

Formulierung und Verarbeitung von Ingenieurwissen zur Verbesserung hydraulischer Systeme

Diplomarbeit zur Erlangung des Grades eines Diplom-Informatikers des
Fachbereichs Mathematik-Informatik an der Universität-Gesamthochschule Paderborn

vorgelegt von:

Thomas Schlotmann

vorgelegt bei:

Prof. Dr. Hans Kleine Büning

Betreuer:

Dr. Benno Stein und Diplom-Mathematiker Marcus Hoffmann

Paderborn, im März 1998

Selbständigkeitserklärung

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig und nur unter Zuhilfenahme der angegebenen Quellen und Hilfsmittel angefertigt habe.

Thomas Schlotmann

Paderborn, 16. März 1998

Inhaltsverzeichnis

1	Einleitung	4
1.1	Motivation	4
1.2	Ziele der Arbeit	5
1.3	Übersicht	6
2	Designwissen	8
2.1	Wissensformen	8
2.1.1	Anforderungswissen	8
2.1.2	Auslegungswissen	11
2.1.3	Änderungswissen	13
3	Das Auslegungsproblem	15
3.1	Beispiele zur Neu- und Änderungsauslegung	15
3.1.1	Komponenten	16
3.1.2	Eigenschaften von Komponenten	16
3.1.3	Fehlverhalten	17
3.1.4	Modifikation	18
3.1.5	Schaltplantopologie	19
3.2	Prinzipieller Aufbau einer Schaltung	21
3.2.1	action-specifier	22
3.2.2	location-specifier	23
3.2.3	Kombinationen aus action- und location-specifier	28
4	Die Designsprache	31
4.1	Einbindung der Designsprache in den Konstruktionsprozeß	31
4.1.1	Anforderungen	32
4.1.2	Konstruktion/Entwurf	32
4.1.3	Simulation	32
4.1.4	Fehlersuche	32
4.1.5	Änderungsmaßnahmen nach Fehler	32
4.1.6	Optimierung	33
4.2	Entwicklungskriterien	33
4.3	Wissensrepräsentation	33
4.3.1	Dynamisches Wissen	34
4.3.2	Statisches Wissen	34
4.3.3	Wissensklassen	35
5	Schichten der Designsprache	37
5.1	Die Sprachpyramide	37
5.1.1	Domänennahe Problemformulierung	37

5.1.2	Makros	39
5.1.3	Schnittstelle zum ArtDeco-Kern	44
6	Anwendung der Designsprache	45
6.1	Wissensklassen	45
6.2	Makros	45
6.3	Veränderung von Simulationsparametern	45
6.3.1	<code>set()</code>	46
6.3.2	<code>get()</code>	46
6.4	Lesender Zugriff auf einen Schaltplan	47
6.4.1	Terminologie	47
6.4.2	Selektieren von Komponenten (<code>select_component()</code>)	48
6.5	Veränderung der Schaltplantopologie	52
6.5.1	Löschen von Komponenten (<code>delete_component()</code>)	52
6.5.2	Austauschen von Komponenten	52
6.5.3	Einfügen von Komponenten (<code>insert_component()</code>)	53
7	Die Designsprache und andere Programmiersprachen	62
7.1	Programmiermodell	62
7.2	Eigenschaften	62
7.3	Bedienung	63
7.4	Besonderheiten	64
8	Verarbeitung von Designwissen	65
8.1	Der Parser	65
8.2	Der Interpretierer	67
8.3	Der Lösungsraum	68
8.4	Kontrolle und Kontrollwissen	70
8.5	Blackboard-Architektur	71
8.5.1	Konzept des theoretischen Blackboard-Modells	71
8.6	Das spezielle Blackboard-Modell	74
8.6.1	Kontrollschleife	76
8.7	Die Verarbeitungskomponente	76
9	Zusammenfassung	79
10	Ausblick	81
A	Sprach-Referenz	85
A.1	Datentypen	85
A.1.1	<i>component</i>	85
A.1.2	<i>float</i>	85
A.1.3	<i>boolean</i>	85

A.1.4	Typen der Simulationsparameter	85
A.1.5	list	86
A.2	Operationen auf Datentypen	86
A.2.1	component	86
A.2.2	float	87
A.2.3	list	87
A.2.4	boolean	88
A.3	Vergleichsoperatoren	88
A.4	Hilfsvariablen	88
A.4.1	Beispiel zur Erzeugung einer Hilfsvariablen	88
A.5	Ablaufstrukturen	89
A.5.1	Schleifen	89
A.5.2	Bedingte Ausführung	89
A.6	Kernfunktionen	90
A.6.1	Der &-Operator	90
A.6.2	print()	90
A.6.3	exit()	90
A.6.4	select_component	90
A.6.5	delete_component	90
A.6.6	insert_component	91
A.7	Formale Sprachdefinition in BNF	91
B	Anwendungsbeispiele	95
B.1	Beispiele zur Veränderung der Schaltplantopologie	95
B.1.1	Einfügen einer Differentialschaltung	95
B.1.2	Einfügen einer Vorlaufdrosselung	96
B.1.3	Einfügen einer Nebenstromdrosselung	96
B.1.4	Einfügen einer Bypassdrossel	96
B.1.5	Einfügen eines Hydrospeichers	97
B.2	Beispiel einer Wissensklasse zur Kompensation von Fehlverhalten	98
B.2.1	Kraftentfaltung am Zylinder reicht nicht aus !	98

1 Einleitung

1.1 Motivation

Die Hydraulik befaßt sich im wesentlichen mit der Übertragung von Kraftwirkungen und Leistungen durch den statischen Druck der verwendeten Hydraulikflüssigkeit. Diese Aufgabe wird durch Hydrauliksysteme realisiert, welche im Wettbewerb mit mechanischen, elektrischen und pneumatischen Antrieben stehen. Gegenüber diesen Antrieben weisen Hydraulikantriebe viele Vorteile aber auch einige Nachteile auf ([Mann93]).

Die Planung und Konstruktion solcher hydraulischer Anlagen ist eine komplexe und schwierige Aufgabe. Aus diesem Grund wurde in der AG Wissensbasierte Systeme der Universität-Gesamthochschule Paderborn und der AG Meß-, Steuer- und Regelungstechnik der Universität-Gesamthochschule Duisburg das wissensbasierte System ArtDeco entwickelt, mit dem es unter anderem möglich ist, einen Hydraulikschaltplan auf einfache Weise zu zeichnen und anschließend zu simulieren. Bei dieser Simulation kann dann die prinzipielle Funktionalität der Anlage überprüft werden.

Bei der Planung mit ArtDeco müssen die einzelnen Komponenten einer Schaltung vom Ingenieur ausgelegt werden. D.h. er muß die Komponenten so dimensionieren, daß die Anforderungen an die Anlage erfüllt werden. Beispielsweise muß ein Zylinder bei gegebenen Pumpendruck eine bestimmte Größe besitzen, damit die zu entfaltende Kraft erreicht werden kann. Diese Auslegung erfordert vom Ingenieur ein hohes Maß an Erfahrung und Fachwissen.

Um dieses Fachwissen allen Anwendern von ArtDeco zur Verfügung zu stellen, wird in dieser Arbeit eine Designsprache entwickelt, mit deren Hilfe das Wissen zur Auslegung einer Hydraulikanlage formuliert werden kann.

Anhand dieses Wissens wird dann die computerunterstützte Auslegung realisiert.

Nachdem der Ingenieur die Anforderungen an eine Anlage vorgegeben hat, versucht das Verarbeitungsmodul der Designsprache, das ebenfalls in dieser Arbeit entwickelt wird, eine gültige Auslegung für die Bauteile der Anlage zu berechnen.

Auf diese Weise können gravierende Konstruktionsfehler sehr früh in der Konstruktionsphase erkannt werden und dann, so weit wie möglich, durch geeignete Modifikationen kompensiert werden. Diese Modifikationen beschränken sich auf Änderungen in der technischen Zeichnung der Anlage oder auf die Dimensionierung einzelner Komponenten.

Fehler, die erst nach Inbetriebnahme der Anlage aufgedeckt werden, erfordern konstruktive Veränderungen an der Anlage. Dann müssen Komponenten oder Komponentengruppen ausgetauscht werden und durch andere ersetzt werden. Dies zieht einen enormen finanziellen Aufwand nach sich, der für die Konstruktionsfirma Verluste bedeuten. Denn je eher ein Fehler bei der Entwicklung einer technischen Anlage erkannt und abgestellt werden kann, desto größer fällt der Gewinn für die Konstruktionsfirma aus.

Außerdem versucht das Verarbeitungsmodul der Designsprache eine nicht korrekt funktionierende Anlage zu „reparieren“. D.h., erfüllt die Anlage nicht die gestellten Anforderungen, „erkennt“ das Modul das entsprechende Fehlverhalten und versucht es, anhand des in der Designsprache formulierten Auslegungswissens, zu beheben. Voraussetzung dafür ist eine prinzipiell funktionierende Anlage.

Falls die Wissensbasis erweitert werden soll, muß eine Beschreibung des Fehlverhaltens mit den dazugehörigen Kompensationsvorschlägen in der Designsprache formuliert werden. Erst dann kann auf das Auftreten des Fehlers automatisch reagiert werden.

1.2 Ziele der Arbeit

Eines der beiden Ziele dieser Diplomarbeit ist die Entwicklung einer imperativen¹ Programmiersprache, mit der ein Hydraulikingenieur sein Fachwissen auf einfache und verständliche Weise formulieren kann.

In Gesprächen mit einem Hydraulikingenieur der Universität Duisburg wurde die Thematik der computerunterstützten Auslegung einer hydraulischen Anlage ausführlich diskutiert und analysiert.

Dabei hat sich herausgestellt, daß das Fachwissen verschiedene Fehlverhalten der Komponenten einer Anlage spezifiziert und Lösungen zur Kompensation dieser Fehler bereitstellt. Die Fehlerspezifikation wird vom Ingenieur in natürlicher Sprache formuliert und ist im allgemeinen zu informal, um von einem Computer direkt verarbeitet zu werden. Aus diesem Grund wird in dieser Arbeit eine formale Sprache entwickelt, mit deren Konzepten und Sprachkonstrukten eine informale Spezifikation formalisiert werden kann.

Wenn das Fachwissen über ein Fehlverhalten in der Programmiersprache formuliert wurde, kann der Computer mit geeigneten Verfahren und Modellen eine Verarbeitung realisieren. Er kann versuchen, die in der Wissensbasis aufgeführten Fehler aufzuspüren und anhand eines Lösungsvorschlages zu beheben.

In der Verarbeitung des in der Designsprache formulierten Fachwissens liegt das zweite Ziel dieser Arbeit. Es wird ein Modell entwickelt, mit dessen Hilfe die Verarbeitung des in der Programmiersprache formulierten Fachwissens effizient möglich ist.

Dabei simuliert die Verarbeitung eine gegebene Schaltung mit ArtDeco und überprüft, ob während der Simulation eine Komponente ein Fehlverhalten zeigt. Ist dies der Fall, werden die zu diesem Fehler gehörenden Lösungsvorschläge (Modifikationen) angewendet. Wenn der Fehler behoben werden konnte, d.h. die Komponente weist den Fehler nicht mehr auf,

¹Durch Ausführung von Zuweisungen können Werte von Variablen im Programmablauf verändert werden. Funktionen können aufgerufen oder als Parameter in anderen Funktionen benutzt werden. Funktionen als Resultat einer Funktion sind nicht erlaubt (\Rightarrow Laufzeitkellerbedingung wird verletzt!!). [Kast95]

wird die Simulation erneut gestartet, um weitere Fehler aufzuspüren und zu beheben. So fortfahrend werden im allgemeinen alle Fehler erkannt und behoben, so daß am Ende eine fehlerfreie Schaltung vorliegt.

Während dieses Prozesses kann die zur Kompensation eines Fehlers nötige Modifikation des Schaltplanes neue Fehler produzieren, die dann wieder behoben werden müssen. Auf diese Weise können Endlosschleifen entstehen, die die Verarbeitungskomponente erkennen muß (s. Kapitel 8). In diesem Fall kann nach Anwendung der Modifikation keine Lösung gefunden werden. Aus diesem Grund wird die Modifikation verworfen (Backtracking).

Mit der in dieser Arbeit entwickelten Sprache und der dazugehörigen Verarbeitung kann ein Hydraulikingenieur seine mit dem CAD-Modul von ArtDeco konstruierte Anlage weitestgehend automatisch auslegen lassen ohne selber die dafür notwendigen Berechnungen durchführen zu müssen.

1.3 Übersicht

Wir haben die Entwicklung einer Designsprache zu computerunterstützten Auslegung einer Hydraulikanlage motiviert und die Ziele dieser Diplomarbeit aufgezeigt.

In Kapitel 2 werden die verschiedenen Wissensformen, die ein Hydraulikingenieur zur Auslegung verwendet, analysiert. Diese Analyse deutet auf die verschiedenen Konzepte hin, die die Designsprache realisieren muß.

Die für die Diskussion erforderlichen Definitionen und Begriffe werden anhand von Beispielen in Kapitel 3 erläutert. Außerdem wird dort die prinzipielle Struktur einer hydraulischen Anlage untersucht.

Das Kapitel 4 beschäftigt sich mit der Einbindung der Designsprache in den Konstruktionsprozeß einer Hydraulikanlage. Zudem wird das Konzept der objektorientierten Wissensklassifizierung zur Wissensrepräsentation eingeführt. Am Ende dieses Kapitels wird eine Systemarchitektur angegeben, die die Komponenten des in dieser Arbeit entwickelten Expertensystems erklärt.

In Kapitel 5 werden die Sprachebenen der Designsprache vorgestellt. Wesentliche Ergebnisse dieses Kapitels sind Wissensklassen und Makros.

Die Anwendung der Designsprache wird in Kapitel 6 demonstriert. An einigen Beispielen werden verschiedene Situationen zur Veränderung von Simulationsparametern und zur Veränderung der Schaltplanstruktur behandelt. Diese Beispiele dienen dem Anwender der Sprache als Muster und weisen in den meisten Fällen auf Lösungen für seine speziellen Konstruktionsprobleme hin.

Die Einordnung der Designsprache zu anderen Programmiersprachen wird in Kapitel 7 diskutiert. Dabei werden Konzepte und Konstrukte anderer Programmiersprachen mit denen der Designsprache verglichen.

In Kapitel 8 wird eine Verarbeitungskomponente für die Designsprache entwickelt. Aufgrund des gewählten Modells (Blackboard-Modell) wird eine effiziente Verarbeitung des Quellco-

des der Wissensbasis möglich sein.

Die Arbeit schließt mit einer zusammenfassenden Beurteilungen der Ergebnisse dieser Arbeit (Kapitel 9) und einem Ausblick auf mögliche Erweiterungen der Designsprache (Kapitel 10).

2 Designwissen

In diesem Kapitel wird das Wissen des Hydraulikingenieurs, das sogenannte Designwissen, untersucht und strukturiert. Dabei werden Unterschiede und Zusammenhänge zwischen den verschiedenen Wissensformen diskutiert. Beispiele werden zeigen, welche Wissensformen in der Designsprache formulierbar sein müssen, und welche Konzepte dafür notwendig sind.

In dieser Arbeit bezeichnet der Begriff *Designsprache* eine formale Sprache zur Formulierung von Ingenieurwissen, welches zur Lösung eines Auslegungsproblems einer hydraulischen Anlage herangezogen wird.

2.1 Wissensformen

Das Wissen, welches zur Planung einer hydraulischen Anlage erforderlich ist, umfaßt nicht nur grundlegende Kenntnisse über hydraulische Systeme, sondern auch Erfahrungen und Heuristiken, mit deren Hilfe eine Hydraulikanlage geplant und ausgelegt werden kann. Dieses Wissen wurde anhand von wissenschaftlichen Arbeiten und praktischen Erfahrungen über einen geraumen Zeitraum gesammelt.

Möchte man das Designwissen in einer Wissensbasis zusammenfassen, erkennt man sehr schnell, daß es viele verschiedene Wissensformen gibt.

Wenn man bei der vollständigen Erfassung des Designwissens überhaupt eine Chance haben will, muß man es strukturieren und aufteilen. Hierzu werden Wissensgruppen eingeführt, in die das Designwissen eingeteilt wird. Außerdem dienen die Gruppenbezeichnungen als Diskussionsgrundlage für die weiteren Betrachtungen in diesem Kapitel.

2.1.1 Anforderungswissen

Die erste große Hauptgruppe, in die das Wissen zur Konstruktion hydraulischer Anlagen eingeteilt wird, heißt Anforderungswissen. Es beschreibt die Verschaltung der einzelnen Komponenten einer Anlage und formuliert die Anforderungen, die einzelne Komponenten oder die gesamte Anlage erfüllen müssen.

In [Vier97] wird der Begriff Anforderungen folgendermaßen verstanden:

„Unter den Anforderungen an einen (hydraulischen) Antrieb werden alle Angaben verstanden, die die Funktion, das Verhalten und die Ausgestaltung der Anlage beschreiben bzw. festlegen.“

Da sich dieses Wissen von Anlage zu Anlage unterscheidet, man denke da nur an einen Schaltplan, muß zwischen dem Anforderungswissen und dem Auslegungswissen (Abschnitt 2.1.2) eindeutig unterschieden werden.

Das Anforderungswissen kann noch in zwei weitere Untergruppen unterteilt werden. Die eine beschreibt die Verschaltung der Komponenten und die andere, wie sich die Komponenten zu verhalten haben.

Schaltplan

Die Art der Komponenten und deren Verschaltung wird graphisch in einem Schaltplan festgehalten. Hiermit ist dann die prinzipielle Funktion der hydraulischen Anlage gegeben. Sie kann mit ArtDeco simuliert werden, und wenn alle Komponenten aufeinander abgestimmt sind, wird sie auch ihre Funktion erfüllen.

Da die Komponenten im allgemeinen nicht aufeinander abgestimmt sind, wird mit Hilfe des Auslegungswissens (Abschnitt 2.1.2) eine gültige Auslegung bestimmt. Die Struktur eines

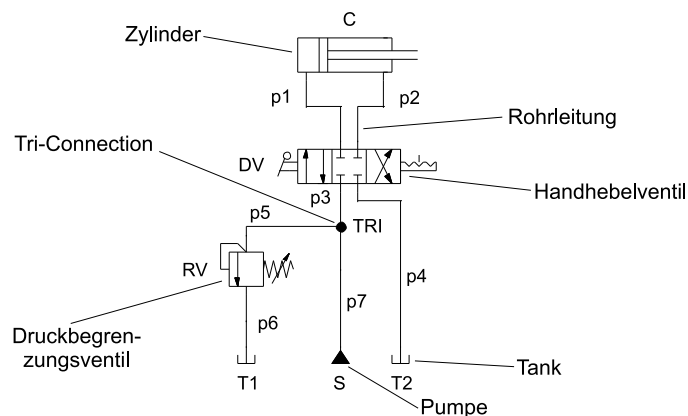


Abbildung 1: Beispiel eines Schaltplanes

Schaltplanes lässt sich noch weiter verfeinern. Er besteht aus Komponenten und Komponentengruppen, die ein bestimmtes Verhalten nach außen hin haben.

Die Abbildung eines Schaltplans im Rechner wird wie in [OFT97] beschrieben durch einen ungerichteten zusammenhängenden Graphen realisiert. Auf diesem Graphen können dann die bekannten Graphenalgorithmien und Varianten davon arbeiten. Diese Algorithmen sollen dann z.B. hydraulische Achsen und andere Strukturen auffinden. Die Abbildung 2 zeigt z.B. den zusammenhängenden Graphen des Schaltplanes in Abbildung 1.

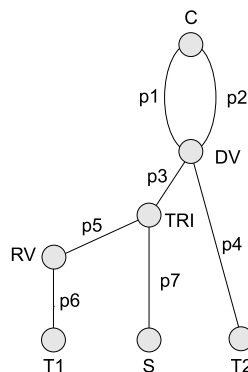


Abbildung 2: Der Graph eines Schaltplanes

Funktionsdiagramme

Eine mit Worten ausgedrückte Bewegungsfolge ist in vielen Fällen schwer verständlich und unvollständig, vor allem aber unübersichtlich. Dies gilt hauptsächlich bei schwierigen Bewegungsabläufen mehrerer Verbraucher, deren Bewegungsabläufe sich überschneiden. Die Diagrammdarstellung bietet daher dem Ingenieur die Möglichkeit, Bewegungsabläufe übersichtlich und einfach zu formulieren. Es lassen sich prinzipiell zwei Klassen von Funktionsdiagrammen unterscheiden. Zum einen sind das die *Wegdiagramme* und zum anderen die *Zustandsdiagramme*.

In einem *Wegdiagramm* wird das Zusammenwirken der Komponenten eines Schaltkreises über der Zeit dargestellt. An dem Beispiel einer Kunststoffpresse ([Mann93]) soll die Mächtigkeit von Diagrammen verdeutlicht werden. Das Diagramm ist in Abbildung 3 dargestellt.

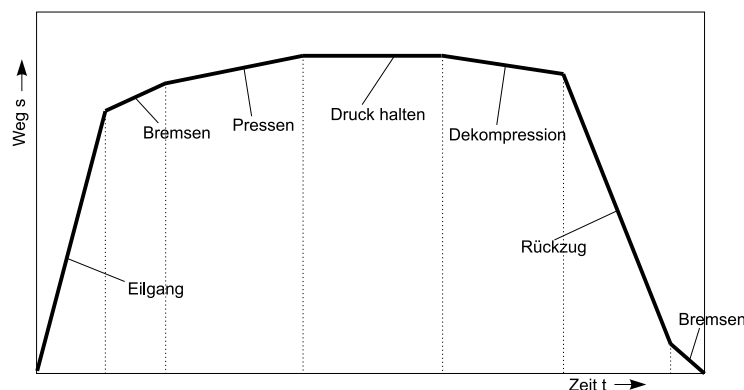


Abbildung 3: Weg-Zeit-Diagramm ([Mann93])

Hier bedient man sich des Weg-Zeit-Diagrammes. Der Bewegungsablauf sieht wie folgt aus:

Der Preßstempel fährt im Eilgang, also in möglichst kurzer Zeit an das Preßgut heran, wird abgebremst und wirkt dann zeitlich festgelegt im Preßgang mit einer ansteigenden Kraft auf die Kunststoffmasse. Nach Erreichen einer bestimmten Kraft soll die Hubbewegung gestoppt, die Kraftwirkung jedoch kurzzeitig aufrechterhalten werden. Nach Beendigung des Aushärtens erfolgt die kontrollierte Dekompression, der Preßstempel fährt in seine Ausgangslage zurück. Es tritt dann eine definierte Pause ein, die für den Auswerfvorgang und das Einbringen des neuen Preßgutes erforderlich ist.

Zustandsdiagramme enthalten die Funktionsfolge der betrachteten Arbeitseinheiten als Bewegungsdiagramm und ihre steuerungstechnische Verknüpfung. In Abbildung 4 ist das

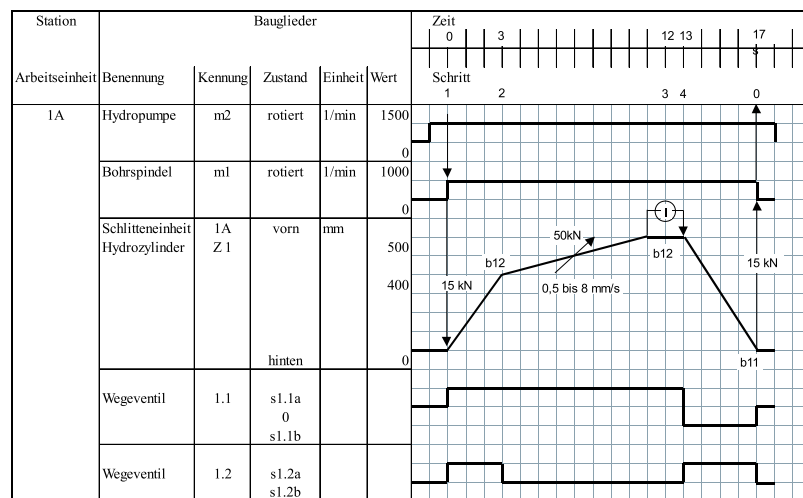


Abbildung 4: Zustandsdiagramm eines Hydrosystems ([Mann93])

Zustandsdiagramm der Kunststoffpresse des Beispiels dargestellt.

Randbedingungen

Zur Untergruppe der Randbedingungen gehören Anforderungen, die sich auf die Systemumgebung beziehen. Die Umgebungstemperatur, Öltemperatur, Luftdruck und alle äußeren Einflüsse, die das Systemverhalten beeinflussen können, zählen zur Gruppe der Randbedingungen. Sie müssen bei der Konstruktion einer Anlage berücksichtigt werden, damit die Anlage während des Betriebes den äußeren Systemeinflüssen standhält und weiterhin einwandfrei funktioniert.

2.1.2 Auslegungswissen

Die zweite Hauptgruppe, in die das Wissen zur Konstruktion und Auslegung hydraulischer Anlagen eingeteilt wird, heißt Auslegungswissen.

Zu ihr gehört das eigentliche Designwissen. Es bezieht sich nicht speziell auf ein bestimmtes Problem, sondern gilt allgemein, d.h. es gilt für jede zu konstruierende Anlage. In dieser Gruppe wird das Wissen des Ingenieurs mit Hilfe von *Regeln*, *Tabellen*, *Formeln*, und *Berechnungsfolgen* (Iterationen) abgebildet. Anhand dieses Wissens kann dann der Rechner, unter Verwendung geeigneter Verfahren und Modelle, die notwendigen Schritte einleiten, um eine gegebene Anlage auszulegen.

Im weiteren werden nun die Untergruppen des Auslegungswissens vorgestellt. Anhand von Beispielen wird erklärt, wie das Wissen prinzipiell aussieht und wie es formuliert werden kann. Die genaue Implementierung wird später in Kapitel 4 diskutiert. An dieser Stelle wird nur angedeutet wie eine mögliche Implementierung aussehen kann.

Bezeichnung der Maßnahme	Wirksamkeit	Rückwirkung	Aufwand	Art des Eingriffs
Anhebung des Versorgungsdruckes	9	5	3	K/P
Absenkung des Tankdruckes	3	5	8	K/P
Vergrößerung der Zylinderflächen	9	2	5	K

Abbildung 5: Beispiel einer Tabelle, mit der Erfahrungswissen formuliert wird

Erfahrungswissen

Eine der zentralen Wissensgruppen ist die des Erfahrungswissens. Zu ihr gehören alle Informationen, die der Ingenieur aufgrund von Erfahrungen und Forschungsarbeiten über die Jahre gesammelt hat. Bei näherer Betrachtung erkennt man, daß sich das Erfahrungswissen auf verschiedene Art und Weise formulieren läßt.

Tabellen

In vielen Bereichen der Ingenieurwissenschaften haben Tabellen einen großen Stellenwert. Mit ihnen können Relationen übersichtlich und einfach dargestellt werden.

Der Hydraulikingenieur benutzt Tabellen, um z.B. die Ergebnisse seiner Versuchsreihen festzuhalten. Anhand dieser Versuchsreihen schließt er dann auf globale Aussagen bezüglich seiner Experimente und faßt diese ebenfalls wieder in Tabellen zusammen.

Ein Beispiel einer solchen Tabelle, wie wir sie in Kapitel 6 benutzen werden, ist in Abbildung 5 aufgeführt.

Die Informationen in Tabellen werden kompakt und übersichtlich dargestellt. Zweideutigkeiten sind aufgrund der eindeutigen Relationen ausgeschlossen. Insofern sind Tabellen zur Formulierung von Designwissen sehr gut geeignet.

Formeln und Abschätzungen

Bei der Auslegung von Hydraulikkomponenten werden häufig sogenannte Faustformeln verwendet, die aufgrund von Erfahrungen oder physikalischen Zusammenhängen annähernde Lösungen für die auszulegenden Komponenten liefern. Diese Faustformeln lassen sich durch mathematische Formeln und Abschätzungen formulieren.

Formeln und Abschätzungen drücken mathematische Zusammenhänge zwischen Parametern aus. Dabei werden durch Formeln exakte Werte und durch Abschätzungen annähernde Werte ermittelt.

Ein Beispiel soll verdeutlichen, daß eine Formel oder Abschätzung große Aussagekraft bezüglich der in ihr vorkommenden Parameter besitzt.

Soll die Baugröße eines Zylinders festgelegt werden, muß bei gegebener Kraft F und gegebenem Druck p die Zylinderfläche A bestimmt werden. Dies wird durch die Beziehung $A = \frac{F}{p}$ ausgedrückt. Wenn nun die Zylinderfläche verkleinert

werden soll, so ist aus der Formel ersichtlich, daß dann z.B. der Druck erhöht werden muß, damit die geforderte Kraft F erreicht werden kann.

Dieses einfache Beispiel soll nur andeuten, wie wichtig Formeln und Abschätzungen zur Formulierung von Designwissen sind.

Aus diesem Grund müssen in der Designsprache mathematische Funktionen zur Berechnung solcher Formeln zur Verfügung gestellt werden.

Iterationen (Algorithmen)

Die Formulierung von Formeln und Abschätzungen kann nicht immer durch einfache mathematische Funktionen verwirklicht werden. Häufig sind zur Berechnung von Werten Berechnungsfolgen oder Mehrstufenverfahren erforderlich. Als Beispiel sei hier die Lösung eines Gleichungssystems genannt, welches z.B. mit dem Mehrstufenverfahren von Gauß gelöst werden kann.

Die Formulierbarkeit solcher Algorithmen wird eine weitere Anforderung an die Designsprache sein. Die dafür notwendigen Konzepte werden in Kapitel 6 vorgestellt.

Heuristiken (Regeln)

Manchmal hat der Ingenieur spezielles Wissen darüber, wie ein Komponentenparameter in Abhängigkeit von Parametern anderer Komponenten auszusehen hat.

Stellt er z.B. fest, daß ein Zylinder bei gegebenen Pumpendruck nicht ausfährt, versucht er diesen Fehler durch Erhöhung des Pumpendrucks zu kompensieren. Diese Aussage kann wie folgt als Regel formuliert werden: „Falls der Zylinder nicht ausfährt, erhöhe den Pumpendruck.“

Die Formulierung solcher Regeln muß in der Designsprache berücksichtigt werden. Wie die Syntax solcher if-then-Anweisungen aussieht, wird in Abschnitt 6 genau beschrieben.

2.1.3 Änderungswissen

Bei der Diskussion mit dem Hydraulikingenieur hat sich herausgestellt, daß sich sein Wissen speziell auf die verschiedenen Komponenten, wie z.B. Zylinder, Ventile oder Leitungen in einer Schaltung bezieht. Er hat das Wissen darüber, was genau getan werden muß, um ein auftretendes Symptom ((Fehl-)Verhalten) einer Komponente abzustellen.

Er überprüft, ob ein Symptom an der Komponente auftritt, um dann darauf zu reagieren. Die Formulierung einer Änderungsmaßnahme kann umgangssprachlich wie folgt aussehen:

„Falls das spezielle Symptom an einer Komponente auftritt, reagiere darauf mit angemessenen Modifikationsmaßnahmen, die das Symptom abstellen.“

Wie an diesem Beispiel zu sehen ist, besitzt die Formulierung des Ingenieurs den Charakter einer if-then-Regel.

Falls das Symptom auftritt, reagiere mit einer Modifikationsmaßnahme.

Aus diesem Grund werden *Regeln* auch als Fundament für die Designsprache eingeführt. Mit ihnen kann die Vorgehensweise des Ingenieurs nachgeahmt werden.

Nachdem der Ingenieur ein Symptom an einer Komponente erkannt hat, versucht er dieses durch Anpassung der Komponente zu bekämpfen. D.h. er verändert die Parameter der Komponente so, daß das Symptom nicht mehr auftritt.

Kann er mit der Veränderung von Komponentenparametern nicht zum Ziel kommen, versucht er das Symptom durch Hinzunahme neuer Komponenten oder durch Änderung der Verschaltung von Komponenten in der Schaltung abzustellen. Diese strukturellen Schaltplanveränderungen besitzen in den meisten Fällen nicht erwünschte Rückwirkungen auf das Gesamtsystem. Auf dieses Problem wird in Kapitel 8 näher eingegangen.

An diesen Schilderungen wird klar, daß sich die Regeln die der Ingenieur zur Kompensation eines Symptoms formuliert, in zwei Klassen einteilen läßt. Das ist die Klasse der

- Regeln zur *Veränderung von Komponentenparametern* und die Klasse der
- Regeln zur *Veränderung der Schaltplanstruktur*.

Aus diesem Grund müssen in der Designsprache Konstrukte zur Verfügung gestellt werden, mit denen Simulationsparameter einzelner Komponenten verändert werden können (Kapitel 6) und mit denen Eingriffe in einen Schaltplan (Abschnitt 3.2) möglich sind. Mit den Eingriffen soll die Topologie der Hydraulikschaltung analysiert und verändert werden können.

Abbildung 6 soll den Zusammenhang der verschiedenen Wissensformen des Designwissens noch einmal verdeutlichen.

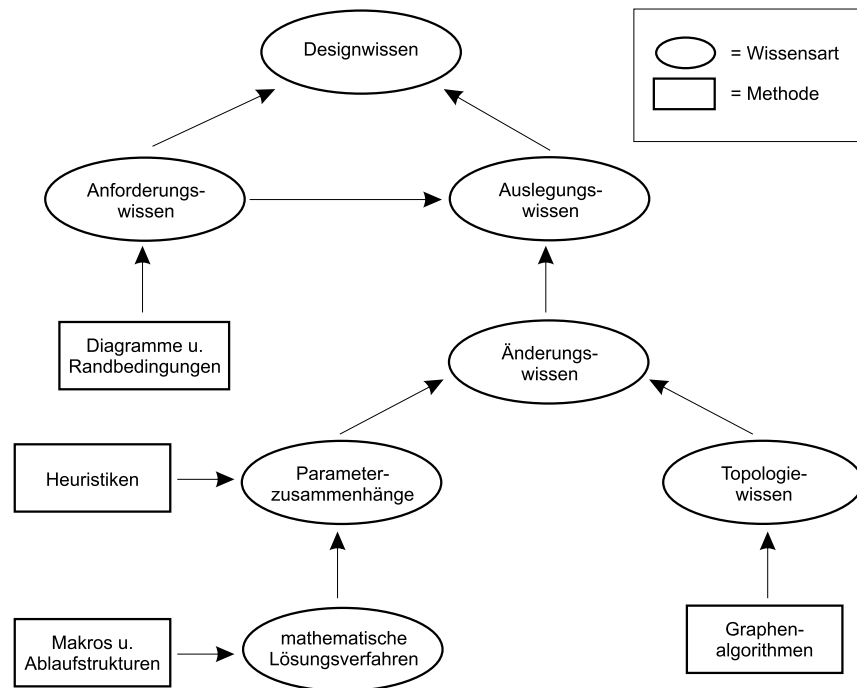


Abbildung 6: Zusammenhang zwischen den vorgestellten Wissensformen

3 Das Auslegungsproblem

In diesem Kapitel wird das Auslegungsproblem diskutiert und analysiert. An einigen Beispielen wird auf die Problematik des Auslegungsvorgangs eingegangen. Dabei werden grundlegende Begriffe auftauchen, die dann ausführlich erklärt werden.

Außerdem werden einige graphentheoretische Grundbegriffe definiert, mit deren Hilfe die Analyse der Schaltplantopologie vereinfacht wird.

Danach werden dann wichtige Erkenntnisse über die Struktur von Hydraulikschaltungen erarbeitet. Mit deren Hilfe kann dann der Zugriff auf die Komponenten einer Schaltung realisiert werden.

Am Ende dieses Kapitels werden Konzepte vorgestellt, die in die Designsprache aufgenommen werden und mit denen der Ingenieur den lesenden und schreibenden Zugriff auf einen Schaltplan bekommt.

3.1 Beispiele zur Neu- und Änderungsauslegung

Im folgenden soll kurz anhand von Beispielen das Auslegungsproblem erklärt werden. Dabei werden Begriffe auftauchen, die zum Verständnis der Problematik von großer Bedeutung sind. Auf dieser Diskussionsgrundlage aufbauend wird dann die Designsprache entwickelt und systematisch mit ihren Konzepten erklärt.

3.1.1 Komponenten

Bei der Konstruktion einer hydraulischen Anlage werden im ersten Schritt die Anforderungen an die Anlage definiert. Anhand dieser Anforderungen müssen die einzelnen Bauteile, die zur Verwirklichung der Anlage benötigt werden, ausgewählt werden. Diese Bauteile bezeichnen wir als Komponenten, die eine bestimmte Funktion realisieren. Beispiele solcher Komponenten sind in Abbildung 7 aufgeführt.

Die dort aufgeführten Komponenten übernehmen grundsätzlich verschiedene Funktionen in einer Schaltung. Erst das Zusammenspiel aller miteinander verschalteten Komponenten verwirklicht die gewünschte Funktionalität der gesamten Anlage.

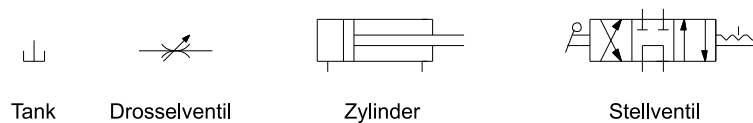


Abbildung 7: Verschiedene Komponententypen

3.1.2 Eigenschaften von Komponenten

Nachdem die Komponenten eines hydraulischen Kreislaufes ausgewählt und miteinander verschaltet wurden, müssen sie dimensioniert werden. Unter einer Dimensionierung versteht man die Form und Einstellungen einer Komponente. Hierbei wird strikt zwischen zwei verschiedenen Eigenschaften unterschieden. Das sind zum einen die Kenngrößen und zum anderen die einstellbaren Parameter einer Komponente.

Kenngrößen

Eine Parameterart der Komponenten sind ihre Kenngrößen. Sie legen die Bauform fest und werden vom Hersteller vorgegeben. Spezialanfertigungen sind natürlich auch möglich, aber wegen der hohen Herstellungskosten werden sie meist nicht eingesetzt. In solchen Fällen werden die Kenngrößen einer Komponente so gewählt, daß ihre Mindestanforderungen erfüllt werden.

Kenngrößen wurden in [Vier97] wie folgt definiert:

Definition: Kenngröße (einer Komponente)

„Kenngrößen charakterisieren die spezifischen Eigenschaften einer Komponente und bestimmen somit ihre Funktionsweise und ihr Verhalten. Der Wert einer Kenngröße ist abhängig von der technischen Realisierbarkeit und liegt nach Auswahl der entsprechenden Komponenten fest.“

Statische Kenngrößen sind z.B. der Hub S und die Wirkflächen A_K und A_R eines Differentialzylinders. In Abbildung 8 werden zwei Komponenten gleichen Typs gezeigt, die sich durch ihre statische Kenngröße Hub voneinander unterscheiden. Zu den dynamischen Kenngrößen zählen etwa die Eigenfrequenz ω_0 und die Dämpfung D eines Servoventils.

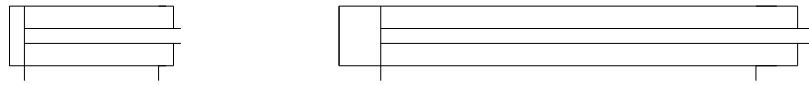


Abbildung 8: Zylinder, die sich durch ihre Kenngröße Hub voneinander unterscheiden

In der Designsprache werden Kenngrößen mit dem Schlüsselwort ***characteristic*** gekennzeichnet. Dieses Schlüsselwort unterstreicht, daß bei der Veränderung eines Simulationsparameters diesen Typs die Komponente ausgetauscht wird.

Parameter

Die Parameter einer Komponente wurden in [Vier97] wie folgt definiert:

Definition: Parameter (einer Komponente)

„Parameter besitzen weitgehend dieselben Eigenschaften wie Kenngrößen. Der Wert eines Parameters ist jedoch auch nach Auswahl der entsprechenden Komponente einstellbar, d.h. er kann im Rahmen seines Definitionsbereichs beliebig fest vorgegeben werden.“

Als Beispiel sei hier ein Drosselventil genannt, dessen Öffnungsgrad über einen Parameter auf 65% eingestellt werden kann.

In der Designsprache werden Einstellgrößen mit dem Schlüsselwort ***parameter*** gekennzeichnet. Dieses Schlüsselwort deutet an, daß bei der Veränderung des Wertes des Parameters keine strukturellen Änderungen in der Schaltung nötig sind, die Komponente wird nicht ausgetauscht.

Unterschied: ***parameter*** <-> ***characteristic***

Die Veränderung beider Parameterarten entspricht eigentlich nur der Veränderung von Simulationsparametern. Da in der Praxis aber ein gravierender Unterschied zwischen der Veränderung eines Einstellparameters und der einer Kenngröße besteht, bei der einen verändert sich der Schaltplan nicht und bei der anderen muß eine Komponente ausgetauscht werden, werden in der Designsprache diese Begriffe streng auseinandergehalten. Aus diesem Grund sind die Parametertypen ***parameter*** für Einstellgrößen und ***characteristic*** für Kenngrößen eingeführt worden.

3.1.3 Fehlverhalten

Nachdem der Schaltplan gezeichnet und die Dimensionierung (Auslegung) der Anlage erfolgt ist, kann sie auf ihre Funktionalität untersucht werden. Diese Funktionsprüfungen sind in der Realität teuer, da die Anlage dazu aufgebaut werden muß. Wird im Nachhinein festgestellt, daß es sich um eine Fehlkonstruktion handelt, zieht das einen enormen finanziellen

Schaden für die Konstruktionsfirma nach sich. Je früher ein Fehler während der Konstruktionsphasen entdeckt wird, um so geringer sind die Kosten und, um so höher ist der Gewinn für die Firma, die die Anlage baut. Aus diesem Grund wird eine Anlage so weit wie möglich im Rechner „konstruiert“ und simuliert.

Ein System, das sowohl das Zeichnen des Schaltplans unterstützt als auch eine Simulation ermöglicht, ist in ArtDeco implementiert. Hydraulikanlagen können so auf ihre prinzipielle Funktionalität überprüft werden. Mit ArtDeco können somit die hohen Kosten bei einer Fehlkonstruktion in Grenzen gehalten werden.

Wenn eine Anforderung an eine Anlage nicht erfüllt ist, sprechen wir von einem *Fehlverhalten*. Ein solches Fehlverhalten zieht das Nichtfunktionieren der gesamten Anlage nach sich. Wird ein Fehlverhalten festgestellt, muß durch geeignete konstruktive Maßnahmen darauf reagiert werden. Dies soll in Zukunft mit Hilfe des in dieser Arbeit entwickelten Zusatzmoduls von ArtDeco teilweise möglich werden.

Ein Beispiel eines typischen Fehlverhaltens ist das Nichterreichen der geforderten Ausfahrgeschwindigkeit eines Zylinders (s. Abbildung 9). Wird die Ausfahrgeschwindigkeit nicht

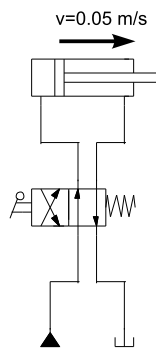


Abbildung 9: Das Fehlverhalten „Ausfahrgeschwindigkeit $v = 0,1 \frac{\text{m}}{\text{s}}$ wird nicht erreicht“

erreicht, ist sicher, daß das Verhalten der Anlage nicht den Anforderungen entspricht. Diesem Fehlverhalten kann durch verschiedene Modifikationen mit verschiedenen starken Auswirkungen auf das Gesamtsystem entgegengewirkt werden. Eine sehr einfache und mit Sicherheit nicht die beste Kompensationsmöglichkeit für das Beispiel in Abbildung 9 wäre die Erhöhung des Pumpendruckes. Ein solcher Kompensationsvorschlag kann mit der Designsprache formuliert werden und wird als *Modifikation* bezeichnet.

3.1.4 Modifikation

Wenn ein Fehlverhalten abgestellt werden soll, muß auf das Auftreten des entsprechenden Symptoms angemessen reagiert werden. Die dafür notwendigen Anweisungsfolgen werden in sogenannten *modification-Blöcken* aufgeschrieben. In einem modification-Block wird die notwendige Modifikation mit Konstrukten der Designsprache beschrieben. Die Beschreibung kann mit Hilfe von

- *Makros* (Funktionen),
- Wiederholungen (*Schleifen*) und
- bedingten Ausführungen (***if*** ... ***then***)

vorgenommen werden.

Da ein Fehlverhalten im allgemeinen durch mehrere verschiedene Modifikationen kompensiert werden kann, können in einer Regel mehrere modification-Blöcke hintereinander aufgeführt werden. Diese Blöcke sind in einer Reihenfolge aufgeschrieben, in der nach Erfahrung des Ingenieurs die Modifikation einen Erfolg verspricht.

3.1.5 Schaltplantopologie

Die Verschaltung der einzelnen Komponenten eines hydraulischen Schaltkreises stellt eine Topologie im graphentheoretischen Sinn dar. Aufgrund der Verschaltung der Komponenten kann eine Aussage über das Gesamtsystem getroffen werden. Daher ist die Analyse der Schaltplantopologie unerlässlich. Egal, ob es sich um das Finden von hydraulischen Achsen oder nur um das Lokalisieren einer bestimmten Komponente in der Schaltung handelt. Immer geht dem Ganzen eine Analyse der Schaltplantopologie voraus.

Im folgenden werden die graphentheoretischen Grundlagen für die Analyse der Schaltplantopologie ([OFT97]) definiert. Außerdem wird erklärt, wie eine Schaltung im Rechner als zusammenhängender Graph dargestellt wird.

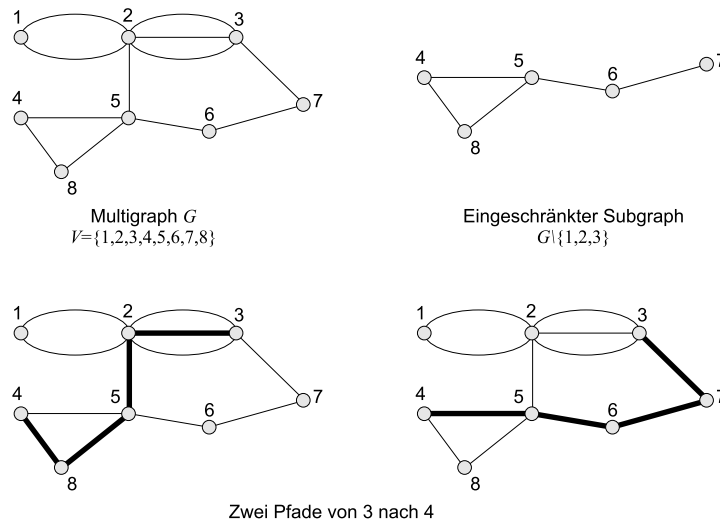
Definition:

1. Ein *Multigraph* G ist ein Tripel $\langle V, E, g \rangle$, wobei $V, E \neq \emptyset$ endliche Mengen sind, $V \cap E = \emptyset$ gilt und $g : E \rightarrow 2^V$ eine Abbildung mit $2^V = \{U \mid U \subseteq V, |U| = 2\}$ ist. V heißt Menge der Knoten, E heißt Menge der Kanten und g heißt inzidente Abbildung².
2. Ein Graph $H = \langle V_H, E_H, g_H \rangle$ heißt *Subgraph* von $G = \langle V, E, g \rangle$, falls $V_H \subseteq V, E_H \subseteq E$, und g_H eingeschränkt auf E_H ist. Ein Subgraph heißt eingeschränkter Subgraph von V_H , falls $E_H \subseteq E$ exakt die Kanten enthält, die inzident zu den Knoten von V_H sind. Für $T \subset V$ bezeichnet $G \setminus T$ den eingeschränkten Subgraph von $V \setminus T$.
3. Ein Tupel (e_1, \dots, e_n) heißt *Weg* von v_0 nach v_n , falls $g(e_i) = \{v_{i-1}, v_i\}$, $v_i \in V$, $i = 1, \dots, n$ gilt. Ein Weg heißt *Pfad*, falls die v_i paarweise verschieden sind. Ein Pfad kann anstelle eines Tupels von Kanten durch ein Tupel von Knoten (v_0, \dots, v_n) repräsentiert werden.

²Wir brauchen Multigraphen anstelle von Graphen, weil Komponenten einer hydraulischen Anlage parallel verschaltet sein können. Außerdem beschränken wir uns auf endliche Graphen.

4. G heißt *zusammenhängend*, falls es für je zwei Knoten $v_i, v_j \in V$ einen Weg von v_i nach v_j gibt. Falls G zusammenhängend ist und $G \setminus v$ nicht zusammenhängend ist, heißt v Schnittpunkt.
5. Ein Weg ist ein *Zyklus*, wenn er am Ausgangspunkt endet.
6. Ein Weg heißt *einfacher Zyklus*, wenn kein Knoten zweimal auf dem Pfad durchlaufen wird.

Abbildungung 10 veranschaulicht die Definitionen an einem Beispiel.



Abbildungung 10: Veranschaulichung der Graphendefinitionen ([OFT97])

Um mit einem hydraulischen Schaltkreis C wie mit einem gewöhnlichen Multigraphen $G(V, E, g)$ arbeiten zu können, benötigen wir eine Abbildungsvorschrift, die für den Kreislauf C den zusammenhängenden hydraulischen Graphen $G_h(C)$ definiert.

Definition: Zusammenhängender hydraulischer Graph

„Given is a hydraulic circuit C . Its related hydraulic graph $G_h(C) := \langle V_C, E_C, g_C \rangle$ is defined as follows. (i) V_C is a set; each non-pipe component of C is associated one-to-one with a $v \in V_C$, V_C does not contain other elements. (ii) E_C is a set; each pipe component of C is associated one-to-one with an $e \in E_C$, E_C does not contain other elements. (iii) $g : E_C \rightarrow 2^{V_C}$ is a function that maps e onto v_i, v_j , if there is a pipe between the components associated with v_i, v_j , and if e is associated with this pipe.“ ([OFT97])

Abbildungung 11 zeigt einen hydraulischen Kreislauf C und seinen zusammenhängenden hydraulischen Graphen $G_h(C)$. Die Bezeichnungen im Graphen sollen die eins-zu-eins Abbildung zwischen den Knoten des Graphen und den Komponenten des hydraulischen Kreislaufes verdeutlichen.

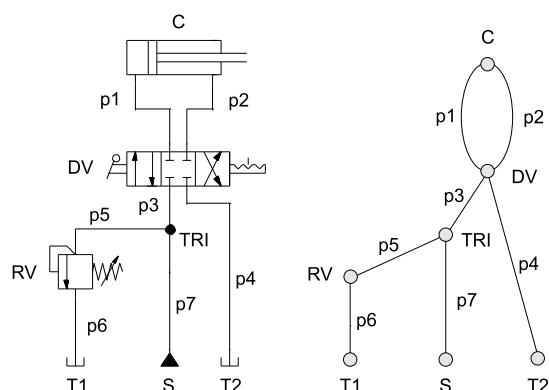


Abbildung 11: Beispiel eines Schaltkreises mit seinem zusammenhängenden Graphen ([OFT97])

Bemerkung: Zu jedem hydraulischen Kreislauf C gibt es genau einen zusammenhängenden Graphen $G_h(C)$.

Mit den in diesem Kapitel aufgeführten graphentheoretischen Grundlagen kann der Zugriff auf einen Schaltplan C , d.h. der Zugriff auf den Graphen $G_h(C)$ realisiert werden. Dieser Zugriff beschränkt sich, wie im folgenden Abschnitt beschrieben wird, auf das Selektieren, Löschen und Einfügen von Komponenten.

3.2 Prinzipieller Aufbau einer Schaltung

Wie wichtig die Schaltplantopologie ist, wurde in Abschnitt 3.1.5 angedeutet. In diesem Abschnitt geht es nun darum, sich in dem Schaltplan einer Anlage zurechtzufinden. Die Frage, die sich für die Entwicklung der Designsprache stellt ist:

Wie kann man Komponenten im Schaltplan lokalisieren, einfügen und entfernen, und wie sieht die Formulierung in der Designsprache aus ?

Um diese Frage zufriedenstellend beantworten zu können, muß man sich zuerst klar machen, welche Aktionen überhaupt auf einem Schaltplan möglich sind.

Überlegt man sich, wie eine technische Zeichnung einer Anlage zustandekommt, stößt man direkt auf zwei wesentliche Aktionsmöglichkeiten, die realisiert werden müssen. Das ist zum einen das

- *Einfügen von Komponenten* , und zum anderen das
- *Selektieren von Komponenten*

aus einem Schaltkreis. Die Selektion wird z.B. immer dann gebraucht, wenn zwei Komponenten mit einer Leitung miteinander verbunden werden sollen. Hat man die zu verbindenden Komponenten gefunden, braucht nur noch die Leitung an die richtigen Anschlüsse der Komponenten gelegt werden und man ist fertig.

Wenn während der Konstruktionsphase festgestellt wird, daß eine zuvor eingefügte Komponente nicht die richtige war oder nicht die Aufgabe erfüllen kann, die sie erledigen soll, so muß sie wieder gelöscht werden. Erst dann kann an die freiwerdende Stelle die richtige Komponente eingesetzt werden.

Im Laufe der Diskussion hat sich herausgestellt, daß es nur drei verschiedene Aktionsmöglichkeiten gibt, die zum Erstellen eines Schaltplanes benötigt werden. Das sind das *Einfügen*, *Selektieren* und *Löschen* von Komponenten.

Weil diese drei Aktionen für den Ingenieur ein wesentliches Hilfsmittel zur Veränderung einer Schaltung sind, müssen entsprechende Grundfunktionen in der Designsprache vorhanden sein. Diese Funktionen heißen

- `select_component()` für das Selektieren,
- `insert_component()` für das Einfügen und
- `delete_component()` für das Löschen

von Komponenten im Schaltplan.

Diese Aktionen geben *nicht* an, an welcher Stelle sie in der Schaltung angewendet werden sollen. Damit eine Aktion ausgeführt werden kann, muß also noch der Ort, wo sich die Komponente befindet, beschrieben werden. Diese Beschreibung wird den Funktionen als Parameterliste übergeben.

Als Ergebnis ist festzuhalten, daß sich ein Eingriff in einen Schaltkreis aus zwei verschiedenen Teilen zusammensetzt. Das ist zum einen die Aktion (*action*) die durchgeführt werden soll und zum anderen die Position (*location*) im Schaltplan wo diese Aktion durchzuführen ist. D.h., ein Eingriff in einen Schaltplan ist wie folgt definiert:

$$\boxed{\text{Eingriff} = \text{action-specifier}(\text{location-specifier})}$$

Die beiden Teile eines Eingriffs sind streng auseinanderzuhalten und werden von uns im weiteren als *action-specifier* für die Aktion und *location-specifier* für die Beschreibung der Position einer Komponente im Schaltplan bezeichnet.

3.2.1 action-specifier

Der *action-specifier* gibt die Aktion an, die auf dem Schaltkreis ausgeführt werden soll. Es werden drei verschiedene Aktionen benötigt, um alle Änderungsmaßnahmen an einer

Schaltung durchführen zu können. Die action-specifier sind in der Designsprache als spezielle Kernfunktionen³ implementiert:

```
select_component(location-specifier)
delete_component(location-specifier)
insert_component(location-specifier)
```

Als Parameter dieser Funktionen wird der location-specifier angegeben, der im folgenden Abschnitt beschrieben wird.

3.2.2 location-specifier

Mit dem location-specifier wird die Position in einem Schaltkreis beschrieben, an dem eine Aktion (s. Abschnitt 3.2.1) durchgeführt werden soll. Wie genau eine Beschreibung der Position aussehen kann und wie die Syntax in der Designsprache aussieht, wird jetzt diskutiert. Die in der Diskussion erarbeiteten Ergebnisse werden dann ausreichen, um alle notwendigen Modifikationsmöglichkeiten abzudecken.

Die erste Frage, die sich nun stellt, ist die, wie eine Komponente mit anderen Komponenten verschaltet sein kann. Die Diskussion mit dem Hydraulikingenieur hat ergeben, daß es zwei verschiedene Verschaltungsarten gibt. Das sind die

1. *serielle* und
2. *parallele*

Verschaltung von Komponenten.

in_series

Bei der Verschaltung von Komponenten in Reihe handelt es sich um einen Pfad von Komponenten, auf dem die betrachtete Komponente liegt. Ein Pfad in einem Graphen $G = (V, E)$ ist wie folgt definiert:

Definition: Pfad

Ein Tupel (v_1, \dots, v_n) heißt Pfad, falls $(v_i, v_{i+1}) \in E$, $i = 1, \dots, n - 1$ gilt und alle v_i paarweise verschieden sind.

Das Beispiel in Abbildung 12 zeigt z.B. den Pfad (*Pumpe, Stellventil, Zylinder*). Pfade sind Listen von hintereinander verbundenen Komponenten. Möchte man eine dieser Komponenten auf dem Pfad auswählen, kann man dies mit der **select_component**-Funktion tun, wobei als Parameter das Schlüsselwort **in_series** angegeben werden muß. Mit dem **in_series** Schlüsselwort geht immer die Angabe eines Pfades einher. Dieser Pfad wird durch seine erste (*first*) und letzte (*last*) Komponente definiert. Der Satz

³Kernfunktionen werden in Abschnitt 5.1.2 erklärt

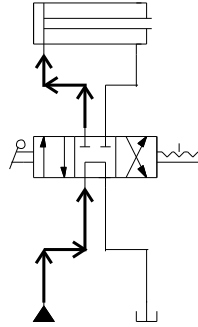


Abbildung 12: Pfad in einer Schaltung

`select_component(in_series Pumpe Zylinder)`

würde gerade den Pfad (*Pumpe*, *Stellventil*, *Zylinder*) liefern.

in_parallel

Die zweite Art und Weise, wie eine Komponente oder ein Pfad in einer Schaltung verschaltet sein kann, ist die Parallelschaltung. Der Begriff Parallelschaltung wird hier folgendermaßen definiert:

Definition: (Parallelschaltung)

C sei ein hydraulischer Kreislauf und $P_1 := (A_1, \dots, A_n)$, $n \geq 1$, $P_2 := (B_1, \dots, B_m)$, $m \geq 1$ seien Pfade in C . Dann gilt:

- (i) Der **Pfad** P_1 ist parallel zum Pfad P_2 geschaltet, falls ein einfacher Zyklus $(v_{B_1}, \dots, v_{B_m}, v_{A_1}, \dots, v_{A_n}, v_{B_1})$ in $G_h(C)$ existiert. Dabei assoziiert v_{A_i} zur Komponente A_i auf dem Pfad P_1 und v_{B_j} zur Komponente B_j auf dem Pfad P_2 .
- (ii) Eine **Komponente** K ist parallel zum Pfad P_2 geschaltet, falls $K = A_l$, $1 \leq l \leq n$ gilt.

Bemerkung: Es ist darauf zu achten, daß der Pfad P_1 nur am Anfang oder Ende des Pfades P_2 beginnen darf.

In Abbildung 13 wird die Definition an einem Beispiel verdeutlicht. In diesem Beispiel liegen der Zylinder $B1$ und das Drosselventil $A2$ auf einem einfachen Zyklus $(B1, A1, A2, A3, B1)$ und sind somit parallel zueinander geschaltet.

Das Druckbegrenzungsventil $V2$ liegt auf einem Abzweig und kann deshalb auf keinem einfachen Zyklus liegen. Es ist also zu keiner anderen Komponente in der Schaltung parallel geschaltet.

Da mit der Designsprache auf parallel geschaltete Komponenten zugegriffen werden muß, wird das Schlüsselwort **in_parallel** eingeführt, das gerade diesen Zugriff erlaubt. Wie beim **in_series**-Schlüsselwort muß auch hier ein Pfad mit angegeben werden, zu dem die Komponente, auf die zugegriffen werden soll, parallel geschaltet ist.

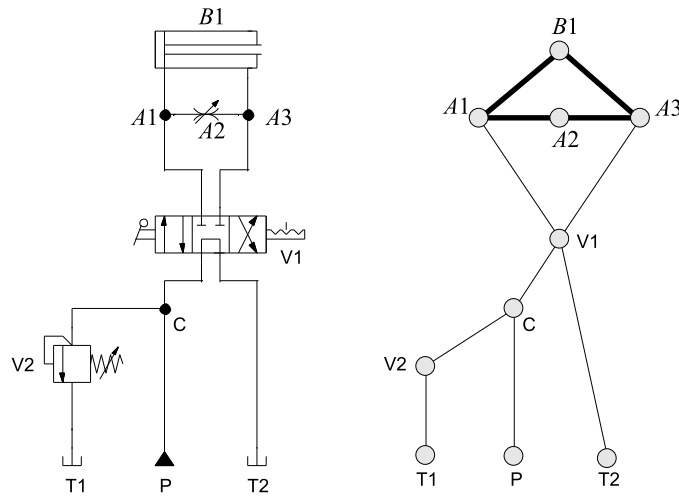


Abbildung 13: Parallelgeschaltete und nicht parallelgeschaltete Komponenten

Möchte man z.B. das zum Zylinder in Abbildung 13 parallel geschaltete Drosselventil A2 selektieren, so kann man dies mit dem folgenden Satz tun:

```
select_component(in_parallel B1 B1)
```

In diesem Beispiel wird aber gleichzeitig das Stellventil mit selektiert, weil es ebenfalls zum Zylinder parallelgeschaltet ist. Außerdem ist an diesem Beispiel zu sehen, daß Pfade der Länge 1 definiert werden können. Mit der Länge eines Pfades ist die Anzahl der auf ihm vorkommenden Komponenten gemeint.

Eine weitere Bemerkung ist zu den Parameterlisten (location-specifier) der `select_component`, `insert_component` und `delete_component`-Funktionen zu machen. Sie sind nicht fest vorgegeben und können wie beim WHERE-Konstrukt in SQL beliebig kombiniert werden. D.h., die Reihenfolge und Anzahl der Parameter ist nicht vorgegeben. Es müssen nur die in Abschnitt 3.2.3 beschriebenen Konventionen eingehalten werden.

Pfade zwischen Anschlüssen (Tri-Connections)

Der Ingenieur betrachtet häufig Pfade, die an einer Tri-Connection beginnen und evtl. an einer Tri-Connection enden. Ein Beispiel ist der Pfad (A1, A2, A3) in Abbildung 13.

Da Tri-Connections in einem hydraulischen Kreislauf C , wie Komponenten, durch Knoten im hydraulischen Graphen $G_h(C)$ repräsentiert werden, können diese wie gewöhnliche Komponenten behandelt werden. D.h., sie können selektiert, gelöscht und eingefügt werden. Für den Pfad des Beispiels in Abbildung 13 bedeutet das, daß zuerst die Tri-Connections A1 und A3 selektiert werden müssen. Anschließend kann der Pfad zwischen A1 und A3 mit `select_component(in_series A1 A3)` selektiert werden. Die Selektion von Pfaden zwischen Anschlüssen muß somit in zwei Schritten erfolgen:

1. Selektiere die erforderlichen Tri-Connections.
2. Selektiere den Pfad zwischen den Tri-Connections.

Damit solche Anweisungsfolgen nicht wiederholt im Quellcode der Wissensbasis aufgeschrieben werden müssen, gibt es in der Designsprache das Makrokonzept, das die Zusammenfassung von Anweisungsfolgen erlaubt (s. Abschnitt 5.1.2). Ein solches Makro kann zum Beispiel die Selektion von Pfaden zwischen Tri-Connections realisieren. Dann können durch den Aufruf dieses einen Makros, beide Selektionsschritte durchgeführt werden. Der Quellcode wird dadurch wesentlich übersichtlicher und lesbarer.

after / before

In vielen Fällen hat der Ingenieur konkretes Wissen darüber, wo eine Komponente in der Schaltung vorkommen muß. Er weiß z.B., daß eine Komponente direkt vor oder hinter einer anderen liegen muß. Um diese relativ konkreten Angaben formulieren zu können, werden für den location-specifier die Konstrukte **after** und **before** zugelassen. Das Beispiel in Abbildung 14 soll die Funktionalität der Schlüsselwörter **after** und **before** verdeutlichen. Möchte man z.B. das Drosselventil *D2* selektieren, so kann man dies mit dem Satz

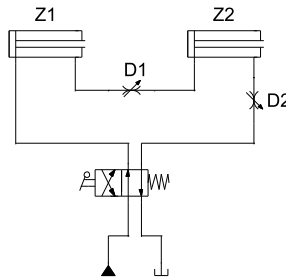


Abbildung 14: Komponenten **after** / **before** anderen Komponenten

```
select_component(in_series Pumpe Tank , after Z2)
```

realisieren. Die Syntax der **before** -Variante sieht genauso aus und liefert die Komponente direkt vor einer anderen.

Weil aufgrund des Ventils zwei verschiedene Pfade von der Pumpe zum Tank existieren,

$((Pumpe, Ventil, Z1, D1, Z2, \mathbf{D2}, Ventil, Tank)$ und
 $(Pumpe, Ventil, D2, Z2, \mathbf{D1}, Z1, Ventil, Tank))$

wird gleichzeitig die Komponente *D1* mitselektiert. Hierdurch ergeben sich in diesem Beispiel zwei Möglichkeiten, die beide betrachtet werden müssen und somit bei der Verarbeitung Laufzeit kosten.

Da die Schaltstellung (Durchlaßrichtung) des Ventils implizit den zu verfolgenden Pfad

vorgibt, reduziert sich dieses Problem wieder auf einen eindeutigen Pfad. Dies muß bei der Implementierung der Funktionen berücksichtigt werden, die einen Eingriff in den Schaltplan realisieren (s. Abschnitt 6.4).

Die Abbildung 15 soll den Sachverhalt noch einmal verdeutlichen. Falls in einer Situation

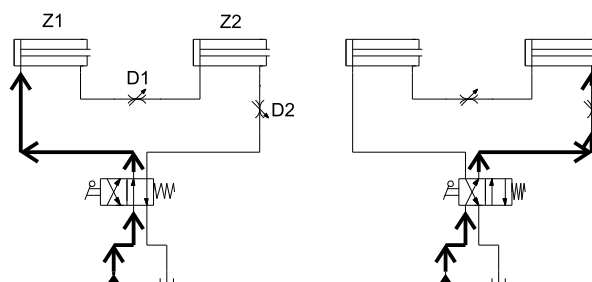


Abbildung 15: Abhängigkeit der Pfade von der Schaltstellung eines Ventils

nicht klar sein sollte welche Richtung⁴ ein Pfad hat, wird die Komponente sowohl vor als auch hinter der angegebenen Komponente ausgewählt. In Abbildung 16 würden z.B. mit

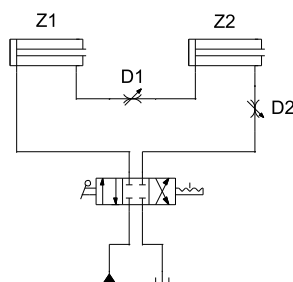


Abbildung 16: Nicht eindeutige Pfade

`select_component(in_series Pumpe Tank, after Z2)` beide Drosselventile *D1* und *D2* ausgewählt.

Ähnlich verhält es sich mit dem Satz `select_component(after Z2)`. Auch hier werden beide Drosselventile *D1* und *D2* ausgewählt.

In diesem Beispiel wird noch einmal angedeutet, daß für den location-specifier einige der hier aufgeführten Schlüsselwörter kombiniert werden können. Die formale Syntax des location-specifiers wird in Anhang A.7 angegeben.

Komponentenanschlüsse

Der Ingenieur verfügt häufig über so detaillierte Informationen, wie und wo die Anschlüsse

⁴Die Richtung eines Pfades *first last* ist die Reihenfolge der Komponenten, in der der Pfad von *first* nach *last* durchlaufen wird.

Typ	location-specifier	Beschreibung
1	in_series <i>first last</i>	Den Pfad von Komponente <i>first</i> nach Komponente <i>last</i> betrachten. Dabei dürfen <i>first</i> und <i>last</i> durch konkrete Komponenten, aber auch durch Qualifiers angegeben werden.
1	in_parallel <i>first last</i>	Alle Pfade betrachten, die parallel zum Pfad <i>first last</i> geschaltet sind. Dabei dürfen <i>first</i> und <i>last</i> durch konkrete Komponenten, aber auch durch Qualifiers angegeben werden.
2	(<i>qualifiers</i>)	Dieser location-specifier wird nur im Zusammenhang mit select_component verwendet. Er gibt an, welche Bedingungen die zu betrachtenden Komponenten erfüllen müssen.
3	after	Die Position direkt hinter der Komponente <i>component</i> betrachten. Die Komponente <i>component</i> darf auch durch Qualifiers beschrieben werden.
3	before	Die Position direkt vor der Komponente <i>component</i> betrachten. Die Komponente <i>component</i> darf auch durch Qualifiers beschrieben werden.

Tabelle 1: Die möglichen location-specifiers

In diesem Beispiel überlegt man sich zuerst, welche der action-specifiers man benutzen will. Weil wir eine Komponente auswählen wollen, müssen wir die select-Funktion benutzen. Dann muß man sich überlegen, ob die Komponente im Schaltplan in Serie, parallel oder zwischen zwei Anschlüssen verschaltet ist. Im Beispiel sei die Komponente in Serie mit der Pumpe und dem Zylinder geschaltet. Zuletzt muß man sich überlegen, wie man den Ort der Komponente noch genauer spezifizieren kann. Da wir wissen, daß die Komponente in unserem Beispiel direkt hinter dem Zylinder liegt, geben wir diese Einschränkung mit **after** im location-specifier an. Der vollständige Satz lautet dann:

```
select_component(after Zylinder, in_series Zylinder Pumpe)
```

Mit dieser Anweisung wird dann die Komponente direkt hinter dem Zylinder auf

dem Pfad zur Pumpe selektiert.

Weitere Beispiele zum Gebrauch des action- und location-specifiers werden im Kapitel 6 Anwendung der Designsprache ausführlich diskutiert.

Wie bei der Zusammenstellung eines Eingriffs grundsätzlich vorgegangen werden muß, läßt sich wie folgt als Algorithmus aufschreiben:

1. Wähle einen action-specifier aus der Menge {**select_component()**, **delete_component()**, **insert_component()**} aus.
2. Falls es sich um den action-specifier **select_component()** oder **delete_component()** handelt, verfare wie folgt:
 - (a) Falls die Komponente in Serie oder parallel verschaltet ist, wähle entsprechend einen Spezifizierer aus der Menge {**in_series**, **in_parallel**} aus.
 - (b) Falls die Komponente direkt vor oder hinter einer Komponente liegt, wähle den entsprechenden Spezifizierer aus der Menge {**after**, **before**} aus.
 - (c) Falls eine Komponente mit bestimmten Eigenschaften gemeint ist, formuliere einen entsprechenden Qualifier.
3. Falls es sich um den action-specifier **insert_component()** handelt, verfare wie folgt:
 - (a) Falls die Komponente in Serie oder parallel verschaltet werden soll, wähle entsprechend einen Spezifizierer aus der Menge {**in_series**, **in_parallel**} aus.
 - (b) Falls die Komponente direkt vor oder hinter einer Komponente liegt, wähle den entsprechenden Spezifizierer aus der Menge {**after**, **before**} aus.

Die formale Definition der Syntax für den location-specifier und für die Kernfunktionen **select_component()**, **delete_component()** und **insert_component()** wird in Form einer Backus-Naur-Form (**BNF**) in Anhang A.7 gegeben.

4 Die Designsprache

Einer der wesentlichen Punkte bei der Entwicklung eines Expertensystems ist die Art der Formulierung des Expertenwissens und die geeignete Zusammenfassung in einer Wissensbasis. Dabei muß darauf geachtet werden, daß die Formulierung unmißverständlich und eindeutig ist. In Kapitel 2.1 wurde gezeigt, daß beim Entwurf einer hydraulischen Anlage mehrere verschiedene Wissensformen vorkommen, die auf adäquate Weise beschrieben werden müssen. Hierzu werden in diesem Kapitel Konzepte für die Designsprache entwickelt, mit denen eine Formulierung möglich sein wird.

4.1 Einbindung der Designsprache in den Konstruktionsprozeß

Bevor man sich weitere Gedanken über die Sprache selbst machen kann, muß klar sein, wie sie in den Konstruktionsprozeß einer hydraulischen Anlage eingebunden werden kann. Dabei werden dann Zusammenhänge zwischen den verschiedenen Konstruktionsstufen und der Sprache aufgedeckt und erläutert. Anhand des Ablaufdiagramms in Abbildung 18 wird nun kurz geschildert, wie der Konstruktionsprozeß im allgemeinen aussieht, und wie die Designsprache eine Auslegung unterstützt.

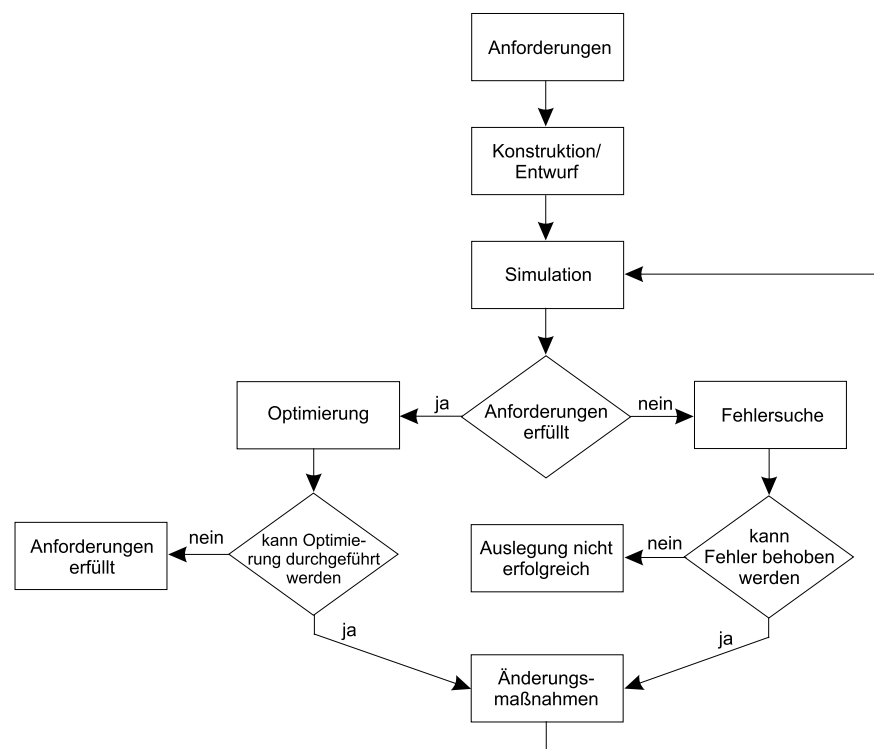


Abbildung 18: Einbindung der Designsprache in den Konstruktionsprozeß

4.1.1 Anforderungen

Am Anfang einer Konstruktion werden die Anforderungen an die zu konstruierende Anlage definiert. Diese Systemanforderungen gehören zur Gruppe des Anforderungswissens und wurden im Abschnitt 2.1 erläutert.

4.1.2 Konstruktion/Entwurf

Nachdem die Anforderungen definiert wurden, kann ein Schaltplan entworfen werden, dessen Komponenten die gestellte Aufgabe prinzipiell erledigen können. Dieser Schaltplan gehört ebenfalls in die Gruppe des Anforderungswissens (Abschnitt 2.1).

4.1.3 Simulation

Mit der im Grobentwurf vorliegenden Anlage kann eine Simulation des Schaltkreises gestartet werden. Diese Simulation wird immer dann angehalten, wenn sich der Zustand mindestens einer Komponente ändert. In diesen definierten Pausen können dann die Simulationsparameter der einzelnen Komponenten abgefragt werden. Weil die Werte der Simulationsparameter den Zustand der einzelnen Komponenten beschreiben, können Symptome, die Fehlverhalten oder spezielle dynamische Verhalten (Optimierung) spezifizieren, festgestellt werden.

An dieser Stelle setzt nun die Designsprache an. Mit ihrer Hilfe werden solche Symptome formuliert und durch Modifikationen, die Parameterveränderungen oder strukturelle Veränderungen im Schaltplan bewirken, abgestellt.

4.1.4 Fehlersuche

Sind die Anforderungen an die Anlage nicht erfüllt, kann der aufgetretene Fehler, sofern er der Wissensbasis bekannt ist, mittels Fehlersuche aufgedeckt werden. Tritt ein nicht bekanntes Fehlverhalten auf, endet die Auslegung mit einem nicht funktionierenden hydraulischen Schaltkreis. D.h., der Schaltkreis kann bezüglich der vorliegenden Wissensbasis nicht automatisch ausgelegt werden.

4.1.5 Änderungsmaßnahmen nach Fehler

Wenn ein Symptom eines Fehlers erkannt wurde, können die notwendigen Änderungsmaßnahmen zum Beheben des Fehlers eingeleitet werden. Da in einem Schaltplan mehrere Komponenten gleichzeitig einen Fehler aufweisen können, muß darauf geachtet werden, daß bei der Verarbeitung des Designwissens diese Fehler auch gleichzeitig behoben werden können. Dies steigert zum einen die Effizienz des Verfahrens und entspricht zum anderen auch eher dem menschlichen Vorgehen. Allerdings dürfen nur voneinander unabhängige

Komponenten gleichzeitig verändert werden. Aber hierzu mehr im Kapitel 8 über die Verarbeitung von Designwissen.

Nachdem die Änderungsmaßnahmen durchgeführt wurden, kann durch Neustarten der Simulation der Verarbeitungszyklus geschlossen werden. Durch eine erneute Überprüfung des Schaltkreises kann dann festgestellt werden, ob die Anlage fehlerfrei ist oder nicht.

4.1.6 Optimierung

Eine funktionierende Anlage muß nicht gleichzeitig eine optimal funktionierende Anlage implizieren. Darum bietet sich an dieser Stelle eine Optimierung der Anlage an, die das dynamische Verhalten weiter verbessern kann.

Ist in der Wissensbasis ein Optimierungsvorschlag vorhanden, kann im allgemeinen durch entsprechende Änderungsmaßnahmen eine Optimierung erreicht werden. Danach muß die Simulation neu gestartet werden, um zu überprüfen, ob die Änderung geglückt ist. Hat sie widererwarten einen neuen Fehler zur Folge, wird sie zurückgenommen (Backtracking). Aber auch hierzu mehr im Kapitel 8 über die Verarbeitung von Designwissens.

Falls keine Optimierungsvorschläge aus der Wissensbasis angewendet werden können, endet die Auslegung mit einem der Wissensbasis entsprechenden, optimal ausgelegten Schaltkreis.

4.2 Entwicklungskriterien

Eine Sprache zur Formulierung von Designwissen muß bestimmten Anforderungen genügen, damit sie effektiv eingesetzt werden kann. Dabei muß darauf geachtet werden, daß sie nicht zu unübersichtlich, aber trotzdem so mächtig gemacht wird, daß der Ingenieur sein Expertenwissen einfach und ohne großen Aufwand formulieren kann. Außerdem muß darauf geachtet werden, daß die Sprache Konzepte anbietet, mit denen der Wissensakquisiteur das Wissen strukturieren kann. Die Sprache soll nicht, wie man es von den üblichen Programmiersprachen her kennt, Low-Level Funktionen enthalten, die den Wissensakquisiteur dazu verleitet, Fehler in die Wissensbasis einzubauen. Z.B. führt die falsche Anwendung der Zeigerarithmetik in C häufig zu Programmabstürzen. Solche Fehler lassen sich auch mit einem Debugger nur schwer aufdecken. Verbreitete Programmiersprachen wie C, C++, Pascal oder Ada erfüllen diese Eigenschaften nicht ([Koch97]).

Da die Grundaufgaben der Designsprache bekannt sind, nämlich die Formulierung von Designwissen, können die Konstrukte der Sprache so darauf abgestimmt werden, daß mit Hilfe dieser Grundfunktionen alle notwendigen Modifikationen durchführbar sind, ohne daß sich der Wissensakquisiteur Gedanken darüber machen muß, wie diese Funktionen implementiert sind. Für ihn ist allein das Ergebnis einer Funktion von Interesse.

4.3 Wissensrepräsentation

Grundvoraussetzung für die computerunterstützte Auslegung eines hydraulischen Kreislaufes ist eine adäquate Beschreibung des Designwissens sowie ein effizientes Verfahren, welches

das Wissen verarbeitet. Die Architektur, die im Laufe dieser Arbeit entwickelt wird, soll beide Voraussetzungen zufriedenstellend erfüllen. Im folgenden wird der Aufbau der Wissensbasis und ihre Einbindung in das Gesamtsystem erläutert. Zur Veranschaulichung ist in Abbildung 19 der erste Entwurf einer Systemarchitektur dargestellt.

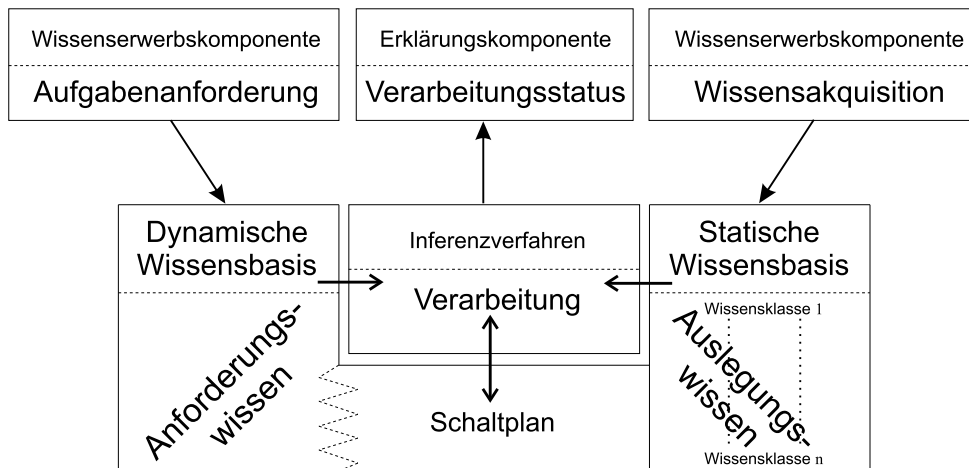


Abbildung 19: Architektur zur Repräsentation und Verarbeitung von Designwissen

4.3.1 Dynamisches Wissen

Die gesamte Wissensbasis teilt sich in einen dynamischen und in einen statischen Teil auf. In der dynamischen Wissensbasis befindet sich das Wissen, welches sich von Anlage zu Anlage ändert. Es handelt sich hierbei um das Anforderungswissen, das in Abschnitt 2.1 beschrieben wurde.

Die Aufgabenanforderungen werden durch eine eigenständige Systemkomponente der Wissensbasis hinzugefügt. Mit ihr wird das problemspezifische Wissen so aufbereitet, daß es den Konventionen der Wissensbasis entspricht.

Außerdem gehört der Schaltplan zur dynamischen Wissensbasis, der im Gegensatz zum restlichen Anforderungswissen während der Verarbeitung verändert werden kann.

4.3.2 Statisches Wissen

In der statischen Wissensbasis befindet sich das eigentliche Designwissen. Es ist ein Abbild des Expertenwissens zur Auslegung hydraulischer Systeme und wird bei der Wissensakquisition in der Designsprache formuliert und als ASCII-Datei in die Wissensbasis aufgenommen. Bei dem statischen Wissen handelt es sich ausschließlich um das Auslegungswissen, auf das im Abschnitt 2.1 eingegangen wurde.

Beim Wissenserwerb ist nach [Pupp91] zwischen drei verschiedenen Arten zu unterscheiden:

1. *Indirekter Wissenserwerb*: Ein Wissensakquisiteur befragt einen Experten und fügt dessen Wissen in das System ein.
2. *Direkter Wissenserwerb*: Ein Experte formuliert sein Wissen selbst.
3. *Automatischer Wissenserwerb*: Das Analysesystem extrahiert das Wissen aus Falldaten und Literatur.

Für die Designsprache müssen die Varianten 1. und 2. besonders berücksichtigt werden. D.h., die Sprache wird so konzipiert, daß sowohl ein Hydraulikexperte als auch eine andere Person, Wissen formulieren kann. Die Alternative 3. wird im Moment noch nicht unterstützt, kann aber bei der Weiterentwicklung der Sprache berücksichtigt werden (s. Kapitel 10).

4.3.3 Wissensklassen

Ein Hauptziel der Designsprache ist die Strukturierung des Wissens derart, daß es vom Wissensakquisiteur leicht überschaut werden kann. Weil sich das Auslegungswissen immer auf bestimmte Objekte wie z.B. einzelne Komponenten oder Komponentengruppen bezieht, bietet sich die direkte Zuordnung dieses Wissens zu den einzelnen Objektklassen geradezu an. Daher wird an dieser Stelle der Begriff der objektorientierten Wissensklassifizierung eingeführt. Bei dieser Art der Klassifizierung wird das Wissen über eine Klasse von gleichartigen Objekten in einer speziellen Wissensklasse mit dem Namen der Objektklasse zusammengefaßt. Dieses Wissen kann dann auf allen instanziierten Objekten der Klasse angewendet werden.

Insofern ist das Wissen in einer Wissensklasse mit den Methoden einer Klasse in einer objektorientierten Programmiersprache wie z.B. Smalltalk, Eiffel oder C++ zu vergleichen ([Kast95]). Lediglich das mächtige Konzept der Vererbung wird im Moment noch nicht berücksichtigt. Die Problematik der einfachen und mehrfachen Vererbung in objektorientierten Systemen wurde von uns im Rahmen einer Projektgruppenarbeit ([Schl93]) auf dem Gebiet der wissensbasierten Systeme behandelt und im Bezug auf die Designsprache als zu anspruchsvoll für den Anwender der Designsprache empfunden. In späteren Versionen kann dieses Konzept aber durchaus mit übernommen werden.

Definition: Objektorientierte Wissensklassifizierung

Wird das Wissen über eine Klasse von gleichartigen Objekten in einer speziellen Wissensklasse zusammengefaßt und kann dieses Wissen auf allen instanziierten Objekten der Objektklasse angewendet werden, sprechen wir von einer objektorientierten Wissensklassifizierung.

Für den Aufbau der statischen Wissensbasis bedeutet das, daß sie sich aus mehreren voneinander unabhängigen Wissensklassen zusammensetzt (s. Abbildung 19). Jede von ihnen wird einer Komponentenklasse aus der Komponentenbibliothek zugeordnet. D.h., das Wissen der Wissensklasse bezieht sich auf Instanzen der Komponentenklasse.

Der Vorteil dieses Vorgehens liegt auf der Hand. Soll neues Designwissen in die Wissensbasis aufgenommen werden, muß nur die Wissensklasse der Komponente erweitert werden, auf die sich das Wissen bezieht, alle anderen bleiben unberührt.

Genauso verhält es sich bei der Wartung der Wissensbasis. Ist die Komponente bekannt, für die eine Änderung erforderlich ist, braucht auch nur die Wissensklasse dieser Komponente verändert zu werden. Die Suche der entsprechenden Stelle in der Wissensbasis beschränkt sich somit nur auf diese eine Klasse.

Ein weiterer Vorteil der Aufteilung des Wissens in Klassen besteht darin, daß bei der Erweiterung der Komponentenbibliothek der Hersteller der neuen Komponente gleichzeitig die dazugehörige Wissensklasse mitliefert, mit deren Hilfe gängige Fehlverhalten der Komponente behoben werden können, oder gezielte Optimierungen durchgeführt werden können.

In der Abbildung 19 ist die Architektur zur Repräsentation und Verarbeitung von Designwissen dargestellt. Außer den Eingabekomponenten und der Wissensbasis ist noch die Verarbeitungskomponente aufgeführt. Diese und andere Zusatzkomponenten werden in Kapitel 8 entwickelt und erklärt. Hier reicht es zu wissen, daß es eine Verarbeitung gibt, die anhand des Designwissens versucht, einen Schaltkreis automatisch auszulegen. Dabei liefert sie während der Verarbeitung Informationen über den Verarbeitungsstatus (Erklärungskomponente).

5 Schichten der Designsprache

In diesem Kapitel wird der Aufbau der Designsprache beschrieben. Dazu gehören die grundlegenden Konzepte *Wissensklasse* und *Makros*. Mit beiden kann das Wissen auf verschiedenen Abstraktionsebenen formuliert werden.

Dabei wird erklärt, wie eine Wissensklasse aufgebaut ist, und welche Bedeutung die verwendeten Schlüsselwörter haben.

Danach wird das Makrokonzept mit den Kernfunktionen als Basis beschrieben.

5.1 Die Sprachpyramide

In Abschnitt 2.1 über Wissensformen wurde deutlich, daß ein Ingenieur zur Formulierung des allgemeinen Auslegungsproblems hydraulischer Anlagen Wissen verschiedener Abstraktionsebenen verwendet. Daher wird die Designsprache aus drei aufeinander aufbauenden Schichten zusammengesetzt. Die dadurch entstehende Sprachpyramide ist in Abbildung 20 dargestellt. Die Spitze der Sprachpyramide soll die Sprachkonstrukte zur Verfügung stellen,



Abbildung 20: Schichten der Designsprache

mit denen der Ingenieur das Auslegungsproblem beschreibt. Die Beschreibung kann mit Hilfe einfacher Regeln, wie in Abschnitt 2.1.3 beschrieben, vorgenommen werden.

In der mittleren Schicht werden Makros definiert, die komplexe Aufgaben erledigen und die von einem Spezialisten, dem Wissensakquisiteur, implementiert werden.

Das Fundament der Pyramide ist die Schnittstelle zwischen der Verarbeitung der Designsprache und dem ArtDeco Kern.

5.1.1 Domänennahe Problemformulierung

Daß die Formulierung von Designwissen nahe an der Problemstellung liegen muß, ist eine Grundvoraussetzung für die Designsprache. Sie muß außerdem leicht erlernbar und verständlich sein. Daher wird für die oberste Schicht der Sprachpyramide das Konzept der objektorientierten Wissensklassifizierung (Abschnitt 4.3.3) angewendet. D.h. es wird für


```

class Name_der_Komponentenklasse{
  gates{
    gate1;
    ...
    gatei;
  }
  parameters{
    var1 type1;
    ...
    varm typem;
  }
  repair_rule(p1){
    symptoms{(S11),..., (S1k)}
    modification{
      ...
    }
    ...
    modification{
      ...
    }
  }
  repair_rule(pn){
    symptoms{(S1n),..., (Snl)}
    modification{
      ...
    }
  }
  usw.
}

```

Abbildung 21: Prinzipieller Aufbau einer Wissensklasse

jede Komponentenklasse eine Wissensklasse eingeführt, die das Auslegungswissen in Form von Regeln formuliert. An einem Beispiel soll der Aufbau einer Wissensklasse erläutert werden.

Wissensklasse

Das Gerüst einer Wissensklasse ist in Abbildung 21 dargestellt. Sie wird immer mit dem Schlüsselwort **class** eingeleitet gefolgt von dem Namen der Komponentenklasse, auf die sich die Regeln der Wissensklasse beziehen. Dieser Name ist eindeutig und ist durch den Typ der Komponente der Klasse vorgegeben.

Dann werden nach dem Schlüsselwort **gates** die Namen der Anschlüsse der Komponente angegeben. Über diese Namen können dann die Gates direkt angesprochen werden.

Danach wird die Liste der Parametervariablen der Komponente mit den dazugehörigen Typen aufgeführt. Typen können **characteristic** oder **parameter** sein.

Erst jetzt werden die eigentlichen Regeln ihrer Priorität nach aufgeführt. D.h., die Regelpriorität nimmt in der Wissensklasse von oben nach unten ab.

Eine Regel wird durch ihren **repair_rule**-Teil eingeleitet. Die von der Sprache vorgegebenen Prioritäten für eine Regel sind **strict**, **possible** und **welcome**. Hierbei bedeutet **strict**, daß bei Beobachtung eines Symptoms auf jeden Fall Handlungsbedarf besteht. **possible** bedeutet, daß die Regel nach Möglichkeit angewendet werden sollte, und **welcome** bedeutet, daß eine Anwendung der Regel gewünscht aber nicht gefordert wird. Weiteres hierzu in Kapitel 8 über die Verarbeitung von Designwissen.

Kommen mehrere Regeln mit gleicher Priorität in einer Klasse vor, legt die Reihenfolge, in der sie aufgeführt sind, die genaue Priorität fest. D.h. die Regel, die an erster Stelle steht, wird bei der Verarbeitung als erstes berücksichtigt, und so weiter. Die daraus folgende starre Verarbeitungsstrategie kann durch explizite Vergabe von Prioritäten an die Regeln aufgebrochen werden. Der in dieser Arbeit entwickelte Prototyp der Designsprache unterstützt diese Möglichkeit nicht. In Kapitel 10 werden weitere Vorschläge zur Vergabe von Regelprioritäten diskutiert.

Als nächstes folgt in der Definition einer Regel ihr Bedingungsteil. In den Klammern der **symptoms**-Anweisung stehen dann die mit Komma getrennten Symptome, die ein Fehlverhalten oder ein spezielles dynamisches Verhalten beschreiben. Die Symptome können mit Operatoren verknüpfte boolsche Ausdrücke oder Makros sein, und stellen somit eine Formel im aussagenlogischen Sinn dar. Können bei der Verarbeitung mehrere Regeln gleichzeitig angewendet werden, ergibt sich durch die Prioritäten der Regeln eine eindeutige Reihenfolge, nach der sie beim Backtracking angewendet werden (s. Kapitel 8).

modification

Hinter dem Bedingungsteil (**symptoms()**) einer Regel werden die Modifikationen aufgelistet, die eine Kompensation des Fehlverhaltens, das durch die Symptome beschrieben wurde, bewirken können. Jede der in den Anweisungsblöcken aufgeführten Modifikationsmöglichkeiten kann den Fehler beheben, so daß eine Auswahlmöglichkeit besteht. Auch die Aktionen in den Regeln sind deswegen nach ihrer Priorität geordnet aufgeschrieben und werden beim Backtracking in der so vorgegebenen Reihenfolge angewendet. Wie genau die Regeln und Modifikationen verarbeitet werden, wird in Kapitel 8 beschrieben.

5.1.2 Makros

Mit dem Modifikationsteil einer Regel soll angemessen auf die Beobachtung eines Symptoms reagiert werden. Bei einer Modifikation werden die zur Kompensation des Fehlers notwendigen Handlungen in Form von Befehlen aufgeführt. Da diese Handlungsfolgen im allgemeinen komplex und für den Anwender zu unübersichtlich sind, werden sie zu Makros zusammengefaßt, die einen eindeutigen Namen mit dazugehöriger Parameterliste bekommen. Anhand des Namens und der Parameterliste des Makros soll die Wirkung auf den Schaltplan ersicht-

```

macro Name(type1 par1, ..., typen parn) {
  :
  return (ergebnis);
}

```

Abbildung 22: Skelett einer Makro-Definition

lich sein. Damit kann der Ingenieur auf einfache Weise komplexe Operationen durchführen ohne den gesamten Code der Operation selber schreiben zu müssen. Außerdem braucht er auch nicht unbedingt verstehen wie das Makro implementiert ist. Allein die Wirkung ist für ihn von Bedeutung.

Zur Verdeutlichung wird in Abbildung 22 das Skelett eines Makros aufgeführt. Der Aufbau eines Makros ist sehr einfach. Es wird mit **macro** eingeleitet, gefolgt von seinem eindeutigen Namen. Danach wird in Klammern die Parameterliste aufgeführt. Da in der Designsprache keine polymorphen Funktionen⁵ erlaubt sind, muß zu jedem Parameter sein Typ mit angegeben werden. Andernfalls kommt es mit Sicherheit zu Mißverständnissen, die von vornherein ausgeschlossen werden sollen.

In einem Makro können die von der Sprache vorgesehenen Konstrukte und Ablaufstrukturen verwendet werden. Auch andere Makros können aufgerufen werden. Rekursive Aufrufe werden hingegen nicht unterstützt, da sie zu weit von der Problemstellung entfernt sind. Falls ein Problem doch elegant mit einem rekursiven Makro gelöst werden kann, wird es sich aufgrund der Struktur des Schaltplanes im allgemeinen um ein endrekursives Makro handeln. Dies kann nach [Kast95] in eine iterative Schleife umgewandelt werden, die dieselbe Aufgabe erfüllt. Nachdem der Makrorumpf erfolgreich ausgeführt wurde, wird das Ergebnis mit **return()** dem aufrufenden Programmbereich übergeben.

An dieser Stelle soll ein konkretes Beispiel aufgeführt werden, daß alle Komponenten auf einem Pfad selektiert, die einen einstellbaren hydraulischen Widerstand besitzen. Als Ergebnis soll das Makro eine Liste der Komponenten zurückgeben.

```

macro get_resistors(component f, component l) {
// Holt alle Komponenten, die einen einstellbaren hydraulischen Widerstand haben.
  h := select_component(in_series f l, (LEVEL >= 0));
  return(h);
}

```

⁵Der Typ des formalen Parameters einer polymorphen Funktion wird erst zur Laufzeit gebunden. Daher arbeiten solche Funktionen auf verschiedenen Typen [Kast95].

```

macro name1(...){
...
}
:
macro namen(...){
...
}
class Name_der_Komponentenklasse{
  gates{
    ...
  }
  parameters{
    ...
  }
  repair_rule(p1){
    symptoms{(S11), ..., (S1k)}
    ...
  }
  usw.
}

```

Abbildung 23: Definition von Makros in einer Wissensklasse

Die Zusammenstellung von *Makrobibliotheken* zählt zu einer der wesentlichen Vorteile des Makrokonzeptes. In einer Bibliothek kann der Wissensakquisiteur die häufig verwendeten Makros sammeln und nach Bedarf in die Wissensbasis einbinden. Durch Hinzufügen neuer Makros in die Bibliothek kann die Sprache auf der Anwenderebene leicht erweitert werden. Weil die Implementierung von Makros ein wesentliches Stück von der Problemstellung entfernt ist, wird dies einem Wissensakquisiteur überlassen, der die Sprache genau kennen muß. Da die Programmierung von Makros der von üblichen Programmiersprachen wie C oder Pascal ähnelt, eröffnen sich flexible und mächtige Möglichkeiten für den Wissensakquisiteur.

Im Moment werden in der Designsprache aufgrund der kleinen und überschaubaren Wissensbasis noch keine externen Makrobibliotheken unterstützt. Dies läßt sich aber leicht realisieren und kann in spätere Versionen eingebaut werden.

In der jetzigen Version der Designsprache werden alle Makros der Wissensbasis global am Anfang der Wissensbasis definiert und stehen somit allen Wissensklassen zur Verfügung. Makrodefinitionen müssen in der Wissensbasis vor der ersten Klassendefinition stehen (s. Abbildung 23). Dann können sie in allen Klassen benutzt werden.

Kernfunktionen

Im Laufe der Arbeit wird sich herausstellen, daß einige Grundfunktionen low-level, d.h. in einer maschinennahen Programmiersprache wie C programmiert werden müssen, damit sie effizient arbeiten. Hierzu gehören zum Beispiel Makros, die einen Zugriff auf den Schaltplan erlauben und Makros, die die Kommunikation mit dem Anwender ermöglichen. Diese Makros werden als fester Bestandteil in die Sprache aufgenommen und können nicht verändert oder gelöscht werden. Dies bleibt allein den Entwicklern neuer Versionen der Designsprache vorbehalten.

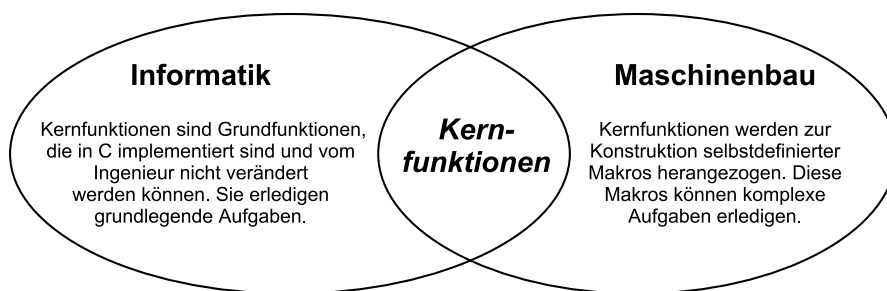


Abbildung 24: Schnitt der Entwickler- und Anwenderebene

Der Wissensakquisiteur kann aus den Kernfunktionen Makros entwickeln, die dann wiederum in neue Makros eingebunden werden können. D.h., ein Makro setzt sich aus

- *Kernfunktionen*,
- *Ablaufstrukturen* und
- *Makros*

zusammen (s. Abbildung 24).

Der Begriff „Kernfunktion“ wurde bewußt für die fest in der Sprache integrierten Funktionen gewählt, weil sie wesentliche und grundlegende Aufgaben erledigen. Als Beispiele seien hier die mathematischen Funktionen *sin()*, *cos()* und *tan()*, genannt.

Das Vorgehen, aus kleinen Grundfunktionen komplexe Funktionen zu konstruieren, ist höheren Programmiersprachen wie C und Pascal abgeschaut worden. Dort hat sich das Funktionskonzept bewährt. Aus diesem Grund wurde eine spezielle Variante, das Makrokonzept, in die Designsprache aufgenommen.

Selbstdefinierte Makros

Wie wir in Abschnitt 5.1.2 gesehen haben, können aus Kernfunktionen selbstdefinierte Makros entwickelt werden, die dann komplexe Aufgaben erledigen.

Aus den dann vorhandenen Makros können dann wieder neue Makros entwickelt werden, die dann noch komplexere Aufgaben erledigen und so fort.

So wird im Laufe der Zeit ein Pool von Makros entstehen, mit dem der Ingenieur seine verschiedenen Auslegungsprobleme einfach und komfortabel formulieren kann.

In Abbildung 25 wird der Wachstumsprozeß des Makropools noch einmal veranschaulicht. Wie beim Aufbau von C++ Bibliotheken muß auch beim Erweitern des Makropools darauf

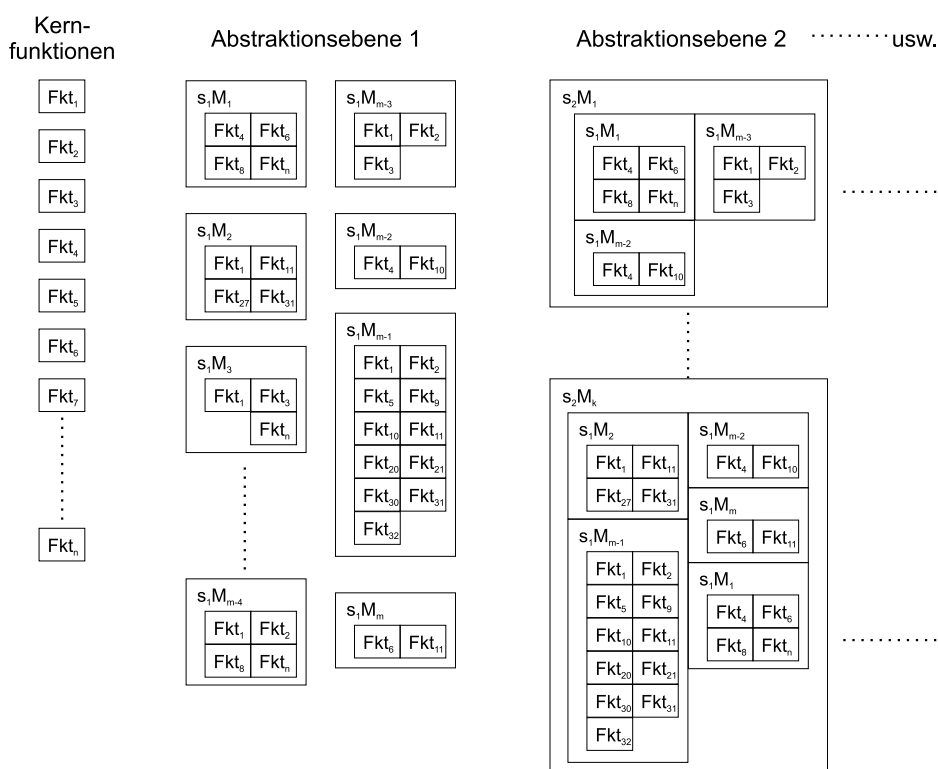


Abbildung 25: Makrohierarchie

geachtet werden, daß die verschiedenen Makros nicht zu unübersichtlich werden. D.h., jedes Makro sollte so implementiert werden, daß es aus Makros einer Abstraktionsebenen tiefer aufgebaut ist. Komplexe Befehlsfolgen sollten immer in eigenen Makros implementiert werden. Häufig können sie auch von anderen Makros verwendet werden, so daß sich dann auch automatisch die Länge eines neuen Makros reduziert und sich der Code der Wissensbasis in Grenzen hält. Die so aufgebaute Makrohierarchie bietet dem Wissensakquisiteur zahlreiche Möglichkeiten und kann aufgrund der Struktur leicht überschaut werden.

Qualifiers als formaler Parameter („&“)

In Kapitel 3.2 haben wir gesehen, daß Komponenten in einer Schaltung durch Qualifiers der Form $var_1 op_1 val_1, \dots, var_n op_n val_n$ beschrieben werden können. Da es für einen Qualifier keinen Variablentyp in der Designsprache gibt, aber Qualifiers trotzdem als aktueller Parameter in einem Makroaufruf angegeben werden können, müssen sie durch den „&“-Operator in der Makrodefinition kenntlich gemacht werden. Ein Qualifier darf nur

```

macro waehle(&qualifier){
    ergebnis:=select_component((qualifier));
    return(ergebnis);
}

```

Abbildung 26: Qualifiers in Makros

als letztes Argument in einem Makro vorkommen. Der Grund dafür liegt in seiner unbekannten Länge, wobei die Länge durch die Anzahl n der spezifizierenden Ausdrücke $var_1\ op_1\ val_1, \dots, var_n\ op_n\ val_n$ definiert ist. Je länger ein Qualifier ist, desto genauer ist die Komponente spezifiziert, die mit dem Qualifier gemeint ist. Erkennt der Interpretierer, der in Kapitel 8 beschrieben wird, den „&“-Operator, so liest er alle folgenden Ausdrücke bis zum Ende des Makrokopfes und kennt anschließend die Länge des Qualifiers.

In der Abbildung 26 ist ein Makro angegeben, das bei dem Aufruf *waehle(comp_type=pump-unit, P_LIM>= 50)* alle Komponenten liefert, die den Qualifier erfüllen. Dieses Makro ist eine eins zu eins Übersetzung der *select_component*-Funktion eingeschränkt auf den Qualifier-Parameter.

5.1.3 Schnittstelle zum ArtDeco-Kern

Die Repräsentation und Verarbeitung des Designwissens ist in eigenständigen Modulen implementiert. Die Kommunikation des Verarbeitungsmoduls mit dem Graphik- und Objektsystem von ArtDeco wird über eine Schnittstelle abgewickelt.

Mittels Interpretierer werden die Befehle und Ablaufstrukturen einer Wissensklasse abgearbeitet. Sobald auf den Schaltplan zugegriffen wird, kommen die Befehle der Schnittstelle ins Spiel. Mit ihnen können z.B. Parameterwerte der einzelnen Komponenten im Schaltplan gelesen und geschrieben werden. Strukturelle Veränderungen, wie das Bewegen von Komponenten, das Austauschen von Komponenten und das Ziehen von Leitungen sind weitere Manipulationsmöglichkeiten. Zudem kann das Verarbeitungsmodul über Steuerungsfunktionen der Schnittstelle die Simulation eines Schaltkreises starten, anhalten und stoppen.

6 Anwendung der Designsprache

Dieses Kapitel beschäftigt sich mit einigen Kernfunktionen, die für die Veränderung von Simulationsparametern und der Veränderung der Schaltplantopologie benötigt werden.

An zahlreichen Beispielen werden häufig auftretende Situationen zur Veränderung der Schaltplantopologie behandelt. Diese kann der Ingenieur als Vorlage für seine Problemstellungen benutzen. In den meisten Fällen wird er dann einen Lösungsansatz für sein spezielles Formulierungsproblem finden. Andernfalls lassen sich aus Kombinationen mehrerer Beispiele Lösungen erarbeiten.

6.1 Wissensklassen

In Kapitel 5 wurde die Struktur von Wissensklassen vorgestellt. Daher kann eine Implementierung einer neuen Wissensklasse leicht durch einfaches Ausfüllen des Klassenskeletts (Abbildung 21 S. 38) vorgenommen werden.

In Anhang B sind ausführliche Beispiele zur Entwicklung von Wissensklassen aufgeführt. An diesen Beispielen kann auch das prinzipielle Vorgehen beim Schreiben neuer Wissensklassen erlernt werden.

6.2 Makros

Daß Funktionen und Makros wichtige Konzepte einer imperativen Programmiersprache sein können, zeigen die üblichen Programmiersprachen wie C, C++ und Pascal.

Die in der Designsprache definierten Makros können wie in Abschnitt 5.1.2 beschrieben benutzt werden. Rekursion wird von der Verarbeitung der Designsprache nicht unterstützt.

An dieser Stelle muß darauf hingewiesen werden, daß die Kernfunktionen für den Eingriff in die Schaltplantopologie von der üblichen Makrosyntax abweichen.

Zum einen ist die Anzahl der Parameter nicht fest vorgegeben und zum anderen können die Parameter in beliebiger Reihenfolge angegeben werden. Damit soll verdeutlicht werden, daß diese Funktionen eine besondere Stellung in der Designsprache einnehmen. Außerdem ähnelt die Angabe der Parameter dieser Funktionen, das ist ja der location-specifier, eher der natürlichen Sprache.

6.3 Veränderung von Simulationsparametern

Bevor die Parameter und Kenngrößen einer Komponente verändert werden können, muß eine Selektion der Komponente aus dem Schaltplan stattfinden. Erst dann ist der Bezug auf den Schaltplan, und somit auf die Komponente hergestellt.

Bei der Selektion unterscheiden wir drei Arten:

1. Die Parameter der Komponente, die das Symptom aufweist, sollen verändert werden.

In diesem Fall wird die Komponente im modification-Block durch das Schlüsselwort **this** selektiert. Dann kann auf die Parameter mit **set()** oder **get()** zugegriffen werden.

2. Sollen in einem modification-Block Parameter nicht bekannter Komponenten aus dem Schaltplan verändert werden, müssen diese explizit mit der **select_component**-Funktion ausgewählt werden.
3. Die Werte der Parameter einer mit **new()** erzeugten Komponente können über den Namen der zur Komponente assoziierenden Variablen gelesen oder geschrieben werden.

Wie die Selektion von Komponenten realisiert ist, wird in Abschnitt 6.4.2 beschrieben.

6.3.1 set()

Der Befehl zum Setzen eines Simulationsparameters lautet:

`set(k, p, w)`

wobei *k* die Komponente bezeichnet, deren Parameter *p* auf den Wert *w* gesetzt wird. Falls der Typ des Parameters *w* nicht vom gleichen Typ ist wie der des Parameters *p*, wird die Verarbeitung angehalten und ein Laufzeitfehler vom Interpreter angezeigt.

Jetzt ein Beispiel für den **set**-Befehl: Der einstellbare Druck einer Pumpe *P* soll auf 50 bar gesetzt werden:

$$\text{set}(P, P_LIM, 50)$$

6.3.2 get()

Mit **get()** wird der Wert eines Parameters einer Komponente erfragt:

`get(k, p)`

Hier bezeichnet *k* die Komponente und *p* den Name des Parameters, dessen Wert gelesen werden soll. Der Rückgabewert wird an die Stelle des Aufrufs zurückgegeben. D.h. man kann sich den get-Aufruf im Programmcode durch das Ergebnis des Aufrufs ersetzt vorstellen. Falls *p* kein Parameter der Komponente *k* ist, wird ein Laufzeitfehler ausgegeben.

Beispiel für den lesenden Zugriff auf einen Komponentenparameter:

Der einstellbare Druck einer Pumpe *P* soll erfragt werden:

$$\text{get}(P, P_LIM)$$

liefert den Wert 50 des Parameters *P_LIM* nach Anwendung des Beispiels aus Abschnitt 6.3.1.

6.4 Lesender Zugriff auf einen Schaltplan

In diesem Abschnitt wird anhand von Beispielen das Zusammenspiel des action- und location-specifiers erklärt. Diese Beispiele können dem Ingenieur als Vorlage für eigene Problemstellungen dienen.

6.4.1 Terminologie

Bevor wir zu den Beispielen kommen, müssen noch einige wichtige Begriffe zum Verständnis definiert werden.

Graphendarstellung eines Schaltkreises

Im Zusammenhang dieser Diplomarbeit wird die Definition des zusammenhängenden hydraulischen Graphen $G_h(C)$ eines Schaltkreises C erweitert. In der ersten Definition besteht der Graph aus Knoten, die zu Komponenten assoziieren. In der erweiterten Definition unterscheiden wir zwei Arten von Knoten. Das sind zum einen die Knoten, die zu Tri-Connections assoziieren (Kreise), und zum anderen die Knoten, die zu allen anderen Komponenten assoziieren (Quadrate). Außerdem bekommen alle quadratischen Knoten eine Nummer, die zur Identifikation dienen. Die runden Knoten bekommen keine Identifikationsnummer, da aus dem Kontext eines Beispiels hervorgeht, welcher Knoten gemeint ist. Sie werden alle über den Namen *TRI* angesprochen. Diese Benennung soll auf die besondere Stellung der Tri-Connections in einer Schaltung hinweisen.

Pfad

Ein Pfad wird durch seine erste (*first*) und letzte (*last*) Komponente definiert. Gleichzeitig gibt die Pfadrichtung von *first* nach *last* die *Durchlaßrichtung* an. Dies wird z.B. beim Einfügen von Ventilen benutzt, die dann in Flußrichtung, d.h. in Richtung von *first* nach *last*, eingesetzt werden. Die Abbildung 27 zeigt ein Beispiel, wo die Durchlaßrichtung des Rückschlagventils $K3$ durch die Richtung a) des Pfades ($K1, K5$) festgelegt wird und die Richtung des Ventils $K2$ durch die Richtung b) des Pfades ($K4, K1$) festgelegt wird.

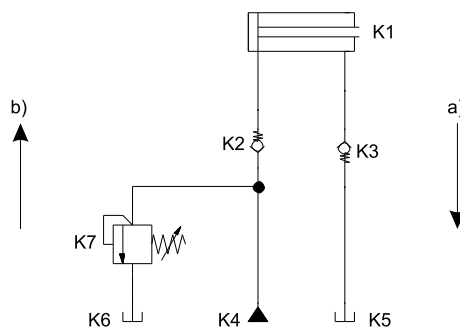


Abbildung 27: Durchlaßrichtung definiert durch die Richtung des Pfades

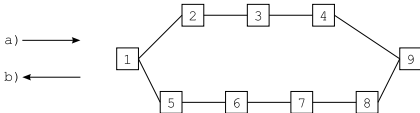
Zweig

Ein Zweig ist eine spezielle Form eines Pfades. Er beginnt an einer Tri-Connection und endet an einer Komponente, die nur ein Gate (Anschluß) besitzt. Die umgekehrte Variante ist natürlich auch möglich. In Abbildung 27 ist ein Zweig geschaltet, der aus dem Tank *K6* und dem Druckbegrenzungsventil *K7* besteht.

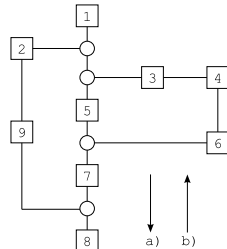
6.4.2 Selektieren von Komponenten (`select_component()`)

Mit der `select_component`-Funktion werden Komponenten oder Pfade aus einem Schaltplan selektiert. Das Ergebnis dieser Funktion ist eine Liste, die die selektierten Pfade oder Komponenten enthält. Im folgenden wird die `select_component`-Funktion mit den möglichen location-specifiers anhand von Beispielen erklärt. Die Reihenfolge der Schlüsselworte des location-specifiers darf beliebig gewählt werden (s. Abschnitt 3.2.2).

<code>select_component</code> <i>(qualifiers)</i>	
<pre>select_component((type = pump_unit, P_LIM = 50))</pre> erzeugt eine Liste aller Pumpen, die in der Schaltung vorkommen und deren Einstelldruck 50 bar ist.	<p>Beschreibung: Alle Komponenten, für die die Bedingungen in <i>qualifiers</i> erfüllt sind, selektieren.</p> <p>Eingabe: Liste der Qualifiers</p> <p>Rückgabe: Liste aller Komponenten, die die Qualifiers erfüllen.</p>

<code>select_component</code> <i>(in_series first last)</i>	
 <pre>a) select_component((in_series 1 9))=((1,2,3,4,9),(1,5,6,7,8,9))</pre> <pre>b) select_component((in_series 9 1))=((9,4,3,2,1),(9,8,7,6,5,1))</pre>	<p>Beschreibung: Alle Pfade, die durch die Komponenten <i>first</i> und <i>last</i> gegeben sind, selektieren.</p> <p>Eingabe: Erste (<i>first</i>) und letzte (<i>last</i>) Komponente des Pfades.</p> <p>Rückgabe: Liste aller Pfade von <i>first</i> nach <i>last</i>.</p>

select_component (in_parallel first last)



a) `select_component((in_parallel 7 5)=((2,9))`
 b) `select_component((in_parallel 5 7)=((9,2))`
`select_component((in_parallel 5 5)=((3,4,6),(6,4,3))`

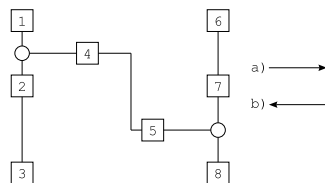
Beschreibung:

Alle Pfade, die parallel zum Pfad von Komponente *first* nach Komponente *last* geschaltet sind, selektieren.

Eingabe: Erste (*first*) und letzte (*last*) Komponente des Pfades.

Rückgabe: Liste aller Pfade, die zum Pfad von *first* nach *last* parallel geschaltet sind.

select_component (Pfad zwischen Tri-Connections)



Preprozeß

1) `select_component(in_series 1 3,`
`(comp_type=Tri_Connection))=(TRI1)`
 2) `select_component(in_series 6 8,`
`(comp_type=Tri_Connection))=(TRI2)`

a) `select_component(in_series TRI1 TRI2)=(4,5)`
 b) `select_component(in_series TRI2 TRI1)=(5,4)`

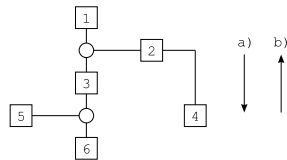
Beschreibung:

Komponenten, die auf dem Pfad zwischen Tri-Connections liegen, selektieren. Die Reihenfolge der Tri-Connections gibt die Richtung des Pfades an.

Eingabe: Beschreibung der Tri-Connections.

Rückgabe: Pfad, der zwischen den Tri-Connections liegt.

select_component (Zweig nach Tri-Connection)



Preprozeß

1) `select_component(in_series 1 3,`
`(comp_type=Tri_Connection))=(TRI)`

a) `select_component(TRI 4)=((2,4))`

b) `select_component(4 TRI)=((4,2))`

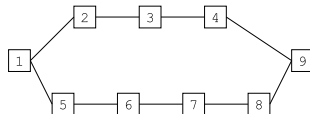
Beschreibung:

Komponenten, die auf einem Abzweig liegen, selektieren.

Eingabe: Tri-Connection, an der der Zweig angeschlossen ist, und Komponente des Zweiges, die nur einen Anschluß besitzt.

Rückgabe: Zweig, der zwischen der Tri-Connection und der Komponente liegt.

select_component (after component)



`select_component((after 7)=(6,8))`

`select_component((after 7, in_series 1 9)=(8))`

`select_component((in_series 9 1, after 7)=(6))`

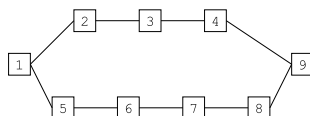
Beschreibung:

Komponenten, die direkt hinter der Komponente *component* liegen, selektieren. Wenn kein Pfad und damit keine Richtung vorliegt, werden die zwei Komponenten vor und hinter *component* selektiert.

Eingabe: Die Komponente *component*.

Rückgabe: Liste der Komponente(n), die direkt hinter (und vor) *component* liegen.

select_component (before component)



`select_component((before 3)=(2,4))`

`select_component((before 3, in_series 1 9)=(2))`

`select_component((in_series 9 1, before 3)=(4))`

Beschreibung:

Komponenten, die direkt vor der Komponente *component* liegen, selektieren. Wenn kein Pfad und damit keine Richtung vorliegt, werden die zwei Komponenten vor und hinter *component* selektiert.

Eingabe: Die Komponente *component*.

Rückgabe: Liste der Komponente(n), die direkt vor (und hinter) *component* liegen.

An dieser Stelle soll noch darauf hingewiesen werden, daß die Beispiele nur die einzelnen Typen 1 bis 3 des location-specifiers abdecken. Die verschiedenen Typen können wie in

Abschnitt 3.2.2 beschrieben beliebig kombiniert werden. Wie das Ergebnis der Varianten dann aussieht, dürfte nach den Beispielen klar sein.

6.5 Veränderung der Schaltplantopologie

Mit der `select_component`-Funktion können Komponenten aus dem Schaltplan ausgewählt werden. Diese Auswahl beschränkt sich auf den lesenden oder schreibenden Zugriff auf Simulationsparameter. Die Topologie der Schaltung bleibt unverändert.

Häufig bestehen Modifikationen einer Schaltung aus Veränderungen der Schaltplantopologie. Diese können durch das

- *Löschen* von Komponenten, das
- *Austauschen* von Komponenten und dem
- *Einfügen* von Komponenten

realisiert werden. Die dafür in der Designsprache vorgesehenen Kernfunktionen werden in den folgenden Abschnitten beschrieben.

6.5.1 Löschen von Komponenten (`delete_component()`)

Die `delete_component`-Funktion unterscheidet sich nur in einem Punkt von der `select_component`-Funktion. Der einzige Unterschied besteht darin, daß `delete_component` die selektierten Komponenten aus dem Schaltplan entfernt. Die dann freiwerdenden Anschlüsse müssen explizit neu verschaltet werden.

Kommen bei einem Löschvorgang mehrere Komponenten zur Auswahl in Frage, werden alle hintereinander ausprobiert (s. Kapitel 8), um die richtige unter ihnen herauszufinden (Backtracking). Kommen ganze Pfade zum Löschen in Frage, wird dies ebenfalls ausprobiert. Anschließend wird überprüft, ob die Modifikation zum gewünschten Abstellen eines Symptoms geführt hat.

6.5.2 Austauschen von Komponenten

Sobald ein Simulationsparameter vom Typ *characteristic* verändert wird, handelt es sich um das Austauschen einer Komponente (s. Abschnitt 3.1.2). Da sich aber die Komponentenklasse des veränderten Bauteiles nicht geändert hat, d.h. es ist immer noch eine Komponente vom selben Typ, muß nicht in die Schaltplantopologie eingegriffen werden. Der Simulationsparameter wird einfach neu gesetzt, und der Komponentenaustausch ist fertig.

Anders sieht es beim Austausch von zwei verschiedenen Komponententypen aus. Hier haben die Bauteile verschiedene Funktionalitäten und müssen wie folgt ausgetauscht werden:

1. Lösche die zu ersetzende Komponente mit `delete_component()`.
2. Füge die neue Komponente zwischen den offenen Anschlüssen mit `insert_component()` ein.

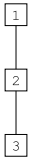

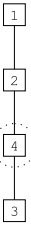
Dieses Vorgehen unterscheidet sich grundlegend von der ersten Variante. Es müssen für den Austausch strukturelle Veränderungen im Schaltplan vorgenommen werden.

Weil beim strukturellen Eingriff in den Schaltplan mehrere Befehle hintereinander ausgeführt werden, und der strukturelle Austausch von Komponenten eine Grundfunktion der Schaltplanveränderung ist, bietet sich die Implementierung eines speziellen selbstdefinierten Makros (`replace()`) an, das diese Aufgabe erledigt.

6.5.3 Einfügen von Komponenten (`insert_component()`)

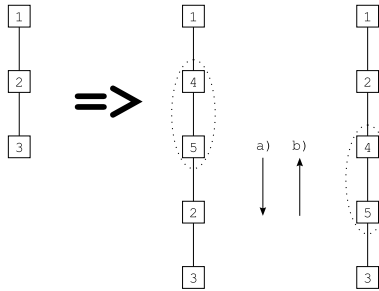
Mit der `insert_component`-Funktion können Komponenten und Pfade in eine existierende Schaltung eingefügt werden. Neben dem Objekt (Liste der Komponenten), das eingefügt werden soll, muß der location-specifier angegeben werden, der die Position in der Schaltung festlegt, an der das Objekt eingefügt werden soll.

In den folgenden Beispielen werden viele verschiedene Fälle behandelt, die der Ingenieur als Vorlage für seine eigenen Anwendungen benutzen kann. Da hier nicht alle möglichen Fälle behandelt werden können, werden nur die am häufigsten auftretenden angegeben.

insert_component <i>(in_series first last, list)</i>		
<div style="display: flex; justify-content: space-around; align-items: flex-start;"> <div style="text-align: center;"> <p>vor insert</p>  </div> <div style="font-size: 2em; margin: 0 10px;">=></div> <div style="display: flex; flex-direction: column; align-items: center;"> <div style="text-align: center;"> <p>nach insert (1. Möglichkeit)</p>  </div> <div style="margin: 5px 0;">a) ↓</div> <div style="text-align: center;"> <p>nach insert (2. Möglichkeit)</p>  </div> <div style="margin: 5px 0;">b) ↑</div> </div> </div>	<p>Beschreibung: Einfügen einer Komponente auf dem Pfad von Komponente <i>first</i> nach Komponente <i>last</i>.</p> <p>Eingabe: Pfad <i>first last</i>, auf dem die Komponente in <i>list</i> eingefügt werden soll.</p> <p>Rückgabe: TRUE, falls das Einfügen erfolgreich war, FALSE sonst.</p>	
<p>a) <code>insert_component((in_series 1 3, (4)))</code> b) <code>insert_component((in_series 3 1, (4)))</code></p>		

insert_component (in_series first last, list)

vor insert nach insert nach insert
(1. Möglichkeit) (2. Möglichkeit)



a) `insert_component((in_series 1 3, (4,5))`
 b) `insert_component((in_series 3 1, (5,4))`

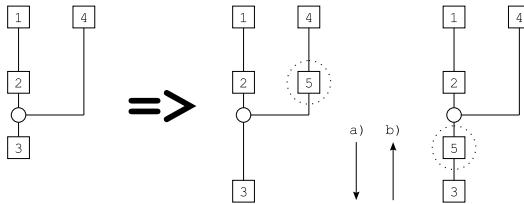
Beschreibung:

Einfügen eines Pfades *list* auf dem Pfad von Komponente *first* nach Komponente *last*.

Eingabe: Pfad *first last*, auf dem der Pfad *list* eingefügt werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

vor insert nach insert nach insert
(1. Möglichkeit) (2. Möglichkeit)



a) `insert_component((in_series 4 3, (5))`
 b) `insert_component((in_series 3 4, (5))`

Weil auf dem Pfad (4 3) eine Tri-Connection liegt, gibt es zwei Einfügemöglichkeiten.

Beschreibung:

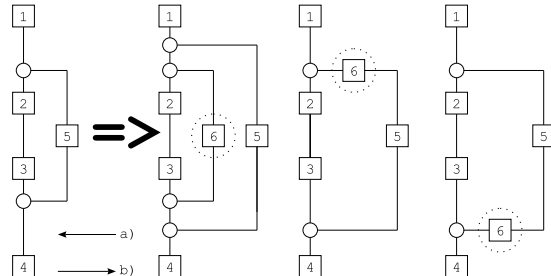
Einfügen einer Komponente auf dem Pfad von Komponente *first* nach Komponente *last*.

Eingabe: Pfad *first last*, auf dem die Komponente in *list* eingefügt werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component (in_parallel first last, list)

vor insert nach insert nach insert nach insert
(1. Möglichk.) (2. Möglichk.) (3. Möglichk.)



a) `insert_component((in_parallel 2 3, (6))`
 b) `insert_component((in_parallel 3 2, (6))`

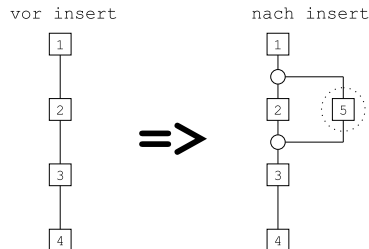
Beschreibung:

Einfügen einer Komponente, die parallel zu einem Pfad *first last* geschaltet werden soll.

Eingabe: Liste *list* der Komponenten, die zum Pfad *first last* parallel geschaltet werden sollen.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component (in_parallel *first last*, *list*)



`insert_component((in_parallel 2 2, (5))`

Da in diesem Beispiel keine Richtung durch den Pfad *first last* gegeben ist, werden beide Richtungen berücksichtigt.

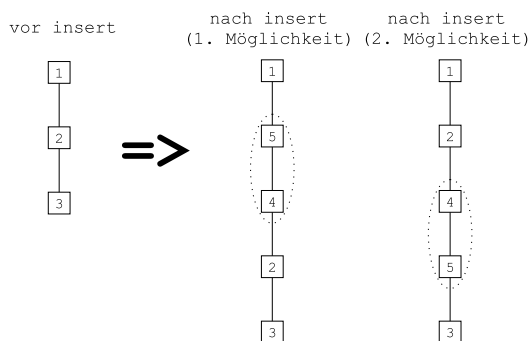
Beschreibung:

Einfügen eines Pfades *list*, der parallel zum Pfad *first last* geschaltet werden soll.

Eingabe: Pfad *list*, der zum Pfad *first last* parallel geschaltet werden sollen.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component (after *component*, *list*)



`insert_component((after 2, (4,5))`

Da in diesem Beispiel durch den location-specifier keine Richtung gegeben ist, müssen beide Möglichkeiten, vor und nach Komponente 2 berücksichtigt werden. Sobald eine Richtung vorliegt, ist die Position des einzufügenden Objektes eindeutig.

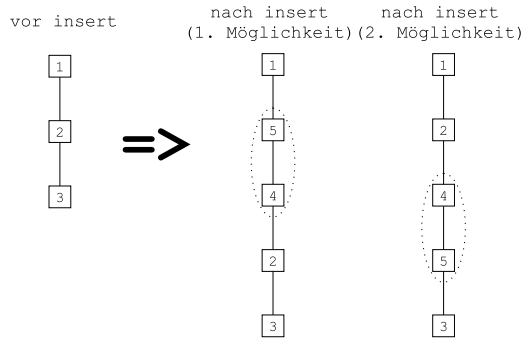
Beschreibung:

Einfügen eines Pfades *list* direkt hinter einer Komponente *component*.

Eingabe: Komponente *component*, hinter der die Komponenten der Liste *list* eingefügt werden sollen.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component (before component, list)



`insert_component((before 2, (5,4)))`

Da in diesem Beispiel durch den location-specifier keine Richtung gegeben ist, müssen beide Möglichkeiten, vor und nach Komponente 2 berücksichtigt werden. Sobald eine Richtung vorliegt, ist die Position des einzufügenden Objektes eindeutig.

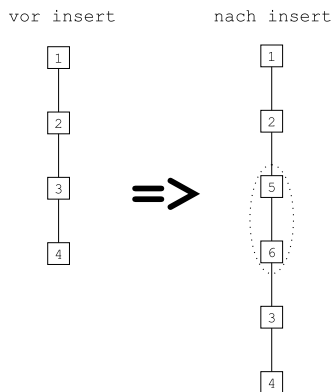
Beschreibung:

Einfügen eines Pfades *list* direkt vor einer Komponente *component*.

Eingabe: Komponente *component*, vor der die Komponenten der Liste *list* eingefügt werden sollen.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component (in_series first last, after component, list)



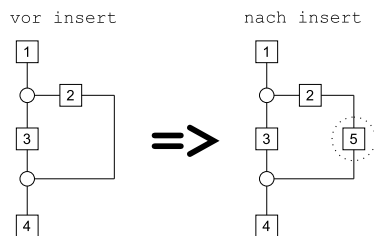
`insert_component((in_series 1 4, after 2, (5,6)))`

Beschreibung:

Einfügen eines Pfades *list* direkt hinter einer Komponente *component* auf einem Pfad von Komponente *first* nach Komponente *last*.

Eingabe: Komponente *component*, hinter der die Komponenten der Liste *list* auf dem Pfad *first last* eingefügt werden sollen.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.



`insert_component((in_series 1 4, after 2, (5)))`

Beschreibung:

Einfügen eines Pfades *list* direkt hinter einer Komponente *component* auf einem Pfad von Komponente *first* nach Komponente *last*.

Eingabe: Komponente *component*, hinter der die Komponenten der Liste *list* auf dem Pfad *first last* eingefügt werden sollen.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component

(in_series *first last*, before component, list)

Beschreibung:

Einfügen eines Pfades *list* direkt vor einer Komponente *component* auf einem Pfad von Komponente *first* nach Komponente *last*.

Funktioniert prinzipiell genauso wie die **after** -Variante.

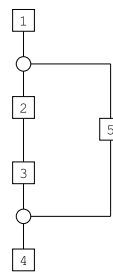
Eingabe: Komponente *component*, vor der die Komponenten der Liste *list* auf dem Pfad *first last* eingefügt werden sollen.

Rückgabe: **TRUE** , falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component

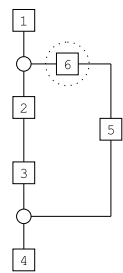
(in_parallel *first last*, after component, list)

vor insert



=>

nach insert



insert_component(in_parallel 2 3,(6),after 5)

Beschreibung:

Einfügen eines Pfades *list*, der parallel zu einem Pfad von Komponente *first* nach Komponente *last* geschaltet ist und direkt hinter der Komponente *component* liegt.

Eingabe: Liste *list* der Komponenten, die parallel zum Pfad *first last* geschaltet werden sollen und direkt hinter der Komponente *component* eingefügt werden sollen.

Rückgabe: **TRUE** , falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component

(in_parallel *first last*, before component, list)

Beschreibung:

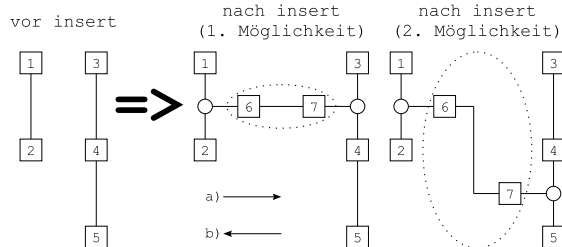
Einfügen eines Pfades *list*, der parallel zu einem Pfad von Komponente *first* nach Komponente *last* geschaltet ist und direkt vor der Komponente *component* liegt.

Funktioniert prinzipiell genauso wie die **after** -Variante.

Eingabe: Liste *list* der Komponenten, die parallel zum Pfad *first last* geschaltet werden sollen und direkt hinter der Komponente *component* liegen.

Rückgabe: **TRUE** , falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component (Pfad zwischen zwei Tri-Connections)



Preprozeß

- 1) `insert_component(in_series 1 2, (TRI1))`
- 2) `insert_component(in_series 3 5, (TRI2));` Achtung Auswahl!

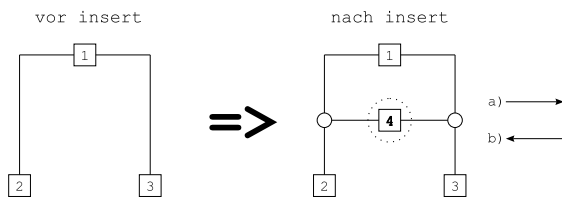
- a) `insert_component(in_series Tri1 TRI2, (6,7))`
- b) `insert_component(in_series Tri2 TRI1, (7,6))`

Beschreibung:

Einfügen eines Pfades, der zwischen den Tri-Connections *TRI1* und *TRI2* liegt.

Eingabe: Pfad, der zwischen den Tri-Connection *TRI1* und *TRI2* eingefügt werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.



Preprozeß

- 1) `insert_component(in_series 1 2,(TRI1))`
- 2) `insert_component(in_series 1 3,(TRI2))`

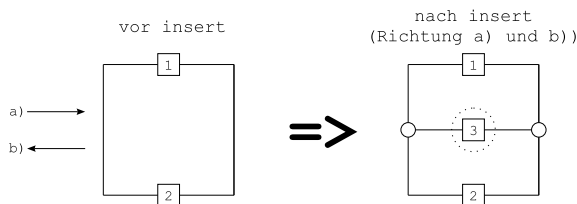
- a) `insert_component(in_series TRI1 TRI2, (4))`
- b) `insert_component(in_series TRI2 TRI1, (4))`

Beschreibung:

Einfügen eines Pfades, der zwischen den Tri-Connections *TRI1* und *TRI2* liegt.

Eingabe: Pfad, der zwischen den Tri-Connection *TRI1* und *TRI2* eingefügt werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.



Preprozeß

- 1) `insert_component(in_series 1 2, (TRI1))`
- 2) `insert_component(in_series 1 2, (TRI2))`

`insert_component(in_series TRI1 TRI2, (3))`

Hier muß darauf geachtet werden, daß *TRI1* ≠ *TRI2* gilt.

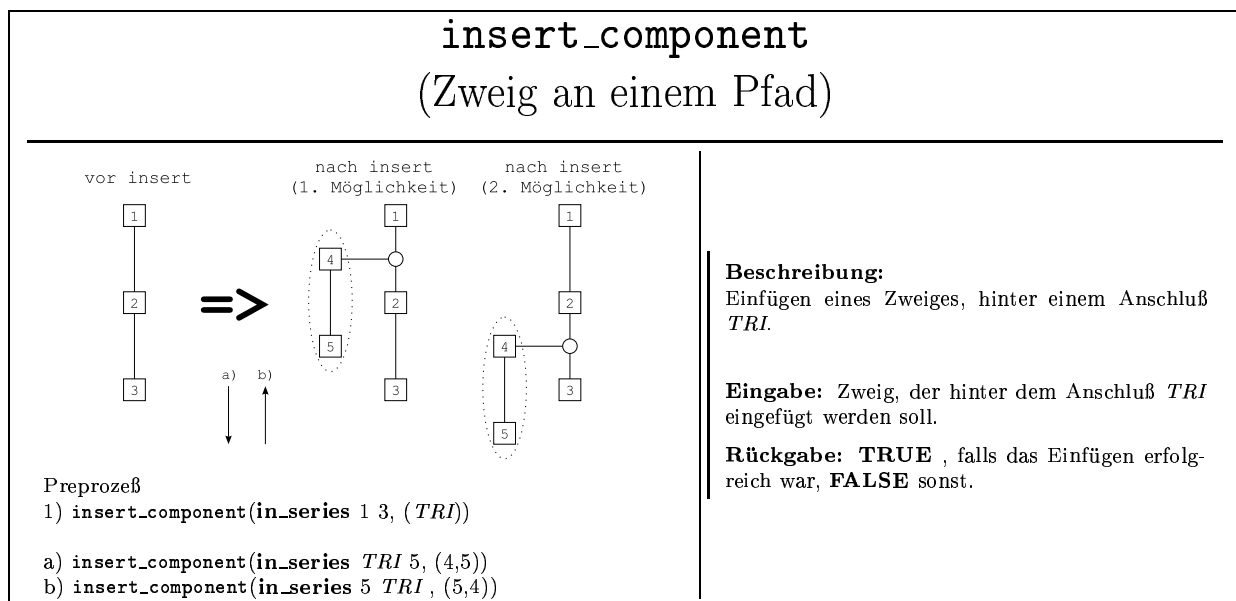
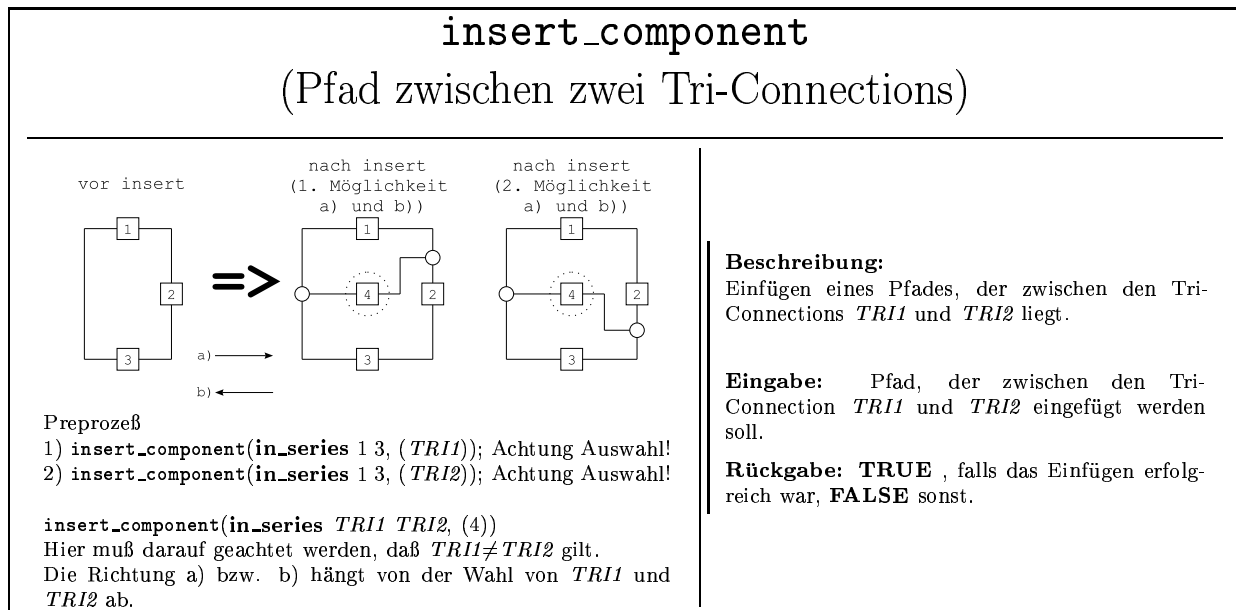
Die Richtung a) bzw. b) hängt von der Wahl von *TRI1* und *TRI2* ab.

Beschreibung:

Einfügen eines Pfades, der zwischen den Tri-Connections *TRI1* und *TRI2* liegt.

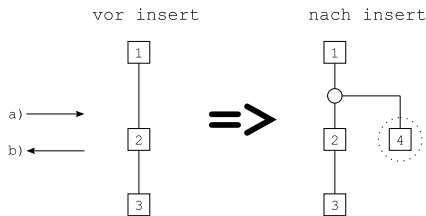
Eingabe: Pfad, der zwischen den Tri-Connection *TRI1* und *TRI2* eingefügt werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.



insert_component

(Pfad/Zweig direkt hinter einer Komponente)



Preprozeß

1) `insert_component(in_series 1 3, after 1, (TRI))`

a) `insert_component(in_series TRI 4, (4))`

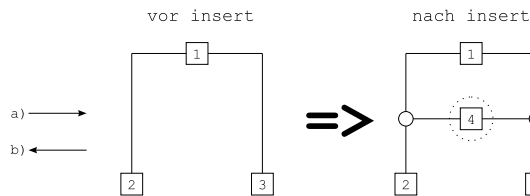
b) `insert_component(in_series 4 TRI, (4))`

Beschreibung:

Einfügen eines Zweiges, dessen Anschluß direkt hinter einer Komponente liegt.

Eingabe: Zweig, der direkt hinter einer Komponente angeschlossen werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.



Preprozeß

1) `insert_component(in_series 1 2, after 1, (TRI1))`

2) `insert_component(in_series 1 3, after 1, (TRI2))`

a) `insert_component(in_series TRI1 TRI2, (4))`

b) `insert_component(in_series TRI2 TRI1, (4))`

Beschreibung:

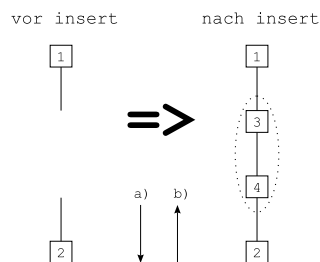
Einfügen eines Pfades, dessen Tri-Connections direkt hinter einer Komponente liegen.

Eingabe: Pfad, dessen Anschlüsse direkt hinter einer Komponente angeschlossen werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component

(Pfad zwischen offenen Anschlüssen)



a) `insert_component(in_series 1.gate 2.gate, (3,4))`

b) `insert_component(in_series 2.gate 1.gate, (4,3))`

Beschreibung:

Einfügen eines Pfades, der an den offenen Anschlüssen zweier Komponenten angeschlossen werden soll.

Eingabe: Pfad, der zwischen den offenen Anschlüssen der beiden Komponenten angeschlossen werden soll.

Rückgabe: **TRUE**, falls das Einfügen erfolgreich war, **FALSE** sonst.

insert_component (Pfad zwischen offenen Anschlüssen)	
<div><div><div><div><div>vor insert</div><div><div><div>1</div><div>2</div></div><div><div>3</div><div>4</div></div></div><div>=></div><div><div>nach insert</div><div><div><div>1</div><div>2</div></div><div><div><div>3</div><div>4</div></div><div><div>5</div></div></div></div></div></div><div><div>a) →</div><div>b) ←</div></div></div><div><p>Preprozeß</p><p>1) <code>insert_component(in_series 1 2,(TRI))</code></p><p>a) <code>insert_component(in_series TRI 3.gate, (5))</code></p><p>b) <code>insert_component(in_series 3.gate TRI , (5))</code></p></div></div></div>	<div><p>Beschreibung:</p><p>Einfügen eines Pfades, zwischen einem offenen Anschluß und einer Tri-Connection <i>TRI</i> auf einem Pfad.</p><p>Eingabe: Pfad, der zwischen dem Anschluß der Komponente und der Tri-Connection <i>TRI</i> angeschlossen werden soll.</p><p>Rückgabe: TRUE , falls das Einfügen erfolgreich war, FALSE sonst.</p></div>
<div><div><div><div><div>vor insert</div><div><div><div>1</div><div>2</div><div>3</div><div>4</div></div><div><div>5</div></div></div><div>=></div><div><div>nach insert (1. Möglichkeit)</div><div><div><div>1</div><div>2</div><div>3</div><div>4</div></div><div><div><div>5</div><div>6</div><div>7</div></div></div></div><div>nach insert (2. Möglichkeit)</div><div><div><div>1</div><div>2</div><div>3</div><div>4</div></div><div><div><div>5</div><div>6</div><div>7</div></div></div></div></div></div><div><div>a) ↓</div><div>b) ↑</div></div></div><div><p>Preprozeß</p><p>1) <code>insert_component(in_series 2 4, (TRI)); Achtung Auswahl!</code></p><p>a) <code>insert_component(in_series 5.gate TRI, (6,7))</code></p><p>b) <code>insert_component(in_series TRI 5.gate, (7,6))</code></p></div></div></div>	<div><p>Beschreibung:</p><p>Einfügen eines Pfades, zwischen dem offenen Anschluß einer Komponente und einer neuen Tri-Connection.</p><p>Eingabe: Pfad, der zwischen dem offenen Anschluß der Komponente und der neuen Tri-Connection eingefügt werden soll.</p><p>Rückgabe: TRUE , falls das Einfügen erfolgreich war, FALSE sonst.</p></div>
<div><div><div><div><div>vor insert</div><div><div><div>1</div><div>2</div></div><div><div>3</div><div>4</div></div></div><div>=></div><div><div>nach insert</div><div><div><div>1</div><div>2</div></div><div><div><div>3</div><div>4</div></div></div></div></div></div><div><div>a) →</div><div>b) ←</div></div></div><div><p>a) <code>insert_component(in_series 2.gate 4, (3,4))</code></p><p>b) <code>insert_component(in_series 4 2.gate, (4,3))</code></p></div></div></div>	<div><p>Beschreibung:</p><p>Anhängen eines Zweiges an einem offenen Anschluß einer Komponente.</p><p>Eingabe: Zweig, der hinter dem offenen Anschluß eingefügt werden soll.</p><p>Rückgabe: TRUE , falls das Einfügen erfolgreich war, FALSE sonst.</p></div>

7 Die Designsprache und andere Programmiersprachen

Die Designsprache ist das Handwerkszeug eines Hydraulikingenieurs, der sein Auslegungswissen formulieren und in einer Wissensbasis zusammenfassen will.

Bei der Entwicklung der Designsprache wurde darauf geachtet, daß wenige mächtige Konstrukte zur Verfügung gestellt werden, mit denen der Ingenieur sein Wissen formulieren kann. Dabei sind die verschiedensten Konzepte aus anderen Programmiersprachen (teilweise) übernommen worden.

Wo die Designsprache in der Menge der Programmiersprachen einzuordnen ist, und welche Konzepte aus anderen Sprachen übernommen wurden, soll im folgenden diskutiert werden.

7.1 Programmiermodell

Bevor die Konzepte der Designsprache entwickelt werden konnten, mußte ein Programmiermodell festgelegt werden, an dem sich die Konzepte orientieren.

Aus der Diskussion mit dem Hydraulikingenieur hat sich ergeben, daß er mit den Begriffen Variable, Zuweisung und Prozeduren umgehen kann. Gerade diese Merkmale machen eine imperative Programmiersprache aus ([Kast95]). Aus diesem Grund wurde für die Designsprache das *imperative Programmiermodell* gewählt. In ihm sind Zuweisungen an Variablen, Ablaufstrukturen und Prozeduren (Makros) möglich. Im Gegensatz zu den üblichen imperativen Programmiersprachen wie C, PASCAL oder BASIC verfügt die Designsprache nur über einen geringen Befehlssatz. Dadurch soll sichergestellt werden, daß die Sprache übersichtlich bleibt und einfach zu erlernen und anzuwenden ist.

7.2 Eigenschaften

Im folgenden sollen die Eigenschaften der Designsprache diskutiert werden. Dabei werden einige Besonderheiten im Vergleich zu anderen Programmiersprachen erläutert und begründet. Zu den Eigenschaften einer Programmiersprache gehören die

1. Bindung von Definitionen an Bezeichnern,
2. Typbindung und
3. Ablaufstrukturen

Zu 1.

Der erste Punkt befaßt sich mit der Bindung einer Definition an einen Bezeichner. Man unterscheidet zwischen der statischen (z.B. C, PASCAL) und dynamischen (z.B. LISP) Bindung von Definitionen an Bezeichner. Bei der *statischen* Bindung legen die Bindungsregeln der Sprache fest, welche Definition für einen Bezeichner im Quellcode gültig ist.

Bei der *dynamischen* Bindung wird ein freier (globaler) Bezeichner in einem Programmabschnitt an eine Definition des Bezeichners in dem zuletzt ausgeführten, noch nicht beendeten

Programmblock gebunden.

Aus Übersichtlichkeitsgründen wurde für die Designsprache die statische Bindung gewählt. So kann aus dem Programmcode heraus erkannt werden, welche Definition zu welchem Bezeichner gehört. Die Bindungsregeln der Designsprache werden in Abschnitt 8.2 angegeben.

Zu 2.

Der zweite Punkt befaßt sich mit der Typbindung. Bei der *statischen Typbindung* (z.B. PASCAL, C) ist jedem Programmobjekt sein Typ statisch zugeordnet. Nur die für den Typ definierten Operationen sind erlaubt. Typumwandlungen von Werten (Konversion) sind möglich.

Bei der *dynamischen Typbindung* (z.B. LISP, Smalltalk) wird der Typ eines Programmobjektes bei der Auswertung bestimmt. Nur die für den Typ definierten Operationen sind erlaubt (Laufzeitbedingung: Typ muß aus dem Wert erkennbar sein).

Bei der *freien Typbindung* (Assembler, FORTRAN) interpretieren die Operationen ihre Operanden als Werte eines bestimmten Typs. Dabei sind Uminterpretationen möglich.

An dieser Stelle unterscheidet sich die Designsprache von den üblichen Programmiersprachen, weil bei denen entweder eine statische, dynamische oder freie Typbindung zur Anwendung kommt. In der Designsprache können die Typen von Bezeichnern sowohl statisch (als formaler Parameter in Makroköpfen) oder dynamisch (bei der Erzeugung von Hilfsvariablen) gebunden werden. Dieses Vorgehen erleichtert die Anwendung von Hilfsvariablen und zwingt den Anwender dennoch dazu, Makros mit den richtigen Typen der aktuellen Parameter aufzurufen.

Zu 3.

Ablaufstrukturen sind ein wesentliches Konzept höherer imperativer Programmiersprachen wie C und PASCAL. Weil sie die Wiederholung eines Programmblockes und die bedingte Ausführung realisieren, wurden sie in die Designsprache integriert.

Die Besonderheit der Schleifen in der Designsprache besteht darin, daß die Laufvariable über die Elemente einer Liste läuft. So können Speicherobjekte wie Pfade und Zweige, die in Listen gespeichert werden, einfach elementweise betrachtet werden. Außerdem können Listen, in denen zum Beispiel nach einem select-Befehl mehrere Auswahlmöglichkeiten stehen, Element für Element, d.h. Möglichkeit für Möglichkeit betrachtet werden. Dieses Listenkonzept ähnelt dem von LISP und eröffnet daher flexible Möglichkeiten für den Anwender.

7.3 Bedienung

Wie fast alle herkömmlichen Programmiersprachen basiert auch die Designsprache auf einer Programmdatei, in der die Makros und Wissensklassen aufgeschrieben sind. Interaktionen mit dem Anwender während des Programmlaufes sind nur beschränkt möglich, weil die Designsprache im Gegensatz zu BASIC oder PASCAL keine Eingabefunktionen besitzt. Es

können lediglich Texte und Werte von Variablen ausgegeben werden.

Wie in LISP, BASIC und Prolog wird das Programm interpretiert, d.h. es wird in einem Schritt analysiert und ausgeführt.

7.4 Besonderheiten

Ein wesentliches Merkmal der Designsprache ist das Konzept der objektorientierten Wissensklassifizierung.

Dieses Konzept hat die Designsprache von den objektorientierten Programmiersprachen wie C++ und Smalltalk geerbt. Allerdings heißen Methoden in der Designsprache Regeln und Objektklassen werden Wissensklassen genannt. Nur das mächtige Konzept der einfachen und mehrfachen Vererbung wird nicht unterstützt.

Am Ende dieses Kapitels soll in Abbildung 28 noch einmal zusammengefaßt werden, von welchen Sprachen die Designsprache ihre wesentlichen Konzepte und Konstrukte geerbt hat.

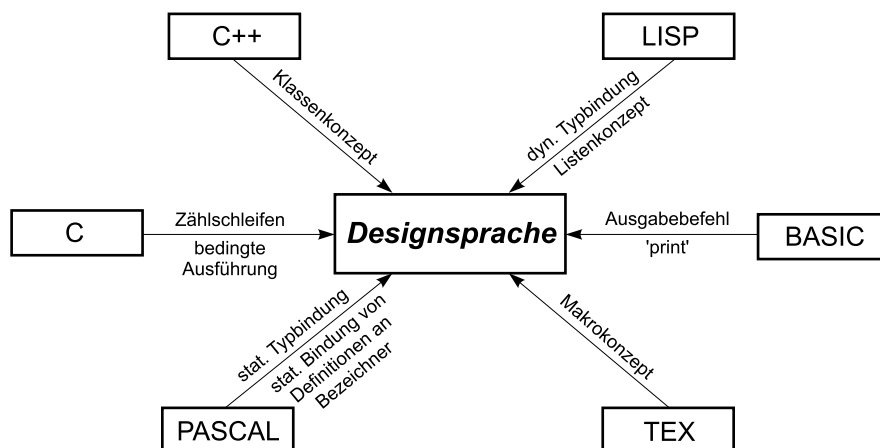


Abbildung 28: Konzepte und Konstrukte, die die Designsprache von anderen Sprachen geerbt hat.

8 Verarbeitung von Designwissen

In den vorherigen Kapiteln wurde das Gerüst einer formalen Sprache zur Formulierung von Expertenwissen zur Auslegung hydraulischer Anlagen erarbeitet. Mit dieser Sprache kann der Ingenieur sein Wissen in einer Wissensbasis zusammenfassen. Die formale und vollständige Sprachdefinition wird in Anhang A.7 aufgeführt.

Damit ist das erste Ziel dieser Diplomarbeit erreicht.

Das zweite Ziel ist die effiziente Verarbeitung des in der Designsprache formulierten Expertenwissens.

Hierzu werden wir zunächst einen Parser entwickeln, der die Sprache erkennt und auf Syntaxfehler aufmerksam macht.

Dann wird ein Interpretierer konstruiert, der die Kernfunktionen ausführt und Ablaufstrukturen abarbeitet. Da beim Eingriff in einen Schaltkreis im allgemeinen mehrere Auswahlmöglichkeiten vorliegen (z.B. `select_component((comp_type = cylinder))`), muß dies bei der Verarbeitung entsprechend berücksichtigt werden.

Ein Modell zur effizienten Verarbeitung des Designwissens ist durch eine sogenannte Blackboard-Architektur gegeben. Eine spezielle Variante dieses Modells wird für die Verarbeitung der Designsprache entwickelt.

Am Ende dieses Kapitels wird eine Systemarchitektur angegeben, die die Repräsentation des Designwissens und dessen effiziente Verarbeitung ermöglicht.

8.1 Der Parser

Der Code einer Wissensbasis besteht aus einer Folge von Zeilen. Jede Zeile kann aus mehreren Ausdrücken bestehen, die durch Semikola oder Leerzeichen getrennt sind. Die Basiseinheiten eines Ausdrucks sind Zahlen, Namen, Operationen und Token. Namen für Hilfsvariablen können ohne vorhergehende Deklaration verwendet werden und bestehen aus Zeichen.

Die lexikalische und syntaktische Analyse geschieht im Stil des rekursiven Abstiegs (*recursive descent*), einer populären und gradlinigen top-down-Technik.

Weil der Parser in C implementiert ist, und in einer Sprache wie C die Funktionsaufrufe wenig kostspielig sind, ist diese top-down-Technik auch relativ effizient. Jede Produktionsregel der Grammatik (s. Anhang A.7) entspricht einer Funktion, die andere Funktionen aufruft. Terminale Symbole (wie z.B. **macro**, **if**, **then**) werden von den Syntaxanalyse-Funktionen erkannt. Sobald die Operanden eines (Unter-)Ausdrucks bekannt sind, wird der Ausdruck ausgewertet. In einem Compiler würde an dieser Stelle Code erzeugt ([Kast95]).

Zur Verdeutlichung soll die Funktionsweise des implementierten recursive-descent-Parsers anhand des Ableitungsbaumes in Abbildung 29 verdeutlicht werden.

Die Blätter des Baumes werden in der Reihenfolge von 1 bis 28 durchlaufen und bestehen aus terminalen Zeichenketten.

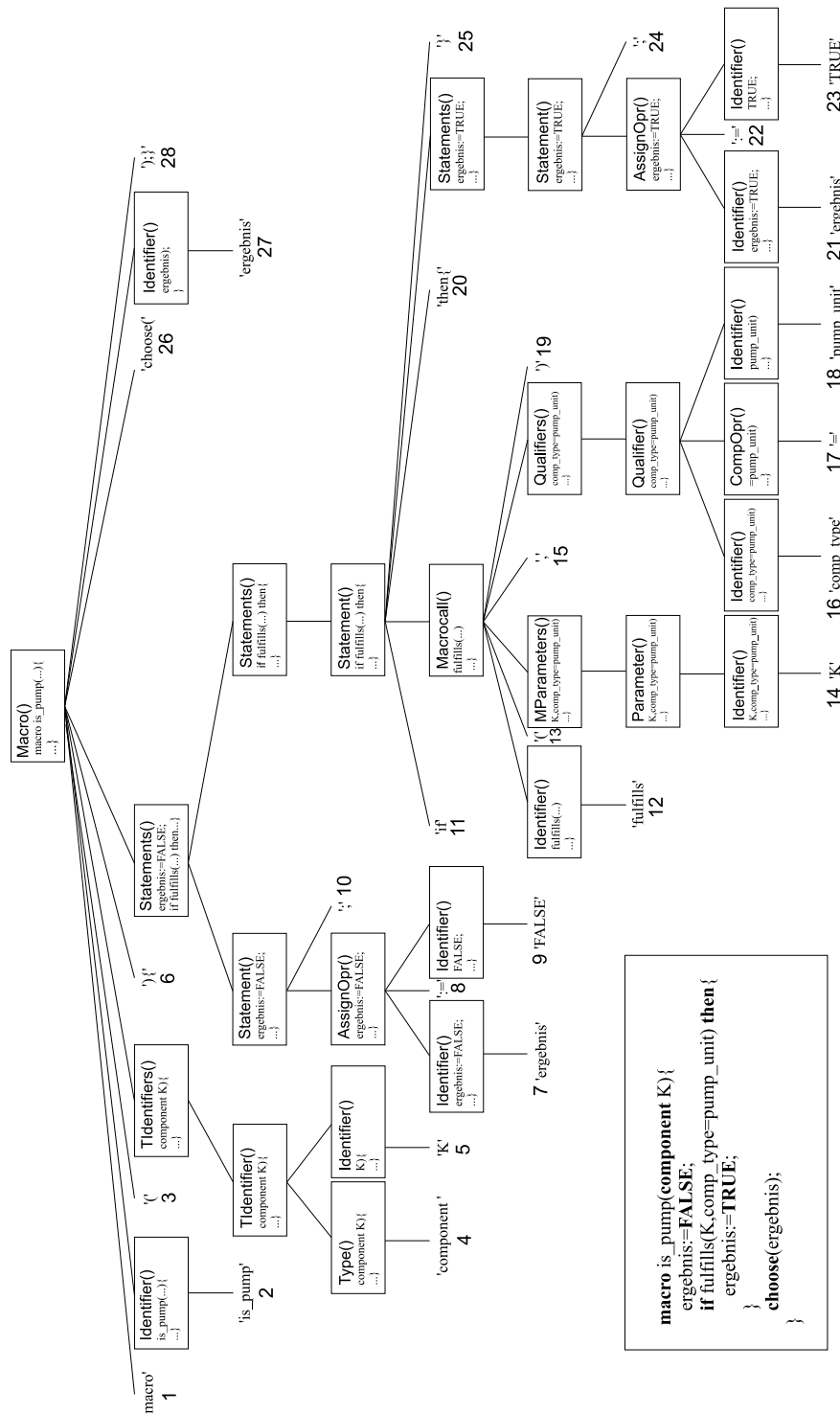


Abbildung 29: Der Ableitungsbaum eines Makros mit den dazugehörigen Funktionsaufrufen des Parsers

Die Regeln, die diesen Ableitungsbaum erzeugen, sind in Anhang A.7 in Backus-Naur-Form (BNF s. [Hopc90], [Kast95]) aufgeführt.

Der Programmcode der Wissensbasis wird aus der Eingabedatei gelesen und online geparkt. Dabei ist zu bemerken, daß von der Eingabe immer das nächste Zeichen gelesen werden muß. Dadurch ist der Parser immer bereits „ein Zeichen voraus“. Die Parser-Funktionen sind verpflichtet, jeweils ein Zeichen mehr zu lesen, als es von der Produktionsregel vorgeesehen ist (s. **LR**(1)-Grammatik [Hopc90]).

Der Lesevorgang wird mit einer Fehlermeldung abgebrochen, falls der Parser einen Syntaxfehler erkennt. Die Meldung gibt die Zeile und die Stelle in der Zeile aus, an der der Syntaxfehler gefunden wurde. Außerdem wird eine Fehlerbeschreibung angegeben, die auf eine mögliche Ursache hinweist.

Wenn der Parser keinen Fehler im Quellcode finden konnte, terminiert er ohne Fehlermeldung und die Verarbeitung kann den Interpretierer starten.

8.2 Der Interpretierer

Bevor der Interpretierer gestartet wird, scannt der Parser den Quellcode auf mögliche Syntaxfehler. Erst wenn der Parser ohne Syntaxfehler terminiert, kann davon ausgegangen werden, daß der Code der Definition der Designsprache entspricht.

Der Lader des Interpretierers liest den Quellcode der Wissensbasis in den Hauptspeicher, und entfernt gleichzeitig Kommentare und überflüssige Leerzeichen. Dabei werden spezielle Datenstrukturen angelegt, die einen schnellen Zugriff auf Wissensklassen, Makros und den lokalen Variablen der Makros während der Verarbeitung erlauben.

Erst jetzt kann der Programmcode interpretiert werden. Dabei liest der Interpretierer Wort für Wort und entscheidet, ob es sich um eine Kernfunktion, um einen Aufruf eines selbstdefinierten Makros oder um ein anderes Token handelt.

Handelt es sich um einen *Makroaufruf*, wird die Parameterliste des Makros von links nach rechts rekursiv ausgewertet (*Linksrekursivität*). Dabei werden Variablen durch ihre Werte ersetzt und Makroaufrufe ausgewertet und durch ihr Ergebnis ersetzt. Die Parameterübergabeart bei Makroaufrufen ist somit „*call-by-value*“, weil nur Werte übergeben werden. So entsteht, ähnlich wie beim Parsen, ein Interpretiererbaum, der top-down abgearbeitet wird.

Ist die Parameterliste des aktuellen Makroaufrufes vervollständigt, wird der Rumpf des Makros ausgeführt.

Handelt es sich beim aktuellen Wort um ein Token, wird zwischen einer *Schleife*, einer *bedingten Ausführung* und einer *Zuweisung* an eine Hilfsvariablen unterschieden. Die Zuweisung kann direkt ausgeführt werden. Sie deklariert und definiert gleichzeitig einen neuen

Bezeichner. Dieser Bezeichner hat nur in dem ihn umgebenden Makro- oder Modifikationsblock seine Gültigkeit. Generell läßt sich über den Gültigkeitsbereich von Variablen folgendes sagen:

Eine Definition eines Bezeichners b gilt von der Definitionsstelle bis zum Ende des Makro- oder Modifikationsblockes, in dem er definiert wurde. Das Überladen des Bezeichners b innerhalb eines Blocks wird nicht unterstützt. Ein Elementbezeichner hat nur innerhalb seines Schleifenrumpfes seine Gültigkeit.

Bei einer bedingten Ausführung wird zuerst der Bedingungsausdruck rekursiv ausgewertet. Ist dieser Ausdruck wahr, wird der then-Block ausgeführt.

Ähnlich wird eine Schleife behandelt. Zuerst wird der Elementbezeichner definiert, bevor rekursiv die Liste berechnet wird, über deren Elemente gelaufen werden soll. Dann wird auf jedes Element der Liste, von links nach rechts, der Schleifenrumpf angewendet. Da der Gültigkeitsbereich für den Elementbezeichner der Schleifenrumpf ist, kann nach Beendigung der Schleife nicht mehr auf ihn zugegriffen werden.

Weil in der Designsprache das Konzept der dynamischen Typbindung⁶ verwendet wird, muß während der Interpretation des Quellcodes ständig die dynamische Semantik des Codes überprüft werden (vgl. [Kast95]). Tauchen Fehler während der Laufzeit auf, unterbricht der Interpretierer mit einem Laufzeitfehler die Arbeit, und versucht eine mögliche Fehlerursache zu beschreiben.

In Abbildung 30 wird der gesamte Vorgang veranschaulicht, der zur Erkennung und Ausführung von Programmcode in der Designsprache erforderlich ist.

8.3 Der Lösungsraum

Die Struktur des Lösungsraumes eines Konfigurationsproblems bestimmt im wesentlichen die Laufzeit eines Verfahrens zur Generierung von Lösungen für das Problem.

Für die computerunterstützte Auslegung eines hydraulischen Kreislafes gilt das gleiche. Weil ein Fehlverhalten einer Komponente im allgemeinen durch verschiedene Modifikationsmöglichkeiten kompensiert werden kann, spannt sich bei der Suche durch den Lösungsraum ein Baum mit entsprechend vielen Söhnen an einem Knoten auf. Für alle Söhne muß dann rekursiv das Verfahren fortgesetzt werden.

Falls in einer Wissensklasse für ein Symptom viele verschiedene Lösungsvorschläge aufgeführt sind, impliziert das einen hohen Aufwand für die Verarbeitung. Aus diesem Grund sollten auch nur vielversprechende Lösungsvorschläge für ein Symptom in die Wissensbasis aufgenommen werden.

⁶Der Typ eines Bezeichners wird erst während der Laufzeit gebunden [Kast95] .

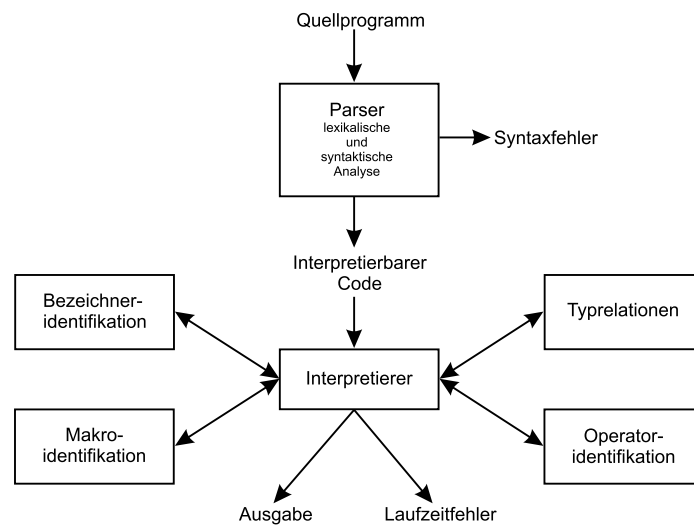


Abbildung 30: Phasen zur Erkennung und Ausführung eines Quellprogramms in der Designsprache

In Abbildung 31 ist ein Baum dargestellt, an dem das Prinzip der Suche durch den Lösungsraum erläutert wird. Die Knotennummern geben die Reihenfolge an, in der die Knoten besucht werden. Jeder Knoten im Baum korrespondiert zu einer Schaltung, die durch

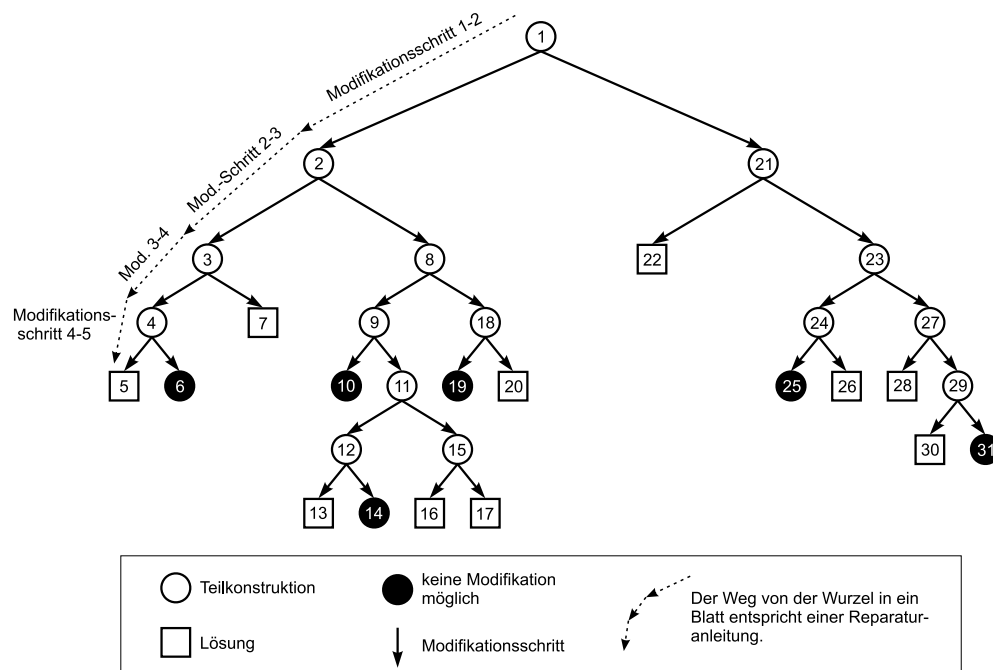


Abbildung 31: Die Suche im Lösungsraum

eine Modifikation entstanden ist. Diese Modifikation wird im Knoten gespeichert. In der Wurzel des Baumes steht der ursprüngliche Schaltplan. Ein Weg von der Wurzel in ein Blatt (Quadrat) des Baumes beschreibt eine Folge von Modifikationen, die den Schaltplan fehlerfrei macht. Dieser Weg kann deshalb als *Reparaturanleitung* verstanden werden.

Kann das Symptom eines Fehlers in einem Schaltplan (Knoten) durch keine Modifikation abgestellt werden, muß ein Backtracking⁷ einsetzen, bei dem die Modifikation eines Knotens zurückgenommen wird. Dann wird in dem Vaterknoten die nächste Modifikationsmöglichkeit angewendet (der nächste Sohn), die vielleicht einen Erfolg verspricht. So fortfahrend kann der ganze Lösungsraum nach Lösungen abgesucht werden. Darum werden alle Lösungen gefunden, unter denen sich die bezüglich der Wissensbasis optimale befindet.

Wenn eine Bewertungsfunktion gefunden werden kann, die eine Aussage bezüglich der Qualität einer gefundenen Teilkonstruktion (Knoten im Baum) macht, brauchen Berechnungszweige, die „teurer“ sind als schon gefundene, gar nicht erst verfolgt werden. Das kann die Suche nach einer guten Konstruktion im Lösungsraum erheblich beschleunigen. In der momentanen Version werden die Teilkonstruktionen nicht bewertet, so daß alle Lösungen gefunden werden.

Der Lösungsraum, also die Zahl der prinzipiell möglichen Konstruktionen, ist oftmals sehr groß. Hieraus folgt, daß der Kontrolle als Steuerung der Suche im Lösungsraum eine besondere Bedeutung zukommt.

8.4 Kontrolle und Kontrollwissen

Die Kontrolle kann intuitiv als „Steuerung der Suche im Lösungsraum“ beschrieben werden. In [Günt92] wurde der Begriff Kontrolle wie folgt definiert:

„Mit dem Begriff Kontrolle bezeichnen wir alles, was sich auf den dynamischen Ablauf während einer Problemlösung bezieht (Ablaufsteuerung). Kontrolle kann als Steuerung des Konstruktionsvorgangs (allgemein des Problemlösungsprozesses) verstanden werden, oder anders ausgedrückt, als Steuerung der Suche im Lösungsraum.“

Das Wissen darüber, wie die Suche im Lösungsraum durchgeführt werden soll, heißt Kontrollwissen und wurde von Günter wie folgt definiert:

„Kontrollwissen definieren wir als Wissen zur expliziten Steuerung des Konstruktionsprozesses. Die Kontrollentscheidungen basieren auf dem Kontrollwissen.“

Kontrollwissen ist demnach als „Wissen über die Anwendung und Verarbeitung von Wissen“ zu verstehen.

Für die Verarbeitung des Designwissens sind somit die

⁷Wir benutzen chronologisches Backtracking (s. [Günt92])

- *Reihenfolge der Regeln*, wie sie in einer Wissensklasse aufgeschrieben sind, und innerhalb einer Regeln die
- *Reihenfolge der Modifikationsmöglichkeiten*

zu sehen. Anhand dieser Reihenfolgen werden die Fehlverhalten (Symptome) und Modifikationen abgearbeitet (s. Abschnitt 5.1.1).

Bei der Diskussion mit dem Hydraulikingenieur hat sich herausgestellt, daß beim Auftreten mehrerer Anforderungsverletzungen, eine Reihenfolge bestimmt werden kann, in der man den Fehlfunktionen entgegenwirkt. Hier wird die Strategie des Ingenieurs übernommen, eine Lösung mit möglichst wenig Iterationsschritten zu bekommen:

1. Betrachte alle leistungsbezogenen (***strict***) Anforderungen in der Reihenfolge
 - (a) Leistungsquellen und -senken (Pumpe, Zylinder)
 - (b) Leistungsüberträger (Rohre, Schläuche)
 - (c) Leistungssteuerung (Ventile)
2. Gleiche die Anforderungen an die Regelung ab (***possible, welcome***).

Das Zeitverhalten und die Genauigkeit der Regelung können sowohl durch Veränderung der Schaltplantopologie als auch durch Modifikation der Regelungsstrategie beeinflußt werden.

Daß die Kontrolle mit dem dazugehörigen Kontrollwissen eine große Rolle bei der Verarbeitung des Designwissens spielt, hat sich im Laufe der Diskussion herauskristallisiert. Die Art des Kontrollwissens, wie wir es für die Verarbeitung verwenden, ist nicht vom Problem vorgegeben. Es kann auch auf andere Weise vorgegeben werden (s. Kapitel 10).

8.5 Blackboard-Architektur

Blackboard-Architekturen sind ein interessantes Konzept zur Realisierung einer flexiblen Kontrolle. Aus diesem Grund haben wir uns für die Verarbeitung des Designwissens für eine solche Architektur entschieden.

Ausgehend vom theoretischen Blackboard-Modell werden wir die notwendigen Anpassungen für unser spezielles Blackboard-System entwickeln.

Am Ende dieses Abschnittes wird dann eine Systemarchitektur vorgestellt, die die Repräsentation und Verarbeitung von Designwissen ermöglicht.

8.5.1 Konzept des theoretischen Blackboard-Modells

Das Konzept des theoretischen Blackboard-Modells besteht aus einer gemeinsamen, globalen Datenbasis, dem *Blackboard*, und einer Anzahl von *Wissensquellen*⁸, die unabhängig

⁸Sie werden auch als Spezialisten bezeichnet.

voneinander bestimmte Teilaufgaben bearbeiten. Die Wissensquellen kommunizieren ausschließlich über das Blackboard miteinander. Sie können vom Blackboard lesen und es auch modifizieren. Eine Wissensquelle besteht aus einem Bedingungs- und einem Aktionsteil. Sie kann aktiv werden, wenn ihr Bedingungsteil erfüllt ist. Die einzelnen Wissensquellen sind unabhängig voneinander und können parallel arbeiten.

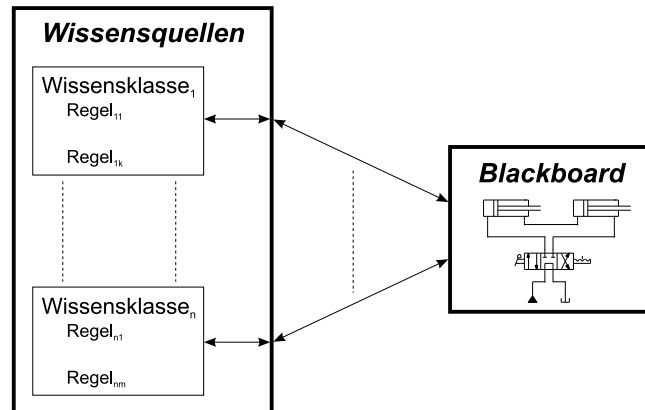


Abbildung 32: Das elementare Blackboard-Modell

Weil in diesem Ansatz keinerlei Kontrollfluß vorgegeben ist, entscheiden die Wissensquellen selbständig, wann sie aktiv werden. Abbildung 32 illustriert diese elementare Blackboard-Architektur. Anschaulich kann man sich das Blackboard als Diskussionsrunde von Experten vorstellen, die gemeinsam eine Lösung für ein Problem suchen und dabei durcheinander reden dürfen (s. Abbildung 33). Da im elementaren Blackboard-Modell die Wissensquellen

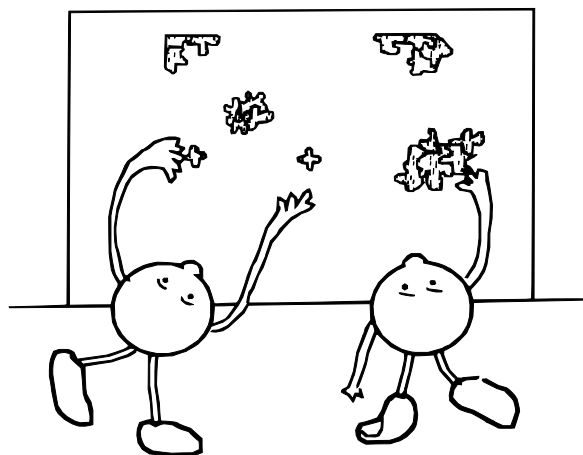


Abbildung 33: Die Zusammenarbeit von Experten ([Enge88])

selber entscheiden, wann sie aktiv werden (es findet kein Kontrollfluß statt), kann es zu

Ineffizienz kommen ([Günt92]).

Scheduler und Agenda

Die Lösung des Problems ist die Einführung einer Kontrollkomponente (Scheduler), die die Wissensquellen gezielt aktiviert. Als Grundlage für die Auswahl einer Wissensquelle dient eine Liste der aktivierbaren Wissensquellen (Agenda). Eine Wissensquelle ist aktivierbar, wenn ihr Bedingungsteil erfüllt ist. Im Bezug auf die diskutierenden Experten entspricht der Scheduler einem Diskussionsleiter, der einen sich meldenden Experten auswählt.

Kontroll-Blackboard

Eine weitere Erweiterung des elementaren Blackboard-Modells ist das Kontroll-Blackboard, das der Scheduler für die Entscheidung heranzieht, welche Wissensquelle aus der Agenda ausgewählt wird. Auf dem Kontroll-Blackboard befindet sich das Kontrollwissen, welches Informationen über die Anwendbarkeit der Wissensquellen beinhaltet.

Die Abbildung 34 zeigt das um den Scheduler, die Agenda und ein Kontroll-Blackboard erweiterte Blackboard-Modell.

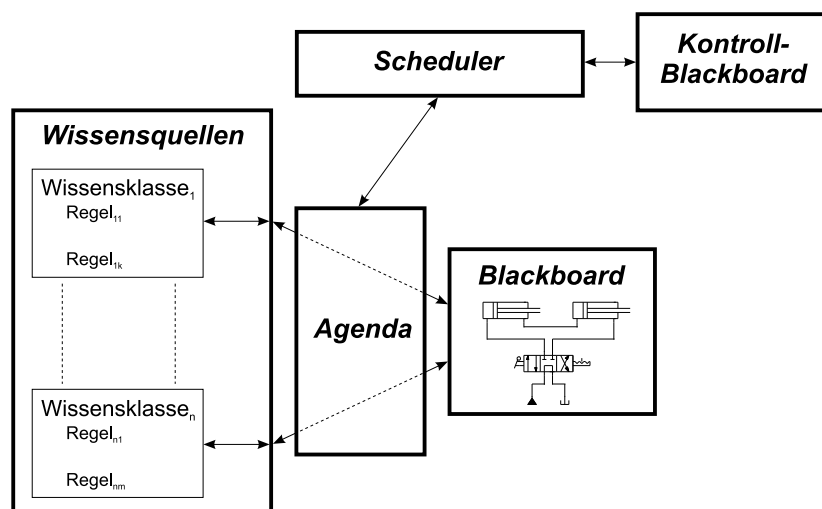


Abbildung 34: Erweitertes Blackboard-Modell (1)

Fokus

Da die Anzahl der Wissensquellen sehr groß werden kann, aber nicht immer alle gleichzeitig betrachtet werden müssen, kann ein Fokus eingebaut werden, der dann nur bestimmte Wissensquellen betrachtet. Dadurch bleibt die Menge der betrachteten Wissensquellen klein, und Ineffizienz wird vermieden. Abbildung 35 zeigt das komplette Blackboard-Modell mit allen geschilderten Erweiterungen.

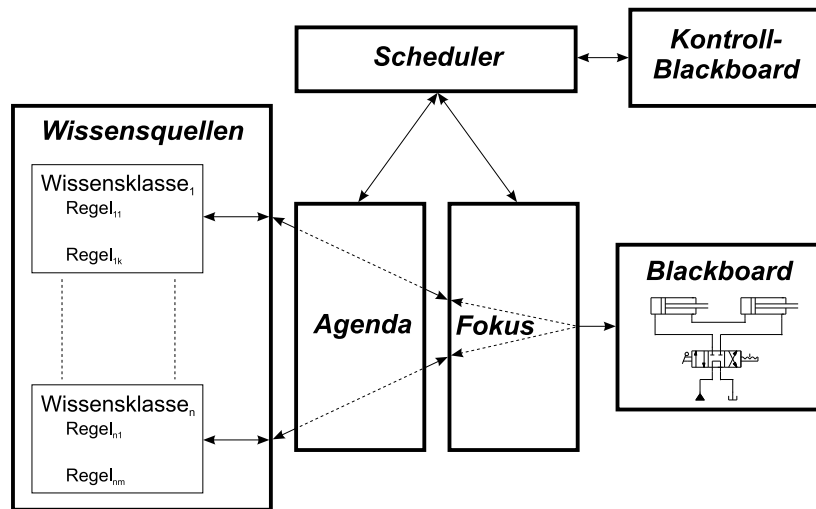


Abbildung 35: Erweitertes Blackboard-Modell (2)

8.6 Das spezielle Blackboard-Modell

Der Bezug des speziellen Blackboard-Modells für die Verarbeitung des Designwissens und dem theoretische Blackboard-Modell, welches in Abschnitt 8.5 erklärt wurde, wird nun mit den folgenden Betrachtungen hergestellt.

Das Konfigurationsobjekt, auf dem das Designwissen angewendet werden soll, ist der Schaltplan. Demzufolge wird er auf das *Blackboard* gelegt (s. Abbildung 35). Er besteht aus einzelnen Komponenten, die das Blackboard in mehrere Bereiche aufteilt.

Diese Bereiche können durch die Wissensquellen (Experten für die Komponenten) beeinflusst werden. *Wissensquellen* sind also gleich zu setzen mit Wissensklassen der Wissensbasis. In ihnen ist das Expertenwissen über eine Klasse von gleichartigen Objekten zusammengefaßt und kommt dem anschaulichen Vergleich eines Experten für diese Komponentenklasse sehr nahe. Eine Wissensquelle wird dann aktiv, wenn eine Regel der entsprechenden Wissensklasse angewendet werden kann.

Die *Agenda* des speziellen Blackboard-Modells kann sehr einfach zusammengestellt werden. In ihr werden alle Regeln der Wissensquellen gesammelt, deren symptoms-Teil (Bedingungsteil) erfüllt ist. Eine Wissensquelle kann durchaus durch mehrere Regeln gleichzeitig aktiviert werden, deren symptoms-Teil erfüllt ist.

Die Aufgabe des *Schedulers* ist klar definiert. Er soll die Regeln aus der Agenda auswählen, die angewendet werden sollen.

An dieser Stelle kommt das Konzept des Blackboard-Modells erst richtig zum Tragen. Die Idee des Blackboard-Modells zielt darauf, daß mehrere Wissensquellen gleichzeitig (parallel)

angewendet werden können. Dabei dürfen sich die Modifikationen der Wissensquellen nicht gegenseitig beeinflussen. Daß bei so einem Vorgehen Zeit gespart werden kann, liegt auf der Hand.

Damit der Scheduler nicht alle Regeln einer Wissensklasse (-quelle) betrachten muß, greift er auf das Kontrollwissen des Kontroll-Blackboards zu.

Anhand dieses Kontrollwissens entscheidet er, ob eine Regel angewendet werden darf oder nicht.

Das *Kontrollwissen* über die Anwendung der Regeln in den Wissensklassen ist implizit in den Klassen definiert. Allein die Reihenfolge der Regeln in einer Wissensklasse und die Reihenfolge der Modifikationsmöglichkeiten in einer Regel bestimmen ihre Anwendungspriorität (s. Abschnitt 5.1.1). In Kapitel 10 werden einige andere Ansätze für das Kontrollwissen vorgeschlagen.

Der Scheduler schränkt die Menge der ausführbaren Regeln ein. Durch den *Fokus* des speziellen Blackboard-Modells wird die Menge der Regeln noch weiter eingeschränkt. Da jede Regel mit einer Priorität ***strict***, ***possible*** oder ***welcome*** versehen ist (s. Abschnitt 5.1.1 und 8.4), werden sie in der Reihenfolge

1. ***strict***
2. ***possible***
3. ***welcome***

abgearbeitet. D.h., solange noch aktive strict-Regeln vorhanden sind, werden ausschließlich diese betrachtet; sind keine strict- aber noch possible-Regeln aktiv, werden diese angewendet und zum Schluß, wenn keine strict- und possible-Regeln mehr aktiv sind, werden die welcome-Regeln behandelt.

Aktiviert die Anwendung einer Regel mit einer niedrigeren Priorität eine Regel mit einer höheren Priorität, wird durch die Rücknahme der Modifikation (Backtracking) der Ursprungszustand wiederhergestellt. So werden dann auch Endlosschleifen bei der Verarbeitung vermieden. Wenn keine strict-Regeln mehr aktiv sind, bedeutet das, daß der Schaltkreis ausgelegt ist und seine geforderte Funktion erfüllt. Wenn zusätzlich keine possible-Regeln aktiv ist, konnten die in der Wissensbasis formulierten Optimierungen vorgenommen werden. Falls gleichzeitig keine welcome-Regeln aktiv ist, kann der Schaltkreis bezüglich der Wissensbasis als optimal ausgelegt angesehen werden. Das bedeutet aber nicht, daß diese eine gefungene Lösung — es gibt ja im allgemeinen mehrere — auch die bezüglich des Lösungsraumes globale optimale Lösung ist.

Die Anzahl der aktiven Regeln gibt daher ein Maß für die Qualität der Auslegung an. Die Bewertung der Regeln kann ihrer Priorität nach vorgenommen werden. Demnach gilt eine Schaltung als fehlerhaft, falls mindestens eine strict-Regel aktiv bleibt. Sind es mehr,

kann von gravierenden Konstruktionsfehlern gesprochen werden.

Je weniger Regeln aktiv bleiben, desto besser ist die gefundene (Teil-) Konstruktion.

8.6.1 Kontrollschleife

Die Kontrollschleife legt fest, in welcher Reihenfolge die Module der Verarbeitungskomponente arbeiten. Sie legt die grobe Struktur des Verarbeitungsalgorithmus fest. In Abbildung 36 ist die Kontrollschleife für die Verarbeitung von Designwissen dargestellt, wie sie von uns implementiert wurde. Beim Einstieg in die Schleife werden die aktiven Regeln der Wissens-

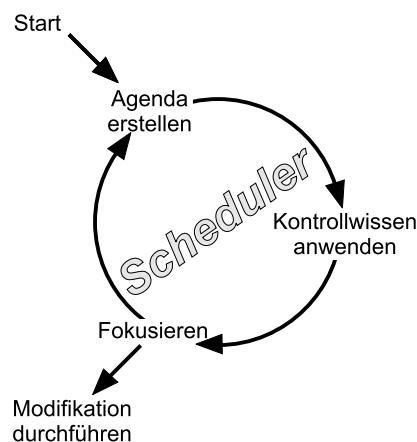


Abbildung 36: Die Kontrollschleife des Schedulers

quellen berechnet, die in der Agenda aufgeführt werden. Danach entscheidet der Scheduler, welche aktivierten Regeln unabhängig voneinander sind und weiter betrachtet werden. Der Fokus überprüft, welche dieser Regeln ihrer Priorität nach (*strict*, *possible*, *welcome*) betrachtet werden dürfen. Erst jetzt können die Regeln angewendet werden. Dann schließt sich die Kontrollschleife.

8.7 Die Verarbeitungskomponente

Nachdem wir das allgemeine Blackboard-Modell für die Verarbeitung von Designwissen angepasst haben, können wir die Verarbeitungskomponente genauer spezifizieren. Bei dieser Spezifikation werden die verschiedenen Details analysiert, die bei der Verarbeitung berücksichtigt werden müssen. Anhand eines (Programm-)Ablaufdiagrammes in Abbildung 37 soll die Funktionsweise des implementierten Verarbeitungsmoduls beschrieben werden.

Simulation bis Zustandsänderung

Basis für die Verarbeitungskomponente ist das ArtDeco-System. Mit ihm kann eine Hydraulikschaltung simuliert werden.

Diese Simulation wird angehalten, sobald eine Zustandsänderung mindestens einer Komponente festgestellt wird. In dieser Pause kann mit Hilfe der Simulationsparameter auf den

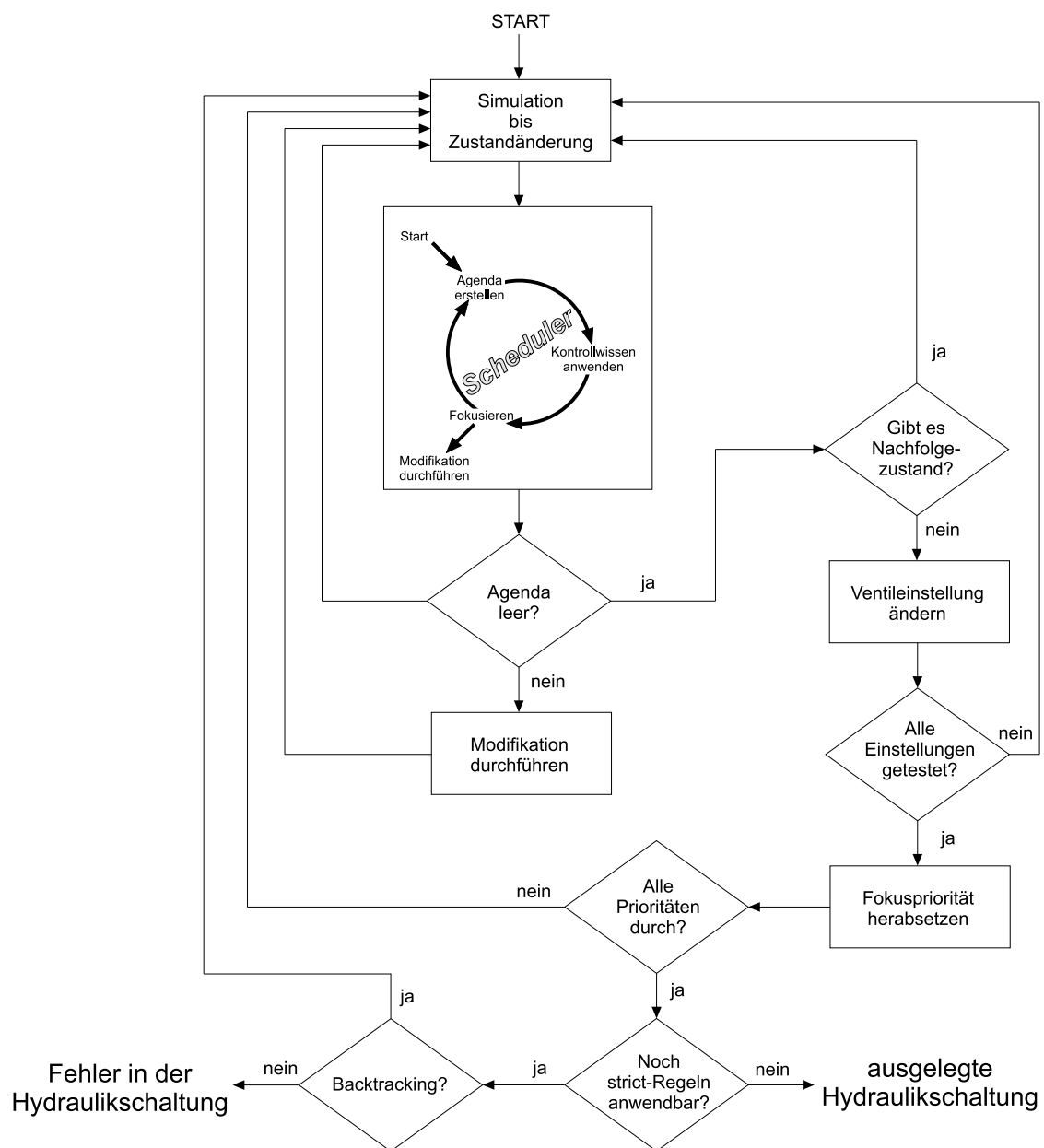


Abbildung 37: Ablaufdiagramm des implementierten Verarbeitungsmoduls

Zustand jeder Komponente geschlossen werden. An dieser Stelle beginnt die Arbeit der Kontrolle (Blackboard-Modell), die die anwendbaren Regeln (aktiven Wissensquellen) berechnet.

Kontrolle

Das Blackboard-Modell wurde in 8.6 ausführlich beschrieben, so daß hier nur die wesentli-

chen Aktionen der Kontrollschleife aufgeführt werden.

In einer Simulationspause berechnet der Scheduler des Blackboard-Modells die Agenda, die die anwendbaren Regeln der Wissensklasse enthält. Dabei werden nur die Regeln berücksichtigt, die aufgrund des Kontrollwissens und nach der Fokussierung in Frage kommen.

Modifikationen durchführen

Falls die Agenda nicht leer ist, d.h. es gibt anwendbare Regeln, werden die Modifikationen der Regeln durchgeführt. Anschließend wird das Verarbeitungsmodul mit dem veränderten Schaltplan rekursiv aufgerufen. D.h. die Simulation wird gestartet, um weitere Symptome zu bekämpfen.

Falls die Agenda leer ist, sind keine Regeln anwendbar. Somit ist bis zum aktuellen Simulationszustand der Schaltplan fehlerfrei. Es kann zum nächsten Zustand übergegangen werden, der durch eine Umstellung eines Ventils oder durch Fortsetzung der Simulation erreicht wird.

Ventileinstellungen ändern

Falls es keinen Nachfolgezustand gibt, muß die Ventileinstellung eines Ventils verändert werden. Erst dann ändert sich der Simulationszustand und das Verarbeitungsmodul kann rekursiv aufgerufen werden. Dabei ist zu berücksichtigen, daß alle möglichen Schalterstellungskombinationen aller Ventile geprüft werden müssen. Bei drei Ventilen mit je zwei Schaltstellungen ergeben sich somit $2^3 = 8$ Schaltstellungen.

Falls alle Schaltstellungen geprüft wurden, kann die Fokuspriorität herabgesetzt werden.

Fokuspriorität um eine Stufe zurücksetzen

Wenn die Fokuspriorität (*strict*, *possible*, *welcome*) herabgesetzt wird, ist keine Regel mit höherer Priorität anwendbar.

Dann können die Regeln mit der neuen Priorität behandelt werden. Dazu wird das Verarbeitungsmodul rekursiv aufgerufen (innerer Knoten im Baum; s. Abbildung 31 auf S. 69). Produziert eine Modifikationsmaßnahme während dieses Aufrufs neue Fehler einer höheren Priorität oder wesentlich mehr Fehler der gleichen Priorität als es vorher waren, so gilt dieser Aufruf als gescheitert und die Modifikationsmaßnahme muß zurückgenommen werden (Backtracking). Auf diese Weise ist der Rekursion eine Grenze gesetzt, so daß Endlosrekursionen verhindert werden.

Bevor ein Backtracking einsetzt wird überprüft, ob es aktive *strict*-Regeln für die aktuelle Schaltung gibt. Ist dies der Fall, sind wesentliche Fehler vorhanden, und die Schaltung kann als fehlerhaft angesehen werden (schwarzer Knoten im Baum). Dann muß das Backtracking sofort einsetzen.

Sind keine *strict*-Regeln aktiv, handelt es sich bei der Schaltung um eine Lösung (Blatt im Baum). D.h., der Schaltplan kann bezüglich der Wissensbasis ausgelegt werden und wird als Lösung ausgegeben. Die Anzahl der dann noch aktiven *possible*- und *welcome*-Regeln ist dann ein Maß für die Qualität der Lösung.

9 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung einer Sprache, in der der Hydraulikingenieur sein Expertenwissen auf einfache Weise formulieren kann. Außerdem sollte ein Verfahren entwickelt werden, welches diese Sprache und somit das Wissen des Ingenieurs verarbeitet.

In der Diskussion mit einem Hydraulikingenieur hat sich herausgestellt, daß er sein Wissen in natürlicher Sprache formuliert. Diese Formulierung ist im allgemeinen zu informal, so daß eine formale Sprache, die Designsprache, entwickelt wurde, die den Ingenieur quasi dazu zwingt, sich formal auszudrücken. Da die Konzepte der Designsprache so einfach sind und nahe an der Problemstellung liegen, können die umgangssprachlichen Formulierungen des Ingenieurs in den meisten Fällen schnell in die Designsprache übersetzt werden.

Hat der Ingenieur sein Expertenwissen in der Designsprache formuliert, kann eine computerunterstützte Auslegung das Vorgehen des Ingenieurs nachahmen. Die Verarbeitungskomponente, die die Auslegung realisiert, besteht aus zwei verschiedenen Komponenten, dem Parser und dem Interpretierer. Der Parser überprüft den vom Ingenieur geschriebenen Quellcode auf lexikalische und syntaktische Fehler, und der Interpretierer führt die Befehle und Makros des Quellcodes aus. Eine weitere Aufgabe des Interpretierers ist die Kontrolle der Suche im Lösungsraum.

Zusammen mit dem Parser und der Designsprache setzt er das in Abschnitt 8.5 entwickelten Blackboard-Modell um.

Im Zusammenhang dieser Diplomarbeit ist der in Abschnitt 8.1 beschriebene Parser softwaremäßig umgesetzt worden. Der Quellcode des Parsers ist so aufgebaut, daß eine Änderung der Designsprache durch einfaches Hinzunehmen oder Entfernen von Funktionen, die die Regeln der BNF umsetzen, schnell möglich ist.

Ebenso verhält es sich mit dem in Abschnitt 8.2 beschriebenen Interpretierer. Da er ähnlich wie der Parser arbeitet, können auch hier Veränderungen der Sprache leicht vorgenommen werden. Das im Interpretierer verankerte Blackboard-Modell hält sich an die im Abschnitt 8.6 erarbeiteten Ergebnisse.

Die Beispiele in Anhang B haben gezeigt, daß mit der Softwarelösung, die im Zusammenhang dieser Arbeit programmiert wurde, eine computerunterstützte Auslegung von Hydraulikschaltungen prinzipiell möglich ist.

Verbesserungen zur Steigerung der Effizienz sind im Bereich der Suche im Lösungsraum möglich. Da wir den gesamten Lösungsraum absuchen, werden, auf Kosten der Laufzeit, alle Lösungen gefunden. Die Antwortzeit für einen auszulegenden Schaltplan hält sich aber trotzdem in Grenzen (einige Sekunden).

Dennoch gibt es Grenzen des Modifikationsansatzes der im Grobentwurf gegebenen Hydraulikschaltung.

Die wissensbasierte Projektierungsunterstützung setzt implizit einen Grobentwurf der Anlage voraus, der prinzipiell funktionsfähig ist, bzw. in die richtige Richtung geht. Es ist weder beabsichtigt noch denkbar, von einem beliebigen Ausgangspunkt ausgehend beliebige Aufgabenstellungen zu lösen.

Über die computerunterstützte Auslegung läßt sich eine grundsätzliche Aussage treffen:

Die Qualität der computerunterstützten Auslegung einer Hydraulikanlage hängt wesentlich von der Qualität des Expertenwissens in der Wissensbasis ab.

D.h., enthält die Wissensbasis semantische Fehler oder Modifikationen, die zwangsläufig neue Fehler produzieren, zieht das sofort Qualitätseinbußen der Ergebnisse nach sich oder äußert sich in der Antwortzeit der Verarbeitungskomponente.

Zum Abschluß werden im nächsten Kapitel noch einige weiterführende Gedanken erläutert. Diese können als Erweiterungsvorschläge für weitere Versionen der Designsprache angesehen werden.

10 Ausblick

Die vorgestellte Version der Designsprache stellt nur die grundlegenden Konzepte und Strukturen zur Formulierung von Designwissen zur Verfügung. An dieser Stelle sollen jetzt einige weiterführende Gedanken zur Erweiterung der Sprache diskutiert werden, die dann in neuere Versionen aufgenommen werden können.

Wie wir in Abschnitt 5.1.1 gesehen haben, werden die Modifikationen einer Regel in der Reihenfolge abgearbeitet, wie sie im Programmcode aufgeführt sind. Dieses Vorgehen ist nicht zufriedenstellend, da beim Einfügen einer neuen Modifikation die richtige Stelle in der Regel gesucht werden muß. Hier hilft die Vergabe von Prioritäten an die Modifikationen. Diese Priorität berechnen sich nach [Ueck97] aus dem Aufwand, der Rückwirkung, der Art des Eingriffs und der Wirksamkeit der Modifikation. Anhand dieses Wertes können die Modifikationsmöglichkeiten sortiert werden und in der Reihenfolge der Prioritäten ausführen werden. Dann entfällt die Suche nach der richtigen Stelle für eine neue Modifikation im Programmcode.

Die jetzige Version der Sprache ermöglicht eine Auslegung der Komponenten einer hydraulischen Schaltung derart, daß das Gesamtsystem den Anforderungen entspricht, und fehlerfrei funktioniert. Die Komponenten selber sind allerdings nicht real, d.h. sie stehen dem Konstrukteur in den meisten Fällen nicht wie berechnet zur Verfügung, da sie vom Hersteller nicht mit den berechneten Kenngrößen hergestellt werden. Er liefert im allgemeinen Standardkomponenten, die aufgrund der Massenproduktion wesentlich günstiger sind als Spezialanfertigungen. Daher wäre es wünschenswert, am Ende des Auslegungsprozesses eine Liste von Standardkomponenten zu bekommen, aus denen die Anlage dann montiert werden kann. Selbstverständlich soll auch diese Anlage einwandfrei funktionieren. Hier setzt nun der zweite Erweiterungsvorschlag an. Damit dieses Ziel erreicht wird, muß die Auslegung der Anlage in zwei Phasen aufgeteilt werden. In der ersten Phase werden die Komponenten so ausgelegt, daß das System fehlerfrei funktioniert. In der zweiten Phase werden dann für diese ausgelegten Komponenten Standardkomponenten aus Herstellerlisten so ausgewählt, daß die Anlage mit ihnen ebenfalls fehlerfrei funktioniert.

Ein weiterer wesentlicher Punkt ist der, daß in der Industrie darauf geachtet werden muß, daß nicht nur eine funktionierende Anlage konstruiert wird, sondern daß die Kosten auch so niedrig wie möglich gehalten werden. Um dieses Optimierungsziel zu erreichen, kann eine Erweiterung der Regeln in Betracht gezogen werden, mit deren Hilfe dann aktuelle Preistabellen verschiedener Hersteller abgefragt werden. Mit diesen Informationen kann dann die günstigste Konfiguration der verwendeten Komponenten ermittelt werden.

Ein Nachteil dieses Vorgehens ist der Zugriff auf die aktuellen Preistabellen der Hersteller. Aufgrund von Preisänderungen müßte die Wissensbasis sehr häufig aktualisiert werden, was den Wartungsaufwand unnötig in die Höhe treibt. Eine Lösung dieses Problems wäre die Anbindung der Wissensbasis an die Datenbanken der Hersteller. Dann können ohne viel Aufwand die aktuellen Preise abgefragt werden.

Der dritte Erweiterungsvorschlag, der an dieser Stelle aufgeführt werden soll, bezieht sich auf die Reihenfolge der Anwendung der verschiedenen Modifikationsteile einer Regel. Im Moment werden sie in der Reihenfolge der Priorität, d.h. in der Reihenfolge, wie sie im Programmcode aufgeführt sind, abgearbeitet. Wenn eine Lösung erst ganz am Ende beim Durchsuchen des Lösungsraumes gefunden wird, impliziert das sofort hohe Rechenzeiten. Wird bei der Auslegung einer anderen Anlage die gleiche Regel unter den gleichen Bedingungen gefeuert, muß *wieder* der gesamte Lösungsraum durchsucht werden, obwohl eigentlich klar ist, welche Modifikation in diesem Fall am schnellsten zu einer Lösung führt. Wenn sich die Verarbeitung „merken“ würde, welche Aktionen unter welchen Bedingungen am geeignetsten sind, um am schnellsten ans Ziel zu kommen, können die Rechenzeiten zum Finden von guten Lösungen erheblich verkürzt werden. Eine gute Lösung kann dann auch ohne den gesamten Lösungsraum zu durchsuchen schnell gefunden werden. Voraussetzung für diesen Ansatz ist allerdings, daß die Verarbeitung an genügend vielen unterschiedlichen Beispielen „lernt“, welche Modifikation in welcher Situation am geeignetsten ist.

Der letzte Erweiterungsvorschlag bezieht sich direkt auf die Designsprache. In dieser Arbeit ist der Begriff der objektorientierten Wissensklassifizierung eingeführt worden. D.h., das Wissen über gleichartige Objekte wird in einer Wissensklasse zusammengefaßt. Dieses Wissen gilt dann für alle Instanzen einer Wissensklasse. Dieses Konzept ist den objektorientierten Programmiersprachen sehr ähnlich wobei die Regeln mit ihren Modifikationen den Methoden der objektorientierten Sprachen entsprechen. Die Erweiterung zielt nun in die Richtung des mächtigen Konzeptes der Vererbung in objektorientierten Systemen. Mit ihr kann der Wissensakquisiteur Vererbungshierarchien aufbauen und Beziehungen zwischen den verschiedenen Komponentenklassen ausnutzen. Dabei müssen aber verschiedene Probleme (einfache-, mehrfache Vererbung) gelöst werden, die in [Schl93] näher beschrieben sind.

Literatur

- [Enge88] E. Englemore, T. Morgan. *Blackboard Systems* , Addison-Wesley 1988
- [Günt92] A. Günter. *Flexible Kontrolle in Expertensystemen zur Planung und Konfiguration in technischen Domänen*, Sankt Augustin: infix, 1992
- [Haye83] B. Hayes-Roth. *Te Blackboard Architecture: A General Framework for Problem Solving?* , Heuristic Programming Project, Report No. HPP-83-30, May 1983
- [Haye85] B. Hayes-Roth. *A Blackboard Architecture for Control* , Heuristic Programming Project, Stanford University 1985
- [Hopc90] John E. Hopcroft, Jeffrey D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie* , Addison-Wesley 1990
- [Inte84] *TellAndAsk Reference Manual* , IntelliCorp 1989
- [Kast95] U. Kastens. *Grundlagen der Programmiersprachen.*, Vorlesungsskript Sommersemester 1995
- [Klei92] H. Kleine Büning. *Wissensbasierte Systeme* , Vorlesungsskript Wintersemester 1992—93, Universität-GH Paderborn
- [Klei93] H. Kleine Büning. *Wissensbasierte Diagnose und Konfiguration* , Vorlesungsskript Sommersemester 1993, Universität-GH Paderborn
- [Koch97] H.-H. Kochsmeier. *Entwurf und Realisierung eines Werkzeuges zur Analyse komplexer analoger und gemischt analoger—digitaler Schaltungen* , Diplomarbeit 1997
- [Kraf97] C. Krafthöfer, E. Vier. *Untersuchung konstruktiver Maßnahmen zur Beeinflussung des dynamischen Verhaltens hydraulischer Antriebe*, Gerhard-Mercator-Universität-GH Duisburg, Studienarbeit 1997
- [Küsp90] H.-J. Küspert. *Folien zur Informatik A* , Universität-GH Paderborn, Wintersemester 1990—91
- [Leng91] Th. Lengauer. *Vorlesungsskript: Informatik C* , Universität-GH Paderborn, Wintersemester 1991—92
- [Mann93] Mannesmann Rexroth. *Hydraulik- und Elektronikkomponenten für Proportional- und Servo-Systeme*, Katalog RD 29003—04.93
- [OFT97] *OFT-Objectives and Concepts*, Department of Mathematics and Computer Science, University of Paderborn, 1997

- [Ottm90] T. Ottmann, P. Widmayer. *Algorithmen und Datenstrukturen* , Wissenschaftsverlag 1990
- [Pear84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving* , Addison-Wesley 1984
- [Pupp91] F. Puppe. *Einführung in Expertensysteme* , Springer-Verlag 1991
- [Schl93] Th. Schlotmann. *Vererbung in objektorientierten Systemen* , Seminararbeit zur Projektgruppe Konfiguration, Universität-GH Paderborn, Sommersemester 1993
- [Stro92] B. Stroustrup. *Die C++ Programmiersprache* , Addison-Wesley 1992
- [Ueck97] S. Uecker, E. Vier. *Statische Auslegung hydraulischer Translationsantriebe bei der Neu- und Änderungskonstruktion*, Gerhard-Mercator-Universität-GH Duisburg, Studienarbeit 1997
- [Vier97] E. Vier, B. Stein und M. Hoffmann. *Strukturelle Formulierung von Anforderungen an hydraulische Antriebe*. Gerhard-Mercator-Universität-GH Duisburg, Forschungsbericht Nr.8—97

A Sprach-Referenz

Diese Sprachreferenz gibt einen Überblick über die Konzepte und Kernfunktionen der Designsprache. Sie soll als Nachschlagewerk für den Anwender dienen und ihn schnelle Hilfe für die verschiedenen Konstrukte liefern.

A.1 Datentypen

Bei der Analyse der Wissensformen (Abschnitt 2.1) hat sich herausgestellt, daß die Designsprache mit wenig Datentypen auskommt.

A.1.1 *component*

component bezeichnet ein Datenobjekt, welches zu einer bestimmten Komponente im Schaltplan assoziiert. Innerhalb einer Wissensklasse wird das Objekt der Klasse selbst mit ***this*** bezeichnet. Der Wert von ***component*** ist ein String, der den Namen der Komponente enthält. Über diesen Namen können alle Parameter der Komponente mit **set()** und **get()** angesprochen werden.

A.1.2 *float*

Hier handelt es sich um das übliche Zahlenformat. Die Genauigkeit wird dabei in der Designsprache auf 15 Nachkommastellen festgelegt.

A.1.3 *boolean*

Hier handelt es sich um den üblichen binären Datentyp. Variablen dieses Typs können den Wert ***TRUE*** oder ***FALSE*** annehmen.

A.1.4 Typen der Simulationsparameter

In Abschnitt 3.1.2 wurde beschrieben, warum in der Designsprache zwischen zwei verschiedenen Parametertypen eines Komponentenparameters unterschieden werden muß.

parameter

Komponentenparameter vom Typ ***parameter*** bezeichnen die einstellbaren Parameter (s. Abschnitt 3.1.2) einer Komponente.

characteristic

Komponentenparameter vom Typ ***characteristic*** bezeichnen die Kenngrößen (s. Abschnitt 3.1.2) einer Komponente.

A.1.5 *list*

Bei der einfachen Selektion von Komponenten aus einem Schaltkreis werden im allgemeinen mehrere Komponenten gleichzeitig ausgewählt. Diese Komponenten werden in Listen gespeichert. Als Listenelemente werden nur Objekte vom Typ **component** und **list** zugelassen. Die leere Liste wird mit **nil** bezeichnet.

Hier nun eine rekursive Definition des Datentyps Liste:

Definition: *list*

1. **nil** bezeichnet die leere Liste vom Grad 0.
2. $(\text{objekt}_1, \dots, \text{objekt}_n)$ ist eine Liste vom Grad 1, dessen Objekte vom Typ **component** sind.
3. $(\text{element}_1, \dots, \text{element}_m)$ ist eine mehrelementige Liste, wobei element_1 bis element_n Listen vom selben Grad sind.

Gemischte Listen sind nicht erlaubt. Z.B. ist die Liste $(\text{Zylinder1}, (\text{Ventil1}, \text{Ventil2}))$ nicht erlaubt, wohingegen die Liste $((\text{Zylinder1}), (\text{Ventil1}, \text{Ventil2}))$ erlaubt ist. Die Listenelemente dürfen also immer nur vom selben Typ sein.

Repräsentiert eine Liste einen Pfad, gibt die Reihenfolge der Einträge in der Liste die Reihenfolge der Komponenten an, in der der Pfad durchlaufen wird. Besteht die Liste aus Listen, ist die Reihenfolge der Elemente nicht von Bedeutung. Die Liste ist ungeordnet.

A.2 Operationen auf Datentypen

In diesem Abschnitt werden die Operationen auf den verschiedenen Datentypen aufgelistet. Diese Operationen sind als Kernfunktionen implementiert und werden deshalb auch mit einem Funktionsaufruf angestoßen. Lediglich die Vergleichsoperatoren werden in Infix-Notation geschrieben und haben somit nichts mit Funktionsaufrufen gemein.

A.2.1 *component*

Beim Typ **component** werden mehrere Variablen unter einem Namen zusammengefaßt. Er wird zur Identifikation einer Komponente im Schaltplan benutzt. Diese Komponente kann dann über ihren Variablennamen angesprochen werden (z.B. **set()**). Für den Typ **component** sind folgende Makros erlaubt:

new (<i>t</i>)	erzeugt eine neue Komponente vom Typ <i>t</i> . In der Parameterliste wird nur der gewünschte Komponententyp angegeben.
set (<i>k</i> , <i>p</i> , <i>w</i>)	setzt den Parameter <i>p</i> der Komponente <i>k</i> auf den Wert <i>w</i>
get (<i>k</i> , <i>p</i>)	liefert den Wert des Parameters <i>p</i> der Komponente <i>k</i>
fulfills (<i>k</i> , <i>q</i>)	überprüft, ob Komponente <i>k</i> den Qualifier <i>q</i> erfüllt

Eine Komponente, die in den Schaltplan eingefügt werden soll, muß vorher erzeugt werden. Der **new**-Befehl erzeugt eine neue Komponente vom Typ t . Die Parameter der Komponente können nach **new()** mit **set()** verändert werden.

A.2.2 *float*

add(x, y) berechnet die Summe von x und y
mul(x, y) berechnet das Produkt von x und y
div(x, y) berechnet den Quotienten $\frac{x}{y}$
exp(x) berechnet den Exponenten e^x ;
wobei e die Eulersche Zahl ist
pow(x, y) berechnet den Exponenten x^y ;
demnach läßt sich die Wurzel von x vom Grad y
durch **pow**($x, \text{div}(1, y)$) berechnen
sin(x) berechnet den Sinus von x
cos(x) berechnet den Kosinus von x
tan(x) berechnet den Tangens von x
asin(x) berechnet den Arcussinus von x
acos(x) berechnet den Arcuskosinus von x
atan(x) berechnet den Arcustangens von x
sinh(x) berechnet den Sinushyperbolicus von x
cosh(x) berechnet den Kosinushyperbolicus von x
ln(x) berechnet den natürlichen Logarithmus von x
sqrt(x) berechnet die Quadratwurzel von x
abs(x) liefert den absoluten Wert von x

A.2.3 *list*

Da eine Liste einen Pfad repräsentieren kann und solche Pfade generiert werden müssen bevor sie in eine Schaltung eingefügt werden können, werden die folgenden Listenoperationen eingeführt. Mit Hilfe dieser Listenoperationen können dann beliebige Listen erstellt werden. Es handelt sich beim Typ **list** um einen abstrakten Datentyp ([Küsp90])

clr (l)	die Elemente der Liste l werden gelöscht. l ist anschließend leer.
append (l, k)	hängt das Element k hinten an die Liste an.
first (l)	liefert das erste Element der Liste l .
last (l)	liefert das letzte Element der Liste l .
delete (l)	entfernt das erste Element der Liste l .

Diese Operationen lassen sich auf alle erlaubten Elementtypen anwenden, also auf **component** und **list**.

A.2.4 *boolean*

Auf dem Typ ***boolean*** können die üblichen binären Operationen durchgeführt werden. Die Syntax dieser Operationen sieht wie folgt aus:

and (x, y)	x und y werden logisch mit und verknüpft.
or (x, y)	x und y werden logisch mit oder verknüpft.
exor (x, y)	x und y werden logisch mit exklusiv-oder verknüpft.
not (x)	x wird logisch negiert.

x und y sind dabei vom Typ ***boolean***.

A.3 Vergleichsoperatoren

Folgende Vergleichsoperationen sind erlaubt:

$x = y$	x und y sind im mathematischen Sinn gleich
$x < y$	x ist echt kleiner als y
$x > y$	x ist echt größer als y
$x \leq y$	x ist kleiner oder gleich y
$x \geq y$	x ist größer oder gleich y
$x \neq y$	x ist ungleich y

Die Vergleichsoperatoren werden in Infix-Notation geschrieben, da sie dann als Ausdrücke besser lesbar sind.

A.4 Hilfsvariablen

Manchmal müssen in Makros oder modification-Teilen Zwischenergebnisse gespeichert werden, die in Hilfsvariablen abgelegt werden können. Da eine Hilfsvariable keiner Komponente zugeordnet ist, unterscheidet sie sich von Komponentenparametern. Aus diesem Grund wird für Hilfsvariablen ein spezieller Operator definiert, mit dem ihr Wert verändert werden kann.

A.4.1 Beispiel zur Erzeugung einer Hilfsvariablen

$h := \text{new}(\text{pump_unit})$

Dieser Befehl erzeugt eine neue Komponente vom Typ *pump_unit*. Diese Pumpe kann nun mit **insert_component** in den Schaltplan eingefügt werden. Vorher können die Parameter von h mit dem **set**-Befehl verändert werden.

A.5 Ablaufstrukturen

Ablaufstrukturen sind ein wesentliches Konzept zur Formulierung von Algorithmen. Bei der Analyse der Wissensformen in Abschnitt 2.1 hat sich gezeigt, daß bei der Formulierung von Designwissen solche Ablaufstrukturen unerlässlich sind. Aus diesem Grund wurde eine geeignete Form in die Designsprache aufgenommen.

A.5.1 Schleifen

In einem Programm muß häufig derselbe Anweisungsblock auf mehrere Komponenten oder Komponentengruppen hintereinander angewendet werden. Hierzu hat sich in anderen Programmiersprachen das Konzept der Schleife etabliert. In die Designsprache soll dieses Konzept ebenfalls eingebettet werden.

foreach

Hier soll über die Einträge einer Liste gelaufen werden, auf die der Anweisungsblock angewendet werden soll. Die Syntax einer Listen-Schleife sieht wie folgt aus:

```
foreach element in liste{  
  ... // Anweisungen  
}
```

Mit dieser Schleife werden alle Elemente der Liste ihrer Reihe nach durchlaufen und auf jedes wird der Anweisungsblock ausgeführt. Der Typ des Elementbezeichners *element* wird dynamisch gebunden und hängt somit von der Struktur der Liste *liste* ab, die nicht von vornherein klar sein muß. Besteht die Liste aus Komponenten, ist *element* vom Typ **component**. Andernfalls ist *element* vom Typ **list**.

while

Mit der **while**-Schleife können Schleifen mit einer Abbruchbedingung formuliert werden. Weil diese Art der Schleifen so flexibel ist, können unter anderem Zählschleifen leicht realisiert werden. Die Syntax der **while**-Schleife sieht wie folgt aus:

```
while Ausdruck do{  
  ... // Anweisungen  
}
```

Dabei ist *Ausdruck* ein boolscher Ausdruck oder ein Makro, das ein Ergebnis vom Typ **boolean** liefert.

A.5.2 Bedingte Ausführung

Häufig sollen Anweisungen nur dann ausgeführt werden, wenn eine oder mehrere Bedingungen erfüllt sind. Um dies in der Designsprache zu realisieren wird das Standard **if** · · **then**-Konstrukt mit aufgenommen. In der Designsprache gibt es zwei Möglichkeiten die **if** · · **then** Anweisung zu verwenden:

1. **if** *Bedingung* **then**{Anweisungen}
2. **if** *Bedingung* **then** Makroaufruf

Die erste Methode ermöglicht einen Anweisungsblock hinter dem **then**. Dieser Block ist geklammert und wird ausgeführt, wenn *Bedingung* wahr ist.

Bei der zweiten Methode besteht der **then**-Teil nur aus dem Aufruf eines Makros. Diese Methode sollte immer dann angewendet werden, wenn die durchzuführende Aktion zu umfangreich ist, um als Modifikationsblock direkt hinter dem **then** aufgeschrieben zu werden. Dies würde den Code zusätzlich unleserlich machen.

A.6 Kernfunktionen

Kernfunktionen (Kapitel 5.1.2) sind fest in der Sprache verankerte Grundfunktionen, die in selbstdefinierte Makros verwendet werden dürfen.

A.6.1 Der &-Operator

Der &-Operator in der Parameterliste einer Makrodefinition leitet eine Qualifierliste ein. Diese Liste kann nur als letzter Parameter in einem Makro vorkommen. Mehr zu diesem Thema kann in Abschnitt 5.1.2 nachgelesen werden.

A.6.2 print()

Das `print()` Makro erlaubt die Ausgabe von Zeichenketten, die in Anführungszeichen geklammert sind und die Ausgabe von Parameterwerten. Die Anzahl der Parameter ist nicht fest vorgegeben, so daß verschieden viele Parameter gleichzeitig ausgegeben werden können.

A.6.3 exit()

Der Aufruf dieses Makros bewirkt, daß ein Programm sofort beendet wird. Der einzige Parameter des Makros ist vom Typ **float**. Für ihn wird für gewöhnlich der Wert 0 angegeben, falls das Programm korrekt beendet wurde. Alle von 0 verschiedenen Werte zeigen dann an, daß ein Fehler den Programmabbruch erforderte. Dabei wird nur der ganzzahlige Anzeil der Dezimalzahl berücksichtigt.

A.6.4 select_component

Mit `select_component()` werden einzelne Komponenten oder Pfade aus einem Schaltplan ausgewählt. Beispiele für die Verwendung dieses Makros sind in Abschnitt 6.4.2 aufgeführt.

A.6.5 delete_component

Mit `delete_component()` werden einzelne Komponenten oder Pfade aus einem Schaltplan entfernt. Beispiele für die Verwendung dieses Makros sind in Abschnitt 6.5.1 aufgeführt.

A.6.6 insert_component

Mit `insert_component()` werden einzelne Komponenten oder Pfade in eine Schaltung eingefügt. Beispiele für die Verwendung dieses Makros sind in Abschnitt 6.5.3 aufgeführt.

A.7 Formale Sprachdefinition in BNF

Nachfolgend wird die vollständige kontextfreie Grammatik der Designsprache in BNF angegeben.

Dabei sind in den Mengen **Letter** und **Digit** alle erlaubten Zeichen aufgeführt. Zusätzlich zu den in Hochkomma gesetzten Terminalen der Grammatik werden durch **Number**, **String** und **Identifizier** weitere Terminale spezifiziert. Dabei bezeichnet **Number** eine Dezimalzahl, **String** eine Zeichenkette und **Identifizier** einen Bezeichner. Strings müssen am Anfang und Ende durch Anführungszeichen gekennzeichnet werden und dürfen aus Buchstaben der Grammatik (**Letter**, **Digit**) bestehen. Das Startsymbol der Grammatik ist **Program**.

Jede Regel der Grammatik ist durch eine Funktion des recursive-descent-Parsers implementiert und hat im Quellcode des Parsers den Namen des Regelkopfes.

Letter ∈ { 'a', ..., 'z', 'A', ..., 'Z', ' _', '""', '□' | '□' ≡ Blank }
 Digit ∈ { '1', ..., '0' }

Digits	::= Digit Digits Digit .
Number	::= Digits ' .' Digits Digits '-' Digits ' .' Digits '-' Digits .
String	::= Letter String Digit String Letter Digit .
IdentTail	::= Letter IdentTail Digit IdentTail .
Identifizier	::= Letter IdentTail .
Identifiers	::= Identifizier ' ,' Identifiers Identifizier .
Type	::= 'component' 'float' 'boolean' 'list' .
TIdentifizier	::= Type '□' Identifizier .
TIdentifiers	::= TIdentifizier ' ,' TIdentifiers TIdentifizier .
CompOpr	::= '=' '<' '>' '<=' '>=' '<>' .
Exp	::= Identifizier CompOpr Identifizier Number CompOpr Identifizier Identifizier CompOpr Number Macrocall CompOpr Macrocall Number CompOpr Macrocall Macrocall CompOpt Number Identifizier CompOpr Macrocall Macrocall CompOpr Identifizier .
Exps	::= Exp ' ,' Exps Exp .
Qualifier	::= Identifizier CompOpr Identifizier Number CompOpr Identifizier Identifizier CompOpr Number .
Qualifiers	::= Qualifier ' ,' Qualifiers Qualifier .
What	::= 'in_series' 'in_parallel' .
Path	::= Identifizier '□' Identifizier Identifizier '□' (' Qualifiers ')

	'(' Qualifiers ')' Identifier '(' Qualifiers ')' Identifier '(' Qualifiers ')' .
WafBe	::='after' 'before' .
AfterBefore	::=WafBe Identifier WafBe Identifier '(' Qualifiers ')' .
Series_Parallel_I	::='(' Qualifiers ')' What Identifier Path What Identifier Path ',' AfterBefore AfterBefore ',' What Identifier Path '(' Qualifiers ')', What Identifier Path What Identifier Path '(' Qualifiers ')' , '(' Qualifiers ')', What Identifier Path ',' AfterBefore '(' Qualifiers ')', AfterBefore ',' What Identifier Path What Identifier Path ',' AfterBefore '(' Qualifiers ')' , AfterBefore ',' What Identifier Path '(' Qualifiers ')' , What Identifier Path '(' Qualifiers ')', AfterBefore AfterBefore '(' Qualifiers ')', What Identifier Path .
Series_Parallel_II	::=What Identifier Path ',' List List ',' What Identifier Path What Identifier Path ',' List List ',' What Identifier Path ',' AfterBefore List ',' AfterBefore ',' What Identifier Path What Identifier Path ',' AfterBefore ',' List AfterBefore ',' What Identifier Path ',' List What Identifier Path ',' List ',' AfterBefore AfterBefore ',' List ',' What Identifier Path .
List	::='(Identifiers)' '()' .
LocationSpecifier_I	::=Series_Parallel_I .
LocationSpecifier_II	::=Series_Parallel_II .
Parameter	::=Macrocall Identifier Number List .
MParameters	::=Parameter ',' MParameters Parameter .
IdorNumorMaC	::=Identifier Number Macrocall .
IdorListorMaC	::=Identifier List Macrocall .
IdorQual	::=Identifier '(' Qualifiers ')' .
Macrocall	::=Identifier '(' MParameters ')' Identifier '(' MParameters ',' Qualifiers ')' Identifier '(' Qualifiers ')' 'select_component(' LocationSpecifier_I ')' 'delete_component(' LocationSpecifier_I ')' 'insert_component(' LocationSpecifier_II ')' 'print(' Identifiers ') ' 'print(' String ')' 'exit(' IdorNumorMaC ')' 'add(' IdorNumorMaC ',' IdorNumorMaC ')' 'mul(' IdorNumorMaC ',' IdorNumorMaC ')' 'div(' IdorNumorMaC ',' IdorNumorMaC ')' 'exp(' IdorNumorMaC ')' 'pow(' IdorNumorMaC ',' IdorNumorMaC ')' 'sin(' IdorNumorMaC ')' 'cos(' IdorNumorMaC ')'

	<pre> 'tan(' IdorNumorMaC ')' 'asin(' IdorNumorMaC ')' 'acos(' IdorNumorMaC ')' 'atan(' IdorNumorMaC ')' 'sinh(' IdorNumorMaC ')' 'cosh(' IdorNumorMaC ')' 'ln(' IdorNumorMaC ')' 'sqrt(' IdorNumorMaC ')' 'abs(' IdorNumorMaC ')' 'and(' Identifier ',' Identifier ')' 'or(' Identifier ',' Identifier ')' 'exor(' Identifier ',' Identifier ')' 'not(' Identifier ')' 'clr(' Identifier ')' 'append(' Identifier ',' IdorListorMaC ')' 'first(' IdorListorMaC ')' 'last(' IdorListorMaC ')' 'delete(' Identifier ')' 'set(' Identifier ',' Identifier ',' IdorNumorMaC ')' 'get(' Identifier ',' Identifier ')' 'new(' IdorQual ')' 'fulfills(' Identifier ',' IdorQual ')' </pre>
AssignOpr	::=Identifier ':=' Identifier Identifier ':=' Number Identifier ':=' Macrocall .
Statements	::=Statement Statements Statement .
Statement	<pre> ::=Macrocall ';' AssignOpr ';' 'if' Exp 'then' Macrocall ';' 'if' Macrocall 'then' Macrocall ';' 'if' Exp 'then{' Statements '}' 'if' Macrocall 'then{' Statements '}' 'foreach' Identifier 'in' Identifier '{' Statements '}' 'foreach' Identifier 'in' Macrocall '{' Statements '}' 'while' Exp 'do{' Statements '}' . </pre>
Modification	::='modification{' Statements '}' .
Modifications	::=Modifications Modification Modification .
PExp	::='(' Exps ')' .
PExps	::=PExp ',' PExps PExp .
Repair_Rule	::='strict' 'possible' 'welcome' .
Rule	::='repair_rule(' Repair_Rule '){' 'symptoms{' PExps '}' Modifications '}' .
Rules	::=Rule Rules Rule .
Ptype	::='parameter' 'characteristic' .
Definitions	::=Identifier '⊔' PType ';' Definitions Identifier '⊔' PType ';' .
Cons	::=Identifier ';' Cons Identifier ';' .

Gates	::='gates{' Cons '}' .
Parameters	::='parameters{' Definitions '}' .
Macro	::='macro□' Identifier '(' TIdentifiers ')' '{' Statements 'return(' Identifier ');' '}' 'macro□' Identifier '(' TIdentifiers '&' Identifier ')' '{' Statements 'return(' Identifier ');' '}' 'macro□' Identifier '(' '&' Identifier ')' '{' Statements 'return(' Identifier ');' '}' .
Macros	::=Macro Macros Macro .
Class	::='class□' Identifier '{' Gates Parameters Rules '}' .
Classes	::=Classes Class Class .
Program	::=Macros Classes .

B Anwendungsbeispiele

An dieser Stelle sollen einige Beispiele aufgeführt werden, an denen die Benutzung der Sprache verdeutlicht wird.

Diese Beispiele sind so ausgesucht, daß die wesentlichen Merkmale der Sprache zum Tragen kommen.

B.1 Beispiele zur Veränderung der Schaltplantopologie

In den Studienarbeiten von S. Uecker ([Ueck97]) und C. Krafthöfer ([Kraf97]) werden konstruktive Veränderungen zur Beeinflussung des dynamischen Verhaltens hydraulischer Antriebe vorgeschlagen. Somit sind diese Beispiele als praxisnah anzusehen. An diesen Beispielen soll nun gezeigt werden, wie mit der Designsprache diese Veränderung der Schaltplantopologie durchgeführt werden können.

B.1.1 Einfügen einer Differentialschaltung

Das wesentliche Merkmal der Differentialschaltung ist, daß der zurückfließende Ölstrom der Pumpenleitung zugeführt wird. Auf diese Weise werden dann größere Ausfahrgeschwindigkeiten des Kolbens erreicht.

Das Beispiel der Abbildung 38:

1. Schritt:

- (a) `insert_component(in_series Z V, (TRI1))`
- (b) `insert_component(in_series V P, (TRI2))`
- (c) `insert_component(in_series TRI1 TRI2, (RV1))`

2. Schritt: `insert_component(in_series T Z,(RV2))`

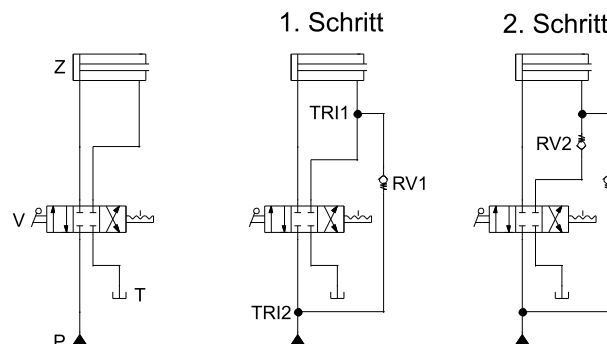


Abbildung 38: Einfügen einer Differentialschaltung

Bemerkung: An diesem Beispiel ist zu sehen, daß es für Schritt 1 zwei und für Schritt 2 drei Möglichkeiten gibt. Bei der Verarbeitung werden alle Möglichkeiten berücksichtigt, so daß die richtige auf jeden Fall gefunden wird. Außerdem achte man auf die Durchlaßrichtung von $RV1$ und 2, die durch die Reihenfolge der Tri-Connections $TRI1$ $TRI2$ und der Richtung des Pfades (T, V, Z) gegeben sind.

B.1.2 Einfügen einer Vorlaufdrosselung

Eine Vorlaufdrosselung kann leicht durch Einfügen eines Drosselventils in den Vorlauf eines Zylinders realisiert werden.

Das Beispiel der Abbildung 39:

```
insert_component(in_series P Z, (DV))
```

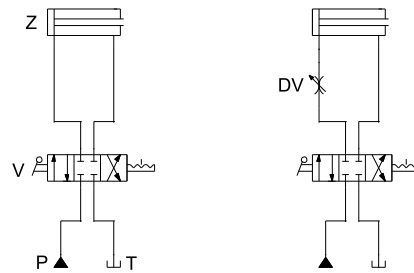


Abbildung 39: Einfügen einer Vorlaufdrossel

Bemerkung: Beim Einfügen gibt es drei Möglichkeiten, die alle berücksichtigt werden. Eine *Rücklaufdrosselung* läßt sich analog verwirklichen.

B.1.3 Einfügen einer Nebenstromdrosselung

Die Nebenstromdrosselung kann durch einfügen eines neuen Zweiges, der ein Drosselventil und einen Tank enthält realisiert werden.

Das Beispiel der Abbildung 40:

1. `insert_component(in_series Z P, (TRI))`
2. `insert_component(in_series TRI T2, (DV, T2))`

Bemerkung: Beim Einfügen gibt es drei Möglichkeiten, die alle berücksichtigt werden.

B.1.4 Einfügen einer Bypaßdrossel

Einfügen eines definierten Leckvolumenstromes zwischen den Zylinderkammern eines Zylinders.

Das Beispiel der Abbildung 41:

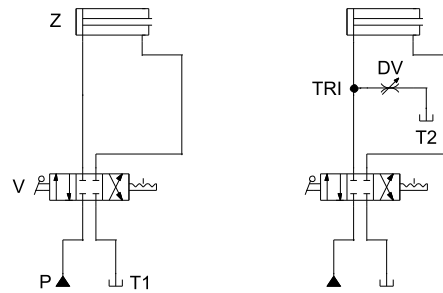


Abbildung 40: Einfügen einer Nebenstromdrosselung

1. `insert_component(in_series Z P, (TRI1))`
2. `insert_component(in_series Z T, (TRI2))`
3. `insert_component(in_series TRI1 TRI2, (DV))`

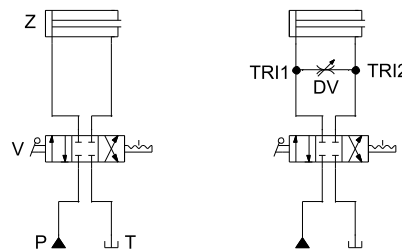


Abbildung 41: Einfügen einer Bypassdrossel

Bemerkung: Hier werden beide Richtungen von DV berücksichtigt.

B.1.5 Einfügen eines Hydrospeichers

Der Hydrospeicher wird durch Einfügen eines Zweiges in die Pumpenleitung realisiert. Das Beispiel der Abbildung 42:

1. `insert_component(in_series P V, (TRI))`
2. `insert_component(in_series TRI HS, (HS))`

Bemerkung: In diesem Beispiel ist die Position eindeutig, an der die Komponente eingefügt werden muß.

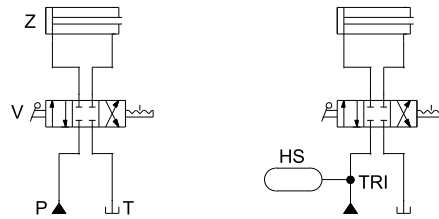


Abbildung 42: Einfügen eines Hydrospeichers

B.2 Beispiel einer Wissensklasse zur Kompensation von Fehlverhalten

Im folgenden wird die Entwicklung einer Wissensklasse beschrieben. An diesem Beispiel ist die prinzipielle Vorgehensweise bei der Entwicklung von Regeln zu erkennen.

Dieses Beispiel ist lauffähig und kompensiert das in der Wissensklasse formulierte Fehlverhalten in einer Hydraulikschaltung.

B.2.1 Kraftentfaltung am Zylinder reicht nicht aus !

Kriterium zur Erkennung der Fehlfunktion:

I. Zylinder bewegt sich nicht; er fährt nicht aus.

II. Es gilt: $p_A \leq p_0$ (Kammerdruck $p_A \leq$ Betriebs-/Versorgungsdruck p_0).

Mögliche Maßnahmen zur Kompensation des Fehlverhaltens:

1. Anhebung des Druckpotentials auf der Zulaufseite
 - Verfolgung der Verbindung zur Druckversorgung
 - (a) Absenkung durchflossener hydraulischer Widerstände
 - (b) Anhebung des Einstelldruckes parallel geschalteter Druckbegrenzungsventile
 - (c) Anhebung des Versorgungs-(Pumpen-)Druckes
2. Absenkung des Druckpotentials auf der Ablaufseite
 - Verfolgung der Verbindung zum Tank
 - (a) Absenkung durchflossener hydraulischer Widerstände
 - (b) Absenkung des Tankdruckes p_T (falls $p_T > 0$ bar)
3. Vergrößerung der wirksamen Kolbenfläche(n)
 - (a) Vergrößerung des Kolbendurchmessers d_k

(b) Verkleinerung des Kolbenstangendurchmessers d_s

Anhand dieser verbalen Beschreibung des Fehlverhaltens und den möglichen Kompensationsmaßnahmen kann nun eine Formulierung in der Designsprache stattfinden.

In der Zylinderklasse werden Makros benutzt, die am Ende dieses Abschnittes aufgeführt und beschrieben werden. Die Namen und die Parameterliste dieser Makros erklären die Funktion, so daß eine Erklärung im Quellcode entfällt.

Bevor die Regeln einer Wissensklasse aufgeführt werden, müssen die Gates und Parametervariablen der Komponenten der Klasse aufgeführt werden. Für die Zylinderklasse sieht der Klassenkopf wie folgt aus:

```
class cylinder{
  gates{
    A;
    B;
  }
  parameters{
    A_K characteristic ; // Kolbenfläche
    A_R characteristic ; // Ringfläche
    DZ characteristic ; // Gleitreibung
    HUB characteristic ; // Kolbenhub
    S parameter ; // Kolbenposition
    F parameter ; // Auf den Zylinder wirkende Kraft
    V parameter ; // Ausfahrgeschwindigkeit
  }
}
```

Nach dem Klassenkopf können die Regel der Zylinderklasse aufgeführt werden, die die Fehlverhalten, mit den dazugehörigen alternativen Kompensationsvorschlägen, beschreiben.

```
repair_rule(strict){
  symptoms{(V = 0),(Kammerdruck(TRUE)<=Betriebsdruck(TRUE))}

  modification{ // Möglichkeit 1.(a)
    // Absenkung durchflossener hydraulischer Widerstände
    foreach e in get_pump_suppliers(this){
      mul_resistance(e,this,1.1); // Erhöhe Durchfluß um 10%
    }
  }

  modification{ // Möglichkeit 1.(b)
    // Anhebung des Einstelldruckes parallel geschalteter Druckbegrenzungsventile
```

```

foreach p in get_pump_suppliers(this){
    tri:=select_component(in_series this p,(comp_type=Tri_Connection));
    foreach t in tri{
        v:=select_component(in_series t (comp_type=tank),
            (comp_type=pressure_relief_valve));
        foreach e in v{
            set(e,P_SOLL,add(get(e,P_SOLL),-2)); // Druck um 2 bar verringern
        }
    }
}

```

```

modification{ // Möglichkeit 1.(c)
// Anhebung des Versorgungs-(Pumpen-)Drucks
    foreach e in get_pump_suppliers(this){
        set(e,P_LIM,add(get(e,P_LIM),2)); // Druck um 2 bar erhöhen
    }
}

```

```

modification{ // Möglichkeit 2.(a)
// Absenkung durchflossener hydraulischer Widerstände
    foreach e in get_tank_suppliers(this){
        mul_resistance(e,this,1.1); // verringere Widerstand um 10%
    }
}

```

```

modification{ // Möglichkeit 2.(b)
// Absenkung des Tankdrucks
    foreach e in get_tank_suppliers(this){
        if get(e,PRESSURE)> 0 then{ // nur falls Tankdruck > 0
            set(e,PRESSURE,add(get(e,PRESSURE),-2)); // um 2 bar verringern
        }
    }
}

```

```

modification{ // Möglichkeit 3.(a)
// Vergrößerung des Kolbendurchmessers um 10%
    mul_Kolbendurchmesser(this,1.1);
}

```

```

modification{ // Möglichkeit 3.(b)

```

```

    // Verkleinerung des Kolbenstangendurchmessers um 10%
    mul_Kolbenstangendurchmesser(this,0.9);
  }
} // Ende der Regel

```

Die selbstdefinierten Makros

Hier werden nun die Makros aufgeführt, die in der Zylinderklasse benutzt werden. Diese Makros müssen in der Wissensbasis vor der ersten Wissensklasse definiert werden.

```

macro Kammerdruck(boolean b){
// Definiert den Kammerdruck eines Zylinders.
// In neueren Versionen der Designsprache werden die Anforderungen von Außen vorgegeben.
  h:=35.0;
  return(h);
}

```

```

macro Betriebsdruck(boolean b){
// Holt den Betriebsdruck der Pumpe, die den höchsten Druck liefert.
  philf:=0;
  foreach e in select_component((type=pump_unit_simplified)){ // alle Pumpen
    p:=get(e,P_LIM);
    if p>philf then{ // falls Pumpendruck der höchste ist
      philf:=p;
    }
  }
  return(philf);
}

```

```

macro get_resistors(component f,component l){
// Holt alle Komponenten, die einen einstellbaren hydraulischen Widerstand haben.
  h:=select_component(in_series f l,(LEVEL>=0));
  return(h);
}

```

```

macro get_pump_suppliers(component c){
// Holt die Pumpen, die die Komponente c versorgen.
  erg:=NIL;
  foreach e in select_component((P_LIM<>0)){ // alle Pumpen
    h:=0;
    foreach el in first(select_component(in_series e c)){
      // zähle Elemente im Ergebnis
      h:=add(h,1);
    }
  }
  return(h);
}

```



```

    }
    if  $h > 0$  then{ // falls Liste nicht leer
        append(erg,e); // Pumpe in Ergebnisliste
    }
}
return(erg);
}

```

```

macro mul_resistance(component f,component l,float faktor){
// Multipliziert den Widerstand der Komponenten auf dem Pfad
// von f nach l mit dem Faktor faktor.
foreach e in get_resistors(f,l){ // alle hydraulischen Widerstände
    if get(e,LEVEL)<100 then{ // nur wenn nicht 100%
        set(e,LEVEL,mul(get(e,LEVEL),faktor));
    }
}
return(TRUE);
}

```

```

macro get_tank_suppliers(component c){
// Holt die Tanks, die mit Komponente c verbunden sind
erg:=NIL;
foreach e in select_component((PRESSURE<>0)){ // alle Tanks
    h:=0;
    foreach el in first(select_component(in_series e c)){
        // zähle Elemente im Ergebnis
        h:=add(h,1);
    }
    if  $h > 0$  then{ // falls Liste nicht leer
        append(erg,e); // Tank in Ergebnisliste
    }
}
return(erg);
}

```

```

macro mul_Kolbendurchmesser(component k,float faktor){
// Multipliziert den Kolbendurchmesser des Zylinders k mit faktor.
// Dabei werden alle abhängigen Parameter konsistent mitverändert.
ak:=get(k,A_K);
ar:=get(k,A_R);
ast:=add(ak,mul(ar,-1));

```

```

    ak:=mul(ak,faktor);
    ar:=add(ak,mul(ast,-1));
    set(k,A_K,ak);
    set(k,A_R,ar);
    return(TRUE);
}

macro mul_Kolbenstangendurchmesser(component k,float faktor){
// Multipliziert den Komlbenstangendurchmesser des Zylinders k mit faktor.
// Dabei werden alle abhängigen Parameter konsistent mitverändert.
    ak:=get(k,A_K);
    ar:=get(k,A_R);
    ar:=mul(ar,add(2,mul(faktor,-1)));
    if ar<ak then{
        set(k,A_R,ar);
    }
    return(TRUE);
}

```