0pt0pt

TUHH

Technische Universität Hamburg-Harburg

Prototypenentwicklung eines regelbasierten Konfigurationssystems für den Einsatz in der Vorentwicklung von Landeklappenantriebssystemen

Studienarbeit

vorgelegt von: Babak Tavassol

Prüfer: Prof. Dr. Ralf Möller, Institut für Softwaresysteme

Prof. Dr.-Ing Udo Carl, Institut für Flugzeug-Systemtechnik

Betreuer: Dipl.-Ing. Malte Pfennig

Hamburg
Dezember 2006

Hamburg, den 22. Dezember 2006

Ich, Babak Tavassol (Student des Faches Informatik-Ingenieurwesen an der Technischen Universität Hamburg-Harburg, Matrikelnummer 21806), versichere an Eides statt, daß ich die vorliegende Studienarbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Babak Tavassol

Inhaltsverzeichnis

1 Einleitung

Moderne Verkehrsflugzeuge benötigen während verschiedenen Flugphasen unterschiedliche Flügelprofile. Zum Beispiel ist bei hohen Fluggeschwindigkeiten eine möglichst aerodynamische Flügelform mit relativ geringer Fläche erforderlich, um mittels geringen Widerstands den Treibstoffverbrauch zu minimieren. Dagegen bedarf es während Start und Landung einer Flügelform mit stärkerer Wölbung und größerer Fläche. Mit den zuletzt genannten Eigenschaften ist es möglich, den Auftriebsbeiwert zu erhöhen und das Flugzeug bei niedrigeren Geschwindigkeiten zu starten bzw. zu landen. Dadurch wird das Starten und Landen auf kurzen Stecken ermöglicht und gleichzeitig die Lärmbelästigung verringert [12]. Um für jede Flugphase das optimale Flügelprofil verwenden zu können, werden verstellbare Klappensegmente an den Flügelvorderkanten und -hinterkanten eingesetzt. Das Ein- und Ausfahren der sog. *slats* und *flaps* wird von einer zentralen Antriebseinheit im Flugzeugrumpf, der PCU (*engl.* Power Control Unit), über ein komplexes mechanisches Transmissionssystem gesteuert. Das Schema der Systemarchitektur eines solchen High-Lift-Antriebssystems ist in Abb. 1.1 dargestellt.

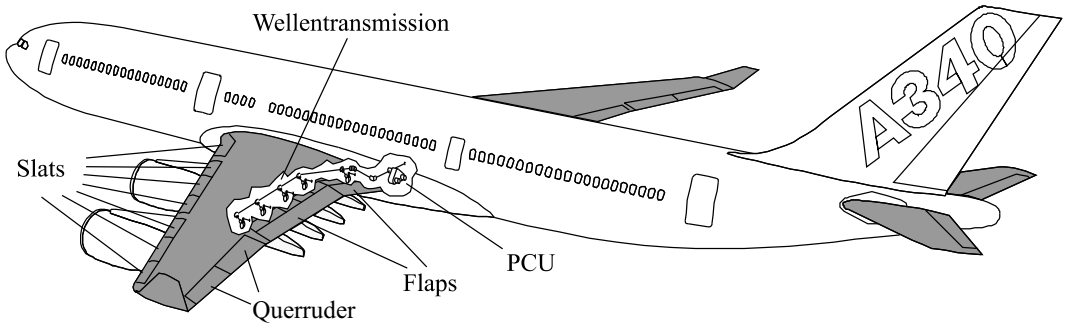

Bild 1.1: Systemarchitektur des High-Lift-Antriebssystems beim AIRBUS A340

In der Entwicklungsphase von HL-Antriebssystemen kommt der rechnergestützten Konfigurierung eine große Bedeutung zu, da mit deren Hilfe der Entwicklungsvorgang entschieden optimiert werden kann. Jedoch liegen in der Vorentwicklungsphase hauptsächlich unvollständige Informationen vor, weshalb in jener Phase eine Abschätzung unbekannter Parameter durch Heuristiken notwendig ist. Zu beachten ist, dass im weiteren Verlauf der Entwicklung zahlreiche Veränderungen von Komponentenparametern auftreten können, die das Gesamtsystem beeinflussen. Damit spätere Änderungen nachvollziehbar bleiben, wird ein System zur effektiven Repräsentation von angesammeltem *Wissen* benötigt. Eine Möglichkeit dazu stellen *wissensbasierte Konfigurierungssysteme* dar. Da mit dem Einsatz solcher Systeme das firmenspezifische Wissen in kompakter Form allen Entwicklern zugänglich gemacht werden kann, sind eine Verringerung der Entwicklungskosten und -zeit sowie der Fehlerrate erzielbar [7].

Das Thema dieser Arbeit besteht in der Konzipierung und Erstellung eines Prototypen für ein wissensbasiertes System zur Konfigurierung von HL-Antriebssystemen in der

Vorentwicklungsphase. Dieser Prototyp soll in eine bestehende JAVA-Anwendung zur Auslegung von HL-Antriebssystemen integriert werden. Dazu werden einige wissensbasierte Vorgehensweisen untersucht, aus denen anschließend eine zur Prototypenerstellung ausgewählt wird. Abschließend findet eine Bewertung auf Basis beobachterer Stärken und Schwächen des ausgewählten Systems statt.

Die Arbeit gliedert sich in sechs Kapitel einschließlich dieser Einleitung. Zunächst wird der zu untersuchende Problemfall beschrieben und eine Zusammenfassung über den Auslegungsprozess von HL-Antriebssystemen gegeben. Anschließend folgt ein Überblick über wissensbasierte Systeme, worin generelle Stärken und Schwächen aufgeführt werden. Eine Analyse bezüglich benötigter Eigenschaften des zu erstellenden Systems findet statt, auf Basis derer die Auswahl des wissensbasierten Systems für die Entwicklung des Prototypen getroffen wird. In dem Zusammenhang wird ebenfalls erläutert, woran angeknüpft wird und was bereits im JAVA-Programm vorhanden ist. In der Umsetzungsphase werden Probleme aufgeworfen und Lösungen genannt, und der Aufbau des erstellten Systems wird beschrieben. Schließlich folgt am Ende der Arbeit eine Zusammenfassung und ein Ausblick für weiterführende Arbeiten.

2 Der Entwicklungsprozess der Antriebssysteme von Hochauftriebssystemen

In diesem Kapitel folgt eine Darstellung des betrachteten Einsatzbereiches für ein wissensbasiertes System. Darin werden zunächst im Rahmen einer einführenden Systembeschreibung die Hauptfunktion und der generelle Aufbau von Hochauftriebssystemen erklärt. Im zweiten Unterkapitel werden die zum Beginn der Auslegung benötigten Vorgaben und Randbedingungen aufgeführt. Anschließend werden die Funktionen und Auslegungsschritte einzelner Komponenten genauer betrachtet und wichtige Zusammenhänge zwischen diesen geschildert. Im abschließenden Teil werden die Schritte für den Entwicklungsprozess des Gesamtsystems separat beschrieben. Die Kapitel 2.2 bis 2.4 geben die Reihenfolge des Auslegungsprozesses wieder.

2.1 Einführende Systembeschreibung

Eine Tragfläche, welche für einen effizienten Reiseflug bei hoher Geschwindigkeit ausgelegt worden ist, unterscheidet sich stark von einer speziell für Start und Landung optimierten Tragfläche. Die Länge von Start- und Landestrecken hängt vor allem von der minimalen Fluggeschwindigkeit v_{min} ab, welche sich nach [8] durch

$$v_{min} = \sqrt{\frac{2}{\rho} \cdot \frac{F_{Li}}{A_W \cdot C_{Li}(\alpha)}} \tag{2.1}$$

beschreiben lässt. Bei konstantem Flugzeuggewicht F_{Li} und gleichbleibender Luftdichte ρ kann die minimale Fluggeschwindigkeit nur durch eine Zunahme der Flügelfläche A_W und ein Ansteigen des Auftriebsbeiwertes $C_{Li}(\alpha)$ reduziert werden. Der Hauptzweck von Hochauftriebssystemen liegt in der Veränderbarkeit der zuletzt genannten Größe. Hochauftriebshilfen lassen sich unterteilen in verstellbaren Profilflächen an der Flügelvorderkante (*engl.* slats) und -hinterkante (*engl.* flaps) [12]. Mit dem Ein- und Ausfahren der slats und flaps lässt sich das Flügelprofil verändern und somit der Auftriebsbeiwert den unterschiedlichen Anforderungen verschiedener Flugphasen anpassen.

2.1.1 Genereller Aufbau eines Hochauftriebssystems

Ein High Lift System lässt sich im Allgemeinen grob in drei Teile zerlegen [12]:

- Eine zentrale hydromechanische Antriebseinheit, die Power Control Unit (PCU),
- ein Wellentransmissionssystem mit mechanischen Komponenten und Aktuatoren zur Momentübertragung und Kraftumwandlung und
- Monitore und Sensoren zur Überwachung.

Eine schematische Darstellung der wichtigsten Komponenten eines Landeklappenantriebssystems ist in Abb. 2.1 zu sehen. Die zentrale PCU liefert das benötigte Antriebsmoment für *beide* Flügel, während die beiden zuletzt genannten Bereiche *pro* Flügel

einmal vorkommen. Da der Aufbau des Flugzeugs symmetrisch ist, wird im Folgenden nur das flap-System eines Flügels betrachtet.

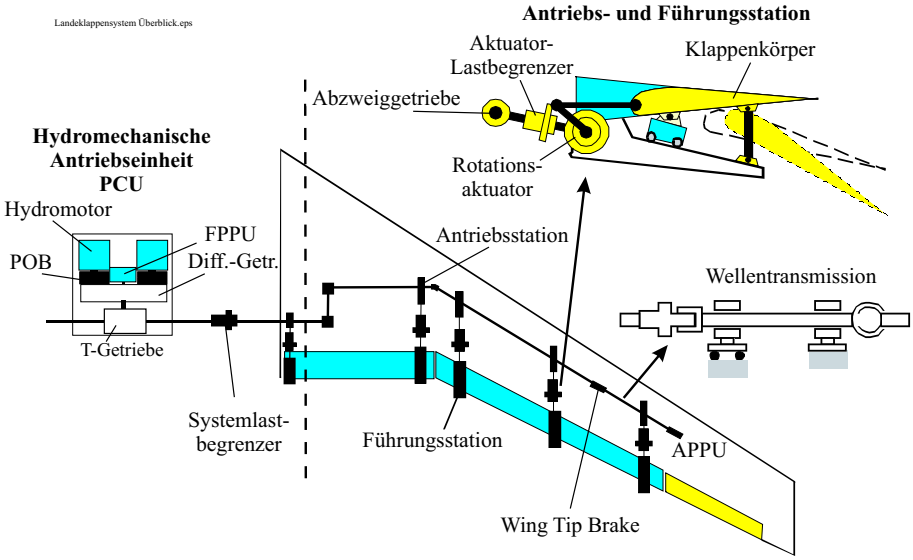

Bild 2.1: Schema der Landeklappen-Systemarchitektur des Airbus A340

Jede Tragfläche besitzt in der Regel ein *inboardflap* in der Nähe des Flugzeugrumpfes und ein *outboardflap* weiter außen [1]. Die flaps sind jeweils mit mindestens zwei Aktuatoren verbunden, die rotatorische Bewegungen des Wellenstrangs in translatorische Klappenbewegung umwandeln und somit den Ausfahrwinkel der flaps ändern können. Die Aktuatoren sind ihrerseits jeweils an einem Abzweiggetriebe (*engl.* Downdrive Gearbox, DDGB) angebunden, welches sie mit dem restlichen Transmissionssystem in Verbindung setzt. Das Transmissionssystem besteht weiterhin aus Wellen, die durch Gelenke untereinander und Stützlager mit der Struktur verbunden sind, und aus Getrieben zur Umlenkung des Transmissionsverlaufs. Die Getriebe sind vor allem notwendig zur Anpassung des Wellenstrangs an die Flügelform. Zum Beispiel werden zur Überwindung des Flügelknicks Kegelrad-Umlenkgetriebe (*engl.* Kink Bevel Gear, KBG) oder bei rechtwinkligen Verläufen Winkelgetriebe (*engl.* Right Angle Gear, RAG) eingesetzt. Um Fehlerfälle erkennen zu können, sind Sensoren an der PCU und am Ende des Transmissionssystems (an der Flügelspitze) angebracht. Zum Reagieren auf erkannte Fehler befinden sich am Anfang des Wellenstrangs (nach der PCU) ein Lastbegrenzer (*engl.* System Torque Limiter, STL) und vor dem letzten Abzweiggetriebe eine Wellenbremse (*engl.* Wing Tip Break, WTB). Der STL schützt das System und seine Einzelteile vor Zerstörung durch Überlasten, während die WTB den Wellenstrang im Falle eines Wellenbruchs abbremst und festsetzt.

[1] Bei einigen Flugzeugtypen, wie z. B. der A380, liegen drei Klappen vor.

2.2 Grundlegende Vorgaben, Anforderungen und Randbedingungen der Systemauslegung

Da im Vorauslegungsprozess die benötigten Parameter noch nicht vollständig vorliegen, muss anfangs auf Erfahrungswerten und Heuristiken zugegriffen werden. Damit sich dennoch ein relativ genaues Modell ergibt, haben eine Reihe von Vorgaben, Anforderungen und Randbedingungen aufgeführt und eingehalten zu werden. Anschließend werden Auslegungswerte herausgegeben, die sich im Laufe der Entwicklung in einem iterativen Prozess den genauen Zielwerten annähern.

2.2.1 Benötigte Vorgaben und Eingangsgrößen

Bevor der Auslegungsprozess gestartet werden kann, ist es notwendig am Anfang der Auslegung einige Daten vorzugeben. Damit werden die Grundstruktur und Einstellungen des zu untersuchenden Systems festgelegt. Für viele der folgenden Informationen können *default*-Daten vorgegeben werden. Diese können jedoch im Laufe des Auslegungsprozesses jederzeit geändert werden.

- *Systemlayout*
  Der allererste Schritt der Auslegung besteht in der Vorgabe der Struktur des Antriebssystems. Dazu gehören der Transmissionsverlauf sowie die Anzahl und Anordnung der verwendeten Komponenten und Aktuatoren.
- *Luftlasten*
  Die anliegenden Luftlasten an den Aktuatoren müssen einzeln für diese eingegeben werden.
- *Übersetzungen und Wirkungsgrade*
  Für alle Getriebe sind die Übersetzungen und Wirkungsgrade zu benennen.
- *Klappenkinematik*
  Betrachtet werden hier nur Systeme mit Rotationsaktuatoren. Für diese ist der maximale Stellwinkel vorzugeben.
- *Stellzeit*
  Die Stellzeit für einen vollständigen Ausfahrvorgang der flaps darf im fehlerfreien Betrieb auch bei ungünstigsten Bedingungen nicht überschritten werden.

2.2.2 Grundlegende Anforderungen

Neben den voraussetzenden Eingangsinformationen sind zur Einhaltung von Sicherheitsvorgaben eine Reihe von Anforderungen zu beachten. Die wichtigsten Anforderungen an HL-Antriebssytemen lassen sich wie folgt zusammenfassen [12]:

- *Lastkriterium*
  Die Luftlasten während des Fluges müssen bei jeder Geschwindigkeit für jede

Klappenstellung gehalten werden. Beim Ausfahren der Klappen wirkt eine sog. *Gegenlast*, beim Einfahren eine *helfende Last*.

- *Stellzeit-Kriterium*
  Das Ausfahren der Klappen muss innerhalb der vorgegebenen Stellzeit erreicht werden.

- *Genauigkeitskriterium*
  Die gewünschte Klappenposition muss mit einer bestimmten Genauigkeit erzielt werden.

- *Störfallkriterium*
  Das System darf durch mögliche Störereignisse nicht beschädigt werden oder ausfallen.

- *Sicherheitskriterium*
  Asymmetrische Klappenstellungen, welche die Kontrollierbarkeit des Flugzeugs gefährden würden, dürfen nicht eintreten. Diese Gefahr würde zum Beispiel bei einem Wellenbruch bestehen.

Bezug nehmend auf die oben genannten Kriterien werden im späteren Auslegungsprozess verschiedene Fehlerfälle betrachtet, welche ungünstigste Bedingungen zur Einhaltung der Anforderungen untersuchen. Neben den Fehlerfällen sind zusätzlich extreme Umgebungsbedingungen zu berücksichtigen. Dies erfolgt im nächsten Unterkapitel.

2.2.3 Betrachtete Betriebsfälle

Der gesamte Auslegungsprozess wird für zwei Betriebsfälle betrachtet, in denen unterschiedliche Temperaturniveaus berücksichtigt werden. Diese Temperaturen stellen für die Untersuchung und Dimensionierung der Parameter ausgewählte Extrembedingungen dar, bei denen maximale Belastungen für das System und die Komponenten ermittelt werden können.

- *Fall A*
  Es wird eine Umgebungstemperatur $T_{Umg} = +71°C$ betrachtet, wobei sich die geringsten Reibverluste ergeben. In diesem Fall sollen die Klappen bei maximal helfenden Lasten eingefahren werden.

- *Fall B*
  Bei einer Umgebungstemperatur von $T_{Umg} = -55°C$ stellen sich die stärksten Reibverluste ein. Dabei werden die Klappen bei maximalen Gegenlasten ausgefahren.

Ein Unterschied in der Umgebungstemperatur betrifft hauptsächlich die Übersetzungen, Wirkungsgrade und Losbrechmomente der Bauteile. Daher ist es in bestimmten Auslegungsphasen sinnvoll, eine solche Fallunterscheidung zu berücksichtigen.

2.3 Funktionale Beschreibung und Auslegung signifikanter Komponenten

Nach der Eingabe der ersten Vorgaben kann der restliche Auslegungsvorgang in zwei Teile gegliedert werden:

- Die Auslegung besonders wichtiger Komponenten mit zusätzlichen Attributen
- Die Auslegung des Gesamtsystems aus globaler Sicht

In diesem Unterkapitel ist eine Beschreibung der Auslegungsschritte für bestimmte Komponentenklassen zu finden. Gleichzeitig wird auch näher auf die Funktion der Komponenten eingegangen. Zu den erwähnten mechanischen Teilen gehören die PCU, der STL, die WTB und die Drehwellen. Aus den ersten drei genannten Teilen ergibt sich eine Unterteilung des Transmissionsverlaufs in drei Abschnitte, welche zwischen diesen Teilen liegen [12]:

- Der Bereich zwischen PCU und STL
- Der Abschnitt zwischen STL und WTB
- Der Bereich von der WTB bis zum Ende der Transmission

Für diese Bereiche und den darin enthaltenen Komponenten werden zur Dimensionierung anschließend im Unterkapitel 2.4 unter anderem Fehlerfälle betrachtet, da sich aus diesen die höchsten Belastungen ablesen lassen.

2.3.1 Power Control Unit (PCU)

Die Power Control Unit besteht aus zwei identischen Motoren, welche über ein Differentialgetriebe an das Transmissionssystem beider Flügel verbunden sind. Beim Ausfall eines der Motoren kann durch den anderen Motor das System weiterhin mit dem vollen benötigten Moment bei halber Drehzahl gespeist werden. Eine schematische Darstellung des Aufbaus einer Power Control Unit ist am Beispiel der PCU eines Airbus A340-600 in Abb. 2.2 dargestellt.

Die Aufgabe der PCU besteht darin, die erforderliche Leistung aufzubringen, um das Transmissionssystem auf eine gegebene Nenndrehzahl zu beschleunigen und die Klappen in einer gegebenen Stellzeit auszufahren. Das erfoderliche PCU-Moment M_{erf} wird aus den eingegebenen Luftlasten an den Aktuatoren und den Übersetzungen und Wirkungsgraden der Komponenten des Transmissionssystems berechnet (vgl. 2.4.1).
Falls der Fehlerfall auftritt, dass innerhalb der PCU ein Motor mit seiner Höchstdrehzahl arbeitet und einer mit seiner Nenndrehzahl, so ergibt sich die resultierende Drehzahl n_{res} (*engl.* transmission speed for motor runaway) mit

$$n_{res} = \frac{n_{Mot,max} + n_{Mot,nom}}{i_{DG}}, \tag{2.2}$$

wobei i_{DG} die Übersetzung des Differentialgetriebes beschreibt. Diese lässt sich mit der Nenndrehzahl eines Motors und der Nenndrehzahl des Systems bestimmen. Im Betriebsfall A (s. 2.2.3) liegt eine sehr hohe Umgebungstemperatur vor, wodurch sich beim Einfahren geringe Reibungen ergeben. Durch die anliegenden Luftlasten kann ein generatorisches Moment an der Antriebseinheit eintreten. Da die gewonnene Energie nicht ins Bordnetz eingespeist werden darf, muss diese „vernichtet„ werden. Im Fall B dagegen muss die PCU bei $T_{Umg} = -55°C$ stärkste Reibverluste überwinden und das maximal benötigte Drehmoment hervorbringen zum Ausfahren der Klappen.

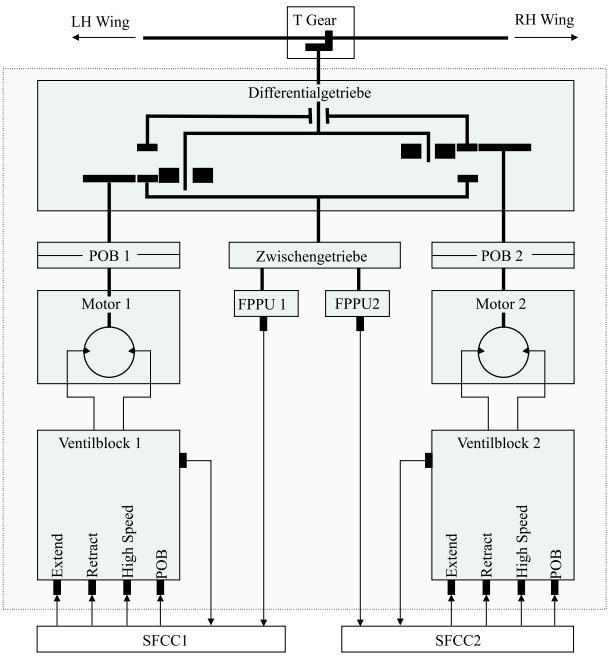

Bild 2.2: Schema der Power Control Unit des AIRBUS A340-600 [3]

2.3.2 System Torque Limiter (STL)

Lastbegrenzer dienen dazu, die Systemkomponenten bei Fehlerfällen vor zu hohen Momenten zu schützen. Auf jeder Flügelseite befindet sich in der Regel zwischen der Antriebseinheit und dem ersten Antrieb ein Systemlastbegrenzer. Der STL besitzt als zusätzliche Attribute einen oberen und einen unteren Schwellenwert: Das *lower STL setting* und das *upper STL setting*. Der Lastbegrenzer wird frühestens beim unteren und spätestens beim oberen Schwellenwert ausgelöst. Es sind zwei Schwellenwerte zu betrachten, da es sich beim STL um ein mechanisches Bauteil mit konstruktiven Toleranzen und Sicherheitsmargen handelt. Der untere Schwellenwert berechnet sich aus dem anliegenden Moment $M_{STL,max}$ und einem sicherheitsbedingten Robustheitsfaktor C_1:

$$M_{STL,LS} = C_1 \cdot M_{STL,max} \tag{2.3}$$

Das anliegende Moment $M_{STL,max}$ lässt sich wie bei der PCU aus den Schnittstellen-

lasten der Komponenten berechnen. Das upper STL setting ergibt sich aus dem lower STL setting und einer vorgegebenen Herstellermarge C_2:

$$M_{STL,US} = C_2 \cdot M_{STL,LS}. \tag{2.4}$$

2.3.3 Wing Tip Break (WTB)

Wie der Lastbegrenzer dient auch die Wellenbremse dem Abfangen bestimmter Fehlerfälle. Sie befindet sich auf jeder Flügelseite zwischen den letzten beiden Abzweiggetrieben und wird im Fehlerfall im Gegensatz zum Lastbegrenzer *elektronisch* über die Sensoren ausgelöst. Bei einem erkannten Fehlerfall arretiert sie die Wellentransmission und hält diese fest. Die WTB wird über den Fehlerfall *powered run away* dimensioniert, bei dem die Transmission angehalten werden muss, obwohl die PCU weiterhin mit voller Leistung antreibt. Das minimale Bremsmoment der WTB muss dabei gross genug sein, um den Lastbegrenzer auszulösen. Somit muss der untere Schwellenwert der WTB mindestens dem oberen Schwellenwert der STL entsprechen:

$$M_{WTB,LS} \geq M_{STL,US}. \tag{2.5}$$

Für den oberen Schwellenwert wird eine vorgegebene Herstellermarge C_3 mit berücksichtigt:

$$M_{WTB,US} = M_{WTB,LS} \cdot C_3. \tag{2.6}$$

2.3.4 Drehwellen

Die Hauptaufgabe von Drehwellen besteht in der Übertragung von Drehmomenten. Mit ihrer Hilfe ist es möglich das von der PCU aufgebrachte Moment an die einzelnen Aktuatoren zu transportieren. Jede Welle besitzt die geometrischen Parameter Innendurchmesser d, Außendurchmesser D und Länge L. Außerdem liegen folgende Werkstoffeigenschaften abhängig vom gewählten Material vor: Zulässige Torsionsspannung τ_{zul}, Schubmodul G und Dichte ρ. Aus diesen Daten können die dynamischen Parameter Steifigkeit c und Massenträgheit J folgendermaßen bestimmt werden [12]:

$$c = \frac{\pi}{32} \cdot G \cdot \frac{(D^4 - d^4)}{L}, \tag{2.7}$$

$$J = \frac{\pi}{32} \cdot L \cdot \rho \cdot (D^4 - d^4). \tag{2.8}$$

Schließlich werden die Wellen hinsichtlich folgender Anforderungen dimensioniert:

- *Kritische Drehzahl* n_{krit}
  Ab der kritischen Drehzahl fängt die Welle an, in ihrer Resonanzfrequenz zu schwingen, welches zu ihrer Zerstörung führen kann. Daher müssen alle auftretenden Drehzahlen darunter liegen. Sie wird mit folgender Formel ermittelt:

$$n_{krit} = \frac{\pi}{2 \cdot \sqrt{8}} \cdot \frac{\bar{d}}{L^2} \cdot \sqrt{\frac{\hat{E}_x}{\rho}}. \tag{2.9}$$

Dabei entsprechen $\hat{E}_x$ dem E-Modul in Rohr-Längsrichtung und $\bar{d}$ dem mittleren Durchmesser.

- *Maximal zulässiges Moment $M_{t,max}$*
  Abhängig von der zulässigen Torsionsspannung des gewählten Materials ergibt sich ein maximales Drehmoment, welches an der Welle auftreten darf und wie folgt berechnet wird:

$$M_{t,max} = \frac{\pi}{16} \cdot (D^3 - d^3) \cdot \tau_{zul}. \tag{2.10}$$

- *Zulässige Biegespannung σ_{max} (Hanging Man)*
  Dieses Kriterium wird bei AIRBUS unter der Annahme überprüft, dass sich ein Arbeiter von 100 kg an die Mitte der Welle hängen könnte, ohne dass diese brechen würde. Die nötige Berechnung wird mit der folgenden Formel druchgeführt:

$$\sigma_{max} = \frac{1}{\pi} \cdot \frac{F \cdot L}{D^3 - d^3}. \tag{2.11}$$

  Hierbei stellt F eine Kraft dar, die mittig an einer Welle angreift.

2.4 Der Auslegungsprozess des Gesamtsystems

Zusätzich zu der Auslegung der einzelnen Komponenten ist im zweiten Teil des Auslegungsprozesses die Konfiguration des Gesamtsystems aus globaler Sicht durchzuführen. Dieser Teil wird in fünf Schritte gegliedert, welche während des Auslegungsprozesses chronologisch abgearbeitet werden. Von besonderer Bedeutung sind hierbei die folgenden untersuchten Auswirkungen von Fehlerfällen infolge eines Transmissionswellenbruchs oder Klemmfalls, welche für das System dimensionierend sind:

- *Overspeed*
  Es tritt eine Überschreitung der zulässigen Höchstdrehzahl der Transmission auf (s. Kap. 2.4.2).

- *Blowback*
  Durch einen Wellenbruch erreichen die aufgebrachten Motormomente die abgetrennten Bereiche nicht, und die Klappen in diesen Bereichen werden durch die Luftlasten wieder hineingedrückt. Man unterscheidet zwischen symmetrischer und asymmetrischer Blowback-Fälle (s. Kap. 2.4.3).

- *Asymmetrie*
  Es entsteht eine asymmetrische Klappenstellung zwischen den beiden Tragflächen. Diese kann z.B. durch einen asymmetrischen Blowback-Fall entstehen (s. Kap. 2.4.5).

2.4.1 Grundlegende Definitionen

In diesem Schritt werden für das Gesamtsystem grundsätzliche Daten eingegeben und berechnet, welche in den anschließenden Schritten weitere Verwendung finden. Dazu gehören Informationen zum Ausfahrprofil, benötigten Gesamtmoment und Gesamtwirkungsgrad des Systems.

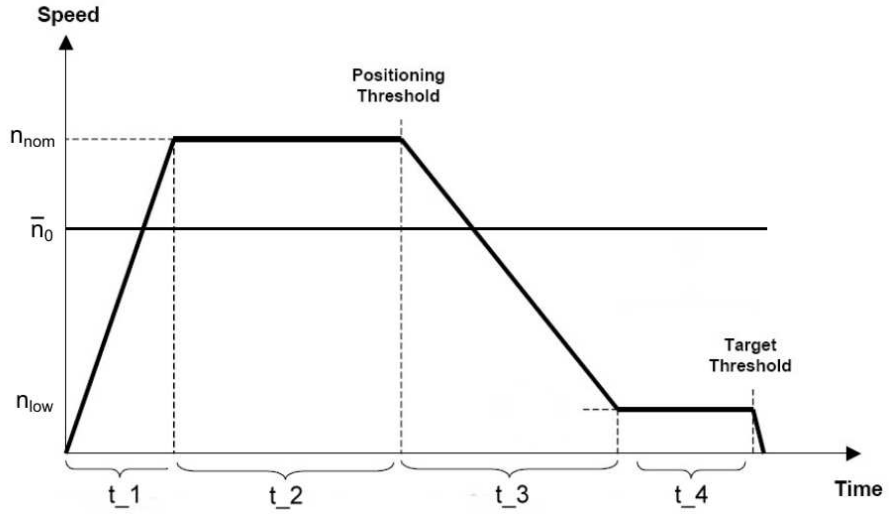

Bild 2.3: Typisches Ausfahrdrehzahlprofil der Landeklappen [11]

Der Ausfahrvorgang der Start- und Landeklappen lässt sich generell in fünf zeitliche Phasen einteilen (s. Abb. 2.3): Zunächst wird die Transmission angefahren bis sie eine Nenndrehzahl n_{nom} (*engl.* nominal speed) erreicht (t_1). Anschließend dreht sich die Transmission bei konstanter Nenndrehzahl während der Zeitspanne t_2 weiter. Während t_3 wird die Drehzahl dann reduziert auf eine niedrigere Drehzahl n_{low}(*engl.* low speed), welche während t_4 konstant beibehalten wird. Nach t_4 werden die Transmission und die Klappen schließlich in der (relativ kurzen) Zeit t_5 zum Stillstand gebracht. Der gesamte Ausfahrvorgang findet in der Ausfahrzeit t_{ext} (*engl.* extension time) statt. Für sämtliche Zeitphasen wird ein linearer Verlauf der Transmissionsdrehzahl angenommen.

Die Werte für die Nenndrehzahl n_{nom}, die Zeitintervalle t_1, t_3, t_4, t_5, t_{ext} und den Ausfahrwinkel δ_{ext} (*engl.* extension angle) müssen zu Beginn festgelegt werden. Aus der Nenndrehzahl lässt sich darauffolgend die niedrigere Drehzahl n_{low} berechnen, welche als 18% der Nenndrehzahl angenommen wird. Im Weiteren werden aus den eingegebenen Daten die durchschnittliche Drehzahl n_{avg} (*engl.* average speed) und die Anzahl der Wellenumdrehungen U pro Ausfahrvorgang (*engl.* shaft rotations per extension) berechnet:

$$n_{avg} = \frac{n_{nom}(\frac{1}{2}t_1 + t_2) + n_{low}(t_3 + t_4 + \frac{1}{2}t_5) + \frac{1}{2}t_3(n_{nom} - n_{low})}{t_{ext}}, \tag{2.12}$$

$$U_{nom} = n_{nom} \cdot t_{ext}, \tag{2.13}$$

$$U_{avg} = n_{avg} \cdot t_{ext}. \tag{2.14}$$

Das benötigte Gesamtmoment M_{erf}, welches die Antriebseinheit aufbringen muss, ergibt sich aus der Summe der Schnittstellenlasten aller Komponenten unter Berücksichtigung der Übersetzungen i, Wirkungsgrade η und Schleppmomente M_D. Dazu werden zunächst die Schnittstellenlasten M_{DT} (*engl.* Drive Torque) der einzelnen Komponenten für das Ein- und Ausfahren der Klappen ermittelt:

$$M_{DT} = \begin{cases} \frac{M_L}{i \cdot \eta_S} + M_D; n > 0, M_L > 0 \\ \frac{M_L}{i} + M_D; n > 0, M_L < 0 \\ \frac{M_L}{i} - M_D; n < 0, M_L > 0 \\ \frac{M_L}{i \cdot \eta_S} - M_D; n < 0, M_L < 0 \end{cases} \tag{2.15}$$

Dabei entspricht das Lastmoment M_L jeweils dem M_{DT} der vorherigen Komponente im Wellenstrang. Bei den Aktuatoren werden die angreifenden Luftlasten als M_L angenommen. Das erforderliche PCU-Moment ergibt sich demnach für beide Flügel mit:

$$M_{erf} = 2 \cdot M_{DT,PCU}. \tag{2.16}$$

Der Gesamtwirkungsgrad des Systems $\eta_{Transmission}$ lässt sich anschließend aus M_{erf} und dem Gesamtlastmoment $M_{load} = \sum_i M_{L,i}$ unter Berücksichtigung sämtlicher Übersetzungen berechnen:

$$\eta_{Transmission} = \left[\frac{|M_{load}|}{|M_{erf}|} \right]^{sign(n_{PCU})} \tag{2.17}$$

2.4.2 Drehzahlen

Für die Auslegung des Gesamtsystems sind signifikante Drehzahlen und Drehzahlgrenzen festzulegen bzw. zu berechnen. Diese sind auf den Hauptwellenstrang bezogen und werden für sämtliche Fehlerbetrachtungen benötigt. Ein Überblick bezüglich Drehzahlgrenzen lässt sich aus dem Diagramm in Abb. 2.4 entnehmen. Aus der Nenndrehzahl der PCU n_{nom} lässt sich die PCU Höchstdrehzahlgrenze $n_{PCU,OS}$ (*engl.* PCU overspeed threshold) ableiten, welche im Allgemeinen 10% über n_{nom} liegt. Weiterhin wird der Grenzwert für kritische Drehzahlen n_{krit} etwa 10% oberhalb der gewählten PCU-Höchstdrehzahlgrenze angesetzt. Alle kritischen Drehzahlen der Transmissionswellen müssen unterhalb des *whirling speed limits* liegen, da sie sonst in den Bereich ihrer Resonanzfrequenzen gelangen können.

Neben der gewählten PCU Höchstdrehzahlgrenze ist weiterhin die minimale Höchstdrehzahlgrenze des Systems $n_{Sys,OS,min}$ (*engl.* min. system overspeed threshold) zu bestimmen. Diese wird für die Systemüberwachung benötigt und muss zwischen n_{res} und der maximalen Höchstdrehzahlgrenze $n_{Sys,OS,max}$ (*engl.* max. system overspeed threshold) liegen, welche sich aus der Blowback-Analyse ergibt (s. Kap. 2.4.3). Letzlich

existiert nur ein Grenzwert $n_{Sys,OS}$ als Höchstdrehzahlgrenze, jedoch gelten die Anforderungen $n_{Sys,OS,min} < n_{Sys,OS} < n_{Sys,OS,max}$. Als weitere Eingabe ist die höchste zulässige dynamische Drehzahl der Transmission im Fehlerfall $n_{dyn,max}$ (*engl.* max. dynamic transmission failure speed) erforderlich. Diese wird u.a. bei der Blowback-Untersuchung zur Berechnung der maximalen Höchstdrehzahlgrenze benötigt und beträgt erfahrungsgemäß 3000 rpm.

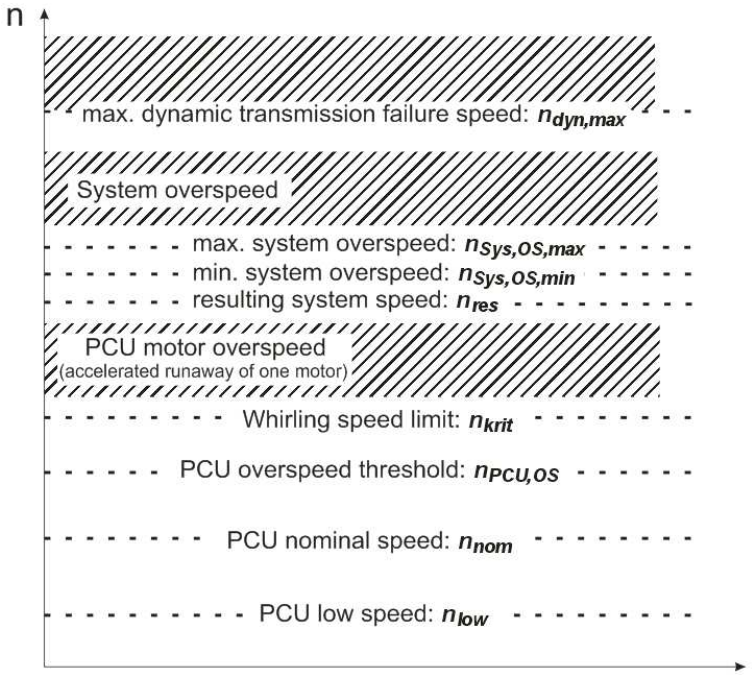

Bild 2.4: Drehzahlgrenzen für die Systemüberwachung [11]

2.4.3 Blowback-Untersuchung

Unter einem Blowback versteht man das ungewollte Hineindrücken von entkoppelten Klappen nach einem Wellenbruch durch anliegende Luftlasten. Es ist zwischen drei Fällen zu unterscheiden, welche jeweils davon abhängen an welcher Stelle der Wellenbruch stattgefunden hat und welche Klappen somit von der PCU entkoppelt sind:

- Fall 1: Alle Klappen auf beiden Flügelseiten sind entkoppelt. Bei diesem Fall liegt ein sog. *symmetrischer Blowback* vor, d. h. dass auf beiden Seiten keine Verbindung zwischen der PCU und der Transmission vorhanden ist.
- Fall 2: Alle Klappen auf einer Seite sind abgetrennt, während die andere Seite intakt ist. Es liegt ein *asymmetrischer Blowback* vor.
- Fall 3: Die äußere Klappe auf einer Seite ist entkoppelt vom Rest der Transmission. Hier liegt ebenfalls ein *asymmetrischer Blowback* vor.

Bezüglich der globalen Betriebsfälle (s. Kap. 2.2.3) ist bei der Blowback-Analyse nur Fall A dimensionierend, weshalb in diesem Abschnitt nur dieser Fall betrachtet wird.

Wie bereits in Kap. 2.4.2 erwähnt, besteht eines der Ziele in diesem Schritt darin, die maximale Höchstdrehzahlgrenze des Systems $n_{Sys,OS,max}$ (*engl.* max. system overspeed threshold) aus den genannten Fehlerfällen zu ermitteln. Doch bevor dies erfolgen kann, sind zunächst andere Kennwerte festzulegen bzw. zu ermitteln.
Durch das bereits ermittelte benötigte Gesamtmoment M_{erf} und den Trägheiten $J_{ges} = \sum_{n}^{i=1} J_i$ kann die Beschleunigung mit

$$\dot{\omega} = \frac{|M_{erf}|}{J_{ges}} \tag{2.18}$$

kalkuliert werden. Zur Berechnung der Beschleunigungszeit t_A werden anschließend die minimale Höchstdrehzahlgrenze $n_{Sys,OS,min}$ und die höchste zulässige dynamische Drehzahl der Transmission im Fehlerfall $n_{dyn,max}$ aus Kap. 2.4.2 verwendet:

$$t_A = \frac{1}{\dot{\omega}} \cdot (n_{dyn,max} - n_{Sys,OS,min}). \tag{2.19}$$

Die Gesamterkennungszeit t_D, in der ein Fehler erkannt und das System abgebremst werden soll, berechnet sich mit

$$t_D = t_A - t_B, \tag{2.20}$$

wobei t_B der Bremsverzögerung entspricht und erfahrungsgemäß zwischen 20 ms und 50 ms festzulegen ist. Außerdem hat der Wert von t_D stets größer als 0 zu sein. Neben der Bremsverzögerung t_B sind außerdem die Abtastrate der Sensoren t_s (*engl.* sampling rate), die Anzahl der Bestätigungszyklen für eine Fehlerdetektion N_b (*engl.* confirmation cycles) und die Algorithmusdauer zur Datenauswertung der Sensoren t_c festzulegen. Aus den drei zuletzt genannten Eingaben lässt sich die erforderliche Erkennugszeit $t_{D,erf}$ durch

$$t_{D,erf} = t_s \cdot (N_b + 1) + t_c \tag{2.21}$$

herleiten. Anschließend kann durch $t_{D,erf}$ ebenfalls die erforderliche Beschleunigungszeit $t_{A,erf}$ berechnet werden:

$$t_{A,erf} = t_B + t_{D,erf}. \tag{2.22}$$

Schließlich kann das am Anfang dieses Unterkapitel genannte Ziel der Berechnung der maximalen Höchstdrehzahlgrenze des Systems $n_{Sys,OS,max}$ durch

$$n_{max} = |\dot{\omega} \cdot t_{A,erf} - n_{dyn,max}| \tag{2.23}$$

erreicht werden, wobei $n_{dyn,max}$ die maximal auftretende Drehzahl der Transmission im Fehlerfall bezeichnet.

2.4.4 Limit Loads

Aus der Analyse der Fehlerfälle in der Blowback-Untersuchung (s. Kap. 2.4.3) ergeben sich die Grenzlasten des Systems (*engl.* limit loads) bzw. die maximalen Lasten, auch *Peakmomente* genannt. Die limit loads stellen die maximal zulässigen Belastungen im Betrieb dar und gehören zu den dynamischen Aspekten der Auslegung. Extreme dynamische Beanspruchungen werden gewöhnlicherweise im Laufe der Entwicklung durch Simulationen genauer ermittelt. Im Folgenden werden die limit loads, von einer Vorauslegung ausgehend, durch grobe Abschätzungen vereinfacht berechnet. Es wird dabei von einem Ersatzmodell ausgegangen, bei dem die Transmission als Reihenschaltung von Torsionsfedern mit konzentrierten Massenträgheiten der einzelnen Komponenten beschrieben wird [12](s. Abb. 2.5).

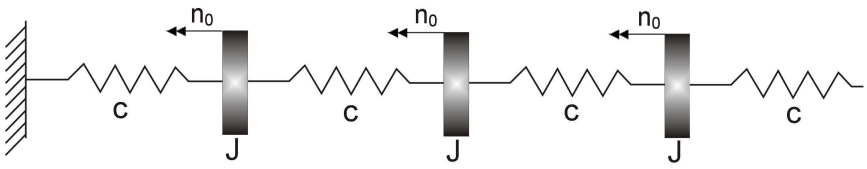

Bild 2.5: Torsionsschwinger als Ersatzmodell für die Transmission [12]

Zur Berechnung der limit loads werden einerseits die dynamisch ermittelten Werte der Massenträgheit J_{ges} und der maximalen Höchstdrehzahlgrenze des Systems $n_{Sys,OS,max}$ (s. Kap. 2.4.3) benötigt. Andererseits ist eine Bestimmung der Eigenfrequenz des Systems ω_1 erforderlich. Aus ω_1 und J_{ges} lässt sich die Steifigkeit des Systems c^* durch

$$c^* = \omega_1^2 \cdot J_{ges} \tag{2.24}$$

berechnen. Unter der Annahme, dass die Höchstdrehzahl des Systems n_0 dem Betrag der maximalen Höchstdrehzahlgrenze $n_{Sys,OS,max}$ entspricht, erhält man anschließend die Grenzlasten mit

$$M_{limit} = \sqrt{c^* \cdot J_{ges}} \cdot n_0. \tag{2.25}$$

2.4.5 Asymmetry Position Limit

Asymmetrische Klappenfehlstellungen in Folge eines Wellenbruchs, die zu unkontrollierten Flugzuständen führen würden und damit katastrophale Folgen hätten, gilt es zu vermeiden. Um dies zu erreichen, bedarf es neben der Detektion von Blowback-Fällen einer zusätzlichen Analyse zur Detektion von Asymmetrien. In der Praxis kommen zur Asymmetrie-Detektion spezielle Sensoren an den Enden des Wellenstranges auf jeder Seite zum Einsatz (s. Kap. 2.1.1). Falls diese eine Asymmetrie feststellen, werden die Wing Tip Brakes aktiviert und die Klappen in ihrer Position gehalten.

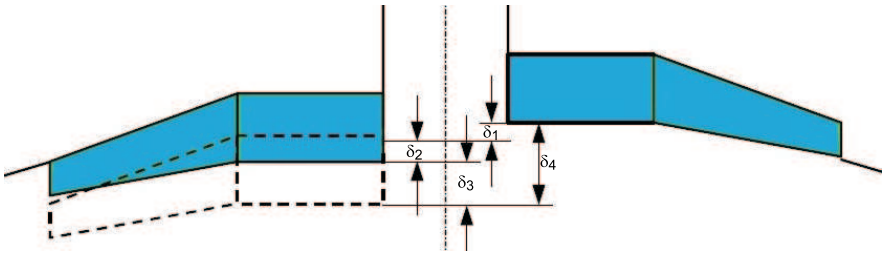

Bild 2.6: Asymmetriegrenzen [11]

Zur Berechnung der Asymmetrie-Grenze, bei welcher die Sensoren den Fehler detektieren müssen, sind zunächst folgende Daten festzulegen (s. Abb. 2.6):

- Die Klappenfehlstellung δ_1, die sich aufgrund von Fertigungstoleranzen ergibt. Erfahrungsgemäß wird ein Wert von 1° angenommen.

- Der Winkel δ_2, der sich aus der mechanischen Einstellung des Systems ergibt. Dieser ist normalerweise relativ gering und bertägt 1,5°.

- Die maximal zulässige Klappenfehlstellung der flaps δ_4, welche aus aerodynamischen Betrachtungen erhaltbar ist und gewöhnlicherweise 5° beträgt.

Aus den festgelegten Werten lässt sich daraufhin die Asymmetrie-Grenze δ_3 durch

$$\delta_3 = \delta_4 - \delta_1 - \delta_2 \tag{2.26}$$

ermitteln. Aus zusätzlichen Sicherheitsgründen wird δ_3 noch ein Faktor δ_S von 1° abgezogen, woraus sich die Asymmetrie-Grenze δ_{mon} (*engl.* monitoring threshold) ergibt, bei der ein Fehler detektiert werden muss.

Der zeitliche Ablauf eines solchen Detektierungsvorgangs ist in Abb. 2.7 dargestellt, wobei darin lineare Beschleunigungen vorausgesetzt werden. Nach einer Wellentrennung werden die intakten Klappen von der PCU weiterhin ausgefahren, während gleichzeitig die abgekoppelten Teile mit der maximal zulässigen Drehzahl $n_{PCU,OS}$ (s. Kap. 2.4.2) durch die Luftlasten eingefahren werden. Nach der Zeit t_1 wird die Asymmetriegrenze erreicht, woraufhin die Sensoren den Fehler in der Zeit t_D (*engl.* detection time) erkennen und die Bremsen aktivieren müssen. Anschließend müssen die Bremsen das System nach einer Bremsverzögerung t_B zum Stillstand bringen. Die Gesamtabweichung der abgetrennten Klappen von den intakten Klappen muss geringer als δ_4 sein.

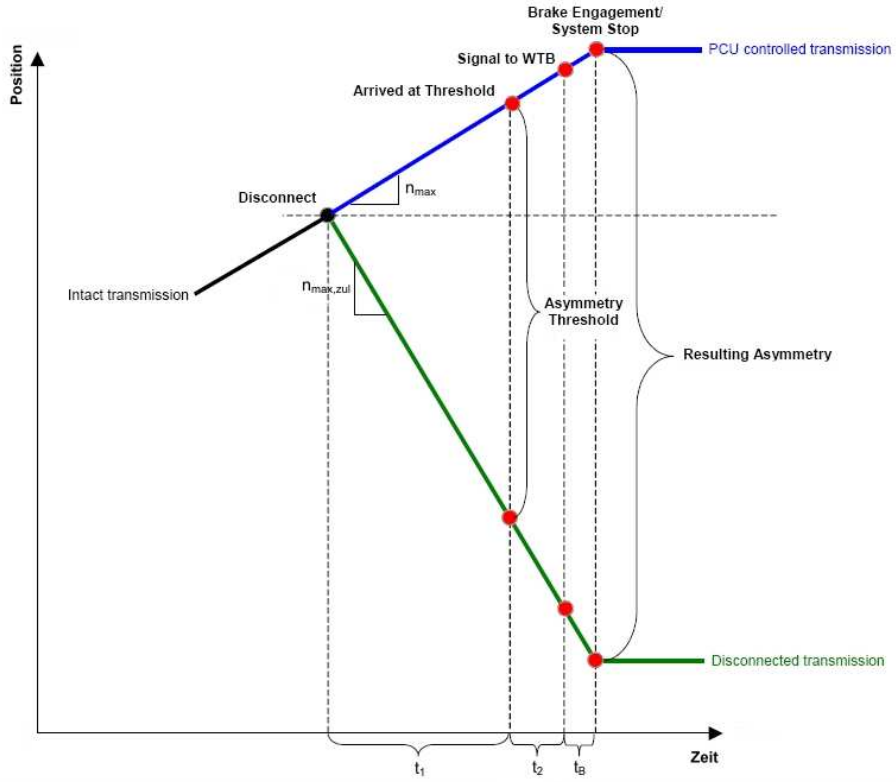

Bild 2.7: Positionsverlauf der intakten und abgetrennten Transmission beim Asymmetriemonitoring [11]

Mit den feststehenden Werten des Ausfahrwinkels δ_{ext} und der Anzahl der Wellenumdrehungen pro Ausfahrvorgang bei durchschnittlicher Drehzahl U_{avg} aus Kap. 2.4.1, kann die Asymmetrie-Monitor-Grenze U_{asym} (*engl.* asymmetry monitoring threshold) ermittelt werden:

$$U_{asym} = \frac{\delta_{nom}}{\delta_{ext}} \cdot U_{avg}. \tag{2.27}$$

Durch rechnerspezifische Kennwerte sind Umsetzungsgrenzen der Systemüberwachung vorhanden, weshalb ein Bereich von $\delta_{band} = 0.3°$ zu berücksichtigen ist. Daraus ergibt sich ein Asymmetry-Monitor-Band $U_{asym,band}$ (*engl.* asymmetry detection band) mit

$$U_{asym,band} = \frac{\delta_{band}}{\delta_{ext}} \cdot U_{avg}. \tag{2.28}$$

Die Positionsdifferenz der Klappen U_{pos} mit der Bremsverzögerung t_B, der minimalen Höchstdrehzahlgrenze des Systems $n_{Sys,OS,min}$ und der PCU overspeed threshold $n_{PCU,OS}$ (s. Kap. 2.4.2) wird in Wellenumdrehungen durch

$$U_{pos} = (|n_{PCU,OS}| + |n_{Sys,OS,min}|) \cdot t_B \tag{2.29}$$

dargestellt. Anschließend ergibt sich aus der Zusammenfassung der Gleichungen (2.28) und (2.29) die resultierende Asymmetrie in Wellenumdrehungen

$$U_{asym,res} = U_{asym} + U_{asym,band} + U_{pos} \tag{2.30}$$

und bezüglich der Klappenwinkel

$$\delta_{asymm,res} = \frac{\delta_{ext}}{U_{avg}} \cdot U_{asym,res}. \tag{2.31}$$

Der resultierende Asymmetrie-Klappenwinkel $\delta_{asymm,res}$ muss kleiner als die festgelegte maximal zulässige Klappenfehlstellung der Flaps δ_4 sein.

Die Zeit zum Erreichen der Asymmetrie-Grenze t_1 (s. Abb. 2.7) beträgt mit Gleichung 2.27

$$t_1 = \frac{U_{asym}}{n_{PCU,OS} + n_{Sys,OS,min}}. \tag{2.32}$$

Die Detection Time t_D, in der die Sensoren den Fehler erkennen und die Bremsen aktivieren müssen wird durch

$$t_D = \frac{U_{asym,band}}{n_{PCU,OS} + n_{Sys,OS,min}} \tag{2.33}$$

gewonnen.

3 Wissensbasierte Konfigurierung

Dieses Kapitel stellt eine grobe Einführung in das Gebiet der wissensbasierten Systeme dar. Es folgt zunächst eine Beschreibung der fundamentalen Prinzipien von wissensbasierten Systemen. Im zweiten Unterkapitel wird dann ein Überblick über einige wissensbasierte Methoden zur Konfigurierung gegeben, wobei ihre allgemeinen Vor- und Nachteile diskutiert werden.

3.1 Grundlagen der wissensbasierten Konfigurierung

Wissensbasierte Systeme sind ein Teilgebiet der Künstlichen Intelligenz und wurden als Forschungswerkzeuge erstmals in den 60er Jahren entwickelt. Oft werden wissensbasierte Systeme auch Expertensysteme oder wissensbasierte Expertensysteme genannt, obwohl zwischen den Begriffen des *Wissens* und des *Expertenwissens* zu unterscheiden ist. Nach der Definition von Professor Edward Feigenbaum [6], einem Pionier im Bereich der wissensbasierten Technologien, ist ein Expertensystem...

> „ein intelligentes Computerprogramm, dass Wissen und Inferenz-Prozeduren benutzt, um Probleme zu lösen die schwer genug sind, so dass sie zur Lösung signifikantes menschliches Expertenwissen benötigen."

Einfacher gesagt handelt es sich bei wissensbasierten Systemen um Computersysteme, die Entscheidungen treffen aufgrund von gespeichertem Wissen.

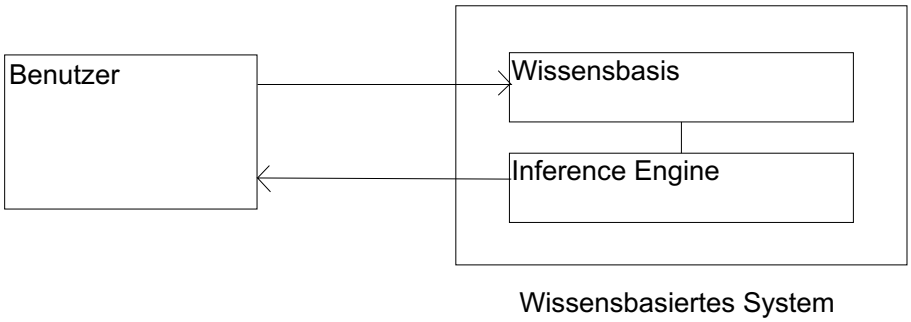

Bild 3.1: Grundlegende Funktionsweise eines wissensbasierten Systems [6]

Abb. 3.1 stellt das grundlegende Schema eines wissensbasierten Systems dar. Der Benutzer gibt spezielles Wissen und seine Anfragen in das System ein, woraufhin das System ein Resultat oder eine Entscheidung aufgrund der eingegebenen Daten zurückliefert. Intern besteht das System aus zwei wesentlichen Komponenten: Der *Wissensbasis*, welche das eingegebene Wissen speichert und der *inference engine*, die aus den Fakten der Wissensbasis Schlüsse zieht [6]. Trotz ihrer Bezeichnung können Expertensysteme nicht den menschlichen Experten ersetzen, sondern fungieren in der Anwendung vielmehr als unterstützendes Werkzeug des Experten.

Es gibt verschiedene Klassifikationen für wissensbasierte Systeme. Eine Art der Klassifikation sieht eine Unterteilung nach Problemdomänen vor. Eine Problemdomäne ist hierbei das spezielle Fachgebiet, wie z. B. Medizin oder Informatik, worin ein Experte Probleme sehr gut lösen kann. Eine andere Klassifikation besteht aus einer domänenunabhängigen, abstrahierten Einteilung nach Problemklassen (s. [13], [7]).

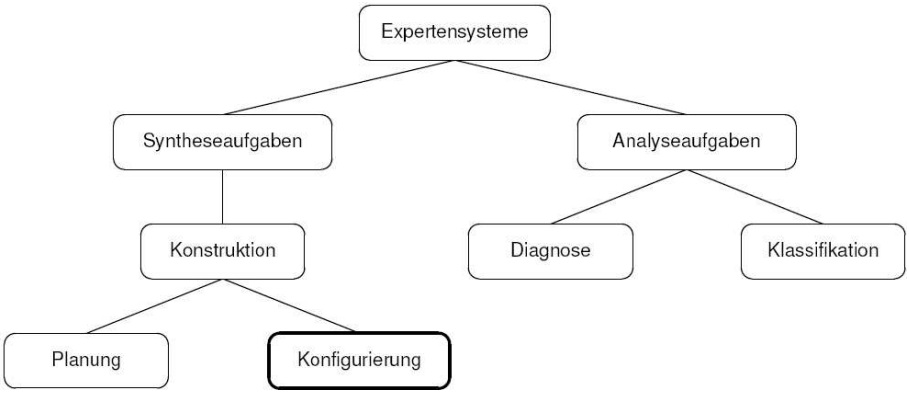

Bild 3.2: Einordnung der Konfigurierung [13]

Wie in Abb. 3.2 dargestellt kommt es in dieser aufgabenorientierten Klassifikation zu einer Unterteilung in zwei Hauptgruppen: Den *Syntheseaufgaben* und den *Analyseaufgaben*. Es sei angemerkt, dass diese Gebiet keine scharfen Grenzen definieren und sich schneiden oder gegenseitig ergänzen können. Während bei den Anaylseaufgaben existierende Objekte behandelt werden, befassen sich Syntheseaufgaben mit der Konstruktion neuer Objekte. Konstruktionsaufgaben wiederum lassen sich weiter in die Unterpunkte *Planung* und *Konfigurierung* zerlegen. Mit Konfigurierung ist in diesem Kontext die Erstellung von neuen Systemen unter Beachtung bestimmter Randbedingungen gemeint. Folgende Merkmale lassen sich bei Konfigurierungsaufgaben generell beobachten (vgl. [10], [13], [7], [1]):

- Eine Spezifikation der Aufgabe (**Konfigurationsziele**), welche die Anforderungen an die Konfiguration spezifiziert.
- Eine Menge von **Objekten** in der Problemdomäne mit ihren Attributen (Parameter).
- Eine Menge von **Relationen** zwischen den Domänenobjekten.
- Wissen über die Vorgehensweise bei der Konfigurierung (**Kontrollwissen**).
- Rücknahme von Entscheidungen: Zur Erhaltung einer Lösung müssen anfangs festgelegte Entscheidungen später wieder revidierbar sein, da diese oftmals nicht haltbar sind.
- Hierarchisches Vorgehen (*Top-Down*-Vorgehen): Analog zu Hierarchien von Objekten in vielen Anwendungsgebieten wird auch der Konfigurierungsprozess oft in eine hierarchiesche Gliederung zerlegt.

- Behandlung von Abhängigkeiten: Die Repräsentation und Verarbeitung von Abhängigkeiten zwischen Objekten nimmt im Konfigurierungsprozess eine zentrale Rolle ein.

3.2 Methoden der wissensbasierten Konfigurierung

Im Folgenden wird eine Auswahl gängiger Methoden zur wissensbasierten Konfigurierung vorgestellt. Zu jeder Methode wird die Art der Wissensmodellierung kurz erläutert. Danach werden die allgemein bekannten Vor- und Nachteile dargestellt. Auf Basis dieser Informationen wird nach der Analyse in Kapitel 4 entschieden, welche Methode zur Auslegung von Hochauftriebssystemen eingesetzt und untersucht werden soll.

3.2.1 Regelbasierte Konfigurierung

Bei der regelbasierten Konfigurierung kommen sogenannte *deklarative* Paradigmen zum Einsatz [6]). Im Gegensatz zu *prozeduralen* Programmen, bei denen angegeben wird *wie* eine Lösung implementiert werden muss, wird in deklarativen Programmen das Ziel von den Methoden zum Erreichen des Ziels getrennt. Stattdessen wird angegeben *was* erreicht werden muss. Dazu benutzen regelbasierte Systeme zur Repräsentation von Wissen *Fakten* und *If-Then-Regeln*, sogenannte *productions*. Die Regeln stellen Abhängigkeiten zwischen den Fakten dar und haben folgende Form:

IF <Bedingungen> THEN <Aktionen>

Der Bedingungsteil, auch *left-hand side* (LHS) genannt, spezifiziert wann eine Regel aktiviert werden soll. Sind alle Bedingungen in der LHS erfüllt, so werden die Aktionen im Aktionsteil, der *right-hand side* (RHS), ausgeführt.

In Abb. 3.3 ist die typische Architektur eines regelbasierten Systems zu sehen. Die *Regelbasis* enthält sämtliche Regeln, die das System kennen muss. Daneben werden in der *working memory* die Informationen gespeichert, mit denen das System arbeitet. Diese Informationen können sich sowohl auf die Premissen, als auch auf die Aktionen der Regeln beziehen und werden meist in Form von *Fakten* dargestellt. Die *inference engine* kontrolliert den Prozess der Anwendung von Regeln auf die working memory. Normalerweise geschieht dies in diskreten Zyklen, die folgendermaßen ablaufen [5]:

1. *Pattern matcher*
   Der *pattern matcher* vergleicht die LHS der Regeln mit den Daten in der working memory und entscheidet, welche Regeln *aktiviert* werden sollen. Eine Liste der aktivierten Regeln, die sog. *Konfliktmenge*, wird erstellt.

2. *Agenda*
   Die Konfliktmenge wird geordnet zu der Liste der Regeln, die „abgefeuert“ [2]

[2]Das Abfeuern von Regeln ist gleichzusetzen mit der Ausführung ihrer RHS

werden sollen.

3. *Execution Engine*
   Die *execution engine* feuert die erste Regel auf der Agenda ab, was zu einer Änderung der working memory führen kann, und der gesamte Prozess wird von vorn wiederholt.

Grundsätzlich ist bei regelbasierten Expertensystemen zwischen zwei Vorgehensweisen zu unterscheiden: *Vorwärtsverkettung* und *Rückwärtsverkettung* [7]. Bei der Vowärtsverkettung werden vorliegende Fakten mit den LHS verglichen, wobei geprüft wird welche Regeln ausführbar sind. Bei der Rückwärtsverkettung dagegen werden die RHS erfasst und mit einem Ziel verglichen. Regeln, deren RHS das Ziel enthalten werden in einer Menge zusammengefasst, aus der schließlich eine Regel ausgewählt wird. Die LHS der ausgewählten Regel ist dann das neue Ziel. Die Rückwärtsverkettung ist mit dem Vorgehen bei einer Polizeiermittlung zu vergleichen, bei der Beweise gesammelt werden (analog zu den RHS), die dem Ziel dienen, rückschließend herausfinden zu können was stattgefunden hat (analog der LHS der ausgewählten Regeln)[5].

Die Stärken regelbasierter Systeme liegen vor allem in der Beschreibung von kausalen Abhängigkeiten zwischen Objekten und in der Repräsentation und Evaluierung von heuristischen Relationen, sogennanten *Daumenregeln* [7]. Anscheinend existiert außerdem eine Gemeinsamkeit zwischen Regeln und dem kognitiven Prozess des Menschen, denn nach der Beschreibung von Newell und Simon [6] scheinen Regeln eine natürliche Art der Modellierung zu sein, wie Menschen Probleme lösen würden. Bei der Anbindung von regelbasierten Systemen an komplexeren Anwendungsprogrammen lassen sich die Regeln vom Rest des Programms trennen und in der Regelbasis seperat verwalten. Weiterhin sind (fortgeschrittenere) regelbasierte Systeme von relativ kurzen Antwortzeiten geprägt. Dies gilt insbesondere bei der Vorwärtsverkettung, also beim datenorientierten Inferenzmechnismus. Für die Auswahl einer Regel aus der Konfliktmenge, wenn mehrere Regeln gegebene Anforderungen erfüllen, existieren unterschiedliche Strategien. Es

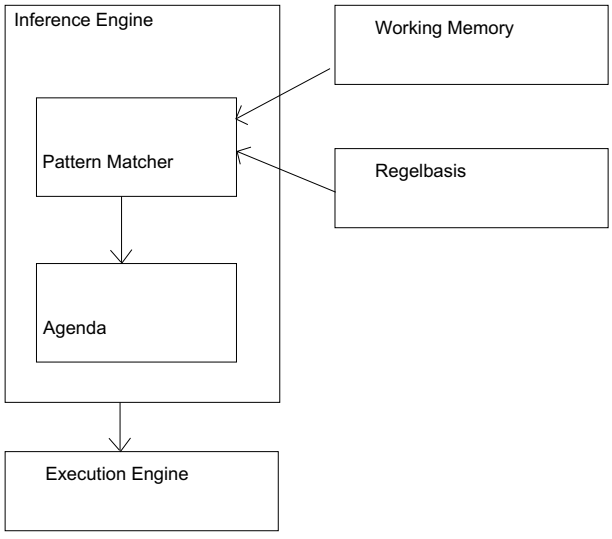

Bild 3.3: Typische Architektur eines regelbasierten Systems [5]

gibt sowohl domänenunabhängige Strategien (z. B. Auswahl aus der aktuellsten oder speziellsten Regel) als auch domänenabhängige Vorgehensweisen wie z. B. Prioritäten, Bewertungsfunktionen, Metaregeln [10]. Eine Unterteilung der Wissensbasis in Module ermöglicht eine Verfeinerung der Regelbasis und somit das Abfeuern bestimmter Regelmengen bei unterschiedlichen Zuständen.

Neben den Vorteilen sind während der intensiven Nutzung in der Vergangenheit allerdings auch viele Schwächen von regelbasierten Systemen zum Vorschein gekommen. So findet durch die Repräsentation des gesamten Wissens als Regeln und Fakten eine Vermischung von Domänenwissen und Kontrollwissen statt. Dadurch entstehen enorme Unzulänglichkeiten während der Wissensaquisition und der Sicherstellung der Konsistenz in der Wissensbasis. Außerdem treten Schwierigkeiten bezüglich der Adaptierbarkeit und Wartung auf, da die Veränderung einer kleinen Menge von Abhängigkeiten eine große Menge von Regeln betrifft. Zudem wird die Modularisierung der Wissensbasis damit kritisiert, dass dadurch zusätzliche Abhängigkeiten zwischen den Regeln entstehen und die Regeln keine unabhängigen Wissenseinheiten mehr darstellen würden [7][13]. Als weiterer Nachteil ist schließlich die schlechte Integration von Benutzeranweisungen zu nennen.

Klassische Beispiele aus dem Bereich der regelbasierten Systeme sind XCon, Clisp, Jess und Prolog [13][6]. Generell wird aufgrund der Schwächen empfohlen, regelbasierte Systeme nur in lokalen Anwendungen einzusetzen oder bei komplexeren Aufgaben in Kombination mit anderen Methoden in einem *Konfigurationsframework* [7].

3.2.2 Strukturbasierte Konfigurierung

In jedem Konfigurationssystem besteht die Notwendigkeit der Repräsentation des Objektwissens. Meist wird dazu eine objektorientierte bzw. framebasierte Repräsentationsform gewählt. Diese macht eine zusammengefasste Spezifikation der Objektstruktur und ihren zulässigen Belegungen möglich [7]. Die strukturbasierte Konfiguration orientiert sich an der Struktur des Domänenmodells [10]. Dazu wird innerhalb der Wissensbasis eine Begriffshierarchie erzeugt, bei der die Objekte in *taxonomischen* und *partonomischen* Hierarchien klassifiziert werden [13]. Taxonomische Hierarchien stellen Objekte vertikal über *is-a*-Relationen miteinander in Verbindung, während partonomische Hierarchien über *has-parts*-Relationen Aggregate mit ihren Komponenten horizontal verbinden. Beim Konfigurationsprozess mit einer solchen Begriffshierarchie ist sowohl ein *top-down*- als auch ein *bottom-up*-Vorgehen möglich [7]. In diesem Zusammenhang hat der Begriff der *Strukturinformation* zwei Bedeutungen [7]:

- Die Struktur einer Lösung (Konfiguration) ist vordefiniert durch die konzeptuelle Hierarchie und besteht aus hierarchisch geordneten Objekten.
- Ziele können in Teilziele zerlegt werden entsprechend ihrer Struktur.

Durch dieses Level der Abstraktion können generische Beschreibungen von Objekten entsprechend Objektklassen geformt werden. Das Wissen wird strukturierter und

effektiver repräsentiert, da Vererbungsmechanismen redundante Beschreibungen einschränken oder gar vermeiden [7].

Auf der anderen Seite besteht ein Nachteil dieses Verfahrens darin, dass kleine Veränderungen in der Spezifikation große Effekte haben können (*Schwelleneffekt*)[14]. Zum Beispiel wird während des Konfigurierungsprozesses einer Maschine eine Unterkomponente durch eine andere ersetzt, was dazu führt, dass ein Gehäuse nicht mehr ausreicht oder ein inakzeptables Gewicht zustande kommt. Kurz gesagt: Globale Ressourcen sind abhängig von lokalen Entscheidungen.

Bekannte Vertreter strukturbasierter Systeme stellen EngCon sowie seine Vorgänger PlaKon und KonWerk dar [7]. Die zentrale Voraussetzung des strukturbasierten Ansatzes ist die Strukturiertheit der Domäne. Weniger strukturierte Domänen sind daher ungünstig geeignet für diese Vorgehensweise.

3.2.3 Constraintbasierte Konfigurierung

Unter einem *Constraint* versteht man eine Restriktion oder Relation zwischen Objekten bzw. ihren Attributen. Mit Constraints können ungerichtete Abhängigkeiten in der Wissensbasis dargestellt und evaluiert werden und die Zielmenge eingegrenzt werden [2]. Sie ermöglichen zum einen das Ableiten von spezifischen Eigenschaften aus vorhandenen Eigenschaften, zum anderen die Überprüfung der Konsistenz der aktuellen Konfiguration (*engl.* constraint satisfaction)[7][13]. Um dies zu erreichen, werden Constraints zu *Constraint-Netzen* zusammengefasst, in denen sie über ihre Variablen Relationen zu einander aufbauen. Durch *Constraint-Propagation* können die Wertebereiche der Variablen durch Auswerten der Constraints sukzessive eingeschränkt werden. Lokale Änderungen der Wertebereiche breiten sich durch wiederkehrende Propagation durch das gesamte Constraint-Netz aus und schränken so den Lösungsraum immer weiter ein [2][13]. Im Kontrast zu anderen Anwendungsgebieten existieren bei der Konfigurierung die folgenden Anforderungen an Constraint Systemen [7]:

- Das Constraint-Netz sollte inkrementell während des Konfigurationsvorgangs aufgebaut werden. Zu keinem Zeitpunkt (außer nach Feststellung der finalen Lösung) ist die komplette Anzahl von Constraints bekannt.

- Constraints sollten so formuliert werden, dass sie die Anzahl neuer Objekte entweder verlangen oder eingrenzen.

- Das Constraint-System sollte unterschiedliche Arten von Constraints, wie zum Beispiel numerische, symbolische oder funktionale Constraints, verarbeiten können. Im vorliegenden Problemfall würden allerdings nur numerische Constraints benötigt werden.

Als größter Nachteil von Constraint Systemen mit numerischen Constraints hat sich in der Praxis die Komlexität erwiesen. Da diese *NP-vollständig* ist [2], ist eine komplette Evaluation der Constraint-Relationen nicht möglich. Dem zufolge werden nur jeweils die Ober- und Untergrenzen von numerischen Parametern „propagiert“ jedoch nicht jeder

individuelle Wert [7]. Deshalb sollte das Constraint-System einen Trade-off zwischen Effizienz und Qualität erlauben.

Ein kommerzielles Constraint-System, welches als Konfigurierungswerkzeug alle Anforderungen erfüllt, ist derzeit nicht bekannt [7]. Jedoch gibt es eine Reihe vielversprechender Ansätze wie das *Dynamic Constraint Satisfaction*, das *Generative Constraint Satisfaction* oder die bedingte Propagation in *ConBaCon*. Allgemein sind constraintbasierte Systeme besonders gut geeignet für die flexible Beschreibung und Auflösung von ungerichteten Abhängigkeiten.

3.2.4 Ressourcenbasierte Konfigurierung

Die ressourcenbasierte Konfigurierung basiert auf einem Domänenmodell, welches Beziehungen zwischen Komponenten durch den Austausch von Ressourcen (abstrakten Leistungen) beschreibt [10]. Der elementare Grundgedanke ist der, dass Komponeneten in einem technischen System eingesetzt werden, weil sie etwas anbieten, dass vom Gesamtsystem oder von anderen Komponenten gebraucht wird [7]. Die Schnittstellen zwischen den Komponenten werden durch die ausgetauschten Ressourcen spezifiziert, ebenso das System und seine Umgebung. Beispiele für technische Ressourcen stellen Stromverbrauch und Speicherkapaziät dar, für wirtschaftliche Ressourcen Preis und Wartungsaufwand. Komponenten stellen Ressourcen zur Verfügung und konsumieren gleichzeitig andere Ressourcen. Die initialen Ressourcenanforderungen zu Beginn der Konfiguration stellen die Aufgabenstellung dar. In einem iterativen Konfigurierungsprozess werden die initialen Ressourcendefizite erkannt und durch Komponenten, die entsprechende Ressourcen anbieten, ausgeglichen (*engl.* balancing methods)[7]. Der Konfigurierungsverlauf entspricht somit einem Prozess des Bilanzierens und Ausgleichens von Ressourcenanforderungen [13].

Der Vorteil dieser Modellierungsweise liegt im geringen Wartungsaufwand, da abstrakte Ressourcen über eine längere Lebensdauer verfügen als konkrete Komponenten. Beispiele für den Einsatz ressourcenbasierter Systeme sind Cosmos [7] und KIKon [13].

3.2.5 Fallbasierte Konfigurierung

Der fallbasierte Ansatz geht davon aus, dass Wissen über bereits gelöste Konfigurierungsaufgaben innerhalb einer Fallbibliothek vorliegt und benutzt werden kann zur Lösung neuer Aufgaben. Die Grundannahme besteht darin, dass ähnliche Probleme zu ähnlichen Lösungen führen. Die Methoden zu Auswahl der Fälle können in zwei Gruppen unterteilt werden: Solche, die ein Preprozessing zur Strukturierung der Fallbasis durchführen und Methoden, die einfach auf der unstrukturierten Menge von fallbasiertem Wissen arbeiten [7]. Ein Vorteil besteht darin, dass beliebige Teillösungen einbezogen werden können. Es ist daher nicht notwendig ausschließlich gesamte Fälle zu betrachten. Generell lassen sich zwei Vorgehensweisen zur Auswahl von fallbasiertem Wissen aufzählen [7]:

- Auswahl eines hinreichend ähnlichen Falls und das Ergebnis adaptieren (*engl.* transformational analogy)

- Auswahl des fallbasierten Wissens in individuellen Schritten. Die Fallbasis kann bei dieser Vorgehensweise als Kontrollwissen der Konfigurierung betrachtet werden (*engl.* derivational analogy).

In jedem Fall muss eine Anpassung des fallbasierten Wissens an das aktuelle Problem durchgeführt werden, welche normalerweise von großem Umfang ist. Außerdem sind die angebotenen Lösungen relativ *konservativ*, d. h. diese beinhalten nur ein geringes Maß an Innovation, da diese aus bekannten Fällen entnommen worden sind. Deshalb existiert in der Regel auch keine kausale Erklärung für die vorgeschlagene Lösung. Diese Art der Konfigurierung eignet sich aufgrund der genannten Nachteile bestenfalls für Probleme, die keine oder kaum Innovation verlangen und bei denen die Kombination alter Lösungsteile hinreichend ist, oder zur Kombination mit anderen Verfahren bei denen vergangene Lösungen mit berücksichtigt werden sollten.

4 Analyse und Auswahl

Dieses Kapitel beschäftigt sich mit der Analyse des zu erstellenden Systems. Dazu wird im Kapitel 4.1 zunächst die vorhande JAVA-Umgebung untersucht, an welcher das wissensbasierte System angeknüpft werden soll. Danach werden die nötigen Komponenten des zu erstellenden Systems genauer betrachtet. Auf Basis des Analyseteils folgt dann die Auswahl eines wissensbasierten Systems zur Prototypenerstellung. Abschließend in diesem Kapitel wird das ausgewählte System näher vorgestellt.

4.1 Die vorhandene JAVA-Anwendung

Der erste Schritt der Analyse besteht notwendigerweise in der Betrachtung des bereits Vorhandenen, an das angeknüpft werden soll. In diesem Unterkapitel folgt anfangs eine funktionale Betrachtungsweise der vorhandenen JAVA-Applikation. Anschließend wird der Aufbau des Programms genauer untersucht.

4.1.1 Funktionale Beschreibung

Das Ziel dieser Arbeit besteht in der Fertigstellung und Integration eines wissensbasierten Systems in eine bestehende JAVA-Anwendung für den Vorentwurf der Antriebssysteme von Hochauftriebssystemen. In dem zu Beginn der Arbeit vorliegenden Programm waren für diesen Zweck bereits einige Grundfunktionalitäten vorhanden, die im Folgenden beschieben werden.

Abb. 4.1 zeigt die Benutzeroberfläche (*engl.* graphical user interface, GUI) der Anfangsversion der JAVA-Applikation. Darin ist unter der Menü- und Symbolleiste eine Unterteilung des Hauptfensters in zwei Bereiche zu sehen. Auf der linken Seite liegt eine generelle Klassifikation von Flugzeugbauteilen in Form einer Baumstruktur

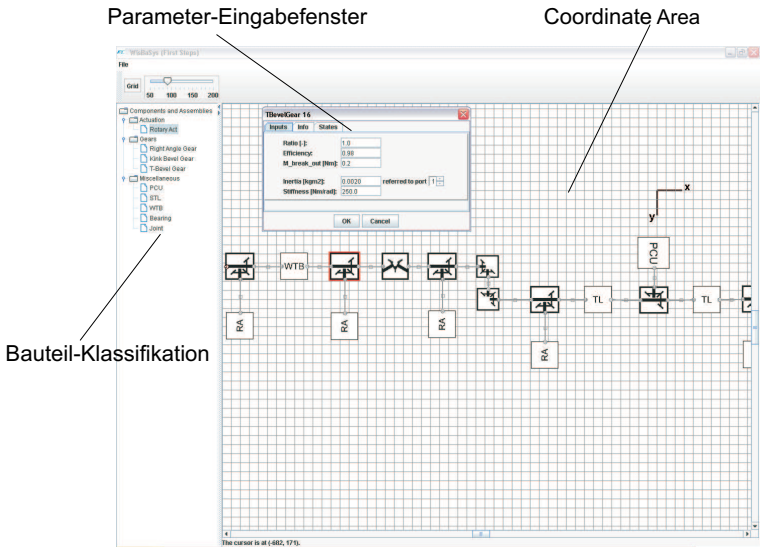

Bild 4.1: Benutzeroberfläche der JAVA-Anwendung

vor, während auf der rechten Seite die Arbeitsfläche, ähnlich einer Art „digitales Reißbrett“ zur Erstellung von Entwürfen vorhanden ist. Die Instanziierung von Bauteilen auf der Arbeitsfläche erfolgt durch einfaches *drag and drop* von Bauteilen aus der Baumstruktur hinüber auf eine beliebige Position auf der Arbeitsfläche. Dabei wird davon ausgegangen, dass von jedem Bauteil ein unendlicher Vorrat vorhanden ist. Die Erstellung der Transmissionswellen zwischen den Bauteilen (vgl. Kap. 2.3.4) lässt sich anschließend mit Anklicken und Halten der linken Maustaste auf einem Port (Schnittstelle eines Bauteils), Ziehen der Maus und Loslassen der linken Maustaste auf einem anderen Port bewerkstelligen. Auf diese einfache Weise kann der Benutzer über die GUI ein Layout für ein Antriebssystem aufbauen. Einsicht in die spezifischen Attribute bzw. Parameter eines jeden Bauteils kann per Doppelklick auf das jeweilige Teil gewonnen werden, welches ebenfalls in Abb. 4.1 dargestellt ist. Im geöffneten Parameterfenster sind Änderungen der Bauteil-Attribute über die Textfelder möglich. Neben jedem Textfeld befindet sich zusätzlich eine Checkbox zum „Einfrieren“ von Variablenwerten, d. h. die Werte sind im aktivierten Zustand der Checkbox nicht editierbar.

4.1.2 Aufbau des JAVA-Programms

Nachdem eine Beschreibung der hauptsächlichen Funktionalitäten der Anfangsversion gegeben ist, wird nun der Aufbau des Programms näher beleuchtet. Es sei darauf hingewiesen, dass nur Programmteile beschrieben werden, die für den weiteren Verlauf dieser Arbeit von Bedeutung sind.

Das Paketdiagramm[3] in Abb. 4.2 stellt die Abhängigkeiten der zentralen Module des JAVA-Programms in der anfänglichen Phase dar.

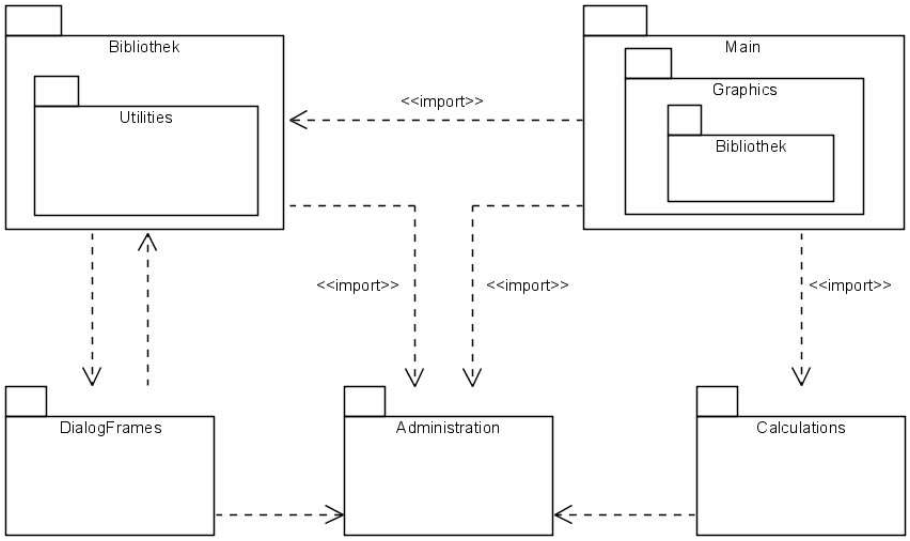

Bild 4.2: Abhängigkeiten der Pakete

[3]entsprechend der Unified Modeling Language 2.1 (UML)

Eine detaillierte Betrachtungsweise zum Package `Main` stellt die Abb. 4.3 dar. Daraus ist entnehmbar, dass dieses Paket sowohl den Eintrittspunkt des Programms mit der Klasse `MyApp`, als auch das Hauptfenster der Anwendung mit `MainPanel` enthält. Das darin enthaltene Subpaket `Main.Graphics` beherbergt die Klasse `CoordinateArea` zur Erstellung und Verwaltung der graphischen Arbeitsfläche (s. Kap. 4.1.1). Schließlich ist das weitere Unterpaket `Main.Graphics.Bibliothek` zu erwähnen, welches die Oberklasse für alle Bauteile `GraphicComponent` beinhaltet. Da die Bauteile eine große Zahl an Parametern für unterschiedliche Zwecke besitzen, ist es zur übersichtlichen Verarbeitung erforderlich, die graphischen Attribute von den Eigenschaften des physikalischen Bauteils zu trennen. Die Klasse `GraphicComponent` beinhaltet lediglich die Attribute bezüglich der graphischen Repräsentation der Bauteile, wie zum Beispiel ihre Position auf der Arbeitsfläche. Da diese Parameter während der weiteren Bearbeitung zur Identifikation der physikalischen Bauteile wichtig sind (s. u. Paket `Administration`), werden die Klassen bezüglich der physikalischen Teile von `GraphicComponent` abgeleitet.

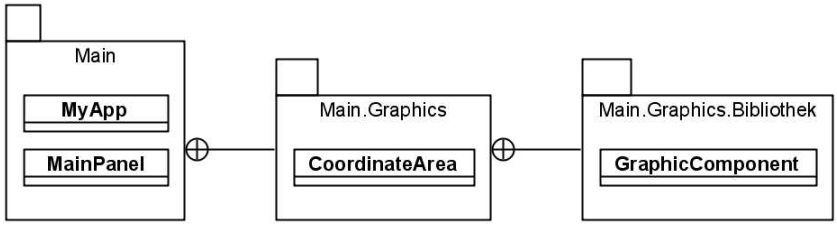

Bild 4.3: Das Main-Paket

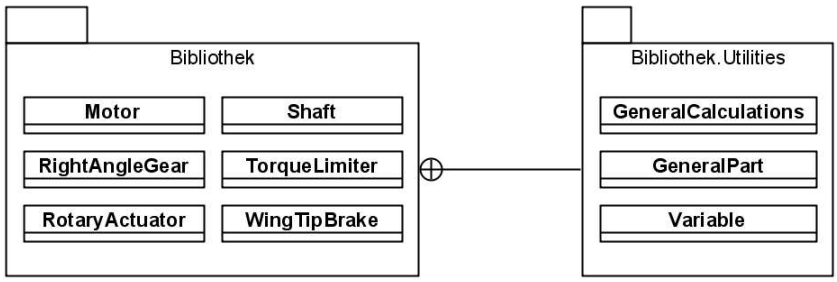

Bild 4.4: Das Bibliothek-Paket

Die Klassen zur Beschreibung der physikalischen Komponenten sind im Package `Bibliothek` zusammengefasst. Einige dieser Klassen sind in Abb. 4.4 repräsentativ dargestellt. Das Subpaket `Bibliothek.Utilities` enthält die Oberklassen für die physikalischen Teile `GeneralCalculations` und `GeneralPart` sowie die Klasse `Variable` zur Speicherung von Parameterwerten. Die genaue Ableitungshierarchie aller Bauteil-Klassen ist zur Übersicht in Abb. 4.5 repräsentiert. `GeneralCalculations` beinhaltet Variablen, die in jeder Bauteilgruppe vorkommen (im Diagramm exemplarisch durch die Variable `Ratio` dargestellt) und die Liste `ParameterList` zur Verwaltung sämtlicher

Variablen des Teils. Die Kind-Klasse `GeneralPart` wird zur Speicherung des Komponentennamens und der Ports verwendet. Schließlich fügen die Komponenten auf der spezialisiertesten Hierarchieebene bauteilspezifische Variablen zu den Membern hinzu.

Wie in Abb. 4.2 zu sehen ist, nimmt das Package `Administration` eine zentrale Lage im Programm ein. Wichtig für diese Arbeit ist lediglich die darin enthaltene Klasse `Admin`, welche die Aufgabe der Verwaltung aller erstellten Bauteile hat. Alle anderen Pakete übernehmen diese Klasse über eine *import*-Beziehung. Beim Starten des Programms wird genau eine Instanz von `Admin` erstellt, welche über die gesamte Benutzungsdauer verwendet wird.

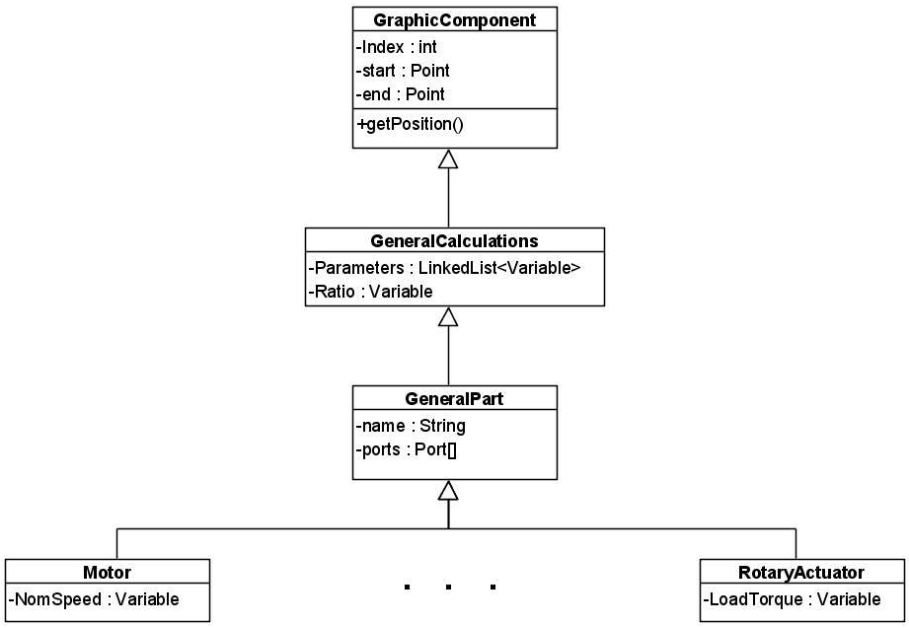

Bild 4.5: Klassenhierarchie der Bauteilklassen

Das Package `DialogFrames` enthält die Klasse `StandardDialog` zum Anzeigen und Modifizieren von Bauteil-Parametern (vgl. Kap. 4.1.1). Aus Abb. 4.2 ist entnehmbar, dass zwischen `DialogFrames` und `Bibliothek` eine gegenseitige Abhängigkeit besteht. Das ist damit zu erklären, dass `StandardDialog` die Variablenwerte aus den `GeneralPart`-Objekten zur graphischen Repräsentation übernimmt, aber gleichzeitig die vom User eingegebenen Werte in `StandardDialog` zu den `GeneralPart`-Objekten übermittelt werden. Schließlich bleibt das Modul `Calculations` zu erklären, welches Klassen mit Algorithmen zur dynamischen Berechnung von wichtigen Variablenwerten enthält. Diese Algorithmen werden später während des Auslegungsprozesses von Bedeutung sein. Die beinhaltete Klasse `WindowMinimumDriveTorque` berechnet dynamisch je nach Layout das nötige Motormoment (s. Kap. 2.4.1). `GetLoadTorque` und `InertiaCalc` führen Algorithmen zur Berechnung der Gesamtlast und der Gesamtträgheit durch.

4.2 Das zu erstellende System

Das im vorigen Kapitel erläuterte Grundprogramm liefert hauptsächlich Funktionalitäten zur Layouterstellung, sowie zur Modifikation von einzelnen Beuteilparametern. Damit wird lediglich der erste Schritt in einem Auslegungsprozess abgedeckt. Das zu erstellende System sollte der Grundapplikation zur Durchführung eines Auslegungsvorgangs angemessene Erweiterungen hinzufügen. In diesem Zusammenhang werden im Folgenden die Hauptziele des benötigten Endprogramms genannt:

- Das schrittweise Bearbeiten der Auslegungsphasen sollte in einem programmgeführten Workflow durchlaufen werden können.
- Zusätzlich soll der Zugriff auf einzelne Bauteilattribute und globale Parameter möglich sein.
- Es sollte jederzeit sichergestellt sein, dass alle Systemanforderungen erfüllt sind. Nach Eingaben sollten nötige Constraints geprüft werden. Falls Forderungen verletzt sind, so sollte der Benutzer informiert werden und ihm sollten nach Möglichkeit Grenzwerte aufgezeigt werden.
- Eine benutzerfreundliche, übersichtliche und intuitive Bedienung mit möglichst geringer Granularität aus Benutzersicht sollte vorliegen.
- Die Wissensbasis muss jederzeit erweiterbar, editierbar und für den Benutzer einsehbar sein. Ein einfacher Zugriff auf die Wissensbasis für schnelle Veränderungen zur Laufzeit ist wünschenswert.
- Die Nachvollziehbarkeit von Ergebnissen und Entscheidungen sollte gewährleistet sein.

In Abb. 4.6 werden diese Ziele in einem Use-Case-Diagramm verdeutlicht. Mit dem Use-Case „Auslegung erstellen“ kann die zusammenhängende Abarbeitung des Auslegungsprozesses begonnen werden. Nach der Auswahl dieses Anwendungsfalls hat der Benutzer zunächst ein Layout zu erstellen. Danach wird er durch die Schritte zur Auslegung signifikanter Bauteile geführt (vgl. Kap. 2.4). Die Parameter *aller* betrachteten Bauteile können je nach Auslegungsphase zusammen in übersichtlicher Weise (z. B. tabellarisch) dargestellt und modifiziert werden. Anschließend werden die Schritte zur globalen Auslegung durchlaufen.
Neben dem Auslegungsprozess ist das Konfigurieren von *einzelnen* Bauteilattributen im Grundprogramm bereits gegeben (s. Kap. 4.1.1) und muss lediglich um den optionalen Zugriff auf die globalen Parameter erweitert werden. Dies ist im Use-Case-Diagramm durch eine *extension* des Falls „Bauteilparameter modifizieren“ hervorgehoben.

Neben den grundlegenden Zielen sind außerdem konkretere Anforderungen an die Geschwindigkeit und die Strukturierung des Systems zu benennen. Diese sind im Einzelnen:

- Nach Benutzereingaben hat eine sofortige Aktualisierung aller betroffenen Daten zu erfolgen. Dies sollte durch eine möglichst schnelle Verarbeitung in akzeptabler Zeit stattfinden.

- Bei Eingabe unzulässiger Werte sollte neben der Benachrichtigung an den Benutzer (s. o. Hauptziele) der eingegebene Wert nicht angenommen werden. Auch andere betroffene Variablen sollten demnach nicht geändert werden.

- Das System sollte in logische Module, entsprechend der Auslegungsphasen, unterteilt sein.

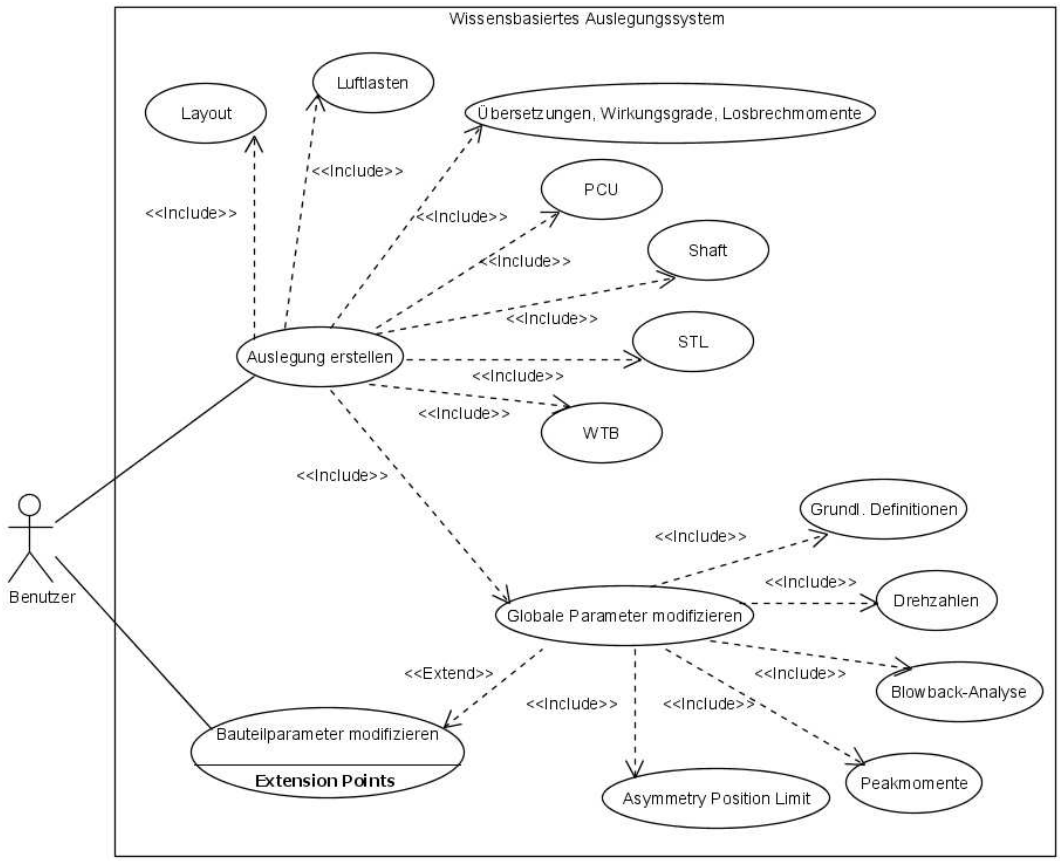

Bild 4.6: Use-Case-Diagramm für das zu erstellende System

4.3 Auswahl der Konfigurierungsmethode

Auf Basis der betrachteten wissensbasierten Konfigurationssysteme in Kapitel 3.2 und der durchgeführten Analyse in Kapitel 4.2 kann nun die Entscheidung für eine wissensbasierte Methode zur Prototypenerstellung gefällt werden.

Aus den untersuchten Methoden der Konfigurierung scheint die anfangs betrachtete regelbasierte Konfigurierung die beste Wahl darzustellen, da es in der Vorauslegung stark

auf die Beschreibung von heuristischen Relationen ankommt. Für diese sind im vorliegenden Problemfall hauptsächlich vorwärtsverkettende Regeln erforderlich. Ebenfalls positiv zu bewerten sind die schnellen Antwortzeiten und die Möglichkeit der Modularisierung, die beim Auslegungsprozess benötigt werden. Da der betrachtete Problemfall keine übermäßig große Anzahl an Regeln erfordert (60 Regeln), sondern eher als übersichtliches, lokales Problem einzuordnen ist, sind die Nachteile wie zum Beispiel die schlechte Wartbarkeit und die Vermischung vom Domänenwissen mit dem Kontrollwissen im Vergleich mit den Vorteilen geringer zu gewichten. Wie stark allerdings die Nachteile tatsächlich eintreten werden, sei Gegenstand der weiteren Untersuchung in dieser Arbeit.

Der strukturbasierte Ansatz kann in die Konfigurierung mit einfließen, da an einem objektorientierten JAVA-Programm angeknüpft wird, aus dem die nötigen Klassen (s. Kap. 5.1) samt taxonomischen und partonomischen Hierarchien auf die Elemente der Wissensbasis abgebildet werden. Ein rein strukturbasiertes System würde jedoch die Abhängigkeiten zwischen den Variablen nicht hinreichend repräsentieren können.

Neben dem regelbasierten Ansatz scheint die constraintbasierte Konfigurierung ebenfalls eine sehr gute Wahl darzustellen, da Constraints eine abstraktere Beschreibung als Regeln darstellen und insbesondere die ungerichteten Abhängigkeiten zwischen den Variablen effektiver repräsentieren und verarbeiten können. Bei jedoch nur 10 solcher Constraints und überwiegend vorwärtsverkettenden Abläufen im vorliegenden Fall stellt sich die Frage, ob ein komlexeres System als eine Regelbasis überhaupt nötig ist. Aus anwendungsorientierter Sicht jedenfalls scheint die regelbasierte Konfigurierung den minimalsten Aufwand zur Lösung des untersuchten Problemfalls mit sich zu bringen.

Die ressourcenbasierte Konfigurierung scheint für das vorliegende Problem ungeeignet zu sein, da eine Abbildung der Abhängigkeiten auf den Prozess des Bilanzierens von Ressourcenanforderungen einen zusätzlichen Aufwand darstellen würde. Der Vorteil des geringen Wartungsaufwandes bei dieser Methode würde bei der relativ kleinen Anzahl der Abhängigkeiten nicht zur Geltung kommen.

Ebenso wäre ein rein fallbasiertes Vorgehen in der betrachteten technischen Domäne nicht sinnvoll, da in dieser durch Innovation Veränderungen auftreten können. Es sollte daher keine zu konservative Vorgehensweise gewählt werden.

Aufgrund der obigen Überlegungen fällt die Auswahl auf ein regelbasiertes System, da ein solches anscheinend den benötigten Anforderungen mit dem geringsten Aufwand gerecht wird. Aus den vorhandenen Tools zur regelbasierten Konfigurierung scheint das Shell JESS (Java Expert System Shell) zur Erweiterung der bestehenden JAVA-Applikation am Besten geeignet zu sein. Der Grund dafür ist einerseits die Möglichkeit der Erstellung, Verwendung und Ausführung sämtlicher JAVA-Klassen und -Methoden vom JESS-Code aus, ohne dabei JAVA-Code kompilieren zu müssen. Dadurch können taxonomische und partonomische Hierarchien aus der objektorientierten Welt auf Jess-Fakten abgebildet werden. Auf der anderen Seite sind die Einbettung von JESS-Anweisungen im JAVA-Programm und der einfache, dynamische Zugriff auf die Wissensbasis als weitere Argumente für JESS hervorzuheben. Damit kann das im vorigen Kapitel genannte Ziel der Editierbarkeit und Erweiterung der Wissensbasis zur

Laufzeit ohne großen Aufwand realisiert werden. Eine nähere Beschreibung zu Jess ist im nächsten Unterkapitel zu finden.

4.4 Java Expert System Shell (JESS)

Nachdem die Wahl nun auf Jess als Konfigurationstool gefallen ist, werden die Möglichkeiten und Funktionsweise dieses Shells im vorliegenden Abschnitt näher beleuchtet. In Unterkapitel 4.4.1 werden zunächst kurz die für diese Arbeit signifikanten Möglichkeiten des Tools aufgeführt. Anschließend folgt eine zusammenfassende Erklärung des Rete-Algorithmus, dem Kern von Jess. Für eine ausführliche Beschreibung der Jess-Sprachelemente sei auf [5] verwiesen.

4.4.1 Generelle Beschreibung

Jess ist eine *rule engine* and Scripting-Umgebung für Java zugleich, und wurde von Ernest Friedman-Hill im Jahre 1995 als „Übersetzung“ von CLisp komplett in Java geschrieben.
Bei der Anbindung von Jess an eine Java-Applikation werden die Java-Objekte auf sog. *shadow facts* abgebildet. Entscheidend bei dieser Abbildung ist, dass die verwendeten Java-Klassen stets als sog. *beans* vorliegen. Die shadow facts beinhalten neben dem Namen sog. *slots*, welche den Membervariablen der Java beans entsprechen und dynamisch über die Instanziierung eines `PropertyChangeListener` an Änderungen in den beans angepasst werden können.
Regeln haben in Jess generell die folgende Form:

(defrule Regelbezeichnung
Prämissen
=>
Aktionen
)

Darin können sowohl auf der LHS als auch auf der RHS lokale Variablen deklariert werden. Lokalen Variablen auf der LHS können zum Beispiel ein Faktum oder ein slot zugewiesen werden. Neben der Definition von lokalen Variablen gibt es zusätzlich die Möglichkeit von globalen Variablen, die zum Beispiel für Daten benutzt werden können, welche aus Java-Funktionen gesetzt werden sollen und in mehreren Regeln benötigt werden. Generell werden Variablentypen in Jess automatisch nach dem ersten zugewiesenen Wert bestimmt und müssen nicht angegeben werden. Daher ist der Gebrauch von Regeln und Fakten in Form von Templates möglich, wenn erst zur Laufzeit die Typen der Variablen bzw. slot-Werte feststehen. Darüber hinaus kann jeder Variable der Wert `nil`, das Pendant zu `null` in Java, zugewiesen werden.
Im Bereich des Kontrollwissens gibt es die Möglichkeit der Modulbildung und Fokussierung, sowie das Zuweisen von Prioritäten. Auf die Nutzung von Prioritäten wird

an dieser Stelle nicht weiter eingegangen, da von ihrem Gebrauch weitgehend abgeraten wird [5] und sie in dieser Arbeit nicht verwendet werden. Die Modulbildung erfolgt mit dem `defmodule`-Befehl, das Umschalten zu anderen vorhanden Modulen mit `set-current-module`. Alle anschließend definierten Regeln und Fakten gehören zum zuletzt gesetzten oder definfierten Modul, außer diese werden explizit einem anderen Namensraum mit einem vorangestellten „<Modulname>::“ zugewiesen [5]. Das initiale Modul ist immer das `MAIN`-Modul. Module definieren nicht nur Namensräume, sondern spielen bei dem Abfeuern der Regeln eine bestimmende Rolle: Es werden immer nur Regeln des fokussierten Moduls abgefeuert. Am Anfang jedes Abfeuerns hat immer das `MAIN`-Modul den Fokus. Das Zuweisen des Fokus an andere Module erfolgt mit dem `focus`-Befehl. Regeln, die Fakten aus verschiedenen Modulen auf ihrer LHS als Prämissen haben, können mit der `auto-focus`-Anweisung ihr Modul automatisch fokussieren und abgefeurt werden, wenn ihre Prämissen erfüllt sind. Das Abfeuern der Regeln geschieht mit der `run`-Anweisung. Wie bereits erwähnt, können sämtliche Anweisungen von JAVA aus aufgerufen werden.

4.4.2 Der Rete-Algorithmus

Die Hauptaufgabe eines regelbasierten Systems ist das Vergleichen (*engl.* matching) der LHS der Regeln mit den Fakten in der working memory, um zu entscheiden welche Regeln aktiviert und abgefeuert werden sollen. Eine zweifellos ineffiziente Vorgehensweise wäre das Iterieren über alle Regeln und das Vergleichen mit allen Fakten in jedem Iterationsschritt.

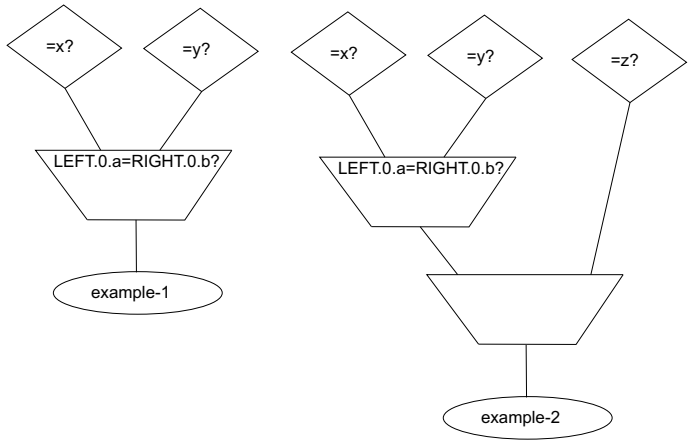

Bild 4.7: Rete-Netzwerk zu den Regeln `example-1` und `example-2` [5]

Der Rete-Algorithmus ist eine Methode zum effizienten Vergleichen einer Menge von Mustern (*engl.* patterns) mit einer Menge von Objekten um anschließend alle möglichen Treffer zu bestimmen [4]. Der Algorithmus ist sogar effizient, wenn große Mengen an Mustern und Objekten vorliegen, da es nicht über die Mengen iteriert. Es ist empirisch bewiesen, dass der größte Anteil von Fakten im working memory über eine bestimmte Zeiteinheit gleich bleibt [5]. Der Rete-Algorithmus merkt sich Testergebnisse über einzelne Durchläufe hinaus und prüft nur veränderte Elemente in jedem Schritt. Hinzu

kommt, dass Rete den pattern matcher so organisiert, dass die verbleibenden Fakten nur mit den Regeln verglichen werden, deren LHS eine Übereinstimmung liefern könnten.
Der Rete-Algorithmus erreicht dies, indem er ein Netzwerk von *Knoten* aufbaut. Jeder Knoten repräsentiert einen oder mehrere Vergleiche, die auf der LHS einer Regel gefunden werden. Ein Knoten hat einen oder mehrere Inputs und beliebig viele Outputs. Nur Fakten, die in der working memory eine Veränderung zum vorigen Zustand darstellen, werden beachtet. Die *Input-Knoten* befinden sich oben im Netzwerk, während die *Output-Knoten* ganz unten vorliegen [5]. Diese Knoten bilden zusammen das *Rete-Netzwerk*. Zwischen den Inputs und den Outputs kommen zwei Arten von Knoten vor: *Ein-Input-Knoten* und *Zwei-Input-Knoten* [5]. Ein-Input-Knoten führen an einzelnen Fakten Vergleiche durch, während Zwei-Input-Knoten Tests über mehrere Fakten vollziehen.

Als Beispiel ist in Abb. 4.7 das Netzwerk zu den folgenden Regeln dargestellt [5]:

```
(defrule example-1
      (x (a ?v1))
      (y (b ?v1))
      =>
)
(defrule example-2
      (x (a ?v2))
      (y (b ?v2))
      =>
)
```

Zur weiteren Optimierung eliminiert der Rete-Algorithmus redundante Ein- und Zwei-Input-Knoten indem diese von anderen Knoten gemeinsam benutzt werden. Dieser Optimierungsschritt ist in Abb. 4.8 veranschaulicht.

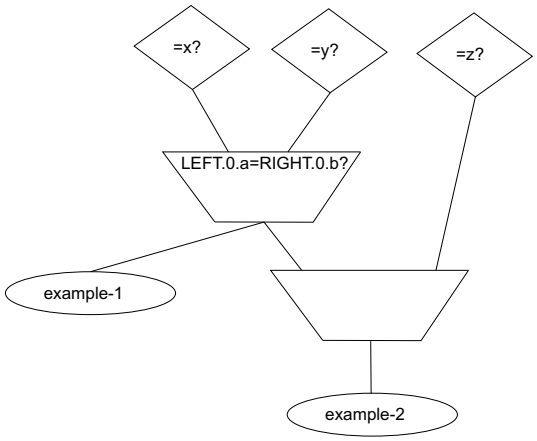

Bild 4.8: Optimiertes Rete-Netzwerk zu den Regeln `example-1` und `example-2` [5]

5 Realisierung des Programms

Der erste Schritt bei der Entwicklung eines regelbasierten Systems ist immer das Zusammentragen des benötigten Expertenwissens (*engl.* knowledge engineering). Im vorliegenden Fall ist die Wissensakquisition einerseits aus der Arbeit von Roye [12], andererseits aus Gesprächen mit Experten erfolgt. Die Ergebnisse des knowledge engineering wurden in Kapitel 2 bereits vorgestellt. Für das weitere Vorgehen bei der Erstellung eines regelbasierten Systems sind folgende generelle Schritte zu bearbeiten [5]:

- *Datenstrukturen bilden*
  Benötigte Variablen und ihre Charakteristika sollten ermittelt werden.

- *Interface erstellen*
  Das regelbasierte System sollte mit seiner Anwendungsumgebung verbunden werden. Im vorliegenden Fall ist dies die vorhandene JAVA-Applikation.

- *Regeln schreiben*
  Das Schreiben der Regeln sollte einer Struktur unterliegen, da die Regelbasis sehr leicht unübersichtlich werden kann. Dies hängt vor allem mit dem genannten Nachteil der Vermischung von Domänenwissen und Kontrollwissen zusammen (s. Kap. 3.2.1).

- *Iteratives Entwickeln*
  Regelbasierte Systeme werden gewöhnlicherweise iterativ entwickelt. Das heißt, man schreibt einige Regeln, testet diese, und dann fügt man weitere Regeln hinzu. Im Rahmen dieses Vorgangs werden immer wieder alle Schritte, begonnen beim knowledge engineering, durchlaufen.

Im Unterkapitel 5.1 werden die benötigten Datenstrukturen, sowohl für Java, als auch für Jess, untersucht. Im anschließenden Kapitel folgen Überlegungen und Konzepte zum Aufbau der Regelbasis. In Kapitel 5.3 wird der Aufbau des Java-Programms in Verbindung mit dem regelbasierten System hinterleuchtet. Schließlich werden im letzten Teil die bekannten Vor- und Nachteile von regelbasierten Systemen in Bezug auf den erstellten Prototypen untersucht sowie zusätzliche beim vorliegenden Anwendungsfall beobachtete Stärken und Schwächen geschildert.

5.1 Die Datenstruktur

Generell sind heuristische Entscheidungen in der Vorentwicklungsphase nicht zu vermeiden und notwendig aufgrund des Nichtvorhandenseins von Wissen. Solche Entscheidungen implizieren unter anderem folgende Gesichtspunkte [7]:

- Das Finden einer Lösung kann nicht garantiert werden, das einer optimalen Lösung erst recht nicht.

- Frühe Entscheidungen stellen sich zu einem späteren Zeitpunkt des Entwicklungsprozesses als nachteilhaft oder gar inkompatibel mit den restlichen Einschränkungen des System heraus.

Der zuletzt genannte Aspekt wird als *Horizonteffekt* bezeichnet und gilt u. a. als einer der Forschungsschwerpunkte im Bereich von Schachcomputern [14]. Als einfachster Lösungsvorschlag wird dazu häufig die Methode des *partial commitment* erwähnt, also dem Festlegen von Variablen in der „richtigen“ Reihenfolge [14]. Darin werden zunächst nur Variablen festgelegt, die sicher alle Anforderungen erfüllen, während andere Variablen offen gelassen werden. Im Laufe des Entwicklungsprozesses werden diese nach und nach iterativ festgelegt. Diese Methode erweist sich in der Praxis jedoch als äußerst schwierig, sobald die Anzahl an Variablen und Abhängigkeiten zunimmt. Zusätzliche Mechanismen für die weitere Arbeit sind daher nötig. Unabhängig von der Domäne gibt es zwei Methoden zur Lösung von Konflikten [7]:

- *Revision von Entschiedungen*
  Die Überarbeitung von Konfigurationsschritten ist auch als sog. *backtracking* bekannt. Dabei werden eine oder mehrere getroffene Entscheidungen zurückgenommen und erneut durchgeführt mit einem anderen Wert.

- *Reparieren von partiellen Lösungen*
  Anstatt Entscheidungen zu überarbeiten, ist es manchmal möglich partielle Lösungen zu „reparieren“Dies kann mit der Modifikation oder Hinzunahme zusätzlicher Domänenobjekte vorgenommen werden, um so die vorliegende partielle Lösung wieder in einen konsistenten Zustand zu überführen.

Für die Datenstrukturen im zu entwickelnden JAVA-Programm bedeuten die oben aufgeführten Methoden, dass alle Variablen jederzeit neben ihrer Veränderbarkeit auch löschbar, also auf `null` zurücksetzbar, sein müsssen. Da sämtliche Parameter im Auslegungsprozess als *floating number* vorliegen, ist es für diese daher sinnvoll, Objekte vom Typ `Float` zu erstellen, anstatt den primitiven Datentyp `float` zu verwenden.
In JESS stellt dies kein Problem dar, da Typen automatisch zugeteilt werden und Variablen immer auf `nil` zurücksetzbar sind (s. Kap. 4.4.1). Die in JAVA erstellten Variablen werden dazu bei der Initialisierung der Regelbasis auf die entsprechenden slots der shadow facts abgebildet und erhalten dabei die jeweiligen Typen.

Für die Abspeicherung eines Parameters im JAVA-Programm bedarf es neben dem Variablennamen und aktuellen Wert zusätzlich weiterer Eigenschaften in der Klasse `Variable`, die im Folgenden genannt werden. Eine visuelle Darstellung des Klassen-Interfaces von `Variable` ist in Abb. 5.1 zu sehen.

- Bei einigen Parametern sollen default-Werte vorhanden sein und angezeigt werden.

- Da als Anfordrung das automatische Ablehnen bzw. Zurücksetzen von unzulässigen Werten aufgeführt wurde (s. Kap. 4.2), ist eine weitere Variable zur Speicherung des alten Wertes einzuführen (s. dazu auch Kap. 5.2).

- Aus der Analyse des Problemfalls in Kap. 2 geht weiterhin hervor, dass zwischen Variablen zu unterscheiden ist, bei denen der Benutzer Werte eingeben soll, und solchen deren Wert grundsätzlich (von der rule engine) berechnet wird. Für diese Unterscheidung wird die `boolean`-Variable `toBeCalculated` eingeführt.

- Das „Festhalten“ oder die Unveränderbarkeit von Parameterwerten wird durch die `boolean`-Variable `editable` bewerkstelligt.

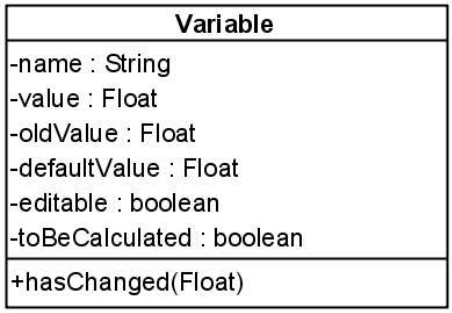

Bild 5.1: Das Interface der Klasse `Variable`

Neben den üblichen *getter*- und *setter*-Methoden der oben genannten Membervariablen von `Variable` wird die Methode `hasChanged()` benötigt, um beim Einlesen von eingegebenen Daten einen Vergleich mit den alten Werten druchzuführen, damit nur geänderte Werte übenommen werden.

5.2 Die Regelbasis

Um den bekannten Problemen von regelbasierten Systemen (s. Kap 3.2.1) möglichst entgegenzuwirken, sollten die Regeln nach mehreren Gesichtspunkten strukturiert werden. Auf der einen Seite ist zu analysieren, was für *Arten* von Regeln benötigt werden, und eine Unterteilung nach diesem Aspekt sollte erstellt werden. Auf der anderen Seite sollte eine sinnvolle *Modularisierung* der Regeln entsprechend der Domänenstruktur erfolgen. Diese beiden Betrachtungspunkte sind Gegenstände der folgenden Unterkapitel.

5.2.1 Arten von Regeln

Aus der Analyse der Zusammenhänge zwischen den Variablen des vorliegenden Problemfalls (s. Kap. 2) wird deutlich, dass zwei generelle Arten von Regeln benötigt werden. In diesem Zusammenhang wird zur weiteren Untersuchung der Begriff der *Constraints* wieder verwendet (s. Kap. 3.2.3), da die folgende Unterteilung prinzipiell auf einer abstrakteren Ebene als die der Regeln stattfindet. Die unterschiedlichen Constrainttypen werden in der Regelbasis jeweils durch einen unterschiedlichen Aufbau der dazugehörigen Regeln reflektiert. Die komplette Regelbasis des erstellten Prototyps

kann im Anhang eingesehen werden. [4]
Folgende Regeltypen werden im System für unterschiedliche Constraintarten erstellt [9]:

Regeln für Funktionsconstraints

Funktionsconstraints beschreiben die Abhängigkeiten zwischen den Variablen, die durch einen Formelausdruck (z. B. A + B = C) dargestellt werden können [9]. Sie werden zur Berechnung von Variablenwerten verwendet, welche von anderen Variablenwerten abhängig sind, bzw. Funktionen anderer Variablenwerte sind. Da es sich bei dieser Constraintart um reine Vorwärtsverkettung handelt, kann sie als gerichteter Graph dargestellt werden [2], welches in Abb. 5.2 an einem Beispiel aus dem erstellten Prototypen visualisiert wird.

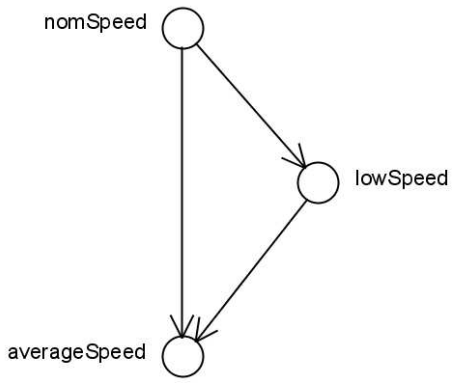

Bild 5.2: Beispiel eines gerichteten Graphen für Funktionsconstraints

Eine entsprechende Regel bezüglich einer der Kanten, bzw. der Berechnung einer der Variablen, kann folgendermaßen aussehen:

```
(defrule low-speed
      ?a <- (GlobalCalculations (nomSpeed ?n))
      =>
      (if (neq ?n nil) then
            (modify ?a (lowSpeed (* ?n 0.18)))
      else
            (modify ?a (lowSpeed nil))
      )
)
```

Die dargestellte Regel wird zur Berechnung der Variable `lowSpeed` aus dem Wert von `nomSpeed` verwendet. Zunächst wird auf der LHS das Vorliegen des shadow facts

[4] Es sei darauf hingewiesen, dass es generell mehrere Möglichkeiten zur Realisierung von Constraints in Regeln gibt. Die vorgestellten Vorgehensweisen stellen nur einige dieser Möglichkeiten dar.

`GlobalCalculations` mit der benötigten Variable `nomSpeed` in einem seiner Slots geprüft. Bei Erfüllung erfolgt auf der RHS als Aktion die entsprechende Berechnung und Zuweisung des Variablenwertes. Aufgrund der Anforderungen an die Datenstruktur (s. Kap. 5.1) findet eine zusätzliche Prüfung auf den Wert `nil` statt, damit bei Löschung des benötigten Variablenwertes der davon abhängige Variablenwert auf `nil` zurückgesetzt wird.
In der Regelbasis des betrachteten Falls kommen 44 solcher Regeln vor. Generell lassen sich Funktionsconstraints sehr gut durch das regelbasierte System realisieren.

Regeln für Prädikatsconstraints

Prädikatsconstraints beschreiben Formeln mit einfachen Prädikaten (z. B. $<, >, =$) für die Constraint-Variablen [9]. Diese entsprechen ungerichteten Graphen, da zur Erfüllung solcher Constraints die Änderung jeder beteiligten Variablen signifikant ist. Dieser Zusammenhang wird in Abb. 5.3 anhand eines Beispiels verdeutlicht.

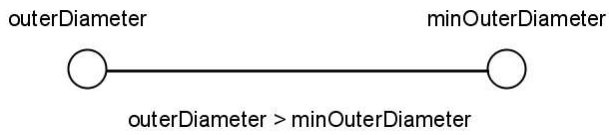

Bild 5.3: Beispiel einer ungerichteten Relation bei einem Prädikatsconstraint

Aus dem Beispiel ist ersichtlich, dass die Erfüllung des Constraints durch Änderungen an *beiden* beteiligten Variablen erreicht werden kann. Weiterhin sind zur Einhaltung von Prädikatsconstraints im vorliegenden Fall drei Arten von Konsistenzgraden zu beachten ([13], [2]):

- *Knotenkonsistenz*
  Der einfachste Konsistenzgrad ist die Knotenkonsistenz. Dabei werden für Variablenwerte bei Zuweisungen überprüft, ob die zulässigen Wertebereiche der jeweiligen Variable eingehalten werden. Dadurch werden die Wertebereiche der Variablen auf die erlaubten Grenzen beschränkt und ein Teil der unzulässigen Werte aussortiert.

- *Kantenkonsistenz (arc-consistency)*
  Die Kantenkonsistenz stellt einen höheren Konsistenzgrad als die Knotenkonsistenz dar. Sie bezieht sich auf die Domänen der Knoten, die durch eine Kante verbunden sind, also über ein Constraint zueinander in Relation stehen. Dabei wird sichergestellt, dass die Wertebereiche jeder Variablen nur auf solche Werte beschränkt werden, die kompatibel zu unmittelbar benachbarten Variablen sind.

- *Pfadkonsistenz (path-consistency)*
  Was passiert, wenn anstatt eines Variablenpaares eine Sequenz von constraintbehafteten Variablen vorliegt und für diese Konsistenz gewährleistet sein soll. In einem solchen Fall wird die Konsistenz für einen sog. *Pfad* benötigt. Zur Erfüllung von Pfadkonsistenz müssen alle Variablendomänen entlang des Pfades bezüglich aller Constraints im Pfad geprüft werden.

Das Matching-Diagramm [2] in Abb. 5.4 veranschaulicht das Prinzip der Kantenkonsistenz am oben genannten Beispiel aus Abb. 5.3. Darin sind für jede Variable exemplarsich drei Domänenwerte möglich. Da die Variable `outerDiameter` in jedem Fall größer sein soll als `minOuterDiameter`, stehen der kleinste Wert aus der Domäne von `outerDiameter` und der größte Wert aus dem Wertebereich von `minOuterDiameter` mit keinen anderen Wert in Beziehung. Diese Domänenelemente können daher ausgeschlossen werden.
Das Modell in diesem Beispiel kann zu einem Pfad ausgebaut werden, indem eine dritte Variable eingeführt wird, die jeweils mit den beiden erstgenannten Variablen in einer Relation steht.

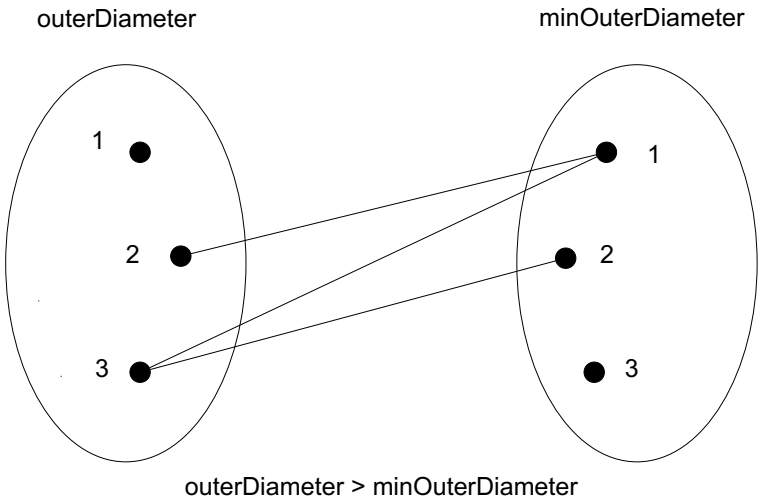

Bild 5.4: *Matching-Diagramm* zur Beschreibung von Kantenkonsistenz [2]

Zur Umsetzung von Prädikatsconstraints mit verschiedenen Konsistenzgraden in der Regelbasis werden weitere Typen von Regeln benötigt. In der Praxis erweist sich insbesondere eine vollständige Realisierung von Kanten- und Pfadkonsistenz mit Regeln als äußerst schwierig, jedoch können bestimmte Regelarten formuliert werden und Regeln miteinander so verknüpft werden, dass die oben genannten Anforderungen teilweise abgedeckt werden.

Das Hauptproblem bei der Abbildung von Prädikatsconstraints auf Regeln liegt darin, dass es mit Regeln nicht möglich ist von vornherein die Eingabe unzulässiger Werte zu *vermeiden*. Es müssen erst Werte angenommen werden, die Regeln abgefeuert werden und dann erst kann bestimmt werden, ob eine Constraintverletzung vorliegt oder nicht. Bezogen auf das Beispiel in Abb. 5.4 bedeutet dies, dass in beiden dargestellten Domänen die ausgeschlossenen Werte vom Benutzer eingegeben werden können. Erst nach Abfeuern der Regeln kann dann eine entsprechende Reaktion auf die zunächst zugelassene Inkonsistenz erfolgen. Kurz gesagt, lassen sich Intervalleinschränkungen nicht auf direkte Weise realisieren.

Eine mögliche (Teil-)Lösung für dieses Problem stellt jedoch bei einer Constraintver-

letzung das automatische Zurücksetzen der eingegebenen Variable auf ihren vorherigen Wert und das anschließend erneute Abfeuern der Regeln dar. Dadurch werden in einer Kettenreaktion von Regelaktivierungen sowohl Werte, die über Kantenkonsistenz direkt mit der veränderten Variable in Verbindung stehen zurückgesetzt, als auch weiter entfernte Variablenwerte entlang eines Pfades. Aus der Anwendersicht sieht das Ergebnis so aus, als wäre der unzulässige Wert nicht angenommen worden, und zwar bezogen auf allen Constraints entlang des Pfades, zu welchem die veränderte Variable gehört. Da der Benutzer immer nur eine Variable zur Zeit ändern kann, stellt das Abspeichern des alten Wertes der jeweils geänderten Variable kein Problem dar. Beim Durchlaufen der Regeln kann jede „Prädikatsconstraint-Regel“ beim Abfeuern eine Nachricht ausgeben, dass das entsprechende Constraint verletzt worden ist. Damit kann der Benutzer nach dem automatischen Zurücksetzen des eingegebenen Wertes eine Liste der Constraints erhalten, die durch seine Eingabe verletzt wurden. Zusätzlich kann jede Prädikatsconstraint-Regel seinen maximal bzw. minimal zulässigen Wert mit ausgeben, da dieser in der Prüfungsbedingung auf seiner LHS vorkommt.

In der erstellten Regelbasis des Prototypen sind zur Übersicht die Bezeichnungen für Prädikatsconstraint-Regeln mit der Endung „kriterium“ erweitert worden. Im Folgenden werden zwei vereinfachte Beispielregeln vorgestellt. Die erste dient zur Prüfung von Knotenkonsistenz und die zweite zur Prüfung von Kantenkonsistenz:

```
(defrule detection-time-kriterium
        (BlowbackCalc (detectionTime ?d))
        (test (< ?d 0))
        =>
        (warningList “detectionTime minimum limit violated!” ?d 0 ?*sourceVar*)
        (assert (undo-source))
)

(defrule shaft-min-aussendurchmesser-kriterium
        (Shaft (outerDiameter ?D) (minOuterDiameter ?mD))
        (test (< ?D ?mD))
        =>
        (warningList “Minimum Outer Diameter exceeded!” ?D ?mD ?*sourceVar*)
        (assert (undo-source))
)
```

Wie aus dem Beispiel zu entnehmen ist, ist der Aufbau der beiden Regelarten nahezu identisch. Der Hauptunterschied liegt auf der LHS beim Testen des jeweiligen Constraints. Bei der ersten Regel wird der Variablenwert von `?d` gegen die Konstante `0` verglichen. Dagegen findet bei der zweiten Regel ein Vergleich zwischen den zwei Variablenwerten von `?D` und `?mD` statt. Falls das jeweilige Constraint verletzt ist, wird anschließend die entsprechende RHS ausgeführt. Bei beiden Regeln wird dazu zunächst die JAVA-Funktion `warningList` ausgeführt (s. Kap. 5.3), welche eine Nachricht über den verletzten Constraint, sowie die beteiligten Variablen und Grenzwerte für das Constraint als Parameter erhält. Anschließend wird das Faktum `undo-source` in die working

memory von JESS geschrieben, welche die Regel zum Zurücksetzen der eingegebenen Variable aktiviert. Die wesentlichen Elemente dieser Regel sind folgendermaßen aufgebaut:

```
(defrule undo-source
      ?c <- (undo-source)
      ?s <- (or (GeneralPart (index ?i))(Calculation (index ?i)))
      (test (eq ?i ?*sourcePart*))
      =>
      modify ?s (?*sourceVar* ?*sourceOldVal*))
      (retract ?c)
)
```

Zum Zurücksetzen des eingegebenen Wertes werden drei Informationen benötigt: Die Klasse, in der die entsprechende Variable gespeichert ist, der Variablenname und der alte Wert der Variable vor der Eingabe des Benutzers. Diese Daten werden in den *globalen* Variablen `?*sourcePart*`, `?*sourceVar*` und `?*sourceOldVal*` verwaltet, da sie vom JAVA-Programm aus ihre Werte erhalten (vgl. Kap. 4.4.1). Zunächst wird auf der LHS geprüft, ob das Faktum `undo-source` gesetzt worden ist. Danach wird der Index der Komponente bzw. der Klasse für globale Berechnungen (s. Kap. 5.3), welche die eingegebene Variable als Member besitzt, ermittelt. Auf der RHS wird als Aktion die veränderte Variable wieder auf ihren ursprünglichen Wert gesetzt und das Faktum `undo-source` aus der working memory entfernt. Das erneute Abfeuern der Regeln zum Zurücksetzen der abhängigen Werte im verbundenen Pfad erfolgt anschließend aus dem JAVA-Programm (s. Kap. 5.3).

Regeln zum Management der working memory

Neben den Regeln für Funktions- und Prädikatsconstraints ist eine dritte Art von Regel erforderlich, die leicht übersehen werden kann: Regeln zum Entfernen nicht gebrauchter shadow facts.
Da JESS über keinen *garbage collector* verfügt, werden zwar slot-Werte der shadow facts an die entsprechenden Member der JAVA-Objekte dynamisch angepasst, jedoch bleiben die Fakten nach der Vernichtung der entsprechenden JAVA-Objekte weiterhin in der working memory bestehen und deren Löschung muss explizit aufgerufen werden. Somit wird die working memory zunehmend mit nicht benötigten Fakten gefüllt, worunter nach einer bestimmten Zeit die Ausführungszeiten und die Übersicht leiden können. Zur Entfernung von nicht benötigten Fakten können wiederum Regeln formuliert werden, die aktiviert werden und diese entfernen sobald ihr Gebrauch beendet ist. Dies geschieht entweder, wenn im JAVA-Programm ein Bauteil-Objekt explizit entfernt wird, oder wenn in einer Regel ein dort dynamisch erzeugtes JAVA-Objekt nur lokal verwendet wird und danach überflüssig ist.

5.2.2 Gliederung der Regelbasis

Das *Kontrollwissen* stellt die Informationen zur Steuerung des Konfigurationsverlaufes dar [9]. Eine Möglichkeit zur Ablaufskontrolle besteht in der Modularisierung der

Regelbasis. Die Unterteilung der Regelbasis in Module orientiert sich in erster Linie an der Struktur des Domänenwissens. In Kapitel 2 wurden dazu die einzelnen Schritte des Entwicklungsprozess der Antriebssysteme von Hochauftriebssystemen erläutert. In der Praxis führt jedoch die direkte Modulbildung entsprechend der Auslegungsschritte zu Problemen, da intermodulare Abhängigkeiten zwischen den Schritten existieren und berücksichtigt werden müssen. Da immer nur ein Modul zur Zeit fokussiert sein kann, müssen die Abhängigkeiten zwischen Variablen aus verschiedenen Schritten beachtet werden. Die benötigten Regeln aus anderen Schritten würden schlichtweg nicht abgefeuert werden, da deren Modul nicht fokussiert wäre.

Es würde zweifellos einen großen Zusatzaufwand bedeuten, für jede Regel und jedes Faktum den Namensraum mit der Operation „::“ festzulegen. Dies sollte nur bei dynamisch erstellten Fakten, die in unterschiedlichen Modulen zum Einsatz kommen, Verwendung finden.
Eine Lösung wäre, eine etwas gröbere Unterteilung zu wählen und Module mehrere voneinander abhängige Schritte beinhalten zu lassen. Schritte, die andererseits weitgehend abgekapselt vom restlichen Programm durchlaufen werden können, sollten ein eigenes Modul besitzen. Weiterhin sollte bedacht werden, welche Regeln bewusst zu einem frühen Zeitpunkt der Auslegung nicht beachtet werden sollten, da sie anfangs noch nicht erfüllt sein können. Durch Modularisierung kann erreicht werden, dass diese Regeln erst in einer späteren Phase abgefeurt werden. Eine zweite Lösung stellt der JESS-Befehl `auto-focus` für Regeln dar (s. Kap. 4.4.1). Damit können Regeln, die sich auf Objekte unterschiedlicher Module beziehen „modulübergreifend“ gemacht werden und abgefeuert werden, auch wenn der Fokus bei einem anderen Modul liegt. Eine intensive Nutzung dieses Befehls in einem Modul sollte jedoch vermieden werden, da sie das Modul an sich in Frage stellen würde: Wenn die meisten Regeln in einem Modul modulübergreifend sind, dann scheint dessen Existenz nicht gerechtfertigt.

Aus den obigen Überlegungen geht hervor, dass neben der Struktur des Domänenwissens vor allem die Einschätzung des Entwicklers eine Rolle spielt, an welchen Stellen und wie fein Module gebildet werden sollten. Als Grundlage des benötigten Einschätzungsvermögens sollte jedoch vorher eine detaillierte Analyse der Abhängigkeiten zwischen den Variablen und erstellten Regeln durchgeführt werden. Da eine solche Untersuchung mit der Größe der Regelbasis zunehmend erschwert, zeitaufwendig und unübersichtlich wird, spiegelt sich die Unangemessenheit von regelbasierten Systemen bei komplexeren Problemen auch an dieser Stelle wider.

Mit insgesamt 60 Regeln und der gegebenen Struktur der Domäne mit zwölf Auslegungsschritten ist jedoch beim betrachteten Fall eine sinnvolle Unterteilung der Regelbasis möglich. Nach der Analyse der Abhängigkeiten scheint eine Zerlegung der Regelbasis in vier Modulen sinnvoll: Zunächst findet eine Trennung der Auslegungsschritte für Bauteile von den globalen Schritten statt, da nur vereinzelt Abhängigkeiten zwischen diesen Teilen bestehen. Die Schritte für die Auslegung von Bauteilen bekommen das `MAIN`-Modul zugewiesen, da diese Schritte den Anfang der Auslegung darstellen und `MAIN` das initiale Modul ist. Zusätzlich wird der Schritt zur Auslegung der Drehwellen mit einem eigenen Modul `SHAFT` versehen, da dieser Schritt weitgehend abgekapselt von den restlichen Teilen stattfindet. Die globalen Auslegungsschritte werden im Modul `GLOBAL` durchgeführt, wobei die beiden Schritte zur Drehzahl- und Blowback-

Untersuchung ein eigenes Modul `ROTBLOW` erhalten, da zwischen diesen Teilen viele Abhängigkeiten festzustellen sind.

In der Praxis hat sich erwiesen, dass eine weitere Unterteilung der vorliegenden Regelbasis in zusätzliche Module nachträglichen Aufwand sowie unnötige Abhängigkeiten zwischen zu fein gegliederten Modulen mit sich bringen würde.

5.3 Das Java-Programm

5.3.1 Funktionale Beschreibung

Als Idee für ein kompaktes, programmgeführtes Durchlaufen der Auslegungsschritte hat sich eine Art *Wizard* ergeben. Mit Hilfe des Wizards kann der User durch die Auslegungsschritte geführt werden wobei eine übersichtliche Darstellung angepasst an die jeweilige Auslegungsphase möglich ist. Nach dem Starten des Wizards hat der Benutzer zunächst ein Layout zu erstellen. Danach wird er durch die Schritte zur Auslegung signifikanter Bauteile geführt (vgl. Kap. 2.3). Die Parameter *aller* betrachteten Bauteile können je nach Auslegungsphase zusammen in übersichtlicher Weise (z. B. tabellarisch) dargestellt und modifiziert werden. Eine Beispielsituation dazu ist in Abb. 5.5 dargestellt. Anschließend werden die Schritte zur globalen Auslegung durchlaufen.
Bei jeder Eingabe eines Wertes durchläuft die rule engine im Hintergrund wie im vorigen Kapitel beschrieben die Regeln, prüft verschiedene Abhängigkeiten und setzt die neuen Werte oder bei einer Constraintverletzung die veränderte Variable zurück.

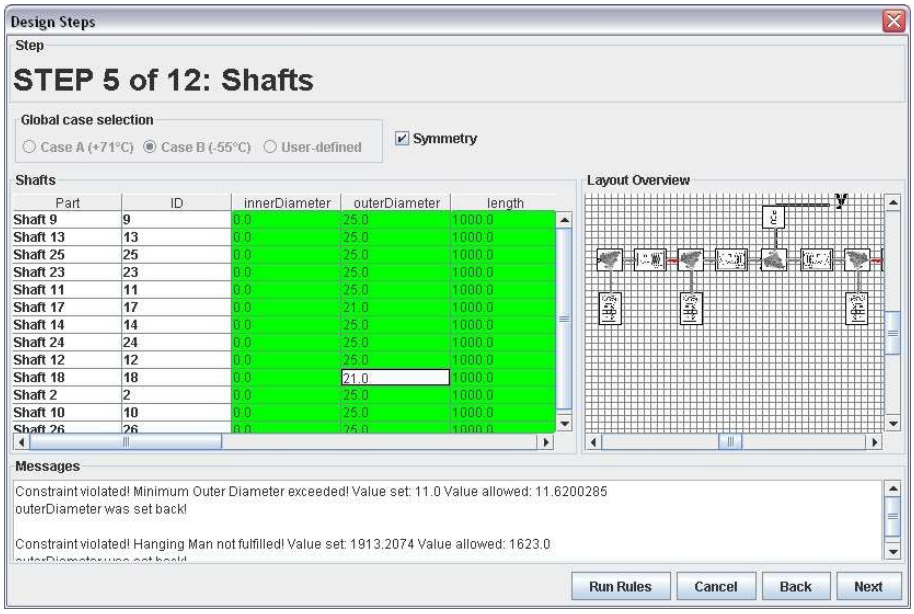

Bild 5.5: Beispielsituation zu der Benutzung des Auslegungswizards

Das Konfigurieren von *einzelnen* Bauteilattributen war im Grundprogramm bereits gegeben (s. Kap. 4.1.1) und musste lediglich um den zusätzlichen Zugriff auf die globalen Parameter erweitert werden.

5.3.2 Neue Programmteile

Zur Einbindung von JESS in die JAVA-Anwendung muss eine Instanz der `Rete`-Klasse erzeugt werden und die working memory mit der Importierung nötiger JAVA-Klassen zur Bildung von shadow facts initialisiert werden. All dies geschieht in der `Admin`-Klasse (s. Kap. 4.1.2), welche zu diesem Zweck einmalig beim Programmstart die Methode `jessRegistration()` aufruft. Das generelle Interface von `Admin` ist in Abb. 5.6 graphisch dargestellt.

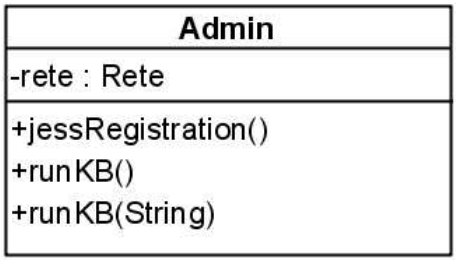

Bild 5.6: Das Interface der Klasse `Admin`

Zum Abfeuern der Regeln wird die Methode `runKB()` eingeführt, welche immer aufgerufen wird, wenn der Benutzer eine Eingabe macht. Zusätzlich wird diese Methode mit einer zweiten Version überladen, bei der ein String mit übergeben werden kann. In dem String-Parameter kann ein JESS-Befehl enthalten sein, welcher vor dem Abfeuern der Regeln evaluiert wird. Diese zweite Version findet Gebrauch, wenn zum Beispiel globale Variaben in JESS geändert werden sollen vor dem Abfeuern der Regeln (vgl. Kap. 5.2.1).

Weiterhin werden Klassen zum Verwalten der Parameter der globalen Auslegungsschritte benötigt. Für jede Phase der Auslegung wird dazu eine Klasse eingeführt, welche die spezifischen Variablen der jeweiligen Phase beinhaltet. Diese sind vom Prinzip her wie die Klassen für die Bauteile aufgebaut (s. Abb. 4.2). Gemeinsame Teile der Klassen werden in einer Oberklasse `Calculation` verwaltet. In dem Klassendiagramm in Abb. 5.7 wird dieser Zusammenhang verdeutlicht.

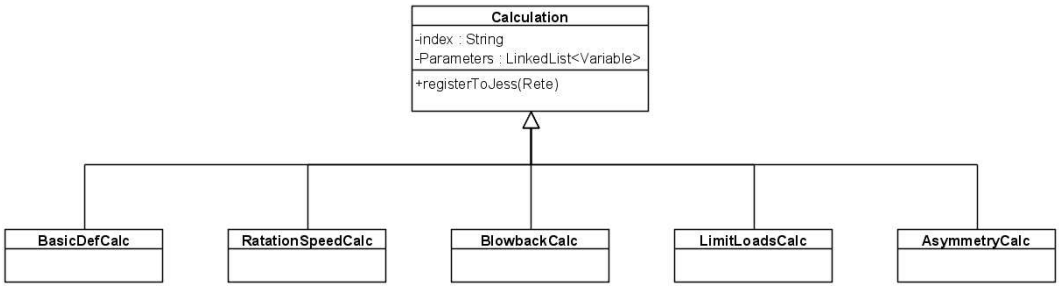

Bild 5.7: Klassenhierarchie der Klassen zur globalen Auslegung

Die Klasse `Calculation` entspricht hierbei der Oberklasse `GeneralCalculations` für

Bauteile, wobei im Unterschied zu dieser graphische Komponeneten nicht benötigt werden.

5.4 Vor- und Nachteile des regelbasierten Systems

Als Abschluss zu diesem Kapitel werden im Folgenden noch einmal die Vor- und Nachteile der regelbasierten Methode bezogen auf den untersuchten Problemfall aufgeführt.

In Kapitel 3 wurden bereits generelle Stärken und Schwächen der regelbasierten Konfigurierung gegenübergestellt, auf Basis derer später eine Auswahl getroffen wurde.
Die bekannten Vorteile bei der Vorwärtsverkettung und der Darstellung heuristischer Zusammenhänge haben sich auch im erstellten Prototypen bewahrheitet. Mit insgesamt 44 Regeln für reine Vorwärtsverkettung zeigte das regelbasierte System sehr schnelle Ausführungszeiten und kann sicherlich in diesem Bereich um viele weitere Regeln problemlos erweitert werden. Auch die einfache Anknüpfung an JAVA und die Interaktion mit JAVA-Klassen hat sich an mehreren Stellen als Vorteil erwiesen.

Auf der anderen Seite sind trotz der relativ geringen Anzahl an Regeln (60 insgesamt) einige Nachteile stärker in Erscheinung getreten als erwartet. Dies lag vor allem daran, dass bei nur zehn Prädikatsconstraints schon relativ hohe Konsistenzgrade nötig wurden, welche einzig mit Regeln schwer realisierbar sind. Trotz der Möglichkeit des Zurücksetzens inkonsistenter Werte, muss der Benutzer selbst Werte durchlaufen bis er eine Lösung findet. Festzuhalten bleibt daher, dass nur die reine Anzahl an Regeln bzw. Constraints nicht aussagekräftig genug ist, sondern auch die benötigten Konsistenzgrade berücksichtigt werden sollten. Eine übersichtliche Auflistung der Nachteile wird im Folgenden gegeben:

- Das Umformulieren von Constraints in Rules führt dazu, dass unzulässige Eingaben vom Benutzer erst ausprobiert und zurückgesetzt werden müssen. „Echte“ Kanten- und Pfadkonsistenz sind daher nicht gegeben. Der Benutzer selbst muss Möglichkeiten durchlaufen.

- Durch das heuristische Vorgehen kann ggf. Unerfüllbarkeit von vornherein nicht ermittelt werden (wenn es keine Lösung bei vorliegenden Werten gibt). Es existiert auch keine Garantie auf das erfolgreiche Finden einer Lösung.

- Es besteht die Gefahr in einem lokalen Optimum „hängen“ zu bleiben. Eine Optimierung kann schwer durchgeführt werden.

- Zyklische Abhängigkeiten können nicht abgefangen werden und führen zu Endlosschleifen. Das liegt daran, dass keine Vergangenheitsinformationen über einzelne Durchläufe hinaus abspeichert werden. Die Regeln werden immer wieder aktiviert beim Setzen von entsprechenden Fakten.
  Da im betrachteten Fall Zyklen auftraten, musste ein Teil der Regeln aus der Regelbasis entfernt werden.

- Die Modularisierung der Regelbasis ist nur bei einer relativ kleinen Anzahl von Regeln sinnvoll, bzw. für Regeln, welche wenige gegenseitige Abhängigkeiten auf-

weisen, da eine Untersuchung der Abhängigkeiten zur Modulbildung unverzichtbar ist. Bei komplexeren Systemen würde die Modularisierung mehr Zusatzaufwand als Übersicht mit sich bringen.
Für die Dimension des erstellten Prototypen brachte die Modulbildung Vorteile, jedoch würde bei einem Ausbau der Regelbasis die Modulstuktur wieder verändert werden müssen, welches in dem Fall als großer Nachteil zu bewerten wäre. Daher lässt sich weiterhin behaupten, dass die Modulbildung die Struktur der Regelbasis *unflexibel* macht.

- Das visuelle Darstellen von Constraints zwischen Variablen kann nur auf sehr umständlicher Weise erreicht werden. Die mitgelieferte JESS-Funktion `view` stellt keine Hilfe dar, da sie lediglich die Knoten im Rete-Netzwerk anzeigt, und diese wie beschrieben von mehreren Variablen geteilt werden (s. Kap. 4.4.2).

Als Fazit kann festgehalten werden, dass vorwärtsverkettende, zykelfreie Abhängigkeiten zwischen Variablen mit regelbasierten Systemen sehr gut bewältigt werden können. Weiterhin sollte für diese Methode eine Wissensbasis mit relativ wenigen Abhängigkeiten verwendet werden, da sonst Probleme mit der Übersicht und der Wartbarkeit auftreten. Der Einsatz von Modulen ist in dem Fall dann auch nicht hilfreich.

Für die betrachtete Problemstellung können regelbasierte Systeme noch als akzeptabel beschrieben werden, da sich die Einschränkungen durch Unzulänglichkeiten dieser in Grenzen halten und die Vorteile des gewählten Vorgehens immer noch das größere Gewicht haben. Für eine höhere Flexibilität und Dynamik, sowie „echter“ Kanten- und Pfadkonsistenz sind constraintbasierte Konfigurierungssysteme empfehlenswerter. Eine Kombination anderer Systeme mit regelbasierten Methoden wäre außerdem denkbar.

6 Zusammenfassung

Das Ziel dieser Arbeit ist die Entwicklung eines Prototypen zur wissensbasierten Konfigurierung von High-Lift-Antriebssystemen in der Vorauslegungsphase und dessen Integration in eine bestehende JAVA-Applikation. Dazu werden zunächst im Rahmen des *knowledge engineering* der Aufbau von HL-Antriebssystemen sowie darin vorkommende Anforderungen und Abhängigkeiten aus der Arbeit von Roye [12] entnommen. Anschließend werden einige gängige Methoden der wissensbasierten Konfigurierung als Kandidaten für den zu erstellenden Prototypen vorgestellt und deren generelle Vor- und Nachteile diskutiert. Eine Analyse für das zu erstellende System legt unter Berücksichtigung des existierenden JAVA-Programms die Ziele und Anforderungen fest. In diesem Zusammenhang werden insbesondere Anforderungen an Ausführungszeiten, den Zugriff auf die Wissensbasis, die Konsistenz zwischen den Abhängigkeiten und die Modularisierung der Wissensbasis hervorgehoben.

Auf Basis der durchgeführten Analyse findet die Auswahl eines wissensbasierten Systems zur weiteren Entwicklung statt. Als erste Erkenntnis stellt sich heraus, dass die Methoden der *regelbasierten* und *constraintbasierten* Konfigurierung für das analysierte Problem scheinbar die besten Vorgehensweisen darstellen, und der regelbasierte Ansatz wird zur Prototypenerstellung gewählt. Als Begründung für die Wahl wird aus anwendungsorientierter Sicht das Argument hervorgebracht, dass die regelbasierte Methode für den betrachteten Fall die Lösung mit dem geringsten Aufwand darstelle. Der Problemfall wird hierbei mit insgesamt 60 Abhängigkeiten, davon 44 vorwärtsverkettend, als eher kleines Problem klassifiziert. Weiterhin ist die Eignung eines regelbasierten Systems für das betrachtete Problem zu untersuchen. Als Tool wird die Shell JESS gewählt, da dies eine einfache Integration in JAVA ermöglicht und mit dem Rete-Algorithmus zur Inferenz eine schnelle Option darstellt.

Der Prozess der Realisierung beginnt mit der Festlegung der benötigten Datenstrukturen, sowohl in JAVA als auch in JESS. Weiterhin müssen zum Aufbau der Regelbasis die Regeln nach Arten unterteilt werden, wobei der abstraktere Begriff der *Constraints* Verwendung findet. Es stellt sich heraus, dass drei Sorten von Regeln benötigt werden: Regeln für *Funktions-* und *Prädikatsconstraints* sowie Regeln zum Management der *working memory*. Zudem ist eine feinere Unterteilung von Prädikatsconstraints nach drei Konsistenzgraden nötig: *Knoten-*, *Kanten-* und *Pfadkonsistenz*. Da sich eine direkte Realisierung der Prädikatsconstraints mit einem höheren Grad als eins mit Regeln als schwierig herausstellt, wird eine teilweise Verwirklichung durch Annahme, Ausprobieren und Zurücksetzen von inkonsistenten Eingaben und erneutes Abfeuern der Regeln verwirklicht.

Abschließend findet zur Eignungsuntersuchung des regelbasierten Ansatzes für den vorliegenden Fall eine Analyse von beobachteten *Vor- und Nachteilen* statt. Wie erwartet bestätigt sich der Vorteil des schnellen und problemlosen Ausführens von vorwärtsverkettenden Funktionsconstraints. In diesem Punkt wäre sogar ein Ausbau der Regelbasis um eine Reihe weiterer vorwärtsverkettender Regeln denkbar.
Auf der anderen Seite treten einige Nachteile stärker in Erscheinung als angenommen. So wird bei ledigich zehn Prädikatsconstraints bereits der Grad der Pfadkonsistenz erforderlich, welches wie bereits oben erwähnt nicht direkt realisierbar ist. Weiterhin

treten stellenweise zyklische Abhängigkeiten zwischen einigen Regeln auf, was bei der regelbasierten Methode zu Endlosschleifen führt. Daher müssen die Regeln, die in den Zyklen vorkommen, aus der Regelbasis entfernt werden.
Der Nachteil schlechter Wartbarkeit der Regelbasis tritt im vorliegenden Fall aufgrund der relativ geringen Regelanzahl nicht in Erscheinung, deutet sich jedoch bei der Modularisierung der Regelbasis an. Die sinnvolle Unterteilung der Regelbasis in Modulen stellt sich mit steigenden Regelzahlen als aufwendig und unflexibel heraus. Für kleine Regelmengen mit relativ wenigen gegenseitigen Abhängigkeiten kann sie jedoch sinnvoll eingesezt werden.

Fazit und Ausblick

Als Fazit bleibt festzuhalten, dass regelbasierte Systeme sich hervorragend für Probleme mit rein vorwärtsverkettenden, zykelfreien Abhängigkeiten und relativ kleinen Regelmengen eignen. Fälle mit großen Anzahlen an Abhängigkeiten führen in der Regelbasis zu Unübersichtlichkeit und Wartungsproblemen, auch bzw. insbesondere beim Einsatz von Modulen. Problemstellungen, bei denen kanten- oder pfadkonsistente Prädikatsconstraints notwendig sind, eignen sich ebenfalls schlecht für den regelbasierten Ansatz. In solchen Fällen ist eine Kombination des regelbasierten Systems mit anderen Konfigurationsmethoden oder die Verwendung eines constraintbasierten Systems empfehlenswert. Im erstgenannten Vorschlag könnte das regelbasierte System z. B. Teilaufgaben für vorwärtsverkettende Abhängigkeiten übernehmen.

Die betrachtete Problemstellung kann als Grenzfall eingestuft werden, da sich die Einschränkungen durch Unzulänglichkeiten des regelbasierten Systems noch in akzeptablen Grenzen halten und die Vorteile des gewählten Vorgehens immer noch überwiegen. Der Umstieg auf eine constraintbasierte Methode empfielt sich dann, wenn eine Erweiterung der Anforderungen und Abhängigkeiten in Sicht ist oder ein dynamischeres System mit vollständiger Erfüllung von Prädikatsconstraints gewünscht wird.

Neben dem Umstieg auf eine constraintbasierte Methode sind auch bei der Beibehaltung des regelbasierten Vorgehens eine Reihe von Verbesserungen denkbar. So könnten zum Beispiel die Grenzwerte von Variablen, welche entsprechend Kapitel 5.2.1 während der Berechnungen vorliegen, gleich neben den Eingabefeldern angezeigt werden, damit der Benutzer diese durch inkonsistente Eingaben nicht erst herausfinden muss. Weiterhin könnte das „Hängenbleiben“ in einem lokalen Optimum durch die Hinzunahme weiterer Heuristiken eingeschränkt werden. Dazu sei an dieser Stelle auf [2] verwiesen. Es wäre für den Anwender außerdem sehr hilfreich zu jeder Variablen die in Relation stehenden Variablen mitgeteilt zu bekommen.

A Die Regelbasis

```
1   ;;=======================================
2   ;; K N O W L E D G E B A S E
3   ;;=======================================
4
5
6   (import JessUserfunctions.*)
7   (import java.util.LinkedList)
8   (load-function WarningList)
9   (load-function PartSelection)
10
11  (set-reset-globals nil)
12
13  ;;---------------------------------------
14  ;; Global variables
15  ;;---------------------------------------
16
17  (defglobal ?*PartsList*      = nil)
18  (defglobal ?*selectedPartNo* = -1)
19
20  (defglobal ?*sourcePart*        = nil)
21  (defglobal ?*sourceVar*     = nil)
22  (defglobal ?*sourceOldVal*  = nil)
23
24
25
26
27  ;;*****************************************
28  ;;*****************************************
29  ;;                       R U L E S
30  ;;*****************************************
31  ;;*****************************************
32
33
34  ;;=======================================
35  ;; Part rules
36  ;;=======================================
37
38
39  ;;---------------------------------------
40  ;; Motor (PCU)
41  ;;---------------------------------------
42
43  (defrule MAIN::uebersetzung_differentialgetriebe
44          (GlobalCalculations (nomSpeed ?gn))
```

```
45          ?m <- (Motor (nomSpeed ?n)(index ?i))
46          =>
47          (if (and (neq ?gn nil)(neq ?n nil)) then
48              (modify ?m (ratio_differentialGear (/ (* ?n 2) ?gn)))
49          (assert (ratio_differentialGear-set ?i))
50          else
51          (modify ?m (ratio_differentialGear nil))
52                  (assert (ratio_differentialGear-set ?i))
53      )
54
55  )
56
57
58  ;;----------------------------------------
59  ;; Shaft
60  ;;----------------------------------------
61
62  (defmodule SHAFT)
63
64
65  (defrule shaft_einspannungsfall
66          ?s <- (Shaft (neighborsOfPorts ?n) (index ?i))
67          (test (eq (?n size) 2))
68          =>
69          (if (and (?n contains "SplineJoint")
70          (?n contains "UniversalJoint")) then
71                  (modify ?s (alpha_restraint
72                  (* 1.9 (** 10 8))))
73                  else(if (and (eq (?n get 0) "SplineJoint")
74                  (eq (?n get 1)
75                  "SplineJoint")) then
76                   (modify ?s (alpha_restraint (* 2.77 (** 10 8))))
77                 else(if (and (eq (?n get 0) "UniversalJoint")
78                 (eq (?n get 1)
79                      "UniversalJoint")) then
80                       (modify ?s (alpha_restraint (* 1.22 (** 10 8))))
81
82                    )
83                  )
84          )
85  )
86
87
88  (defrule shaft_kritische_drehzahl
89          ?s <- (Shaft (innerDiameter ?d)
90          (outerDiameter ?D)(length ?L)
```

```
91          (alpha_restraint ?a))
92          =>
93          (if (and (neq ?d nil)  (neq ?D nil)  (neq ?L nil)
94          (neq ?a nil)) then
95                  (modify ?s (criticalSpeed
96                  (/ (* ?a (sqrt(+ (** ?D 2)
97                  (** ?d 2)))) (** ?L 2)) ))
98                  else
99                   (modify ?s (criticalSpeed nil ))
100         )
101 )
102
103
104 (defrule shaft_kritische_drehzahl_kriterium
105         (declare (auto-focus TRUE))
106         ?s <- (Shaft (criticalSpeed ?c))
107         ?r <- (RotSpeedCalc (whirlingSpeed_limit ?w))
108         (GraphicAdmin (graphicAdmin ?ga))
109         (MAIN::whirlingSpeed-set)
110         (test (and (neq ?c nil)(neq ?w nil)(neq ?ga nil)))
111         (test (< ?c ?w))
112         =>
113         (warningList "Critical Speed exceeded!" ?c ?w
114         ?*sourceVar* ?ga)
115         (assert (undo_source))
116         (printout t "Kritische Drehzahl erreicht! Ist:
117         "?c " Soll: >= " ?w  crlf)
118 )
119
120
121 (defrule shaft_inertia
122         ?s <- (Shaft (innerDiameter ?d)(outerDiameter ?D)
123         (length ?L)(material ?m) )
124         =>
125         (bind ?rho (?m getDensity))
126         (if (and (neq ?d nil)  (neq ?D nil)  (neq ?L nil)
127         (neq ?m nil)(neq ?rho nil)) then
128                 (modify ?s (inertia (/ (* (pi) ?rho (- (** ?D 4)
129                 (** ?d 4)) ?L) 32 (** 10 12))) )
130                 else
131                 (modify ?s (inertia nil))
132         )
133 )
134
135
136 (defrule shaft_stiffness
```

```
137         ?s <- (Shaft (innerDiameter ?d)(outerDiameter ?D)
138         (length ?L)(material ?m) )
139         =>
140         (bind ?gModule (?m getGModule))
141         (if (and (neq ?d nil)  (neq ?D nil)  (neq ?L nil)
142         (neq ?m nil)(neq ?gModule nil)) then
143                 (modify ?s (stiffness (/ (* (pi)  ?gModule
144                 (- (** ?D 4) (** ?d 4))  ) 32 ?L (** 10 9))) )
145                 else
146                         (modify ?s (stiffness nil))
147         )
148 )
149
150
151 (defrule shaft_mass
152         ?s <- (Shaft (innerDiameter ?d)(outerDiameter ?D)
153         (length ?L)(material ?m))
154         =>
155         (bind ?rho (?m getDensity))
156         (if (and (neq ?d nil)  (neq ?D nil)  (neq ?L nil)
157         (neq ?m nil)(neq ?rho nil)) then
158                 (modify ?s (mass (/ (* (pi)  ?rho ?L (- (** ?D 2)
159                  (** ?d 2)) ) 4 (** 10 9))) )
160                 else
161                  (modify ?s (mass nil))
162         )
163 )
164
165
166 (defrule shaft_torsionsspannung
167         ?s <- (Shaft (innerDiameter ?d)(outerDiameter ?D)
168         (maxTorque ?Tmax))
169         =>
170         (if  (and (neq ?d nil) (neq ?D nil)(neq ?Tmax nil)) then
171                 (modify ?s (tau_T (/ (* ?Tmax 1000 16 ?D) (pi)
172                 (- (** ?D 4)(** ?d 4)) ) ))
173                 else
174                  (modify ?s (tau_T nil))
175         )
176 )
177
178
179 (defrule shaft_torsionsspannung_kriterium
180         ?s <- (Shaft (material ?m)(tau_T ?tau))
181         (GraphicAdmin (graphicAdmin ?ga))
182         (test (and (neq ?m nil)(neq ?tau nil)(neq
```

```
183          (?m getTau_allowed) nil)(neq ?ga nil)))
184          (test (> ?tau (?m getTau_allowed)))
185          =>
186          (warningList "Tau_allowed exceeded!" ?tau
187          (?m getTau_allowed) ?*sourceVar* ?ga)
188          (printout t "zulässige Torsionsspannung überschritten!
189          Ist: "?tau " Soll: <= " (?m getTau_allowed)  crlf)
190          (assert (undo_source))
191  )
192
193
194  (defrule shaft_zul_moment
195          ?s <- (Shaft (innerDiameter ?d)(outerDiameter ?D)
196          (material ?m))
197          =>
198          (bind ?tau_zul (?m getTau_allowed))
199          (if (and (neq ?d nil) (neq ?D nil)(neq ?m nil)
200          (neq ?tau_zul nil)) then
201                  (modify ?s (maxTorque_allowed
202                  (/ (* (pi) (- (** ?D 3) (** ?d 3)) ?tau_zul)
203                  16 (** 10 3))))
204                  else
205                   (modify ?s (maxTorque_allowed nil))
206          )
207  )
208
209
210  (defrule shaft_zul_moment_kriterium
211          ?s <- (Shaft (maxTorque ?m) (maxTorque_allowed ?m_allowed))
212          (GraphicAdmin (graphicAdmin ?ga))
213          (test (and (neq ?m nil)(neq ?m_allowed nil)(neq ?ga nil)))
214          (test (> ?m ?m_allowed))
215          =>
216          (warningList "M_allowed exceeded!"
217          ?m ?m_allowed ?*sourceVar* ?ga)
218          (assert (undo_source))
219          (printout t "zulässiges Moment überschritten! Ist: "?m "
220          Soll: <= " ?m_allowed  crlf)
221  )
222
223  ;;würde zur Endlosschleife führen -> nehmen Länge an und
224  ;;berechnen nur in eine Richtung
225  ;;(defrule shaft_zul_laenge
226  ;;      ?s <- (Shaft (innerDiameter ?d)(outerDiameter ?D))
227  ;;      (whirlingSpeed-set)
228  ;;      (test(and (neq ?d nil) (neq ?D nil)))
```

```
;;      =>
;;      (modify ?s (maxLength (* (sqrt(sqrt(/
(* 3 (- (** (/ ?D 1000) 4) (** (/ ?d 1000) 4)) (* 21 (** 10 10)))  4
;;      (- (** (/ ?D 1000) 2) (** (/ ?d 1000) 2)) 7850 (** (pi) 2)
(** (/ ?*whirlingSpeed* 60) 2)   ))) 1000) ))
;;)


;;(defrule shaft_zul_laenge_kriterium
;;      ?s <- (Shaft (length ?l) (maxLength ?ml))
;;      (test (and (neq ?l nil)(neq ?ml nil)))
;;      (test (> ?l ?ml))
;;      =>
;;      (?*WarningList* add "Zul. Länge überschritten!")
;;      (assert (warning-message))
;;)


(defrule shaft_hanging_man
        ?s <- (Shaft (material ?m)(length ?L)
        (innerDiameter ?d)
        (outerDiameter ?D))
        (GraphicAdmin (graphicAdmin ?ga))
        (test(and (neq ?m nil) (neq ?L nil)
        (neq ?d nil) (neq ?D nil)
        (neq (?m getSigma_B) nil)(neq ?ga nil)))
        (test (< (?m getSigma_B) (/ (* 8000 ?L)
        (pi) (- (** ?D 3)(** ?d 3))) ))
        =>
        (bind ?sigmaMax (/ (* 8000 ?L)  (pi)
        (- (** ?D 3)(** ?d 3))) )
        (warningList "Hanging Man not fulfilled!" ?sigmaMax
        (?m getSigma_B) ?*sourceVar* ?ga)
        (assert (undo_source))
        (printout t "Hanging Man nicht erfüllt! sigma_B_zul: "
        (?m getSigma_B) ", ist: " ?sigmaMax crlf)
)


(defrule shaft_min_aussendurchmesser
        ?s <- (Shaft (material ?m)(length ?L))
        =>
        (bind ?sigmaB (?m getSigma_B))
        (if (and (neq ?m nil) (neq ?L nil)(neq ?sigmaB nil)) then
         (modify ?s (minOuterDiameter (* 2 (** (/ (* 1000 ?L)
         (pi) ?sigmaB) (/ 1 3))) ))
```

```
275           else
276        (modify ?s (minOuterDiameter nil))
277           )
278   )
279
280
281   (defrule shaft_min_aussendurchmesser_kriterium
282           ?s <- (Shaft (outerDiameter ?D) (minOuterDiameter ?mD))
283           (GraphicAdmin (graphicAdmin ?ga))
284           (test (and (neq ?D nil)(neq ?mD nil)(neq ?ga nil)))
285           (test (< ?D ?mD))
286           =>
287           (warningList "Minimum Outer Diameter exceeded!" ?D ?mD
288           ?*sourceVar* ?ga)
289           (assert (undo_source))
290   )
291
292
293   ;;-----------------------------------------------
294   ;; Torque Limiter (STL)
295   ;;-----------------------------------------------
296
297   (set-current-module MAIN)
298
299
300   (defrule MAIN::STL_M_drive
301           ?tl <- (TorqueLimiter (index ?i))
302           (GraphicAdmin (graphicAdmin ?ga))
303           (test (neq ?i nil))
304           =>
305           (try
306               (bind ?w (new WindowMinimumDriveTorque ?ga))
307           (definstance MAIN::WindowMinimumDriveTorque ?w dynamic)
308           (bind ?driveTorque (?w checkDriveTorqueOfNeighbor ?i))
309           (modify ?tl (m_drive ?driveTorque))
310       catch
311
312           )
313   )
314
315
316   (defrule MAIN::lower-stl-setting
317           ?tl <- (TorqueLimiter (m_drive ?m)(robustnessFactor ?r)
318           (index ?i))
319           =>
320           (if (and (neq ?m nil)(neq ?r nil)(neq ?i nil)) then
```

```
321                     (modify ?tl (lower_STL_Setting (* ?m ?r)))
322                     else
323                             (modify ?tl (lower_STL_Setting nil))
324             )
325     )
326
327
328     (defrule MAIN::upper-stl-setting
329             ?tl <- (TorqueLimiter (lower_STL_Setting ?l)
330             (marginForProducer ?m)(index ?i))
331             =>
332             (if (and (neq ?l nil)(neq ?m nil)(neq ?i nil)) then
333                     (modify ?tl (upper_STL_Setting (* ?l ?m)))
334                     else
335                             (modify ?tl (upper_STL_Setting nil))
336             )
337     )
338
339
340     ;;-----------------------------------------------
341     ;; Wing Tip Brake (WTB)
342     ;;-----------------------------------------------
343
344     (defrule MAIN::lower-wtb-setting
345             (TorqueLimiter (upper_STL_Setting ?u))
346             ?w  <- (WingTipBrake)
347             =>
348             (if (neq ?u nil) then
349                     (modify ?w (lowerWTBSetting ?u))
350                     else
351                             (modify ?w (lowerWTBSetting nil))
352             )
353     )
354
355
356     (defrule MAIN::upper-wtb-setting
357             (TorqueLimiter (upper_STL_Setting ?u))
358             ?w  <- (WingTipBrake (marginForProducer ?m))
359             =>
360             (if (and (neq ?u nil)(neq ?m nil)) then
361                     (modify ?w (upperWTBSetting (* ?u ?m)))
362                     else
363                             (modify ?w (upperWTBSetting nil))
364             )
365     )
366
```

```
367
368
369   ;;=========================================
370   ;; Global rules
371   ;;=========================================
372
373   (defmodule GLOBAL)
374
375
376   ;;-----------------------------------------
377   ;; Basic Definitions
378   ;;-----------------------------------------
379
380   (defrule low-speed
381           ?a <- (MAIN::GlobalCalculations (nomSpeed ?n))
382           =>
383           (if (neq ?n nil) then
384                   (modify ?a (lowSpeed (* ?n 0.18)))
385                   else
386                           (modify ?a (lowSpeed nil))
387           )
388   )
389
390
391   (defrule T_2
392           ?a <- (MAIN::GlobalCalculations (extensionTime ?tt)
393           (t_1 ?t1)(t_3 ?t3)(t_4 ?t4)(t_5 ?t5))
394           =>
395           (if (and (neq ?tt nil)(neq ?t1 nil)(neq ?t3 nil)
396           (neq ?t4 nil)(neq ?t5 nil)) then
397                   (modify ?a (t_2 (- ?tt ?t1 ?t3 ?t4 ?t5)))
398                   else
399                           (modify ?a (t_2 nil))
400           )
401   )
402
403
404   (defrule average-speed
405           ?a <- (MAIN::GlobalCalculations (t_1 ?t1)(t_2 ?t2)(t_3 ?t3)
406           (t_4 ?t4)(extensionTime ?tt)(nomSpeed ?n)(lowSpeed ?l))
407           =>
408           (if (and (neq ?t1 nil)(neq ?t2 nil)(neq ?t3 nil)(neq ?t4 nil)
409           (neq ?tt nil)(neq ?n nil)(neq ?l nil)) then
410            (modify ?a (averageSpeed (/ (+ (* (/ ?n 2) ?t1) (* ?n ?t2)
411         (* (/ (+ ?n ?l) 2) ?t3) (* ?l ?t4)) ?tt)  ))
412                   else
```

```
413                     (modify ?a (averageSpeed nil))
414             )
415     )
416
417
418     (defrule ShaftRotationsPerExtension_nomSpeed
419             ?g <- (MAIN::GlobalCalculations (extensionTime ?eT)
420             (nomSpeed ?nS))
421             =>
422             (if (and (neq ?nS nil)(neq ?eT nil)) then
423                     (modify ?g (shaftRotationsPerExtension_nomSpeed
424                     (/ (* ?nS ?eT) 60)))
425                     else
426                      (modify ?g (shaftRotationsPerExtension_nomSpeed nil))
427             )
428     )
429
430
431     (defrule ShaftRotationsPerExtension_avgSpeed
432             ?g <- (MAIN::GlobalCalculations (extensionTime ?eT)
433             (averageSpeed ?aS))
434             =>
435             (if (and (neq ?aS nil)(neq ?eT nil)) then
436                     (modify ?g (shaftRotationsPerExtension_avgSpeed
437                     (/ (* ?aS ?eT) 60)))
438                     else
439                      (modify ?g (shaftRotationsPerExtension_avgSpeed nil))
440             )
441     )
442
443
444     (defrule M_necessary_global
445             (declare (auto-focus TRUE))
446             ?gc <- (MAIN::GlobalCalculations)
447             (MAIN::Motor)
448             (MAIN::GraphicAdmin (graphicAdmin ?ga))
449             (test(neq ?ga nil))
450             =>
451             ;;berechnetes PCU-Moment
452             (try
453                 (bind ?w (new WindowMinimumDriveTorque ?ga))
454             (definstance MAIN::WindowMinimumDriveTorque ?w dynamic)
455             (bind ?driveTorque (?w checkDriveTorque))
456             (modify ?gc (m_necessary ?driveTorque))
457             catch
458             ;;
```

```
459           )
460   )
461
462
463   (defrule eta_transmission
464           (declare (auto-focus TRUE))
465           ?gc <- (MAIN::GlobalCalculations (m_necessary ?Mn))
466           (MAIN::Motor)
467           (MAIN::GraphicAdmin (graphicAdmin ?ga))
468           (test (and (neq ?Mn nil)(neq ?ga nil)))
469           =>
470           ;;gesamtes Lastmoment
471           (try
472           (bind ?glt (new GetLoadTorque ?ga))
473           (definstance MAIN::GetLoadTorque ?glt dynamic)
474           (bind ?loadTorque (?glt checkLoadTorque))
475
476           ;;eta_transmission
477           (if (and (neq ?loadTorque nil)(neq ?loadTorque 0.0)) then
478            (if (< ?loadTorque ?Mn) then
479                 (modify ?gc (eta_transmission
480                 (abs (/ ?loadTorque ?Mn))))
481                  else (if (> ?loadTorque ?Mn) then
482                  (modify ?gc (eta_transmission
483                  (abs (/ ?Mn ?loadTorque))))
484                  )
485            )
486           )
487           catch
488           ;;
489           )
490   )
491
492
493   ;;-----------------------------------------
494   ;; Rotation Speeds
495   ;;-----------------------------------------
496
497   (defmodule ROTBLOW)
498
499
500   (defrule pcu_overspeed_threshold
501           (declare (auto-focus TRUE))
502           ?r <- (MAIN::RotSpeedCalc)
503           (MAIN::GlobalCalculations (nomSpeed ?n))
504           =>
```

```
505           (if (neq ?n nil) then
506                   (modify ?r (pcu_OverspeedThreshold (* ?n 1.1)))
507                   else
508                           (modify ?r (pcu_OverspeedThreshold nil))
509           )
510   )
511
512
513   (defrule pcu_overspeed_kriterium
514           ?r <- (MAIN::RotSpeedCalc (pcu_OverspeedThreshold ?ot)
515           (pcu_OverspeedThreshold_choosen ?o))
516           (MAIN::GraphicAdmin (graphicAdmin ?ga))
517           (test (and (neq ?ot nil)(neq ?o nil)(neq ?ga nil)))
518           (test (< ?o ?ot))
519           =>
520           (warningList "PCU overspeed threshold reached!"
521           ?ot ?o ?*sourceVar* ?ga)
522           (assert (undo_source))
523           (printout t "PCU overspeed threshold unterschritten!
524           Ist: "?o " Soll: >= " ?ot  crlf)
525   )
526
527
528   (defrule WhirlingSpeed_limit
529           ?r <- (MAIN::RotSpeedCalc (pcu_OverspeedThreshold_choosen ?o))
530           =>
531           (if (neq ?o nil) then
532                   (modify ?r (whirlingSpeed_limit (* ?o 1.1)))
533                   (assert (whirlingSpeed-set))
534                   else
535                           (modify ?r (whirlingSpeed_limit nil))
536           )
537   )
538
539
540   (defrule transmission_speed_for_motor_runaway
541           (declare (auto-focus TRUE))
542           ?r <- (MAIN::RotSpeedCalc)
543           ?m <- (MAIN::Motor (maxSpeed_peak ?mp)(nomSpeed ?n)
544           (ratio_differentialGear ?rd))
545           =>
546           (if (and (neq ?mp nil)(neq ?n nil)(neq ?rd nil)) then
547                   (modify ?r
548                   (transmission_speed_for_motor_runaway
549                   (/ (+ ?mp ?n) ?rd)))
550                   else
```

```
551                     (modify ?r
552                     (transmission_speed_for_motor_runaway nil))
553             )
554     )
555
556     ;; 2 rules, da selbe LHS
557     (defrule minSystemOverspeedThreshold_kriterium
558             ?r <- (MAIN::RotSpeedCalc
559             (transmission_speed_for_motor_runaway ?t)
560             (minSystemOverspeedThreshold ?mi))
561             ?b <- (MAIN::BlowbackCalc (maxSystemOverspeedThreshold ?ma))
562             (MAIN::GraphicAdmin (graphicAdmin ?ga))
563             (test (and (neq ?t nil)(neq ?mi nil)
564              (neq ?ma nil)(neq ?ga nil)))
565             (test (or (< ?mi ?t)(> ?mi ?ma)))
566             =>
567             (if (< ?mi ?t) then
568              (warningList "minSystemOverspeedThreshold unterschritten!"
569           ?mi ?t ?*sourceVar* ?ga)
570                else (if (> ?mi ?ma) then
571                (warningList "maxSystemOverspeedThreshold exceeded!"
572                ?mi ?ma ?*sourceVar* ?ga)
573                )
574             )
575
576             (assert (undo_source))
577     )
578
579
580     ;;----------------------------------------
581     ;; Blowback-Untersuchung
582     ;;----------------------------------------
583
584     ;;------ Beschleunigungsgradienten ------
585
586     (defrule acceleration-torque
587             (declare (auto-focus TRUE))
588             ?bc <- (MAIN::BlowbackCalc (caseNo ?c))
589             ?gc <- (MAIN::GlobalCalculations (m_necessary ?m))
590             (MAIN::GraphicAdmin (graphicAdmin ?ga))
591             (test (and (neq ?m nil)(neq ?ga nil)))
592             (not (part-selected))
593             =>
594             (if (eq ?c 1) then
595                     (modify ?bc (accelerationTorque (abs ?m)))
596                     else (if (eq ?c 2) then
```

```
597                     (modify ?bc (accelerationTorque (/ (abs ?m) 2)))
598                 else (if (eq ?c 3) then
599                      ;;choose part over GUI
600
601                      ;;->direkter Aufruf in Java aus BlowbackPanel,
602                      sonst wird
603                      ;;PartSelector bei jedem Durchlauf von
604                      Jess immer wieder aufgerufen.
605                     )
606             )
607         )
608 )
609
610
611 (defrule inertia-total
612         ?bc <- (MAIN::BlowbackCalc (caseNo ?c))
613         (MAIN::GraphicAdmin (graphicAdmin ?ga))
614         (test(neq ?ga nil))
615         (not (part-selected))
616         =>
617         (bind ?i (new InertiaCalc ?ga))
618         (definstance MAIN::InertiaCalc ?i dynamic)
619         (bind ?inertia (?i getTotalInertia))
620         (if (eq ?c 1) then
621                 (modify ?bc (inertia_total (* ?inertia 2)))
622                 else (if (eq ?c 2) then
623                         (modify ?bc (inertia_total ?inertia))
624                         else (if (eq ?c 3) then
625                                 ;;choose part over GUI
626                         )
627                 )
628         )
629 )
630
631
632 ;;Is called after a part is choosen: Calculates
633 M_drive and Inertia_total for choosen part
634
635 (defrule accelerationTorque_Inertia-total_choosen-part
636         ;;(declare (auto-focus TRUE))
637         ?bc <- (MAIN::BlowbackCalc)
638         (MAIN::GraphicAdmin (graphicAdmin ?ga))
639         ?p  <- (part-selected)
640         (test (and (neq ?*selectedPartNo* -1)(neq ?ga nil)))
641         =>
642         (try
```

```
643             ;;calculate M_drive for choosen part
644         (bind ?w (new WindowMinimumDriveTorque ?ga))
645         (definstance MAIN::WindowMinimumDriveTorque ?w dynamic)
646         (bind ?driveTorque (?w checkDriveTorque ?*selectedPartNo*))
647         (modify ?bc (accelerationTorque ?driveTorque))
648
649         ;;calculate Inertia_total for choosen part
650         (bind ?i (new InertiaCalc ?ga ?*selectedPartNo*))
651         (definstance MAIN::InertiaCalc ?i dynamic)
652         (bind ?inertia (?i getChoosenTotalInertia))
653         (modify ?bc (inertia_total ?inertia))
654         catch
655         ;;
656         )
657
658         (retract ?p)
659 )
660
661
662 (defrule acceleration
663         ?bc <- (MAIN::BlowbackCalc (accelerationTorque ?at)
664         (inertia_total ?it))
665         =>
666         (if (and (neq ?at nil)(neq ?it nil)) then
667                 (modify ?bc (acceleration (/ (* ?at 60) ?it 2 (pi))))
668                 else
669                         (modify ?bc (acceleration nil))
670         )
671 )
672
673
674 (defrule brake-delay_kriterium
675         ?bc <- (MAIN::BlowbackCalc (brakeDelay ?br))
676         (MAIN::GraphicAdmin (graphicAdmin ?ga))
677         (test (and (neq ?br nil)(neq ?ga nil)))
678         (test (or (< ?br 20)(> ?br 50)))
679         =>
680         (if (< ?br 20) then
681                 (warningList "brakeDelay minimum limit violated!"
682                 ?br 20 ?*sourceVar* ?ga)
683                 else (if (> ?br 50) then
684                         (warningList "brakeDelay maximum limit
685                         violated!"
686                         ?br 50 ?*sourceVar* ?ga)
687                 )
688         )
```

```
689         (assert (undo_source))
690 )
691
692
693 (defrule acceleration-time
694         (declare (auto-focus TRUE))
695         ?bc <- (MAIN::BlowbackCalc (acceleration ?a))
696         ?r  <- (MAIN::RotSpeedCalc (minSystemOverspeedThreshold ?mso)
697         (maxDynamicTransmissionFailureSpeed ?mdt))
698         =>
699         (if (and (neq ?a nil)(neq ?mso nil)(neq ?mdt nil)) then
700                 (if (neq (/ (- ?mdt ?mso) ?a) NaN) then
701                  (modify ?bc (accelerationTime (/ (- ?mdt ?mso) ?a)))
702                 )
703                 else
704                  (modify ?bc (accelerationTime nil))
705         )
706 )
707
708
709 (defrule detection-time
710         ?bc <- (MAIN::BlowbackCalc (brakeDelay ?b)
711         (accelerationTime ?a))
712         =>
713         (if (and (neq ?b nil)(neq ?a nil)) then
714                 (modify ?bc (detectionTime (- ?a ?b)))
715                 else
716                  (modify ?bc (detectionTime nil))
717         )
718 )
719
720
721 (defrule detection-time-kriterium
722         (MAIN::BlowbackCalc (detectionTime ?d))
723         (MAIN::GraphicAdmin (graphicAdmin ?ga))
724         (test (and (neq ?d nil)(neq ?ga nil)))
725         (test (< ?d 0))
726         =>
727         (warningList "detectionTime minimum limit violated!"
728         ?d 0 ?*sourceVar* ?ga)
729         (assert (undo_source))
730         (printout t "Erkennungszeit unzulässig!"  crlf)
731 )
732
733 ;;------ Maximum System Overspeed Threshold ------
734
```

```
735  (defrule detection-time-calculated
736          ?bc <- (MAIN::BlowbackCalc (samplingRate ?s)
737          (confirmationCycles ?c)(delayDueToSpeedAlgorithm ?d))
738          =>
739          (if (and (neq ?s nil)(neq ?c nil)(neq ?d nil)) then
740                  (modify ?bc (detectionTime_calculated
741                  (+ (* (+ ?c 1) ?s) ?d)))
742                  else
743                   (modify ?bc (detectionTime_calculated nil))
744          )
745  )
746
747
748  (defrule acceleration-time-calculated
749          ?bc <- (MAIN::BlowbackCalc (brakeDelay ?b)
750           (detectionTime_calculated ?d))
751          =>
752          (if (and (neq ?b nil)(neq ?d nil)) then
753                  (modify ?bc (accelerationTime_calculated (+ ?b ?d)))
754                  else
755                   (modify ?bc (accelerationTime_calculated nil))
756          )
757  )
758
759
760  (defrule max-system-overspeed-threshold
761          (declare (auto-focus TRUE))
762          ?bc <- (MAIN::BlowbackCalc (acceleration ?a)
763           (accelerationTime_calculated ?at))
764          ?r  <- (MAIN::RotSpeedCalc
765          (maxDynamicTransmissionFailureSpeed ?m))
766          =>
767          (if (and (neq ?a nil)(neq ?at nil)(neq ?m nil)) then
768                  (modify ?bc (maxSystemOverspeedThreshold
769                  (abs (- (* ?a ?at) ?m))))
770                  else
771                   (modify ?bc (maxSystemOverspeedThreshold nil))
772          )
773  )
774
775
776  ;;----------------------------------------
777  ;; Peakmomente und Limit Loads
778  ;;----------------------------------------
779
780  (set-current-module GLOBAL)
```

```
781
782
783   (defrule Stiffness
784           (declare (auto-focus TRUE))
785           ?l <- (MAIN::LimitLoadsCalc (eigenfrequency ?e))
786           (MAIN::BlowbackCalc (inertia_total ?i))
787           =>
788           (if (and (neq ?e nil)(neq ?i nil)) then
789                   (modify ?l (stiffness (/ (* (** ?e 2) ?i) 1000)))
790                   else
791                           (modify ?l (stiffness nil))
792           )
793   )
794
795
796   (defrule MaxSpeed_n0
797           (declare (auto-focus TRUE))
798           ?l <- (MAIN::LimitLoadsCalc)
799           (MAIN::BlowbackCalc (maxSystemOverspeedThreshold ?m))
800           =>
801           (if (neq ?m nil) then
802                   (modify ?l (maxSpeed_n0 (abs ?m)))
803                   else
804                           (modify ?l (maxSpeed_n0 nil))
805           )
806   )
807
808
809   (defrule LimitLoad
810           (declare (auto-focus TRUE))
811           ?l <- (MAIN::LimitLoadsCalc (stiffness ?s)
812           (maxSpeed_n0 ?m))
813           (MAIN::BlowbackCalc (inertia_total ?i))
814           =>
815           (if (and (neq ?s nil)(neq ?m nil)(neq ?i nil)) then
816                   (modify ?l (limitLoad
817                   (sqrt (/ (* ?s ?i ?m (pi)) 1000 30))))
818                   else
819                    (modify ?l (limitLoad nil))
820           )
821   )
822
823   ;;--------------------------------------
824   ;; Asymmetry Position Limit
825   ;;--------------------------------------
826
```

```
827   ;;------ Klappenfehlstellung ------
828
829   (defrule AsymmetryLimit
830           ?ac <- (MAIN::AsymmetryCalc (asymmetryLimit_aerodynamics ?aa)
831           (flapFalsePos_mechSetting ?fs)
832           (flapFalsePos_manufacturingTolerance ?ft))
833           =>
834           (if (and (neq ?aa nil)(neq ?fs nil)(neq ?ft nil)) then
835                   (modify ?ac (asymmetryLimit (- ?aa ?fs ?ft)))
836                   else
837                    (modify ?ac (asymmetryLimit nil))
838           )
839   )
840
841   ;;------ Aymmetry-Monitoring ------
842
843   (defrule MonitoringThreshold
844           ?ac <- (MAIN::AsymmetryCalc (asymmetryLimit ?a)
845           (safetyFactor ?s))
846           =>
847           (if (and (neq ?a nil)(neq ?s nil)) then
848                   (modify ?ac (monitoringThreshold (- ?a ?s)))
849                   else
850                           (modify ?ac (monitoringThreshold nil))
851           )
852   )
853
854
855   (defrule AsymmetryMonitoringThreshold
856           (declare (auto-focus TRUE))
857           ?ac <- (MAIN::AsymmetryCalc (monitoringThreshold ?m))
858           ?gc <- (MAIN::GlobalCalculations (extensionAngle ?e)
859           (shaftRotationsPerExtension_avgSpeed ?s))
860           =>
861           (if (and (neq ?m nil)(neq ?e nil)(neq ?s nil)) then
862                   (modify ?ac (asymmetryMonitoringThreshold
863                   (/ (* ?m ?s) ?e)))
864                   else
865                    (modify ?ac (asymmetryMonitoringThreshold nil))
866           )
867   )
868
869
870   (defrule AsymmetryDetectionBand
871           (declare (auto-focus TRUE))
872           ?ac <- (MAIN::AsymmetryCalc (detectionBand ?d))
```

```
873           ?gc <- (MAIN::GlobalCalculations (extensionAngle ?e)
874           (shaftRotationsPerExtension_avgSpeed ?s))
875           =>
876           (if (and (neq ?d nil)(neq ?e nil)(neq ?s nil)) then
877                   (modify ?ac (asymmetryDetectionBand (/ (* ?d ?s) ?e)))
878                   else
879                           (modify ?ac (asymmetryDetectionBand nil))
880           )
881   )
882
883
884   (defrule PositionDifference_brakeDelay
885           (declare (auto-focus TRUE))
886           ?ac <- (MAIN::AsymmetryCalc)
887           ?r  <- (MAIN::RotSpeedCalc (pcu_OverspeedThreshold_choosen ?o)
888           (minSystemOverspeedThreshold ?m))
889           ?bc <- (MAIN::BlowbackCalc (brakeDelay ?b))
890           =>
891           (if (and (neq ?o nil)(neq ?m nil)(neq ?b nil)) then
892                   (modify ?ac (positionDifference_brakeDelay
893                   (/ (* (+ ?o ?m) ?b) 60000)))
894                   else
895                    (modify ?ac (positionDifference_brakeDelay nil))
896           )
897   )
898
899
900   (defrule ResultingAsymmetry_rot
901           ?ac <- (MAIN::AsymmetryCalc (asymmetryMonitoringThreshold ?am)
902           (asymmetryDetectionBand ?ad)
903           (positionDifference_brakeDelay ?p))
904           =>
905           (if (and (neq ?am nil)(neq ?ad nil)(neq ?p nil)) then
906                   (modify ?ac (resultingAsymmetry_rot (+ ?am ?ad ?p)))
907                   else
908                           (modify ?ac (resultingAsymmetry_rot nil))
909           )
910   )
911
912
913   (defrule ResultingAsymmetry_deg
914           (declare (auto-focus TRUE))
915           ?gc <- (MAIN::GlobalCalculations (extensionAngle ?e)
916           (shaftRotationsPerExtension_avgSpeed ?s))
917           ?ac <- (MAIN::AsymmetryCalc (resultingAsymmetry_rot ?r))
918           =>
```

```
919          (if (and (neq ?e nil)(neq ?s nil)(neq ?r nil)) then
920                  (modify ?ac (resultingAsymmetry_deg (/ (* ?e ?r) ?s)))
921                  else
922                          (modify ?ac (resultingAsymmetry_deg nil))
923          )
924  )
925
926  ;;------ Asymmetry Detection Time ------
927
928  (defrule Time_reachAsymmetryThreshold_t1
929          (declare (auto-focus TRUE))
930          ?ac <- (MAIN::AsymmetryCalc (asymmetryMonitoringThreshold ?a))
931          ?r  <- (MAIN::RotSpeedCalc
932          (pcu_OverspeedThreshold_choosen ?o)
933          (minSystemOverspeedThreshold ?m))
934          =>
935          (if (and (neq ?a nil)(neq ?o nil)(neq ?m nil)) then
936                  (modify ?ac (time_reachAsymmetryThreshold_t1
937                  (/ (* ?a 60000) (+ ?o ?m))))
938                  else
939                   (modify ?ac (time_reachAsymmetryThreshold_t1 nil))
940          )
941  )
942
943
944  (defrule DetectionTime_t2
945          (declare (auto-focus TRUE))
946          ?ac <- (MAIN::AsymmetryCalc (asymmetryDetectionBand ?a))
947          ?r  <- (MAIN::RotSpeedCalc (pcu_OverspeedThreshold_choosen ?o)
948          (minSystemOverspeedThreshold ?m))
949          =>
950          (if (and (neq ?a nil)(neq ?o nil)(neq ?m nil)) then
951                  (modify ?ac (detectionTime_t2
952                  (/ (* ?a 60000) (+ ?o ?m))))
953                  else
954                   (modify ?ac (detectionTime_t2 nil))
955          )
956  )
957
958
959
960  ;;======================================
961  ;; Retracts
962  ;;======================================
963
964  (set-current-module MAIN)
```

```
965
966   ;;Sorgen dafür, dass sich Java-Instanzen nicht häufen
967   ;;Dynamisch erstellte Instanzen von
968   ;;WindowMinimumDriveTorque, etc. werden nur
969   ;;im erstellenden
970   ;;Rule benötigt. Diese werden jedoch nach
971   ;;Verwendung nicht entfernt,
972   ;;solange nicht retract aufgerufen wird
973   ;;(Jess hat keinen Garbage Collector!).
974   ;;Die folgenden Rules werden jedesmal aktiviert
975   ;;wenn ein anderes Rule eine entsprechende
976   ;;Java-Instanz erzeugt, und nach dem anderen Rule
977   ;;abgefeuert.
978   ;;Damit existieren die erzeugten Java-Instanzen
979   ;;und die dazugehörigen Jess-Facts nur
980   ;;für das Rule, in dem sie erzeugt worden sind.
981
982
983   (defrule retract_WindowMinimumDriveTorque
984           ?w  <- (MAIN::WindowMinimumDriveTorque)
985           =>
986           (retract ?w)
987   )
988
989
990   (defrule retract_GetLoadTorque
991           ?g  <- (MAIN::GetLoadTorque)
992           =>
993           (retract ?g)
994   )
995
996
997   (defrule retract_InertiaCalc
998           ?i  <- (MAIN::InertiaCalc)
999           =>
1000          (retract ?i)
1001  )
1002
1003
1004  (defrule remove
1005          ?s <- (GeneralPart (index ?i))
1006          ?r <- (remove ?i)
1007          =>
1008          (retract ?s)
1009          (retract ?r)
1010          (printout t Removed ?i crlf)
```

```
1011  )
1012
1013
1014  (defrule removeAllParts
1015          (declare (auto-focus TRUE))
1016          ?s <- (or (GeneralPart)(Material))
1017          ?r <- (remove-all-parts)
1018          =>
1019          (retract ?s)
1020  )
1021
1022
1023
1024  ;;=================================
1025  ;; Rule zum Zurücksetzen von Werten
1026  ;;=================================
1027
1028  ;; Bei Verletzung eines Constraints, welches
1029  ;;nicht direkt mit der vom User veränderten Variable
1030  ;; in Verbindung steht ("Eingabe-Variable"), wird
1031  ;;die "Eingabe-Variable" auf ihren ursprünglichen Wert
1032  ;; zurückgesetzt -> als wären der Ursprungswert
1033  ;;und die Folgewerte nicht angenommen worden.
1034
1035  (defrule undo-source
1036          ?c <- (or (undo_source)(SHAFT::undo_source)
1037          (GLOBAL::undo_source)(ROTBLOW::undo_source))
1038          ?s <- (or (GeneralPart (index ?i))
1039          (Calculation (index ?i)))
1040          (test (and (neq ?*sourcePart* nil)(neq ?i nil)))
1041          (test (eq ?i ?*sourcePart*))
1042          =>
1043          (printout t Part: ?*sourcePart* ...
1044          SourceVariable: ?*sourceVar* ...Old Value:
1045          ?*sourceOldVal* crlf)
1046          (try
1047           (modify ?s (?*sourceVar* ?*sourceOldVal*))
1048          catch
1049           (modify ?s (?*sourceVar* nil))
1050          )
1051          (retract ?c)
1052  )
1053
1054
1055
1056
```

```
;; end of rules
;;=======================
```

B Nomenklatur

B.1 Variablen

δ	$^\circ$	Klappenstellwinkel
η	–	Wirkungsgrad
ρ	$\mathsf{kg/mm^3}$	Dichte
σ	$\mathsf{N/mm^2}$	Spannung
τ	$\mathsf{N/mm^2}$	Torsionsspannung
φ	rad	Drehwinkel
ω	rad/s	Drehwinkelgeschwindigkeit
$\dot{\omega}$	$\mathsf{rad/s^2}$	Drehwinkelbeschleunigung
C	–	Konstante
D	mm	Aussendurchmesser
E	$\mathsf{N/mm^2}$	Elastizitätsmodul
F	N	Kraft
G	$\mathsf{N/mm^2}$	Schubmodul
J	$\mathsf{kgm^2}$	Massenträgheitsmoment
L	mm	Länge
M	Nm	Drehmoment
T	$^\circ$C	Temperatur
U	–	Umdrehungen
c	Nm/rad	Torsionssteifigkeit
d	Nm/s	Dämpfung
d	mm	Innendurchmesser
i	–	Getriebeübersetzung
n	1/min	Drehzahl
t	s	Zeit

B.2 Abkürzungen

APPU	Asymmetry Position Pick off Unit
BGB	Bevel Gearbox
DD	Abtrieb (*engl.* Down Drive)
DDGB	Down Drive Gearbox
DG	Differential Gear
FPPU	Feedback Position Pick off Unit
KBG	Kink Bevel Gearbox
PCU	Power Control Unit
POB	Pressure off Brake
RA	Rotary Actuator
RAG	Right Angle Gearbox
STL	System Torque Limiter
TSSU	Transmission Speed Sensor Unit
WTB	Wing Tip Brake

Literatur

[1] Cunis, R., Günter, A., Neumann, B., Syska, I.: *Wissensbasierte Planung und Konfigurierung.* Springer Verlag, Berlin, Heidelberg, New York, 1987.

[2] Dechter, R.: *Constraint Processing.* Morgan Kaufmann Publishers, San Francisco, San Diego, New York, Boston, London, Sydney, Tokyo, 2003.

[3] Deutsche Airbus (Hrsg.): *Flap-System A330/340.* System Desciption Note, 1994.

[4] Forgy, C.L.: *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.* 1982.

[5] Friedman-Hill, E.: *Jess in Action.* Manning, Greenwich, 2003.

[6] Giarratano, J.C., Riley, G.D.,: *Expert Systems, Principles and Programming, Fourth Edition.* Thomson Course Technology, Boston, USA, 2005.

[7] Günter, A., Kühn, C.: *Knowledge-Based Configuration - Survey and Future Directions - Survey and Future Directions, Proceedings of the 5th Biannual German Conference on Knowledge-Based Systems (XPS99), Würzburg, 3. 5. März 1999.* Springer Verlag, Berlin, Heidelberg, New York, 1999.

[8] Hahn, K.-U., Heinrich, R.: *Aerodynamik und Flugmechanik.* Vorlesungsskript, Technische Universität Hamburg–Harburg, 2004.

[9] Krebs, T.: *Erkennen von Benutzerintentionen im inkrementellen Konfigurationsverlauf (am Beispiel von EngCon).* Diplomarbeit, Universität Bremen, Universität Hamburg, 2002.

[10] Kühn, C.: *Vergleich unterschiedlicher Konfigurationsmethoden im Hinblick auf die Nutzbarkeit von Wissen über Zustandsverhalten der Konfigurationsobjekte.* Sauer, Jürgen, Wien, 2001.

[11] Rechter, H.: *A400M HLS–System Performance Report.* Technical Report, Airbus Deutschland GmbH, 2005.

[12] Roye, T.: *Aufbereitung des Entwicklungsprozesses für eine wissensbasierte Systemauslegung von Landeklappenantriebssystemen.* Große Studienarbeit, Flugzeug-Systemtechnik, Technische Universität Hamburg–Harburg, 2006.

[13] Runte, W.: *YACS: Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung.* Diplomarbeit, Universität Bremen, Universität Hamburg, 2006.

[14] Stumptner, M.: *Wissensbasiertes Konfigurieren.* Vorlesungsskript, Technische Universität Wien, 2000.