaltenen Bondelemente und Knoten.
Mit Ausnahme der Ausgangsbonds von Quellen wird für jeden Bond mit Hilfe der Unterfunktion *erstelle_Bondgleichungen()*[14] eine Gleichung erstellt. Die Bonds von Quellen erfordern keine Gleichungen, da zuvor die zugehörigen Quellvariablen, wie bereits in Abschnitt 3.4.2 beschrieben, in den Eingangsvektor $\vec{y}$ der Zustandsraumdarstellung eingetragen wurden und die zugehörigen komplementäre Flow- oder Effortvariablen, welche nicht unmittelbar durch die Quelle definiert sind, nicht verwendet werden.
Die Identifikationsnummer des aktuellen Bonds wird hierfür an die Funktion *erstelle_Bondgleichungen()* übergeben. Diese wertet den Typ des zugehörigen Zielelementes aus, stellt die entsprechende Gleichung oder Gleichungen auf und trägt diese in die globale *Ergebnisliste* oder globale *Grundliste* ein.
Insgesamt werden in dieser Toolbox drei verschiedene Listen zur Speicherung und Verarbeitung der Gleichungen verwendet, wobei es sich bei der dritten Liste um die *Arbeitsliste* zur Verwaltung der Zwischenlösungen handelt. Die Arbeitsliste wird ausschließlich in einem laufenden Ermittlungsprozess verwendet und wird zusammen mit dem zugrunde liegenden Lösungsalgorithmus der Toolbox in Abschnitt 3 genauer erläutert. Alle drei Listen besitzen trotz unterschiedlicher Funktionalität einen identischen Aufbau, wobei je eine Listenzeile den Datensatz für jeweils eine Gleichung enthält und die Spalten der Liste den Eigenschaften der Gleichung entspricht. Die Eigenschaften der einzelnen Spalten, aus welchen sich der Datensatz für die Beschreibung einer Gleichung zusammensetzt, sind im Folgenden gelistet:

1. Spalte: Gleichungsnummer

Die Gleichungsnummer dient zur eindeutigen Identifikation der Gleichungen und wird unter anderem zur Vermeidung von algebraischen Schleifen verwendet, was durch den Abgleich der Identifikationsnummer der in den jeweiligen Lösungsprozess bereits einbezogenen Gleichung mit aktuell zu verwendenden Gleichung realisiert ist.

[14] *Quellcode: siehe Anhang B.2.2 auf Seite 184*

2. Spalte: explizite Variable

Gleichungen werden grundsätzlich nach einer einzelnen Variable umgestellt gespeichert, wodurch diese Variable durch die Gleichung explizit beschrieben wird. Die zweite Spalte enthält diese explizite Variable als symbolische Matlabvariable vom Datentyp *sym*.

3. Spalte: Gleichungsrumpf

Diese Spalte beinhaltet die restlichen Gleichungsterme zur Beschreibung der expliziten Variable auf der gegenüberliegenden Seite des Gleichheitszeichens. Der Gleichungsrumpf ist ebenfalls als *sym* gespeichert.

4. Spalte: Bemerkung

Die Bemerkung liefert zusätzliche Informationen zur Gleichung und gehört dem Datentyp *char* an. Der Inhalt der Bemerkung unterscheidet sich in Abhängigkeit der Liste:

- Grundliste: Enthält den Typ des zugehörigen Bondelementes oder die Angabe, ob die Gleichung eine Bilanz oder der Teilsegment einer Identitätsgleichung ist.
- Arbeitsliste: Enthält Informationen über die Entstehung der jeweiligen Gleichung während der Ermittlung, z.B. ob diese Gleichung hierfür umgestellt wurde. Sollte kein besonderer Schritt zur Entstehung der Gleichung beigetragen haben, ist die ursprüngliche Angabe der Grundliste enthalten.
- Ergebnisliste: Gibt die Art der erfolgreichen Ermittlung der jeweiligen Lösungsgleichung an, z.B. ob diese mit Hilfe des regulären Substitutionsalgorithmus ermittelt wurde, direkt aus einer Identitätsgleichung resultierte oder es sich um eine rekursive Lösung handelt.

Die Einordnung der Gleichungen in die genannten Listen erfolgt in Abhängigkeit von der in den jeweiligen Gleichungen enthaltenen Variablen. Wie bereits in Abschnitt 2.5.3 beschrieben, dürfen die endgültigen Zustands- und Ausgangsgleichungen der Zustandsraumdarstellung nur von Koeffizienten der Matrizen $\underline{A}$ und $\underline{B}$ sowie den Variablen des Eingangsvektors $\vec{u}$ und des Zustandsvektors $\vec{x}$ abhängen.
Hieraus folgt, dass Gleichungen von energiespeichernden Elementen, welche nur von Konstanten und deren elementspezifischen Zustandsvariablen abhängig sind, im weiteren Ermittlungsprozess als Lösung angesehen und direkt in die Ergebnisliste eingefügt werden. Im Gegensatz dazu werden sämtliche Gleichungen in die Grundliste eingetragen, welche nicht ausschließlich von Zustandsvariablen abhängen und somit als unbekannte Gleichung beziehungsweise ungelöste Gleichung aufgefasst werden.

Für die Endpunkte des Graphen werden die in Tabelle 3.13 angegebenen Gleichungen in die jeweiligen Listen eingefügt, wobei die Auswahl anhand der Kausalität des eingehenden Bonds getroffen wird. Es ist zu beachten, dass die Gleichungen des resistiven Verbrauchers immer auf der Grundliste eingefügt werden müssen, da diese nur von einem normalen Effort oder Flow abhängen, welcher zu diesem Zeitpunkt der Ermittlung nicht bekannt sein kann.

	Gleichung	
Element	Kausalität e(f)	Kausalität f(e)
R	$e = R * f$	$f = \frac{1}{R} * e$
C	$e = \frac{1}{C} * q$	$unzulässig$
I	$unzulässig$	$f = \frac{1}{L} * p$

Tabelle 3.13.: Bondgleichungen für Endpunkte

Für die Bonds an kapazitiven Elementen C und inerten Elementen I existiert jeweils nur eine von einer Zustandsvariable abhängige Gleichung, wodurch für den jeweils angeschlossen Bond nur eine Kausalität zulässig ist. Hierdurch müssen für diese Energiespeicher immer die Gleichungen in Integralform gemäß Abschnitt 2.4.1.2 und 2.4.1.3 besitzen. Die Kausalität der Bonds wird hierbei als *integrative Kausalität* beziehungsweise *Integralkausalität* bezeichnet, da aus dieser die Integralform der Speichergleichungen resultiert.
Ist einem Eingangsbond dieser Elemente die differentiellen Kausalität zugeordnet, wird bei Programmausführung ein Fehlermeldung generiert. Während der weiteren Ausführungsebenen der Toolbox können bei gegebenenfalls vorliegenden Kausalitätskonflikten die betroffenen Gleichungen jedoch gezielt von der Integralform in deren differentielle Form umgeformt werden. Die Verarbeitung von differentiellen Kausalitäten ist in den Vorstufen der Toolbox bereits implementiert. Das Vorgehen in der Vorstufe bei Kausalitätskonflikten und mehrdeutigen Lösungen ist in Abschnitt 3.7.2 auf Seite 73 erläutert.

Für Endpunkte wird jeweils nur eine Gleichung in die Grund - oder Ergebnisliste eingefügt, da diese Elemente jeweils nur einen Bond besitzen. Im Gegensatz hierzu besitzen Übertrager wie Transformer und Gyrator je einen Eingangs- sowie einen Ausgangsbond und somit zwei Gleichungen, die den Zusammenhang zwischen Variablen des Ein- und Ausgangsbond beschreiben. Für den Transformer sowie den Gyrator werden die in Abschnitt 2.4.4 beschriebenen Gleichungen zur Beschreibung des Zusammenhanges zwischen Eingangs- und Ausgangsbond verwendet.

In der aktuellen Version werden die Gleichungen der Umformer nicht von den Kausalitäten der anliegenden Bonds beeinflusst, aufgrund dessen die Gleichungen der Umformer immer in der Grundform vorliegen. Für den Transformator TF werden die Gleichungen 2.34 sowie 2.35 und für den Gyrator GY die Gleichungen 2.36 und 2.37 verwendet.

	Transformer		Gyrator	
	Eingang	Ausgang	Eingang	Ausgang
Gleichung	$e_{in} = m * e_{out}$	$f_{out} = m * f_{in}$	$e_{in} = r * f_{out}$	$e_{out} = r * f_{in}$
explizite Variable	e_{in}	f_{out}	e_{in}	e_{out}
Gleichungsrumpf	$m * e_{out}$	$m * f_{in}$	$r * f_{out}$	$r * f_{in}$

Tabelle 3.14.: Bondgleichungen für Übertrager

Die Gleichungen von Gyratoren und Transformern werden immer auf der Grundliste eingefügt, da diese in ihrer Grundform von keiner Zustandsvariable abhängen und somit als unbekannt interpretiert werden. Zusätzlich ist zu beachten, dass die Kausalität des eingehenden und ausgehenden Bonds keinen Einfluss auf die Abbildung der Eingangs- und Ausgangsgleichung dieser beiden Elemente in der Grundliste hat.

Im Fall von parallelen Knoten werden lediglich die zugehörigen Flowbilanzen vermerkt, wohingegen im Fall von seriellen 1-Knoten die Effortbilanzen der Knoten verwendet werden. Bilanzen von Knoten werden grundsätzlich ungelöst angenommen und demnach ausschließlich auf der Grundliste eingetragen.
Die Identitätsgleichungen der beiden Knotentypen werden nicht weiterverarbeitet, sondern im späteren Programmverlauf bei Bedarf direkt von den entsprechenden Feldern der Knotenliste bezogen. Da in der vorliegender Version der Toolbox Knoten lediglich einen eingehenden Bond aufweisen können, wird dessen knotentypabhängige Bilanzvariable als explizite Variable in der zweiten Spalte in der Grundliste verwendet, wobei die restlichen Bilanzvariablen der ausgehenden Bonds in der dritten Spalte der Grundliste den Gleichungsrumpf bilden. Die allgemeinen Erscheinungsformen der Bilanzen in den Listen der Toolbox können der folgenden Tabelle 3.15 entnommen werden:

	Serieller Knoten	Paralleler Knoten
Gleichung	$\sum_{i=1}^{N} e_{in_i} = \sum_{j=1}^{M} e_{out_j}$	$\sum_{i=1}^{N} f_{in_i} = \sum_{j=1}^{M} f_{out_j}$
explizite Variable	e_{in_1}	f_{in_1}
Gleichungsrumpf	$\sum_{j=1}^{M} e_{out_j} - \sum_{i=2}^{N} e_{in_i}$	$\sum_{j=1}^{M} f_{out_j} - \sum_{i=2}^{N} f_{in_i}$

Tabelle 3.15.: Bilanzen von Knoten

Um in späteren Versionen der Toolbox mehrere Eingangsvariablen einen Knotens mit dem Darstellungsprinzip der Grundliste vereinbaren zu können, müssen sämtliche eingehenden Bilanzvariablen des jeweiligen Knotens auf die Ausgangsseite der zugehörigen Bilanzgleichung subtrahiert werden, so dass sich die Bilanz zu null ergibt. Die eingehende Bilanzvariable mit dem niedrigsten Index beziehungsweise der niedrigsten Bondnummer ist hiervon jedoch ausgenommen und verbleibt als explizite Variable auf der ehemaligen Eingangsseite der Bilanz. Dieser Vorgang scheint mit den aktuellen Aufbereitungs- und Ermittlungsverfahren der Toolbox kompatibel und könnte mit geringem Aufwand implementiert werden.

Da für Knoten keine Gleichungen existieren, welche das Verhältnis zwischen den angeschlossenen Efforts und Flows definieren, ist es nicht notwendig die Kausalität der anliegenden Bonds in die Aufbereitung zu berücksichtigen. Hierdurch werden die Bilanzen und Identitätsgleichungen der Knoten nur in einer Erscheinungsform auf der Grundliste hinterlegt, wobei die Kausalität der ausgehenden Bonds des jeweiligen Knotens ausschließlich durch die Gleichungen der angeschlossenen Endpunkte bestimmt wird.

Die Initialinhalte der Grundliste und der Ergebnisliste des als Beispiel dienenden Schwingkreises sind in Tabelle 3.16 und 3.17 dargestellt. Es ist zu beachten, dass es sich bei den aus Matlab entnommenen Inhalten dieser beiden Listen um die Initialwerte des Beispielgraphen direkt vor Beginn des Lösungsprozessen handelt.

Nummer	Explizite Variable	Gleichungsrumpf	Bemerkung
4	f3	e3/R3	'Resistor'
5	E4	e1 + e5	'Bilanz'
6	e5	dp2 + e3	'Bilanz'

Tabelle 3.16.: Grundliste mit Initialwerten

Nummer	Explizite Variable	Gleichungsrumpf	Bemerkung
1	E4	E4	'Eingang'
2	e1	q1/C1	'bekannt'
3	f2	p2/L2	'bekannt'

Tabelle 3.17.: Ergebnisliste mit Initialwerten

Im Fall der Gleichung 1 auf Ergebnisliste 3.17 handelt es sich um eine Pseudogleichung für die Quellspannung U mit dem Effort E_4. Die Quellvariable E_4 wurde hierfür dem Eingangsvektor $\vec{u}$ entnommen.

Quellvariablen werden in der Toolbox generell als Pseudogleichungen vermerkt, wobei die Quellvariable im Datensatz sowohl die explizite Variable in der zweiten Spalte als auch den eigentlichen Gleichungsrumpf in der dritten Spalte ist. Hierdurch werden Quellvariablen direkt als Lösung in den zu ermittelnden Gleichungen erkannt, wodurch keine Sonderbehandlung von bekannten Quellvariablen gegenüber anderen bekannten Lösungen während des Ermittlungsprozesses erforderlich ist.

Des Weiteren wurde bei der dargestellten Grund- und Ergebnisliste bereits eine Ersetzung der Variablenbezeichnungen für die allgemeinen Effort- und Flowvariablen vorgenommen, wie sie der eigentlichen Ermittlung vorgelagert und in Abschnitt 3.7.1 beschrieben ist.
Im Anschluss an die initiale Erstellung der globalen Grund- und der globalen Ergebnisliste durch die Koordinierungsfunktion *analysieren()*[15] wird die Vorstufenfunktion *ersetze_Bezeichnungen()*[16] von dieser aufgerufen, wodurch die zuvor erwähnte Ersetzung der allgemeinen Variablenbezeichnungen vorgenommen wird.

3.7. Vorstufen

Zusätzlich zu der in den Abschnitten 3.2 bis 3.6 beschrieben Erstellung des Bondgraphen sowie der auf diesem basierenden Gleichungen und Listen, müssen vor dem Beginn des eigentlichen Lösungsprozesses die Ergebnisliste sowie die Identitätsgleichungen der Knoten auf das Vorliegen von Eindeutigkeitskonflikten untersucht und diese gegebenenfalls korrigiert werden. Des Weiteren werden die Bezeichnungen der Bondvariablen auf der Grund- und Ergebnisliste sowie der Knotenliste angepasst.

Da diese beiden Vorgänge über die bloße Erstellung von Gleichungen hinausgehen und nachträgliche Modifikationen darstellen, werden diese als Vorstufen bezeichnet, welche in den Abschnitten 3.7.1 und 3.7.2 beschrieben werden.

3.7.1. Variablenbezeichnungen

Während der initialen Generierung der Grund- und Ergebnisliste sowie der Knotenliste werden zu diesem Ausführungszeitpunkt die generalisierten Bezeichnungen e_i für den Effort und f_i für den Flow eines Bond verwendet. Bei der grundlegenden Erstellung wird nicht berücksichtigt, ob es sich bei den jeweiligen Bondvariablen um eine Quellvariable

[15] *Quellcode: siehe Anhang A.2.1 auf Seite 122*
[16] *Quellcode: siehe Anhang A.1.1 auf Seite 113*

des Vektor $\vec{u}$, eine Zustandsvariable des Vektor $\dot{\vec{x}}$ oder deren Ableitung aus $\dot{\vec{x}}$ handelt, wodurch diese von den allgemeinen Variablenbezeichnungen abweichen würden.

Tabelle 3.18 gibt die ursprüngliche Grundliste der Beispielsimulation direkt nach deren Erstellung ohne modifizierte Bezeichner wieder. Gemäß der zugehörigen Elementeliste (Tabelle 3.3 auf Seite 55) handelt es sich beim Effort e2 um die Ableitung $\dot{p}_2$ des allgemeinen Impuls p_2 der Spule und somit um eine Zustandsableitung, welche eigentlich die Variablenbezeichnung dp2 besitzen muss.
Die Spannungsquelle besitzt gemäß der zugrundeliegenden Bondliste (Tabelle 3.5 auf Seite 57) den Ausgangsbond 4 als Verbindung zum seriellen Knoten mit der Elementnummer 4. Hierdurch handelt es sich eigentlich beim Effort e4 um die Quellspannung U beziehungsweise um den Quelleffort E4, wodurch e4 ebenfalls umbenannt werden muss.

Nummer	Explizite Variable	Gleichungsrumpf	Bemerkung
4	f3	e3/R3	'Resistor'
5	e4	e1 + e5	'Bilanz'
6	e5	e2 + e3	'Bilanz'

Tabelle 3.18.: Initiale Grundliste

Zur Anpassung der Bezeichner der generalisierten Bondvariablen an die Ein- und Ausgangsbonds der Elemente, wird in der Steuerfunktion *analysieren()*[17] die zu diesem Zweck erstellte Hilfsfunktion *ersetze_Bezeichnungen()*[18] aufgerufen. Diese Unterfunktion durchsucht alle Listen nach Effort- und Flowvariablen, welche eigentlich Bestandteile der Vektoren $\vec{u}$ und $\vec{x}$ sowie $\dot{\vec{x}}$ repräsentieren sollten und ersetzt deren Bezeichner.
Die den Elementen zugehörigen Ersatzbezeichnungen der zu modifizierenden Bezeichner sind in Übersicht 3.19 gelistet.

Bondelement	allgemeine Bezeichung	spezifische Bezeichnung
Flowquelle S_F	f_i	F_i
Effortquelle S_E	e_i	E_i
Capacitor C	f_i	$\dot{q}_i$ beziehungsweise dq_i
Inertia I	e_i	$\dot{p}_i$ beziehungsweise dp_i

Tabelle 3.19.: Spezifische Variablenbezeichner

[17] *Quellcode: siehe Anhang A.2.1 auf Seite 122*
[18] *Quellcode: siehe Anhang A.1.1 auf Seite 113*

3.7.2. Kausalitätskonflikte

In normalen Bondgraphen und mit Hilfe der Toolbox erstellten Graphen wird üblicherweise integrative Kausalität für energiespeichernde Elemente verwendet, da in diesem Fall deren Grundgleichungen ausschließlich von den Zustandsvariablen der jeweiligen Speicher abhängen und somit direkt als Lösung während der Ermittlung der Zustandsraumdarstellung genutzt werden können.

Bei der manuellen Erstellung und Berechnung von Bondgraphen ohne den Einsatz der vorliegenden Simulinktoolbox oder anderer Hilfsmittel müssen die Kausalitäten der einzelnen Bonds vom Ersteller so gewählt werden, dass die Verkettung der einzelnen Elementgleichungen sowie die eindeutige Lösbarkeit der Gleichungen gegeben ist. Andernfalls ist eine Aufstellung der Zustandsraumdarstellung und die Berechnung des Systemverhaltens nicht möglich.
Sollte bei einem mit der Toolbox erstellten Bondgraphen uneindeutige Lösungen auftreten oder die vorliegenden Kausalitäten der Bonds keine vollständige Verkettung der Element- und Knotengleichungen ergeben, wird dies von der Toolbox automatisch korrigiert, wodurch die Festlegung der Kausalität und deren Korrektheit vom Benutzer vernachlässigt werden kann und dieser sich auf das reine Zusammenfügen der Elemente beschränkten könnte.

Die geforderte integrative Kausalität von Speicherelementen verursacht jedoch in den folgenden Konstellationen uneindeutige Lösungen:

- Zwei oder mehr kapazitive Speicher C an parallelen 0-Knoten
- Zwei oder mehr induktive Speicher I an seriellen 1-Knoten

In beiden Konstellationen sind die Gleichungen der Elemente bekannt und die explizit beschriebenen Bondvariablen dieser Gleichungen sind Bestandteil der Identitätsgleichung des zugehörigen Knotens.
Im Fall zweier induktiven Speicher an einem seriellen 1-Knoten sind die Flows bekannt, wodurch jedoch ein Lösungskonflikt in der Identitätsgleichung des Knotens entsteht, da beide Flows aufgrund eben dieser Gleichung identisch sein müssen, aber durch die Elemente unterschiedliche Lösungsgleichungen besitzen und von den unterschiedlichen Zustandsvariablen der beiden Inertia abhängen.
Der Konflikt weitet sich aus, je mehr induktive Speicher mit integrativer Kausalität an einem seriellen 1-Knoten angeschlossen sind, wodurch proportional zur Anzahl der inerten Elemente entsprechend mehr bekannte Flows durch den Knoten miteinander verknüpft

sind. Aufgrund der vorliegenden Identitätsgleichung des seriellen Knotens müssten dessen Flowvariablen identisch sein, besitzen aber dennoch unterschiedliche Lösungen.
Der Konflikt tritt analog mit kapazitiven Speicherelementen an einem parallelen Knoten auf, wobei die Efforts der Kapazitäten durch deren Gleichungen explizit unterschiedlich definiert sind und durch die Effortidentität des Knotens sich nicht in ihrer Lösung unterscheiden dürften.

Zur Beseitigung derartiger Lösungskonflikte wird in der Vorstufe die Funktion *korrigiere_Causality()*[19] aufgerufen. Diese dient als Steuerfunktion für die eigentliche Korrekturfunktion *korrigiere_Gleichnis()*[20], welche nacheinander für sämtliche vorhandene Identitätsgleichungen aufgerufen wird, diese auf Eindeutigkeit überprüft und gegebenenfalls korrigiert.

Alle betroffenen Elementgleichungen einer Identitätsgleichung mit uneindeutigen Lösungen, ausgenommen der Gleichung mit der niedrigsten Bondnummer, werden nach ihren jeweiligen Zustandsvariablen umgestellt werden, wodurch die Zustandsvariablen von den ehemaligen expliziten Effort- beziehungsweise Flowvariablen abhängen und die Gleichung in differentieller Form vorliegt. Die ursprünglichen Gleichungen werden hierbei von der Elementeliste entfernt.
Die normalen Bondvariablen, von denen die umgeformten Elementgleichungen und somit die Zustandsvariablen abhängen, müssen aufgrund der Identitätsgleichung des jeweiligen Knotens identisch sein und werden anschließend mit der Lösungsgleichung in Integralform der expliziten Variable der betreffenden Identitätsgleichung mit der niedrigsten ID substituiert, welche im vorherigen Schritt von der Umformung in ihre differentielle Form ausgeschlossen war. Hierdurch hängen die explizit beschriebenen Zustandsvariablen der umgeformten Gleichungen ausschließlich von der Zustandsvariable der unveränderten Gleichung mit der niedrigsten BondID ab, wobei diese Gleichung weiterhin als Lösung auf der Ergebnisliste vermerkt bleibt.

Anschließend werden die substituierten Gleichungen symbolisch nach der Zeit abgeleitet, wodurch jeweils eine Gleichung für die Zustandsableitung eines jeden beteiligten Speicherelementes entsteht. Die zeitliche Ableitung der Gleichungen erfolgt hierbei rein formell, durch Umbenennen der expliziten Ergebnisvariable und eingefügten Zustandsvariable des unveränderten Elements in deren jeweilige Zustandsableitungen, da die restlichen Bestandteile der Gleichungen zeitlich unveränderliche Elementparameter sein müssen.

[19] *Quellcode: siehe Anhang A.1.2 auf Seite 118*
[20] *Quellcode: siehe Anhang A.1.3 auf Seite 119*

Diese Gleichungen beschreiben somit die Zustandsänderungen der umgeformten Speichergleichungen in Abhängigkeit der Zustandsänderung des unveränderten Elements. Die aus diesem Vorgang resultierenden Gleichungen der Zustandsänderungen werden auf der Grundliste eingefügt und wie gewöhnliche unbekannte Gleichungen im Lösungsprozess verwendet, können aber während dieses Prozesses mit Hilfe des normalen Lösungsalgorithmus ermittelt werden, auf welchen in Abschnitt (3.9) eingegangen wird.

3.7.2.1. Beispiel: Zwei Inertia an seriellen 1-Knoten

Der Umgang mit den zuvor beschriebenen Konfliktsituationen wird im Folgenden anhand des Graphen 3.8 für zwei inerte Speicher an einem seriellen Knoten beispielhaft erläutert:

Für die beiden Speicher ergeben sich basierend auf der geforderten integrativen Kausalität der Bonds die folgenden bekannten Gleichungen für den Flow f_1 und f_2, welche diese explizit beschreiben und auf der Ergebnisliste vermerkt sind:

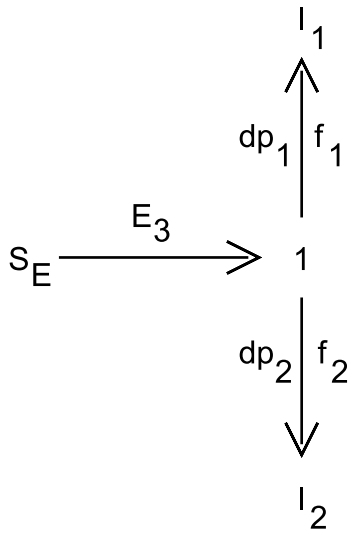


Abbildung 3.8.: Konfliktgraph

$$f_1 = \frac{1}{I_1} p_1 \tag{3.2}$$

$$f_2 = \frac{1}{I_2} p_2 \tag{3.3}$$

Des Weiteren liegt am 1-Knoten die folgende Flowidentität sowie die folgende Effortbilanz vor:

$$f_1 = f_2 \tag{3.4}$$

$$E_3 = \dot{p}_1 + \dot{p}_2 \tag{3.5}$$

Der Identitätsgleichung 3.4 folgend muss der Flow f_1 genau dem Flow f_2 entsprechen, welche offensichtlich nach den bekannten Gleichungen der Ergebnisliste 3.2 und 3.3 unterschiedlich sind.

Um den Konflikt zu lösen, wird Gleichung 3.3 nach der enthaltenen Zustandsvariable p_2 umgestellt und von der Ergebnisliste gestrichen, wodurch sich die folgende differentielle Form der ursprünglichen Elementegleichung als Gleichung 3.6 ergibt:

$$p_2 = I_2 * f_2 \tag{3.6}$$

Der enthaltene Flow f_2 wird Identität 3.4 folgend mit Gleichung 3.2 für f_1 substituiert, wodurch sich p_2 durch das Verhältnis der Konstanten I_2 zu I_1 und der Zustandsvariable p_1 beschreiben lässt:

$$p_2 = \frac{I_2}{I_1} p_1 \tag{3.7}$$

Da I_1 und I_2 zeitlich nicht veränderlich sind, lässt sich durch Ableiten der Gleichung 3.7 nach der Zeit t eine Beschreibung der Zustandsänderung $\dot{p}_2$ in Abhängigkeit von $\dot{p}_1$ finden:

$$\dot{p}_2 = \frac{dp_2}{dt} = \frac{d}{dt}\frac{I_2}{I_1} p_1 = \frac{I_2}{I_1}\frac{d}{dt} p_1 \tag{3.8}$$

$$= \frac{I_2}{I_1} \dot{p}_1 \tag{3.9}$$

Durch das Einfügen der resultierenden Gleichung 3.9 in die Grundliste und das vorherige Entfernen der ursprünglichen Elementegleichung 3.3 von der Ergebnisliste ist die Eindeutigkeit des Gleichungssatzes gegeben und somit der Konflikt behoben.
Die modifizierte Grund- und Ergebnisliste ist mit dem restlichen Lösungsmechanismus der Toolbox vollständig kompatibel und kann ohne weitere Ausnahmen verwendet werden.

Sollte ein analoger Konflikt mit mehr als zwei Speichern und somit mehr als zwei Gleichungen vorliegen, wird der gleiche Korrekturmechanismus angewendet, wobei sämtliche beteiligten Speichergleichungen von der derjenigen Konfliktgleichung mit der niedrigsten Bondnummer abhängig gemacht werden.

3.8. Aktualisierung

Da in den Identitätsgleichungen von Knoten fehlerfreien Fall alle enthaltenen Variablen identisch sind, sind alle Variablen automatisch bekannt, sobald dieser Variable eine Lösung besitzt.
Um diesen Umstand vorteilhaft zur Ermittlung des Zustandsraummodells zu nutzen, wird die Funktion *aktualisieren()*[21] der Toolbox verwendet, welche die jeweils vorliegenden Identitätsgleichungen auf das Vorhandensein von bekannten Lösungen hin durchsucht und diese für die unbekannten Variablen in die Ergebnisliste einfügt, wodurch diese formell gelöst werden und anschließend bekannt sind.

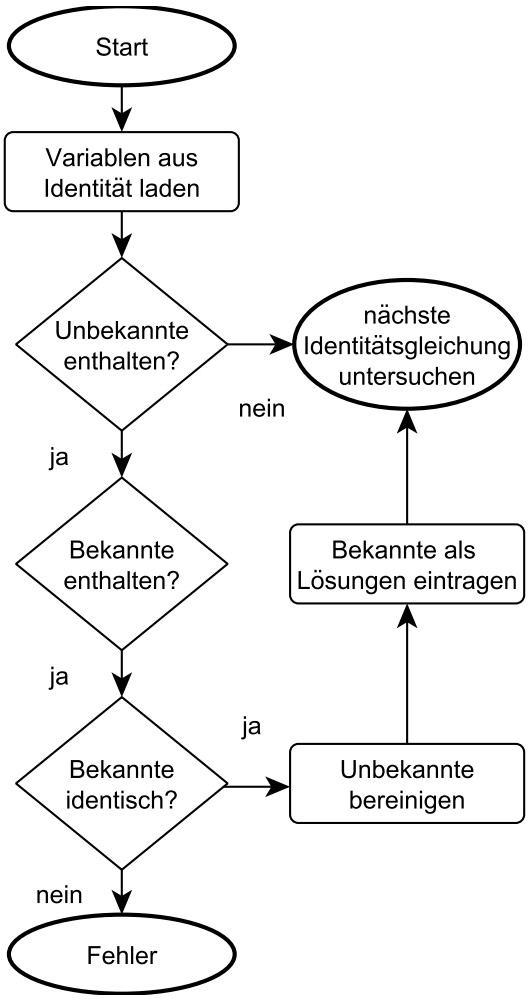


Abbildung 3.9.: Aktualisierungsschema

Während der Durchführung von *aktualisieren()* werden die einzelnen Identitätsgleichungen sämtlicher Knoten nacheinander nach bekannten Variablen durchsucht. Kommen in einem Gleichnis bekannte und unbekannte Variablen vor, können die Lösungen der bekannten Variablen für die Unbekannten verwendet werden, da diese durch die vorliegende Variablenidentität die selbe Lösung besitzen müssen.
Sind die betroffenen unbekannten Variablen im gegebenenfalls vorhandenen oder vorausgehenden Ermittlungsprozess bereits zur Lösungsermittlung in diesen eingebunden und somit für eine weitere Verwendung gesperrt, werden diese freigeben und können in den weiteren Lösungsschritten als Bekannte verwendet werden.

Wird während der Aktualisierung eine Änderung an einer Gleichungsliste vorgenommen und mindestens die Lösung einer Unbekannte in die Ergebnisliste eingetragen, wird im Anschluss an diese Aktualisierung der Aktualisierungsvorgang rekursiv erneut

[21] *Quellcode: siehe Anhang A.4.1 auf Seite 157*

gestartet, wodurch die vorgenommen Änderung in den nachfolgenden Aktualisierungsdurchgängen auch in den Identitätsgleichungen verketteter gleichartiger Knoten übernommen wird.

Sollten in einer Identitätsgleichung mehrere bekannte Variablen vorkommen, werden die Lösungen auf der Ergebnisliste dieser Variablen verglichen, um die Konsistenz der Ermittlungsergebnisse sicherzustellen. Sollten die Lösungen unterschiedlichen sein, wird eine Fehlermeldung ausgegeben und die Ausführung der Toolbox abgebrochen, da derartig Uneindeutigkeiten bereits in der Vorstufe behoben hätten werden müssen.

Die Aktualisierung wird direkt vor Beginn des eigentlichen Ermittlungsprozesses im Anschluss an die Suche und gegebenenfalls notwendige Korrektur von Kausalitätskonflikten sowie direkt nach der erfolgreiche Ermittlung einer Zustandsableitung im regulären Lösungsprozess durchgeführt, wodurch sowohl der Aufwand der weiteren Ermittlungen reduziert als auch die Ergebnisliste stetig auf Fehler überprüft wird.

3.9. Lösungsalgorithmus

Nach der Erstellung sämtlicher Listen, dem erfolgreichen Durchlaufen beider Vorstufen sowie der anschließenden Aktualisierung der Lösungen aus den Identitätsgleichungen, wird von der Steuerfunktion *analysieren()*[22] die Ermittlung der einzelnen Zustandsgleichungen für den Vektor $\dot{\vec{x}}$ und Ausgangsgleichungen für den Vektor $\vec{y}$ der Zustandsraumdarstellung des vom Benutzer modellierten Bondgraphen gestartet.
Hierfür wird die Funktion *ermitteln_ Gleichung()*[23] für die einzelnen Zustandsableitungen aus $\dot{\vec{x}}$ verwendet. Diese enthält den eigentlichen Hauptalgorithmus zur Ermittlung von Variablen.
Die Funktion ermittelt für die als Zeichkenette übergebene Variable die zugehörige Gleichung inklusive der zusätzlichen benötigten Angaben zur Erstellung eines vollständigen Datensatzes auf den Gleichungslisten und gibt diesen an die aufrufende Funktion zurück.

Ist für die angeforderte Variable keine Ergebnisgleichung auf der Ergebnisliste hinterlegt, wird versucht, diese in mehreren Stufen aus der Grund- und Ergebnis sowie der Knotenliste zu ermitteln, wobei der gesamte Ermittlungsprozess in insgesamt *fünf Ermittlungsstufen* unterteilt ist. Der Ablauf der Lösungsermittlung ist in Abbildung 3.10

[22] *Quellcode: siehe Anhang A.2.1 auf Seite 122*
[23] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

schematisch dargestellt. Zusätzlich zum mehrstufigen Lösungsverfahren kommt parallel ein Mechanismus zur Sperrung und Freigabe von Variablen und Gleichungen zum Einsatz, um während der Ermittlung algebraische Schleifen zu vermeiden.

In der *ersten Stufe* der Ermittlungsfunktion wird versucht die Gleichung der übergebenen Variable direkt von der Ergebnisliste zu laden, was nur möglich ist, wenn die Variable beziehungsweise deren Gleichung bekannt ist. Sollte die Variable nicht auf der Ergebnisliste gefunden werden können, geht die Ermittlung der gesuchten Gleichung in die nächste Stufe über.

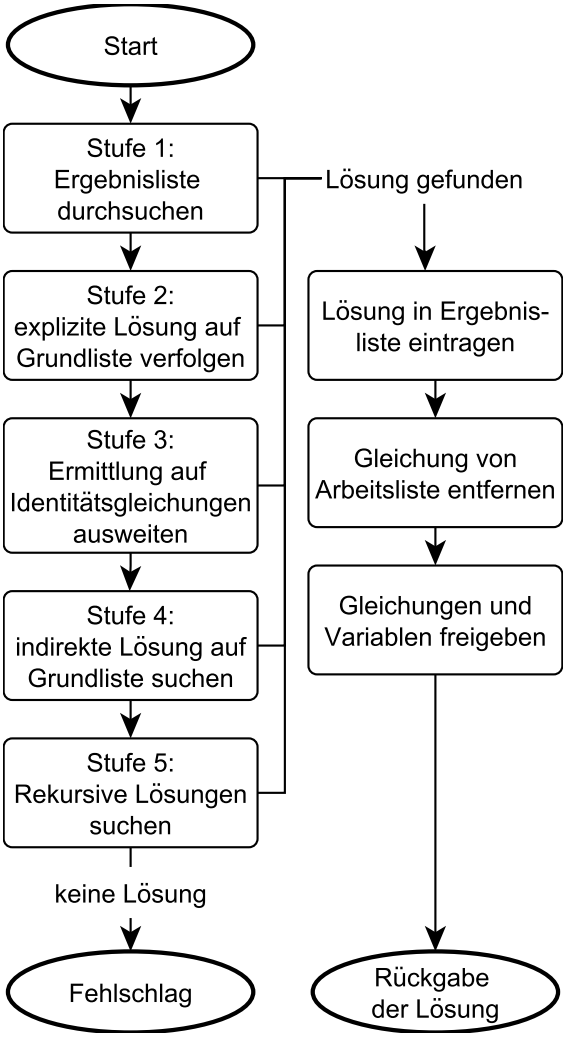


Abbildung 3.10.: Ermittlungsschema

Die *zweite Stufe* beinhaltet die Durchsuchung der Grundliste nach Gleichungen, welche die gesuchte Variable explizit beschreiben. Des Weiteren wird versucht, die gefundenen Gleichungen direkt als rekursive Lösung für die angeforderte Variable zu nutzen, um die Einleitung weiterer Ermittlungsrekursionen zu reduzieren.

Wenn durch die zweite Stufe keine Lösung gefunden werden konnte, wird in der *dritten Stufe* die Ermittlung auf eventuell vorliegende Identitätsgleichungen ausgeweitet, deren Bestandteil die aktuell gesuchte Variable ist.
Die Variablen der zutreffenden Identitätsgleichung oder der zutreffenden Identitätsgleichungen sind per Definition mit der gesuchten Variable identisch, wobei sukzessive für jede einzelne enthaltene Variable die Funktion *ermitteln_Gleichung()*[24] rekursiv aufgerufen wird. Hierdurch wird bei jedem Aufruf ein separater Ermittlungsprozess für die jeweilige Variable ausgelöst.

[24] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

Kann über die identischen Variablen der ursprünglich gesuchten Variable keine Lösung gefunden werden oder kommt die gesuchte Variable in keiner Identitätsgleichung vor, wird zur *vierten Stufe* des Ermittlungsalgorithmus übergegangen.
Da sämtliche auf der Grundliste sowie auf der Ergebnisliste eingetragene Gleichungen gemäß der Kausalität des zugehörigen Bonds erstellt wurden, muss nicht für jede angeforderte oder während der Ermittlung einer Zustandsänderung notwendig gewordene Variable eine diese explizit beschreibende Gleichung in einer der Listen vorkommen.
Um für nicht explizit beschriebene Bondvariablen ebenfalls eine Lösung ermitteln zu können, wird das starre Kausalitätskonzept der Bondgraphen in der *vierten Stufe* der Ermittlungsfunktion aufgeweicht und in der Grundliste nach Gleichungen gesucht, welche die gesuchte Variable in den Gleichungsrümpfen enthalten und somit die gesuchte Variable indirekt beschreiben könnten. Die gefunden Gleichungen werden nach der gesuchten Variable in deren explizite Form umgestellt und der Ermittlungsprozess rekursiv für die enthaltenen Unbekannten fortgesetzt.
Durch dieses Vorgehen werden des Weiteren vom Benutzer eventuell inkorrekte festgelegte Kausalitäten automatisch korrigiert.

Sollten die Ermittlungsstufen 1 bis 4 von *ermitteln_ Gleichung()* für eine gesuchte Variable keine Lösung liefern können, wird in der *fünften Stufe* nach bereits verwendeten Gleichungen gesucht, welche eine oder mehrere bereits im Ermittlungsprozess in expliziter Form enthaltene Variablen beschreiben.
Bei der Substitution derartiger Gleichungen würden für die betroffenen Variablen Gleichungen resultieren, in welchen diese Variablen in ursprünglich expliziter Form beschrieben sind, aber auch gleichzeitig im Gleichungsrumpf vorkommen, wodurch die resultierende Gleichung von diesen abhängen.
Diese speziellen Gleichungen können nach der ursprünglich expliziten Variable aufgelöst werden, was in dieser Arbeit als *rekursive Lösung* bezeichnet wird. Diese Lösungsvariante ist von den normalen rekursiven Funktionsaufrufen zu unterscheiden und wird mit Hilfe der Zusatzfunktion *ermitteln_ rekursiv()*[25] realisiert.

Die Stufen 1 bis 5 sind in den zugehörigen Abschnitten 3.9.1 bis 3.9.5 der vorliegenden Arbeit detaillierter beschrieben.

[25] *Quellcode: siehe Anhang A.2.3 auf Seite 141*

3.9.1. Stufe 1: bekannte Gleichungen

Die erste Stufe der Gleichungsermittlung einer an die Funktion *ermitteln_Gleichung()*[26] übergebene Variable besteht im Durchsuchen der globalen Ergebnisliste.
Hierfür wird die zweite Spalte der Ergebnisliste mit der gesuchte Variable abgeglichen, wobei es sich bei einer Übereinstimmung um die Lösung der Variable handelt. Für die Durchsuchung der Ergebnisliste wird die Hilfsfunktion *durchsuche_Ergebnisliste()*[27] der Toolbox verwendet, welche im Fall eines Treffers die betreffende Zeile der Ergebnisliste an *ermitteln_Gleichung()* zurückgibt.
Wird eine Übereinstimmung auf der Ergebnisliste gefunden, ist diese Variable bereits vorher bekannt gewesen und die Ermittlungsfunktion *ermitteln_Gleichung()* returniert nach Erhalt der Rückgabewerte *durchsuche_Ergebnisliste()* ebenfalls mit diesen Werten. Kann keine zugehörige Gleichung auf der Ergebnisliste gefunden werden, gibt *durchsuche_Ergebnisliste()* leere Rückgabewerte an *ermitteln_Gleichung()* zurück, was als Fehlschlag der ersten Ermittlungsstufe interpretiert wird.

3.9.2. Stufe 2: explizite Gleichungen

In der zweiten Stufe wird die Grundliste nach die gesuchte Variable explizit beschreibenden Gleichungen durchsucht. Auf der Grundliste können jedoch im Gegensatz zur Ergebnisliste theoretisch mehrere derartige Gleichungen vorliegen, da die gesuchte Variable gleichzeitig sowohl von Bilanzen als auch von den Gleichungen der Senken oder Umformer gleichzeitig explizit beschrieben sein kann.

Für den Abgleich der gesuchten Variable mit den expliziten Variablen der Grundliste wird die Hilfsfunktion *durchsuche_Grundliste()*[28] verwendet, welche analog der Funktion *durchsuche_Ergebnisliste()*[29] arbeitet, jedoch im Fall von mehreren Übereinstimmungen die entsprechenden Gleichungsnummern, Gleichungsrümpfe sowie Bemerkungen als Vektor zurückgibt.

Es können maximal zwei Übereinstimmungen für eine gesuchte Variable auf der Grundliste vorliegen, da andernfalls die gesuchte Variable in mehr als einem Bond vorkommen muss, was einen Fehler während der Variablenbezeichnung impliziert.
Sollten zwei Übereinstimmungen für eine Variable vorliegen, muss es sich bei einer der gefundenen Gleichungen um eine Knotenbilanz und bei der anderen um einen Umfor-

[26] *Quellcode: siehe Anhang A.2.2 auf Seite 127*
[27] *Quellcode: siehe Anhang A.5.2 auf Seite 163*
[28] *Quellcode: siehe Anhang A.5.3 auf Seite 164*
[29] *Quellcode: siehe Anhang A.5.2 auf Seite 163*

mer oder eine Senke handeln. Um den Ermittlungsprozess zu beschleunigen, werden die vektoriellen Rückgabewerte von *durchsuchen_Grundliste()* vorsortiert zurückgegeben, wobei die Werte der Knotenbilanz immer an zweiter Stelle in den Rückgabevektoren angeordnet sind. Die an erster Stelle zurückgegebene Gleichung der Senke oder des Umformers werden zuerst und somit vor der Knotenbilanz verarbeitet. Hierdurch kann die Geschwindigkeit der Ermittlung gegebenenfalls erhöht werden, da die Elementgleichungen von Umformern oder Senken in Regel von weniger Bondvariablen als Bilanzen abhängen und somit in günstigen Fällen zu einer schnelleren Lösung führen würden.

Werden keine geeigneten Gleichungen auf der Grundliste gefunden, resultiert dies in einer leeren Rückgabe der Hilfsfunktion *durchsuche_Grundliste()* und die zweite Ermittlungsstufe wird somit unausführbar. Bei einem derartigen Fehlschlag wird in *ermitteln_Gleichung()*[30] zur dritten Stufe übergegangen, auf deren Funktionsweise im folgenden Abschnitt 3.9.3 eingegangen wird.
Die von der Grundliste geladenen Gleichungen werden nacheinander verarbeitet, wobei zuerst überprüft wird, ob die jeweils aktuelle Gleichung mit ihren enthaltenen Variablen für eine rekursive Lösung geeignet ist. Sollte die Gleichung hierfür in Frage kommen, wird die rekursive Lösung mit dieser Gleichung simuliert, bevor diese tatsächlich durchgeführt wird. Wird durch die Probesimulation festgestellt, dass eine Lösung mit dieser Gleichung für die im derzeitigen Ermittlungsprozess enthaltenen Variablen erzielt werden kann, wird die aktuelle Gleichung mit ihren Zusatzinformationen direkt als Lösung zurückgegeben, wodurch die Ermittlung der gesuchten Variable erfolgreich beendet wird. Die Eigenschaften von rekursiven Lösungen, deren Verwendung sowie deren Auflösung und die Testsimulationen der Lösungen wird Abschnitt 3.9.5 auf Seite 90 erläutert.

Sollte die aktuelle Gleichung nicht als rekursive Lösung geeignet sein, da nicht alle in der Gleichung enthaltenen Variablen im aktuellen Ermittlungsprozess eingebunden sind, oder die Verwendung der Gleichung als rekursive Lösung in der Probesimulation keine sinnvollen Ergebnisse liefert, wird überprüft, ob die Gleichung auf der globalen Liste *gesperrteGleichungen* vorhanden ist.
Sollte dies der Fall sein, ist die Verwendung der Gleichung durch den Ermittlungsprozess selbst untersagt. Auf der globalen Sperrliste *gesperrteGleichungen* werden die Gleichungsnummern von Gleichungen vermerkt, die bereits im laufenden Ermittlungsprozess verwendet werden und somit Teil der rückwärtigen Substitutionskette sind. Die erneute Verwendung von bereits in der Ermittlung eingebundenen Gleichungen würde bei

[30] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

Rücksubstitution dazu führen, dass die Gleichung in den vorhergehenden Lösungsschritten in sich selbst eingesetzt wird und eine Auslöschung resultiert, was zu fehlerhaften Ergebnissen oder algebraischen Schleifen führen kann.

Liegt keine Sperrung vor, wird im anschließenden Hauptteil der zweiten Stufe versucht, die in der Gleichung enthaltenen Subvariablen mit normaler Substitution zu ermitteln. Um die Ergebnis- und Grundliste während des Ermittlungsprozesses sowie der in diesem notwendigen Substitutionen und Umformungen von Gleichungen zu schützen, wird eine zusätzliche *globale Arbeitsliste* verwendet. Der Aufbau dieser Arbeitsliste ist ebenfalls identisch mit dem der anderen beiden Listen.

Die aktuelle Gleichung wird mit Hilfe der Zusatzfunktion *kopieren_GL2AL()*[31], an welche zur Identifikation der betroffenen Gleichung deren Gleichungsnummer übergeben wird, von der Grundliste auf die Arbeitsliste kopiert. Der Datensatz wird unverändert übernommen und keine neue Gleichungsnummer vergeben, da es sich lediglich um eine Auslagerung und keine Neuerstellung handelt.
Im Anschluss an die Übertragung der aktuellen Gleichung von der Grundliste auf die Arbeitsliste, werden die Gleichung und die ursprünglich gesuchte Variable gesperrt, wobei die zu sperrende Variable analog dem Sperren von Gleichungen auf der globale Sperrliste *gesperrteVariablen* vermerkt wird. Gesperrte Variablen dürfen nicht durch *ermitteln_Gleichung()*[32] ermittelt werden und sind von einer Verwendung im aktuellen Ermittlungsprozess ausgeschlossen. Diese Variablen sind entweder bereits im aktuellen Ermittlungsprozess enthalten, wobei deren erneute Verwendung zu algebraischen Schleifen führen würde, oder deren Ermittlung schlug im aktuellen Prozess bereits fehl, wodurch die jeweils gesperrten Variablen zu diesem Zeitpunkt nicht ermittelbar sind.

Anschließend werden die zeitveränderlichen Variablen der aktuellen Gleichung extrahiert und in einem internen Vektor zwischengespeichert.
Die Subvariablen werden anschließend in alphabetischer Reihenfolge auf Sperrung überprüft. Sollten keine Sperrungen vorliegen, wird versucht diese durch rekursive Aufrufe der Funktion *ermitteln_Gleichung()* der Reihe nach zu ermitteln. Die im erfolgreichen Fall ermittelten Lösungen werden in der aktuellen Gleichung entsprechend substituiert.

Ist auch nur eine der Subvariablen gesperrt oder kann nicht durch rekursiven Aufruf von *ermitteln_Gleichung()* ermittelt werden, kann die aktuelle Gleichung nicht vollständig ermittelt werden. Daher wird in diesem Fall die Gleichung wieder freigegeben und von

[31] *Quellcode: siehe Anhang B.2.6 auf Seite 194*
[32] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

der Arbeitsliste gestrichen. Sollte eine weitere Gleichung zur expliziten Beschreibung der gesuchten Variable vorliegen, wird der Vorgang für diese wiederholt und die zweite Stufe mit dieser fortgesetzt. Liegt keine derartige Gleichung vor, schlägt die zweite Ermittlungsstufe fehl und dritte Stufe wird eingeleitet.

Im Zuge der Substitution der Sublösung in die aktuelle Gleichung wird überprüft, ob diese von der eigentlich gesuchten abhängen, wodurch es sich bei der betreffenden Sublösung um eine rekursive Lösung handeln muss. In diesem Fall wird nach der Substitution der Sublösung die aktuelle Gleichung auf der Arbeitsliste nach der gesuchten Variable aufgelöst, welche sich zu diesem Zeitpunkt sowohl auf der linken als auch auf der rechten Seite der aktuellen Gleichung befindet.
Wenn alle Lösungen der Subvariablen vollständig ermittelt wurden, wird die Gleichung inklusive der zugehörigen Gleichungsnummer sowie Bemerkung als angeforderte Lösung für die gesuchte Variable in die Ergebnisliste eingetragen und des Weiteren der Datensatz von der Arbeitsliste gestrichen sowie die Gleichung und gesuchte Variable von den entsprechenden Sperrlisten entfernt.

Sollte eine der Subvariablen der aktuellen Gleichung gesperrt sein oder nicht durch rekursive Aufrufe von *ermitteln_ Gleichung()* ermittelt werden können, hängt die Gleichung nicht ausschließlich von den Zustands- und / oder Eingangsvariablen des Bondgraphen ab und ist somit nicht als Lösung verwendbar.
Sind keine weiteren Gleichungen für die übergebene Variable verfügbar, wird von *ermitteln_ Gleichung()* zur dritten Ermittlungsstufe für die gesuchte Variable übergegangen, was im folgenden Abschnitt 3.9.3 beschrieben ist.
Der Ablauf der zweiten Ermittlungsstufe ist in Schema 3.11 auf der nächsten Seite graphisch dargestellt.

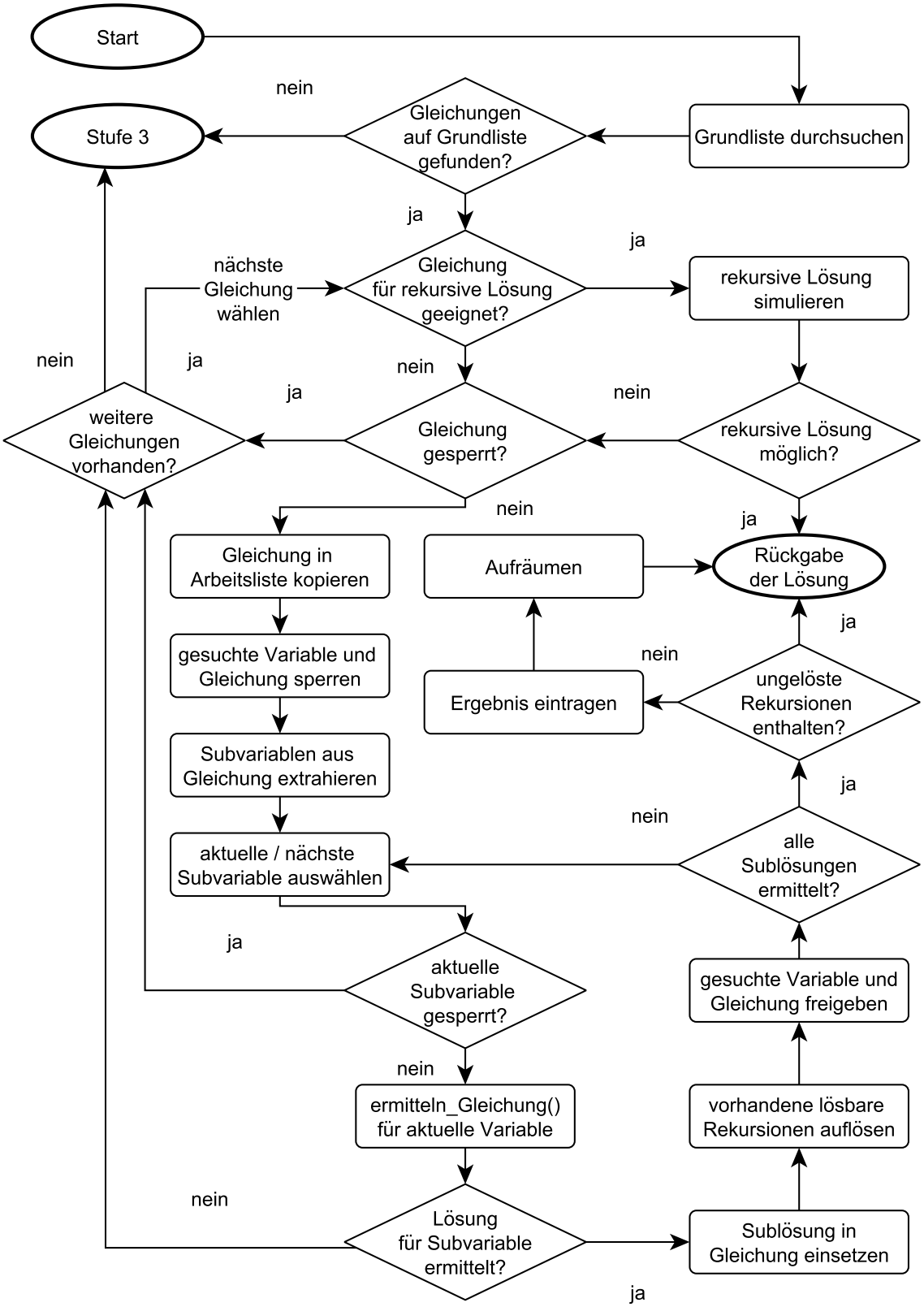


Abbildung 3.11.: Schematischer Ablauf der zweiten Stufe

3.9.3. Stufe 3: Lösung aus Identitätsgleichung

Konnte in der zweiten Stufe des Lösungsalgorithmus für die gesuchte Variable keine Lösung ermittelt werden, wird von der Steuerfunktion *analysieren()*[33] die Ermittlung auf die Identitätsgleichungen der jeweiligen Knoten ausgeweitet, welche die gesuchte Variable enthalten.

Da die in einer Identitätsgleichung enthaltenen Effort- beziehungsweise die Flowvariablen gleich sein müssen, würde die erfolgreiche Ermittlung einer dieser Variablen ebenfalls die Lösung für restlichen in der Identitätsgleichung enthaltenen Variablen und somit auch für die ursprünglich gesuchte Variable liefern.
Es ist hierbei zu beachten, dass die gesuchte Variable bereits auf der Ergebnisliste vorkommen würde und somit bekannt wäre, wenn auch nur eine Variable der selben Identitätsgleichung eine Lösung besäße, da vor dem Beginn eines jeden Ermittlungsprozesses und somit vor jedem Initialaufruf von *ermitteln_ Gleichung()*[34] durchgeführte Aktualisierung der Identitätsgleichungen das Eintragen dieser identischen Lösung bewirkt hätte.
Der schematische Ablauf der dritten Stufe ist begleitend zu dessen anschließender Erläuterung in Abbildung 3.12 dargestellt.

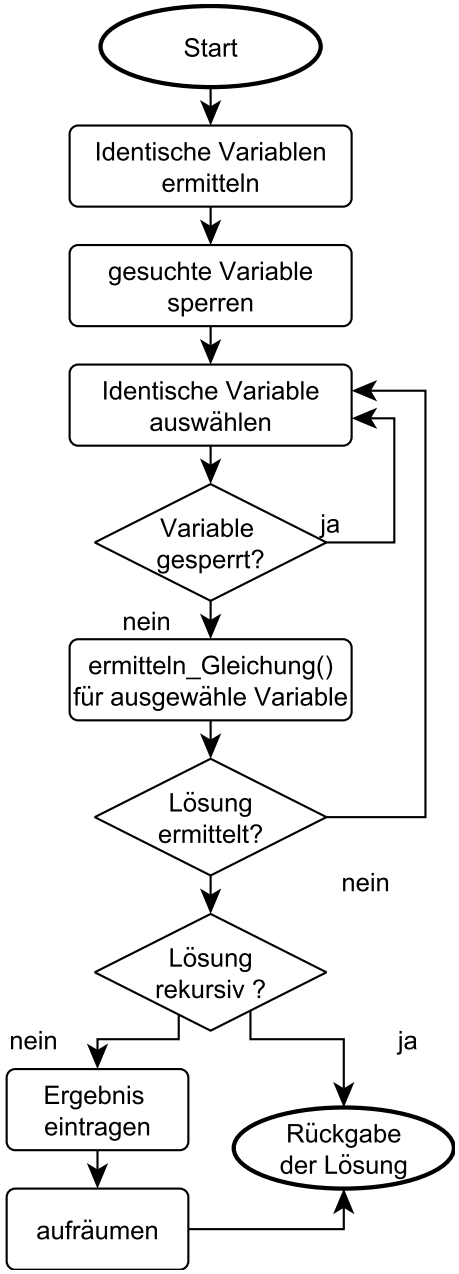


Abbildung 3.12.: Ablauf der dritten Stufe

[33] *Quellcode: siehe Anhang A.2.1 auf Seite 122*
[34] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

Im ersten Schritt der dritten Stufe wird unter der Verwendung der Hilfsfunktion *ermitteln_ Gleichnisvariablen()* überprüft, ob die gesuchte Variable in mindestens einer Identitätsgleichung vorkommt. Die Funktion gibt die Variablen der Identitäten zurück, in welchen die gesuchte Variable enthalten ist und die daher mit der gesuchten Variable identische Lösungen besitzen müssen.

Des Weiteren ist in dieser Hilfsfunktion die Verkettung von Knoten berücksichtigt. Der Umgang miteinander verbundener Knoten gleichen Typs und deren Verkettung sowie der Ablauf von *ermitteln_ Gleichnisvariablen()*[35] ist in Abschnitt 3.10.2 erläutert.

Kommt die gesuchte Variable in keiner Identitätsgleichung vor, geht die Steuerfunktion direkt zur vierten Ermittlungsstufe über. In dieser wird das Kausalitätsprinzip des Bondgraphen teilweise aufgelöst, wodurch unter anderem vom Benutzer eventuell fehlerhaft festgelegte Bondkausalitäten korrigiert werden. Die vierte Stufe ist im folgenden Abschnitt 3.9.4 auf der nächsten Seite beschrieben.

Im Anschluss hieran wird nacheinander versucht, Lösungen für die mit der gesuchten Variable identischen Variablen unter rekursiver Verwendung der Funktion der *ermitteln_ Gleichung()*[36] zu ermitteln. Konnte eine Lösung für eine beliebige in den (verketteten) Identitätsgleichungen enthaltene Variable ermittelt werden, ist diese automatisch für alle Variablen der Identität und somit für auch für die gesuchte Variable gültig. In diesem Fall wird die gefundenen Lösung zurückgegeben.

Vor dem Returnieren der Funktion *ermitteln_ Gleichung()* bei einem erfolgreichem Abschluss der vierten Stufe, wird die ermittelte Identitätslösung darauf hin untersucht, ob diese ausschließlich von Zustands- oder Eingangsvariablen abhängt. Sollte dies nicht der Fall sein und mindestens ein regulärer Effort oder Flow beziehungsweise eine Zustandsableitung in der Lösung enthalten sein, wird diese Lösungsgleichung als rekursive Lösung angenommen.
Ist die im jeweils aktuellen Aufruf von *ermitteln_ Gleichung()* gesuchte Variable in der Lösungsgleichung enthalten, wird die Rekursion durch Umstellen aufgelöst und regulär in die Ergebnisliste eingetragen, sofern hierdurch die Lösung nur noch von Zuständen oder Eingängen abhängen sollte. Kommt die gesuchte Variable im Fall einer rekursiven Lösung nicht in der zurückgegebenen Lösung der jeweiligen Identitätsvariable vor, kann die Rekursion erst in vorhergehenden / übergeordneten Ermittlungsschritte der Rekursion gelöst werden, wodurch die Lösung nicht in die Ergebnisliste eingetragen werden darf.

[35] *Quellcode: siehe Anhang C.3.6 auf Seite 220*
[36] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

Des Weiteren werden auch nach dem erfolgreichem Abschluss der dritten Ermittlungsstufe die betroffenen Identitätsgleichungen nicht gesondert aktualisiert und somit die ermittelte Lösung nicht auf die restlichen Variablen der Identitäten angewendet. Die Aktualisierung erfolgt erst regulär im Anschluss an die erfolgreiche Ermittlung der gesuchten Initialvariable oder Zustandsableitung in der Steuerfunktion *analysieren()*[37], da der von dieser Funktion ausgelöste Gesamtprozess immer vor einer eventuellen Verwendung der nicht im Ermittlungsprozess eingebundenen restlichen Identitätsvariablen beendet werden muss.

Kann aus den Identitätsgleichungen und somit den in diesen enthaltenen identischen Variablen keine gültige Lösung für die gesuchte Variable ermittelt werden, geht die Ermittlung in die vierte Stufe über.

3.9.4. Stufe 4: Umgehen der Kausalität

In den vorhergehenden Ermittlungsstufen wurde versucht, eine Lösung für die gesuchte Variable beziehungsweise der mit dieser identischen Variablen zu finden, indem aus der Grundliste oder Ergebnisliste Gleichungen verwendet wurden, welche diese Variablen in expliziter Form beschreiben.
Wie bereits in Abschnitt 3.6 auf Seite 66 erläutert, werden sämtliche Bondgleichungen gemäß der vom Benutzer oder durch das jeweilige Element definierten Kausalität des Bonds erstellt. Konnte zu diesem Zeitpunkt der Ermittlung keine Lösung für die gesuchte Variable der vorhergehenden Stufen ermittelt werden, muss die Möglichkeit einer fehlerhaft gesetzten Kausalität in Betracht gezogen werden.
Der Ablauf der vierten Ermittlungsstufe ist begleitend zu deren anschließender Erläuterung in Abbildung 3.13 schematisch dargestellt.

In der vierten Stufe werden die Gleichungsrümpfe der auf der Grundliste hinterlegten Bondgleichungen nach der zu ermittelnden Variable durchsucht, wofür die Hilfsfunktion *durchsuche_Gleichungen()*[38] verwendet wird. Diese gibt die einzigartigen Gleichungsnummern der von der gesuchten Variable abhängigen Gleichungen zurück, sofern derartige Gleichungen auf der Grundliste existieren. Die Ergebnisliste wird in diese Suche nicht eingebunden, da deren Gleichungsrümpfe lediglich von Zustandsvariablen und / oder Quellvariablen abhängen und daher keine unbekannten oder gesuchten Bondvariablen enthalten können.

[37] *Quellcode: siehe Anhang A.2.1 auf Seite 122*
[38] *Quellcode: siehe Anhang A.5.4 auf Seite 166*

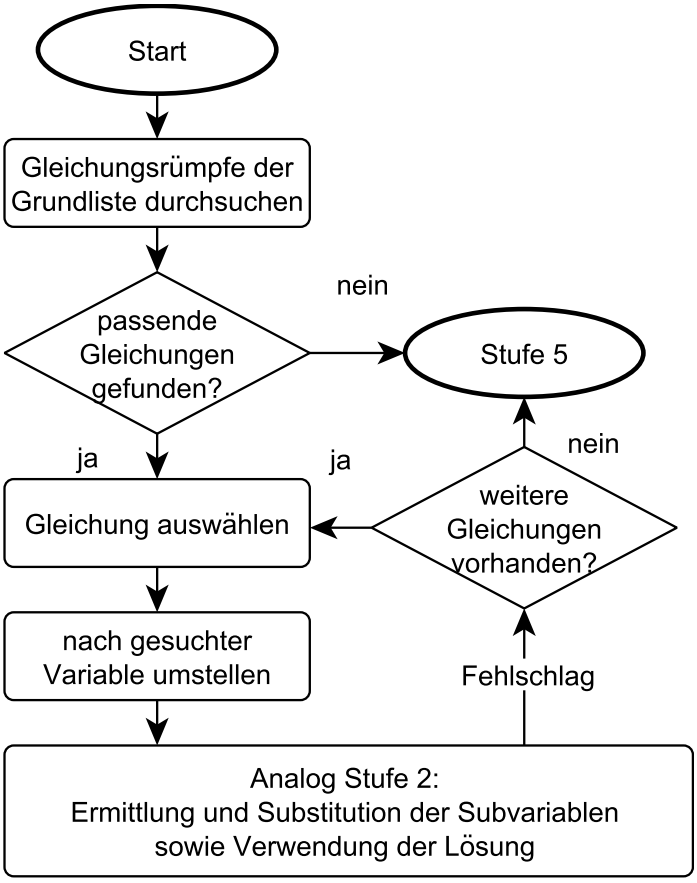


Abbildung 3.13.: Schematischer Ablauf der vierten Stufe

Es kann angenommen werden, dass Elementgleichungen, welche ausschließlich von der gesuchten Variable abhängen, eine falsche Kausalität besitzen und im Fall der korrekten Kausalität eine vergleichsweise schnellere Ermittlung ermöglicht hätten, da hierbei eine Lösung für die gesuchte Variable unter Umständen bereits in der zweiten Ermittlungsstufe erzielt werden könnte. Die betreffenden Gleichungen werden mit Hilfe von *durchsuche_Gleichungen()* zusätzlich zu deren Auswahl von der Grundliste geladen und in den folgenden Schritten einzeln verarbeitet, wobei die Gleichungen funktionsintern mit Hilfe der zugehörigen Gleichungsnummer identifiziert werden.

Im ersten Schritt wird überprüft, ob die aktuelle Gleichung gesperrt ist. Ist dies der Fall, wird diese Gleichung verworfen und mit den gegebenenfalls vorhandenen verbleibenden Gleichungen fortgefahren. Sollte die aktuelle Gleichung nicht gesperrt sein, wird diese gesperrt und anschließend nach der gesuchten Variable umgestellt, wobei die resultierende Gleichung diese gesuchte Variable anschließend explizit beschreibt. Für die Umstellung wird die Hilfsfunktion *umstellen_Gleichung()* verwendet, welche den an diese Funkti-

on übergebenen explizite Variable mit zugehörigen Gleichungsrumpf nach der ebenfalls übergebenen Zielvariable umstellt und zurückgibt.
Die Funktion *umstellen_ Gleichung()*[39] gibt lediglich den resultierenden Gleichungsrumpf zurück, da vorausgesetzt wird, dass die durch die umgestellte Gleichung explizit beschriebene Zielvariable auf der Ausführungsebene des Caller bekannt ist. Von der Funktion wird des Weiteren keine Eintragung der resultierenden Gleichung in eine der drei Gleichungslisten vorgenommen. Die nach der gesuchten Variable umgestellte Gleichung wird anschließend unter Verwendung der Hilfsfunktion *eintragen_ Gleichung()*[40] in die Arbeitsliste eingetragen, wobei die Gleichungsnummer der Ursprungsgleichung übernommen und die Bemerkung der Gleichung auf "umgestellt" gesetzt wird.
Nach dem erfolgreichen Umstellen der Gleichung werden die in dieser enthaltenen Variablen untersucht und gegebenenfalls ermittelt. Die hierfür notwendige Extraktion der enthaltenen Subvariablen und die Ermittlung der zugehörigen Sublösungen sowie die Substitution und Rückgabe der Lösungsgleichung erfolgt analog zur zweiten Stufe, welche in Abschnitt 3.9.2 auf Seite 81 beschrieben ist.
Kann die jeweils aktuelle umgestellte Gleichung nicht vollständig ermittelt werden, wird der Vorgang für die gegebenenfalls vorliegenden von der gesuchten Variable abhängenden restlichen Gleichungen wiederholt.

Die vierte Stufe umgeht das Prinzip der Kausalität, um Lösungen trotz gegebenenfalls fehlerhafter Kausalitäten zu ermitteln. Durch die Implementierung einer Kausalitätskontrolle mit angeschlossener Korrekturmöglichkeit in den Vorstufen des Ermittlungsprozesses könnte die vierte Stufe in späteren Versionen obsolet und der Rechenaufwand reduziert werden.

3.9.5. Stufe 5: rekursive Lösungen

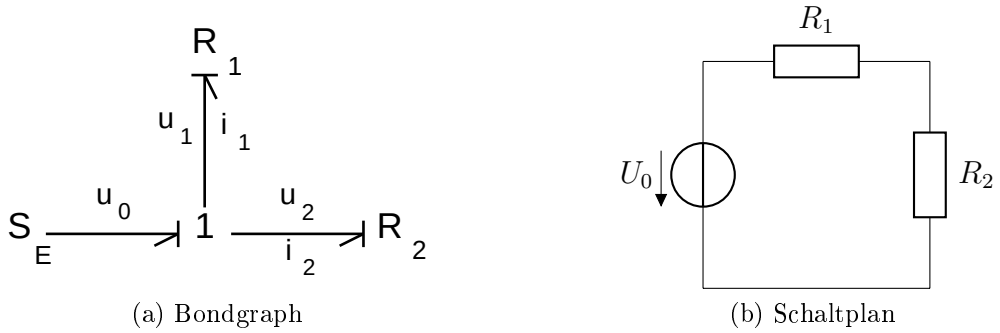
Mit dem Scheitern der vierten Ermittlungsstufe sind alle Versuche zur Ermittlung der gesuchten Variable durch reguläre Substitution mit ausschließlich von Zustandsvariablen und / oder Eingangsvariablen abhängenden Termen fehlgeschlagen.
In der fünften Stufe des Lösungsalgorithmus wird versucht, die gesuchte Variable durch eine Gleichung zu beschreiben, welche die gesuchte Variable explizit beschreibt und gleichzeitig von der gesuchten Variable abhängt, wodurch gezielt algebraische Schleifen herbeigeführt werden. Diese Gleichungen können jedoch nach der gesuchten Variable

[39] *Quellcode: siehe Anhang A.3.3 auf Seite 155*
[40] *Quellcode: siehe Anhang B.2.5 auf Seite 193*

aufgelöst werden und gültige Lösungen ergeben. In der vorliegenden Arbeit sowie in der aktuellen Version der Toolbox werden diese aufgrund des in der Gleichung vorliegenden Selbstbezugs als *rekursive Lösung* bezeichnet.

Rekursive Lösungen werden häufig für die Beschreibung von Bondgraphen im Zustandsraum benötigt, wenn diese keine Energiespeicher besitzen und somit keine Zustandsvariablen besitzen, wodurch der Zustandsvektor $\vec{x}$ sowie dessen Ableitungen $\dot{\vec{x}}$ nicht existieren. In diesem Fall hängen derartige Graphen lediglich von den Eingangsvariablen des Vektor $\vec{u}$ ab.
In den Zustandsraumdarstellungen dieser Graphen entfallen die namensgebenden Zustandsgleichungen vollständig, wodurch zusätzlich zu den beiden Vektoren $\dot{\vec{x}}$ und $\vec{x}$ ebenfalls die Matrizen $\underline{A}$ und $\underline{B}$ wegfallen. Hierdurch existieren nur die Ausgangsgleichungen des Systems für die Elemente des benutzerdefinierten Ausgangsvektors $\vec{y}$, wobei diese Gleichungen lediglich vom Eingangsvektor $\vec{u}$ und der zugehörigen Koeffizientenmatrix $\underline{D}$ abhängen, da durch das Fehlen des Zustandsvektors $\vec{x}$ ebenfalls die zugehörige Matrix $\underline{C}$ in den Ausgangsgleichungen entfällt.

Die Notwendigkeit sowie die Anwendung von rekursiven Lösungen soll anhand des folgenden Bondgraph 3.14 einer Reihenschaltung zweier Widerstände beispielhaft erläutert werden.



(a) Bondgraph (b) Schaltplan

Abbildung 3.14.: Reihenschaltung von zwei Widerständen

Die Reihenschaltung der Widerstände R_1 und R_2 wird durch den seriellen 1-Knoten repräsentiert, wodurch an diesem die folgende Bilanz mit der Quellspannung U_0 der Effortquelle S_E entsteht:

$$U_0 = U_1 + U_2 \tag{3.10}$$

Des Weiteren gilt am Knoten die Gleichheit der elektrischen Ströme durch die beiden Widerstände beziehungsweise deren zugehörige Bondvariablen:

$$I_1 = I_2 \tag{3.11}$$

Je nach Kausalitäten der eingehenden Bonds an den Widerstände, ergibt sich eine von drei möglichen Gleichstellungspolitischen für Verbraucher. Im Beispiel wird davon ausgegangen, dass bei beiden Widerstände die Efforts beziehungsweise Spannungen von den jeweiligen Strömen abhängen, was als Flowkausalität bezeichnet wird und wodurch die folgenden Elementgleichungen für die Spannungen U_1 und U_2 vorliegen:

$$U_1 = R_1 * I_1 \tag{3.12}$$

$$U_2 = R_2 * I_2 \tag{3.13}$$

Da die einzigen Endpunkte des Beispielgraphen resistive Verbraucher sind, verfügt der Bondgraph über keine Zustandsvariablen, wodurch lediglich die Ausgangsgleichungen des Systems in Abhängigkeit der Eingangsspannung U_0 und der zugehörigen Koeffizientenmatrix $\underline{D}$ existieren. Aus diesem Grund werden zuerst die Ausgangsvariablen als Elemente des Vektors $\overrightarrow{y}$ festgelegt, wobei in diesem Beispiel sämtliche unbekannte Efforts und Flows als Ausgangsvariablen fungieren sollen:

$$\vec{y} = \begin{pmatrix} U_1 \\ I_1 \\ U_2 \\ I_2 \end{pmatrix} \tag{3.14}$$

Die Variablen werden gemäß ihrer Reihenfolge in $\dot{\vec{x}}$ und $\vec{y}$ ermittelt, wobei in diesem Fall mit der Spannung U_1 begonnen wird, da kein Ableitungsvektor $\dot{\vec{x}}$ vorliegt.
Hierfür wird in Gleichung 3.12 der Strom I_1 aufgrund der vorliegenden Stromidentität mit der nach I_2 umgestellten Gleichung 3.13 substituiert, wodurch sich die folgende Gleichung für U_1 ergibt:

$$U_1 = \frac{R_1}{R_2} * U_2 \tag{3.15}$$

Im nächsten Schritt wird die Spannungsbilanz 3.10 des seriellen Knotens nach der benötigten Spannung U_2 umgestellt und entsprechend in Gleichung 3.15 eingesetzt, was in folgender Gleichung für U_1 resultiert:

$$U_1 = \frac{R_1}{R_2} * (U_0 - U_1) = \frac{R_1}{R_2} * U_0 - \frac{R_1}{R_2} * U_1 \tag{3.16}$$

In einem Ermittlungsprozess ohne die Beirücksichtung von rekursiven Lösungen würde die Ermittlung an dieser Stelle versagen, da erneute Substitutionen für U_1 in Schleifen enden würde, da bereits sämtliche vorhandenen Gleichungen eingebunden sind. Um dies zu verhindern, wurden die bereits verwendeten Variablen und Gleichungen gesperrt, wodurch sich die Ermittlung nicht in eine Schleife begeben kann.
Eine Lösung für die gesuchte Variable U_1 kann jedoch durch Umformen der Gleichung erzeugt und somit der Selbstbezug der Gleichung aufgelöst werden. Hierbei wird der betreffenden Term für U_1 inklusive Vorfaktor auf die linke Seite subtrahiert:

$$U_1 + \frac{R_1}{R_2} * U_1 = \frac{R_1}{R_2} * U_0 = U_0 * \left(1 + \frac{R_1}{R_2}\right) \tag{3.17}$$

Im Zuge der Umformung wird anschließend die resultierende Gleichung 3.17 durch der Vorfaktor von U_1 dividiert, wodurch sich die Lösung für U_1 ergibt:

$$U_1 = \left(1 + \frac{R_1}{R_2}\right)^{-1} * \frac{R_1}{R_2} * U_0 \tag{3.18}$$

$$= \left(\frac{R_1 + R_2}{R_2}\right)^{-1} * \frac{R_1}{R_2} * U_0 \tag{3.19}$$

$$= \frac{R_1}{R_1 + R_2} * U_0 \tag{3.20}$$

Anschließend wird mit der Ermittlung der Ausgangsvariable I_1 fortgefahren. Da diese durch keine Gleichung explizit beschrieben ist, werden die Elementgleichung 3.12 umgestellt und die bereits ermittelte Lösung für U_1 in die umgestellte Gleichung eingesetzt. Aufgrund der Identitätsgleichung 3.11 für die Ströme / Flowvariablen des seriellen Knotens ergibt sich ebenfalls die Lösung für den Strom I_2:

$$I_1 = I_2 = \frac{1}{R_1} * U_1 = \frac{1}{R_1 + R_2} * U_0 \tag{3.21}$$

Die Lösung für I_2 wird für die Ermittlung der letzten unbekannten Ausgangsvariable U_2 verwendet, wodurch sich folgende Lösung für U_2 ergibt:

$$U_2 = R_2 * I_2 = \frac{R_2}{R_1 + R_2} \tag{3.22}$$

Da für den Beispielgraphen keine Zustandsableitungen $\dot{\vec{x}}$ ermittelt werden müssen, ist die Ermittlung der für die zustandslose Zustandsraumdarstellung benötigten Gleichungen abgeschlossen. Die Ausgangsgleichung des Graphen ergibt in vektorieller Form mit der resultierenden Koeffizienten Matrix, wie folgt, aus den ermittelten Lösungen:

$$\vec{y} = \underline{D} * \vec{u} = \begin{pmatrix} U_1 \\ I_1 \\ U_2 \\ I_2 \end{pmatrix} = \begin{pmatrix} \frac{R_1}{R_1+R_2} & 0 & 0 & 0 \\ 0 & \frac{1}{R_1+R_2} & 0 & 0 \\ 0 & 0 & \frac{R_2}{R_1+R_2} & 0 \\ 0 & 0 & 0 & \frac{1}{R_1+R_2} \end{pmatrix} * U_0 \tag{3.23}$$

3.9.5.1. Eignung von Gleichungen

Eine Gleichung kommt als potentielle Lösung nur dann in Frage, wenn alle in dieser Gleichung enthaltenen Variablen eine in der Ergebnisliste eingetragene Lösung besitzen oder bereits in den laufenden Ermittlungsprozess eingebunden sind.
Hierfür müssen die in der Gleichung enthaltenen Unbekannten auf der Arbeitsliste explizit durch Gleichungen beschrieben und in der zweiten Spalte dieser Liste eingetragen sein sowie über einen Gleichungsrumpf oder eine identische Variable in der zugehörigen dritten Spalte verfügen.

Die Eignung einer Gleichung als rekursive Lösung muss vor deren weiteren Verarbeitung überprüft werden, was unter Verwendung der Hilfsfunktion *ist_rekursiv_geeignet()*[41] geschieht.
Hierfür wird die zu überprüfende Gleichung beim Aufruf an die Funktion übergeben, welche die enthaltenen Variablen aus dem Gleichungsrumpf extrahiert. Eventuell in der übergebenen Gleichung enthaltene Konstanten werden ignoriert. Anschließend wird jede einzelne extrahierte Bondvariable auf Bekanntheit auf der Ergebnisliste und Vorhandensein auf der Arbeitsliste überprüft.

[41] *Quellcode: siehe Anhang A.6.3 auf Seite 170*

Kommt mindestens eine Variable nicht auf diesen Listen vor, muss es sich bei dieser um eine nicht in den aktuellen Ermittlungsprozess eingebundenen Unbekannte handeln, wodurch die aktuelle Gleichung für eine rekursive Lösung nicht geeignet ist.
Eine Gleichung ist ebenfalls als rekursive Lösung ungeeignet, wenn sämtliche enthaltenen Variablen bekannt sind. In diesem Fall wäre die explizit beschriebene Variable der Gleichung ebenfalls bekannt und die Gleichung müsste eigentlich auf der Ergebnisliste vermerkt sein, wodurch diese zuvor direkt als Lösung hätte dienen müssen. Da in diesen Fall ein Fehler in der aufrufenden Funktion von *ist_ rekursiv_ geeignet()* vorliegen muss, wird hierbei eine Fehlermeldung ausgelöst und der gesamte Lösungsprozess abgebrochen.

Ist die übergebene Gleichung als rekursive Lösung geeignet, werden von der Funktion *ist_ rekursiv_ geeignet()* der logische Wert *true* sowie eine Auflistung der in der Gleichung enthaltenen Unbekannten zurückgegeben. Andernfalls wird im gegenteiligen Falls *false* und ein leerer Variablenvektor returniert.

3.9.5.2. Präventive Simulation

Die Eignung von Gleichungen als potentielle rekursive Lösungen muss vor ihrer eventuellen Verwendung vollständig abgeklärt werden, wobei die grundsätzliche Eignung vorausgehend durch die Funktion *ist_ rekursiv_ geeignet()*[42] überprüft wurde, welche im vorherigen Abschnitt 3.9.5.1 auf der vorherigen Seite erläutert wird. Zusätzlich muss überprüft werden, ob die Verwendung einer geeigneten Gleichung als rekursive Lösung tatsächlich zu sinnvollen Ergebnissen führt und in übergeordneten Substitutionen keine algebraischen Schleifen oder Auslöschungen der zu ermittelnden Variablen produziert.

Um fehlerhaften Lösungen trotz geeigneter Gleichungen vorzubeugen, wird eine potentielle Lösung vor der eigentlichen Verwendung simuliert, da sich nur dann sinnvolle Lösungen ergeben, wenn der Vorfaktor der jeweiligen rekursiven Variable bei der Auflösung der Rekursion ungleich 1 ist. Kommt die explizite Variable einer Gleichung im Gleichungsrumpf mit dem Vorfaktor 1 vor, ist eine sinnvolle Auflösung nicht möglich, da sich die Gleichung nach der Subtraktion der rekursiven Variable während des Umstellvorgangs zu null ergibt.

Die eigentlichen Wirkungen von rekursiven Lösungen treten in der Regel erst in späteren Rücksubstitutionen ein, wodurch sich zum Zeitpunkt des Auffindens der jeweiligen potentiellen rekursiven Lösung eine Aussage über deren tatsächliche Korrektheit nur durch eine Simulation der Rücksubstitution treffen lässt. Die Simulation sowie die Bewertung

[42] *Quellcode: siehe Anhang A.6.3 auf Seite 170*

der Verwendbarkeit der als rekursive Lösungen potentiell geeigneten Gleichungen geschieht durch die Simulationsfunktion *teste_rekursive_Substitution()*[43].

Der schematische Ablauf des Simulationsprozesses ist in der umseitigen Abbildung 3.15 als Begleitung zu den folgenden Erläuterungen dargestellt.

An die Funktion *teste_rekursive_Substitution()* wird hierfür die im aktuellen Ermittlungsschritt gesuchte Variable, die durch die Hilfsfunktion *ist_rekursiv_geeignet()* überprüfte Gleichung sowie der aus der Gleichung gewonnene Vektor mit den zugehörigen rekursiven Variablen übergeben.
Die logische Variable *Status* ist der einzige Rückgabeparameter der Simulationsfunktion und signalisiert mit dem Wert *true* einen erfolgreichen Simulationslauf bei Rückkehr der Funktion, wodurch sichergestellt ist, dass die aktuelle Gleichung als rekursive Lösung zu sinnvollen Ergebnissen führt, andernfalls wird der Wert *false* zurückgegeben.

Um einen Probelauf der Lösungssubstitutionen ohne Veränderung der aktuellen Arbeitsliste zu ermöglichen, wird vor Beginn der Simulation eine funktionsinterne Kopie der Arbeitsliste erstellt und als Testliste verwendet. In dieser Testliste wird anschließend die zu überprüfende Gleichung für die gesuchte Variable im Gleichungsrumpf der letzten Zeile der Testliste und somit in der aktuellsten substituiert.
Nach dieser Substitution wird überprüft, ob hierdurch die explizite Variable der Gleichung ebenfalls in deren Rumpf vorkommt, wodurch ein Selbstbezug vorliegen würde. Die Überprüfung des Vorliegens eines derartigen *rekursiven Falls* wird von der Hilfsfunktion *ist_rekursiv()*[44] übernommen, an welche hierfür der zu überprüfende Gleichungsrumpf sowie die zugehörige explizite Variable übergeben wird. Die Hilfsfunktion returniert bei Rückkehr eine logische Variable zur Signalisierung eines etwaigen Selbstbezugs mit dem Wert *true* sowie in die in diesem Fall zugehörige rekursive Variable. Andernfalls werden der logische Wert *false* und eine leere Variablenbezeichnung zurückgegeben.

Liegt ein rekursive Fall vor, wird der Vorfaktor der rekursiven Variable im Gleichungsrumpf ermittelt. Trägt dieser den Wert 1, wird keine sinnvolle Auflösung der Rekursion möglich sein und die Simulation beendet sowie der Rückgabeparameter Status mit *false* zurückgegeben. Die Funktion kehrt in diesem Fall unmittelbar zurück und die Fehlschlagmeldung wird somit an den Caller weitergereicht.
Sollte der Vorfaktor den Wert 0 besitzen, wird die Ausführung mit einer Fehlermeldung abgebrochen, da durch Substitution von normalen und von rekursiven Lösungen

[43] *Quellcode: siehe Anhang A.2.4 auf Seite 149*
[44] *Quellcode: siehe Anhang A.6.2 auf Seite 169*

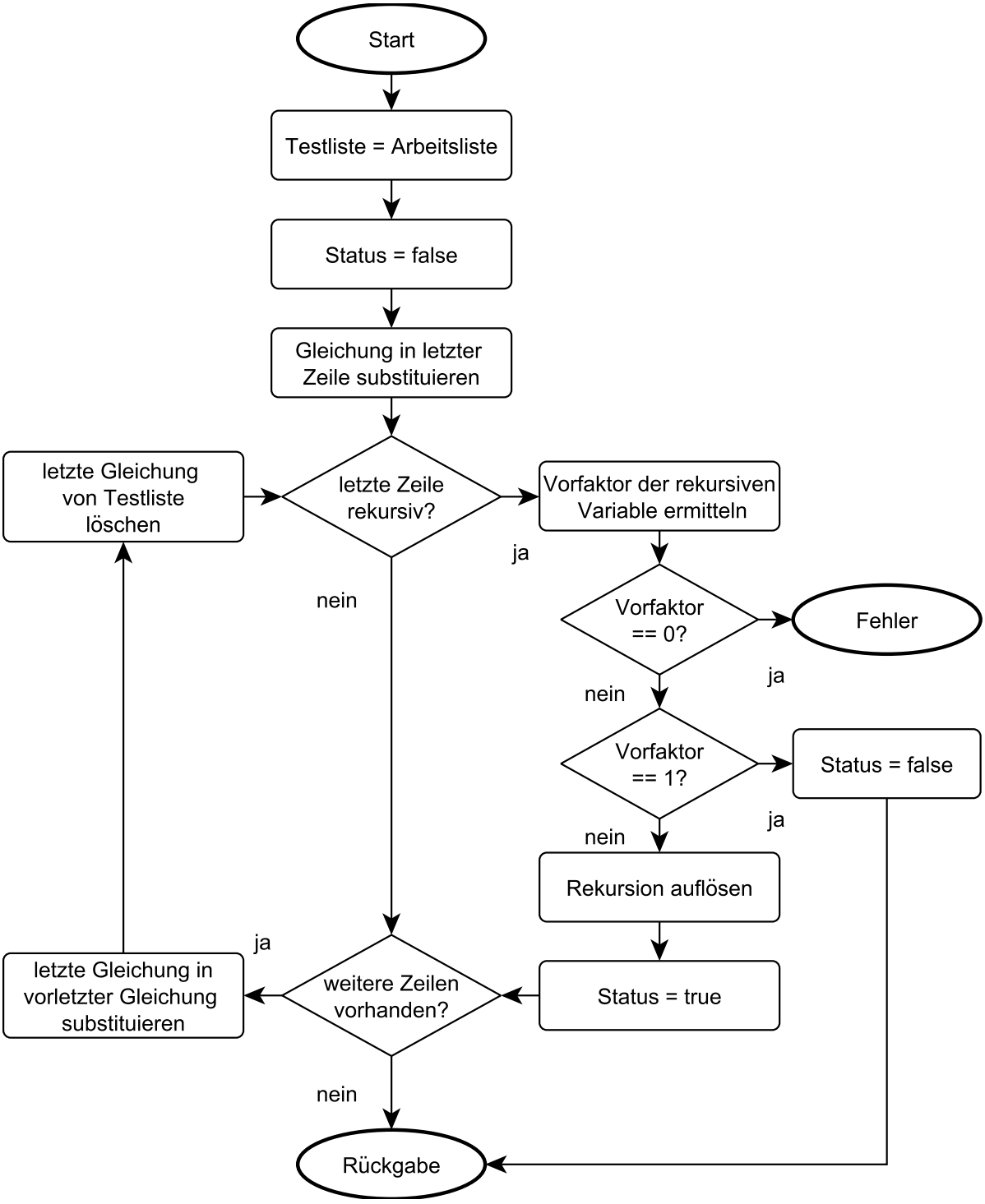


Abbildung 3.15.: Schematischer Ablauf der Simulation rekursiver Lösungen

keine vollständige Auslöschung von Variablen entstehen darf und in diesem Fall somit angenommen wird, dass der Ermittlungsprozess fehlerhaft ist.
Besitzt der Vorfaktor der rekursiven Variable einen Wert ungleich 0, wird die vorliegende Rekursion durch Umstellen der Gleichung aufgelöst und der Funktionsparameter *Status* auf *true* gesetzt. Die Simulationsfunktion wird jedoch erst nach der vollständigen Überprüfung der aktuellen Rücksubstitutionskette beendet, da unter Umständen weitere Rekursionen in dieser vorliegen könnten. Anschließend wird die resultierende Gleichung in den Gleichungsrumpf der vorletzte Zeile eingesetzt, welche sich auf der Testliste direkt oberhalb der aktuellen Zeile befindet. Die aktuelle letzte Zeile wird entfernt, wodurch die ehemals vorletzte Zeile zur aktuellen Zeile wird.

Der beschriebene Prozess wird solange wiederholt, bis die oberste Zeile der Testliste erreicht ist und keine Substitution in die darüberliegende Zeile möglich ist, wodurch die Simulation abgeschlossen ist und die Funktion mit Rückgabeparameter zurückkehrt.
Es ist zu beachten, dass bei der Vorbereitung der Probesimulation *Status* mit *false* initialisiert wird, wodurch eventuelle Simulationsläufe, in denen keine Rekursion gefunden wird, automatisch als Fehlschlag gewertet werden.

3.9.5.3. Umsetzung

Die Implementierung der gezielten Suche nach rekursiven Lösungen wird in der Funktion *ermitteln_rekursiv()*[45] durchgeführt, wofür die bereits in den vorherigen Abschnitten 3.9.5.1 und 3.9.5.2 beschriebenen Hilfsfunktionen grundlegend sind.
Es ist zu beachten, dass während der Ermittlung rekursiver Lösungen sämtliche Sperrungen von Variablen oder Gleichungen ignoriert werden.
Die Suche nach rekursiven Lösungen in der fünften Stufe des normalen Lösungsprozesses kann ebenfalls in mehrere Stufen unterteilt werden, deren Abfolge im nebenstehenden Schema 3.16 auf der nächsten Seite begleitend dargestellt ist, wobei in der ersten Stufe 5.1 die Grundliste analog zur zweiten Stufe des ursprünglichen Ermittlungsprozesses (siehe Abschnitt 3.9.2) nach Gleichungen durchsucht wird.

Die in Stufe 5.1 gefundenen Gleichungen werden auf deren grundsätzliche Eignung als rekursive Lösungen mit Hilfe der Funktion *ist_rekursiv_geeignet()*[46] überprüft. Die geeigneten Gleichungen werden nacheinander mit Hilfe der bereits beschriebenen Funktion *teste_rekursive_Substitution()*[47] als Lösung simuliert.

[45] *Quellcode: siehe Anhang 3.15 auf der vorherigen Seite*
[46] *Quellcode: siehe Anhang A.6.3 auf Seite 170*
[47] *Quellcode: siehe Anhang A.2.4 auf Seite 149*

Durchläuft eine Gleichung die Simulation erfolgreich, wird diese zurückgegeben und von *ermitteln_ Gleichung()*[48] als Lösung für die gesuchte Variable verwendet. Liefert die Durchsuchung der Grundliste keine geeigneten Gleichungen für die explizite Beschreibung der gesuchten Variable oder absolviert keine erfolgreich die Probesimulation, geht *ermitteln_ rekursiv()*[49] zur nächsten internen Ermittlungsstufe 5.2 über.

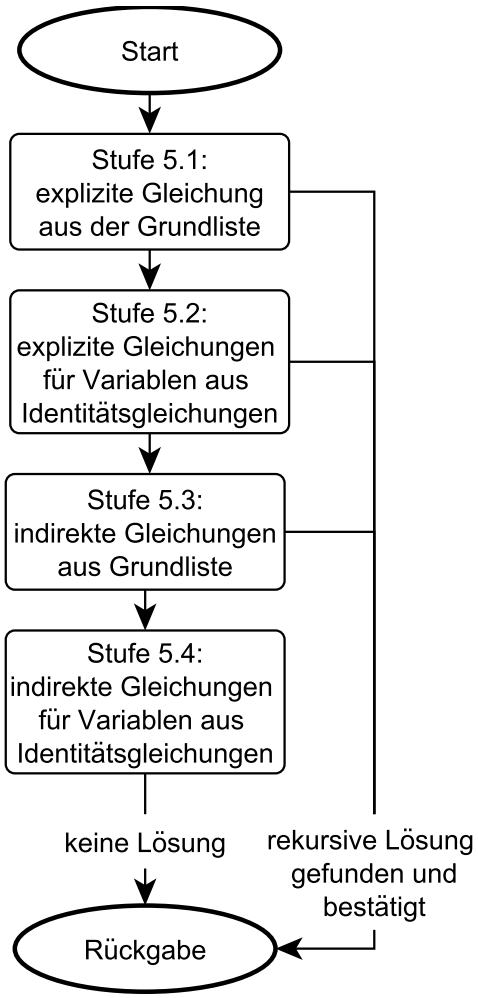


Abbildung 3.16.: Fünfte Stufe

Die folgende Stufe 5.2 entspricht konzeptionell der dritten Stufe des ursprünglichen Ermittlungsprozesses der Funktion *ermitteln_ Gleichung()*[50], wobei analog die mit der gesuchten Variablen identischen Variablen aus der Knotenliste geladen und untersucht werden.
Anschließend wird nacheinander für jede zutreffende Identitätsvariable die Grundliste nach Gleichungen durchsucht, welche diese Variable explizit beschreiben. Die gefundenen Gleichungen werden anschließend der gleichen Prozedur der vorherigen Stufe 5.1 unterzogen, wobei im Erfolgsfall *ermitteln_ rekursiv()* zum Caller zurückkehrt und die jeweilige Gleichung als Lösung zurückgibt, wenn eine explizite Gleichung der identischen Variablen die Testsimulation bestehen sollte.

Sollte in den Stufen 5.1 und 5.2 keine geeignete rekursive Lösung über die explizite Beschreibung der gesuchten Variable und der mit dieser identischen Variablen gefunden werden können, werden analog der ursprünglichen Ermittlungsabfolge aus *ermitteln_ Gleichung()* die Möglichkeiten von indirekten Lösungen untersucht.

Hierbei werden die Gleichungsrümpfe der Grundliste analog zu der vierten Stufe des Ermittlungsprozesses für normale Lösungen, welche in Abschnitt 3.9.4 erläutert ist, nach der gesuchten Variable beziehungsweise deren Identitätsvariable(n) durchsucht. Die gefundenen Gleichungen werden jeweils auf Eignung

[48] *Quellcode: siehe Anhang A.2.2 auf Seite 127*
[49] *Quellcode: siehe Anhang A.2.3 auf Seite 141*
[50] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

als rekursive Lösung von der Funktion *ist_rekursiv_geeignet()*[51] überprüft und im Fall eines positiven Ergebnisses nach der gesuchten Variable beziehungsweise derer Identitätsvariablen umgestellt. Die umgestellten Gleichungen werden anschließend jeweils der bekannten Testprozedur aus Abschnitt 3.9.5.2 zur Validierung von rekursiven Lösungen unterzogen.
Die Durchsuchung der Gleichungsrümpfe nach der gesuchten Variable analog der vierten Stufe entspricht der Stufe 5.3 des internen Funktionsprozesses zur Ermittlung von rekursiven Lösungen, wohingegen die Prozedur für die eventuell vorliegenden Identitätsvariablen die funktionsinterne Stufe 5.4 darstellt. Im Fall einer erfolgreichen Ermittlung, kehrt die Funktion mit Rückgabe der Gleichung wie in den Stufen 5.1 und 5.2 zum Caller zurück.

3.10. Verkettung von Knoten

Die Ermöglichung einer Modellierung und Verwendung von Bondgraphen in Matlab / Simulink ist der ursprüngliche Hauptgrund für Entwicklung der Toolbox. Ein weiterer Grund bestand in dem Ziel, basierend auf dieser allgemeinen Toolbox, weitere spezifische Toolboxen für die verschieden physikalischen Domänen abzuleiten, welchen das Konzept der Bondgraphen und deren Übertragung in den Zustandsraum zugrunde liegt.

Gegenüber der normalen Systemmodellierung in Simulink ist es notwendig, von der ursprünglichen signalflussorientierten zu einer objektbasierten Darstellung überzugehen, da Bondgraphen Systeme prinzipiell objektbasiert darstellen, wobei die grundlegenden Bondelemente nahezu direkt für die Darstellung domänenspezifischer Systeme verwendet werden können. Somit wäre lediglich eine Maskierung der Bondelemente notwendig.
Es ist jedoch zu beachten, dass zum Beispiel in elektrischen Schaltbildern Elemente teilweise direkt miteinander und nicht über Knoten verbunden sind, wodurch der elektrische Strom durch diese hindurch zum nächsten Elemente fließt. In einem Bondgraph würde der Flow, welcher direkt dem elektrischen Strom entspricht, von einem seriellen Knoten ausgehend in den Elementen enden. Dieser Darstellungsunterschied ist beispielhaft im Vergleich des Schaltplan des Beispielschwingkreises und dessen Bondgraph in Abbildung 3.2 auf Seite 49 veranschaulicht.

Dieses Problem trifft auf serielle Elementkonstellationen, wie auch auf parallele in sämtlichen physikalischen Domänen zu und kann durch die Verkettung von gleichartigen

[51] *Quellcode: siehe Anhang A.6.2 auf Seite 169*

Knoten mit jeweils einem einzelnen Element gelöst werden. Hierfür darf der jeweilige Knoten nur über einen Eingangsport und zwei Ausgangsports verfügen, da über den Eingangsport sowie den zweiten, nicht vom Element belegten, Ausgangsport die Verbindung zu dem nächsten gleichartigen Konstrukt oder regulären Element hergestellt werden muss.
Werden diese Element-Knoten-Konstrukte zusammen als ein physikalisches Element mit dem entsprechenden Symbol beziehungsweise Zeichen der domänenspezifischen Darstellungsform in Simulink maskiert, kann auf diese Art die "Durchleitung" von Signalen dargestellt werden, wodurch die jeweiligen Systeme in ihren üblichen Darstellungsformen modelliert und intern trotzdem durch Bondgraphenmethodik verarbeitet werden.

Da zwischen seriellen und parallelen Knoten die Rollen von Effort und Flow in den Bilanzen und Identitätsgleichungen lediglich vertauscht sind, werden in der Verkettung von 0- und 1-Knoten in dieser Hinsicht keine Unterscheidung getroffen und die demzufolge gleichartigen Verkettungsmechanismen direkt in den Ermittlungsprozess beziehungsweise in dessen Hilfsfunktionen integriert.
Die implementierten Mechanismen sind im folgenden Abschnitt 3.10.1 für Bilanzen und im Abschnitt 3.10.2 auf Seite 104 für Identitätsgleichungen beschrieben.

3.10.1. Bilanzen

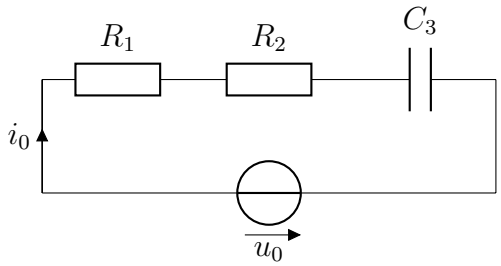
Die Verkettung von Bilanzen gleichartiger Knoten erfordert keine separate Behandlung, da diese bereits durch den Ermittlungsprozess selbst abgedeckt ist.

Die Ausgangsvariable eines Knotens ist im nachfolgend angeschlossenen Knoten gleichen Typs eine Eingangsvariable, da in der Aufbereitungsphase des Graphen die Identifikation der Bondvariablen und deren Indizierung anhand der Bonds erfolgt und nicht an die Elemente gebunden ist (siehe Abschnitt 3.3 auf Seite 52 sowie 3.5 auf Seite 63).
Hierdurch existiert die entsprechende Variable des Verbindungsbonds in Bilanzen der beider Knoten. Beim vorgelagerten Knoten kommt die Variable auf der Ausgangsseite und in der Eingangsseite der Bilanz des jeweils nachfolgenden Knoten vor.

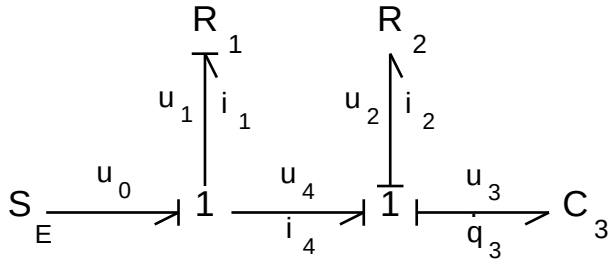
Es ist zu beachten, dass in der aktuellen Version der Toolbox keine unverbundenen Eingangs- oder Ausgangsports zulässig sind sowie weder die Anzahl der Eingangsports, noch Ausgangsports der Knoten variiert werden kann.

Die automatische Berücksichtigung von verketteten Bilanzen während des Ermittlungsprozesses soll anhand der folgenden Beispielschaltung von zwei mit einem Kondensator

in Reihe geschalteten Widerständen in Schaltplan 3.17a und des zugehörigen Bondgraph 3.17b mit verketteten 1-Knoten erläutert werden, wobei zur Demonstration lediglich die Zustandsgleichung für $\dot{q}_3$ zu ermitteln ist.



(a) Schaltplan



(b) Bondgraph

Abbildung 3.17.: Reihenschaltung

Es ist zu beachten, dass auf die Eintragung des Gesamtstroms i_0 im Bondgraph 3.17b verzichtet wurde, da dieser für die Ermittlung der Zustandsraumdarstellung nicht relevant ist und daher üblicherweise vernachlässigt wird.
Für den gegebenen Bondgraph ergeben sich an den beiden Knoten die folgenden zwei Spannungs- beziehungsweise Effortbilanzen, wobei in beiden Bilanzen zusätzlich die im Schaltplan nicht vorkommende Spannung u_4 des Verbindungsbonds enthalten ist.

$$u_0 = u_1 + u_4 \tag{3.24}$$

$$u_4 = u_2 + u_3 \tag{3.25}$$

Zusätzlich zu den Bilanzen 3.24 und 3.25 liegen die folgenden beiden Identitätsgleichungen der Ströme an den beiden Knoten vor, deren Verkettung im folgenden Abschnitt 3.10.2 beschrieben ist.

$$i_1 = i_4 \tag{3.26}$$

$$i_4 = i_2 = \dot{q}_3 \tag{3.27}$$

Des Weiteren existieren für die ohmschen Widerstände R_1 und R_2 sowie für den Kondensator C_3 die folgenden Elementgleichungen, wobei der Widerstands R_1 über Effortkausalität und die beiden anderen Elemente Flowkausalität besitzen.

$$i_1 = \frac{1}{R_1} * u_1 \tag{3.28}$$

$$u_2 = R_2 * i_2 \tag{3.29}$$

$$u_3 = \frac{1}{C_3} * q_3 \tag{3.30}$$

Die zu ermittelnde Zustandsableitung $\dot{q}_3$ ist nicht explizit durch eine Gleichung beschrieben, kann aber über die Identitäten 3.26 und 3.27 mit dem Strom i_1 gleichgesetzt werden. Aus diesem Grund wird die zugehörige Gleichung des Widerstandes R_1 substituiert für die Zustandsableitung $\dot{q}_3$. Anschließend wird für die Spannung u_1 die umgestellten Bilanz 3.24 eingesetzt:

$$\dot{q}_3 = i_1 = \frac{1}{R_1} * u_1 = \frac{1}{R_1} * (u_0 - u_4) \tag{3.31}$$

An dieser Stelle wird durch die weitere Ermittlung automatisch die Verkettung zwischen den Bilanzen durchgeführt. Für die unbekannte Spannung u_4, welche die Verbindungsvariable zwischen den beiden Knoten darstellt, kann nur noch die zweite Bilanz 3.25 eingesetzt werden, da Gleichung 3.24 zur Vermeidung von algebraischen Schleifen bei der Verwendung gesperrt wurde.

$$\dot{q}_3 = \frac{1}{R_1} * (u_0 - u_2 - u_3) \tag{3.32}$$

Die Quellspannung u_0 ist eine Eingangsvariable und die Kondensatorspannung u_3 hängt ausschließlich von der Zustandsvariable q_3 ab, wodurch beide Variablen bekannt sind und entsprechend als Lösung verwendet werden können. Hierdurch muss lediglich die

Spannung u_2 ermittelt werden, wobei $\dot{q}_3$ anschließend mit Hilfe von i_2 unter Anwendung der Identität 3.27 rekursiv gelöst wird:

$$\dot{q}_3 = \frac{1}{R_1} * (u_0 - u_2 - \frac{1}{C_3} q_3) \tag{3.33}$$

$$= \frac{1}{R_1} * (u_0 - R_2 * i_2 - \frac{1}{C_3} q_3) \tag{3.34}$$

$$= \frac{1}{R_1} * (u_0 - R_2 * \dot{q}_3 - \frac{1}{C_3} q_3) \tag{3.35}$$

$$\dot{q}_3 - \frac{R_2}{R_1}\dot{q}_3 = \frac{1}{R_1} * (u_0 - \frac{1}{C_3} q_3) \tag{3.36}$$

$$\frac{R_1 + R_2}{R_1} * \dot{q}_3 = \frac{1}{R_1} * (u_0 - \frac{1}{C_3} q_3) \tag{3.37}$$

$$\dot{q}_3 = \frac{1}{R_1 + R_2} * (u_0 + \frac{1}{C_3} q_3) \tag{3.38}$$

3.10.2. Identitätsgleichungen

Die Verkettung von Identitätsgleichungen aufeinanderfolgender Knoten gleichen Typs wird in der Toolbox durch die Funktion *ermitteln_ Gleichnisvariablen()*[52] durchgeführt, deren Aufgabe darin besteht, zu der an diese Funktion übergebene Variable die zugehörigen identischen Variablen gemäß den Verknüpfung aus der Knotenliste zu ermitteln und zurückzugeben.

Die Funktion durchsucht hierfür die Identitätsgleichungen sämtlicher Knoten auf Vorkommen der übergebenen Variable und speichert die mit der übergebenen Variable identischen Variablen aus den zutreffenden Identitäten sowie die Elementnummern der zugehörigen Knoten.

Wird mindestens eine Identitätsgleichung gefunden, welche die angeforderte Variable enhält, werden die restlichen in dieser Identitätgleichung enthaltenen Variablen mit denen restlichen Identitätsgleichungen der bisher nicht verwendeten Knoten auf Übereinstimmung verglichen.
Kommt eine Variable aus der ursprünglichen Identitätsgleichung in einer weiteren Identität vor, muss im jeweiligen Fall eine direkte Verbindung zweier Knoten gleichen Typs vorliegen. Da die jeweilige in beiden Identitätsgleichungen vorkommende Variable dem Verbindungsbond zwischen den beiden Knoten zugehörig sein muss, liegt folglich eine

[52] *Quellcode: siehe Anhang C.3.6 auf Seite 220*

Verkettung dieser beiden Knoten vor und die Identitätsgleichungen der betreffenden Knoten müssen als eine zusammenhängende Identitätsgleichung interpretiert werden.
Liegt eine derartige Verkettung mehrerer Knoten vor, müssen die Variablen der angebundenen Identitätsgleichung ebenfalls mit der ursprünglichen Identitätsgleichung der übergebenen Variable übereinstimmen. Diese Variablen werden mit den bereits in der Funktion *ermitteln_ Gleichnisvariablen()*[53] gespeicherten Variablen zusammengefasst, wobei eventuelle auftretende Dopplungen gefundenen Variablen beseitigt werden.
Die Prozedur wird solange von Knoten zu Knoten wiederholt, bis sich keine weiteren Verbindungen zu ungenutzten Identitätsgleichungen ergeben oder sämtliche Knoten des Graphen beziehungsweise deren identische Variablen zusammengefasst sind. Anschließend werden die funktionsintern gespeicherten Variablen als Liste an den Caller von *ermitteln_ Gleichnisvariablen()* zurückgegeben.

Durch dieses Vorgehen werden von der Funktion alle Variablen mit identischen Lösungen für die an die Funktion übergebene Variable zurückgegeben, wobei dieses Vorgehen für den eigentlichen Ermittlungsprozess beziehungsweise den Caller von *ermitteln_ Gleichnisvariablen()* nicht sichtbar ist. Da die durch Verkettung identischen Variablen zusammengefasst in einem Vektor zurückgegeben werden, wirkt die Rückgabe aufgrund für die aufrufenden Instanz intransparenten Prozedur wie eine virtuelle Gesamtidentitätsgleichung
Die Verkettung der Identitäten von Knoten gleichen Typs ist ebenfalls durch Verwendung von *ermitteln_ Gleichnisvariablen()* in den Vorstufen des Ermittlungsprozesses sowie der Aktualisierung der Identitätsvariablen durch *aktualisieren()*[54] berücksichtigt.

Die Verkettung von Bilanzen findet ebenfalls im Beispiel des vorherigen Abschnittes 3.10.1 statt, wobei die Identitäten 3.26 und 3.27 bei Abfrage der in diesen enthaltenen Variablen über den Strom i_4 verbunden werden.

In der aktuellen Toolboxversion wird der Überprüfungs- und Verknüpfungsprozess bei jedem Aufruf von *ermitteln_ Gleichnisvariablen()* vollständig ausgeführt, wobei in späteren Versionen eine Verbesserung der Performanz durch eine statische Verkettung in den Vorstufen erzielt werden könnte, was in Abschnitt 4.1 für eine eventuelle Weiterführung angerissen ist.

[53] *Quellcode: siehe Anhang C.3.6 auf Seite 220*
[54] *Quellcode: siehe Anhang A.4.1 auf Seite 157*

4. Abschluss

4.1. Weiterführung

Im Zuge der Entwicklung der vorliegenden Toolboxversion für die Implementierung von Bondgraphen in Matlab / Simulink wurden verschiedene Verbesserungs- und Erweiterungsvorschläge sowie Nachbesserungsnotwendigkeiten zusammengetragen.
Die gesammelten Vorschläge sind in den nachfolgenden Unterabschnitten als Vorbereitung für eine eventuelle Weiterentwicklung der Toolbox beschrieben.

Flowsource an seriellen Knoten und Effortsource an parallelen Knoten

In der aktuellen Version können Quellen nur an Knoten angeschlossen und charakterisiert werden, wenn die Quellvariable in der Bilanz des jeweils nachfolgenden Knotens vorkommt. Hierdurch können lediglich Effortquellen an seriellen Knoten und Flowquellen an parallelen Knoten verwendet werden.
Um die Anbindung von Quellen an Knoten, bei denen die Quellvariablen in den Identitätsgleichungen der jeweiligen Knoten vorkommen würden, zu ermöglichen, müsste die Erstellung der Bondgleichungen sowie die Konflikterkennung (siehe Abschnitte 3.5 bis 3.7) überarbeitet werden. Im Fall von mehreren gleichartigen Quellen in der Identitätsgleichung eines Knotens würden andernfalls Lösbarkeitskonflikte entstehen, da jede Quellvariable eine eigenständige Lösung darstellt und somit eine andere Handhabung erforderlich würde.

Lösungskonflikt unter Beteiligung von Quellvariablen

Die Korrektur von uneindeutigen Lösungen in Identitätsgleichung von Knoten durch mehrere Energiespeicher, deren Energievariable ein Teil der jeweiligen Identität darstellt, ist in der aktuellen Version der Toolbox bereits implementiert.
Der in diesen Fällen verwendete Algorithmus ist in Abschnitt 3.7.2 beschrieben, umfasst jedoch nicht die Korrektur unter Beteiligung von Quellvariablen als Bestandteil der betroffenen Identitätsgleichung. Die Korrektur von uneindeutigen Lösungen durch

eine oder mehrere Quellvariablen und der Energievariable eines Speicherelementes ist ebenfalls noch nicht in den Vorstufen der Toolbox implementiert

Quellen direkt an Senken

Die Toolbox schließt direkte Verbindungen zwischen Quellen und Elementen aus, da Quellen in der vorliegenden Version lediglich an Knoten angeschlossen werden dürfen. Die Festlegung des Quellentyp erfolgt aktuell ausschließlich über die Kausalität des Eingangsbond des nachfolgenden Knotens.

Erweiterung des Sperrmechanismus

Die aktuellen Sperrmechanismen der Variablen und Gleichungen wirken lediglich auf die erste bis vierte Stufe des Ermittlungsprozesses und nicht auf dessen fünfter Stufe. Die Implementierung eines analogen, aber parallel zum Ersten arbeitenden Sperrmechanismus für gezielte Ermittlung rekursiver Lösungen in der fünften Stufe könnte den Rechenaufwand erheblich reduzieren, da es in der fünften Ermittlungsstufe in der aktuellen Toolboxversion teilweise zu redundanten Lösungsversuchen aufgrund fehlender Sperrungen kommen kann.

Verkettung von Knoten während der Initialisierung

In der aktuellen Version der Toolbox werden verkettete Bilanzen und Identitätsgleichungen von aufeinanderfolgenden Knoten gleichen Typs, wie in den Abschnitten 3.10 und 3.10.2 beschrieben, behandelt. Eine Reduzierung des Rechenaufwandes könnte durch eine Modifikation der Knotenliste vor der Erstellung der Bondgleichungen und dem Beginn der Vorstufen erzielt werden, indem die Verkettungen zwischen gleichartigen Knoten identifiziert und die jeweils verketteten Knoten zu virtuellen Gesamtknoten zusammengefasst werden.
Durch Eintragung der resultierenden Gesamtknoten und Entfernung der diesen zugrundeliegenden Einzelknoten auf der Knotenliste, würde die Verkettungsprozedur in der Hilfsfunktion *ermitteln_ Gleichnisvariablen()*[1] obsolet werden.

Aktive Bondelemente

Effort- und Flowquellen sind in der aktuellen Version die einzigen aktiven Elemente der Toolbox, welche Energie in das jeweilige System einspeisen können.
Hierdurch ist es nicht möglich Systeme mit aktiven Zwischenstufen wie zum Beispiel elektrische Verstärkerschaltungen oder Steuerungen von Motoren mit resultierender Wirkung

[1] *Quellcode: siehe Anhang C.3.6 auf Seite 220*

auf nachfolgende Systeme zu modellieren. Um dies zu ermöglichen müssten entsprechende aktive und steuerbare Zwischenelemente der Toolbox hinzugefügt werden.

Verwendung von Verhältnisgleichungen

Die Ermittlung von gesuchten Variablen basiert in der aktuellen Version lediglich auf Substitution von zu ermittelnden Variablen durch bereits bekannte Gleichungen gemäß Elementgleichungen und Knotengleichen.
Der Einbeziehen von Verhältnisgleichungen analog Spannungs- und Stromteilergleichungen für elektrische Reihen- und Parallelschaltungen könnte gegebenenfalls zu einer Reduzierung des Rechenaufwandes für die Ermittlung von Effortvariablen an seriellen 1-Knoten beziehungsweise von Flowvariablen an parallelen 0-Knoten führen.

Vorstufe für Kausalität

Der Rechenaufwand für die Ermittlung des Zustandsraummodells könnte durch die Implementierung einer zusätzlichen Vorstufe zur Verifizierung der Kausalitäten der einzelnen Bonds und deren eventuelle Korrektur reduziert werden, da hierdurch die gegebenenfalls notwendige Suche nach indirekten Lösungsansätzen vermieden werden könnte, was die vierte Stufe des regulären Ermittlungsprozesses der Funktion *ermitteln_ Gleichung()*[2] darstellt und in Abschnitt 3.9.4 auf Seite 88 beschrieben ist.
Des Weiteren könnte diese zusätzliche Vorstufe zur Realisierung einer fehlerfreien und automatische Kausalitätsfestlegung für alle Bonds genutzt werden, wodurch der Nutzer weiter entlastet werden könnte.

Subsystemeübergänge und Maskierung

Die Identifikation und Verbindung der Elemente sowie der Quellen des Bondgraphen geschieht mit Hilfe der Handles der internen Runtime-Objekte der Simulation. Hierdurch kann der jeweilige Bondgraph jedoch nicht über die Grenzen von Subsystem hinaus beziehungsweise zwischen Subsystemen korrekt erfasst werden, da die Input- und Outputblöcke der Subsystemports über jeweils eigene Handles verfügen und somit von der Toolbox als Ein- und Ausgänge des Graphen erkannt werden.
Beispielsweise wird ein Subsystemübergang am Eingang eines Bondelementes als Quellelement identifiziert. Ein derartiger Systemübergang würde am Ausgangsbond eines Elementes nicht als gültiges Zielelement erkannt werden.
Die Verwendung von Subsystemen mit Maskierung ist jedoch notwendig, um eine Abwandlung der Toolbox für die verschiedenen physikalischen Domänen zu ermöglichen.

[2] *Quellcode: siehe Anhang A.2.2 auf Seite 127*

Bezeichnungen des Graphen in Simulink

Die Indizierung von Bonds sowie Elementen geschieht in der aktuellen Version unabhängig von den Bezeichnungen der Elemente in Simulink. Hierdurch ist der Zusammenhang zwischen dem in Simulink modelliertem Bondgraph und den von der Toolbox festgelegten Bondvariablen, den aufgestellten Bondgleichungen und somit den Bezeichnern des resultierenden Zustandsraummodells nicht direkt gegeben.
Diese Problematik würde durch das Anzeigen der Bond- und Elementbezeichnungen im Simulinkmodell beseitigt werden, was die Modellierung sowie die Handhabung des Graphen erleichtern.

Einbettung des Zustandsraummodells in Simulink

Das Zustandsraummodell des Bondgraphen wird im *base*-Workspace von Matlab abgelegt. Es erfolgt jedoch nach Abschluss des Ermittlungsprozesses keine weitere Interaktion mit dem Zustandsraummodell in Simulink, wobei die Simulinksimulation fortgeführt wird und keine auf dem Bondgraph basierende Ergebnisse produziert werden. In der aktuellen Version der Toolbox kann der Bondgraph anschließend nur über dessen im Workspace von Matlab hinterlegten Zustandsraummodell unter Verwendung zusätzlicher Matlabanweisungen manuell durch den Benutzer simuliert werden.
Da die Erstellung der Zustandsraumdarstellung jedoch nach der Initialisierungsphase des Simulinkmodells abgeschlossen ist, könnte eine parallele Simulation des Zustandsraummodells realisiert werden, wobei die Eingangswerte beziehungsweise -signale direkt aus den angeschlossenen Quellblöcken von Simulink verwendet und die Ausgangswerte synchron mit dem Fortschreiten der Simulation an den Ausgängen des Bondgraphen zurückgegeben werden könnten.

Mehrfacheingänge an Knoten

In der aktuellen Version der Toolbox verfügen Knoten über lediglich einen Eingangsport. Daher können Querverbindungen zwischen mehreren Knoten nicht umgesetzt werden und ein Bondgraph des Weiteren über lediglich ein einziges Quellelement als Eingang verfügen, wodurch auch die Länge des Eingangsvektors $\vec{u}$ auf ein Element beschränkt ist, da in beiden Fällen mindestens ein Knoten mehr als einen Eingang aufweisen muss. Um in späteren Toolboxversionen mehrere Eingangsvariablen an einem Knoten mit dem Darstellungsprinzip der Grundliste, welche lediglich eine explizite Variable zur Identifikation und einen zugehörigen Gleichungsrumpf erlaubt, vereinbaren zu können, müssen sämtliche eingehenden Bilanzvariablen der Knotens auf die Ausgangsseite der Bilanz subtrahiert werden. Die eingehende Bilanzvariable mit dem niedrigsten Index soll hier-

von jedoch ausgenommen sein und verbleibt als explizite Variable auf der ehemaligen Eingangsseite der Bilanz, wobei die hierdurch resultierende Bilanz mit den Darstellungen der Gleichungslisten kompatibel ist und entsprechend auf der Grundliste bei der Erstellung der Bondgleichungen eingefügt wird.

Einzelauswahl der Ausgangsvariablen

Die Auswahl der Effort- und Flowvariablen des Ausgangsvektors $\vec{y}$ ist in der aktuellen Toolboxversion lediglich paarweise für die jeweiligen Bonds möglich. Die Einzelauswahl der gewünschten Bondvariablen sollte mit geringen Aufwand in nachfolgenden Versionen der Toolbox ähnlich der momentanen Markierungsverfahren über die GUI der jeweiligen Elemente und geringen Modifikationen der S-Function des Elemente sowie der Funktion *erstelle_Ausgangsvektor()*[3] realisiert werden können.

4.2. Fazit

Die aktuelle Implementierung der Toolbox erfüllt die in Abschnitt 1.3 definierten Anforderungen an Funktionsumfang und Bedienbarkeit, wodurch die Modellierung von Bondgraphen in Simulink sowie deren weiterführende Analyse aufgrund der generierten Zustandsraumdarstellung möglich ist.

Für die Installation der Toolbox genügt lediglich das Einbinden des Dateipfades der Toolboxdateien in Matlab, wobei für deren Verwendung keine weiteren Programme, Bibliotheken und Erweiterungen benötigt werden, welche über die Grundinstallation von Matlab / Simulink hinausgehen.
Der Benutzer benötigt für die Anwendung der Toolbox keine tiefgreifenden Kenntnisse der zu modellierenden Systeme in Bezug auf deren dynamisches Verhalten und internen mathematischen beziehungsweise physikalischen Zusammenhängen, da aufgrund der Bondgraphenmethodik lediglich die Identifikation der physikalischen Elemente beziehungsweise deren Eigenschaften für die Modellierung von Systemen notwendig ist.
Gleiches gilt für die eigentliche Erstellung des Graphen in Simulink, wobei sich die Aufgabe des Benutzers ausschließlich auf die strukturelle Verkettung sowie Parametrisierung der Bondelemente in Bezug auf die Leistungsübertragung zwischen diesen beschränkt. Die weitere Aufbereitung des Bondgraphen und dessen internen Bondgleichungen sowie die Ermittlung des auf diesen basierenden Zustandsraummodells wird vollständig und automatisch von der Toolbox übernommen.

[3] *Quellcode: siehe Anhang B.1.1 auf Seite 174*

Des Weiteren werden gegebenenfalls vorliegende Konflikte der Lösbarkeit des Gleichungssystems und / oder der Eindeutigkeit dieser Lösungen aufgrund fehlerhafter festgelegter Bondkausalitäten von der Toolbox ohne Beteiligung des Benutzers autonom detektiert und korrigiert beziehungsweise umgangen.

Obwohl die Toolbox die grundlegenden Anforderungen erfüllt und grundsätzlich funktionstüchtig ist, sollten im Rahmen einer Projektweiterführung die im vorherigen Abschnitt 4.1 beschriebenen Verbesserungen umgesetzt werden.
Um die Nutzbarkeit der Toolbox zu verbessern, müssen an erster Stelle die Bezeichner der Bondvariablen im Simulinkmodell des Graphen dargestellt werden. In der vorliegenden Implementierung ist der Zusammenhang zwischen den eigentlichen Variablen des modellierten Bondgraphen und den Variablenbezeichnungen des ermittelten Zustandsraummodells für den Benutzer nur indirekt nachvollziehbar, da diese Bezeichner im Graph nicht angezeigt werden und deren Vergabeprozedur der Variablenbezeichner sowie die Nummerierung der einzelnen Bonds für den Benutzer nicht intransparent ist.
Des Weiteren nimmt die Erstellung der Zustands- und Ausgangsgleichungen des Graphen aufgrund teilweise redundanter Lösungsschritte während der Ermittlung unnötig viel Zeit in Anspruch, wodurch eine Optimierung des Lösungsalgorithmus und dessen Sperrmechanismen naheliegt.

Die erstellten Zustandsraumdarstellungen werden als matlabeigener Datentyp *ss* ausgegeben, wodurch deren weitere Verwendung nicht durch die Toolbox limitiert ist. Hierdurch ist eine Implementierung einer parallelen Simulation dieser Modelle im zugrundeliegenden Simulinkmodell prinzipiell möglich, wobei nach deren Umsetzung die Toolbox als Alternative zu den bereits existierenden Lösungen gesehen werden könnte.

Das Potential der Toolbox wird aufgrund des beschriebenen Nachbesserungsbedarfs zum jetzigen Zeitpunkt nicht vollständig ausgeschöpft. Die Vorteile der Modellierung von dynamischen Systemen beliebiger physikalischer Domänen mit Hilfe dieser Bondgraphentoolbox in Matlab / Simulink zeichnen sich jedoch trotzdem ab, wodurch die aktuelle Implementierung als Ausgangspunkt für anschließende und auf dieser Implementierung aufbauende Projekte angesehen werden kann.

Teil II.

Toolboxfunktionen

A. Analysator

A.1. Vorstufe

A.1.1. ersetze_Bezeichnungen()

```
function [  ] = ersetze_Bezeichnungen( )
%ERSETZE_Bezeichnungen
%
%   Ersetzt alle Effort- und Flowvariablen mit den zugehörigen Ableitungen
%   der Zustandsvariablen bzw. Eingangsvariablen, wie die in den Vektoren
%   dx und u vorhanden sind.
%
%   Die Ersetzung wird in folgenden Segmenten vorgenommen:
%   - Grundliste
%   - Ergebnisliste
%   - Gleichnisse in Knotenliste
%
%   Die Funktion muss vor der ersten generellen Aktualisierung ausgeführt
%   werden.
%
%   Voraussetzungen: - global: Knoten-, Grundliste & Ergebnisliste
%                    - globale Vektoren dx und u

%% Vorbereitung
global Knotenliste
global Grundliste
global Ergebnisliste
global dx
global u

if isempty(Knotenliste)
    error('Knotenliste existiert nicht!');
elseif isempty(Grundliste)
    error('Grundliste existiert nicht!');
```

```
elseif isempty(Ergebnisliste)
    error('Ergebnisliste existiert nicht!');
elseif isempty(u)
    error('Vektor u existiert nicht!');
end

if isempty(dx)
    warning('Vektor dx existiert nicht!');
end

%% Eingangsvariablen einsetzen...
for Index = 1:length(u)
    Eingangsvariable = u{Index}
    Zielvariable = lower(Eingangsvariable);
    VarID = num2str(ermittleln_VarID(Zielvariable));

    if strcmp(ermitteln_VarSymbol(Eingangsvariable),'E')
        Pseudovariable = [ 'f' VarID ];
    elseif strcmp(ermitteln_VarSymbol(Eingangsvariable),'F')
        Pseudovariable = [ 'e' VarID ];
    end

    % ... in Grundliste ersetzen
    for Zeile = 1:length(Grundliste(:,1))
        Grundliste{ Zeile, 2 } = subs( Grundliste{Zeile,2}, ...
                Zielvariable, Eingangsvariable, 0);
        Grundliste{ Zeile, 3 } = subs( Grundliste{Zeile,3}, ...
                Zielvariable, Eingangsvariable, 0);
        Grundliste{ Zeile, 2 } = subs( Grundliste{Zeile,2}, ...
                Pseudovariable, 0, 0);
        Grundliste{ Zeile, 3 } = subs( Grundliste{Zeile,3}, ...
                Pseudovariable, 0, 0);
    end

    % ... in Ergebnisliste ersetzen
    for Zeile = 1:length(Ergebnisliste(:,1))
        Ergebnisliste{ Zeile, 2 } = subs( Ergebnisliste{Zeile,2}, ...
            Zielvariable, Eingangsvariable, 0);
        Ergebnisliste{ Zeile, 3 } = subs( Ergebnisliste{Zeile,3}, ...
            Zielvariable, Eingangsvariable, 0);
        Ergebnisliste{ Zeile, 2 } = subs( Ergebnisliste{Zeile,2}, ...
            Pseudovariable, 0, 0);
```

```
        Ergebnisliste{ Zeile, 3 } = subs( Ergebnisliste{Zeile,3}, ...
            Pseudovariable, 0, 0);
    end

    % ... in "Inputs" der Knoteliste ersetzen
    [ maxZeilen maxSpalten ] = size(Knotenliste.Inputs);
    for Zeile = 1:maxZeilen
        for Spalte = 1:maxSpalten
            if strcmp(Knotenliste.Inputs{Zeile,Spalte},Zielvariable)
                Knotenliste.Inputs{Zeile,Spalte} = Eingangsvariable;
            elseif strcmp(Knotenliste.Inputs{Zeile,Spalte},Pseudovariable)
                Knotenliste.Inputs{Zeile,Spalte} = '0';
            end
        end
    end

    % ... in "Outputs" der Knoteliste ersetzen
    [ maxZeilen maxSpalten ] = size(Knotenliste.Outputs);
    for Zeile = 1:maxZeilen
        for Spalte = 1:maxSpalten
            if strcmp(Knotenliste.Outputs{Zeile,Spalte},Zielvariable)
                Knotenliste.Outputs{Zeile,Spalte} = Eingangsvariable;
            elseif strcmp(Knotenliste.Outputs{Zeile,Spalte},Pseudovariable)
                Knotenliste.Outputs{Zeile,Spalte} = '0';
            end
        end
    end

    % in "Gleichnisse" der Knoteliste ersetzen
    [ maxZeilen maxSpalten ] = size(Knotenliste.Gleichnisse);
    for Zeile = 1:maxZeilen
        for Spalte = 1:maxSpalten
            if strcmp(Knotenliste.Gleichnisse{Zeile,Spalte},Zielvariable)
                Knotenliste.Gleichnisse{Zeile,Spalte} = Eingangsvariable;
            elseif strcmp(Knotenliste.Gleichnisse{Zeile,Spalte}, ...
                    Pseudovariable)
                Knotenliste.Gleichnisse{Zeile,Spalte} = '0';
            end
        end
    end
end
```

```
%% Zustandsableitungen einsetzen
for Index = 1:length(dx)
    Zustandsableitung = dx{Index};
    VarID = num2str(ermittleln_VarID(Zustandsableitung));

    if strcmp(ermitteln_VarSymbol(Zustandsableitung),'dp')
        Zielvariable = [ 'e' VarID ];
    elseif strcmp(ermitteln_VarSymbol(Zustandsableitung),'dq')
        Zielvariable = [ 'f' VarID ];
    end

    % in Grundliste ersetzen
    for Zeile = 1:length(Grundliste(:,1))
        Grundliste{Zeile,2} = subs( Grundliste{ Zeile,2}, ...
            Zielvariable, Zustandsableitung );
        Grundliste{Zeile,3} = subs( Grundliste{ Zeile,3}, ...
            Zielvariable, Zustandsableitung );
    end

    % in Ergebnisliste ersetzen
    for Zeile = 1:length(Ergebnisliste(:,1))
        Ergebnisliste{ Zeile, 2 } = subs( Ergebnisliste{ Zeile, 2}, ...
            Zielvariable, Zustandsableitung);
        Ergebnisliste{ Zeile, 3 } = subs( Ergebnisliste{Zeile, 3}, ...
            Zielvariable, Zustandsableitung);
    end

    % in Inputs der Knoteliste ersetzen
    [ maxZeilen maxSpalten ] = size(Knotenliste.Inputs);
    for Zeile = 1:maxZeilen
        for Spalte = 1:maxSpalten
            if strcmp(Knotenliste.Inputs{Zeile,Spalte},Zielvariable)
                Knotenliste.Inputs{Zeile,Spalte} = Zustandsableitung;
            end
        end
    end

    % in Outputs der Knoteliste ersetzen
    [ maxZeilen maxSpalten ] = size(Knotenliste.Outputs);
    for Zeile = 1:maxZeilen
        for Spalte = 1:maxSpalten
            if strcmp(Knotenliste.Outputs{Zeile,Spalte},Zielvariable)
```

```
                Knotenliste.Outputs{Zeile,Spalte} = Zustandsableitung;
            end
        end
    end

    Knotenliste.Gleichnisse ( Knotenliste.Gleichnisse == ...
        sym(Zielvariable) )  = dx(Index,1);
end
end
```

A.1.2. korrigiere_Causality()

```
function [  ] = korrigiere_Causality(  )
%KORRIGIERE_CAUSALITY()
%
%   Löst für jeden einzelnen Knoten die Überprüfung für das jeweilige
%   Gleichnis aus, um Konflikte in den Gleichnissen und Kausalitätsprobleme
%   zu beseitigen!

if nargin > 0
    error('Zuviele Eingabeparameter!')
end

global Knotenliste

if isempty(Knotenliste)
    error('Knotenliste ist leer... dürfte nicht sein!')
end

for i = 1:1:length(Knotenliste.ID)
    korrigiere_Gleichnis(Knotenliste.ID(i));
    %pause
end

end
```

A.1.3. korrigiere_Gleichnis()

```
function [ Aenderung ] = korrigiere_Gleichnis( KnotenID )
%   Korrigiert uneindeutige Lösungen während Initialphase im Fall von
%   mehreren gleichartigen Speicher deren Energievariablen im Gleichnis des
%   zugehörigen Knotens angeschlossen sind.
%
%   Fall 1: mindestens 2 Capacitors an parallelen 0-Knoten
%   Fall 2: mindestens 2 Inertia an seriellen 1-Knoten

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl der Eingabeparameter!')
end

global Knotenliste
global Ergebnisliste
global Grundliste

Zeile = Knotenliste.ID == KnotenID;

if sum(Zeile) > 1
    error('Es wurden MEHR ALS EIN Knoten mit KnotenID %d gefunden!',KnotenID);
elseif sum(Zeile) < 1
    error('Es wurden KEIN Knoten mit KnotenID %d gefunden!',KnotenID);
end

% Initialisierungen
bekannteVariablen = {};
Aenderung = [];
Nummern = [];
diffGleichung = sym([]);


%% Variablenlisten aufstellen
Testvariable = Knotenliste.Gleichnisse(Zeile, ...
    Knotenliste.Gleichnisse(Zeile,:) ~= sym('0') );
gleicheVariablen = ermitteln_Gleichnisvariablen(Testvariable(1,1));

for i = 1:length(gleicheVariablen)

    if gleicheVariablen(i,1) == sym('0')
```

```
        continue
    end

    [ aktuelleGleichung aktuelleNummer aktuelleBemerkung ] = ...
        durchsuche_Ergebnisliste( gleicheVariablen(i,1) );

    % Gleichung eintragn
    if ~isempty(aktuelleGleichung)
        bekannteVariablen{end+1, 1} = aktuelleNummer;
        bekannteVariablen{end, 2} = sym(gleicheVariablen(i,1));
        bekannteVariablen{end, 3} = aktuelleGleichung;
        bekannteVariablen{end, 4} = aktuelleBemerkung;
    end
end

%% Abbruch 1: keine bekannte Variable im Gleichnis --> kein Konflikt
if isempty(bekannteVariablen) % Alles unbekannt... nix geht!
    fprintf('Keine bekannte Variable im Gleichnis bekannt!\nRETURN\n');
    return
end

%% Überprüfen und Eintragen in EL
Konfliktvariablen = teste_Korrektheit_Gleichnis( bekannteVariablen );

if isempty(Konfliktvariablen)
    disp('Keine Konflikte auf Ergebnisliste!');
    return
end

%% Zustandsvariablen ermitteln, Ableitungen erstellen
for i = 1:1:length(Konfliktvariablen(:,1))

    [ Konfliktgleichung(i,1) diffGlNummer diffBemerkung ] = ...
        durchsuche_Ergebnisliste( Konfliktvariablen(i,1) );
    Subvariablen = transpose(symvar( Konfliktgleichung(i,1) ) );

    % Subvariabeln einzeln durchgehen...
    for j = 1:length( Subvariablen )

        [ Symbol ] = ermitteln_VarSymbol( Subvariablen(j) )

        % Zustandsvariable aus Gleichung ermitteln
```

```
        if ( strcmp(Symbol,'p') || strcmp(Symbol,'q') )
            Zustandsvariable(i,1) = Subvariablen(j);
            break
        elseif ( strcmp(Symbol,'E') || strcmp(Symbol,'F') )
            error(['Konfliktlösung unter Beteiligung von Quell', ...
                'variablen in Konfliktgleichnis ist in dieser ', ...
                'Version nicht implementiert!']);
        end
    end

    % Gleichung in differentielle Form umstellen und auf GL eintragen
    diffGleichung(i,1) = umstellen_Gleichung( diffGlNummer, ...
        Zustandsvariable(i,1), Ergebnisliste, 'Ergebnisliste' );
    Zustandsableitung(i,1) = sym([ 'd' char( Zustandsvariable( i, 1 ) ) ]);

    if i < length(Konfliktvariablen(:,1))
        Nummern(end+1,1) = erstelle_Gleichung( Zustandsableitung(i,1), ...
            diffGleichung(i,1), 'differentiell', 'Grundliste' );
    end
end

for i = 1:1:length(Konfliktvariablen(:,1))-1

    Zeile = Grundliste(:,2) == Zustandsableitung(i,1);

    disp('Differentielle Gleichung mit Folgegleichung substituieren!')
    Grundliste{Zeile, 3} = subs( Grundliste{Zeile, 3}, ...
        Konfliktvariablen(i,1), Konfliktgleichung(i+1,1), 0);
    anzeigen_Liste(Grundliste, 'Grundliste')

    disp('Differentielle Gleichung: Folgezustandsvariable "ableiten"!')
    Grundliste{Zeile, 3} = subs( Grundliste{Zeile, 3}, ...
        Zustandsvariable(i+1,1), Zustandsableitung(i+1,1), 0);
    anzeigen_Liste(Grundliste, 'Grundliste')

    streichen_Ergebnisliste( Konfliktvariablen(i,1) );
end

%% Ende
% Sollte die Funktion bis zu diesem Punkt gekommen sein,
%, wurden Änderungen in Ergebnis- und Grundliste vorgenommen!
end
```

A.2. Hauptfunktionen

A.2.1. analysieren()

```
function [   ] = analysieren()
%ANALYSIEREN
%
%   Steuert den Analysator und ermittelt SS für Bondliste
%
%   Schritte:
%       1. Zustandsvektoren x und dx erstellen
%       2. Ausgangsvektor y
%       3. Eingangsvektor u
%       4. Gleichungen für Quellen erstellen
%       5. Bilanzen und Gleichnisse aufstellen
%       6. Bondgleichungen aufstellen
%       7. Variablenbezeichner ersetzen
%       8. Kausalitätskontrolle
%       9. initiale Aktualisierung der Gleichnisse und Ergebnisliste
%       10. Gleichungen für Zustandsableitungen ermitteln
%       11. Ausgangsgleichungen ermitteln
%       12. (optional) Substitution der Parameter
%       13. Zustands- und Ausgangsgleichungen von EL extrahieren
%       14. Matrizen A, B, C und D erzeugen
%       15. (optional) ss-Modell erstellen
%       16. Ausgabe

%% Vorbereitung
    global Elementezahl
    global Bondliste
    global Grundliste
    global Ergebnisliste
    global registrierteBonds
    global Analysestufe
    global x
    global dx
    global y
    global u

if isempty(Analysestufe)
    Analysestufe = 1;
end
```

```
%% Durchführung der Analyse
if Analysestufe == 1

    if registrierteBonds == Elementezahl - 1
        erstelle_Zustaende();           % Vektoren x und dx erstellen
        erstelle_Ausgangsvektor();      % Vektor y erstellen
        u = erstelle_Eingangsvektor();  % Vektor u erstellen

        % Gleichungen für Quellen auf Ergebnisliste erstellen...
        for i = 1 : length(u)
            erstelle_Gleichung( u{i}, u{i}, 'Eingang', 'Ergebnisliste' );
        end

        % Gleichnisse und Bilanzen auf Knotenliste ergänzen
        erstelle_Knotengleichungen();

        % Gleichungen für einzelne Bonds erstellen
        for BondIndex = 1:length(Bondliste.BondID)
            erstelle_Bondgleichungen( Bondliste.BondID(BondIndex) )
        end

        % Standartbezeichner durch Quell-, Zustands- und
        % Ableitungsbezeichener ersetzen
        ersetze_Bezeichnungen();

%         subsituieren_Werte();
        korrigiere_Causality();

        % initialer Aktualisierungsvorgang
        aktualisieren();

        % Zuständsgleichungen ermitteln
        for j = 1 : length(dx)
            ermitteln_Gleichung( dx{j,1} )
            aktualisieren();
            clear global gesperrteGleichungen
            clear global gesperrteVariablen
        end

        % Ausgangsgleichungen ermitteln
        for j = 1 : length(y)
```

```
    ermitteln_Gleichung( y{j,1} )
    aktualisieren();
    clear global gesperrteGleichungen
    clear global gesperrteVariablen
end

anzeigen_Liste(Ergebnisliste, 'Ergebnisliste');

% Benutzerabfrage: Art der Ausgabe
Auswahl = -1;
fprintf('Wählen Sie die Art der Ausgabe: \n');
while ( Auswahl ~= 1 && Auswahl ~= 0 )
    fprintf('\n\n[0] Symbolische Ausgabe der Systemgleichungen\n');
    fprintf('[1] Ausgabe des ss-Models mit Werten\n');
    Auswahl = input('Auswahl [ 0, 1 ]: ');
end

% Substitution aller Variablen mit Parametern
if Auswahl == 1
    subsituieren_Werte();
end

% Extraktion der Zustands- und Ausgangsgleichungen aus EL
Zustandsgleichungen = ermitteln_Gleichungssatz(Ergebnisliste, dx);
Ausgangsgleichungen = ermitteln_Gleichungssatz(Ergebnisliste, y);

% Matrizen A und B aus Zustandsgleichungen extrahieren
if isempty(Zustandsgleichungen)
    warning('Es wurden keine Zustandsgleichungen ermittelt!');
    A = 0
    B = 0
else
    disp('Zustandsmatrizen ermitteln... ');
    A = ermitteln_Matrix( Zustandsgleichungen, x )
    B = ermitteln_Matrix( Zustandsgleichungen, u )
end

% Matrizen C und D aus Ausgangsgleichungen extrahieren
if isempty(Ausgangsgleichungen)
    warning('Es wurden keine Ausgangsgleichungen ermittel!');
    C = zeros(size(A))
    D = zeros(size(B))
```

```
        else
            disp('Ausgangsmatrizen ermitteln... ');
            C = ermitteln_Matrix( Ausgangsgleichungen, x )
            D = ermitteln_Matrix( Ausgangsgleichungen, u )
        end

        % Matrizen von sym in double konvertieren und ss-Modell erstellen
        if Auswahl == 1
            A = eval(sym(A));
            B = eval(sym(B));
            C = eval(sym(C));
            D = eval(sym(D));
            Modell = ss(A, B, C, D);
            assignin('base', 'Modell', Modell)
        end



        assignin('base','A',A);
        assignin('base','B',B);
        assignin('base','C',C);
        assignin('base','D',D);
        assignin('base','Ausgangsgleichungen',Ausgangsgleichungen);
        assignin('base','Zustandsgleichungen',Zustandsgleichungen);


        clear global Arbeitsliste Ergebnisliste Grundliste
        clear global Elementezahl registrierteBonds
        clear global gesperrteGleichungen gesperrteVariablen

%% Fehlerbehandlung
    elseif registrierteBonds > Elementezahl - 1
        error('Fehler: registrierteBonds > Elementezahl - 1')
    elseif registrierteBonds < Elementezahl - 1
        error('Fehler: registrierteBonds < Elementezahl - 1!')
    end

    Analysestufe = Analysestufe + 1;
    return

elseif Analysestufe > 1
    disp('Analyse wurde bereits gestartet/ausgeführt')
    assignin('base','Analysestufe',Analysestufe);
```

```
    return
end
end
```

A.2.2. ermitteln_Gleichung()

```
function [ gesGleichung gesGlNummer gesBemerkung ] = ...
    ermitteln_Gleichung( gesVariable )
%ERMITTELN_GLEICHUNG ( Variable )
%
%   Die Funktion ermittelt die Gleichung für die gesuchte Variable und gibt
%   diese mit Gleichungsnummer und Bemerkung zurück.
%
%   Stufe 1: Durchsuchen der Ergebnisliste
%   Stufe 2:
%   Stufe 3:
%   Stufe 4:
%   Stufe 5:

%% Vorbereitung & Prüfung / Konvertierung der Eingabeparameter
    if nargin ~= 1
        error('Fehlerhafte Anzahl an Eingabeparameter!');
    else
        global Grundliste           %#ok<TLEV>
        global Arbeitsliste         %#ok<TLEV>
        global Knotenliste          %#ok<TLEV>

        if isempty(Grundliste)
            error('Grundliste ist leer!');
        elseif isempty(Knotenliste)
            error('Knotenliste ist leer!');
        end

        gesVariable = sym(gesVariable);
        Subvariablen = sym([]);
        fprintf('======> AUFRUF: ermitteln_Gleichung( "%s" )\n', ...
            char(gesVariable) );
    end

%% Stufe 1: Ergebnisliste durchsuchen
    fprintf(' \n%s: ------- STUFE 1 - \n\n', char(gesVariable) );
    [aktGleichung aktGlNummer aktBemerkung] = ...
        durchsuche_Ergebnisliste(gesVariable);

    if ~isempty(aktGleichung)
```

```
        fprintf('Gleichung %d für Variable %s auf EL gefunden!\n', ...
                aktGlNummer, char(gesVariable) );

        gesGleichung = aktGleichung
        gesGlNummer = aktGlNummer;
        gesBemerkung = aktBemerkung;

        fprintf( '<====== BEENDET: ermitteln_Gleichung("%s")\n\n', ...
            char(gesVariable));
        return
    end

    % Aufräumen vor Stufe 2
    clear aktGleichung aktGlNummer aktBemerkung % aufräumen

%% Stufe 2: Grundliste durchsuchen
    fprintf('\n\n%s: ------- STUFE 2 - \n\n', char(gesVariable) );

    % Grundliste durchsuchen
    [aktGleichung aktGlNummer aktBemerkung] = ...
        durchsuche_Grundliste(gesVariable);

    % Wenn Variable auf Grundliste gefunden wurde....
    if ~isempty(aktGleichung)
        fprintf( '-- Gleichung %d für Variable %s auf GL gefunden!\n', ...
                 aktGlNummer, char(gesVariable) );

        % Gleichungen nacheinander überprüfen
        for GlNrIndex = 1:length(aktGleichung)

            [ rekursiveEignung rekursiveVariable ] = ...
                ist_rekursiv_geeignet( aktGleichung( GlNrIndex ) )

            % Lösungsversuch durch rekursives Einsetzen
            if rekursiveEignung
                fprintf('- Gleichung %d ist mögliche rekursive Lösung!',...
                    aktGlNummer( GlNrIndex ) );
                [ Status ] = teste_rekursive_Substitution( gesVariable, ...
                    aktGleichung( GlNrIndex ) ); % Simulation

                % Test bestanden... erfolgreich beenden...
                if Status
```

```
            fprintf('- - Gleichung rekursiv lösbar!\n');
            gesGleichung = aktGleichung( GlNrIndex )
            gesGlNummer = aktGlNummer( GlNrIndex )
            gesBemerkung = aktBemerkung( GlNrIndex )
            freigeben_Variable(gesVariable)
            fprintf( [ '<===== BEENDET: ermitteln_Gleichung' ...
                       '("%s")\n\n' ], char(gesVariable) );
            return
        end
    end

    % Wenn aktuelle Gleichung ...
    % ... gesperrt ist, wird diese übersprungen.
    if ist_gesperrt_Gleichung( aktGlNummer( GlNrIndex ) )
        continue

     % ... nicht gesperrt ist, werden die Subvariablen ermittelt.
    else

        % Gleichung auf AL kopieren und sperren
        kopieren_GL2AL( aktGlNummer( GlNrIndex ) );
        sperre_Gleichung( aktGlNummer( GlNrIndex ) );
        sperre_Variable( gesVariable );

        % Variablen aus Gleichung extrahieren...
        preSubvariablen = transpose( symvar( ...
                                    aktGleichung( GlNrIndex ) ) );

        % Subvariabeln einzeln durchgehen...
        for i = 1:length( preSubvariablen )

            fprintf('- - Untersuche Variable %s:\n', ...
                char(preSubvariablen(i)));
            [ Symbol ] = ermitteln_VarSymbol( preSubvariablen(i) );

            % Konstanten ausschließen...
            if ~(   strcmp(Symbol,'e')  || ...
                    strcmp(Symbol,'f')  || ...
                    strcmp(Symbol,'dp') || ...
                    strcmp(Symbol,'dq') || ...
                    strcmp(Symbol,'E')  || ...
                    strcmp(Symbol,'F')     )
```

```
            fprintf(['- - - Variable %s ist kein'    ...
                     'Effort oder Flow!\n'],         ...
                     char(preSubvariablen(i)));
            continue
        else
            Subvariablen(end+1,1) = preSubvariablen(i);
        end
    end

    clear preSubvariablen

    % nach einzelner SubVariable suchen
    for i = 1:length(Subvariablen)

        % Sperrung überprüfen
        if ist_gesperrt_Variable(Subvariablen(i))
            break % bei Sperrung abbrechen (Gleichung)

        else
            % Ermittlung d. Subvariable durch rekursiven Aufruf
            % von ermitteln_Gleichung( Subvariable )
            [ SubGleichung SubGlNummer SubBemerkung ] = ...
                ermitteln_Gleichung( Subvariablen(i) )

            fprintf(['\n+ + STUFE 2: Rückkehr aus ' ...
                     'Rekursion: %s ---> %s \n'], ...
                     char(Subvariablen(i)), char(gesVariable));
            anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');

            % Abbruch bei Fehlschlag
            if isempty(SubGleichung)
                fprintf('- - SubVariable nicht ermittelbar!');
                streichen_Arbeitsliste(gesVariable);
                freigeben_Gleichung( aktGlNummer(GlNrIndex ));
                break % aktuelle Gleichung abbrechen
            end
        end

        % nach rekursiven Variablen in Lösung suchen
        [ rekursiv rekursiveVariable ] = ...
            ist_rekursiv_Gleichung( gesVariable, SubGleichung);
```

```
            % Rekursion ggf. Auflösen
            if rekursiv
                disp(['Explizite Ergebnisvariable in Gleichung' ...
                      'gefunden... Schleife wird aufgelöst...!' ]);
                Ergebnisgleichung = substituieren_rekursiv(     ...
                                    aktGlNummer( GlNrIndex ), ...
                                    Subvariablen(i),          ...
                                    SubGleichung );
            else
                Ergebnisgleichung = substituieren_Variable(     ...
                                    aktGlNummer( GlNrIndex ), ...
                                    Subvariablen(i),          ...
                                    SubGleichung );
            end

            % Überprüfen ob alle Subvariablen bekannt sind, andern-
            % falls ist eine rekursive Lösung entstanden
            fertig = ist_fertig(Ergebnisgleichung);

            if i == length(Subvariablen)

                if  fertig
                    disp('Alle Subbariablen ermittelt!')
                    eintragen_Ergebnis(aktGlNummer (GlNrIndex) );
                else
                    disp(['Gleichung enthält rekursive '       ...
                          'Variablen. Keine Eintragung in EL!' ])
                end

                % Freigeben und Aufräumen
                freigeben_Variable(gesVariable);
                freigeben_Gleichung( aktGlNummer( GlNrIndex ) );
                [ gesGleichung gesGlNummer gesBemerkung ] = ...
                    durchsuche_Arbeitsliste( gesVariable )
                streichen_Arbeitsliste(gesVariable);
                fprintf(['<===== BEENDET: ermitteln_Gleichung' ...
                         '("%s")\n\n'], char(gesVariable));
                return
            end

        end
    end
```

```
        end
    else
        fprintf(['\n... Es wurden keine passenden Gleichungen in ' ...
                 'der Grundliste gefunden!\n\n']);
    end

    % Aufräumen vor Stufe 3
    streichen_Arbeitsliste(gesVariable);
    clear rekursiveVariable
    clear aktGleichung aktGlNummer aktBemerkung
    clear gesBemerkung gesGleichung
    clear GlNrIndex
    clear fertig
    clear Ergebnisgleichung
    clear rekursiv rekursiveVariable
    clear SubGleichung Subvariable SubBemerkung SubGlNummer
    clear Symbol


%% Stufe 3: Gleichnisse durchsuchen
%   Stufe 3 wird nur erreicht, wenn Stufe 1 und 2 fehlgeschlagen sind.
%   Sämtliche SPERRUNGEN aus Stufe 2 sind noch aktiv!
%   In dieser Stufe des Ermittlungsverfahrens wird versucht unbekannte die
%   Variable aus den Gleichnissen der Knoten zu ermitteln.

    fprintf('\n\n%s: ------- STUFE 3 - \n\n', char(gesVariable) );

    % KnotenIDs der Gleichnisse ermitteln, in denen die Variable vorkommt.
    Gleichnisvariablen = ermitteln_Gleichnisvariablen( gesVariable );

    % gesVariable sperren, falls diese in Stufe 2 nicht in GL war und somit
    % noch nicht gesperrt ist!
    sperre_Variable( gesVariable );

    % Wenn Knoten
    % ... nicht vorhanden sind: weiter mit Stufe 4
    if isempty(Gleichnisvariablen)
        fprintf('Die Variable %s kommt in keinem Gleichnis vor!\n', ...
            char(gesVariable));
    else
        % Gleichnisvariablen durchgehen.
        for i = 1:length(Gleichnisvariablen)
```

```
% Wenn aktuelle Variable...
% ... gesperrt ist, wird diese übersprungen.
if ist_gesperrt_Variable( Gleichnisvariablen( i, 1 ) )
    fprintf('Variable "%s" ist gesperrt... nächste..\n', ...
        char( Gleichnisvariablen( i, 1 ) ) );
    continue
% ... nicht gesperrt ist, wird diese gesperrt und ermittelt.
else
    fprintf('Variable "%s" ist NICHT gesperrt ...\n', ...
        char( Gleichnisvariablen( i, 1 ) ) );

    sperre_Variable( Gleichnisvariablen( i, 1 ) );

    % Variablengleichnis auf AL erstellen.
    % --> Schema: gesuchte Variable = Gleichnisvariable
    erstelle_Gleichung( gesVariable,                    ...
                        Gleichnisvariablen( i, 1 ), ...
                        'Gleichnis', 'Arbeitsliste');

    % Gleichungsnummer für neues Gleichnis von AL holen
    [ aktGleichung aktGlNummer aktBemerkung ] = ...
        durchsuche_Arbeitsliste(gesVariable)

    % Gleichnisvariable mit rekursiven Aufruf ermitteln
    [ gesGleichung gesGlNummer gesBemerkung ] = ...
        ermitteln_Gleichung( Gleichnisvariablen( i, 1 ) );

    fprintf(['\n ++ STUFE 3: Rückkehr aus Rekursion:'  ...
             '%s ---> %s \n'],                          ...
             char( Gleichnisvariablen( i, 1 ) ),        ...
             char( gesVariable ) );

    anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');

    % Abschluss analog Stufe 2
        % Fehlschlag
        if isempty(gesGleichung)
            fprintf(['Variable "%s" konnte nicht durch '    ...
                     'Gleichnisse ermittelt werden...\n'], ...
                     char(Gleichnisvariablen(i,1))          );
            streichen_Arbeitsliste(gesVariable);
```

```
                continue % nächste Gleichnisvariable versuchen

            else
                % nach rekursiven Variablen in Lösung suchen...
                % ... und Selbstbezug auflösen
                if ist_rekursiv_Gleichung(gesVariable,gesGleichung)
                    substituieren_rekursiv( aktGlNummer,          ...
                                Gleichnisvariablen( i, 1 ), ...
                                gesGleichung );

                %... oder normal substituieren
                else
                    substituieren_Variable( aktGlNummer,          ...
                                Gleichnisvariablen( i, 1 ), ...
                                gesGleichung );
                end

                freigeben_Variable(gesVariable);

                [ gesGleichung gesGlNummer gesBemerkung ] = ...
                    durchsuche_Arbeitsliste(gesVariable);


                % Überprüfen ob alle Subvariablen bekannt sind,
                % andernfalls ist eine rekursive Lösung entstanden
                if ist_fertig( gesGleichung )
                    disp('Ergebis wird jetzt eingetragen ..... ')
                    eintragen_Ergebnis( gesGlNummer, 'GleichnisX')
                    disp(' ..... Ergebis wurde eingetragen!')
                else
                    disp(['Gleichung enthält rekursive ' ...
                          'Variablen. Keine Eintragung in EL!']);
                end

                % Aufräumen und Rückgabe
                streichen_Arbeitsliste(gesVariable);
                fprintf(['<===== BEENDET: ermitteln_Gleichung' ...
                         '("%s")\n\n'], char(gesVariable));
                return
            end
        end
    end
```

```
    end

    % Aufräumen vor Stufe 4
    streichen_Arbeitsliste(gesVariable);

    clear Gleichnisvariablen
    clear aktGleichung aktGlNummer aktBemerkung
    clear gesGleichung gesGlNummer gesBemerkung


%% Stufe 4: Alternativen finden
%   Diese Stufe kann nur erreicht werden, wenn Stufen 1 - 3 fehlgeschlagen
%   sind. In dieser Stufe wird in den Gleichungsrümpfe der Grundliste nach
%   der gesuchten Variable gesucht. Die gefundenen Gleichungen werden der
%   Reihe nach umgestellt und anschließend versucht die gesuchte Variable
%   aus diesen neuen, umgestellten Gleichungen zu ermitteln.

fprintf('\n\n%s: ------ STUFE 4 - \n\n', char(gesVariable) );

    % Gleichungsnummern der zutreffenden Gleichungen von GL ermitteln
    aktGlNummern = durchsuche_Gleichungen ( gesVariable );

    if ~isempty( aktGlNummern )
        fprintf(['Die Variable kommt in mindestens einem ' ...
                 'Gleichungsrumpf der Grundliste vor!\n']);

        % Gleichungen nacheinander überprüfen
        for Index = 1:length(aktGlNummern)

            % Wenn die aktuelle Gleichung ...
            % ... gesperrt ist, wird diese übersprungen.
            if ist_gesperrt_Gleichung( aktGlNummern(Index) )
                continue

            % ... nicht gesperrt ist, wird die Gleichung umgestellt.
            else

                % Gleichung sperren
                sperre_Gleichung(aktGlNummern(Index));

                % Gleichung nach gesuchter Variable umstellen
                aktGleichung = umstellen_Gleichung( aktGlNummern(Index),...
```

```
    gesVariable, Grundliste, 'Grundliste' )

% neue Gleichung auf Arbeitsliste eintragen
eintragen_Gleichung( aktGlNummern(Index), gesVariable, ...
    aktGleichung, 'umgestellt');

% Eignung der Gleichung als rekursive Lösung übertprüfen
[ rekursiveEignung rekursiveVariable ] = ...
    ist_rekursiv_geeignet( aktGleichung )

if rekursiveEignung
    [ Status ] = teste_rekursive_Substitution( ...
        gesVariable, aktGleichung );

    if Status
        fprintf(['Neue Gleichung kann direkt als ' ...
            'rekursive Lösung genutzt werden!\n Return.'])
        gesGleichung = aktGleichung
        gesGlNummer = aktGlNummern(Index)
        gesBemerkung = 'umgestellt'
        freigeben_Variable(gesVariable)
        freigeben_Gleichung(aktGlNummern(Index));
        fprintf(['<===== BEENDET: ermitteln_Gleichung' ...
                '("%s")\n\n'], char(gesVariable));
        return
    end
end

%Variablen aus Gleichung extrahieren...
preSubvariablen = transpose(symvar(aktGleichung));

% Subvariabeln einzeln durchgehen...
for i = 1:length(preSubvariablen)

    [ Symbol ] = ermitteln_VarSymbol( preSubvariablen(i) );

    % Konstanten ausschließen... also überspringen
    if ~(   strcmp(Symbol,'e')  || ...
            strcmp(Symbol,'f')  || ...
            strcmp(Symbol,'dp') || ...
            strcmp(Symbol,'dq') || ...
            strcmp(Symbol,'E')  || ...
```

```
            strcmp(Symbol,'F')   )
        fprintf(['- - - Variable %s ist kein Effort ' ...
            'oder Flow!\n'], char(preSubvariablen(i)));
        continue

    % Effort oder Flow zur Weiterverarbeitung vermerken
    else
        Subvariablen(end+1,1) = preSubvariablen(i);
    end
end

clear preSubvariablen

% Subvariablen nacheinander durchgehen
for i = 1:length(Subvariablen)

    % Falls eine gesperrt ist, wird eine rekursive Lösung
    % m.H. der neuen Gleichung getestet
    if ist_gesperrt_Variable(Subvariablen(i))
        disp(['Ein Subvariable ist gesperrt...rekursive'...
            'Ermittlung ("Stufe 4.5") wird versucht!'])

        % Rekursiver Lösungsversuch
        [ SubGleichung SubGlNummer SubBemerkung ] = ...
            ermitteln_rekursiv( Subvariablen(i) )

        if isempty(SubGleichung) % Abbruch bei Fehlschlag
            disp(['...  nicht erfolgreich ...' ...
                'nächste Gleichung oder Stufe 5!']);

            % umgestellte Gleichung verwerfen und aufräumen
            streichen_Arbeitsliste(gesVariable);
            freigeben_Gleichung(aktGlNummern(Index));
            break
        else
            disp(['... neue Gleichung ist als ' ...
                'rekursive Lösung verwendbar!'])
        end
    else

        % Ermittlung der Subvariable durch rekursiven Auf-
        % ruf von ermitteln_Gleichung( Subvariablen(i) )
```

```
        [ SubGleichung SubGlNummer SubBemerkung ] = ...
            ermitteln_Gleichung( Subvariablen(i) );

        fprintf(['\n +++ STUFE 4: Rückkehr aus '      ...
                 'Rekursion: %s ---> %s \n'],          ...
                 char(Subvariablen(i)), char(gesVariable));
        anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');

        % Bei Fehlschlag neue Gleichung verwerfen
        if isempty(SubGleichung)
            fprinf(['Subvariable konnte nicht ' ...
                    'ermittelt werden!']);
            freigeben_Gleichung( aktGlNummern(Index) );
            streichen_Arbeitsliste(gesVariable);
            break
        end
    end

    % Wenn alle Subvariablen ermittelt wurde, wird die
    % Gleichung auf Verwendbarkeit als rekursive Lösung ge-
    % testet und ggf. entsprechend substituiert oder auf-
    % gelöst.
    [rekursiv rekursiveVariable] = ...
        ist_rekursiv_Gleichung( gesVariable, SubGleichung);

    if rekursiv
        disp(['Explizite Ergebnisvariable in Gleichung '...
              'gefunden... Schleife wird aufgelöst...!' ]);
        Ergebnisgleichung = substituieren_rekursiv(     ...
                                    aktGlNummern(Index),...
                                    Subvariablen(i),    ...
                                    SubGleichung );
    else
        Ergebnisgleichung = substituieren_Variable(     ...
                                    aktGlNummern(Index),...
                                    Subvariablen(i),    ...
                                    SubGleichung );
    end

    % Überprüfen ob alle Subvariablen bekannt sind, andern-
    % falls ist eine rekursive Lösung entstanden
    fertig = ist_fertig(Ergebnisgleichung);
```

```
                    if i == length(Subvariablen)

                        if fertig
                            disp('Es wurden alle Subvariablen ermittelt!')
                            eintragen_Ergebnis( aktGlNummern(Index) );
                        else
                            disp(['Gleichung enthält rekursive ' ...
                                  'Variablen. Keine Eintragung in EL!'])
                        end

                        % Aufräumen
                        freigeben_Variable(gesVariable);
                        freigeben_Gleichung( aktGlNummern(Index) );
                        [ gesGleichung gesGlNummer gesBemerkung ] = ...
                            durchsuche_Arbeitsliste( gesVariable )
                        streichen_Arbeitsliste(gesVariable);
                        fprintf(['<===== BEENDET: ermitteln_Gleichung' ...
                                 '("%s")\n\n'], char(gesVariable));
                        return
                    end
                end
            end
        end
    end

    % Aufräumen vor Stufe 5
    streichen_Arbeitsliste(gesVariable);
    clear aktGlNummern aktGlNummer aktGleichung
    clear Subvariablen preSubvariablen
    clear gesGleichung gesGlNummer gesBemerkung
    clear SubGleichung SubGlNummer SubBemerkun


%% STUFE 5: rekursive Ermittlung
% Konnte in den vorherigen Stufen 1 - 4 keine Lösung durch eine direkte
% Suche für die übergebene Variable ermittelt werden, wird in der fünften
% Stufe versucht, eine Lösungsgleichung, welche direkt von der gesuchten
% abhängt.
%
% Die Ermittlung von Lösungen mit Selbstbezug ist in die Funktion
% ermittlen_rekursiv ausgelagert, welche bereits in Stufe 4 für die ver-
```

```
% suchte Ermittlung gesperrter Subvariablen verwendet wurde.

    fprintf('\n\n%s: ------- STUFE 5 - \n\n', char(gesVariable) );

    % "Aufruf" der fünften Stufe
    [ gesGleichung gesGlNummer gesBemerkung ] = ...
        ermitteln_rekursiv( gesVariable )

    if isempty(gesGleichung)
        warning('<===== FEHLSCHLAG: ermitteln_Gleichung("%s")\n', ...
            char(gesVariable));
    else
        freigeben_Variable(gesVariable)
    end

end % Ende der Funktion
```

A.2.3. ermitteln_rekursiv()

```
function [ gesGleichung gesGlNummer gesBemerkung ] = ...
                                          ermitteln_rekursiv( gesVariable )
%[ gesGleichung gesGlNummer gesBemerkung ] = ...
%                                         ermitteln_rekursiv( gesVariable )
%
%   Die Funktion stellt die fünfte Stufe des eigentlichen Ermittlungs-
%   prozesses einer Variable der Funktion ermitteln_Gleichung() dar.
%
%   Es wird versucht, die gesuchte Variable mit Hilfe von selbsbezogenen
%   Gleichunge zu ermitteln. Die Funktion besteht auf 4 Stufen:
%
%   Stufe 5.1: Grundliste nach rekursiven Lösungen durchsuchen
%
%   Stufe 5.2: Gleichungsrümpfe nach gesuchter Variable umstellen und ver-
%              suchen, diese als rekursive Lösung zu verwenden
%
%   Stufe 5.3: Gleichnisvariablen der gesuchten Variable ermitteln und
%              versuchen, für diese eine selbstbezogene Lösung zu finden
%
%   Stufe 5.4: Stufe 5.2 für Gleichnisvariablen
%
%   Jede potentielle Lösung wird m. H. von teste_sekursive_Substitution()
%   getestet.
%
%   Sollte eine Lösung gefunden werden, wird der zugehörige Gleichungssatz
%   zurückgegeben, wodurch die Rücksubstitutionkette ausgelöst wird.

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl an Eingabeparametern!');
end

fprintf('\n\n\n----- STUFE 5: Rekursive Ermittlung!\n\n');

global Grundliste

gesVariable = sym(gesVariable);
gesGleichung = sym([]);
gesGlNummer = [];
gesBemerkung = char([]);
```

```
%% Stufe 5.1. Variable explizit suchen und testen
%   Die Grundliste wird nach Gleichungen gesucht, welche die übergebene
%   Variable explizit beschreiben.
%   Diese werden anschließen auf Eignung als rekursive Lösung unersucht und
%   bei gebenener Eignung die Rücksubstitution mit diesen simuliert.
%   Werden keine expliziten Gleichungen gefunden oder können diese nicht
%   verwendet werdet, wird Stufe 5.2. ausgeführt
fprintf('Stufe 5.1.: Variable %s wird auf Grungliste gesucht...\n', ...
    char(gesVariable));

% Grundliste nach gesuchter Variable durchsuchen
[ Gleichung Gleichungsnummer Bemerkung ] = ...
    durchsuche_Grundliste( gesVariable );

fprintf('Stufe 5.1.: Es wurden %d Gleichungen gefunden...\n', ...
    length(Gleichungsnummer));

% Schleife wird hier nur verwendet, falls die gesuchte Variable
% explizit in einer Bilanz und in einer Elementegleichung vorkommt, in
% diesem Fall würde durchsuche_GL() zwei Ergebnisse liefern.
for i = 1:length(Gleichungsnummer)

    % Eignung der Gleichnisgleichung überprüfen
    [ Eignung rekursiveVariable ] = ist_rekursiv_geeignet( Gleichung(i) );

    % Eignung?
    if Eignung
        fprintf('Stufe 5.1.: Gleichung %d ... wird simuliert!\n', ...
            Gleichungsnummer(i) );

        [ Status ] = teste_rekursive_Substitution( gesVariable, Gleichung );

        % Nächste Gleichung bei nicht bestandener Probesimulation
        if Status == false
            continue
        end

        fprintf(['Stufe 5.1.: Gleichung %d für ' ...
                 'Variable %s funktioniert!\n'], ...
                Gleichungsnummer(i), char(gesVariable) );
```

```
        % Rückgabe und Ende der Funktion
        gesGleichung = simplify(Gleichung(i))
        gesGlNummer = Gleichungsnummer(i);
        gesBemerkung = 'rekursiv_S1';
        return

    else
        fprintf('Stufe 5.1.: Gleichung %d ist ungeeignet!\n', ...
            Gleichungsnummer(i) );
    end
end

% Aufräumen mit Stufe 5.2
clear i Eignung Status
clear Gleichungsnummer Gleichung Bemerkung
clear rekursiveVariable

%% Stufe 5.2. Variable in Gleichungsrümpfen suchen und testen
%   Grundsätzlich wie Schritt 1, jedoch werden hier die Gleichungen ver-
%   wendet, in deren Rümpfen die gesVariable vorkommt.
%   Die zutreffende Gleichungen werden analog Stufe 2 umgestellt und an-
%   schließen Eignung überprüft, simuliert und ggf. verwendet.

fprintf('\n\nStufe 5.2.: Variable %s wird Gleichungen gesucht...\n', ...
    char(gesVariable));

% IDs der Gleichungen ermitteln, welche von der ges. Variable abhängen
VorkommenID = ermitteln_Vorkommen(gesVariable, Grundliste, 'Grundliste');

fprintf('Stufe 5.2.: Variable %s wurde in %d Gleichungen gefunden.\n', ...
    char(gesVariable), length(VorkommenID) );

% Jede gefundene Gleichung einzeln überprüfen
for m = 1:length(VorkommenID)

    fprintf('Stufe 5.2.: Gleichung %d  wird überprüft ...\n', ...
        VorkommenID(m));

    % Gleichung nach Ersatzvariable aus Gleichnis umstellen
    Gleichung = umstellen_Gleichung( VorkommenID(m), gesVariable, ...
        Grundliste, 'Grundliste')
```

```
    % Eignung testen & rekursive Variable ermitteln
    [ Eignung rekursiveVariable ] = ist_rekursiv_geeignet( Gleichung )

    % Wenn Gleichung geeignet ist...
    if Eignung

        fprintf(['Stufe 5.2 .: umgestellte Gleichung %d ' ...
                 'ist grundsätzlich geeignet!\n'], VorkommenID(m) );

        % rekursive Lösung simulieren...
        [ Status ] = teste_rekursive_Substitution(gesVariable, Gleichung);

        % Nächste Gleichung bei nicht bestandener Probesimulation
        if Status == false
            continue
        end

        fprintf(['Stufe 5.2.: Gleichung %d für Variable %s ' ...
                 'hat Probesimulation bestanden!\n'], ...
                 VorkommenID(m), char(gesVariable) );

        % Rückgabe und Ende der Funktion
        gesGleichung = expand(Gleichung)
        gesGlNummer = VorkommenID(m) ;
        gesBemerkung = 'rekursiv_S2';
        return
    end
end

% Aufräumen vor Stufe 5.3.
clear m VorkommenID Zeile
clear Gleichungsnummer Gleichung Bemerkung
clear Eignung Status rekursiveVariable

%% Stufe 5.3. Gleichnisse durchsuchen
%  Grundsätzlich wie Stufe 5.1. jedoch wird hier versucht die Gleichnis-
%  variablen der gesuchten Variable zu ermitteln

fprintf('\n\nStufe 5.3.: Gleichnisvariablen werden ermittelt!\n');

% Alle Variablen aus Gleichnissen ermitteln
Gleichnisvariablen = ermitteln_Gleichnisvariablen( gesVariable );
```

```
% gesVariable aus Gleichnisvariable entfernen, da diese in Stufe 5.1
% bereits überprüft wurde.
if ~isempty(Gleichnisvariablen)
    Gleichnisvariablen(Gleichnisvariablen == gesVariable ) = sym([])
else
    disp('Stufe 5.3.: Es wurden keine Variablen im Gleichnis gefunden!')
    return
end

% Jede Gleichnisvariable einzeln überprüfen...
for i = 1:length( Gleichnisvariablen )

    fprintf('Stufe 5.3.: Gleichnisvariable %s wird untersucht...\n', ...
        char(Gleichnisvariablen(i,1)) );

    [ Gleichung Gleichungsnummer Bemerkung ] = ...
        durchsuche_Grundliste( Gleichnisvariablen( i, 1 ) );

    for j = 1:length(Gleichungsnummer)

        fprintf(['Stufe 5.3 --> Gleichnis %s: %s wurde als ' ...
                 'Gleichung %d auf der Grundliste gefunden.\n'], ...
                 char(gesVariable), char(Gleichnisvariablen(i,1)), ...
                 Gleichungsnummer(j) );

        % Eignung testen und rekursive Variable ermitteln
        [ Eignung rekursiveVariable ] = ist_rekursiv_geeignet(Gleichung(j))

        % Wenn Gleichung geeignet ist...
        if Eignung

            fprintf(['Stufe 5.3 --> Gleichnis %s: Gleichung %d für %s ' ...
                     'ist grundsätzlich geeignet...\n'], ...
                     char(gesVariable), Gleichungsnummer(j), ...
                     char(Gleichnisvariablen(i,1)) );

            [ Status ] = teste_rekursive_Substitution( gesVariable, ...
                                                       Gleichung(j) )

            % Nächste Gleichung bei nicht bestandener Probesimulation
            if Status == false
```

```
                continue
            end

            fprintf(['Stufe 5.3 --> Gleichnis %s: Gleichung %d für %s ' ...
                     'hat Probesimulation bestanden!\n'], ...
                     char(gesVariable), Gleichungsnummer(j), ...
                     char(Gleichnisvariablen(i,1)) );

            gesGleichung = expand(Gleichung(j))
            gesGlNummer = Gleichungsnummer(j);
            gesBemerkung = 'rekursiv_S3';
            return
        else
            fprintf(['Stufe 5.3 --> Gleichnis %s: Gleichung %d für %s ' ...
                     'kann nicht verwendet werden!\n\n'], ...
                     char(gesVariable), Gleichungsnummer(j), ...
                     char(Gleichnisvariablen(i,1)) );
        end
    end

end

fprintf('Stufe 5.3.: Fehlschlag!\n\n');

clear i j Zeile rekursiveVariable
clear Gleichungsnummer Gleichung Bemerkung
clear Eignung Status

%% Stufe 5.4. Gleichnisse durchsuchen (rechte Seite der GL)

    fprintf('\n\nStufe 5.4 --> Gleichnis ( Gleichungsrümpfe ): ...!\n');

        % Jede Gleichnisvariable einzeln überprüfen...
    for i = 1:length( Gleichnisvariablen )

        fprintf(['Stufe 5.4 --> Gleichnis (rechte Seite) %s: ' ...
                 'Vorkommen in Gleichungsrümpfen wird ermittelt...!\n'],...
                 char(Gleichnisvariablen( i, 1 )) );

        VorkommenID = ermitteln_Vorkommen( Gleichnisvariablen( i, 1 ), ...
                                           Grundliste, 'Grundliste')
```

```
% gefundene Gleichungen durchgehen...
for m = 1:length(VorkommenID)

    fprintf(['Stufe 5.4 --> Gleichnis (Gleichungsrümpfe) %s: ' ...
             'Gleichung %d wird untersucht... \n'], ...
             char(Gleichnisvariablen( i, 1 )), VorkommenID(m,1) );

    % aktuelle gefundene Gleichung umstellen...
    Gleichung = umstellen_Gleichung( VorkommenID(m), ...
        Gleichnisvariablen( i, 1 ), Grundliste, 'Grundliste')

    %... und Eignung überprüfen
    [Eignung rekursiveVariable] = ist_rekursiv_geeignet(Gleichung)

    % Wenn diese geegnet ist, wird die Substitution
    % simuliert und bei positiver Simulation durchgeführt.
    if Eignung

        fprintf(['Stufe 5.4 --> Gleichnis (Gleichungsrümpfe) ' ...
                 '%s: Gleichung %d ist grundsätzlich geeignet' ...
                 ' und wird simuliert ... \n'], ...
                 char(Gleichnisvariablen( i, 1 )), ...
                 VorkommenID(m,1) );

        [ Status ] = teste_rekursive_Substitution( gesVariable, ...
                                                   Gleichung )

        % Nächste Gleichung bei nicht bestandener Probesimulation
        if Status == false
            continue
        end

        fprintf(['Stufe 5.4 --> Gleichnis (Gleichungsrümpfe) ' ...
                 '%s: Probesimulation bestanden!\n'], ...
                char(Gleichnisvariablen( i, 1 )) );

        gesGleichung = expand(Gleichung)
        gesGlNummer = VorkommenID(m,1) ;
        gesBemerkung = 'rekursiv_S4';
        return
    else
        fprintf(['Stufe 5.4 --> Gleichnisvariable %s '...
```

```
                        '(Gleichungsrümpfe) : Gleichung %d kann '...
                        'nicht verwendet werden!\n'], ...
                        char(Gleichnisvariablen( i, 1 )), ...
                        VorkommenID(m,1) );
            end
        end
    end

fprintf('\nFEHLSCHLAG: ermitteln_rekursiv( %s )\n\n', char(gesVariable) );
end
```

A.2.4. teste_rekursive_Substitution()

```
 function [ Status ] = teste_rekursive_Substitution(gesVariable, Gleichung)
% [ Status] = TESTE_REKURSIVE_SUBSTITUTION ( gesVariable, Gleichung )
%
%   Die Funktion überprüft unter Verwendung der übergebenen Gleichug, ob
%   diese für die ebenfalls übergebene gesuchte Variable eine verwendbare
%   Lösung ist. Es wird versucht, eine vollständige Rücksubstitutionskette
%   aufzubauen und die Rekursionen von ermitteln_Gleichung() komplett zu
%   durchlaufen.
%
%   Substituitionskette wird rückwärts auf einer Testliste durchgeführt,
%   welche eine Momentaufnahme der originalen Arbeitsliste ist. Die jeweils
%   letzte Zeile der Testliste wird in die höherliegende Zeile eingesetzt
%   und anschließend gestrichen.
%
%   Ein Selbstbezug kann nur dann aufgelöst werden, wenn der Koeffizient der
%   expliziten Variable auf der linken Seite der Gleichung und im Rumpf der
%   Gleichung ungleich 1 ist. Andernfalls löscht sich die Variable aus und es
%   mit Status = false zum Caller zurückgekehrt.
%
%   Ist die Probesimulation bis zu ersten Zeile der Testliste erfolgreich
%   durchlaufen worden, wird Status = true zurückgebeben.

%% Vorbereitung
if nargin ~= 2
    error('Unerwartete Anzahl an Eingabeparametern!')
end

global Arbeitsliste;
gesVariable = sym(gesVariable);
Gleichung = sym(Gleichung);
Status = false;

% Arbeitsliste in Testliste umschreiben, um globale Liste zu schützen
Testliste = Arbeitsliste;

% Gleichung für gesuchte Variable in unterste Zeile der Testliste einsetzen
Testliste{end,3} = subs(Testliste{end,3}, gesVariable, Gleichung, 0);


%% Probesimulation
```

```
for i = length(Testliste(:,1)) : -1 : 1

    [ rekursiverFall rekursiveVariable ] =  ...
        ist_rekursiv( Testliste{ i, 2 }, Testliste{ i, 3 } );

    % Wenn ein rekursiver Fall in dieser Zeile gefunden wurde, wird der
    % Koeffizient dieser Variable untersucht und bewertet.
    if rekursiverFall
        fprintf(':: Selbstbezug in Durchlauf %d gefunden: %s', i, ...
            char( rekursiveVariable ));
        anzeigen_Liste(Testliste, 'Testliste')

        % Koeffizient der betreffenden Variable ermitteln
        rekursiverKoeffizient = ermitteln_Koeffizent( ...
            Testliste{ i, 3 }, rekursiveVariable );

        % Wert des Koeffizienten untersuchen
        % ... Abbruch bei Koeffizient == 1: Auslöschung beim Umstellen
        if rekursiverKoeffizient == 1
            fprintf(':: Koeffizient von %s ist 1! Abbruch.\n', ...
                char( rekursiveVariable ) );
            Status = false;
            return

        % ... Abbruch bei Koeffizient == 0: Auslöschung beim Einsetzen
        elseif rekursiverKoeffizient == 0
            error(':: Variable %s wurde ausgelöscht! Abbruch.\n', ...
                char( rekursiveVariable ) );

        % ... Weiter bei Koeffizient ~= 1: Selbstbezug auflösen
        else
            fprintf('::: rekursiver Koeffizient von %s ~= 1!\n\n', ...
                char( rekursiveVariable ));
            Testliste{ i, 3} = simplify( umstellen_Gleichung( ...
                                        Testliste{ i, 1}, ...
                                        Testliste{ i, 2}, ...
                                        Testliste,        ...
                                        'Testliste' ));
            Status = true;
        end
    end
```

```
    % letzte Zeile der Testliste in vorletzte Zeile einsetzen und streichen
    if i > 1
        disp('Testsubstitution wird ausgeführt...')
        Zwischenliste = Testliste(end,:);
        Testliste(end,:) = [];
        Testliste{ i - 1, 3 } = simplify(subs( Testliste{ i - 1, 3 }, ...
                    Zwischenliste{ 1, 2 }, Zwischenliste{ 1, 3 }, 0 ));
    else
        disp('Letzter Durchgang...')
    end
end

anzeigen_Liste(Testliste, 'Testliste');
fprintf(':::: Probesimulation erfolgreich!\n\n');
end
```

A.3. Manipulation

A.3.1. substituieren_rekursiv()

```
function [ Ergebnisgleichung ] = substituieren_rekursiv( ...
                                    Gleichungsnummer, Variable, Ersatzterm)
%[ Ergebnisgleichung ] = substituieren_rekursiv( ...
%                                    Gleichungsnummer, Variable, Ersatzterm)
%
%   Substituiert die übergebene Variable mit dem ebenfalls übergebenen
%   Ersatzterm (Gleichung oder Variable). Die Substition erfolgt nur auf
%   der Arbeitsliste, daher muss keine Zielliste angebenen werden.
%   Der Ersatzterm wird so wie er übergeben wurde, mit der Variable
%   substituiert. D.h. wenn Ersatzerm "e4" ist, wird auch genau "e4" für
%   die Variable eingesetzt. Gleiches gilt für längere Terme, wie z.B:
%   Ersatzterm = " ( p2 / C2 ) * R4 "
%
%   Die Gleichung wird über die Gleichungsnummer auf AL identifiziert.
%
%   WARNUNG:    Nach der erfolgreichen Substitution des Terms mit Hilfe von
%               substituieren_Variable, wird die Gleichung nach ihrer auf
%               der linken Seite des Gleichheitszeichens umgestellt und auf
%               die AL geschrieben.

fprintf(['+++AUFRUF: substituieren_rekursiv( Gl. %d, "%s", ' ...
         '[Ersatzterm] )\n'], Gleichungsnummer, char(Variable));

    global Arbeitsliste

    if nargin ~= 3
        error('Falsche Anzahl an Eingabeparametern!');
    end

    if isempty(Arbeitsliste)
        error('Arbeitsliste existiert nicht! Hier wird NIX substituiert!')
    else
        Variable = sym(Variable);
        Ersatzterm = sym(Ersatzterm);
    end

substituieren_Variable( Gleichungsnummer, Variable, Ersatzterm )
```

```
Treffer = transpose([Arbeitsliste{:,1}]) == Gleichungsnummer;

Zielvariable = Arbeitsliste(Treffer,2)

if sum(Treffer) == 1
    Arbeitsliste{Treffer,3} = simplify( umstellen_Gleichung( ...
        Arbeitsliste{ Treffer, 1 }, Zielvariable, ...
        Arbeitsliste, 'Arbeitsliste' ) );
    Ergebnisgleichung = Arbeitsliste{Treffer,3};
    anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');
else
    error('Falsche Trefferzahl für Gleichungsnummer: sum(Treffer) = %d',...
        sum(Treffer) );
end

    fprintf(['++ BEENDET: substituieren_rekursiv( Gl. %d, "%s", ' ...
             'Ersatzterm )\n'], Gleichungsnummer, char(Variable));
end
```

A.3.2. substituieren_Variable()

```
function [ Ergebnisgleichung ] = substituieren_Variable( ...
                                    Gleichungsnummer, Variable, Ersatzterm)
%Ergebnisgleichung = SUBSTITUIEREN_VARIABLE( ...
%                                    Gleichungsnummer, Variable, Ersatzterm)
%
%   Substituiert übergebene Variable mit Ersatzterm in der mit der eben-
%   übergebenen Gleichungsnummer identifizierten Gleichung.
%
%   Die Substition erfolgt aussschließlich auf der Arbeitsliste, wobei die
%   veränderte Gleichung zusätzlich zurückgegeben wird.

%% Vorbereitung
if nargin ~= 3
    error('Unerwartete Anzahl an Eingabeparametern!');
end

global Arbeitsliste

if isempty(Arbeitsliste)
    error('Arbeitsliste existiert nicht!')
end

Variable = sym(Variable);
Ersatzterm = sym(Ersatzterm);

%% Gleichung auf Arbeitsliste finden und Term substituieren
Treffer = transpose([Arbeitsliste{:,1}]) == Gleichungsnummer;

if sum(Treffer) == 1
    Arbeitsliste{Treffer,3} = simplify(subs(Arbeitsliste{Treffer,3}, ...
                                    Variable, Ersatzterm , 0));
    Ergebnisgleichung = Arbeitsliste{Treffer,3}
else
    error('Falsche Trefferzahl fürGleichungsnummer: sum(Treffer) = %d', ...
        sum(Treffer) );
end

anzeigen_Liste( Arbeitsliste, 'Arbeitsliste' );

end
```

A.3.3. umstellen_Gleichung()

```
function [ Ergebnisgleichung ] = umstellen_Gleichung( Gleichungsnummer, ...
                                                      Zielvariable, ...
                                                      Liste, ...
                                                      Listenbezeichnung )
% [ Ergebnisgleichung ] = UMSTELLEN ( Gleichungsnummer,
%                                     Zielvariable,
%                                     Liste,
%                                     Listenbezeichnung  )
%
%   Stellt die durch Gleichungsnummer identifizierte Gleichung nach der
%   Zielvariable um und gibt die neue Gleichung zurück. Die neue Gleichung
%   wird in keine Liste eingetragen und erhält somit keine neue Nummer,
%   wodurch "letzteGleichungsnummer" unverändert bleibt.
%
%   Kann die Gleichung nicht umgestellt werden, wird ein Fehler ausgelöst.

%% Vorbereitung
if nargin ~= 4
    error('Falsche Anzahl an Eingabeparametern!');
end

Zielvariable = sym(Zielvariable);
Treffer = transpose([Liste{:,1}]) == Gleichungsnummer;
AnzahlTreffer = sum(Treffer);

fprintf('---> AUFRUF: umstellen_Gleichung( %d, "%s", %s, "%s" ) \n', ...
        Gleichungsnummer, char(Zielvariable), char(Listenbezeichnung), ...
        char(Listenbezeichnung));

%% Umstellen
if AnzahlTreffer == 1
    fprintf('Gleichung %d wurde %d-mal gefunden!\n', Gleichungsnummer, ...
        AnzahlTreffer);
    Ursprungsgleichung = simplify(Liste{Treffer, 3});
    Nullgleichung = simplify(Ursprungsgleichung - Liste{Treffer, 2});
    Ergebnisgleichung = simplify(solve( Nullgleichung, Zielvariable ));

    if isempty(Ergebnisgleichung)
        error('Die Variable %s existiert in der Gleichung %s nicht!');
    end
```

```
else
    error('Gleichung %d wurde %d-mal gefunden!', Gleichungsnummer, ...
        AnzahlTreffer);
end

fprintf('<--- BEENDET: umstellen_Gleichung( %d, "%s", %s, "%s" ) \n', ...
        Gleichungsnummer, char(Zielvariable), char(Listenbezeichnung), ...
        char(Listenbezeichnung));
end
```

A.4. Aktualisierungen

A.4.1. aktualisieren()

```
function [ Aenderung ] = aktualisieren( varargin )
%AKTUALISIEREN ( varargin )
%
%   Die Funktion aktualisiert alle Gleichnisse und trägt die
%   Ergebnisse für bekannte Variablen in Gleichnisste ein.
%
%   Die Funktion sollte nach jeder erfolgreichen Ermittlung einer
%   unbekannten Variable ausgeführt werden. In der Funktion ist ein
%   rekursiver Mechanismus integriert, welcher bei einer Änderung bzw.
%   Eintragung eines neuen Ergebnisses die Aktualisierung erneut
%   durchführt.

%% Vorbereitung
if nargin == 0
    Rekursionstiefe_aktualisieren = 0;
elseif nargin == 1
    Rekursionstiefe_aktualisieren = varargin{1} + 1;
elseif nargin > 1
    error('Unerwartete Anzahl an Eingabeparameter!')
end

global Knotenliste

if isempty(Knotenliste)
    error('Knotenliste ist leer... dürfte nicht sein!')
end

Aenderung = false;

%% Aktualisierung der einzelnen Gleichnisse
for i = 1:1:length(Knotenliste.ID)
    aktuelleAenderung = aktualisieren_Gleichnis(Knotenliste.ID(i));

    if aktuelleAenderung
        Aenderung = true;
    end
end
```

```
%% Rekursiver Aufruf bei mindestens einer Änderung
if Aenderung
    Aenderung = aktualisieren(Rekursionstiefe_aktualisieren);
end
end
```

A.4.2. aktualisieren_Gleichnis()

```
function [ Aenderung ] = aktualisieren_Gleichnis( KnotenID )
%ANALYSIEREN_GLEICHNIS ( KnotenID )
%
%   Die Funktion aktualisiert das Gleichnis des durch die übergebene
%   KnotenID identifizierten Knotens. Sollte in diesem Gleichnis bekannte
%   und unbekannte Variablen existieren werden den unbekannten Variablen
%   die Werte / Gleichungen der bekannten Variablen zugewiesen und in die
%   Ergebnisliste eingetragen.
%
%   Sollte eine Änderung vorgenommen worden sein, wird TRUE zurückgegeben.

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl der Eingabeparameter!')
end

global Knotenliste

Zeile = Knotenliste.ID == KnotenID;

if sum(Zeile) > 1
    error('KnotenID %d kommt mehrfach auf Knotenliste vor!',KnotenID);
elseif sum(Zeile) < 1
    error('Es wurden KEIN Knoten mit KnotenID %d gefunden!',KnotenID);
end

%% Variablenlisten aufstellen
gleicheVariablen = transpose(Knotenliste.Gleichnisse(Zeile,:));

unbekannteVariablen = {};
bekannteVariablen = {};
Aenderung = false;

for i = 1:length(gleicheVariablen)

    if gleicheVariablen{i} == sym('0')
        continue
    end

    [ aktuelleGleichung aktuelleNummer aktuelleBemerkung ] = ...
```

```
        durchsuche_Ergebnisliste( gleicheVariablen{i} );

    % Variable als "unbekannt" vermerken
    if isempty(aktuelleGleichung)
        unbekannteVariablen{end+1, 1} = gleicheVariablen{i};

    % Variable als "bekannte" Lösung vermerken
    else
        bekannteVariablen{end+1, 1} = aktuelleNummer;
        bekannteVariablen{end, 2} = sym(gleicheVariablen{i});
        bekannteVariablen{end, 3} = aktuelleGleichung;
        bekannteVariablen{end, 4} = aktuelleBemerkung;
    end

end

%% Abbruchkriterien
if isempty(unbekannteVariablen) % Alles Variablen sind bekannt...
    return
elseif isempty(bekannteVariablen) % Alle Variablen sind unbekannt...
    return
end

%% Überprüfen und Eintragung in Ergebnisliste
Konfliktvariablen = teste_Korrektheit_Gleichnis( bekannteVariablen );

if ~isempty(Konfliktvariablen)
    error('Uneindeutige Lösungen im Gleichnis!!');
end

for i = 1:1:length(unbekannteVariablen(:,1))
    [ uGleichung uGleichungsnummer uBemerkung ] = ...
        durchsuche_Arbeitsliste( unbekannteVariablen{i} );

    if isempty(uGleichung) % Wenn Variable nicht auf Arbeitsliste
        erstelle_Gleichung( unbekannteVariablen{i}, ...
                            bekannteVariablen{end, 3}, ...
                            ['Gleichnis' num2str(KnotenID)], ...
                            'Ergebnisliste' );
        Aenderung = true;
        freigeben_Variable( unbekannteVariablen{i} );
```

```
    else % Wenn Variable auf AL... alte Gleichungsnummer
        warning( ['Variable steht bereits auf Arbeitsliste!' ...
                  'Ergebnis wird dieser Gleichungsnr in EL eingetragen!'] );
        eintragen_Gleichnis( uGleichungsnummer, unbekannteVariablen{i}, ...
            bekannteVariablen{end, 3}, ['Gleichnis' num2str(KnotenID)] );
        Aenderung = true;
        freigeben_Variable( unbekannteVariablen{i} );
        freigeben_Gleichung( uGleichungsnummer );
        streichen_Arbeitsliste( unbekannteVariablen{i} );
    end
end
end
```

A.5. Suchfunktionen

A.5.1. durchsuche_Arbeitsliste()

```
function [ Gleichung Gleichungsnummer Bemerkung ] = ...
    durchsuche_Arbeitsliste( Variable )
% [ Gleichung Gleichungsnummer Bemerkung ] = ...
%       durchsuche_Arbeitsliste( Variable )
%
%   Durchsucht die Arbeitsliste nach der übergebenen Variable. Wird diese
%   gefunden, wird die zugehörige Gleichung, Gleichungsnummer und Bemerkung
%   zurückgegeben.

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl von Eingabeparametern');
end

global Arbeitsliste
Variable = sym(Variable);
Gleichung = [];
Gleichungsnummer = [];
Bemerkung = [];

if isempty(Arbeitsliste)
    warning('Die Arbeitsliste ist leer!');
        return
end

%% Arbeitsliste durchsuchen
Treffer = Arbeitsliste(:,2)  == Variable;
if sum(Treffer) == 1
    fprintf('Variable %s auf Arbeitsliste gefunden!!!\n',char(Variable));
    Gleichung = Arbeitsliste{ Treffer, 3 };
    Gleichungsnummer = Arbeitsliste{ Treffer, 1 };
    Bemerkung = Arbeitsliste{ Treffer, 4 };
elseif sum(Treffer) == 0
    fprintf('Variable %s ist nicth auf Arbeitsliste!\n',char(Variable));
else
    error('Arbeitsliste ist fehlerhaft!', char(Variable))
end
end
```

A.5.2. durchsuche_Ergebnisliste()

```
function [ Gleichung Gleichungsnummer Bemerkung ] = ...
    durchsuche_Ergebnisliste( Variable )
%[ Gleichung Gleichungsnummer Bemerkung ] = ...
%                                          durchsuche_Ergebnisliste( Variable )
%
%   Durchsucht die Ergebnisliste nach der übergebenen Variable. Wird diese
%   gefunden, wird die zugehörige Gleichung, Gleichungsnummer und Bemerkung
%   zurückgegeben.
%
%   Die Funktion wird auch zur Überprüfung der Bekanntheit einer Variable
%   verwendet, da alle Variablen, welche sich auf der Ergebnisliste
%   befinden vollständig bekannt sind.

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl von Eingabeparametern');
end
global Ergebnisliste

if isempty(Ergebnisliste)
    warning('Die Ergebnisliste ist leer!'); return
end

Gleichung = []; Gleichungsnummer = []; Bemerkung = [];
Variable = sym(Variable);

%% Durchsuchen der Ergebnisliste
Treffer = Ergebnisliste(:,2)  == Variable;

if sum(Treffer) == 1
    fprintf('Variable %s auf EL gefunden!!!\n',char(Variable));
    Gleichung = Ergebnisliste{ Treffer, 3 };
    Gleichungsnummer = Ergebnisliste{ Treffer, 1 };
    Bemerkung = Ergebnisliste{ Treffer, 4 };
elseif sum(Treffer) == 0
    fprintf('Variable %s auf EL nicht gefunden!\n',char(Variable));
else
    error('Variabe %s kommt auf EL mehrfach vor!', char(Variable))
end
end
```

A.5.3. durchsuche_Grundliste()

```
function [ Gleichung Gleichungsnummer Bemerkung ] = ...
    durchsuche_Grundliste( Variable )
%DURCHSUCHE_GRUNDLISTE ( Variable )
%
%   Durchsucht die Grundliste nach der übergebenen Variable. Wird diese
%   gefunden, wird die zugehörige Gleichung, Gleichungsnummer und Bemerkung
%   zurückgegeben.

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl von Eingabeparametern');
end

global Grundliste

if isempty(Grundliste)
    warning('Die Grundliste ist leer!');
        return
end

Gleichung = [];
Gleichungsnummer = [];
Bemerkung = [];
Variable = sym(Variable);

if isempty(Grundliste)
    error('Die Grundliste ist leer!');
end

%% Durchsuchung der Grundliste
Treffer = Grundliste(:,2)  == Variable;

% Regulärer Treffer auf GL (Element ODER Bilanz)
if sum(Treffer) == 1
    fprintf('Variable %s auf Grundliste 1x gefunden!!!\n',char(Variable));
    Gleichung = Grundliste{ Treffer, 3 };
    Gleichungsnummer = Grundliste{ Treffer, 1 };
    Bemerkung = Grundliste( Treffer, 4 );

% Doppelter Treffer auf GL (Element UND Bilanz)
```

```
elseif sum(Treffer) == 2
    fprintf('Variable %s auf Grundliste 2x gefunden!!!\n',char(Variable));
    preGleichung = transpose([Grundliste{ Treffer, 3 }]);
    preGleichungsnummer = transpose([Grundliste{ Treffer, 1 }]);
    preBemerkung = Grundliste( Treffer, 4 );

    %Ergebnisse nach Bemerkung sortieren... Bilanz immer als zweites!
    Position_Bilanz = strcmp(preBemerkung, 'Bilanz');

    Gleichung = [ preGleichung( ~Position_Bilanz ) ;
                  preGleichung( Position_Bilanz ) ];
    Gleichungsnummer = [ preGleichungsnummer( ~Position_Bilanz ) ; ...
                         preGleichungsnummer( Position_Bilanz ) ];
    Bemerkung = [ preBemerkung( ~Position_Bilanz ) ;
                  preBemerkung( Position_Bilanz ) ];

elseif sum(Treffer) == 0
    fprintf('Variable %s auf Grundliste nicht gefunden!\n',char(Variable));

else
    error('Grundliste ist fehlerhaft', char(Variable));
end
end
```

A.5.4. durchsuche_Gleichungen()

```
function [ Gleichungsnummer ] = durchsuche_Gleichungen( Variable )
%[ Gleichungsnummer ] = DURCHSUCHE_GLEICHUNGEN ( Variable )
%
%   Ermittelt sämtliche Gleichungen, in denen die übergebene Variable auf
%   der rechten Seite vorkommt. Es werden nur die Gleichungsrümpfe der
%   Grundliste durchsucht.
%
%   Die Gleichungsnummern der gefunden Gleichungen werden zurückgegeben. Es
%   wird nicht überprüft, ob die Gleichungen gesperrt sind oder nicht.

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl von Eingabeparametern');
end

global Grundliste

Variable = sym(Variable);
Gleichungsnummer = [];

if isempty(Grundliste)
    error('Die Grundliste ist leer!');
end

%% Gleichungsrümpfe der Grundliste durchsuchen
for i = 1:length( Grundliste( :, 1 ) )
    aktVariablen = Grundliste{ i, 3};
    Vorkommen = symvar(aktVariablen) == Variable;
    if sum( Vorkommen ) > 0
        Gleichungsnummer(end+1,1) = Grundliste{ i , 1 };
    end
end

if isempty(Gleichungsnummer)
    fprintf('Variable %s kommt in keiner Gleichung vor!\n',char(Variable));
end
end
```

A.5.5. ermitteln_Vorkommen()

```
function [ Gleichungsnummern ] = ermitteln_Vorkommen( Variable, ...
                                                       Liste, Listenname )
%[ Gleichungsnummern ] = ERMITTELN_VORKOMMEN ( Variable, Liste, Listenbezeichnung )
%
%   Durchsucht die Gleichungsrümpfe der übergebene Liste nach Vorkommen der
%   übergebenen Variable und gibt die jeweiligen Gleichungsnummern aus.

%% Vorbereitung
if nargin ~= 3
    error('Falsche Anzahl an Eingabeparametern!')
end

fprintf( '---> AUFRUF: ermitteln_Vorkommen( %s, %s, "%s" )\n', ...
         char(Variable), Listenname, Listenname )

Variable = sym(Variable);
Gleichungsnummern = [];

anzeigen_Liste(Liste, Listenname);

for i = 1:1:length(Liste(:,3))
    Variablen = unique(transpose(symvar(Liste{i,3})));
    Treffer = Variablen == Variable;

    if sum(Treffer) == 1
        Gleichungsnummern(end+1,1) = Liste{i,1};
    end
end

Gleichungsnummern
fprintf('<--- BEENDET: ermitteln_Vorkommen( %s, %s, "%s" )\n', ...
    char(Variable), Listenname, Listenname)

end
```

A.6. Statusabfragen

A.6.1. ist_fertig()

```
function [ fertig ] = ist_fertig( Gleichung )
%[ fertig ] = IST_FERTIG (Gleichung)
%
%   Die Funktion überprüft die übergebenen Gleichung, ob diese vollständig
%   bekannt ist.
%
%   Eine Gleichung ist dann bekannt, wenn diese nur von Konstanten,
%   Quellvariablen und / oder Zustandsvariablen abhängt!

% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl an Eingabeparametern!')
end

Gleichung = sym(Gleichung)
Terme = symvar(Gleichung)

%% Überprüfung der einzelnen enthaltenen Terme
for i = length(Terme)
    Symbol = ermitteln_VarSymbol( Terme(i) )

    % Wenn ein Effort, Flow oder eine Zustandsableitung enhalten ist, wird
    % die Funktion beendet und fertig = false zurückgegeben.
    if (    strcmp(Symbol,'e')  || strcmp(Symbol,'f')       ...
            ||  strcmp(Symbol,'dp') || strcmp(Symbol,'dq')  )
        disp(['Es ist eine nicht aufgelöste Variable enthalten. ' ...
              'Die Gleichung ist daher nicht vollständig bekannt!']);
        fertig = false;
        return
    end
end

disp('Gleichnung vollständig bekannt!');
fertig = true;
end
```

A.6.2. ist_rekursiv()

```
function [ Status rekursiveVariable ] = ist_rekursiv( ...
                                          expliziteVariable, Gleichung )
%[ Status rekursiveVariable ] = ist_rekursiv( expliziteVariable, Gleichung)
%
%   Die Funktion überprüft die übergebene Gleichung nach Vorkommen der
%   zugehörigen und ebenfalls übergebenen expliziten Variable. Kommt diese
%   in der Gleichung vor, ist die Gleichung grundsätzlich als rekursive
%   Lösung geeignet. In diesem Fall wird Status = true und die betreffende
%   Variable von der Funktin zurückgegeben, andernfalls false.

%% Vorbereitung
if nargin ~= 2
    error('Unerwartete Anzahl an Eingabeparametern!')
end
expliziteVariable = sym(expliziteVariable);
Gleichung = sym(Gleichung);
Status = false;
rekursiveVariable = sym([]);
Variablen = symvar( Gleichung );

%% Hauptteil: Subvariable nach expliziter Variable durchsuchen
for h = 1: length(Variablen)

    [ Symbol ] = ermitteln_VarSymbol( Variablen(1, h) );

    % Konstanten ausschließen... also überspringen
    if (   strcmp(Symbol,'e') || strcmp(Symbol,'f')          ...
        || strcmp(Symbol,'dp') || strcmp(Symbol,'dq')    )

        if expliziteVariable == Variablen(1,h)
            fprintf('ist_rekursiv: Rekursion möglich mit Variable %s!', ...
                char(Variablen(1, h)) );
            rekursiveVariable = expliziteVariable;
            Status = true;
            return
        end
    end
end
end
```

A.6.3. ist_rekursiv_geeignet()

```
 function [ Eignung rekursiveVariable ] = ist_rekursiv_geeignet( Gleichung)
%[ rekursiveVariable ] = ist_rekursiv_geeignet( Gleichung )
%
%   Überprüft die übergebene Gleichung, ob diese als rekursive Lösung
%   verwendbar ist.
%
%   Eine Gleichung ist dann grundsätzlich geeignet, wenn alle enthaltenen
%   Variablen entweder bekannt oder bereits in den aktuellen Ermittlungs-
%   prozess eingebunden sind. Bereits eingebundene Variablen werden im Zuge
%   der Substitution entsprechend aufgelöst.
%
%   Diese Funktion ist als Vorstufe für die Probesimulation der rekursiven
%   Lösung gedacht. Die Funktion liefert keine Aussage darüber, ob die
%   Verwendung der überprüften Gleichung tatsächlich sinnvolle Ergebnisse
%   liefern würde.

%% Vorbereitung
    if nargin ~= 1
        error('Falsche Anzahl an Eingabeparater!');
    end

    global Arbeitsliste

    Gleichung = sym(Gleichung);
    Eignung = false;
    rekursiveVariable = sym([]);

    if isempty(Arbeitsliste)
        fprintf('Arbeitsliste ist leer!\n')
        return;
    end

%% Auflistung der enthaltenen unbekannten Variablen

% Variablen aus Gleichung extrahieren
Variablen = symvar(Gleichung);

for i = 1: length(Variablen)

    [ Symbol ] = ermitteln_VarSymbol( Variablen(i) );
```

```
        % Konstanten ausschließen und Ergebisliste durchsuchen
    if ( strcmp(Symbol,'e') || strcmp(Symbol,'f')    || ...
         strcmp(Symbol,'dp') || strcmp(Symbol,'dq') || ...
         strcmp(Symbol,'E') || strcmp(Symbol,'F')    )

        [ aktGleichung aktGlNummer aktBemerkung ] = ...
            durchsuche_Ergebnisliste( Variablen(1, i) );

        % Falls Variable unbekannt ist, wird diese vorgemerkt
        if isempty(aktGlNummer)
                rekursiveVariable(end+1,1) = Variablen(1, i);
        end
    end
end

% Falls keine unbekannten Variablen enthalten sind, kann die Gleichung
% nicht verwendet werden
if isempty(rekursiveVariable)
    fprintf('Gleichung ist nicht geeignet, da alle Variablen bekannt sind')
    Eignung = false;
    rekursiveVariable = sym([]);
    return
end

fprintf('Die Gleichung enthält %d unbekannte Variablen... \n', ...
    length(rekursiveVariable(:,1)));

%% Abgleich der enthaltenen unbekannten Variablen mit Arbeitsliste
%  Alle unbekannten Variablen müssen bereits in den vorhergehenden
%  Ermittlungsschritten eingebunden sein, um eine rekursive Lösung zu
%  ermöglichen.
for i = 1:length(rekursiveVariable(:,1))

    Vorkommen = Arbeitsliste(:,2) == rekursiveVariable(i,1);

    if sum(Vorkommen) == 1
        fprintf(['Die Unbekannte %s kommt auf Arbeitsliste genau ' ...
                 'einmal vor!\n'], char(rekursiveVariable(i,1)));
        Eignung = true;
    else
        fprintf(['Gleichung nicht geeignet, da Variable %s ' ...
```

```
                    ' nicht auf Arbeitliste vorkommt!\n'], ...
                    char(rekursiveVariable(i,1)));
            Eignung = false;
            rekursiveVariable = sym([]);
            return
        end
    end
end
```

A.6.4. ist_rekursiv_Gleichung()

```
 function [ Status rekursiveVariable ] = ist_rekursiv_Gleichung( ...
                                                Ergebnisvariable, Gleichung )
%[ Status  rekursiveVariable ] = ist_rekursiv_Gleichung( ...
%                                               Ergebnisvariable, Gleichung)
%
%   Überprüft die übergebene Gleichung mit der zugehörigen
%   Ergebnisvariable, ob die Gleichung in impliziter Form vorliegt.
%
%   Hierfür werden die in der Gleichung enthaltetenen Variablen extrahiert
%   und mit der expliziten Ergebnisvariable verglichen. Ist die explizite
%   Variable in den Variablen der Gleichung (rechte Seite) enthalten, wird
%   Status = true und für rekursiveVariable die explizite Varibale zurück-
%   gegeben. Andernfalls Status = false und rekursiveVariable = sym([]).

%% Vorbereitung
    if nargin ~= 2
        error('Falsche Anzahl an Eingabeparater!');
    else
        Gleichung = sym(Gleichung);
        Ergebnisvariable = sym(Ergebnisvariable);
        Status = false;
        rekursiveVariable = sym([]);
    end

    Variablen = symvar(Gleichung);

%% Hauptteil
    Vorkommen = Variablen == Ergebnisvariable;

    if sum(Vorkommen) == 1
        disp('Die Gleichung besitzt einen Selbstbezug!');
        Status = true;
        rekursiveVariable = Ergebnisvariable;
    elseif sum(Vorkommen) == 0
        disp('Gleichung besitzt keinen Selbstbezug.');
    else
        error('Unbekannter Fehler in übergebener Gleichung!');
    end
end
```

B. Verwaltung

B.1. Vektorerstellung

B.1.1. erstelle_Ausgangsvektor()

```
function [] = erstelle_Ausgangsvektor( )
%[] = erstelle_Ausgangsvektor( )
%
%   Erstellt auf Basis der globalen Bondliste den Ausgangsvektor y.

%% Vorbereitung

    global Bondliste y

    if isempty(Bondliste)
        error('Bondliste is leer')
    end

    if isempty(y)
        y = {}
    end

%% Hauptteil
disp('=======> Ausgangsvektor y wird erstellt:')

for Index = 1:1:length(Bondliste.BondID)

    if Bondliste.Ausgang(Index) == true

        % Wenn der Bond ein Ausgang ist, werden e & f in y vermerkt. In
        % Abhängigkeit des Senkenelementes werden die Bezeichner für den
        % Flow und Effort gewählt.
        switch Bondliste.Destination(Index).Typ
            case 'Resistor'
```

```
                Effortsymbol = 'e';
                Flowsymbol = 'f';
            case 'Ausgang'
                Effortsymbol = 'e';
                Flowsymbol = 'f';
            case 'Capacitor'
                Effortsymbol = 'e';
                Flowsymbol = 'dq';
            case 'Inductor'
                Effortsymbol = 'dp';
                Flowsymbol = 'f';
            case 'Serieller Knoten'
                Effortsymbol = 'e';
                Flowsymbol = 'f';
            case 'Paralleler Knoten'
                Effortsymbol = 'e';
                Flowsymbol = 'f';
            case 'Gyrator'
                Effortsymbol = 'e';
                Flowsymbol = 'f';
            case 'Transformer'
                Effortsymbol = 'e';
                Flowsymbol = 'f';
            otherwise
                error('Fehler: Typ nicht erkannt!')
        end

        % y-Vektor mit Effort und Flow erweitern
        y(end + 1, 1) = {[ Effortsymbol num2str(Bondliste.BondID(Index))]};
        y(end + 1, 1) = {[ Flowsymbol num2str(Bondliste.BondID(Index))]};
    end
end
y
assignin('base','y',y)
end
```

B.1.2. erstelle_Eingangsvektor()

```
function [u_out] = erstelle_Eingangsvektor( )
%[u] = erstelle_Eingangsvektor( )
%
%   Erstellt auf Basis der globalen Bondliste den Eingangsvektor u.
%
%   Ist die SRC eines Bond vom Typ "Eingang", wird diese entsprechend der
%   Kausalität des Bondes gewertet und in den Eingangsvektor u eingefügt.
%
%   Kausalität f(e) ---> E ---> SRC ist Effortsource
%   Kausalität e(f) ---> F ---> SRC ist Flowsource

%% Vorbereitung
global Bondliste u

if isempty(Bondliste)
    error('Bondliste is leer')
end
if isempty(u)
    u = {};
end

%% Hauptteil: Quellelemente aller Bonds nacheinander überprüfen
for Index = 1:1:length(Bondliste.BondID)

    % Wenn Quelle vom Typ "Eingang" ist, Quellvariable entsprechend der
    % Kausalität dem Vektor u hinzufügen
    if strcmp(Bondliste.Source(Index).Typ,'Eingang')
        if strcmp(Bondliste.Causality(Index),'f(e)')
            u(end + 1, 1) = {[ 'E' num2str(Bondliste.BondID(Index))]};
        elseif strcmp(Bondliste.Causality(Index),'e(f)')
            u(end + 1, 1) = {[ 'F' num2str(Bondliste.BondID(Index))]};
        else
            error('Kausalität von Bond Nr. %d ist fehlerhaft!', ...
                num2str(Bondliste.BondID(Index)));
        end
    end
end
assignin('base','u',u); u_out = u;
end
```

B.1.3. erstelle_Zustaende()

```
function [] = erstelle_Zustaende( )
%[] = erstelle_Zustaende( )
%
%   Die Funktion erstellt den Zustandsvektor x und dessen Ableitung dx
%   basierend auf der globalen Bondliste.
%
%   Die Bondliste wird nach energiespeichernden Elementen vom Typ Capacitor
%   oder Inertia durchsucht. Die Zustandsvariablen der zutreffenden
%   Elemente werden mit der Nummer des zugehörigen Bonds versehen und dem
%   Zustandsvektor sowie dessen Ableitung hinzugefügt.
%
%   Des Weiteren wird in dieser Funktion zusätzlich die Kausalität der
%   beiden Typen auf der Bondliste auf Korrektheit überprüft.

%% Vorbereitung
global Bondliste x dx

if isempty(Bondliste)
    error('Bondliste is leer')
end

if isempty(x)
    x = {}
end

if isempty(dx)
    dx = {}
end

%% Hauptteil
disp('=======> Zustaende werden erstellt')

% Einträge der Bondliste nacheinander durchsucehn
for Index = 1:1:length(Bondliste.BondID)

    % Wenn Inertia gefunden wird der Impuls p und dessen Ableitung mit der
    % Bondnummer auf dx und x eingetragen
    if strcmp({Bondliste.Destination(Index).Typ},'Inductor')

        switch cell2mat(Bondliste.Causality(Index))
```

```
            case 'f(e)'
                disp('------------ Zustand hinzugefügt:');
                x(end + 1, 1) = {[ 'p' num2str(Bondliste.BondID(Index))]}
                dx(end + 1, 1) = {[ 'dp' num2str(Bondliste.BondID(Index))]}
            case 'e(f)'
                error('Unzulässige Kausalität an Inductor: "e(f)"');
            case 'auto'
                warning('Kausalität an Inductor eingestellt: "f(e)"');
                Bondliste.Causality(Index) = {'f(e)'};
                disp('------------ Zustand hinzugefügt:');
                x(end + 1, 1) = {[ 'p' num2str(Bondliste.BondID(Index))]};
                dx(end + 1, 1) = {[ 'dp' num2str(Bondliste.BondID(Index))]}
                Bondliste2Cell( Bondliste )
            otherwise
                error('Unbekannte Kausalität!')
        end

    % Wenn Capacitor gefunden wird die Verschiebung q und deren Ableitung
    % mit der Bondnummer auf dx und x eingetragen
    elseif strcmp({Bondliste.Destination(Index).Typ},'Capacitor')

        switch cell2mat(Bondliste.Causality(Index))
            case 'f(e)'
                error('Unzulässige Kausalität an Capacitor: "f(e)"');
            case 'e(f)'
                disp('------------ Zustand hinzugefügt:');
                x(end + 1, 1) = {[ 'q' num2str(Bondliste.BondID(Index))]}
                dx(end + 1, 1) = {[ 'dq' num2str(Bondliste.BondID(Index))]}
            case 'auto'
                warning('Kausalität an Inductor eingestellt: "e(f)"');
                Bondliste.Causality(Index) = {'e(f)'};
                disp('------------ Zustand hinzugefügt:');
                x(end + 1, 1) = {[ 'q' num2str(Bondliste.BondID(Index))]};
                dx(end + 1, 1) = {[ 'dq' num2str(Bondliste.BondID(Index))]}
                Bondliste2Cell( Bondliste )
            otherwise
                error('Unbekannte Kausalität!')
        end
    end
end
assignin('base','x',x);    assignin('base','dx',dx);
end
```

B.2. Gleichungserstellung

B.2.1. erstelle_Gleichung()

```
function [ neueGleichungsnummer ] = erstelle_Gleichung( Variable,   ...
                                                         Gleichung,  ...
                                                         Bemerkung,  ...
                                                         Zielliste )
%function [ neueGleichungsnummer ] = erstelle_Gleichung( Variable,     ...
%                                                         Gleichung,    ...
%                                                         Bekanntheit,  ...
%                                                         [Zielliste]   )
%
%   Fügt die über die Eingabeparameter beschriebene Gleichung in die
%   angebene Zielliste ein und vergibt eine neue Gleichungsnummer.
%
%   Eingabeparameter:   1. Variable
%                       2. Gleichung
%                       3. Bemerkung
%                       4. Zielliste
%
%   Zielliste:  "Arbeitsliste"
%               "Grundliste"
%               "Ergebnistlise"
%
%   Bemerkung kann sich je nach Zielleiste unterscheiden:
%
%       Grundliste:     "Bilanz"
%                       "Gleichung"
%
%       Arbeitsliste/Ergebisliste:  "bekannt"
%                                   "unbekannt"
%
%   Aufbau der Listen:      Spalte  Inhalt              Datentyp
%                           1.      Gleichungsnummer    { double }
%                           2.      Variable            { sym  }
%                           3.      Gleichung           { sym  }
%                           4.      Bemerkung           { char }

%% Vorbereitung
if nargin ~= 4
    error('Unerwarte Anzahl an Eingabeparametern');
```

```
end

fprintf('---> AUFRUF: erstelle_Gleichung( %s, %s, %s, %s )\n', ...
    char(Variable), char(Gleichung), Bemerkung, Zielliste);

global Grundliste
global Ergebnisliste
global Arbeitsliste
global letzteGleichungsnummer

Variable = sym( Variable );
Gleichung = sym( Gleichung );
Bemerkung = char( Bemerkung );
Zielliste = char ( Zielliste );

if isempty(letzteGleichungsnummer)
    warning('Es wurde noch keine Gleichung erstellt!');
    letzteGleichungsnummer = 0;
end

%% Hinzufügen der Gleichung auf angegebene globale Zielliste
switch Zielliste

        case 'Arbeitsliste'
            if isempty(Arbeitsliste)
                warning('Arbeitsliste ist leer und wird neu erstellt!');
                Arbeitsliste = [];
                AnzahlVorkommen = 0;
            else
                Vorkommen = Arbeitsliste(:,2) == Variable;
                AnzahlVorkommen = sum(Vorkommen);
            end

            if AnzahlVorkommen == 0
                letzteGleichungsnummer = letzteGleichungsnummer + 1;

                Arbeitsliste{end+1,1} = letzteGleichungsnummer;
                Arbeitsliste{end  ,2} = Variable;
                Arbeitsliste{end  ,3} = Gleichung;
                Arbeitsliste{end  ,4} = Bemerkung;

                anzeigen_Liste( Arbeitsliste, 'Arbeitsliste' )
```

```
            teste_Korrektheit_Dopplungen( Grundliste, 'Grundliste' );

        elseif AnzahlVorkommen > 0
            anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');
            error('Variable %s kommt auf AL bereits %d-mal vor!', ...
                char(Variable), AnzahlVorkommen);
        end


    case 'Grundliste'
        if isempty(Grundliste)
            warning('Grundliste ist leer und wird neu erstellt!');
            Grundliste = [];
            AnzahlVorkommen = 0;
        else
            Vorkommen = Grundliste(:,2) == Variable;
            AnzahlVorkommen = sum(Vorkommen);
        end

        if AnzahlVorkommen == 0
            letzteGleichungsnummer = letzteGleichungsnummer + 1;

            Grundliste{end+1,1} = letzteGleichungsnummer;
            Grundliste{end  ,2} = Variable;
            Grundliste{end  ,3} = Gleichung;
            Grundliste{end  ,4} = Bemerkung;

            anzeigen_Liste( Grundliste, 'Grundliste' )
            teste_Korrektheit_Dopplungen( Grundliste, 'Grundliste' );

        elseif AnzahlVorkommen == 1
            fprintf(['Variable %s kommt auf GL bereits 1-mal vor! ' ...
                     'Bemerkungen werden geprüft...'], char(Variable));
            aktuelleBemerkung = Grundliste{Vorkommen,4};

            if strcmp(aktuelleBemerkung, Bemerkung)
                error('FEHLER: Beide Bemerkungen sind gleich....')
            else
                letzteGleichungsnummer = letzteGleichungsnummer + 1;

                Grundliste{end+1,1} = letzteGleichungsnummer;
                Grundliste{end  ,2} = Variable;
```

```
                Grundliste{end   ,3} = Gleichung;
                Grundliste{end   ,4} = Bemerkung;

                anzeigen_Liste( Grundliste, 'Grundliste' )
                teste_Korrektheit_Dopplungen(Grundliste, 'Grundliste');
                warning('Bemerkungen sind unterschiedlich! Weiter?')
                pause
            end

        elseif AnzahlVorkommen > 1
            anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');
            error('Variable %s kommt auf GL bereits %d-mal vor!', ...
                char(Variable), AnzahlVorkommen)

        elseif AnzahlVorkommen < 0
            anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');
            error(['Anzahl der Vorkommen in GL ist negativ! ' ...
                    '(Wert: %d)'], AnzahlVorkommen);

        end

    case 'Ergebnisliste'
        if isempty(Ergebnisliste)
            warning('Ergebnisliste ist leer und wird neu erstellt!');
            Ergebnisliste = [];
            AnzahlVorkommen = 0;
        else
            Vorkommen = Ergebnisliste(:,2) == Variable;
            AnzahlVorkommen = sum(Vorkommen);
        end

        if AnzahlVorkommen == 0
            letzteGleichungsnummer = letzteGleichungsnummer + 1;

            Ergebnisliste{end+1,1} = letzteGleichungsnummer;
            Ergebnisliste{end   ,2} = Variable;
            Ergebnisliste{end   ,3} = Gleichung;
            Ergebnisliste{end   ,4} = Bemerkung;

            anzeigen_Liste( Ergebnisliste, 'Ergebnisliste' )
            teste_Korrektheit_Dopplungen( ...
                Ergebnisliste, 'Ergebnisliste' );
```

```
            elseif AnzahlVorkommen > 0
                anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');
                error('Variable %s kommt auf EL bereits %d-mal vor!', ...
                    char(Variable), AnzahlVorkommen)

            elseif AnzahlVorkommen < 0
                anzeigen_Liste(Arbeitsliste, 'Arbeitsliste');
                error(['Anzahl der Vorkommen in EL ist negativ! ' ...
                        '(Wert: %d)'], AnzahlVorkommen);

            end

        otherwise
            error('Zielliste fehlerhaft angegeben!');
end

neueGleichungsnummer = letzteGleichungsnummer;

fprintf('<--- BEENDET: erstelle_Gleichung( %s, %s, %s, %s )\n', ...
    char(Variable), char(Gleichung), Bemerkung, Zielliste);
end
```

B.2.2. erstelle_Bondgleichungen()

```
function [  ] = erstelle_Bondgleichungen( BondID )
%[] = erstelle_Bondgleichungen( BondID )
%
%       Erstellt die Effort- und Flowgleichungen für den eingebenen Bond,
%       fügt diese entsprechend in die globale Ergebnisliste, Grundliste oder
%       Arbeitsliste ein.
%
%   Aufbau der Listen:      Spalte  Inhalt              Datentyp
%                           1.      Gleichungsnummer    { num }
%                           2.      Variable            { sym  }
%                           3.      Gleichung           { sym  }
%                           4.      Bemerkung           { char }
%
%   Bemerkung kann sich je nach Zielliste unterscheiden:
%
%       Grundliste:     "Bilanz"
%                       "Gleichung"
%
%       Arbeitsliste/Ergebisliste:  "bekannt"
%                                   "unbekannt"
%
%   Die Gleichungen werden anhand der Bondnummer und und der Elementenummer
%   erstellt.
%
%   Vorraussetzungen:   - globale Bondliste
%                       - globale Knotenliste
%                       - erstelle_Gleichung()
%
%   Eingabeparameter:   - BondID

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl an Eingabeparametern!');
end

global Bondliste
global Knotenliste


if isempty(Bondliste)
```

```
    error('Globale Bondliste is leer, existierte also nicht!')
end

if isempty(Knotenliste)
    error('Globale Knotenliste is leer, existierte also nicht!')
end

fprintf('AUFRUF: erstelle_Bondgleichung( %d )\n', BondID);


%% Hauptteil

% Kausalität des Bonds aus Bondliste ermitteln
Causality = Bondliste.Causality{Bondliste.BondID == BondID};

% ID und Typ der Senke ermitteln
DST_Typ = Bondliste.Destination(Bondliste.BondID == BondID).Typ;
DST_ID = Bondliste.Destination(Bondliste.BondID == BondID).ID;

% Fehlerüberprüfung für Angabe der Kausalität
if strcmp(Causality,'auto')
    error('Kausalität "auto" ist noch nicht implementiert!')
elseif ~( strcmp(Causality,'e(f)') || strcmp(Causality,'f(e)') )
    error('Falsche Kausalität!')
end

% Entsprechend dem Typ des Senkenelementes und der Kausalität des Bonds
% sowie der Bond-ID die entsprechende Gleichung symoblisch zusammensetzen
% und erstellen
switch DST_Typ

    case 'Resistor'
        Effortvariable = ['e' num2str(BondID) ];
        Flowvariable = ['f' num2str(BondID) ];
        Variable = ['R' num2str(DST_ID) ];

        if strcmp(Causality,'e(f)')
            erstelle_Gleichung( Effortvariable, ...
                [Variable '*' Flowvariable], 'Resistor', 'Grundliste' );

        elseif strcmp(Causality ,'f(e)')
            erstelle_Gleichung( Flowvariable, ...
```

```
            [Effortvariable '/' Variable], 'Resistor', 'Grundliste' );
    end

case 'Inductor'
    Effortvariable = ['p' num2str(BondID) ];
    Flowvariable = ['f' num2str(BondID) ];
    Variable = ['L' num2str(DST_ID) ];

    if strcmp(Causality,'e(f)')
        error('Inductor: Kausalität unzulässig!')

    elseif strcmp(Causality,'f(e)')
        erstelle_Gleichung( Flowvariable, ...
            [Effortvariable '/' Variable], 'bekannt', 'Ergebnisliste');
    end

case 'Capacitor'
    Effortvariable = ['e' num2str(BondID) ];
    Flowvariable = ['q' num2str(BondID) ];
    Variable = ['C' num2str(DST_ID) ];

    if strcmp(Causality,'e(f)')
        erstelle_Gleichung( Effortvariable, ...
            [Flowvariable '/' Variable ], 'bekannt', 'Ergebnisliste' );
    elseif strcmp(Causality,'f(e)')
        error('Capacitor: Kausalität unzulässig!');
    end

case 'Serieller Knoten'
    Effortvariable = ['e' num2str(BondID) ];
    Knotenzeile = find(Knotenliste.ID == DST_ID);

    % Anzahl der Ausgänge für Bilanz ermitteln
    [ Knoten_Hoehe Anzahl_Outputs ] = ...
        size(Knotenliste.Outputs(Knotenzeile,:)); %#ok<ASGLU>

    for OutputIndex = 1:Anzahl_Outputs
        if OutputIndex > 1
            Effortgleichung = Effortgleichung + ...
                sym(Knotenliste.Outputs(Knotenzeile,OutputIndex));

        elseif OutputIndex == 1
```

```
            Effortgleichung = ...
                sym(Knotenliste.Outputs(Knotenzeile,OutputIndex));
        end
    end

    [ Knoten_Hoehe Anzahl_Inputs ] = ...
        size(Knotenliste.Inputs(Knotenzeile,:));

    if Anzahl_Inputs > 1
        error('Nur ein Eingang möglich!')
    end

    erstelle_Gleichung( Effortvariable, Effortgleichung, ...
        'Bilanz', 'Grundliste')

case 'Paralleler Knoten'
    Flowvariable = ['f' num2str(BondID) ];
    Knotenzeile = find(Knotenliste.ID == DST_ID);

    [ Knoten_Hoehe Anzahl_Outputs ] = ...
        size(Knotenliste.Outputs(Knotenzeile,:)); %#ok<ASGLU>

    for OutputIndex = 1:Anzahl_Outputs
        if OutputIndex > 1
            Flowgleichung = Flowgleichung + ...
                sym(Knotenliste.Outputs(Knotenzeile,OutputIndex));

        elseif OutputIndex == 1
            Flowgleichung = sym(Knotenliste.Outputs(Knotenzeile,OutputIndex));
        end
    end

    [ Knoten_Hoehe Anzahl_Inputs ] = ...
        size(Knotenliste.Inputs(Knotenzeile,:));      %#ok<ASGLU>

    if Anzahl_Inputs > 1
        error('Nur ein Eingang!')
    end

    erstelle_Gleichung( Flowvariable, Flowgleichung, ...
        'Bilanz', 'Grundliste')
```

```
    case 'Transformer'
        EffortIn = sym(['e' num2str(BondID)]);
        FlowIn = sym(['f' num2str(BondID)]);

        Ausgangsbond = ermitteln_Post_Bond(DST_ID);

        EffortOut = sym(['e' num2str(Ausgangsbond)]);
        FlowOut = sym(['f' num2str(Ausgangsbond)]);

        Variable = sym( [ 'm' num2str(BondID) ] );

        erstelle_Gleichung( EffortIn, Variable*EffortOut, ...
            'Transformer', 'Grundliste')

        erstelle_Gleichung( FlowOut, Variable*FlowIn, ...
            'Transformer', 'Grundliste')

    case 'Gyrator'
        EffortIn = sym(['e' num2str(BondID)]);
        FlowIn = sym(['f' num2str(BondID)]);

        Ausgangsbond = ermitteln_Post_Bond(DST_ID);

        EffortOut = sym(['e' num2str(Ausgangsbond)]);
        FlowOut = sym(['f' num2str(Ausgangsbond)]);

        Variable = sym( [ 'r' num2str(BondID) ] );

        erstelle_Gleichung( EffortIn, Variable*FlowOut, ...
            'Gyrator', 'Grundliste')
        erstelle_Gleichung( EffortOut, Variable*FlowIn, ...
            'Gyrator', 'Grundliste')

    case 'Eingang'
        warning('Eingang wird nicht bearbeitet!')

    otherwise
        error('Elementtyp ist fehlerhaft')
end

fprintf('BEENDET: erstelle_Bondgleichung( %d )\n', BondID);
end
```

B.2.3. erstelle_Knotengleichungen()

```
function [] = erstelle_Knotengleichungen()
%ERSTELLE_KNOTENGLEICHUNGEN()
%
% Hilffunktion zur Erstellung der Bilanzen und Gleichnisse von Knoten in
% Abhängigkeit der Bondliste.
%
% Prinzip:
%   1. Element-ID von Knoten auswählen
%   2. Bond wählen und überprüfen, ob Knoten Quelle oder Senke ist
%       --> für jeden Bond durchführen und jeden Knoten durchführen
%   3. In Abhängigkeit des Knotentyps Effort und Flowvariablen in Bilanz
%      oder Gleichnis eintragen

%% Vorbereitung
global Bondliste
global Knotenliste

if isempty(Bondliste)
    error('Fehler: globale Eingangsvektor u existiert nicht')
end
if isempty(Bondliste)
    error('Fehler: globale Bondliste existiert nicht')
end
if isempty(Knotenliste)
    error('Fehler: globale Knotenliste existiert nicht')
end

%% Hauptteil: Nacheinander alle Knoten durchgehen
for Index_Knoten = 1:length(Knotenliste.ID)

% Hilfsvariablen initialisieren / zurücksetzen
Pos_Input = 1; Pos_Output = 1; Pos_Gleichnisse = 1;

% Anschließend nacheinander sämtliche Bonds der Bondliste durchgehen
% und überprüfen, ob der aktuelle Knoten am aktuellen Bond als Quelle
% oder Senke angeschlossn ist.
for Index_Bond = 1:length(Bondliste.BondID)

    %Wenn Knoten Quelle ist...
    if Bondliste.Source(Index_Bond).ID == Knotenliste.ID(Index_Knoten)
```

```
        if strcmp(Knotenliste.Typ(Index_Knoten), 'Serieller Knoten')
            Knotenliste.Gleichnisse(Index_Knoten, Pos_Gleichnisse) = ...
                {['f' num2str(Bondliste.BondID(Index_Bond))]};
            Knotenliste.Outputs(Index_Knoten, Pos_Output) = ...
                {['e' num2str(Bondliste.BondID(Index_Bond))]};

        elseif strcmp(Knotenliste.Typ(Index_Knoten), 'Paralleler Knoten')
            Knotenliste.Gleichnisse(Index_Knoten, Pos_Gleichnisse) = ...
                {['e' num2str(Bondliste.BondID(Index_Bond))]};
            Knotenliste.Outputs(Index_Knoten, Pos_Output) = ...
                {['f' num2str(Bondliste.BondID(Index_Bond))]};
        else
            error('Knoten %d: Typ ', Knotenliste.ID(Index_Knoten) );
        end

        Pos_Output = Pos_Output + 1;
        Pos_Gleichnisse = Pos_Gleichnisse + 1;

       %Wenn Knoten Senke ist...
    elseif Bondliste.Destination(Index_Bond).ID == Knotenliste.ID(Index_Knoten)
        if strcmp(Knotenliste.Typ(Index_Knoten), 'Serieller Knoten')
            Knotenliste.Gleichnisse(Index_Knoten, Pos_Gleichnisse) = ...
                {['f' num2str(Bondliste.BondID(Index_Bond))]};
            Knotenliste.Inputs(Index_Knoten, Pos_Input) = ...
                {['e' num2str(Bondliste.BondID(Index_Bond))]};

        elseif strcmp(Knotenliste.Typ(Index_Knoten), 'Paralleler Knoten')
            Knotenliste.Gleichnisse(Index_Knoten, Pos_Gleichnisse) = ...
                {['e' num2str(Bondliste.BondID(Index_Bond))]};
            Knotenliste.Inputs(Index_Knoten, Pos_Input) = ...
                {['f' num2str(Bondliste.BondID(Index_Bond))]};
        else
            error('Knoten %d: Typ ', Knotenliste.ID(Index_Knoten) );
        end

        Pos_Input = Pos_Input + 1;
        Pos_Gleichnisse = Pos_Gleichnisse + 1;
    end
end
end
end
```

B.2.4. eintragen_Ergebnis()

```
function [ ] = eintragen_Ergebnis( varargin )
%EINTRAGEN_ERGEBNIS ( Gleichungsnummer )
%
%   Die Funktion trägt eine Gleichung als ermitteltes Ergebnis von der
%   Arbeitsliste in die Ergebnisliste ein. Die Identifikation erfolgt
%   über die eingebene Gleichungsnummer.
%
%   Die Gleichung wird von AL auf EL kopiert ,wobei die Gleichungsnummer
%   übernommen wird. Die Variable und die zugehörige Gleichung werden nicht
%   freigeben oder von Arbeitsliste entfernt!
%
%   Vorraussetzungen:   - globale Arbeitsliste
%                       - globale Ergebnisliste
%
%   Eingabeparameter:   - Gleichungsnummer von AL
%                       - Bemerkung (für Gleichnis: "Gleichnis +KnotenID")

%% Vorbereitung
if nargin == 1
    Gleichungsnummer = varargin{1};
    Bemerkung = 'ermittelt';
elseif nargin == 2
    Gleichungsnummer = varargin{1};
    Bemerkung = varargin{2};
else
    error('Unerwartete Anzahl an Eingabeparametern!')
end

global Arbeitsliste
global Ergebnisliste

if isempty(Arbeitsliste)
    error('Arbeitsliste existiert nicht!')
end

% Zeilennr. von AL ermitteln
Zeile = transpose([Arbeitsliste{:,1}]) == Gleichungsnummer;

if sum(Zeile) == 0
    error('Gleichungsnummer nicht auf AL... doch kein Ergebnis?');
```

```
elseif sum(Zeile) > 1
    error('Gleichungsnummer mehrfach auf Arbeitsliste... Logikfehler');
end

if isempty(Ergebnisliste)
    warning('Ergebnisliste ist leer und wird neu erstellt!');
    Ergebnisliste = [];
    AnzahlVorkommen = 0;
else
    Vorkommen = transpose([Ergebnisliste{:,1}]) == Gleichungsnummer;
    AnzahlVorkommen = sum(Vorkommen);
end

if AnzahlVorkommen == 0
    Ergebnisliste{end+1,1} = Gleichungsnummer;
    Ergebnisliste{end  ,2} = Arbeitsliste{Zeile, 2};
    Ergebnisliste{end  ,3} = Arbeitsliste{Zeile, 3};
    Ergebnisliste{end  ,4} = Bemerkung;
    anzeigen_Liste(Ergebnisliste, 'Ergebnisliste');

elseif AnzahlVorkommen ~= 0
       error('Die Variable steht bereits auf der Ergebnisliste!');
end

teste_Korrektheit_Dopplungen( Ergebnisliste, 'Ergebnisliste' );
end
```

B.2.5. eintragen_Gleichung()

```
function [ ] = eintragen_Gleichung( Gleichungsnummer, Variable, ...
                                    Gleichung, Bemerkung)
%eintragen_Gleichung( Gleichungsnummer, Variable, Gleichung, Bemerkung)
%
%   Trägt übergebene Gleichung in die Arbeitsliste.

%% Vorbereitung
if nargin ~= 4
    error('Fehlerhafte Anzahl an Eingabeparameter!')
end

global Arbeitsliste
Variable = sym(Variable);
Gleichung = sym(Gleichung);
Bemerkung = char(Bemerkung);

if isempty(Arbeitsliste)
    warning('Arbeitsliste ist leer und wird neu erstellt!');
    Ergebnisliste = [];
    AnzahlVorkommen = 0;
else
    Vorkommen = Arbeitsliste(:,2) == Variable;
    AnzahlVorkommen = sum(Vorkommen);
end

%% Eintragung der Gleichung
if AnzahlVorkommen == 0
    Arbeitsliste{end+1,1} = Gleichungsnummer;
    Arbeitsliste{end  ,2} = Variable;
    Arbeitsliste{end  ,3} = Gleichung;
    Arbeitsliste{end  ,4} = Bemerkung;

elseif AnzahlVorkommen > 0 % eigentlich Fehler? --> manuell prüfen
    warning( 'Variable %s kommt auf AL bereits %d-mal vor!', ...
             char(Variable), AnzahlVorkommen )
    anzeigen_Liste(Arbeitsliste,'Arbeitsliste');
    pause
end
teste_Korrektheit_Dopplungen( Arbeitsliste, 'Arbeitsliste' );
end
```

B.2.6. kopieren_GL2AL()

```
function [ ] = kopieren_GL2AL( Gleichungsnummer)
%KOPIEREN_GL2AL ( Gleichungsnummer )
%
%   Die Funktion kopiert eine Gleichung von der Grundliste in Arbeitsliste.
%   Die Identifikation erfolgt durch die übergebene Gleichungsnummer.
%   Es erfolgt keine Sperrung.
%   Die Gleichungsnummer wird beibehalten.

%% Vorbereitung
if nargin ~= 1
    error('Fehlerhafte Anzahl an Eingabeparameter!');
end

global Arbeitsliste
global Grundliste

if isempty(Grundliste)
    error('Grundliste ist leer!');
end

%% Gleichung auf Grundliste ermitteln und auf Arbeitsliste einfügen
Vorkommen = transpose([Grundliste{:,1}]) == Gleichungsnummer;
AnzahlVorkommen = sum(Vorkommen);

% Arbeitsliste ggf. neu erstellen
if isempty(Arbeitsliste)
    Arbeitsliste = {};
end

if AnzahlVorkommen == 1
        Arbeitsliste(end+1,:) = Grundliste(Vorkommen,:);
elseif AnzahlVorkommen ~= 1
    error('Gleichungsnummer oder Grundliste fehlerhaft!')
end

% Arbeitsliste nach neuen Dopplungen durchsuchen
teste_Korrektheit_Dopplungen( Arbeitsliste, 'Arbeitsliste' );

end
```

B.3. Bereinigung

B.3.1. streichen_Arbeitsliste()

```
function [  ] = streichen_Arbeitsliste( Variable  )
%STREICHEN_ARBEITSLISTE ( Variable )
%
%   Entfernt die gesamte Gleichung von Arbeitsliste.


%% Vorbereitung
if nargin ~= 1
    error('Falsche Anzahl an Eingabeparameter!');
end

Variable = sym(Variable);
global Arbeitsliste

%% Gleichung von Arbeitsliste steichen
if ~isempty(Arbeitsliste)

    Treffer = transpose([Arbeitsliste{:,2}]) == Variable;

    if sum(Treffer) == 1
        Arbeitsliste(Treffer,:) = [];
        fprintf('Variable "%s" wurde mit Gleichung von AL entfernt!\n', ...
            char(Variable));
        anzeigen_Liste(Arbeitsliste,'Arbeitsliste');

    elseif sum(Treffer) == 0
        fprintf('Variable "%s" nicht auf Arbeitsliste\n', char(Variable));

    elseif sum(Treffer) > 1
        error('Variable kommt auf Arbeitsliste mehr als einmal vor!');
    end

elseif isempty(Arbeitsliste)
    warning('Arbeitsliste existiert nicht!')
end
end
```

B.3.2. streichen_Ergebnisliste()

```
function [  ] = streichen_Ergebnisliste( Variable  )
%STREICHEN_ERGEBNISLISTE ( Variable )
%
%   Entfernt die gesamte Gleichung von Ergebnisliste.

%% Vorbeireitung
if nargin ~= 1
    error('Unerwartete Anzahl an Eingabeparametern!')
end

Variable = sym(Variable);
global Ergebnisliste

%% Gleichung von Grundliste streichen
if ~isempty(Ergebnisliste)

    Treffer = Ergebnisliste(:,2) == Variable;

    if sum(Treffer) == 1
        Ergebnisliste(Treffer,:) = [];
        fprintf('Variable "%s" wurde von EL entfernt!\n', char(Variable));
        anzeigen_Liste(Ergebnisliste,'Ergebnisliste');

    elseif sum(Treffer) == 0
        error('Variable "%s" nicht auf Ergebnisliste!\n', char(Variable));

    elseif sum(Treffer) > 1
        error('Variable kommt auf Ergebnisliste mehr als einmal vor!');
    end

elseif isempty(Ergebnisliste)
    error('Ergebnisliste existiert nicht!');
end



end
```

B.4. Grapherstellung

B.4.1. registrieren_Bond()

```
function [  ] = registrieren_Bond()
%[  ] = registrieren_Bond()
%
%   Fügt sämtliche dem Element zugehörigen Bonds der Bondliste hinzu.
%
%   Die Funktion wird direkt von der S-Function des jeweiligen Elementes
%   ausgeführt. Es wird mit Hilfe von gcbh auf das RTO zugegriffen und die
%   benötigten Werte extrahiert.
%
%   Daten aus RTO:  - Anzahl der Inputs & Outputs
%                   - Vorgänger-Handle
%                   - Nachfolger-Handle
%
%   Die Funktion registrieren_Bond() ist die Vorstufe für die Funktionen
%   registrieren_Inputs() und registrieren_Outputs(), welche die
%   eingehenden und ausgehenden Bonds des Elementes in die Bondliste
%   eintragen.

%% Vorbereitung
    global Bondliste
    global Elementezahl

% Elementezahl überprüfen
    if isempty(Elementezahl)
        error('Elementezahl existiert nicht!');
    end

% Bondliste überprüfen und ggf. neu erstellen
    if ~isstruct(Bondliste)
        Bondliste.BondID = [];
        Bondliste.Inversion = boolean([]);
        Bondliste.Causality = {''};
        Bondliste.Gateway = boolean([]);
        Bondliste.Ausgang = boolean([]);

        Bondliste.Source.ID = [];
        Bondliste.Source.Handle = [];
        Bondliste.Source.Typ = '';
```

```
        Bondliste.Source.Name = '';
        Bondliste.Source.Wert = [];

        Bondliste.Destination.ID = [];
        Bondliste.Destination.Handle = [];
        Bondliste.Destination.Typ = '';
        Bondliste.Destination.Name = '';
        Bondliste.Destination.Wert = [];
    end

%% Hauptteil
% RTO mit gcbh laden
    lokalesElement = get(gcbh);

% Handles der Vorgänger aus RTO laden
    InputHandles = [lokalesElement.PortConnectivity.SrcBlock]';

% Sicherstellen das mindestens ein Vorgänger vorhanden ist.
    if sum(InputHandles == -1) >= 1
        error('Einige Eingangsports sind nicht verbunden!');
    end

% Handles der Nachfolger laden.
    OutputHandles = [lokalesElement.PortConnectivity.DstBlock]';
    if isempty(OutputHandles)
        OutputHandles = [];
    end

% Inputbonds hinzufügen
    Bondliste = registrieren_Inputs( Bondliste, ...
        InputHandles, lokalesElement );

% Outputbonds hinzufügen
    Bondliste  = registrieren_Outputs( Bondliste, ...
        OutputHandles, lokalesElement );
end
```

B.4.2. registrieren_Element()

```
function [ ] = registrieren_Element( Element, Modus )
%[] = REGISTRIEREN_ELENENT (Element, Modus)
%
%   Fügt das übergebene Element der Elementeliste hinzu. Es wird unter-
%   schieden, ob das Element wie ein normales Bondelement während der
%   Analyse behandelt werden soll oder ob es später als Signalein- bzw.
%   ausgang im Modell verwendet werden soll.
%
%   ACHTUNG: Die Funktionalität für die Verwendung der Ein- und Ausgänge im
%   Modell ist noch nicht implementiert und sondern nur vorbereitet!
%
%   Modus:
%       'intern' -  Verwendet die in der Struktur "Element"
%                   spezifizierten Eigenschaften
%
%       'Ausgang' - Fügt Element Markierungen für Verwendung als Ausgang
%                   im Simulinkmodell hinzu: - neue Elementenummer als ID
%                                            - Typ wird 'Ausgang'
%                                            - Wert wird 0
%
%       'Eingang'- Analog 'Ausgang'

%% Vorbereitung
if nargin ~= 2
    error('Unerwartete Anzahl an Eingabeparametern');
end

global Elementeliste
global Elementezahl

% if isempty(Elementeliste)
%     error('Elementeliste ist leer!');
% elseif isempty(Elementezahl)
%     error('Elementezahl existiert nicht!');
% end

%% Hauptteil

    % Normales Element hinzufügen
    if strcmp(Modus,'intern')
```

```
        Elementeliste(Element.ID).ID = Element.ID;
        Elementeliste(Element.ID).Name = Element.Name;
        Elementeliste(Element.ID).Typ = Element.Typ;
        Elementeliste(Element.ID).Wert = Element.Wert;
        Elementeliste(Element.ID).Handle = Element.Handle;

    % Eingang / Ausgang hinzufügen
    elseif (strcmp(Modus,'Eingang') || strcmp(Modus,'Ausgang'))

        Elementezahl = Elementezahl + 1;

        Elementeliste(Elementezahl).ID = Elementezahl;
        Elementeliste(Elementezahl).Name = Element.Name;
        Elementeliste(Elementezahl).Typ = Modus;
        Elementeliste(Elementezahl).Wert = 0;
        Elementeliste(Elementezahl).Handle = Element.Handle;

    else
        error('Falscher Modus bei "registrieren_Element( Data, Modus )"');
    end
end
```

B.4.3. registrieren_Inputs()

```
function [ Bondliste ] = registrieren_Inputs( Bondliste, ...
                                              InputHandles, lokalesElement)
%[ Bondliste ] = registrieren_Inputs( Bondliste, ...
%                                     InputHandles, lokalesElement)
%
%   Die Funktion fügt für das übergebene Element die zugehörigen Bond auf
%   der übergebenen Bondliste ein.

%% Vorbereitung
if nargin ~= 3
    error('Unerwartete Anzahl an Ausgabeparametern!');
end

global registrierteBonds
global Elementeliste

%% Hauptteil
% Übergebene Input-Handles nacheinander durchgehen
for Index = 1:1:length(InputHandles)

    % externes Element überprüfen und ggf. registrieren.
    externesElement = get(InputHandles(Index,1));

    if isstruct(externesElement.UserData)
        if isfield(externesElement.UserData,'Toolbox')
            if strcmp(externesElement.UserData.Toolbox,'Bondgraph')
                % --> externes Element MUSS folglich Bondgraphenelement sein!

                %Bond erstellen und konfigurieren
                Bondliste.BondID(end+1,1) = length(Bondliste.BondID) + 1;
                Bondliste.Inversion{Bondliste.BondID(end,1),1} = ...
                    lokalesElement.UserData.Invertierung;
                Bondliste.Causality{Bondliste.BondID(end,1),1} = ...
                    lokalesElement.UserData.Causality;
                Bondliste.Gateway(Bondliste.BondID(end,1),1) = boolean(0);
                Bondliste.Ausgang(Bondliste.BondID(end,1),1) = ...
                    lokalesElement.UserData.Ausgang;

                % Ziel-Element eintragen (lokales Element)
                Bondliste.Destination(Bondliste.BondID(end,1),1).Handle=...
```

```
                gcbh;
            Bondliste.Destination(Bondliste.BondID(end,1),1).ID = ...
                lokalesElement.UserData.ID;
            Bondliste.Destination(Bondliste.BondID(end,1),1).Typ = ...
                lokalesElement.UserData.Typ;
            Bondliste.Destination(Bondliste.BondID(end,1),1).Name = ...
                lokalesElement.UserData.Name;
            Bondliste.Destination(Bondliste.BondID(end,1),1).Wert = ...
                lokalesElement.UserData.Wert;

            % Quell-Element eintragen (externes Element)
            Bondliste.Source(Bondliste.BondID(end,1),1).Handle = ...
                externesElement.Handle;
            Bondliste.Source(Bondliste.BondID(end,1),1).ID = ...
                externesElement.UserData.ID;
            Bondliste.Source(Bondliste.BondID(end,1),1).Typ = ...
                externesElement.UserData.Typ;
            Bondliste.Source(Bondliste.BondID(end,1),1).Name = ...
                externesElement.Name;
            Bondliste.Source(Bondliste.BondID(end,1),1).Wert = ...
                externesElement.UserData.Wert;

            if ~isempty(registrierteBonds)
                registrierteBonds = registrierteBonds + 1;
            elseif isempty(registrierteBonds)
                registrierteBonds =  1;
            end
            continue
        end
    end
elseif ~isstruct(externesElement.UserData)
    fprintf(['\nEingangselement mit Handle %f an Element %s ' ...
             'mit Nummer %d ist kein Bondgraphenelement!\n\n'], ...
             lokalesElement.Name, InputHandles(Index,1), ...
             lokalesElement.UserData.ID)

    if strcmp(lokalesElement.UserData.Kategorie,'Knoten') || ...
            strcmp(lokalesElement.UserData.Kategorie,'Leitung')
        fprintf('\nElement Nr. %d darf Eingang in Bond sein!\n\n', ...
            lokalesElement.UserData.ID)
    else
        error('Element Nr. %d kann kein Eingang sein!', ...
```

```
        lokalesElement.UserData.ID );
end

% Eingang auf Elementeliste registrieren
registrieren_Element( externesElement, 'Eingang' )

if strcmp(lokalesElement.UserData.Causality,'auto')
    error(['Kausalität von Bond mit DST "', ...
           lokalesElement.UserData.Name, ...
           '" hat Kausalität "auto"! Bei externen SRC' ...
           ' muss Kausalität e(f) oder f(e) sein!']);
end

%Bond erstellen und konfigurieren
Bondliste.BondID(end+1,1) = length(Bondliste.BondID) + 1;
Bondliste.Inversion{Bondliste.BondID(end,1),1} = ...
    lokalesElement.UserData.Invertierung;
Bondliste.Causality{Bondliste.BondID(end,1),1} = ...
    lokalesElement.UserData.Causality;
Bondliste.Gateway(Bondliste.BondID(end,1),1) = ...
    boolean(0);
Bondliste.Ausgang(Bondliste.BondID(end,1),1) = ...
    lokalesElement.UserData.Ausgang;

% Ziel-Element eintragen (lokales Element)
Bondliste.Destination(Bondliste.BondID(end,1),1).Handle = gcbh;
Bondliste.Destination(Bondliste.BondID(end,1),1).ID = ...
    lokalesElement.UserData.ID;
Bondliste.Destination(Bondliste.BondID(end,1),1).Typ = ...
    lokalesElement.UserData.Typ;
Bondliste.Destination(Bondliste.BondID(end,1),1).Name = ...
    lokalesElement.UserData.Name;
Bondliste.Destination(Bondliste.BondID(end,1),1).Wert = ...
    lokalesElement.UserData.Wert;

% Quell-Element eintragen (externes Element)
Bondliste.Source(Bondliste.BondID(end,1),1).Handle = ...
    Elementeliste(end).Handle;
Bondliste.Source(Bondliste.BondID(end,1),1).ID = ...
    Elementeliste(end).ID;
Bondliste.Source(Bondliste.BondID(end,1),1).Typ = ...
    Elementeliste(end).Typ;
```

```
        Bondliste.Source(Bondliste.BondID(end,1),1).Name = ...
            Elementeliste(end).Name;
        Bondliste.Source(Bondliste.BondID(end,1),1).Wert = ...
            Elementeliste(end).Wert;
        Bondliste.Gateway(Bondliste.BondID(end,1),1) = boolean(1);

        if ~isempty(registrierteBonds)
            registrierteBonds = registrierteBonds + 1;
        elseif isempty(registrierteBonds)
            registrierteBonds =  1;
        else
            error('Irgendwas stimmt mit "registrierteBonds" nicht!');
        end
    end
end
end
```

B.4.4. registrieren_Outputs()

```
function [ Bondliste ] = registrieren_Outputs( Bondliste, ...
    OutputHandles, lokalesElement )
% [ Bondliste ] = registrieren_Outputs( Bondliste, ...
%   OutputHandles, lokalesElement )
%
%   AUSGÄNGE IN SIMULINKMODELL SIND MOMENTAN NICHT IMPLEMENTIERT, DIE
%   FUNKTION IST DAHER NUR EIN PLATZHALTER

global Elementeliste
global registrierteBonds

for Index = 1:1:length(OutputHandles)

    externesElement = get(OutputHandles(Index,1));

    if ~isstruct(externesElement.UserData)
        warning('Ausgänge sind in dieser Version nicht implementiert!')
        continue
    end
end

end
```

C. Hilfsfunktionen

C.1. Darstellung

C.1.1. anzeigen_Liste()

```
function [ ] = anzeigen_Liste( Liste, Listenbezeichnung )
% anzeigen_Liste( Liste, Listenbezeichnung )
%
% Zeigt übergebene Liste formatiert auf Kommandozeile an und speichert
% diese im Base-Workspace

if isempty(Liste)
    warning('Liste "%s "ist leer! Es wird nichts angezeigt!\n', ...
        upper(Listenbezeichnung));
    assignin('base', [ 'Zelle_' Listenbezeichnung ], sym([]) );
else
    Zelle = sym([]);

    fprintf('Angezeigte Liste: %s \n', upper(Listenbezeichnung))

    if ~strcmp(Listenbezeichnung,'Variablenstatus')
        for Index = 1:length(Liste(:,1))
            Zelle(Index,1) = Liste{Index,1};    % Gleichungsnummer
            Zelle(Index,2) = Liste{Index,2};    % Variable
            Zelle(Index,3) = Liste{Index,3};    % Gleichung
            Zelle(Index,4) = Liste{Index,4};    % Bemerkung
        end
    elseif strcmp(Listenbezeichnung,'Variablenstatus')
        for Index = 1:length(Liste(:,1))
            Zelle(Index,1) = Liste{Index,1};    % Variablenname
            Zelle(Index,2) = Liste{Index,2};    % Freigabe: implizite Suche
            Zelle(Index,3) = Liste{Index,3};    % implizites Ergebnis
        end
    end
```

```
    assignin('base', ['Zelle_' Listenbezeichnung], Zelle);
    Zelle
end
end
```

C.1.2. Bondliste2Cell()

```
function [ Bondzelle ] = Bondliste2Cell( Bondliste )
%[ Bondzelle ] = Bondliste2Cell( Bondliste )
%
% Die Funktion konvertiert die übergebene Bondliste ( Struktur ) in den
% Datentyp cell, um eine bessere Lesbarkeit zu ermöglichen und legt das
% Ergebnis im base-Workspace ab

%% Vorbereitung
    if nargin ~= 1
        error('Unerwartete Anzahl an Eingabeparametern!')
    end

%% Hauptteil

    % "Überschriften" in erste Zeile der Zelle
    Bondzelle{1,1} = 'Bond';
    Bondzelle{1,2} = 'Inv';
    Bondzelle{1,3} = 'Causality';
    Bondzelle{1,4} = 'Ausgang';
    Bondzelle{1,5} = 'Gateway';
    Bondzelle{1,6} = 'SourceID';
    Bondzelle{1,7} = 'DestID';
    Bondzelle{1,8} = 'SourceTyp';
    Bondzelle{1,9} = 'DestTyp';

    % Bondliste in Zelle übertragen
    for Index = 1:1:length(Bondliste.BondID)
        Bondzelle{Index+1,1} = Bondliste.BondID(Index);
        Bondzelle{Index+1,2} = Bondliste.Inversion{Index};
        Bondzelle{Index+1,3} = Bondliste.Causality{Index};
        Bondzelle{Index+1,4} = Bondliste.Ausgang(Index);
        Bondzelle{Index+1,5} = Bondliste.Gateway(Index);
        Bondzelle{Index+1,6} = Bondliste.Source(Index).ID;
        Bondzelle{Index+1,7} = Bondliste.Destination(Index).ID;
        Bondzelle{Index+1,8} = Bondliste.Source(Index).Typ;
        Bondzelle{Index+1,9} = Bondliste.Destination(Index).Typ;
    end

    assignin('base', 'Bondzelle', Bondzelle)

end
```

C.2. Validierung

C.2.1. teste_Korrektheit_Dopplungen()

```
function [ ] = teste_Korrektheit_Dopplungen( Liste, Listenbezeichnung )
%teste_Korrektheit_Dopplungen ( Liste, Listenbezeichnung )
%
%   Die Funktion untersucht die übergebene Liste nach Dopplungen in
%   Die Funktion überprüft ob Variablen auf der übergenen Liste
%   mehrfach vorkommen. Sollte dies der Fall sein, wird mit Fehler
%   abgebrochen. Es existieren folgenden Ausnhamen:
%
%   Variablen dürfen auf der Ergebnisliste nur einmal vorkommen.

%% Vorbereitung
if nargin ~= 2
    error('Unerwartete Anzahl an Eingabeparameter!');
end

fprintf('Teste "%s" auf Dopplungen ... ', upper(Listenbezeichnung) );

Variablen = sort( transpose( [ Liste{:,2} ] ) );
Einzigartige = unique( Variablen );

Dopplungen = sym([]);

%% Hauptteil
% Wenn Anzahl der "Einzigartigen" sich von der Anzahl der Variablen
% unterscheidet, muss mindestens eine Dopplung vorliegen!
if length(Variablen) ~= length(Einzigartige)
        fprintf('Es liegen Dopplungen vor!\n');

    % Doppelte Variablen ermitteln
    for i = 1:length( Einzigartige )
        Vorkommen = Einzigartige(i) == Variablen;
        Anzahlvorkommen = sum(Vorkommen);

        if Anzahlvorkommen == 2
            fprintf('Variable %s kommt doppelt vor!\n', ...
                char(Einzigartige(i)) );
            Dopplungen(end+1) = Einzigartige(i); %#ok<AGROW>
        elseif Anzahlvorkommen > 2
```

```
            error('Variable %s kommt auf %s mehr als zweimal vor!', ...
                char(Einzigartige(i)), upper(Listenbezeichnung));
        end
    end

    for i = 1:length(Dopplungen(:,1))
        fprintf('Dopplung %s wird untersucht!\n', char(Dopplungen(i)) )

        Zeilen = Liste(:,2) == Dopplungen(:,1);
        Subliste = Liste(Zeilen, :);

        anzeigen_Liste(Subliste,'Subliste');

        if strcmp(Subliste{1,4}, Subliste{2,4})
            error('Variable %s ist FEHLERHAFTE DOPPLUNG!', ...
                char(Einzigartige(i)));
        else
            fprintf('Variable %s ist OK!\n', char(Einzigartige(i)) );
        end
    end

else
    fprintf('... keine unzulässigen Dopplungen! \n');
end
end
```

C.2.2. teste_Korrektheit_Gleichnis()

```
function [ Konfliktvariablen ] = teste_Korrektheit_Gleichnis( ...
                                                    Testliste_Gleichnis )
%TESTE_KORREKTHEIT_GLEICHNIS ( Testliste_Gleichnis )
%
%   Die Funktion überprüft die Lösungen der bekannten Variablen der über-
%   gebenen Testliste. Liefern Variablen unterschiedliche Lösungen, werden
%   die Bezeichner dieser Konfliktvariablen zurückgegeben.

%% Vorbereitung
if nargin ~= 1
    error('Falsche Anzahl an Eingabeparameter!');
end
    Konfliktvariablen = sym([]);

fprintf('Überprüfe Lösungen des übergebenen Gleichnisses ... \n');

%% Hauptteil
for i = 1:1:length(Testliste_Gleichnis(:,1))
    Testgleichung = Testliste_Gleichnis{i,3};

    for j = 1:1:length(Testliste_Gleichnis(:,1))
        if i == j
            continue
        end
        if Testgleichung ~= Testliste_Gleichnis(j,3)

            Konfliktvariablen(end+1, 1) = Testliste_Gleichnis(i,2);
            Konfliktvariablen(end+1, 1) = Testliste_Gleichnis(j,2);

            anzeigen_Liste(Testliste_Gleichnis, 'Testliste_Gleichnis')
            disp('Es wurde ein Konflikt gefunden!')
        end
    end
end

Konfliktvariablen = unique(Konfliktvariablen)

end
```

C.3. Informationen

C.3.1. ermitteln_Bilanzen()

```
function [ Bilanznummern ] = ermitteln_Bilanzen( Variable )
%ERMITTELN_BILANZEN ( VARIABLE )
%
%   Ermittelt basierend auf der Grundliste die Bilanzen, in deren
%   Gleichungsrümpfen die übergebene Variable vorkommt.
%   Explizite Suche auf GL sollte vorher mit durchsuche_Grundliste()
%   durchgeführt werden.
%
%   Es werden die Gleichungsnummern der Bilanzen zurückgegeben.

if nargin ~= 1
    error('Falsche Anzahl an Eingabeparametern!');
end

global Grundliste

if isempty(Grundliste)
    error('Grundliste ist leer!');
end
Variable = sym(Variable);
Bilanznummern = [];

% Vollständiges Vorkommen auf GL ermitteln
Gleichungsnummern = ermitteln_Vorkommen(Variable,Grundliste,'Grundliste');

% Alles außer Bilanzen aussortieren
for i = 1:1:length(Grundliste(:,1))
    if sum(Gleichungsnummern == Grundliste{i,1}) == 1
        if strcmp( Grundliste{i,4}, 'Bilanz' )
            Bilanznummern(end+1,1) = Grundliste{i,1};
        end
    end
end
end
```

C.3.2. ermitteln_DST()

```
function [ DST_ID ] = ermitteln_DST( Variable, Bondliste )
%function [ DST_ID ] = ermitteln_DST( Variable, Bondliste )
%
%   Die Funktion ermittelt für den übergebenen Variablenbezeichern die ID
%   des zugehörigen Senkenelementes
%
%   Input:  - Variable
%           - Bondliste
%
%   Output: - ID der Senke

%% Vorbereitung
    if nargin ~= 2
        error('Unerwartete Anzahl an Eingabeparameter!');
    end

%% Hauptteil
    % BondID der Variable extrahieren
    BondID = ermittleln_VarID(Variable);

    % ID der Senke aus zugehöriger Zeile der Bondliste ermitteln
    DST_ID = Bondliste.Destination(BondID).ID;
end
```

C.3.3. ermitteln_DST_Typ()

```
function [ DST_Type ] = ermitteln_DST_Typ( Variable, Bondliste )
%function [ DST_Typ ] = ermitteln_DST_Typ( Variable, Bondliste )
%
%   Die Funktion ermittelt für den übergebenen Variablenbezeichern den Typ
%   des zugehörigen Senkenelementes
%
%   Input:  - Variable
%           - Bondliste
%
%   Output: - ID der Senke

%% Vorbereitung
    if nargin ~= 2
        error('Unerwartete Anzahl an Eingabeparameter!');
    end

    global Elementeliste

    if isempty(Elementeliste)
        error('Elementeliste existiert nicht!')
    end

%% Hauptteil

    % ID der Senke ermitteln
    DST_ID = ermitteln_DST( Variable, Bondliste );

    % Typ der Senke ermitteln
    DST_Type = Elementeliste(DST_ID).Typ;
end
```

C.3.4. ermitteln_Gleichnisse()

```
function [ VektorKnotenID ] = ermitteln_Gleichnisse( Variable )
%[ VektorKnotenID ] = ERMITTELN_GLEICHNISSE ( Variable )
%
%   Ermittelt basierend auf der Knotenliste die IDs aller Knoten, an
%   welchen die übergebene Variable in Gleichnissen vorkommt.
%
%   Die zutreffenden KnotenIDs werden als (Spalten-)Vektor zurückgegeben.

%% Vorbereitung
if nargin ~= 1
    error('Falsche Anzahl an Eingabeparametern!');
end

global Knotenliste
Variable = sym(Variable);
VektorKnotenID = [];

if isempty(Knotenliste)
    error('Knotenliste ist leer!');
end

%% Knoten ermitteln
for i = 1:1: length(Knotenliste.ID)
    aktuellesGleichnis = Knotenliste.Gleichnisse(i ,:);
    Treffer = sum( aktuellesGleichnis == Variable ) == 1;

    if sum( Treffer ) == 1
        fprintf('Variable %s in Gleichnis von Knoten %d gefunden!\n', ...
            char(Variable), Knotenliste.ID(i));
        VektorKnotenID(end+1,1) = Knotenliste.ID(i);
    elseif sum( Treffer ) > 1
        error('Variable %s mehrfach in Gleichnis an KnotenID %d!', ...
            char(Variable), Knotenliste.ID(i));
    end
end

if isempty(VektorKnotenID)
    fprintf('Variable %s kommt in keinem Gleichnis vor!\n',char(Variable));
end
end
```

C.3.5. ermitteln_Gleichungsnummer()

```
function [ Gleichungsnummer ] = ermitteln_Gleichungsnummer( Variable, ...
                                                            Gleichungstyp )
%ERMITTELN_GLEICHUNGSNUMMER ( Variable, Gleichungstyp )
%
%   Ermittelt für die übergebene Variable die zugehörige Gleichungsnummer.
%   Es wird nur die Gleichungsnummer ausgeben, welche dem übergebenem
%   Gleichungstyp entspricht.
%
%   Die Auswahl des Gleichungstyp ist nur für Gleichungen auf Grundliste
%   relevant, da nur auf dieser Liste mehrere Gleichungen eine Variable
%   explizit beschreiben könneb. Bei Gleichungen auf Arbeitsliste und
%   Ergebnisliste gilt Gleichungstyp "egal".
%
%   Gleichungstypen:    - Bilanz
%                       - egal
%                       - Verbraucher (R, L, C, G, T)
%
%   Wird nur eine Gleichung gefunden und ist der Gleichungstyp mit "egal"
%   angegeben, wird diese Gleichungsnummer ausgeben.
%
%   !!! Für die Verwendung in der Funktion freigeben(Variable) wird die
%   !!! Gleichungsnummer aus der globalen Arbeitsliste ermittelt. Wenn die
%   !!! Variable dort nicht vorhanden sein sollte, wird die Variable aus
%   !!! der Grundliste ermittelt, wobei nur Elementgleichungen und keine
%   !!! Bilanzen verwendet werden dürfen.
%
%   Suchreihenfolge:    1. Arbeitsliste
%                       2. Grundliste (ohne Bilanzen)
%                       3. Ergebnisliste

%% Vorbereitung
if nargin ~= 2
    error('Anzahl der Eingabeparameter ist falsch!')
end

fprintf( 'AUFRUF: ermitteln_Gleichungsnummer( "%s", "%s" )\n', ...
         char(Variable), Gleichungstyp );

global Arbeitsliste
global Grundliste
```

```
global Ergebnisliste

Gleichungsnummer = [];
Variable = sym ( Variable );

%% Arbeitsliste durchsuchen
    if ~isempty(Arbeitsliste) % Wenn AL existiert...
        Position = Arbeitsliste(:,2) == Variable;

        % Wenn gefunden...Gl-Nr. zurückgeben
        if sum(Position) == 1
            fprintf('--- Die Variable wurde auf Arbeitsliste gefunden:\n')
            Gleichungsnummer = Arbeitsliste{Position,1}
            return

        % Kein Treffer...weiter mit Grundliste
        elseif sum(Position) == 0
            fprintf('--- Variable nicht auf Arbeitsliste gefunden!...\n')

        % Fehler bei Vielfachtreffer
        elseif sum(Position) > 1 || sum(Position) < 0
            error('Die Arbeitsliste ist fehlerhaft')
        end

    % Keine Arbeitsliste vorhanden
    elseif isempty(Arbeitsliste)
        fprintf('-- Arbeistliste ist leer / existiert nicht.\n')

    end

    clear Position

%% Grundliste durchsuchen
    if ~isempty(Grundliste) % Wenn GL existiert
        Position = Grundliste(:,2) == Variable;

        if sum(Position) == 1 %Wenn gefunden...Gl-Nr. zurückgeben
            fprintf('--- Die Variable wurde auf Grundliste gefunden:\n')

            if strcmp(Gleichungstyp, 'egal')
                Gleichungsnummer = Grundliste{Position,1};
                return
```

```
        elseif ( strcmp(Gleichungstyp, 'Bilanz') && ...
                 strcmp(Grundliste{Position,4}, 'Bilanz') )
            Gleichungsnummer = Grundliste{Position,1};
            return

        elseif ( strcmp(Gleichungstyp, 'Verbraucher') && ...
                 ~strcmp(Grundliste{Position,4}, 'Bilanz') )
            Gleichungsnummer = Grundliste{Position,1};
            return
        end

    % Wenn 2x gefunden... beide Gleichungsnummern zurückgeben
    elseif sum(Position) == 2
        fprintf('--- Die Variable wurde 2x auf Grundliste gefunden:\n')

        Subliste = Grundliste(Position,:);

        for i = 1:2
            if (strcmp(Gleichungstyp, 'Bilanz') && ...
                    strcmp(Subliste{i,4}, 'Bilanz') )
                Gleichungsnummer = Subliste{i,1};
                return

            elseif (strcmp(Gleichungstyp, 'Verbraucher') && ...
                    ~strcmp(Subliste{i,4}, 'Bilanz') )
                Gleichungsnummer = Subliste{i,1};
                return
            end
        end

    elseif sum(Position) == 0 %Kein Treffer... leere Rückgabe
        fprintf('--- Variable nicht auf Grundliste gefunden!...\n')

    elseif sum(Position) > 2 % Fehler bei mehr als 2 Treffern
        error('Die Variable wurde in Grundliste mehrfach gefunden!')

    end
elseif isempty(Arbeitsliste) % Keine GL, leere Rückgabe
    fprintf('-- Arbeitsliste ist existiert nicht.\n')

end
```

```
    clear Position

%% Ergebnisliste durchsuchen
    if ~isempty(Ergebnisliste) % Wenn GL existiert
        fprintf('-- Ergebnisliste existiert... wird durchsucht!\n')

        Position = Ergebnisliste(:,2) == Variable;

        if sum(Position) == 1 %Wenn gefunden...Gl-Nr. zurückgeben
            fprintf('--- Die Variable wurde in Ergebnisliste gefunden:\n')
            Gleichungsnummer = Ergebnisliste{Position,1};
            return

        elseif sum(Position) == 0 %Kein Treffer... leere Rückgabe
            fprintf('--- Variable nicht auf Ergebnisliste gefunden!...\n')

        elseif sum(Position) > 1 || sum(Position) < 0 % Fehler bei Vielfachtreffer
            error('Variable mehrfach auf Ergebnisliste gefunden!')
        end

    elseif isempty(Ergebnisliste) % Keine GL, leere Rückgabe
        fprintf('-- Ergebnisliste ist existiert nicht. Leere Rückgabe\n')
    end

%% Ende ... Wenn Funktion bis hierher gekommen ist, kein Treffer
    fprintf('-- Zugehörige Gleichungsnummer nicht gefunden!\n')
    fprintf( 'BEENDET: ermitteln_Gleichungsnummer( "%s", "%s" )\n', ...
             char(Variable), Gleichungstyp );
end
```

C.3.6. ermitteln_Gleichnisvariablen()

```
function [ Gleichnisvariablen ] = ermitteln_Gleichnisvariablen( Variable )
%Gleichnisvariablen = ERMITTELN_GLEICHNISSVARIABLEN ( Variable )
%
%   Durchsucht die Gleichnisse der Knoten nach Vorkommen der übergebenen
%   Variable und gibt das Gleichnis in symbolischer Form als Spalten-
%   vektor zurück. Die Verkettung von Knoten wird berücksichtigt.

%% Vorbereitung
if nargin ~= 1
    error('Falsche Anzahl an Eingabeparametern!');
end

global Knotenliste

if isempty(Knotenliste)
    error('Knotenliste ist leer!');
end

Variable = sym(Variable);
Variablen = sym([]);
Gleichnisvariablen = sym([]);
unbenutzteVariablen = sym([]);
VektorKnotenID = [];

%% Knoten ermitteln
    for i = 1:1: length(Knotenliste.ID)
        aktuellesGleichnis = Knotenliste.Gleichnisse(i ,:);
        Treffer = sum( aktuellesGleichnis == Variable ) == 1;

        if sum( Treffer ) == 1
            fprintf(['... Variable in Gleichnis von Knoten ' ...
                    '%d gefunden!\n'], Knotenliste.ID(i));
            VektorKnotenID(end+1,1) = Knotenliste.ID(i);
        elseif sum( Treffer ) > 1
            error(['Variable kommt in Gleichnis an KnotenID ' ...
                   '%d mehrfach vor!'], Knotenliste.ID(i));
        end
    end

% Gefundene Knoten überprüfen: keine Knoten = return
```

```
    if isempty(VektorKnotenID)
        fprintf('Variable %s kommt in keinem Gleichnis vor!\n', ...
                char(Variable));
        return
    end

% Vararіablen der Knoten laden und in Vektor zusammenfassen
    for i = 1:length(VektorKnotenID)
        Zeile = Knotenliste.ID == VektorKnotenID(i,1);
        Variablen = [ Variablen Knotenliste.Gleichnisse(Zeile,:) ];
    end

% Variablenvektor bereinigen: Dopplungen entfernen
    Variablen = unique(Variablen);
    Variablen(Variablen == sym('0')) = [];

% in Gleichnissen nicht verwendete Knoten ermitteln
    unbenutzteKnoten = setdiff(Knotenliste.ID, VektorKnotenID);

% Gleichnisse der nicht verwendeten Knoten mit den ermittelten Variablen
% nach Zusammenhängen untersuchen:
% Sollte eine der ermittelten Variablen im Gleichnis eines bisher nicht
% verwendeten Knoten vorkommen, muss dieses Gleichnis mitverwendet werden.
% Hierfür werden die zusätzlichen Gleichnisvariablen in einem separaten
% Vektor gespeichert, später wird dieser mit dem Variablenvektor zusammen
% gefasst und der Ableich erneut gestartet.
% Ziel ist die vollständige Erfassung belieb lang verketteter Knoten.
disp('Weitere Gleichnisse durchsuchen...');

    while i <= length(unbenutzteKnoten)
        Zeile = Knotenliste.ID == unbenutzteKnoten(i,1);
        unbenutztesGleichnis = Knotenliste.Gleichnisse(Zeile,:);

        for j = 1:length(Variablen)
            Vorkommen = unbenutztesGleichnis == Variablen(j);

            if sum(Vorkommen) == 1
                fprintf(['- Variable %s kommt auch noch an ' ...
                         'Knoten %d vor!\n'], char( Variablen(j) ));
                unbenutzteVariablen = ...
                    [ unbenutzteVariablen unbenutztesGleichnis ];
                VektorKnotenID(end+1,1) = unbenutzteKnoten(i,1);
```

```
                break
            elseif sum(Vorkommen) > 1
                error('Variable %s kommt in Knoten %d mehrfach vor!', ...
                    char(Variablen(j)), unbenutzteKnoten(i,1) );
            end
        end

        if (i == length(unbenutzteKnoten) && ~isempty(unbenutzteVariablen))
            disp('Variable kommt in mehreren Gleichnissen vor!')
            Variablen = [Variablen unbenutzteVariablen];
            Variablen = unique(Variablen);
            Variablen(Variablen == sym('0')) = [];
            unbenutzteVariablen = sym([]);
            unbenutzteKnoten = setdiff(Knotenliste.ID, VektorKnotenID);
            i = 0;
        elseif ( i == length(unbenutzteKnoten) && ...
                 isempty(unbenutzteVariablen) )
            disp('Keine weiteren Gleichnisse gefunden!')
        end

        i = i + 1;
    end

Gleichnisvariablen = transpose(Variablen);
end
```

C.3.7. ermitteln_SRC()

```
function [ SRC_ID ] = ermitteln_SRC( Variable, Bondliste )
%function [ SRC_ID ] = ermitteln_SRC( Variable, Bondliste )
%
%   Die Funktion ermittelt die Identifikationsnummer des Quellelementes für
%   den der übergebenen Variable zugehörigen Bond.
%
%   Input:  - Variablenbezeichner
%           - Bondliste
%
%   Output: - ID des Quellelementes

%% Vorbereitung
    if nargin ~= 2
        error('Unerwartete Anzahl an Eingabeparametern!');
    end

%% Hauptteil
    % BondID aus Variablenbezeichner extrahieren
    BondID = ermittleln_VarID(Variable);

    % ID der Quelle aus Bondliste mit Hilfe der BondID laden
    SRC_ID = Bondliste.Source(BondID).ID;
end
```

C.3.8. ermitteln_SRC_Typ()

```
function [ SRC_Type ] = ermitteln_SRC_Typ( Variable, Bondliste )
%function [ SRC_Type ] = ermitteln_SRC_Typ( Variable, Bondliste )
%
%   Die Funktion ermittelt den Typ des Quellelementes für den der über-
%   gebenen Variable zugehörigen Bonds.
%
%   Input:  - Variablenbezeichner
%           - Bondliste
%
%   Output: - Typ des Quellelementes als char[]

%% Vorbereitung
    if nargin ~= 2
        error('Unerwartete Anzahl an Eingabeparametern!');
    end

    global Elementeliste

%% Hauptteil

    % ID der Quelle ermitteln
    SRC_ID = ermitteln_SRC( Variable, Bondliste );

    % Typ mit Hilfe der ID aus der Elementeliste laden
    SRC_Type = Elementeliste(SRC_ID).Typ;
end
```

C.3.9. ermitteln_VarSymbol()

```
function [ Symbol ] = ermitteln_VarSymbol( Variable )
%ERMITTELN_VARSYMBOL ( Variable )
%
% Die Funktion ermittelt aus dem übergebenen Variablenbezeichner das
% Zeichen zur Angabe des Variablentyp.
%
% Beispiel 1:   Eingabe: 'e6' --> Ausgabe: 'e'
% Beispiel 2:   Eingabe: 'dp6' --> Ausgabe: 'dp'

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl an Eingabeparameter!');
end

Variable = char(Variable);

%% Ermittlung des Sysmbols
if length(Variable) > 1
    Symbol = Variable(1);

    if strcmp(Symbol,'d')
        if ( strcmp(Variable(2),'p') || strcmp(Variable(2),'q') )
            Symbol(2) = Variable(2);
        end
    end
elseif length(Variable) <= 1
    error('Übergebener Variablenbezeichner fehlerhaft!');
end
end
```

C.3.10. ermitteln_VarTyp()

```
function [ Typ ] = ermitteln_VarTyp( Variable )
%[ Typ ] = ermitteln_VarTyp( Variable )
%
%   Die Funktion ermittelt den Grundtyp der übergebenen Bondvariable und
%   gibt diesen als Zeichenkette zurück.


%% Vorbereitung
    if nargin ~= 1
        error('Unerwartete Anzahl an Eingabeparametern');
    end

%% Hauptteil
    % Symbol der Variable extrahieren
    Symbol = ermitteln_VarSymbol(Variable);

    if length(Symbol) == 1
        if strcmp(Symbol,'e')
            Typ = 'Effort';
            return
        elseif strcmp(Symbol,'f')
            Typ = 'Flow';
            return
        end

    elseif length(Symbol) == 2
        if strcmp(Symbol,'dp')
            Typ = 'Effort';
            return
        elseif strcmp(Symbol,'dq')
            Typ = 'Flow';
            return
        end

    else
        error('Zeichenzahl des Variablensymbols passt nicht!')
    end
end
```

C.3.11. ermitteln_VarID()

```
function [ ID ] = ermittleln_VarID( Variable )
%[ ID ] = ermittleln_VarID( Variable )
%
%   Extrahiert aus der übergebenen symbolische Bondvariable (String) die
%   Identifikationsnummer des zugehörigen Bonds.
%
%   Beispiel:
%       Variable:   'dp5'
%
%       Ergebnis:   5
%
%   Input:  Variable
%
%   Output: ID der Variable ( >= 1 )


%% Vorbereitung
    if nargin ~= 1
        error('Unerwartete Anzahl an Eingabeparametern!');
    end

%% Hauptteil
    if isa(Variable,'char')

        % Wenn ersten beiden Zeichen 'dp' oder 'dq' sind, handelt es sich
        % um eine Zustandsableitung, wodurch die ID an der dritten Stelle
        % des Bezeichners beginnt.
        if (strcmp(Variable(1:2),'dp') || strcmp(Variable(1:2),'dq'))
            ID = eval(Variable(3:end));

        % Wenn das erste Zeichen 'e' oder 'f' ist, beginnt die ID an der
        % zweiten Stelle des Bezeichners.
        elseif ( strcmp(Variable(1),'e') ...
                || strcmp(Variable(1),'f') ...
                || strcmp(Variable(1),'q') ...
                || strcmp(Variable(1),'p') )
            ID = eval(Variable(2:end));
        else
            error('Eingegebene Variable ist keine Bondvariable!');
        end
```

```
    elseif (isa(Variable,'char') && length(Variable) <= 1)
        error('Eingegebene Variable ist zu kurz!');

    else
        error('Falscher Datentyp!');
    end
end
```

D. Nachbereitung

D.1. Gleichungen

D.1.1. ermitteln_Gleichungssatz()

```
function [ Gleichungssatz ] = ermitteln_Gleichungssatz( Liste, Ergebnisvektor )
% [ Gleichungssatz ] = ermitteln_Gleichungssatz( Liste, Ergebnisvektor )
%
%   Ermittelt aus der übergebenen Liste die den Variablen im übergebenen
%   Vektor zugehörigen Gleichungen. Die Ausgaberereihenfolge der
%   Gleichungen entspricht der Reihenfolge der Ergbisvariablen im Vektor.

%% Vorbereitung
if nargin ~= 2
    error('Anzahl!')
end
Ergebnisvektor = sym(Ergebnisvektor);
Gleichungssatz = sym([]);

%% Einzelne Elemente des Vektors durchgehen und Gleichungen ggf. ermitteln
for i = 1 : length(Ergebnisvektor)
    Position = Liste(:,2) == Ergebnisvektor(i,1);

    % Wenn Gleichung auf Liste gefunden, wird diese gespeichert
    if sum(Position) == 1
        aktuelleGleichung = Liste( Position ,3);
    % ... andernfalls wird die Gleichung neu ermittelt
    elseif sum(Position) ~= 1
        [aktuelleGleichung aktGlNummer aktBemerkung] = ...
            ermitteln_Gleichung( Ergebnisvektor(i,1) );

        % Aufräumen nach Ermittlung
        clear global gesperrteGleichungen
        clear global gesperrteVariablen
```

```
        if isempty(aktuelleGleichung)
            error('Ergebnis für "%s" nicht wie erwartet gefunden!', ...
                char(Ergebnisvektor(i,1)))
        end
    end
    Gleichungssatz(i,1) = aktuelleGleichung;
end
end
```

D.1.2. substituieren Werte()

```
function [  ] = subsituieren_Werte(  )
%SUBSITUIEREN_WERTE()
%
%   Die Funktion ersetzt in der Elemente-, der Grund- und der Ergebnisliste
%   alle Konstanten durch die in den GUIs des Simulinkmodells angegebenen
%   Werten der individuellen Blockparameter

%% Vorbereitung: Laden der globalen Listen
global Elementeliste
global Grundliste
global Ergebnisliste

%% Finden und substituieren
% Jedes Element einzeln durchgehen und den zugehörigen Parameter in den
% Listen finden und durch den entsprechenden Wert ersetzen.
for i = 1: length(Elementeliste)
    Typ = Elementeliste(i).Typ;
    Konstante_ID = int2str( Elementeliste(i).ID );

    % Symbol der Konstante gemäß Typ des Blocks festlegen
    switch Typ
        case 'Capacitor'
            Konstante_Typ = 'C';
        case 'Inductor'
            Konstante_Typ = 'L';
        case 'Resistor'
            Konstante_Typ = 'R';
        case 'Gyrator'
            Konstante_Typ = 'r';
        case 'Transformer'
            Konstante_Typ = 'm';
        case 'Paralleler Knoten'
            disp('"Knoten" besitzen keine substituierbaren Werte.');
            continue
        case 'Serieller Knoten'
                disp('"Knoten" besitzen keine substituierbaren Werte.');
            continue
        case 'Eingang'
            disp('"Eingängen" besitzen keine substituierbaren Werte.');
            continue
        otherwise
```

```
                error('Unbekanntes Element!');
    end

    % Bezeichner der Konstante konstruieren
    Konstante = [ Konstante_Typ Konstante_ID ];

    % Konstante auf Grundliste substituieren
    for j = 1:length(Grundliste(:,1))
        Grundliste{j,3} = subs( Grundliste{j,3}, Konstante, ...
                                Elementeliste(i).Wert, 0     );
    end

        % Konstante auf Ergebnisliste substituieren
    for j = 1:length(Ergebnisliste(:,1))
        Ergebnisliste{j,3} = subs( Ergebnisliste{j,3}, Konstante, ...
                                   Elementeliste(i).Wert, 0);
    end

end

% fertige Listen anzeigen und in base-Workspace ablegen
anzeigen_Liste(Grundliste, 'Grundliste')
anzeigen_Liste(Ergebnisliste, 'Ergebnisliste')
end
```

D.2. Zustandsmatrizen

D.2.1. ermitteln_Koeffizient()

```
function [ Koeffizient ] = ermitteln_Koeffizent( Gleichung, Variable )
%[ Koeffizient] = ERMITTELN_KOEFFIZENT ( Gleichung, Variable )
%
%   Die Funktion ermitteln für die übergebenen Variable den zugehörigen
%   Koeffizienten in der übergeben Gleichung.
%
%   Wird die Variable in der Gleichung nicht gefunden, wird als Koeffizient
%   der Wert 0 zurückgegeben!

%% Vorbereitung
if nargin ~= 2
    error('Unerwartete Anzahl an Eingabeparametern');
end

%% Alle Terme und Koeffizienten der Gleichung extrahieren
[ Konstanten Terme ] = coeffs( expand(Gleichung), Variable );

%% Koeffizient ermitteln
Position = Variable == Terme;

Anzahl = sum(Position);

    if Anzahl == 1
        Koeffizient = simplify(Konstanten(Position));
    elseif Anzahl == 0
        Koeffizient = 0;
    else
        error(  'Variable %s wurde in Gleichung %d-mal gefunden!',      ...
                char(Variable), sum(Position));
    end
end
```

D.2.2. ermitteln_Matrix()

```
function [ Matrix ] = ermitteln_Matrix( Gleichungen, Vektor )
% ERMITTELN_MATRIX ( Gleichungen, Vektor )
%
%   Die Funktion ermittelt aus dem übergebenen Gleichungsatz die
%   Koeffizientenmatrix die Variablen des übergebenen Vektors.

%% Vorbereitung
if nargin ~= 2
    error('Falsche Anzahl an Eingabeparametern!');

elseif isempty(Gleichungen)
    disp('Übergebene Gleichungen sind leer... Matrix ist 0!')
    Matrix = 0;
    return

elseif isempty(Vektor)
    disp('Übergebene Variablenvektor ist leer... Matrix ist 0!')
    Matrix = 0;
    return

end

Matrix = sym([]);
Gleichungen = sym(Gleichungen);
Vektor = sym(Vektor);

%% Ermittlung der Koeffizienten und Einordnung in Matrix
for i = 1:length(Gleichungen(:,1))
    for j = 1:length(Vektor(:,1))
        Matrix(i,j) = ermitteln_Koeffizent(Gleichungen(i,1), Vektor(j,1));
    end
end
end
```

E. Sperrmechanismus

E.1. Variablen

E.1.1. freigeben_Variable()

```
function [ ] = freigeben_Variable( Variable )
%FREIGEBEN_VARIABLE Summary of this function goes here
%
%   Löscht die übergebene Variable von der Sperrliste "gesperrteVariablen".

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl an Eingabeparametern!');
end
global gesperrteVariablen

%% Freigabe: Wenn Sperrliste für Variablen existiert: Variable freigeben
if ~isempty(gesperrteVariablen)
        Treffer = gesperrteVariablen == Variable;

    if sum(Treffer) == 1
        gesperrteVariablen(Treffer) = [];
        fprintf('Variable "%s" wurde entsperrt!\n', char(Variable));
    elseif sum(Treffer) == 0
        fprintf('Variable "%s" war nicht gesperrt!\n', char(Variable));
    elseif sum(Treffer) > 1
        error('Variable "%s" mehrfach auf Sperrliste!', char(Variable));
    end

% Wenn diese Sperrliste nicht existiert: mit Warnung zurückkehren
elseif isempty(gesperrteVariablen)
    warning('Sperrliste "gesperrteVariablen" existiert nicht!');
end
end
```

E.1.2. ist_gesperrt_Variable()

```
function [ Status ] = ist_gesperrt_Variable( Variable )
%[ Status ] = IST_GESPERRT_VARIABLE ( Variable )
%
%   Durchsucht die Sperrliste "gesperrteVariablen" nach der übergebenen
%   Variable. Steht diese auf der Liste und ist somit gesperrt,
%   wird TRUE zurückgegeben anderfalls FALSE.

%% Vorbereitung
if nargin ~= 1
    error('Falsche Anzahl an Eingabeparametern!');
end
Variable = sym(Variable);

global gesperrteVariablen

%% Abfrage
if isempty(gesperrteVariablen)
    Status = false;
    disp('Sperrliste "gesperrteVariablen" existiert nicht!');
else
    Treffer = gesperrteVariablen == Variable;
    AnzahlTreffer = sum(Treffer);

    if AnzahlTreffer == 1
        Status = true;
    elseif AnzahlTreffer == 0
        Status = false;
    else
        error('Liste "gesperrteVariablen" fehlerhaft!')
    end
end
end
```

E.1.3. sperre_Variable()

```
function [ ] = sperre_Variable( Variable )
%SPERRE_VARIABLE ( Variable )
%
%   Setzt die übergebene Variable auf die Sperrliste "gesperrteVariablen".
%   Löst einen Fehler aus, wenn die Variable sich bereits eingetragen ist.
%   Sollte "gesperrteVariablen" nocht nicht existieren, wird es erstellt.

%% Vorbereitung
if nargin ~= 1
    error('Falsche Anzahl an Eingabeparameter!')
end

global gesperrteVariablen

Variable = sym(Variable);

%% Sperren
if isempty(gesperrteVariablen)
    gesperrteVariablen = sym([]);
    gesperrteVariablen(1,1) = Variable;
else

    Vorkommen = gesperrteVariablen == Variable;

    if sum(Vorkommen) == 0
        gesperrteVariablen(end+1,1) = Variable
    else
        warning('Variable ist bereits gesperrt... keine Änderung!');
    end
end
end
```

E.2. Gleichungen

E.2.1. freigeben_Gleichung()

```
function [ ] = freigeben_Gleichung( Gleichungsnummer )
%FREIGEBEN_GLEICHUNG ( Gleichungsnummer )
%
%   Entfernt die übergebenen Gleichungsnummer von "gesperrteGleichungen".

%% Vorbereitung
if nargin ~= 1
    error('Unerwartete Anzahl an Eingabeparametern!');
end

global gesperrteGleichungen

%% Freigabe
% Wenn Sperrliste für Variablen existiert: Variable freigeben
if ~isempty(gesperrteGleichungen)

    if ~isempty(Gleichungsnummer)
        Treffer = gesperrteGleichungen == Gleichungsnummer;

        if sum(Treffer) == 1
            gesperrteGleichungen(Treffer) = [];
            fprintf('Gleichungsnummer "%d" wurde entsperrt!\n', ...
                Gleichungsnummer);
        elseif sum(Treffer) == 0
            fprintf('Gleichungsnummer "%d" war nicht gesperrt!\n', ...
                Gleichungsnummer);
        elseif sum(Treffer) > 1
            error('Gleichungsnummer mehrfach auf Sperrliste!');
        end
    else
        error('Gleichungsnummer ist leer []!\n');
    end

% Wenn diese Sperrliste nicht existiert: mit Warnung zurückkehren
elseif isempty(gesperrteGleichungen)
    warning('Sperrliste "gesperrteVariablen" existiert nicht!');
end
end
```

E.2.2. ist_gesperrt_Gleichung()

```
function [ Status ] = ist_gesperrt_Gleichung( Gleichungsnummer )
%[ Status ] = IST_GESPERRT_GLEICHUNG ( Gleichungsnummer )
%
%   Durchsucht die Sperrliste "gesperrteGleichungen" nach der übergebenen
%   Gleichungsnummer. Steht diese auf der Liste und ist somit gesperrt,
%   wird TRUE zurückgegeben anderfalls FALSE.

%% Vorbereitung
if nargin ~= 1
    error('Falsche Anzahl an Eingabeparametern!');
end

global gesperrteGleichungen
global Arbeitsliste

if isempty(gesperrteGleichungen)
    Status = false;
    disp('"gesperrteGleichungen" existiert nicht!');
end

%% Abfrage "gesperrteGleichungen"
Treffer = gesperrteGleichungen == Gleichungsnummer;
AnzahlTreffer = sum(Treffer);

if AnzahlTreffer == 1
    Status = true;
    return
elseif ( AnzahlTreffer ~= 0 && AnzahlTreffer ~= 1 )
    error('Liste "gesperrteGleichungen" fehlerhaft!')
end

%% Abfrage "Arbeitsliste"
if isempty(Arbeitsliste)
    Status = false;
    disp('"Arbeitsliste" existiert nicht!');
    return
end

clear Treffer
clear AnzahlTreffer
```

```
Treffer = [ Arbeitsliste{:,1} ] == Gleichungsnummer

AnzahlTreffer = sum(Treffer);

if AnzahlTreffer == 1
    Status = true;
elseif AnzahlTreffer == 0
    Status = false;
    fprintf('Gleichung %d nicht auf Arbeitsliste!\n', Gleichungsnummer);
else
    error('Liste "Arbeitsliste" fehlerhaft!')
end
end
```

E.2.3. sperre_Gleichung()

```
function [ ] = sperre_Gleichung( Gleichungsnummer )
%SPERRE_Gleichung ( Gleichungsnummer )
%
%   Setzt die übergebene Variable auf die Sperrliste "gesperrteVariablen".
%   Löst einen Fehler aus, wenn die Variable sich bereits eingetragen ist.
%   Sollte "gesperrteVariablen" nocht nicht existieren, wird es erstellt.

%% Vorbereitung
if nargin == 1
    Gleichungsnummer = uint8(Gleichungsnummer);
else
    error('Falsche Anzahl an Eingabeparameter!')
end

global gesperrteGleichungen

%% Sperren
if isempty(gesperrteGleichungen)
    gesperrteGleichungen = [];
    gesperrteGleichungen(1,1) = Gleichungsnummer;
else
    Vorkommen = gesperrteGleichungen == Gleichungsnummer;

    if sum(Vorkommen) == 0
        gesperrteGleichungen(end+1,1) = Gleichungsnummer;
    else
        warning('Gleichungsnummer %d ist bereits gesperrt!', ...
            Gleichungsnummer);
    end
end
end
```

Teil III.

S-Functions der Bondelemente

F. Verbraucher

F.1. Resistor R

```
% S-Function für Bondelement "Resistor"
%
%   Basiert auf einem Template von Matlab.
%
%   Element wird in setup()-Funktion registriert, wobei vorher sämtliche
%   Eigenschaft im UserData-Feld des RTO geschrieben werden.
%
%   Die zugehörigen Bonds des Elementes werden in DoPostPropSetup() durch
%   registrieren_Bond() erstellt.
%
%   Als letzer Schritt wird in Start()-Callback die eigentliche Toolbox
%   durch die Funktion analysieren() gestartet.
%
%   Funktionen zur späreten Verwendung und ggf. Realisierung der parallelen
%   Simulation des ss-Modell neben Simulinkmodell:
%
%   - InitializeConditions()
%   - Start()
%   - Update()
%   - Output()

function sf_Verbraucher_R(block)

    setup(block);


function setup(block)

    disp('....SETUP --> Resistor-Block')

    % Register number of ports
```

```
block.NumInputPorts  = 1;
block.NumOutputPorts = 0;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).SamplingMode  = 'Sample';
block.InputPort(1).Dimensions        = 1;


% Register parameters
block.NumDialogPrms     = 4;
% Parameter:  - Wert: double
%             - Kausalität:  1 = 'e2f' = f(e) oder 2 = 'f2e' = e(f)
%             oder 3 = auto
%             - Invertierung: boolean
%             - Ausgang: 0 = kein Ausgang; 1: e & f in y registrieren

block.SampleTimes = [-1 0];

block.SimStateCompliance = 'DefaultSimState';

block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
block.RegBlockMethod('Start', @Start);
block.RegBlockMethod('Outputs', @Outputs);
block.RegBlockMethod('Update', @Update);

% globale Elementezahl erstellen oder erhöhen
    global Elementezahl

    if isempty(Elementezahl)
        Elementezahl = 1
    elseif ~isempty(Elementezahl)
        Elementezahl = Elementezahl + 1
    end

% Kausalität gemäß Userangabe setzen
```

```
    switch block.DialogPrm(2).Data
            case 1
                Causality = 'f(e)';
            case 2
                Causality = 'e(f)';
            case 3
                Causality = 'auto';
            otherwise
                error('Falsche Angabe bei Eingabeparameter KAUSALITÄT!')
    end

    % Eigenschaften in .UserData zusammenfassen
    UserData.Toolbox = 'Bondgraph';
    UserData.ID = Elementezahl;
    UserData.Kategorie = 'Verbraucher';
    UserData.Typ = 'Resistor';
    UserData.Wert = block.DialogPrm(1).Data;
    UserData.Name = get(gcbh,'Name');
    UserData.Invertierung = boolean(block.DialogPrm(3).Data);
    UserData.Causality = Causality;
    UserData.Ausgang = boolean(block.DialogPrm(4).Data);
    UserData.Handle = gcbh;

    % Userdata in RTO schreiben
    set(gcbh,'UserData',UserData)

    % Element registrieren
    registrieren_Element(UserData,'intern');
    %endfunction

function DoPostPropSetup(block)
    disp('... DoPostPropSetup --> Bond(s) registrieren')
    registrieren_Bond()

    block.NumDworks = 1;
    block.Dwork(1).Name            = 'x1';
    block.Dwork(1).Dimensions      = 1;
    block.Dwork(1).DatatypeID      = 0;      % double
    block.Dwork(1).Complexity      = 'Real'; % real
    block.Dwork(1).UsedAsDiscState = true;

    %% Register all tunable parameters as runtime parameters.
```

```
    block.AutoRegRuntimePrms;
    %endfunction

function InitializeConditions(block)
    disp('... InitializeConditions --> hier ss-Modell einbauen?')
    %endfunction

function Start(block)
    analysieren()
    block.Dwork(1).Data = 0;
    disp('.... START ---> Analyse start...')
    %endfunction

function Outputs(block)
    disp('... Outputs --> ggf. in späteren Version benutzen?')
    %endfunction

function Update(block)
    block.Dwork(1).Data = block.InputPort(1).Data;
    disp('... Update --> ggf. in späteren Version benutzen?')
    %endfunction
```

F.2. Capacitor C

```
% S-Function für Bondelement "Transformer"
%
%   Basiert auf einem Template von Matlab.
%
%   Element wird in setup()-Funktion registriert, wobei vorher sämtliche
%   Eigenschaft im UserData-Feld des RTO geschrieben werden.
%
%   Die zugehörigen Bonds des Elementes werden in DoPostPropSetup() durch
%   registrieren_Bond() erstellt.
%
%   Als letzer Schritt wird in Start()-Callback die eigentliche Toolbox
%   durch die Funktion analysieren() gestartet.
%
%   Funktionen zur späreten Verwendung und ggf. Realisierung der parallelen
%   Simulation des ss-Modell neben Simulinkmodell:
%
%   - InitializeConditions()
%   - Start()
%   - Update()
%   - Output()

function sf_Verbraucher_C(block)

    setup(block);

function setup(block)

    disp('....SETUP --> Capacitor-Block')

    % Register number of ports
    block.NumInputPorts  = 1;
    block.NumOutputPorts = 0;

    % Setup port properties to be inherited or dynamic
    block.SetPreCompInpPortInfoToDynamic;
    block.SetPreCompOutPortInfoToDynamic;

    % Override input port properties
    block.InputPort(1).DatatypeID  = 0;  % double
    block.InputPort(1).Complexity  = 'Real';
```

```
block.InputPort(1).SamplingMode  = 'Sample';
block.InputPort(1).Dimensions        = 1;


% Register parameters
block.NumDialogPrms     = 4;
% Parameter:    - 1. Wert: double
%               - 2. Kausalität:  0 = 'e2f' = f(e) oder 1 = 'f2e' = e(f)
%               - 3. Invertierung: boolean
%               - 4. Ausgang:     0 = kein Ausgang
%                                 1 = e & f in y registrieren

block.SampleTimes = [-1 0];

block.SimStateCompliance = 'DefaultSimState';

block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
block.RegBlockMethod('Start', @Start);
block.RegBlockMethod('Outputs', @Outputs);
block.RegBlockMethod('Update', @Update);

% globale Elementezahl erstellen oder erhöhen
    global Elementezahl

    if isempty(Elementezahl)
        Elementezahl = 1
    elseif ~isempty(Elementezahl)
        Elementezahl = Elementezahl + 1
    end

% Eigenschaften in .UserData zusammenfassen
switch block.DialogPrm(2).Data
    case 1
        Causality = 'f(e)';
    case 2
        Causality = 'e(f)';
    case 3
        Causality = 'auto';
    otherwise
        error('Falsche Angabe bei Eingabeparameter KAUSALITÄT!')
end
```

```
    UserData.Toolbox = 'Bondgraph';
    UserData.ID = Elementezahl;
    UserData.Kategorie = 'Verbraucher';
    UserData.Typ = 'Capacitor';
    UserData.Wert = block.DialogPrm(1).Data;
    UserData.Name = get(gcbh,'Name');
    UserData.Invertierung = boolean(block.DialogPrm(3).Data);
    UserData.Causality = Causality;
    UserData.Ausgang = boolean(block.DialogPrm(4).Data);
    UserData.Handle = gcbh;

    % Userdata in RTO schreiben
    set(gcbh,'UserData',UserData)

    % Element registrieren
    registrieren_Element(UserData,'intern');
    %endfunction

function DoPostPropSetup(block)
    disp('... DoPostPropSetup --> Bond(s) registrieren')
    registrieren_Bond()

    block.NumDworks = 1;

    block.Dwork(1).Name            = 'x1';
    block.Dwork(1).Dimensions      = 1;
    block.Dwork(1).DatatypeID      = 0;      % double
    block.Dwork(1).Complexity      = 'Real'; % real
    block.Dwork(1).UsedAsDiscState = true;

    %% Register all tunable parameters as runtime parameters.
    block.AutoRegRuntimePrms;
    %endfunction

function InitializeConditions(block)
    disp('... InitializeConditions --> ggf. in späteren Version benutzen?')
    %endfunction

function Start(block)
    disp('.... START ---> Analyse starten...')
    analysieren();
```

```
    block.Dwork(1).Data = 0;
    disp('.... START ---> hier Daten/Gleichungen zurücknehmen')
    %endfunction

function Outputs(block)
    disp('... Outputs --> ggf. in späteren Version benutzen?')
    %endfunction

function Update(block)
    block.Dwork(1).Data = block.InputPort(1).Data;
    disp('... Update --> ggf. in späteren Version benutzen?')
    %endfunction
```

F.3. Inertia I

```
% S-Function für Bondelement "Inertia"
%
%   Basiert auf einem Template von Matlab.
%
%   Element wird in setup()-Funktion registriert, wobei vorher sämtliche
%   Eigenschaft im UserData-Feld des RTO geschrieben werden.
%
%   Die zugehörigen Bonds des Elementes werden in DoPostPropSetup() durch
%   registrieren_Bond() erstellt.
%
%   Als letzer Schritt wird in Start()-Callback die eigentliche Toolbox
%   durch die Funktion analysieren() gestartet.
%
%   Funktionen zur späreten Verwendung und ggf. Realisierung der parallelen
%   Simulation des ss-Modell neben Simulinkmodell:
%
%   - InitializeConditions()
%   - Start()
%   - Update()
%   - Output()

function sf_Verbraucher_I(block)

    setup(block);

function setup(block)

    disp('....SETUP --> Inertia-Block')

    % Register number of ports
    block.NumInputPorts  = 1;
    block.NumOutputPorts = 0;

    % Setup port properties to be inherited or dynamic
    block.SetPreCompInpPortInfoToDynamic;
    block.SetPreCompOutPortInfoToDynamic;

    % Override input port properties
    block.InputPort(1).DatatypeID  = 0;  % double
    block.InputPort(1).Complexity  = 'Real';
```

```
block.InputPort(1).SamplingMode  = 'Sample';
block.InputPort(1).Dimensions        = 1;

% Register parameters
block.NumDialogPrms     = 4;
% Parameter:     - Wert: double
%                - Kausalität:      1 => 'e2f' = f(e) = f(p)
%                                   2 => 'f2e' = e(f)
%                                   3 => 'auto'
%                - Invertierung: boolean
%                - Ausgang: 0 = kein Ausgang; 1: e & f in y registrieren

block.SampleTimes = [-1 0];

block.SimStateCompliance = 'DefaultSimState';

block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
block.RegBlockMethod('Start', @Start);
block.RegBlockMethod('Outputs', @Outputs);
block.RegBlockMethod('Update', @Update);

% globale Elementezahl erstellen oder erhöhen
    global Elementezahl

    if isempty(Elementezahl)
        Elementezahl = 1
    elseif ~isempty(Elementezahl)
        Elementezahl = Elementezahl + 1
    end

% Kausalität gemäß Userangabe setzen
    switch block.DialogPrm(2).Data
        case 1
            Causality = 'f(e)';
        case 2
            Causality = 'e(f)';
        case 3
            Causality = 'auto';
        otherwise
            error('Falsche Angabe bei Eingabeparameter KAUSALITÄT!')
    end
```

```
    % Eigenschaften in .UserData zusammenfassen
    UserData.Toolbox = 'Bondgraph';
    UserData.ID = Elementezahl;
    UserData.Kategorie = 'Verbraucher';
    UserData.Typ = 'Inductor';
    UserData.Wert = block.DialogPrm(1).Data;
    UserData.Name = get(gcbh,'Name');
    UserData.Invertierung = boolean(block.DialogPrm(3).Data);
    UserData.Causality = Causality;
    UserData.Ausgang = boolean(block.DialogPrm(4).Data);
    UserData.Handle = gcbh;

    % Userdata in RTO schreiben
    set(gcbh,'UserData',UserData)

    % Element registrieren
    registrieren_Element(UserData,'intern');
    %endfunction

function DoPostPropSetup(block)
    disp('... DoPostPropSetup --> Bond(s) registrieren')
    registrieren_Bond()

    block.NumDworks = 1;

    block.Dwork(1).Name            = 'x1';
    block.Dwork(1).Dimensions      = 1;
    block.Dwork(1).DatatypeID      = 0;      % double
    block.Dwork(1).Complexity      = 'Real'; % real
    block.Dwork(1).UsedAsDiscState = true;

    %% Register all tunable parameters as runtime parameters.
    block.AutoRegRuntimePrms;

    %endfunction

function InitializeConditions(block)
    disp('... InitializeConditions --> ggf. in späteren Version benutzen?')
    %endfunction

function Start(block)
```

```
    analysieren();
    block.Dwork(1).Data = 0;
    disp('.... START ---> Analyse starten...')
    %endfunction

function Outputs(block)
    disp('... Outputs --> ggf. in späteren Version benutzen?')
    %endfunction

function Update(block)
    block.Dwork(1).Data = block.InputPort(1).Data;
    disp('... Update --> ggf. in späteren Version benutzen?')
    %endfunction
```

G. Knoten

G.1. Knoten mit einem Ein- und zwei Ausgängen

```
% S-Function für Bondelement "Knoten"
%
%   Basiert auf einem Template von Matlab.
%
%   Element wird in setup()-Funktion registriert, wobei vorher sämtliche
%   Eigenschaft im UserData-Feld des RTO geschrieben werden. Des Weiteren
%   wird während des setup()-Callbacks der Knoten auf der Knotenliste ein-
%   getragen
%
%   Die zugehörigen Bonds des Elementes werden in DoPostPropSetup() durch
%   registrieren_Bond() erstellt.
%
%   Als letzer Schritt wird in Start()-Callback die eigentliche Toolbox
%   durch die Funktion analysieren() gestartet.
%
%   Funktionen zur späreten Verwendung und ggf. Realisierung der parallelen
%   Simulation des ss-Modell neben Simulinkmodell:
%
%   - InitializeConditions()
%   - Start()
%   - Update()
%   - Output()

function sf_Knoten_In1_Out2(block)

    setup(block);

function setup(block)

    disp('....SETUP --> Transformer-Block')
```

```
% Register number of ports
block.NumInputPorts  = 1;
block.NumOutputPorts = 2;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).SamplingMode  = 'Sample';
block.InputPort(1).Dimensions        = 1;

% Override output port properties
% Bondausgang 1
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode  = 'Sample';
block.OutputPort(1).Dimensions        = 1;

% Bondausgang 2
block.OutputPort(2).DatatypeID  = 0; % double
block.OutputPort(2).Complexity  = 'Real';
block.OutputPort(2).SamplingMode  = 'Sample';
block.OutputPort(2).Dimensions        = 1;


% Register parameters
block.NumDialogPrms     = 4;
% Parameter:  1. Knotentyp: 1 = serieller Knoten
%                           0 = paralleler Knoten
%             2. Kausalität:   0 = 'e2f' = f(e),
%                              1 = 'f2e' = e(f),
%                              2 = 'auto'
%             3. Invertierung: boolean
%DEAKTIVIERT: 4. Ausgang: 0 = kein Ausgang; 1: e & f in y registrieren

block.SampleTimes = [-1 0];

block.SetAccelRunOnTLC(false);
block.SimStateCompliance = 'DefaultSimState';
```

```
block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
block.RegBlockMethod('Start', @Start);
block.RegBlockMethod('Update', @Update);
block.RegBlockMethod('Outputs', @Outputs);

% globale Elementezahl erstellen oder erhöhen
    global Elementezahl

    if isempty(Elementezahl)
        Elementezahl = 1;
    elseif ~isempty(Elementezahl)
        Elementezahl = Elementezahl + 1;
    end

% Knotentyp gemäß Userangabe setzen
block.DialogPrm(1).Data
switch block.DialogPrm(1).Data
    case 1
        Typ = 'Paralleler Knoten';
        Wert = 0;
    case 2
        Typ = 'Serieller Knoten';
        Wert = 1;
    otherwise
        error('Falsche Angabe bei Eingabeparameter KNOTENTYP!')
end


% Kausalität gemäß Userangabe setzen
switch block.DialogPrm(2).Data
case 1
    Causality = 'f(e)';
case 2
    Causality = 'e(f)';
case 3
    Causality = 'auto';
otherwise
    error('Falsche Angabe bei Eingabeparameter KAUSALITÄT!')
end
```

```
    % Eigenschaften in .UserData zusammenfassen
    UserData.Toolbox = 'Bondgraph';
    UserData.ID = Elementezahl;
    UserData.Kategorie = 'Knoten';
    UserData.Typ = Typ;
    UserData.Wert = Wert;
    UserData.Name = get(gcbh,'Name');
    UserData.Invertierung = boolean(block.DialogPrm(3).Data);
    UserData.Causality = Causality;
    %UserData.Ausgang = boolean(block.DialogPrm(4).Data);
    UserData.Ausgang = 0;
    UserData.Handle = gcbh;

    % Userdata in RTO schreiben
    set(gcbh,'UserData',UserData)

    % Knoten auf Knotenliste registrieren
        global Knotenliste

        if isempty(Knotenliste)
            Knotenliste.ID(1,1) = Elementezahl;
            Knotenliste.Typ(1,1) = {Typ};
            Knotenliste.Gleichnisse = {};
            Knotenliste.Inputs = {};
            Knotenliste.Outputs = {};
        elseif ~isempty(Elementezahl)
            Knotenliste.ID(end + 1,1) = Elementezahl;
            Knotenliste.Typ(end + 1,1) = {Typ};
        end

    % Element registrieren
    registrieren_Element(UserData,'intern');
    %endfunction

function DoPostPropSetup(block)
    disp('... DoPostPropSetup --> Bond(s) registrieren')
    registrieren_Bond()

    block.NumDworks = 1;

    block.Dwork(1).Name             = 'x1';
```

```
    block.Dwork(1).Dimensions      = 1;
    block.Dwork(1).DatatypeID      = 0;      % double
    block.Dwork(1).Complexity      = 'Real'; % real
    block.Dwork(1).UsedAsDiscState = true;

    %% Register all tunable parameters as runtime parameters.
    block.AutoRegRuntimePrms;
    %endfunction

function InitializeConditions(block)
    disp('... InitializeConditions --> ggf. in späteren Version benutzen?')
    %endfunction

function Start(block)
        disp('.... START ---> Analyse starten...')
    analysieren()

    block.Dwork(1).Data = 0;
    disp('.... START ---> hier Daten/Gleichungen zurücknehmen')
    %endfunction

function Outputs(block)
    disp('... Outputs --> ggf. in späteren Version benutzen?')
    block.OutputPort(1).Data = -1;
    block.OutputPort(2).Data = -1;
    %endfunction

function Update(block)
    disp('... Update --> ggf. in späteren Version benutzen?')
    %endfunction
```

H. Umformer

H.1. Transformer TF

```
% S-Function für Bondelement "Transformer"
%
%   Basiert auf einem Template von Matlab.
%
%   Element wird in setup()-Funktion registriert, wobei vorher sämtliche
%   Eigenschaft im UserData-Feld des RTO geschrieben werden.
%
%   Die zugehörigen Bonds des Elementes werden in DoPostPropSetup() durch
%   registrieren_Bond() erstellt.
%
%   Als letzer Schritt wird in Start()-Callback die eigentliche Toolbox
%   durch die Funktion analysieren() gestartet.
%
%   Funktionen zur späreten Verwendung und ggf. Realisierung der parallelen
%   Simulation des ss-Modell neben Simulinkmodell:
%
%   - InitializeConditions()
%   - Start()
%   - Update()
%   - Output()

function sf_Transformer(block)

    setup(block);

function setup(block)

    disp('....SETUP --> Transformer-Block')

    % Register number of ports
    block.NumInputPorts  = 1;
```

```
block.NumOutputPorts = 1;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).SamplingMode  = 'Sample';
block.InputPort(1).Dimensions        = 1;

% Override output port properties
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode  = 'Sample';
block.OutputPort(1).Dimensions        = 1;

% Register parameters
block.NumDialogPrms     = 4;
% Parameter:     -1. Wert: double
%                -2. Kausalität:  0 = 'e2f' = f(e) oder 1 = 'f2e' = e(f)
%                -3. Invertierung: boolean
%                -4. Ausgang: 0 = kein Ausgang;
%                             1 = e & f in y registrieren

block.SampleTimes = [-1 0];

block.SetAccelRunOnTLC(false);
block.SimStateCompliance = 'DefaultSimState';

block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
block.RegBlockMethod('Start', @Start);
block.RegBlockMethod('Update', @Update);
block.RegBlockMethod('Outputs', @Outputs);

% globale Elementezahl erstellen oder erhöhen
    global Elementezahl

    if isempty(Elementezahl)
        Elementezahl = 1
```

```
        elseif ~isempty(Elementezahl)
            Elementezahl = Elementezahl + 1
        end

    % Kausalität gemäß Userangabe setzen
    switch block.DialogPrm(2).Data
        case 0
            Causality = 'f(e)';
        case 1
            Causality = 'e(f)';
        case 2
            Causality = 'auto';
        otherwise
            error('Falsche Angabe bei Eingabeparameter KAUSALITÄT!')
    end

    % Eigenschaften in .UserData zusammenfassen
    UserData.Toolbox = 'Bondgraph';
    UserData.ID = Elementezahl;
    UserData.Kategorie = 'Leitung';
    UserData.Typ = 'Transformer';
    UserData.Wert = block.DialogPrm(1).Data;
    UserData.Name = get(gcbh,'Name');
    UserData.Invertierung = boolean(block.DialogPrm(3).Data);
    UserData.Causality = Causality;
    UserData.Ausgang = boolean(block.DialogPrm(4).Data);
    UserData.Handle = gcbh;

    % Userdata in RTO schreiben
    set(gcbh,'UserData',UserData)

    % Element registrieren
    registrieren_Element(UserData,'intern');
    %endfunction

function DoPostPropSetup(block)
    disp('... DoPostPropSetup --> Bond(s) registrieren')
    registrieren_Bond()

    block.NumDworks = 1;

    block.Dwork(1).Name             = 'x1';
```

```
    block.Dwork(1).Dimensions      = 1;
    block.Dwork(1).DatatypeID      = 0;      % double
    block.Dwork(1).Complexity      = 'Real'; % real
    block.Dwork(1).UsedAsDiscState = true;

    %% Register all tunable parameters as runtime parameters.
    block.AutoRegRuntimePrms;
    %endfunction

function InitializeConditions(block)
    disp('... InitializeConditions --> ggf. in späteren Version benutzen?')
    %endfunction

function Start(block)
        disp('.... START ---> Analyse starten...')
    analysieren()
    block.Dwork(1).Data = 0;
    %endfunction

function Outputs(block)
    disp('... Outputs --> ggf. in späteren Version benutzen?')
    block.OutputPort(1).Data = -1;
    %endfunction

function Update(block)
    disp('... Update --> ggf. in späteren Version benutzen?')
    %endfunction
```

H.2. Gyrator GY

```
% S-Function für Bondelement "Gyrator"
%
%   Basiert auf einem Template von Matlab.
%
%   Element wird in setup()-Funktion registriert, wobei vorher sämtliche
%   Eigenschaft im UserData-Feld des RTO geschrieben werden.
%
%   Die zugehörigen Bonds des Elementes werden in DoPostPropSetup() durch
%   registrieren_Bond() erstellt.
%
%   Als letzer Schritt wird in Start()-Callback die eigentliche Toolbox
%   durch die Funktion analysieren() gestartet.
%
%   Funktionen zur späreten Verwendung und ggf. Realisierung der parallelen
%   Simulation des ss-Modell neben Simulinkmodell:
%
%   - InitializeConditions()
%   - Start()
%   - Update()
%   - Output()

function sf_Gyrator(block)

    setup(block);

function setup(block)

    disp('....SETUP --> Gyrator-Block')

    % Register number of ports
    block.NumInputPorts  = 1;
    block.NumOutputPorts = 1;

    % Setup port properties to be inherited or dynamic
    block.SetPreCompInpPortInfoToDynamic;
    block.SetPreCompOutPortInfoToDynamic;

    % Override input port properties
    block.InputPort(1).DatatypeID  = 0;  % double
    block.InputPort(1).Complexity  = 'Real';
```

```
block.InputPort(1).SamplingMode  = 'Sample';
block.InputPort(1).Dimensions        = 1;

% Override output port properties
% Bondausgang 1
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode  = 'Sample';
block.OutputPort(1).Dimensions        = 1;


% Register parameters
block.NumDialogPrms     = 4;
% Parameter:     -1. Wert: double
%                -2. Kausalität:  0 = 'e2f' = f(e) oder 1 = 'f2e' = e(f)
%                -3. Invertierung: boolean
%                -4. Ausgang:     0 = kein Ausgang;
%                                 1 = e & f in y

block.SampleTimes = [-1 0];

block.SetAccelRunOnTLC(false);
block.SimStateCompliance = 'DefaultSimState';

block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
block.RegBlockMethod('Start', @Start);
block.RegBlockMethod('Update', @Update);
block.RegBlockMethod('Outputs', @Outputs);

% globale Elementezahl erstellen oder erhöhen
    global Elementezahl

    if isempty(Elementezahl)
        Elementezahl = 1
    elseif ~isempty(Elementezahl)
        Elementezahl = Elementezahl + 1
    end

% Kausalität gemäß Userangabe setzen
    switch block.DialogPrm(2).Data
        case 0
```

```
                Causality = 'f(e)';
            case 1
                Causality = 'e(f)';
            case 2
                Causality = 'auto';
            otherwise
                error('Falsche Angabe bei Eingabeparameter KAUSALITÄT!')
        end

    % Eigenschaften in .UserData zusammenfassen
    UserData.Toolbox = 'Bondgraph';
    UserData.ID = Elementezahl;
    UserData.Kategorie = 'Leitung';
    UserData.Typ = 'Gyrator';
    UserData.Wert = block.DialogPrm(1).Data;
    UserData.Name = get(gcbh,'Name');
    UserData.Invertierung = boolean(block.DialogPrm(3).Data);;
    UserData.Causality = Causality;
    UserData.Ausgang = boolean(block.DialogPrm(4).Data);
    UserData.Handle = gcbh;

    % Userdata in RTO schreiben
    set(gcbh,'UserData',UserData)

    % Element registrieren
    registrieren_Element(UserData,'intern');
    %endfunction

function DoPostPropSetup(block)
    disp('... DoPostPropSetup --> Bond(s) registrieren')
    registrieren_Bond()

    block.NumDworks = 1;

    block.Dwork(1).Name            = 'x1';
    block.Dwork(1).Dimensions      = 1;
    block.Dwork(1).DatatypeID      = 0;      % double
    block.Dwork(1).Complexity      = 'Real'; % real
    block.Dwork(1).UsedAsDiscState = true;

    %% Register all tunable parameters as runtime parameters.
    block.AutoRegRuntimePrms;
```

```
    %endfunction

function InitializeConditions(block)
    disp('... InitializeConditions --> ggf. in späteren Version benutzen?')
    %endfunction

function Start(block)
        disp('.... START ---> Analyse starten...')
        analysieren()
    block.Dwork(1).Data = 0;
    %endfunction


function Outputs(block)
    disp('... Outputs --> ggf. in späteren Version benutzen?')
    block.OutputPort(1).Data = -1;
    %endfunction

function Update(block)
    disp('... Update --> ggf. in späteren Version benutzen?')
    %endfunction
```

Literaturverzeichnis

[1] “System Dynamics”, Third Edition
Karnopp, Dean C.; Margonlis, Donald L. und Rosenberg, Roland C.
ISBN 0-471-33301-8

[2] ”Bond graph simulation and symbolic extraction toolbox in MATLAB / SIMULINK”, Revised 14.10.2005
Umarikar, Amod C.; Mishra, Tusharkant und Umanand L.
Center for Electronic Design and Technology, Indian Institute of Sience, Bangalore 560012, India

[3] Bond graph add-on block library “BG V.2.1”
Geitner, Gert-Helge
Veröffentlichung: 16.05.2006 (Letztes Update: 28.04.2010)
www.mathworks.com/matlabcentral/fileexchange/11092-bond-graph-add-on-block-library-bg-v-2-1