**Universität – Gesamthochschule Paderborn**
Fachbereich 17 – Mathematik/Informatik

Diplomarbeit

# Eine Entwicklungsumgebung für Designaufgaben

Achim Wullenkord
Peitzweg 10
33415 Verl

Prüfer: Prof. Dr. H. Kleine Büning
Betreuer: Dr. T. Lettmann

## Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte Hilfe Dritter angefertigt habe. Alle Stellen, die inhaltlich oder wörtlich aus Veröffentlichungen stammen, sind kenntlich gemacht. Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde vor und wurde bisher noch nicht veröffentlicht.

Paderborn, den 07. Juli 2004

_____

(Achim Wullenkord)

# 1 Einleitung

Was ist ein Designproblem und wie wird es gelöst?
Diese Frage steht am Anfang dieser Diplomarbeit. Wird die Frage aus der Sicht der Informatik betrachtet, so ist das Design von technischen Systemen und Produkten ein wichtiges Anwendungsgebiet von wissensbasierten Systemen. Aus der Sicht des Marketings sind Kundenwünsche essentiell, um ein Produkt überhaupt verkaufen zu können. Ein System muss aus Kostengründen meist speziell auf die Kundenwünsche zugeschnitten werden.
Welche Bedeutung steckt überhaupt hinter dem Wort „Design“? Unter Design kann die Erschaffung und Erzeugung einer bestimmten Sache verstanden werden. Design ist also das bewusste Gestalten einer Sache.
In dieser Diplomarbeit gilt allerdings die Einschränkung, dass nur das Design von technischen Systemen betrachtet werden soll. Beim Design von technischen Anlagen sind bereits Vorgaben gegeben, die erfüllt werden müssen. Soll in einer Fabrikhalle zum Beispiel ein bestimmtes Produkt gefertigt werden, dann heißt das, dass die Halle nur eine bestimmte Größe hat, nur bestimmte Maschinen zur Auswahl stehen, eine bestimmte Menge je Stunde gefertigt werden muss, usw.
Eine weitere Einschränkung ist, dass Funktionen nur durch Objekte angeboten werden können. Eine Kombination von Objekten erzeugt keine neuen Funktionen, die nicht vorher einem Objekt zugeordnet wurden. Beispielsweise haben Mauersteine bestimmte Eigenschaften und Funktionen; wenn diese Mauersteine allerdings in einer bestimmten Art und Weise aufeinander gemauert werden, dann wird die Wand stabiler. Diese Funktion kann aber keinem Mauerstein direkt zugeordnet werden. Alle technischen Systeme, die hier mit Hilfe von Design erzeugt werden sollen, bestehen aus vorgegebenen Objekten und ihre Gesamtfunktion ergibt sich aus den Funktionen der Objekte.
Für ein technisches System muss eine Struktur gefunden werden, die mit Hilfe von vorgegebenen Objekten realisiert werden kann. Ein Satz, der im ersten Moment recht einfach klingt, aber bei genauerer Betrachtung doch zu einigen Problemen führt. Es gibt sehr verschiedene Produkte, für die ein Design gefunden werden muss. Ein Schiff zum Beispiel hat andere Eigenschaften/Anforderungen als ein Flugzeug. Aus der Sicht des Marketings ist die Aufgabe einfach. Das Produkt muss die Kundenwünsche in dem Maße erfüllen, dass der Kunde das Produkt kauft. Aus der Sicht der Informatik muss sowohl für das Schiff, als auch für das Flugzeug ein Programm geschrieben werden, das die Design-Aufgabe löst. Nun sollen aber nicht nur Flugzeuge und Schiffe geplant werden, sondern auch Autos, Stühle, Rasenmäher, Mixer, usw. Für jedes dieser Produkte müsste ein neues Design-Programm geschrieben werden. Ist es nicht möglich eine Entwicklungsumgebung zur Verfügung zu stellen, die die Lösung aller dieser Aufgaben unterstützen kann? Genau das ist das Ziel dieser Diplomarbeit. Aber wie soll eine Entwicklungsumgebung so unterschiedliche Design-Aufgaben unterstützen? Ein Programm, das das Design eines Flugzeuges erstellt, sieht schließlich anders aus, als eines für einen Tisch.

Die meisten Programme, die Design-Aufgaben lösen, benutzen Tools, die nur für diese Aufgabe erstellt wurden. Auch die Art und Weise wie eine Lösung erstellt wird, ist dabei für jede Design-Aufgabe fest. Ein Programm, das eine bestimmte Design-Aufgabe löst, ist wie eine fest verdrahtete Schaltung, die nur eine Aufgabe hat und auch nur diese eine Aufgabe lösen kann. Die Idee hinter dieser Diplomarbeit ist es, autonome Werkzeuge/ Tools zur Verfügung zu stellen. Mit diesen Werkzeugen kann dann eine Design-Aufgabe gelöst werden. Damit eine Lösung entstehen kann, müssen die Werkzeuge in einer bestimmten Reihenfolge, die für jede Design-Aufgabe individuell ist, benutzt werden. Diese Aufgabe wird von der so genannten Strategie übernommen. Sie bestimmt welche Werkzeuge benutzt werden und wie die Ergebnisse der Werkzeuge weiter verarbeitet werden. Die Strategie wird dabei vom Benutzer erstellt und stellt die Flexibilität der Entwicklungsumgebung sicher. Die Werkzeuge können für jede Design-Aufgabe erneut benutzt

werden. Der Unterschied zwischen dem Design eines Flugzeuges und eines Tisches liegt in der Strategie, die unterschiedliche Werkzeuge benutzt um die Aufgabe zu lösen.

Es ist sogar möglich, dass Strategien miteinander arbeiten und Daten austauschen.

Damit ein Datenaustausch zwischen den Strategien stattfinden kann, muss als erstes ein Datenspeicher zur Verfügung gestellt werden, der alle möglichen Datentypen verwalten kann und dafür sorgt, dass es zu keinen inkonsistenten Daten kommt. Diese Aufgabe wird vom Blackboard übernommen. Auf dem Blackboard können die Strategien Daten ablegen und neue Daten anfordern.

Jedes System, das konfiguriert werden soll, besitzt eine Struktur. Ein Auto besteht aus einer Karosserie, vier Rädern und einem Motor. Ein Tisch sieht natürlich ganz anders aus, hat aber ebenfalls eine Struktur: Platte, Rahmen, Beine. Die mögliche Struktur der Systeme muss abgebildet werden, damit ein gültiges Design entstehen kann. Diese Aufgabe kann von einer Design-Graphgrammatik übernommen werden. Beim Design eines technischen Systems werden auch weitere Module benutzt. Mit Hilfe eines Moduls, das Bilanzverarbeitung anbietet, kann effizient eine Auswahl von Komponenten bestimmt werden, die bestimmte Funktionen anbieten und nur vorhandene Ressourcen verwenden. Eine Simulation kann die noch vorhandenen Fehler im Design auffinden. Die Strategie bildet das Bindeglied zwischen den Modulen, sie steuert ihren Aufruf und die Verwendung der Zwischenergebnisse.

Nachfolgend wird ein Überblick über den Aufbau der Arbeit gegeben.
Im zweiten Kapitel wird die Aufgabe von Design/Konfigurationssystemen erläutert. Dabei soll die Frage geklärt werden, warum es überhaupt Design/Konfigurationsaufgaben gibt. Ein weiteres Thema sind mögliche Modelle um Komponenten, die benutzt werden, darzustellen.
Im dritten Kapitel wird ein Konfigurationssystem, das bereits entwickelt wurde, dargestellt. Das Konfigurationssystem heißt „Plakon“ und dient dazu, die Arbeitsweise anderer Entwicklungsumgebungen darzustellen.
Das vierte Kapitel beschreibt die Arbeitsweise der neuen Entwicklungsumgebung, die im Rahmen der Diplomarbeit entwickelt werden soll.
Im fünften Kapitel wird die Structural Database beschrieben, die ein Teil der Entwicklungsumgebung ist, die hier entworfen und implementiert werden soll.
Das sechste Kapitel beschreibt die Strategy Database, ihre Aufgabe ist es, Strategien zu verwalten und diese dem Benutzer zur Verfügung zu stellen.
Im siebten Kapitel wird das Blackboard beschrieben. Das Blackboard dient in der Entwicklungsumgebung als zentraler Datenumschlagplatz.
Das achte Kapitel beschreibt die Funktionsweise der Bilanzverarbeitung. Die Bilanzverarbeitung dient später als ein Tool und wird in einer Strategie benutzt um eine Design-Aufgabe zu lösen.

Bisher wurden alle Punkte nur sehr theoretisch betrachtet, in Kapitel 9 der Diplomarbeit wird dann ein konkretes Problem vorgestellt und mit Hilfe der Entwicklungsumgebung wird ein Design erstellt. Die Aufgabe besteht darin, eine leere Fabrikhalle mit Maschinen zu füllen, so dass ein Werkstück gefertigt werden kann. Am Anfang der Design-Aufgabe ist eine leere Werkhalle gegeben und ein Katalog mit Maschinen unterschiedlichen Typs. In einer technischen Zeichnung sind Werkstücke abgebildet, diese sollen am Ende gefertigt werden.
Die Aufgabe der Strategie ist es nun, Maschinen auszusuchen und diese so aufzustellen, dass die Werkstücke gefertigt werden können. Dabei sind Randbedingungen zu berücksichtigen wie etwa Maschinenkosten, benötigtes Personal, Energiekosten, Transportkosten. Die Strategie erzeugt also eine Lösung, indem sie aus der technischen Zeichnung eine Reihenfolge von Bearbeitungsschritten extrahiert, den Bearbeitungsschritten Maschinen mit passendem Energiebedarf zuordnet, diese

Maschinen in der leeren Werkhalle so platziert, dass geringe Transportkosten für die Werkstücke zwischen den Maschinen entstehen, die Maschinen mit dem vorhandenen Personal bedient werden können und die gewünschte Stückzahl von Werkstücken pro Stunde gefertigt werden kann.

Natürlich war es nicht das Ziel dieser Diplomarbeit, diese Aufgabe zu lösen; zur Lösung dieser Aufgabe werden aber viele der Möglichkeiten verwendet, die die Entwicklungsumgebung für Design-Aufgaben bietet.

In Kapitel zehn wird dann eine Zusammenfassung der vorgestellten Arbeit gegeben und ein Ausblick auf mögliche Verbesserungen. Der Anhang A dient nur dazu die benötigten Dateien vorzustellen mit der die Entwicklungsumgebung arbeitet

# 2 Design/Konfigurationssysteme

In diesem Kapitel soll kurz geschildert werden, was die Aufgabe eines Design/Konfigurationssystems ist und wie diese Aufgabe technisch umgesetzt werden kann. Danach werden mögliche Modelle vorgestellt, die Komponenten abbilden können. Diese Komponenten können dann mit Konfigurationsmethoden verarbeitet werden.

## 2.1 Aufgabe von Produktkonfigurationssystemen/Konfigurationssystemen

Warum gibt es Produktkonfigurationssysteme?

Die meisten produzierten Produkte dienen dem Zweck auf einem Markt angeboten zu werden, um dann an einen Käufer weitergegeben zu werden. Die Frage die sich dann stellt ist: „Warum wählt der Käufer gerade mein Produkt aus?“.

In der Marketingforschung stellte sich heraus, dass sich das Verhalten der Anbieter, die auf einem Markt agieren, im Laufe der Jahre verändert hat. Diese Veränderung im Verhalten der Anbieter wurde nötig, da sich der Markt auf dem sie interagieren ständigen Veränderungen unterworfen ist.

Die Marketingforschung unterteilt die Verhaltensmuster der Anbieter in fünf Stufen, wobei die erste Stufe „weit“ in der Vergangenheit liegt und die fünfte Stufe die heutige Zeit widerspiegelt.

Die erste Stufe wurde „Produktionsorientierung“ genannt, die Nachfrage nach Produkten war riesig, doch leider reichten die Produktionskapazitäten nicht aus um die Nachfrage zu befriedigen. Ein Beispiel hierfür wäre Deutschland nach dem zweiten Weltkrieg.

Die zweite Stufe wurde „Produktorientierung“ genannt, die Idee hierbei war, dass gute und bezahlbare Produkte hergestellt wurden. Der Markt war dabei noch nicht gesättigt, allerdings gab immer mehr Anbieter die ebenfalls ihre Produkte verkaufen wollten.
Ein Beispiel wäre Deutschland in den Jahren nach 1948.

Die dritte Stufe nannte man „Verkaufsorientierung“, zu diesem Zeitpunkt waren viele Produkte auf dem Markt, es gab somit keinen Mangel an Produkten mehr. Die Anbieter fingen an die Preise zu senken und mit „guten Verkaufsmannschaften“ (Stichwort: Hard Selling) die Produkte zu verkaufen.
Natürlich stellte sich nach einer Zeit das Symptom des Überflusses ein und die Produkte konnten nicht mehr abgesetzt werden. Der Engpass war nicht mehr die Maschinenkapazität, sondern der Markt.
Die Käufer waren nicht mehr bereit alle angebotenen Produkte des Marktes zu konsumieren, dies führte zur vierten Verhaltensänderung all derer, die auf dem Markt agierten um Produkte zu verkaufen.

Die vierte Stufe wurde „Kundenorientierung“ genannt.
Hierbei tritt zum ersten Mal der Kunde in den Vordergrund, seine Wünsche werden berücksichtigt. Es gibt nicht mehr ein Standartprodukt X, das jeder Kunde kaufen muss, sondern die Produkte werden auf den Kunden zugeschnitten. Es wird speziell für ihn ein Design erstellt.
Die Konsequenz ist, dass jedes Produkt speziell für einen Kunden erzeugt werden muss oder das es zumindest mehrere Varianten gibt, aus denen der Kunde auswählen kann.

Der Gedanke der Kundenorientierung ist die Grundlage dieser Diplomarbeit. Es gibt ein Produkt das erzeugt werden soll, aber der Kunde kann die Eigenschaften des Produktes bestimmen. Dies ist nötig, damit das Produkt an den Kunden verkauft werden kann

Die letzte Stufe sei nur der Form halber erwähnt, man nennt sie „Marktorientierung". Sie ist eine Verfeinerung der Kundenorientierung, es wird bei der Marktorientierung nicht mehr versucht alle Wünsche des Kunden zu erfüllen, sondern die Wünsche des Kunden „etwas mehr zu befriedigen" als es ein Produkt der Konkurrenz tun würde.

Eine Entwicklungsumgebung für Design-Aufgaben dient also dafür, die Wünsche eines Kunden in ein fertiges Produkt abzubilden, dabei entstehen Produktvarianten und komplexe technische Abhängigkeiten.

## 2.2 Wie arbeitet ein Design/Konfigurationssystem

Nachdem eben die Aufgabe eines Design/Konfigurationssystems aus der Sicht des Marketings beschrieben wurde, soll nun die Frage geklärt werden wie ein Design/Konfigurationssystem technisch realisiert werden kann.
Aus der Sicht der Informatik wird ein technisches System aus vielen Einzelkomponenten, unter den Bedingungen die ein Kunde/Käufer ihm vorgibt, zusammengesetzt.
Der Vorgang des Konfigurierens findet hierbei schrittweise statt, das technische System durchläuft hierbei unterschiedliche Zustände ($Q_n$, $n \in N$).
Das technische System startet im Zustand $Q_0$ und durchläuft eine Reihe von Folgezuständen $Q_i$ bis es den Endzustand $Q_e$ erreicht hat, und somit fertig konfiguriert ist.

$$Q_0 \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \ldots\ldots\ldots \longrightarrow Q_n \longrightarrow \ldots\ldots\ldots \longrightarrow Q_e$$

**Abbildung 2.1: Produktkonfiguration, eine Abfolge von Zuständen und Zustandübergängen**

Jeder Zustand wird einzeln bearbeitet, erst dann wird in den nächsten Zustand übergegangen. Dabei wird darauf geachtet, dass die Bedürfnisse/Anforderungen des Kunden/Käufers berücksichtigt werden.

Ein Design/Konfigurationssystem transformiert „Bedürfnisse und Anforderungen" eines Kunden/Auftraggebers in eine gültige Endkonfiguration $Q_e$ eines technischen Systems.

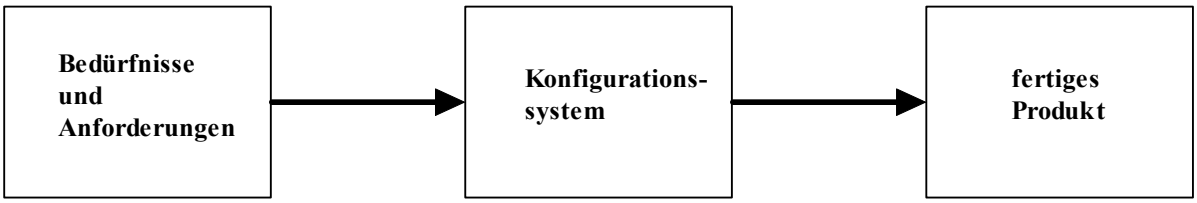



**Abbildung 2.2: Vom Bedürfnis zum Produkt**

Damit das Design/Konfigurationssystem diese Aufgabe übernehmen kann muss Domänenwissen (Wissen über den Anwendungsbereich) vorhanden sein. Unter anderem muss bekannt sein wie die Aufgabe gelöst werden kann, die dem Konfigurationssystem gestellt wurde.

## 2.3 Wissensbasierte Systeme

Eine Technik, die es ermöglicht Domänenwissen und „Wissen über die Lösung“ eines Problems darzustellen, wird „Wissensbasiertes System“ oder „Expertensystem für Konfigurationen“ genannt. Ein wissensbasiertes System ist die Kombination von [12]:

- Domänen unabhängige Interferenz Maschine
- Domänen spezifisches Wissen
- Problemspezifische Datenbank

Eine große Stärke der wissensbasierten Systeme ist hierbei die strikte Trennung von Domänenwissen und der „Art und Weise“ wie dieses Wissen verarbeitet und genutzt wird. Das Domänenwissen spiegelt das Wissen über den Anwendungsbereich wieder, dieses Wissen liegt aber meist nur in verbaler Form vor und muss erst umgewandelt werden damit Standartwerkzeuge dieses Wissen verarbeiten können.
Es existiert eine Lücke zwischen der Terminologie des Experten, der das Domänenwissen erstellt, und den Mechanismen, die mit diesem Domänenwissen arbeiten müssen. Dieses Problem wird auch als „knowledge engineering gap“ bezeichnet [2,12].
Es existieren mehrere Methodologien um diese Lücke zu schließen, ein paar Beispiele wären „Niam“ und „Kads“.

## 2.4 Modelle in Design/Konfigurationssystemen

Damit reale Systeme mit Computern abgebildet werden können, müssen Modelle benutzt werden. Ein Model ist ein bewusst konstruiertes Abbild der Wirklichkeit, das auf der Grundlage einer Struktur-, Funktions- oder Verhaltensanalogie zu einem entsprechenden Original eingesetzt bzw. genutzt wird, um eine bestimmte Aufgabe zu lösen, deren Durchführung mittels direkter Operationen am Original nicht oder zunächst nicht möglich oder zweckmäßig ist [3].

In der Arbeit von Stein [12] wird als Modelltyp für Konfigurationssysteme das so genannte „Komponenten-Modell“ vorgeschlagen.

## 2.5 Komponenten-Modell

Das Komponenten-Modell soll Wissen über die einzelnen Komponenten abbilden, die benötigt werden um das Konfigurationsproblem zu lösen. Dabei ist darauf zu achten, dass die Komponenten unter dem Aspekt ihrer Aufgabe im System beschrieben werden. Es ist zum Beispiel nicht sehr sinnvoll den atomaren Aufbau eines Stoffes zu beschreiben, wenn lediglich sein Gewicht von Interesse ist. Das Komponenten-Modell teilt sich in die Strukturbasierte Beschreibung und die Funktionsbasierte Beschreibung auf.

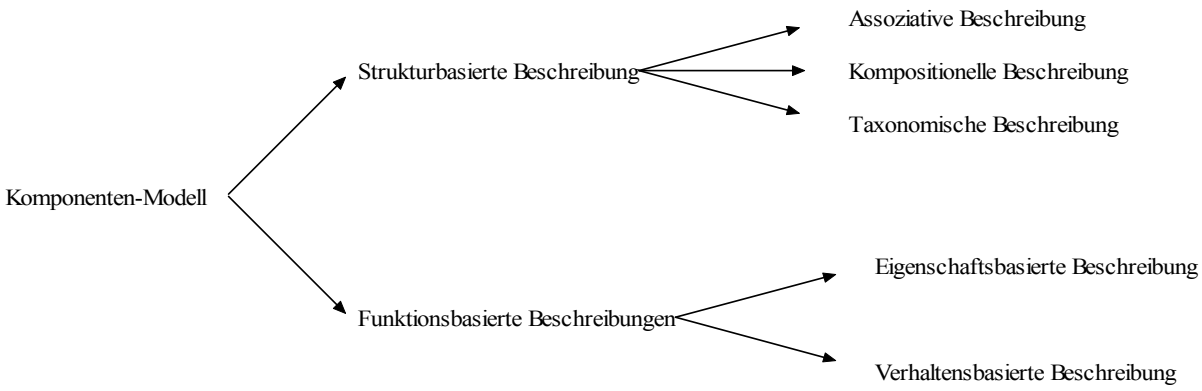



**Abbildung 2.3: Komponenten-Modell**

## 2.5.1 Strukturbasierte Beschreibungen

Die erste Möglichkeit ein Modell zu beschreiben basiert auf Strukturinformationen. Dabei werden strukturelle Beziehungen zwischen den einzelnen Komponenten in den Vordergrund gestellt. Es gibt drei Möglichkeiten die strukturellen Zusammenhänge zu beschreiben.

Die Assoziative Beschreibung ist die erste Möglichkeit. Hierbei wird die Beziehung zwischen den Komponenten durch Aussagenlogik dargestellt.
Karoserie $\wedge$ Motor $\rightarrow$ Reifen $\wedge$ Felgen
Wenn eine Konfiguration die Komponenten „Karoserie“ und „Motor“ enthält, dann müssen auch die Komponenten „Reifen“ und „Felgen“ vorhanden sein.

Die „Kommpositionelle Beschreibung“ zeigt auf, aus was eine Komponente besteht, und wie sie sich somit ersetzen lässt.
Auto $\rightarrow$ Karosserie und Motor
Die Komponente „Auto“ besteht aus Karosserie und Motor, nur wenn Karosserie und Motor vorhanden sind ist auch das „Auto“ vorhanden.

Die dritte und letzte Beschreibung wird „Taxonomische Beschreibung“ genannt. Die Idee ist, eine Komponente durch andere Komponenten zu ersetzen.

Motor $\rightarrow$ „Motor mit 100 PS Benzin“ oder „Motor 100 PS Diesel“
Die Komponente Motor kann entweder durch „Motor mit 100 PS Benzin“ oder „Motor 100 PS Diesel“ ersetzt werden, es finde eine Spezialisierung statt.

## 2.5.2 Funktionsbasierte Beschreibungen

Die zweite Beschreibungsmöglichkeit eines Models basiert auf Funktionsinformationen der einzelnen Komponenten, dabei wird das Verhalten/Eigenschaften der einzelnen Komponenten beschrieben.

Der erste Modelltyp wird „Eigenschaftsbasierte Beschreibung“ genannt. Die Komponenten des Modells werden durch „Eigenschaft/Wert“ Paare beschrieben

Der zweite Modelltyp wird „Verhaltensbasierte Beschreibung“ genannt. Jede Komponente bekommt eine Menge von Eigenschaften, genau wie bei der „Eigenschaftsbasierten Beschreibung“, zusätzlich werden „Gates“ und „Constraints“ hinzugefügt.
Die Aufgabe der „Gates“ ist der Austausch von Informationen zwischen den einzelnen Komponenten.
Die Constraints sind eine Menge von Relationen. Eine Relation ist dabei auf den Eigenschaften der Komponente und deren Schnittselle definiert. Die Constraints bestimmen somit das Verhalten einer Komponente.

## 2.6 Konfigurationsmethoden

Nachdem „das verbale Wissen“ abgebildet wurde muss nun das abgebildete Wissen verarbeitet werden. Wie erzeugt man aus all dem Wissen eine gültige Konfiguration/Design?

Es bieten sich drei Grundprinzipien an wie Wissen bearbeitet werden kann [12]:

- Zerlegung: Jedes Problem kann wieder in ein kleineres Problem zerlegt werden, dieses „Unterproblem“ kann entweder ein Objekt sein oder aber eine Funktion.

- Fallbildung: Es wird überprüft ob das Problem (Fall), das gelöst werden muss, schon einmal in einer ähnlichen Form vorlag und ob man die Lösung des alten Problems (Fall) auf das neue Problem anwenden kann.

- Transformation: Die Anforderungen an die Konfiguration werden mit Hilfe von Transformationsregeln direkt in eine Lösung umgewandelt.

Im Folgenden wird die Zerlegung genauer betrachtet da ihr ein sehr intuitives Prinzip zugrunde liegt: „Wie man ein komplexes System zusammensetzt (konfiguriert), so kann man es auch wieder zerlegen (analisieren)“. Es entstehen dabei keine neuen Abhängigkeiten.

Die eben erwähnten Modellbeschreibungen „Funktionsbasierte Beschreibung“ und „Strukturbasierte Beschreibung“ stehen nicht in Konkurrenz zueinander, sondern können gemeinsam genutzt werden.
Es gibt hierbei verschiedene Methoden, die für beide Modellbeschreibungen anwendbar sind [9,12].

Eine mögliche Methode wäre die „Bilanzverarbeitung“. Sie gehört zu den „Funktionsbasierten Modellen“ und basiert auf der „Eigenschaftsbasierten Beschreibung“.
Die Grundidee ist, dass jede Komponente Ressourcen anbietet und fordert. Die Ressourcen werden in einer Bilanz verwaltet, die jede Komponente besitzt, erst wenn alle Bilanzen positiv sind und somit alle Bedürfnisse befriedigt wurden, ist eine gültige Konfiguration gefunden.

Die „Skelett Konfiguration“ ist eine weitere Methode. Sie gehört zu den „Strukturbasierten Modellen“. Die Grundidee ist, dass es eine Struktur gibt, die bei allen gültigen Lösungen gleich ist.

Die „Assoziative Konfiguration“ ist die letzte hier vorgestellte Methode, sie gehört weder zu den „Strukturbasierten Modellen“ noch zu den „Funktionsbasierten Modellen“. In diesem Ansatz wird eine Tabelle mit Regeln benutzt, jede Zeile ist eine Regel und jede Spalte ist eine Bedienung die erfüllt werden muss. Die letzte Spalte ist die Aktion die ausgeführt werden kann, wenn alle Bedienungen einer Zeile erfüllt sind.

# 3 Das System Plakon

Plakon ist ein Expertensystemkern für Planungs- und Konfigurationsaufgaben in technischen Domänen [2].

Der Expertensystemkern Plakon wurde vom Bundesminister für Forschung und Technologie als Verbundprojekt gefördert, hierfür standen dem Projekt 7,3 Mio. DM zur Verfügung.

Das Ziel des Projektes war es ein System zur
„Entwicklung, Erprobung und betrieblichen Einführung einer Architektur von Expertensystemen zur Konfigurierung, Planung und Konstruktion für technische Systeme“
zu schaffen.

Die Grundidee von Plakon ist, dass eine Konstruktion aus vielen Teilobjekten besteht. Diese Teilobjekte können von der Aufgabenstellung gefordert werden.
Die Objekte können dann mit fest definierten Funktionen bearbeitet werden.

- Objekt zerlegen: Ein großes Objekt wird in kleinere Objekte zerlegt, die kleineren Objekte erfüllen dabei die Aufgabe des großen Objekts. Es wird Top-Down vorgegangen.
  Bsp.: Ein Tisch besteht aus Tischplatte und 4 Tischbeinen und Schrauben. Der Tisch wurde in viele „kleinere“ Einzelobjekte zerlegt.

- Objekt zusammenfügen (Aggregatikon): Dies stellt eine Bottom-Up Vorgehensweise dar, es wird von einem bestimmten Objekt aus vorgegangen.
  Bsp.: Man möchte einen Tisch konfigurieren, aber der Tisch soll nur drei Tischbeine haben.

- Spezialisieren: Ein allgemein gehaltenes Objekt wird einem speziellen Objekt zugeordnet.
  Bsp.: Aus dem Objekt Tisch, wird der Tisch mit dem Katalognamen „Bert“.

- Objekt parametrisieren: Damit werden Eigenschaften des Objekts festgelegt.
  Bsp.: Der Tisch ist 10 kg schwer, die Farbe des Tisches ist grün, usw.

## 3.1 Begriffshierarchie

Damit die Funktionen aus Kapitel 3 angeboten werden können ist das nötige konzeptuelle Domänenwissen deklarativ in einer Begriffshierarchie gespeichert.
Die kommpositionelle Hierarchie ist nötig für die „Zerlegung und Aggregatikon“, die taxonomische Hierarchie dient der „Eigenschaftsbeschreibung und Spezialisierung“.

In dem Buch „Das Plakon Buch“ wird der Begriff der „Begriffshierarchie“ eingeführt. In dieser Hierarchie werden alle Objekte der Anwendungsdomäne und ihren Abhängigkeiten untereinander beschrieben. Dadurch entsteht eine taxonomische (is-a) Hierarchie, die Objekte typisiert und Klassenspezialisierungen beschreibt.

## 3.2 Constraints

Plakon benötigt noch die so genannten „Constraints“. Die Aufgabe dieser Constraints ist es Abhängigkeiten, die in der Domäne zwischen Objekten vorhanden sind, zu überprüfen und darzustellen. Die „Constraints“ werden in einem „Constraint Netz“ verwaltet.

## 3.3 Ablauf eines Konstruktionsschrittes

Ein Konstruktionsschritt würde in Plakon somit wie folgt aussehen:

1. Bestimmung der aktuellen Teilkonstruktion auf syntaktisch durchführbare Konstruktionsschritte. Diese Konstruktionsschritte werden in einer Agenda zusammengefasst. Die Agenda speichert alle möglichen Aktionen, die mit der aktuellen Teilkonstruktion möglich sind.

2. Aus der eben erstellten Agenda wird ein Konstruktionsschritt ausgewählt, dies geschieht mit Hilfe von „Agenda Auswahlkriterien“. Die Agenda Auswahlkriterien können sich dabei auf ein bestimmtes Objekt beziehen oder auf eine Eigenschaft eines Objektes, oder aber die Agenda Einträge selbst werden bewertet, und nach dieser Bewertung ausgewählt

3. Der ausgewählte Konstruktionsschritt wird durchgeführt, dies geschieht mit Hilfe von Bearbeitungsverfahren. Ein Bearbeitungsverfahren ist eine Methode zur Ermittlung eines Wertes für einen Konstruktionsschritt. Dieser Wert wird dann in die aktuelle Teilkonstruktion eingefügt.

4. Optional kann an dieser Stelle das Constraint Netz propagiert werden

## 3.4 Der zentrale Zyklus in Plakon

Der vollständige zentrale Zyklus sieht in Plakon wie folgt aus [2]:

1. Mit Hilfe der Kontrollregeln wird eine Phase zur Bearbeitung bestimmt.

2. Auswahl der als nächstes zu bearbeitenden Teilkonstruktion mit Hilfe der Suchstrategien im TK-Focus.

3. Generierung der Agenda unter Berücksichtigung des Konstruktionsschritt-Fokus. Wenn die Agenda leer ist, zurück zu 1.

4. Auswahl eines Konstruktionsschrittes aus der Agenda mit Hilfe von „Agenda-Auswahlkriterien“.

5. Festlegung der anzuwendenden Bearbeitungsverfahren, diese sind entweder in der Strategie vorgeben oder speziell für einzelne Konstruktionsschritte definiert.

6. Durchführung des Konstruktionsschrittes mit Hilfe der Bearbeitungsverfahren. Weiterentwicklung der aktuellen Teilkonstruktion in der dynamischen Wissensbasis.

7. Optionale Propagierung des Constraint-Netzes.

8. Im Konfliktfall würde an dieser Stelle das Konfliktlösungswissen ausgewertet, ein „Aufsetzpunkt“ für das Backtracking berechnet und der Zyklus in Schritt 3 wieder aufgenommen werden.

9. Ist die Teilkonstruktion entsprechend der Begriffshierarchie komplett, dann ist der Konstruktionsvorgang beendet. Ist die Abbruchbedingung der Phase erfüllt, weiter bei Schritt 1, andernfalls bei Schritt 2

# 4 Ein „neues“ Design/Konfigurationssystem

Das gerade vorgestellte Konfigurationssystem  Plakon besitzt einen festen Ablaufzyklus (Kapitel 3.4). Dieser muss komplett abgearbeitet werden, damit der nächste Schritt vollzogen werden kann. Es ist nicht möglich diesen Zyklus zu unterbrechen oder Dinge zu ändern.

Das Ziel dieser Diplomarbeit ist es, eine Entwicklungsumgebung zu erstellen, die vom Benutzer gesteuert werden kann. Das System soll jederzeit unterbrechbar sein, aber auch gleichzeitig die Möglichkeit geben, Design-Aufgaben, die gestoppt werden, wieder fortzusetzen.
Ein weiteres Ziel ist, mehrere Design/Konfigurationsaufgaben gleichzeitig abarbeiten zu können, dies soll gewährleisten, dass der Benutzer mehrere Lösungen auf denselben Nutzdaten testen kann. Die Nutzdaten sollen flexibel sein und müssen von der Entwicklungsumgebung verwaltet werden. Damit eine größtmögliche Flexibilität geboten werden kann, muss es dem Benutzer möglich sein verschiedene Konfigurationsmethoden zu benutzen.

Das Ziel ist also ein Entwicklungsumgebung zu erstellen, die völlig unabhängig von der gewählten Design-Aufgabe ist.

## 4.1 Aufbau und Struktur der Entwicklungsumgebung

Die Entwicklungsumgebung ist aufgeteilt in:

- Strategy Database
- Structural Database
- Component Database
- Blackboard

Jede dieser Komponenten bildet eine eigenständige abgeschlossene Einheit.
Das Blackboard ist ein zentraler Datenumschlagplatz, der Daten verwaltet. Die Hauptaufgabe des Blackboards ist dafür zu sorgen, dass Informationen gespeichert und wieder angefordert werden können. Dabei muss das Blackboard sicherstellen, dass die Informationen, die bearbeitet werden, nicht mehrfach weitergegeben werden. Ist dies der Fall, dann muss der Empfänger der Informationen zumindest informiert werden das die Informationen mit denen er arbeitet „unsicher“ sind und sich jederzeit ändern können.

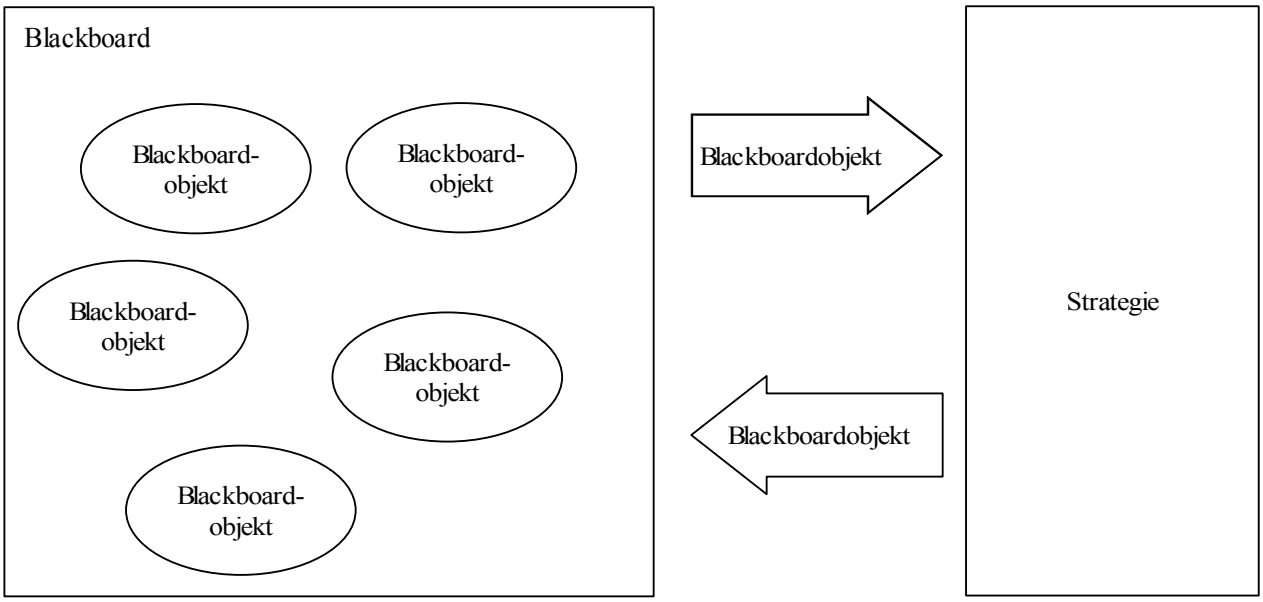



**Abbildung 4.1: Das Blackboard**

Die Strategy Database verwaltet Strategien. Eine Strategie ist eine Lösung für eine Design-Aufgabe. Diese Strategien können vom Benutzer jederzeit gestartet, gestoppt, neu erzeugt, geändert oder gelöscht werden.
Somit hat der Benutzer jederzeit die Kontrolle über die Entwicklungsumgebung. Wenn er eine Design-Aufgabe lösen möchte, kann er eine Strategie, die er vorher erstellt hat, einladen. Danach kann er die Strategie starten. Die Strategie wird dann mit dem Blackboard Daten austauschen und auf diesen Daten arbeiten. Falls die Strategie fehlerhaft ist, kann der Benutzer jederzeit eingreifen und die Strategie stoppen und ändern oder falls es nötig sein sollte die fehlerhafte Strategie löschen.

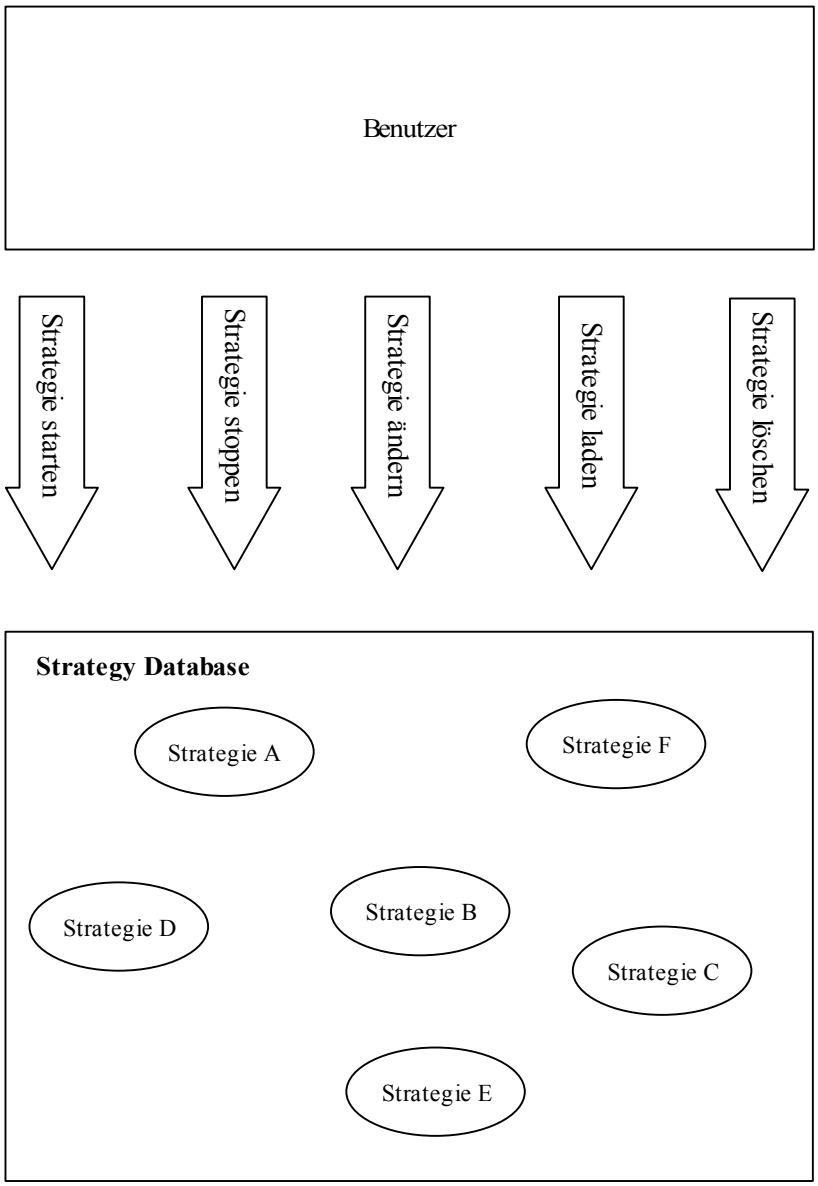



**Abbildung 4.2: Die Strategy Database wird vom Benutzer gesteuert**

Die Structural Database erlaubt es dem Benutzer die Struktur eines Konfigurationsobjektes darzustellen, damit kann der Benutzer festlegen wie das endgültige Produkt aussehen soll und wie es sich zusammensetzt. Diese Informationen werden vom Benutzer auf das Blackboard gespeichert und anschließend von den Strategien verarbeitet.

In der Component Database beschreibt der Benutzer die Komponenten, aus denen sich ein Konfigurationsobjekt zusammensetzt. Damit kann der Benutzer festlegen, wie die Komponenten zusammenwirken und sich gegenseitig beeinflussen. Auch diese Informationen werden vom Benutzer auf das Blackboard gespeichert und anschließend von den Strategien verarbeitet. Ein Beispiel für ein Verfahren, das mit solchen Komponentenmodellen arbeitet, ist die Bilanzverarbeitung [9]. Auf die Component Database wird in dieser Arbeit nicht weiter eingegangen.

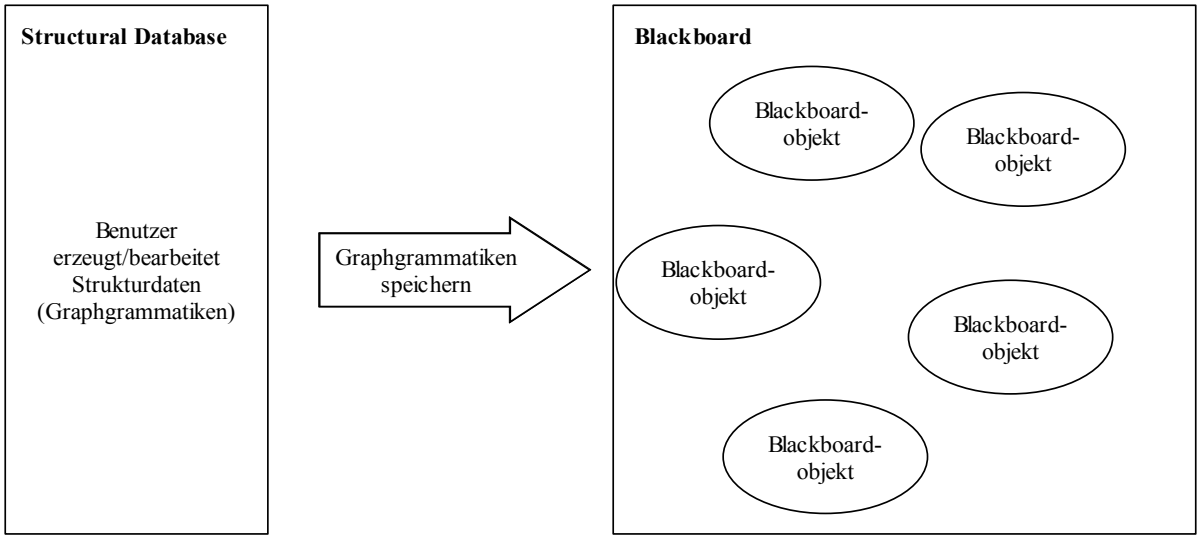



**Abbildung 4.3: Die Structural Database stellt Objekte zur Verfügung, die auf dem Blackboard gespeichert werden können**

Dies waren die Komponenten der Entwicklungsumgebung, als nächstes folgt ein Gesamtbild wie diese Komponenten zusammenarbeiten.

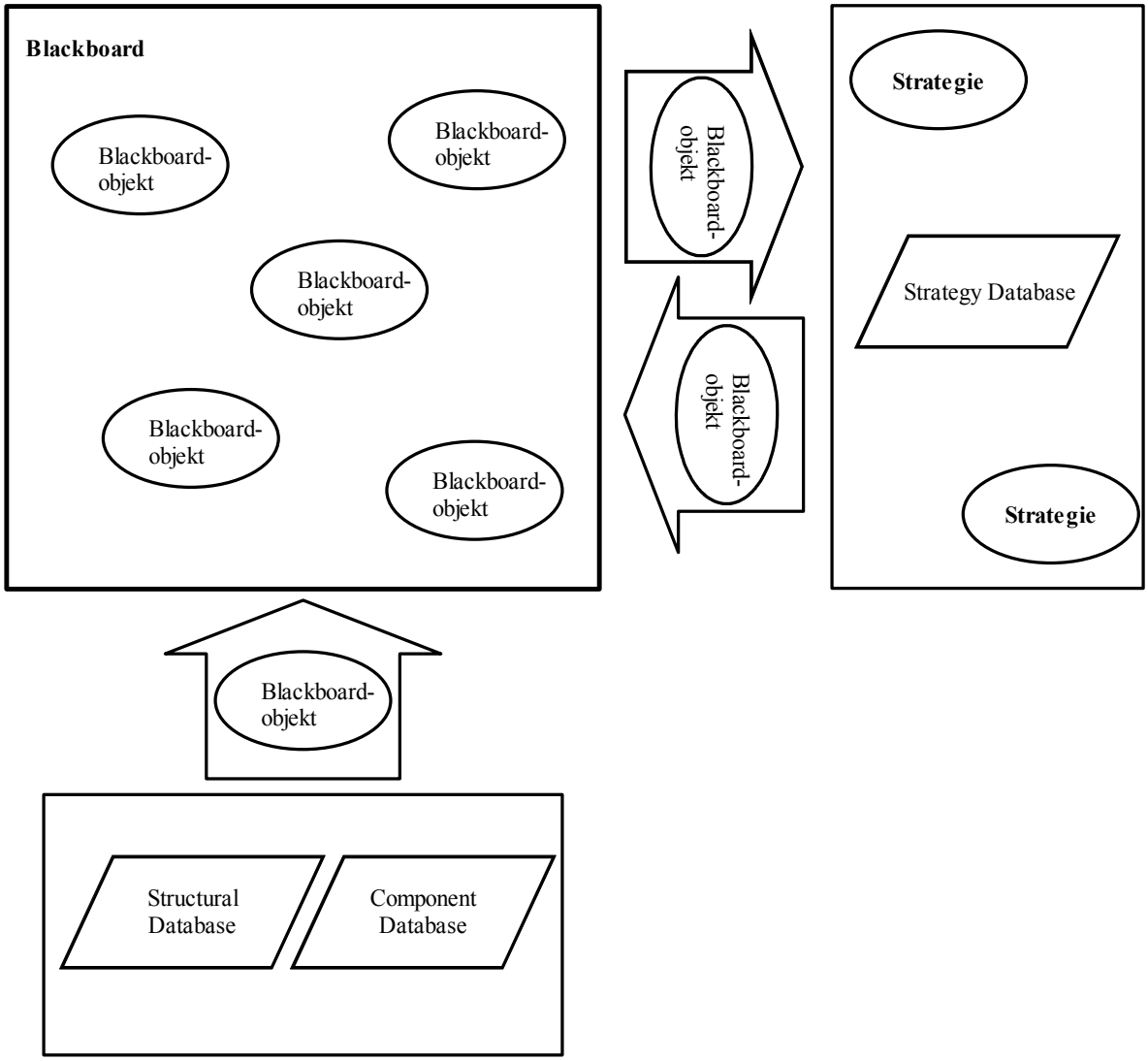



**Abbildung 4.4: Die Zusammenarbeit aller Komponenten**

Der Benutzer definiert in der Structural Database und Component Database den Aufbau des Systems und seiner Komponenten. Dieser Plan wird dann als Blackboardobjekt auf dem Blackboard abgelegt und ist somit für Strategien frei verfügbar. Danach erarbeitet der Benutzer eine Strategie, die die Design-Aufgabe lösen soll. Diese Strategie kann dann vom Benutzer in die Entwicklungsumgebung eingeladen werden.
Der Benutzer hat dann die Option die Strategie zu starten. Die Strategie selber arbeitet mit den Objekten, die auf dem Blackboard gespeichert sind. Eine Strategie kann während sie arbeitet Blackboardobjekte erzeugen und auf dem Blackboard ablegen.
Falls eine Strategie fehlerhaft ist, kann diese vom Benutzer gestoppt und bearbeitet werden.
Jede Strategie die vom Benutzer gestartet wird, kann auf dem Blackboard arbeiten und hat somit Zugriff auf die Blackboardobjekte.

Der Benutzer hat also jederzeit die volle Kontrolle über die Entwicklungsumgebung und kann jederzeit eingreifen. Damit eine Design-Aufgabe gelöst werden kann, müssen also nur Blackboardobjekte auf dem Blackboard vorhanden sein, die durch die Strategie verarbeitet werden können. Diese Blackboardobjekte, die zur Lösung einer Design-Aufgabe nötig sind, werden Werkzeuge/Tools/Module genannt. Die Module bieten der Strategie Funktionen an, die benötigt werden um bestimmte Teilaufgaben zu lösen. Die Abbildung von Struktur wäre ein Beispiel für diese Aufgabe.

# 5 Die Structural Database

In diesem Kapitel wird erläutert, was die Structural Database ist und welche Aufgabe sie erfüllt.

## 5.1 Aufgabe der Structural Database

Die Structural Database stellt Möglichkeiten zur Erzeugung und Beschreibung der Struktur des Zielsystems der Designaufgabe zur Verfügung. Ein Beispiel hierfür sind Design-Graphgrammatiken, diese ermöglicht es Strukturwissen mit Hilfe von Graphen abzubilden. Damit ist es möglich die Struktur eines Objektes einfach darzustellen ohne komplexe Formulierungen finden zu müssen.

## 5.2 Design-Graphgrammatiken

Eine Design-Graphgrammatik ist eine Mischung aus NLC/NCE Graph Grammatiken, einer HR Grammatik und einigen Erweiterungen.
Diese Mischung ermöglicht eine eindeutige Semantik für strukturelle Veränderungen des Modells, hiervon sind gerade Fälle betroffen in denen ein Modell vereinfacht oder transformiert werden soll. Das Erzeugen einer „Design-Graphgrammatik“ bedarf allerdings einer genauen Kenntnis über das Domänenwissen.

Damit ein System mit all seinen Anforderungen richtig dargestellt werden kann, werden unterschiedliche Operationen benötigt:

- Einfügen und Löschen von einzelnen Objekten in ein bestehendes System
- Ändern von bestimmten Objekten und deren Verbindung zu anderen Objekten
- Ändern ganzer Objektmengen

Alle diese Operationen können als Transformationen von Graphen aufgefasst werden. Ein Teil des Graphen wird durch einen anderen Graphen ersetzt, dabei müssen die bestehenden Kanten betrachtet werden und falls nötig neue Kanten eingefügt werden. Für dieses Konzept benötigt man den Begriff des „Matchings“.
Alle Graphen, die im Folgenden betrachtet werden, besitzen Kantenlabel und Knotenlabel, hierbei gibt es keine Einschränkung wie oft ein Label in einem Graphen verwendet werden darf. Das Konzept der Label macht es sehr schwierig, einen Knoten oder eine Kante direkt zu identifizieren, da das Label nicht eindeutig für einen Knoten oder eine Kante, sein muss.

### 5.2.1 Isomorphie von Graphen

Seien $G = (V, E)$ und $H = (V_H, E_H)$ zwei Graphen. Ein Isomorphismus liegt dann vor, wenn es eine bijektive Abbildung $\Psi: V \rightarrow V_H$ mit $\{a, b\} \in E \leftrightarrow \{\Psi(a), \Psi(b)\} \in E_H$ für alle $a, b \in V$.
Falls so eine Abbildung existiert, dann sind H und T isomorph.

Falls Graph G und Graph H aus Knoten und Kanten bestehen die ein Label besitzen, dann nennt man G und T isomorph, wenn gilt:
Es existieren Label Funktionen $\sigma$ und $\sigma_H$ mit $\sigma(a) = \sigma_H(\Psi(a))$ für alle $a \in V_G$ und
$\sigma(e) = \sigma_H(\Psi(e))$ für alle $e \in E$ mit $\Psi(e) = \{\Psi(a), \Psi(b)\}$ wenn $e = \{a,b\}$.

## 5.2.2 Matching

Seien H und T zwei Graphen, deren Knoten und Kanten Label besitzen, dann ist jeder Subgraph in H, der isomorph zu T ist, ein Matching und wird als „Matching von T in Graph H“ bezeichnet. Besteht der Graph T nur aus einem Knoten und es kann in H ein Subgraph gefunden werden zu dem T isomorph ist, so wird dieses Matching „Knoten basiertes Matching“ genannt.
In allen anderen Fällen wird das Matching „Graph basiertes Matching“ genannt.

Beispiel:
Gegeben seien zwei Graphen H und T, deren Knoten und Kanten Label besitzen.

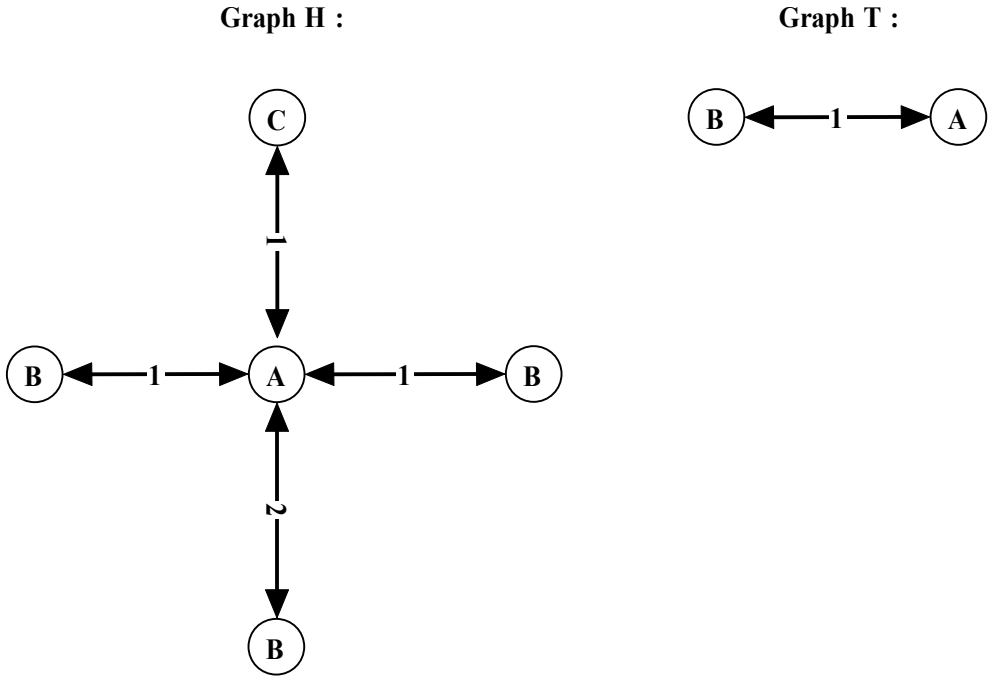



**Abbildung 5.1: Zwei Graphen mit Kantenlabeln und Knotenlabeln**

Ein Matching von Graph T in Graph H würde wie folgt aussehen:

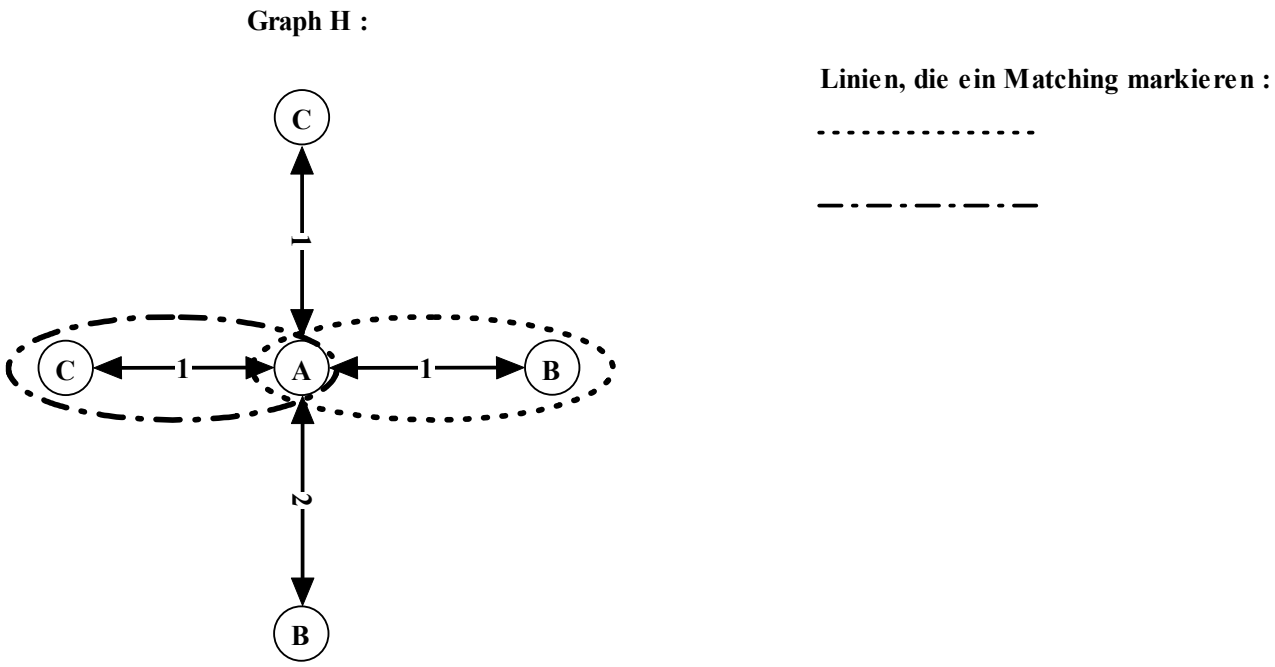



**Abbildung 5.2: Graph T, aus Abbildung 5.1, erzeugt zwei Matchings im Graphen H**

## 5.2.3 Einbettungsregeln

In einer Design-Graphgrammatik findet eine Graphtransformation statt, indem ein Subgraph T im Graphen H gesucht wird. Dieser Subgraph T wird dann durch einen neuen Graphen R ersetzt. Da aber der Subgraph T, der entfernt werden soll, sich in dem Graphen H befindet, muss geklärt werden, was mit den Kanten geschieht, die den Subgraphen T und den Graphen H verbinden. Außerdem muss geklärt werden, wie der neue Teilgraph R in den Graphen H eingefügt werden soll.

Beispiel:
Es sei der folgende Graph H und eine Design-Graphgrammatik Regel gegeben. Wird die Graph Grammatik Regel angewendet, dann würde der Knoten mit Label C in Graph H durch die rechte Seite der Graph Grammatik Regel ersetzt. Dabei würde das Problem auftauchen, wie mit der Kante, deren Label „1“ ist und die Knoten A und Knoten C in Graph H verbindet, verfahren werden soll.

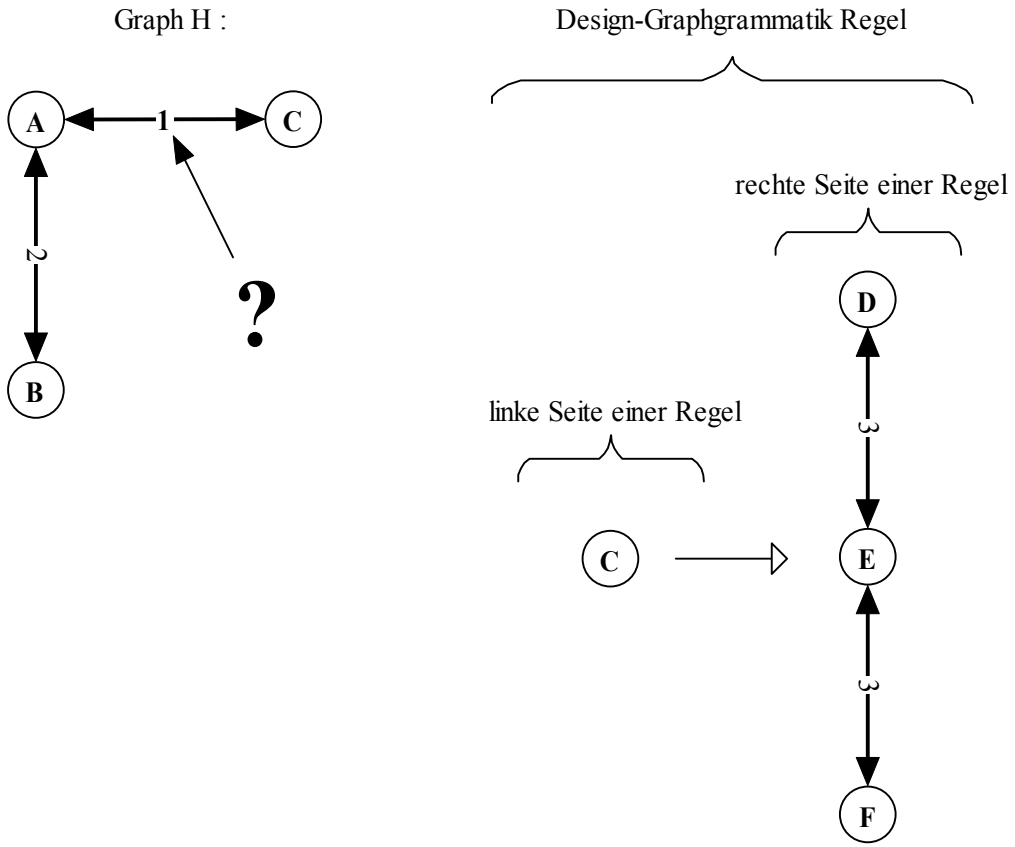



**Abbildung 5.3: Eine Design-Graphgrammatik Regel soll angewendet werden**

Definitionen 5.1:

Host-Graph H: Der Graph auf den die Graphtransformation angewendet werden soll

Graph M: M ist ein Subgraph in H, M ist dabei ein Matching M und T sind von der Struktur identisch.

Zielgraph T: Dieser Graph T wird in H gesucht, daraus entsteht M. T ist die linke Seite einer Graph Grammatik Regel.

Ersetzungsgraph R: Dieser Graph wird in H eingefügt, nachdem M gelöscht wurde. R ist die rechte Seite einer Graph Grammatik Regel.

Verbindungsknoten V: Diese Knoten sind nur in H vorhanden, sie haben eine gemeinsame Kante mit einem Knoten, der in Graph M enthalten ist.

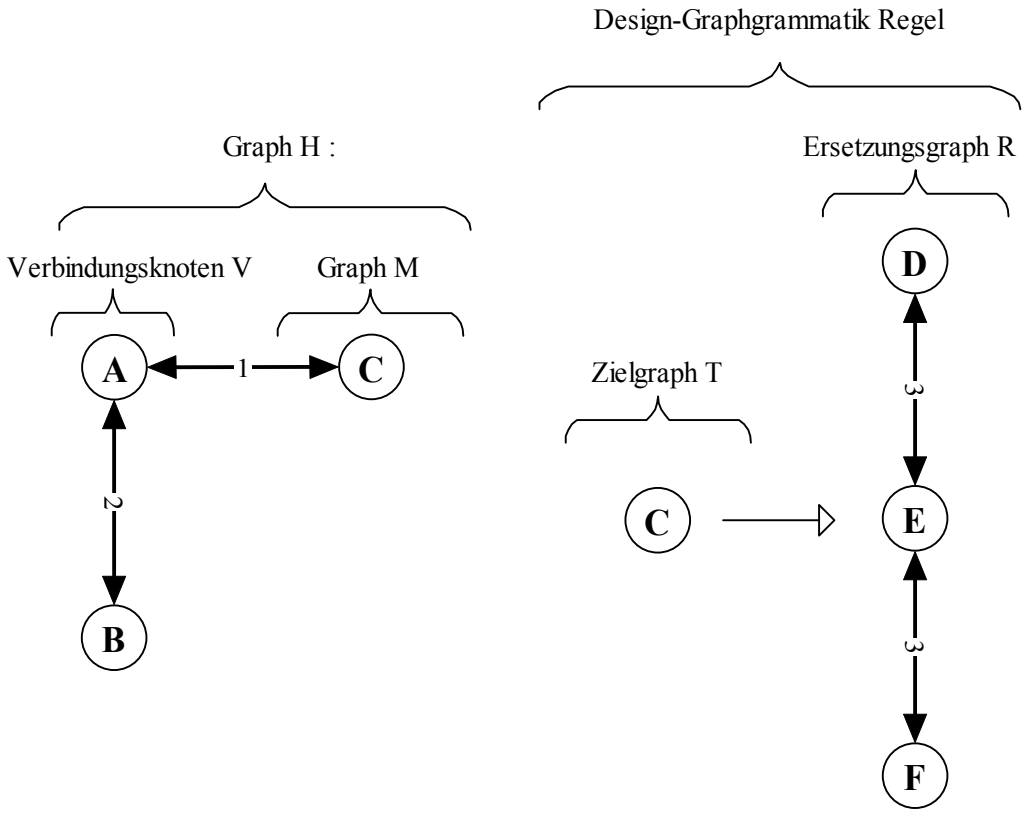



**Abbildung 5.4: Darstellung der Definition 5.1**

Eine kontextfreie Design-Graphgrammatik ist ein Tupel G = ( $\sum$ , P, s) mit:

- $\sum$ ist eine Menge von Knotenlabeln und Kantenlabeln, jeder Knoten und jede Kante besitzt ein Label
- P ist eine Menge von Design-Graphgrammatik Regeln
- s ist ein Startsymbol

Eine Graph Grammatik Regel P ist von der Form $T \rightarrow (R, I)$.
Der Graph T besteht dabei aus Knoten und Kanten, die Label besitzen. Der Graph T soll in dem Host-Graphen H durch Graph R ersetzt werden. Graph R ist ebenfalls ein Graph der aus Knoten und Kanten besteht, die Label besitzen.
Die Regel I ist eine so genannte „Einbettungsregel“. Ihre Aufgabe besteht darin die Kanten der Knoten V zu verarbeiten.

Eine Design-Graphgrammatik Regel wird wie folgt bearbeitet:

1. Suche ein Matching M mit Graph T in dem Graphen H.

2. Lösche M aus H, entferne dabei alle inzidenten Kanten.

3. Füge eine Kopie des Graphen R in Graph H ein, dabei müssen die Einbettungsregeln I beachtet werden.

Eine Einbettungsregel aus I ist ein Tupel ((h, t, e), (h, r, f)). Dabei gilt:

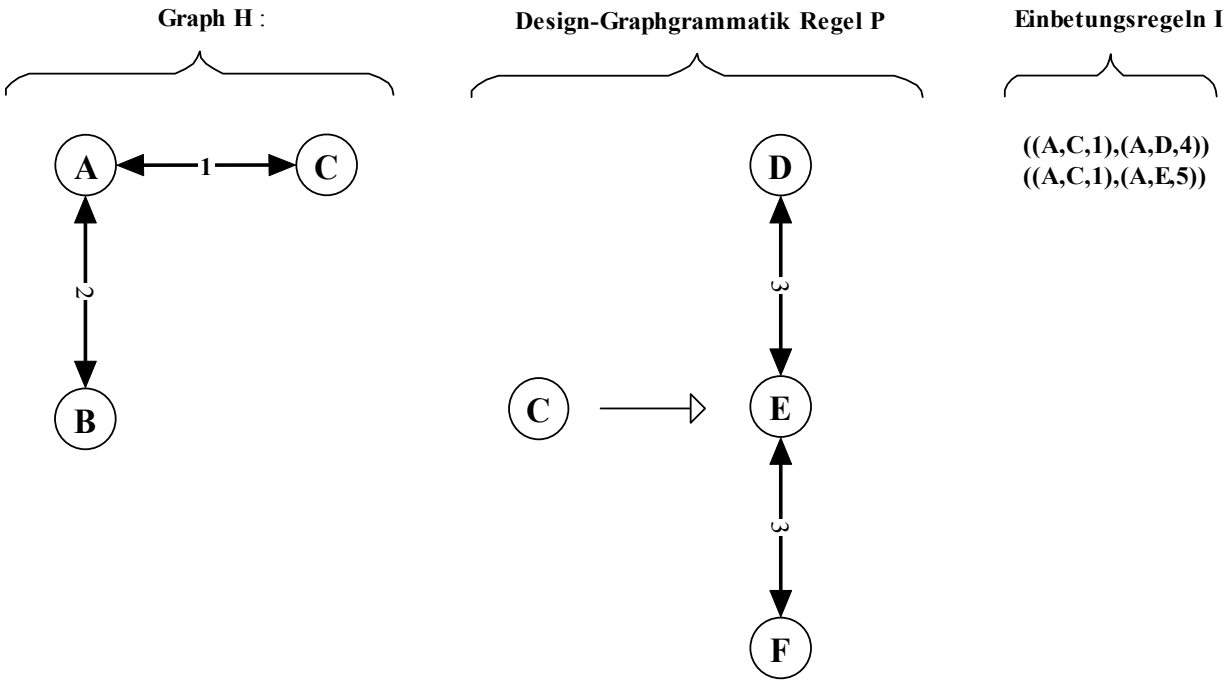
- $h \in \sum$ ist das Label eines Knotens v, dabei ist v in H enthalten, nicht aber in T.
- $t \in \sum$ ist das Label eines Knotens w, w ist dabei in T enthalten.
- $e \in \sum$ ist das Label der Kante, die {v,w} verbindet.
- $f \in \sum$ ist ein Label einer Kante, die Kante muss sich nicht unbedingt von der Kante mit Label e unterscheiden.
- $r \in V_r$ ist ein Knoten in R, dieser Knoten ist eindeutig zu identifizieren.

Eine Einbettungsregel wird wie folgt angewendet:

1. Finde eine Kante mit Label e, die einen Knoten mit Label h und einen Knoten mit Label t verbindet. Der Knoten mit Label h darf dabei nur im Graphen H, aber nicht in Graphen T zu finden sein. In Graph T befindet sich unter anderem der Knoten mit Label t.
2. Erzeuge eine Kante f.
3. Verbinde den Knoten, dessen Label h ist, mit dem Knoten r.

Falls keine Einbettungsregel in I für Kante mit Label e gefunden wird, dann wird die Kante mit Label e gelöscht und nur der Knoten r eingefügt.

Beispiel:
Es sei Graph H und eine Graph Grammatik Regel P gegeben.
Zu der Design-Graphgrammatik Regel gehören zwei Einbettungsregeln I:




**Graph H', nach der Anwendung einer Design-Graphgrammatik Regel**

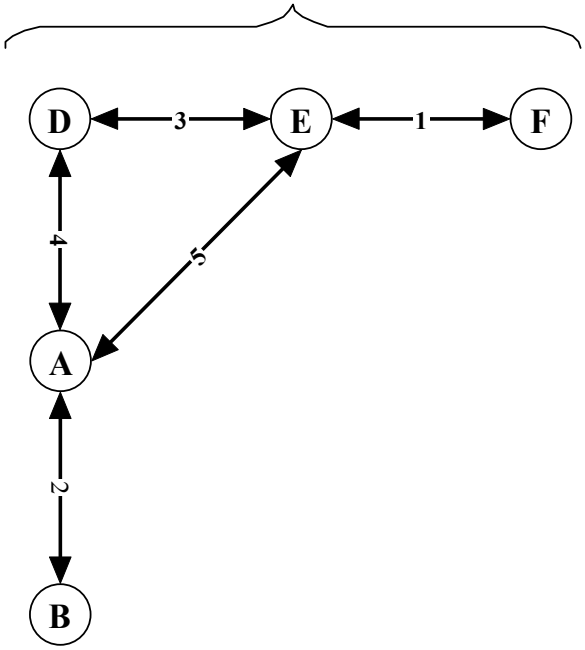



**Abbildung 5.5: Eine Design-Graphgrammatik Regel wurde auf Graph H angewendet**

## 5.3 Die Anwendung von Design-Graphgrammatiken in einem Design/Konfigurationssystem

Eine Design-Graphgrammatik ermöglicht es die Struktur eines Objektes genau abzubilden. Dabei kann genau festgelegt werden, wie sich Graphtransformationen auswirken.

Damit eine Konfigurationsaufgabe gelöst werden kann, ist es nötig, alle möglichen Lösungen zu betrachten. Natürlich kann man im Laufe der Konfiguration manche Lösungen verwerfen.
Zum Beispiel ist es nicht sinnvoll eine Lösung weiter zu betrachten, die bereits eine geforderte Eigenschaft verletzt. Als Beispiel könnte man sich eine Preisschranke überlegen, die nicht überschritten werden darf. Wird diese Preisschranke überschritten und es sollen weitere Komponenten hinzugefügt werden, ist dies nicht sinnvoll, da die zusätzlichen Komponenten den Preis weiter steigern werden. Die Teillösung kann verworfen werden.
Es ist allerdings nötig alle Teilkonfigurationen zu erzeugen damit diese bewertet werden können.

Als erstes muss erkannt werden, dass es mehrere Teilkonfigurationen gibt. In einer Design-Graphgrammatik wären dies die möglichen Matchings in einem Graphen. Das bedeutet, dass eine Regel mehrfach in einem Graphen angewendet werden kann oder aber, dass mehrere Regeln mehrfach auf einen bestimmten Teilgraph angewendet werden können.

Alle diese Teilkonfigurationen, die durch Regelanwendungen entstehen können, müssen betrachtet und weiter verarbeitet werden.
Daraus resultiert ein riesiger Suchbaum in dessen Blättern fertige Konfigurationen vorhanden sind.

Ein Suchbaum ist hierbei ein Baum mit Verzweigungsgrad $n \in N$.

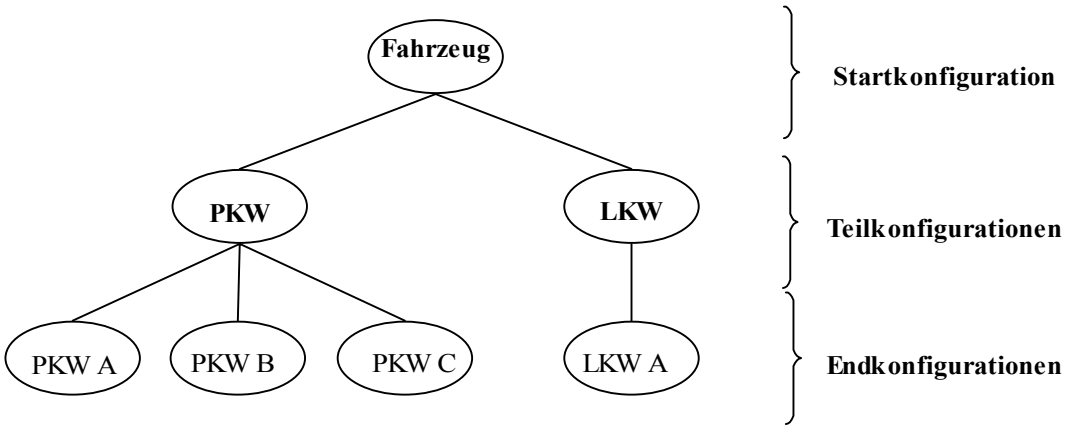



**Abbildung 5.6: Von einer Startkonfiguration, über Teilkonfigurationen, zu mehreren Endkonfigurationen**

## 5.3.1 Mögliche Suchverfahren in einem Suchbaum

Da es nötig ist alle Teilkonfigurationen zu betrachten, muss eine Technik gefunden werden mit der ein Suchbaum vollständig durchlaufen werden kann.
Das angestrebte Ziel, ist eine vollständige Suche. Dabei wird jedes Element im Baum genau einmal betrachtet [7].
Damit eine vollständige Suche garantiert werden kann, bieten sich zwei Techniken an [7].
Die erste Technik wird Breitensuche genannt, die zweite Tiefensuche.

Die Breitensuche betrachtet immer nur alle Teilkonfigurationen/Endkonfigurationen einer Ebene. Erst wenn alle Teilkonfigurationen/Endkonfigurationen einer Ebene betrachtet wurden, wird die nächste Ebene erzeugt und bearbeitet (Abbildung 5.7). Dabei muss auch die gesamte Ebene gespeichert werden, dies kann recht schnell zu Speicherproblemen führen.

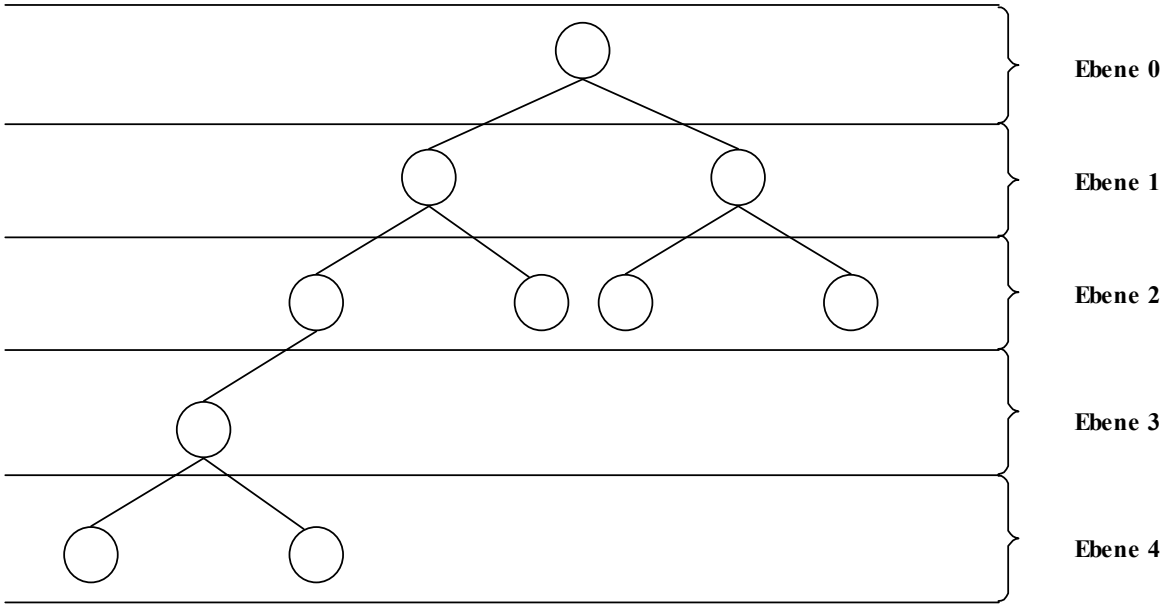



**Abbildung 5.7: Breitensuche**

Die Tiefensuche ist die zweite Möglichkeit einen Suchbaum zu betrachten, hierbei wird ein Weg im Suchbaum durchlaufen, bis dieser in einem Blatt endet (Abbildung 5.8).
Da ein Baum unendliche Tiefen haben kann, sollte die Tiefensuche mit einer Tiefenschranke kombiniert werden.
Das bedeutet, dass die Suche in dem Suchbaum abbricht, sobald der Weg in dem Suchbaum eine gewisse Länge erreiche hat. Auf diese Besonderheit muss in der Breitensuche nicht geachtet werden.

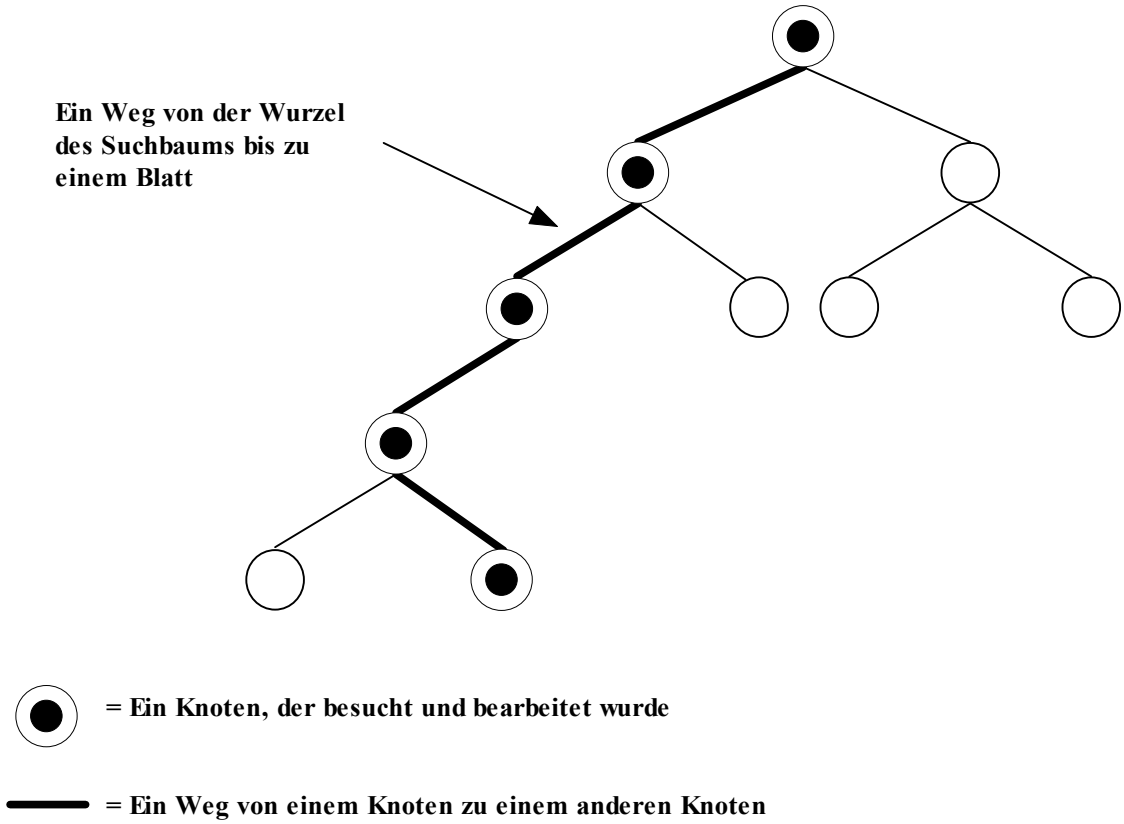



**Abbildung 5.8: Tiefensuche**

Sobald der Weg in einem Blatt endet, wurde eine Konfiguration gefunden.
Dies bedeutet aber nicht, dass alle Konfigurationen betrachtet wurden, es muss also sichergestellt werden, dass alle Blätter bearbeitet und somit alle möglichen Konfigurationen betrachtet werden.

Damit dies geschehen kann muss der gefundene Weg geändert werden, dies kann mit einer Technik realisiert werden die „Backtracking“ genannt wird.
Der Gedanke beim Backtracking ist, dass sobald die Tiefensuche auf ein Blatt trifft oder die Tiefenschranke überschritten wird, der letzte Schritt zurückgenommen wird.
Anstelle des letzten Schrittes wird ein alternativer Weg gewählt. Dadurch wird garantiert, dass der gesamte Suchbaum durchlaufen wird und somit alle Konfigurationen betrachtet werden.

## 5.3.2 Tiefensuche und Regelanwendungen

Bisher wurde erläutert was eine Design-Graphgrammatik ist und wie mit Hilfe einer Tiefensuche ein Suchraum durchlaufen werden kann. In diesem Kapitel sollen die beiden Verfahren zusammen gebracht werden.
Eine Design-Graphgrammatik (DGG) besitzt einen Startgraphen S, eine endliche Menge von Regelanwendungen P und eine Menge Σ von Labeln.
DGG = (S,P, ∑ )

Die Tiefensuche sorgt dafür, dass alle Regelanwendungen, die möglich sind auch durchgeführt werden. Dadurch wird der Suchraum vollständig durchlaufen. Da es durchaus sein kann, dass zwar Graphen gefunden werden, aber diese in der Strategie nicht verarbeitet werden können muss die Möglichkeit gegeben werden die Suche zu steuern.

Die Steuerung der Suche findet mit Hilfe zweier Parameter statt. Einmal die maximale Anzahl von Regelnanwendungen und die minimale Anzahl von Regelanwendungen. Die maximale Anzahl von Regelanwendungen ist eine Tiefenschranke wie sie in Kapitel 5.3.1 erläutert wurde. Es dürfen solange Regeln auf einen Graphen angewendet werden, bis die maximale Anzahl von Regelanwendungen erreicht wird, danach wird der Graph verworfen.
Die minimale Anzahl von Regelanwendungen bestimmt ab welchem Zeitpunkt ein Graph, auf den keine Regeln mehr angewendet werden kann, eine Lösung darstellt. Dieser Parameter gibt die Möglichkeit Graphen zu verwerfen obwohl auf diese keine Regel mehr angewendet werden kann und somit eigentlich eine Lösung gefunden wurde. Im ersten Moment ist dies ziemlich sinnlos, wenn allerdings eine Design-Graphgrammatik mehrfach nacheinander, mit unterschiedlichen Parametern für die maximale Anzahl von erlaubten Regelanwendungen, aufgerufen wird, dann macht es schon Sinn. Eine Strategie, die Graphen anfordert, wird diese Graphen auch weiter verarbeiten. Was passiert aber, wenn die Strategie die Parameter korrigieren muss, weil kein Graph in eine Lösung transformiert werden konnte? Die Strategie würde die Design-Graphgrammatik mit neuen Parametern starten. Dann würde die Design-Graphgrammatik „alte Graphen“ liefern die schon einmal nicht zum Ziel geführt haben und die Strategie würde die „alten Graphen“ erneut prüfen. Da aber die Strategie nun bestimmen kann ab welcher Regelanwendung ein Graph eine Lösung ist, kann sie schon einmal betrachtete Graphen sofort ausschließen lassen.

Der Algorithmus der Tiefensuche benutzt zwei Listen. Die erste Liste wird „Open-Liste“ genannt, auf ihr stehen alle Zustände, die noch nicht in einen gültigen Endzustand überführt werden konnten. Auf der „Closed-Liste“ stehen alle Zustände, die zum aktuellen Zustand geführt haben, eine Art Historie zum aktuellen Zustand.

Als erster Schritt wird der Startgraph S auf die Open-Liste gelegt, dies entspricht einem Eintrag auf der Open-Liste.

Im zweiten Schritt wird der erste Graph auf der Open-Liste betrachtet. Der Graph wird von der Open-Liste entfernt und auf die Closed-Liste gesetzt.
Ist die Open-Liste jedoch leer, so endet der Algorithmus.

In Schritt drei wird eine Regel aus der Regelmenge P ausgewählt und es wird versucht diese Regel auf den Graphen anzuwenden, es werden alle Matchings gesucht.
Kann keine Regel auf den Graphen angewendet werden, so ist der Graph in einem gültigen Endzustand und kann als Lösung weiter gegeben werden, wenn er alle Bedingungen erfüllt. Die Closed-Liste muss bereinigt werden.

In Schritt vier wird für jedes gefundene Matching die Regel angewendet. Der Graph, der dabei entsteht wird, betrachtet. Der neue Graph muss vorgegebene Bedingungen erfüllen, erst dann darf er weiter bearbeitet werden und somit an die Spitze der Open-Liste eingetragen werden.

In Schritt fünf wird geprüft, ob alle Regeln auf einen Graphen angewendet wurden. Ist dies der Fall, so wird mit Schritt zwei des Algorithmus fortgesetzt, andernfalls wird mit Schritt drei fortgesetzt.

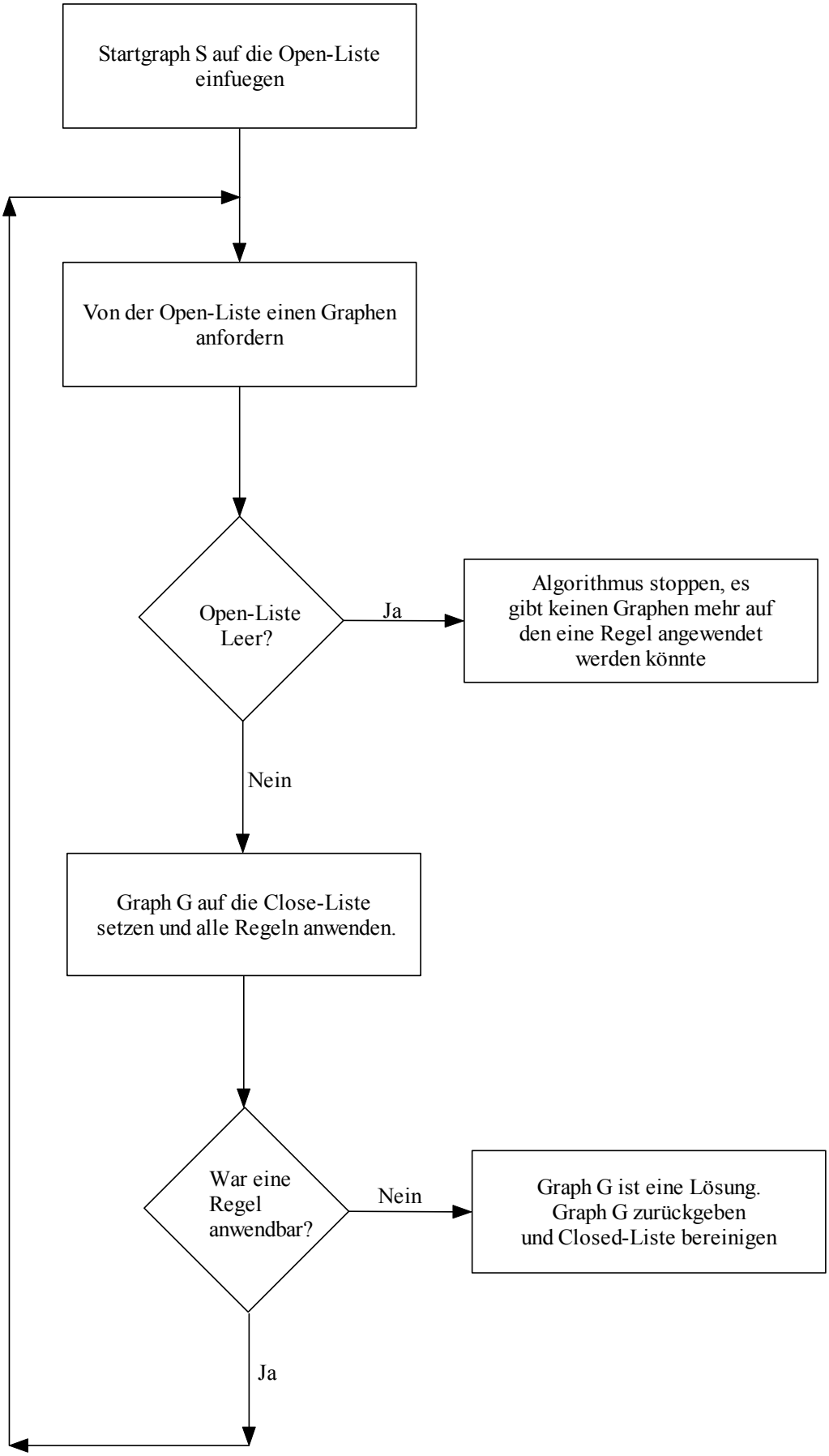



**Abbildung 5.9: Abstrakte Abbildung eines Algorithmus zur Bearbeitung von Design-Graphgrammatik Regeln**

Allerdings ist der Schritt „Graph auf die Closed-Liste setzen und alle Regeln anwenden“ und „Graph G ist eine Lösung“ sehr abstrakt und wird im Folgenden verfeinert.

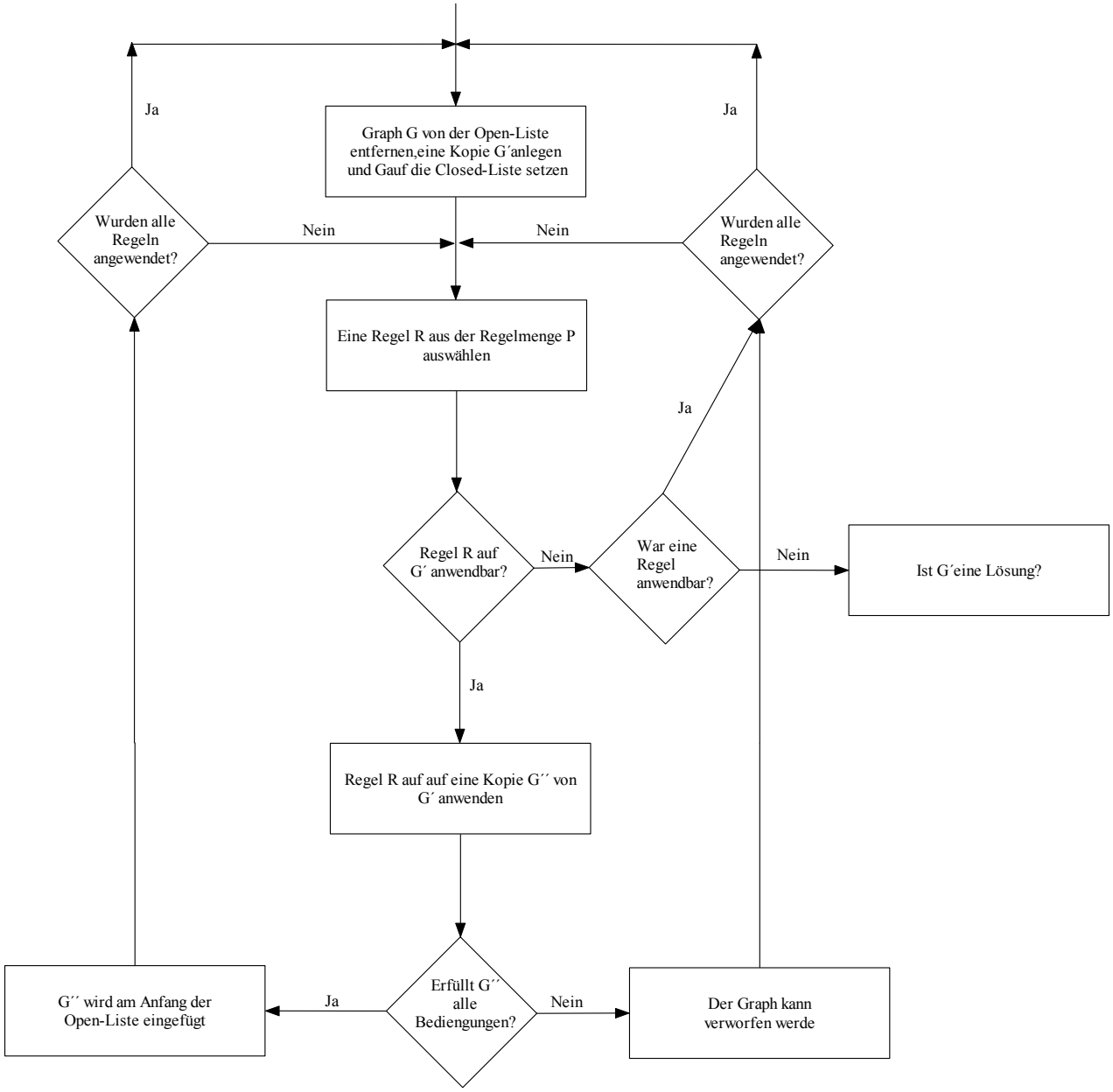



**Abbildung 5.10: Regelanwendungen und Lösungssuche**

In Abbildung 5.10 wurde der Schritt „Graph auf die Closed-Liste setzen und alle Regeln anwenden“ verfeinert. Als erstes wird ein Graph G von der Open-Liste entfernt und auf der Closed-Liste abgelegt. Von dem Graphen G wird eine Kopie G´ angelegt. Wenn auf G´ eine Regel angewendet werden kann, dann wird eine Kopie G´´ erzeugt. Der Graph G´´ wird anschließend mit der gerade gefundenen Regel verändert. Erfüllt G´´ nun noch die geforderten Bedingungen, dann kann er auf der Open-Liste abgelegt werden.
Die Bedingungen sind, dass ein Graph nicht schon einmal erzeugt wurde oder dass der Graph die maximale Anzahl von Regelanwendungen überschreitet.
Falls ein Graph dies tut, dann muss er verworfen werden. Sobald alle Regeln auf einem Graphen angewendet wurden, muss der nächste Graph betrachtet werden. Dazu wird vorher überprüft ob die Open-Liste noch Elemente enthält, siehe Abbildung 5.9.
Sobald auf einen Graphen keine Regel angewendet werden kann, muss geprüft werden ob G´ eine Lösung ist. Der Graph G´ ist eine Lösung, wenn keine Regel auf ihn angewendet werden konnte und er nicht gegen die oben genannten Bedienungen verstößt. Eine weitere Bedingung ist die

minimale Anzahl auszuführender Regelanwendungen. Eine Lösung ist nur dann gültig, wenn eine minimale Anzahl von Regelanwendungen ausgeführt wurde. Dies soll verhindern, dass beim erneuten starten des Algorithmus alte Lösungen erneut ausgegeben werden. Wenn beim ersten Start die maximale Anzahl von Regelanwendungen Zehn betrug und alle Graphen, die gefunden wurde, zu keiner Lösung weiter verarbeitet werden konnten. Dann wäre es unsinnig beim zweiten Start des Algorithmus Lösungen zu liefern, die mit weniger als Zehn Regelanwendungen erzeugt wurden, da diese bereits beim ersten Durchlauf keine Lösung erzeugen konnten.

## 5.4 Probleme, die bei der Anwendung einer Design-Graphgrammatik entstehen

Bisher wurde theoretisch über die Anwendung von Regeln auf einen Graphen gesprochen und wie dies theoretisch realisiert werden kann. Egal wie die Regeln auf einen Graphen angewendet werden, es wird sich nicht verhindern lassen, dass Graphen doppelt erzeugt werden. Ein einfaches Beispiel soll das Problem darstellen.

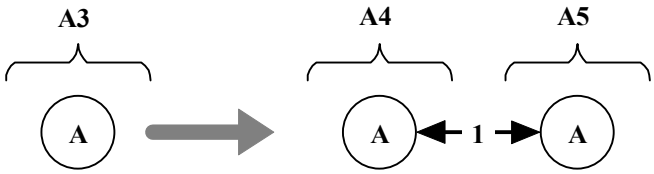



**Abbildung 5.11: Eine Design-Graphgrammatik Regel**

In Abbildung 5.11 ist eine Design-Graphgrammatik Regel dargestellt, die einen Knoten mit dem Label A in zwei neue Knoten umwandelt. Eine entsprechende Einbettungsregel sei vorhanden, wird hier aber nicht dargestellt.
Da alle Knoten das Label A tragen, wurden die Knoten extra mit A4 und A5 gekennzeichnet. Diese Informationen sind normalerweise nicht verfügbar.

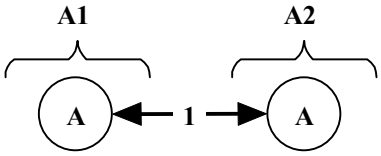



**Abbildung 5.12: Der Host-Graph**

Der Host-Graph besteht aus zwei Knoten, die mit einer Kante verbunden sind.
Die Knoten tragen beide das Label A und sind für den Leser mit A1, A2 markiert, diese Informationen stehen normalerweise nicht zur Verfügung.

Wird nun die Graph Grammatik Regel aus Abbildung 5.11 auf den Host-Graphen aus Abbildung 5.12 angewendet, so würde folgendes passieren:
Der Knoten A3, der die linke Seite einer Regel darstellt, würde mittels Matching im Host-Graphen gefunden. Alle möglichen Matchings wären Knoten A1 und Knoten A2.
Die Regel besagt nun, dass sowohl der Knoten A1 als auch der Knoten A2 ersetzt werden muss.

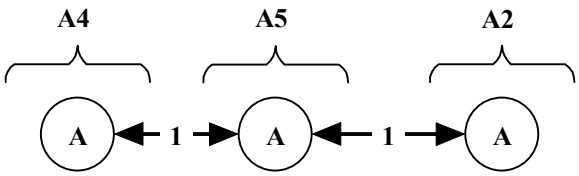



**Abbildung 5.13: Knoten A1 wurde durch die Regel aus Abbildung 5.11 ersetzt**

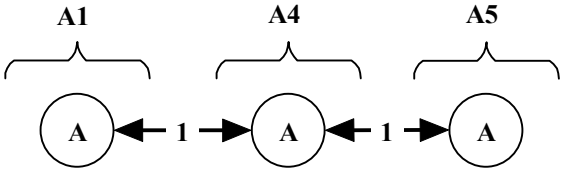



**Abbildung 5.14: Knoten A2 wurde durch die Regel aus Abbildung 5.11 ersetzt**

Die Abbildungen 5.13 und 5.14 zeigen das Resultat der Regelanwendung, natürlich werden beide Graphen auf die Open-Liste übernommen.

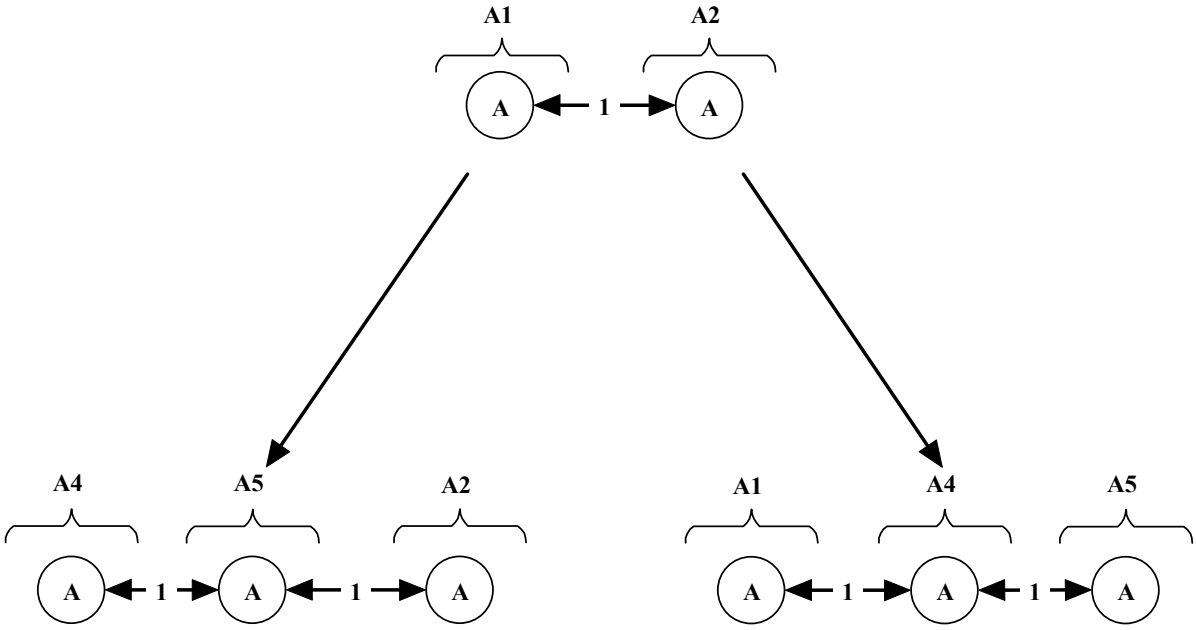



**Abbildung 5.15: Die Anwendung einer Regel führt zu zwei Einträgen in der Open-Liste**

Bei genauerer Betrachtung wird der Leser feststellen, dass die beiden Graphen, die entstanden sind, als isomorph bezeichnet werden können.
Dies hat große Konsequenzen für die weitere Anwendung von Regeln. Alle Regeln, die auf den Graphen aus Abbildung 5.13 angewendet werden können, können auch auf den Graphen aus Abbildung 5.14 angewendet werden, ohne dass dabei neues Wissen entsteht. Allerdings wird Rechenzeit und Speicherplatz verschwendet diese Graphen zu erzeugen.

Eine einfache Lösung wäre, beim Einfügen auf die Open-Liste zu prüfen, ob der Graph isomorph zu einem anderen Graphen ist, der bereits auf der Open-Liste existiert.

Bei der Tiefensuche wird immer nur der aktuelle Pfad zum derzeitigen Graphen und seine alternativen gespeichert. Eine Historie des Graphen ist also nur solange vorhanden, bis der Graph als beendet bezeichnet werden kann. Er verschwindet von der Open-Liste. Danach werden die Alternativen des beendeten Graphen bearbeitet.
Die einzigen Informationen, die bei der Tiefensuche zur Verfügung stehen, sind:

- Der aktuelle Graph (Open-Liste)
- Alternative Wege des aktuellen Graphen (Open-Liste)
- Weg, der zum aktuellen Graphen geführt hat (Closed-Liste)

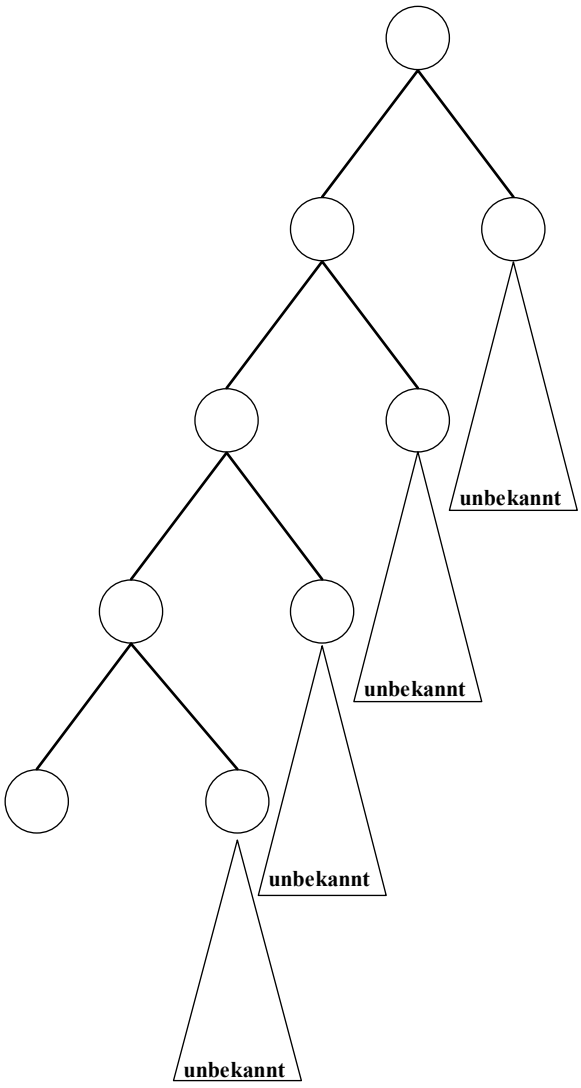



**Abbildung 5.16: Einträge der Open/Closed-Liste bei einer Tiefensuche**

In Abbildung 5.16 wird dieses Problem dargestellt. Die Knoten stehen auf der Open-Liste zur Verfügung und können geprüft werden. Sobald aber der Knoten „unten links“ abgearbeitet wird, steht er nicht mehr zur Verfügung und verschwindet von der Open-Liste.
Es kann nicht garantiert werden, dass ein Graph, der schon auf der Open-Liste war, nicht erneut in einem „unbekannten“ Teil des Baumes auftaucht.
Die Isomorphie von Graphen auf der Open-Liste hilft also nur begrenzt weiter doppelte Graphen zu erkennen. Es müsste gewährleistet werden, dass alle erzeugten Graphen zu jeder Zeit zur Verfügung stehen.

Eine Lösung wäre, alle Graphen in einer extra Liste zu speichern. Bei jedem neuen Eintrag auf die Open-Liste müsste dann diese Liste auf Isomorphie geprüft werden. Wird ein Graph, der in die Open-Liste eingefügt werden soll, in der „Liste aller Graphen“ gefunden, so darf dieser Graph nicht auf die Open-Liste gelangen, da er schon einmal betrachtet wurde.

Es entstehen somit zwei Probleme, Speicherplatz und Laufzeit.

Selbst bei relativ simplen Graph Grammatik Regeln, auf jeden Graphen können 2 Regeln angewendet werden, entstehen Unmengen neuer Graphen.
Bei 30 Regelanwendungen würden $2^{30}$ Graphen auf der letzten Ebene entstehen. Damit doppelte Graphen ausgeschlossen werden können, müssten alle Graphen auf Ebene $2^{29}$, $2^{28}$, $2^{27}$, ...., $2^{0}$ gespeichert werden.
Wenn eine Graph Grammatik keine Regel zum löschen von Graphen vorsieht und in jeder Regelanwendung einen Knoten erzeugt, dann hätten die Graphen nach 30 Regelanwendungen mindestens 30 Knoten.
Bei 4 Byte je Knoten wird somit der Speicherplatz ein großes Problem.
Es müssen also „alte“ Graphen die nicht mehr benötigt werden aus der Liste der gespeicherten Graphen entfernt werden. In jedem Betriebssystemhandbuch werden Techniken angeboten wie ein Speicher mit Seitenverdrängung mit einem Algorithmus zu realisieren ist. Dabei gehen natürlich Daten verloren.

Ein weitaus größeres Problem stellt die Laufzeit der Isomorphieprüfung dar.
Die Laufzeit eines Algorithmus, der zwei Graphen auf Isomorphie testet, ist O(n!).
Selbst wenn es gelingt, das Speicherproblem zu beheben, ist es unmöglich einen Isomorphietest auf große Graphen durchzuführen, da dieser einfach zu lange dauert.
Es muss also ein Weg gefunden werden, zwei Graphen auf Isomorphie zu testen, ohne wirklich Isomorhpie zu überprüfen. Hierfür bietet sich eine Approximation für Graphisomorphie an. Natürlich besteht bei einer Approximation die Gefahr, dass zwei Graphen als nicht isomorph bezeichnet werden obwohl sie es in Wirklichkeit sind und umgekehrt. Es soll nun ein Verfahren angegeben werden, mit dem ein Graph auf eine Zahl abgebildet wird. Wenn zwei Graphen auf eine identische Zahl abgebildet werden, könnten diese Graphen isomorph sein. Je genauer die Abbildung dabei ist, desto Wahrscheinlicher ist es, dass wirklich Graphisomorphie vorliegt, wenn zwei Graphen auf dieselbe Zahl abgebildet werden.

### 5.4.1 Charakteristika von Graphen, lokal

Da ein Isomorphie Test NP-vollständig ist, muss ein Weg gefunden werden einem Graphen „anzusehen“, ob er mit einem anderen Graphen isomorph ist, ohne wirklich einen Test auf Isomorphie durchzuführen.

Da in einer Design-Graphgrammatik mit Graphen gearbeitet wird, die Label besitzen, können folgende Charakteristika eines Graphen als Kriterium für vorhandene Isomorphie gelten:

- Knotenanzahl
- Kantenanzahl
- Label

Daraus kann gefolgert werden, dass ein Graph, dessen Knotenanzahl und Kantenanzahl gleich sind und deren Label mit einer bijektiven Funktion in die natürlichen Zahlen abgebildet wurden, isomorph sein können.
Leider wird hierbei der Struktur des Graphen keine Rechnung getragen.

Beispiel 1:

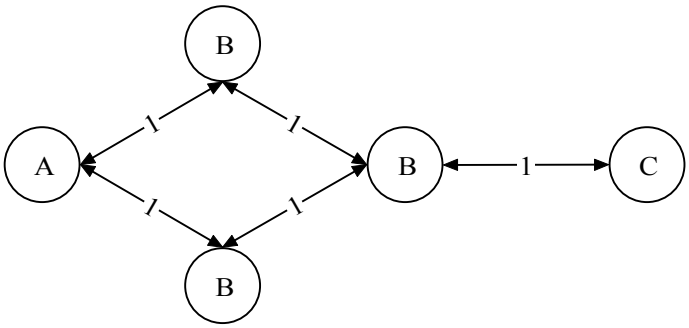



**Abbildung 5.17: Graph 1**

In Abbildung 5.17 ist ein Graph dargestellt, gegeben sei eine Funktion f(x), f: Label → N, x ∈ Label, die ein Label in eine natürliche Zahl umwandelt.
f(A) = 1
f(B) = 2
f(C) = 3
f(1) = 1

Dabei soll Folgende Formel eine Zahl berechnen, die den Graphen repräsentiert:
„Addition aller Label“ + Anzahl Kanten + Anzahl Knoten.

Für Graph 1 würde somit 1+2+2+2+3+1+1+1+1+1+5+5 = 25 eine Zahl darstellen, die den Graphen repräsentiert.

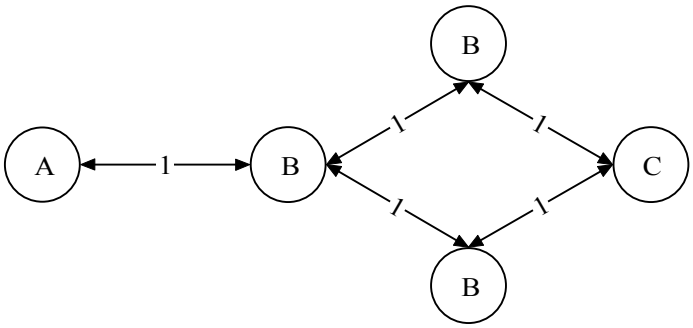



**Abbildung 5.18: Graph 2**

In Abbildung 5.18 wird Graph 2 dargestellt. Obwohl dieser Graph eine völlig andere Struktur aufweist, würde Graph 2 ebenfalls durch die Zahl 25 repräsentiert.
Erst eine Prüfung auf Graphisomorphie würde ergeben, dass die Graphen nicht gleich sind. Dies ist aber aus Zeitgründen nicht möglich.

Eine einfache Addition der Knotenanzahl, Kantenanzahl sowie aller Label eines Graphen reicht nicht aus, um die Struktur eines Graphen genau genug wiederzugeben.

Die folgende Technik weist jedem Knoten einen Wert zu, dieser Wert repräsentiert den Knoten.

$Knoten_n$ = Knoten n des Graphen
Label = beliebige Zeichenkette, sowohl Kanten als auch Knoten besitzen ein Label
F(x) = bijektive Funktion, die Label x in eine natürliche Zahl umwandelt
$LK_n$ = Label des $Knoten_n$
$\#Ka_n$ = Anzahl Kanten des $Knoten_n$
i = Kante i des $Knoten_n$ (i <= #Ka)
$LKa_i$ = Label der Kante i, die zu $Knoten_n$ gehört
$LKn_i$ = Label des Knotens zu dem eine Kante i existiert
$W_n$ = Wert des $Knoten_n$

$$W_n = LK_n + \#Ka_n + \sum_{i=1}^{\#Ka_n} (LKa_i + LKn_i)$$

Eine Möglichkeit den Wert eines Graphen als Zahl darzustellen wäre es, alle lokalen Informationen und die Werte der Knoten, zu addieren. Leider reichen die lokalen Informationen nicht aus, um die globale Struktur eines Graphen genau genug abzubilden.

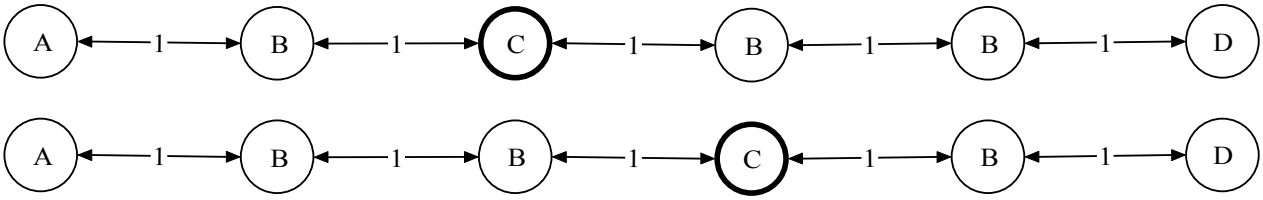



**Abbildung 5.19: lokale Strukturinformationen versagen**

Die beiden Graphen in Abbildung 5.19 sind für das Versagen der lokalen Informationen ein Beispiel. Beide Graphen würden denselben Wert erzeugen und wären somit „isomorph“. Natürlich sind die Graphen nicht isomorph.

## 5.4.2 Charakteristika von Graphen, global

In Kapitel 5.4.1 wurde gezeigt, dass der lokale Wert eines Knotens nicht ausreicht, um die globale Struktur eines Graphen genau genug abzubilden. Damit die globale Struktur eines Graphen erfasst werden kann, stehen folgende Merkmale zur Verfügung:

- Knotenanzahl
- Kantenanzahl
- Label

In Abbildung 5.19 versagt die Addition der lokalen Werte der Knoten, da die „Entfernung“ von Knoten C zu allen anderen Knoten keine Rolle spielt. Falls es gelingt diese „Entfernung“ abzubilden, wäre die globale Struktur erfasst. Abbildung 5.20 soll dies illustrieren.

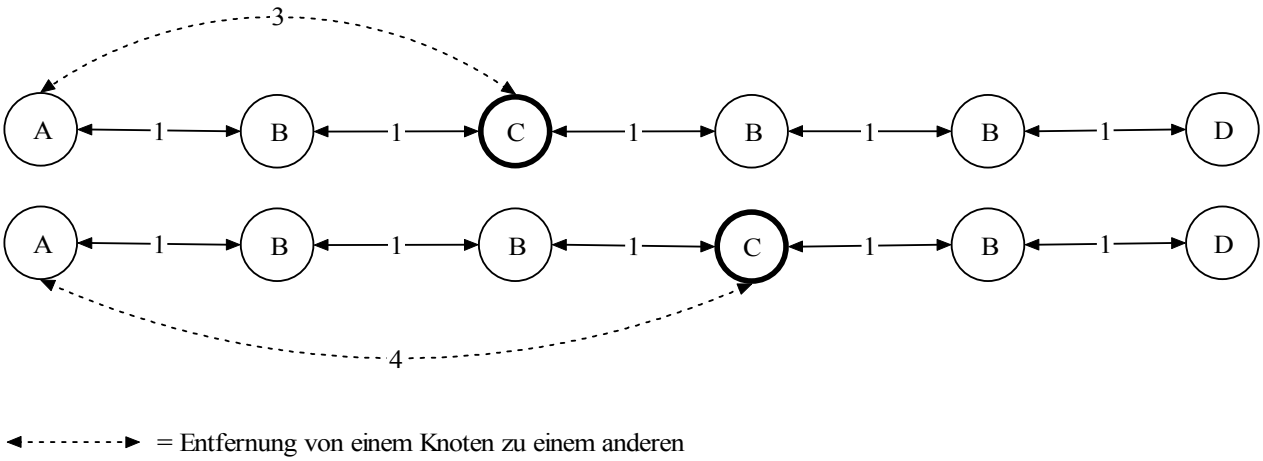



**Abbildung 5.20: Entfernung von Knoten A zu Knoten C am Beispiel zweier Graphen**

Die Entfernung ist die Ebene auf dem ein Knoten bei einer Breitensuche gefunden werden würde. Die Breitensuche garantiert, dass alle Knoten des Graphen besucht werden. Dabei startet sie bei einem fest vorgegebenen Startknoten und zwar bei allen Graphen, somit kann garantiert werden, dass bei allen Graphen die gleiche Berechnung stattfindet. Durch die Breitensuche wird gleichzeitig die Entfernung zu einem Startknoten erzeugt. Jeder Knoten, der sich auf einer Ebene befindet hat den gleichen Abstand zu einem Startknoten (Abbildung 5.21).

Das Problem, das nun besteht, ist einen Knoten zu finden, der als Startknoten für die Breitensuche fungiert. Alle Graphen haben folgende Eigenschaften:

- Ungerichtet
- Keine Quelle/Senke
- Kein fester Start/Endknoten, der in allen Graphen vorhanden ist
- Keinen Knoten, der garantiert vorhanden ist
- Keine Vorgabe der Labelnamen, die Label können frei gewählt werden

Keiner der aufgezählten Punkte kann als Anhaltspunkt für einen Knoten gewählt werden, der immer als Startknoten für die Breitensuche dienen soll.

Eine Design-Graphgrammatik besteht aus:

- $\sum$ , eine Menge von Knotenlabeln und Kantenlabeln
- P, eine Menge von Graph Grammatik Regeln
- S, ein Startsymbol

Die Label, die in einem Graphen auftauchen, sind somit in der Menge $\sum$ vorhanden, diese Tatsache alleine hilft nicht weiter. Allerdings wird die Menge $\sum$ in einem Programm mit einer Programmiersprache verarbeitet und eine Menge wird in einer Programmiersprache nicht als Menge, sondern kann als Array, Liste, Feld, Keller, usw. dargestellt werden.
Die Menge $\sum = \{A, B, C, D\}$ könnte also als Array wie folgt dargestellt werden:

Array „Menge$\sum$“ = array[4];
Menge$\sum$[0] = A
Menge$\sum$[1] = B

Menge $\sum$ [2] = C
Menge $\sum$ [3] = D

Somit hätte die Menge aller Label eine Ordnung. Label A hätte Arrayplatz 0, dies erlaubt den gezielten Zugriff auf Label eines Graphen.
Wenn das Label an der Stelle „Menge $\sum$ [0]“ auch im Graphen vorhanden ist, dann wird der Knoten, der das Label trägt, als Startknoten für die Breitensuche benutzt. Ist kein Knoten vorhanden, wird das Label an Stelle „1“ des Arrays benutzt, usw.

Der Algorithmus, der den Wert eines Graphen berechnet, würde wie folgt vorgehen:

1. Position = 0
2. Nehme das Label, das an Stelle „Position“ in dem Array gespeichert ist
3. Suche das Label in dem Graphen; ist das Label vorhanden? „ Ja, Schritt 4“, „Nein, Position + 1, Schritt 2“
4. Breitensuche mit allen Knoten, die in Schritt 3 gefunden wurden, dabei wird der Wert der Knoten verrechnet
5. Ausgabe „Wert des Graphen“

Als nächstes wird Schritt 4 des Algorithmus näher betrachtet. Nachdem ein/mehrere Knoten gefunden wurde, dienen diese Knoten als Start einer Breitensuche.
Jeder Knoten, der mit der Breitensuche erreicht wird, bekommt eine Zahl zugewiesen. Diese Zahl beschreibt auf welcher Ebene der Breitensuche der Knoten gefunden wurde.
Dieser Wert wird dann mit dem „Wert des Knotens“ multipliziert und auf den Wert des Graphen addiert. Der „Wert des Knotens“ bleibt dabei unberührt.
Die Breitensuche wird für jeden Knoten wiederholt der in Schritt 3 gefunden wurde.

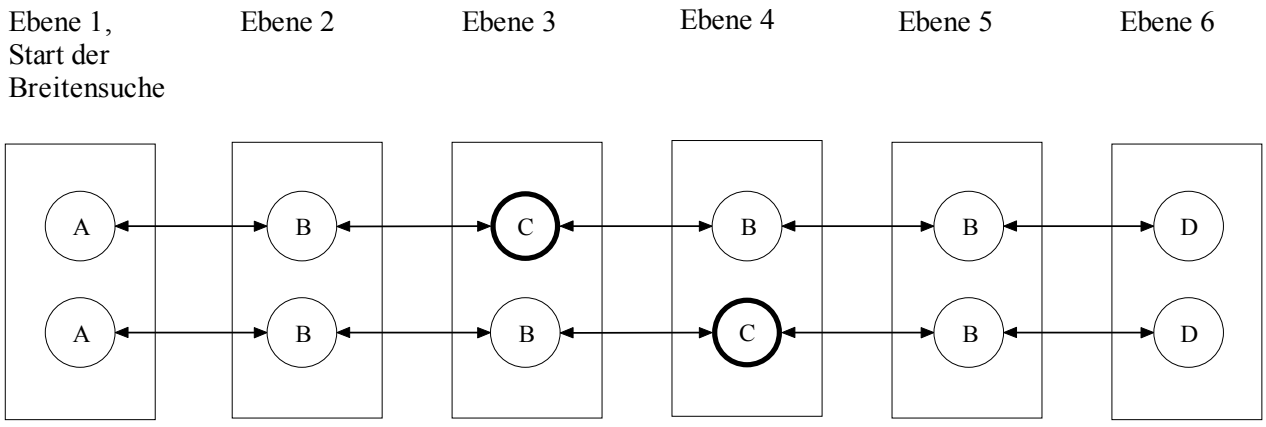



**Abbildung 5.21: Ergebnis der Breitensuche mit Knoten A als Startknoten**

In Abbildung 5.21 wurde der Algorithmus auf die beiden Graphen angewendet. Die Menge $\sum$ = {A, B, C} wurde durch ein Array: Array[0] = A, Array[1] = B, Array[2] = C, dargestellt. Der Startknoten für die Breitensuche wäre somit ein Knoten mit Label A.
Im oberen Graphen, aus Abbildung 5.21, würde der „Wert des Knotens C“ mit der Zahl 3 multipliziert, in dem unteren Graphen würde der „Wert des Knotens C“ mit der Zahl 4 multipliziert. Der Wert der beiden Graphen setzt sich aus der Summe der „Werte der Knoten“ * „Ebene der Knoten“ zusammen.

### 5.4.3 Beispiel

In diesem Kapitel soll an einem Beispiel erläutert werden, wie zwei Graphen auf Graphisomorphie getestet werden.
Die Menge $\sum$ = {A, B, C, D, 1} besteht aus 5 Elementen
Es sei eine Funktion F(X), die alle Label aus $\sum$ in die natürlichen Zahlen ableitet, gegeben.
F(A) = 1; F(B) = 2; F(C) = 3; F(D) = 4; F(1) = 4

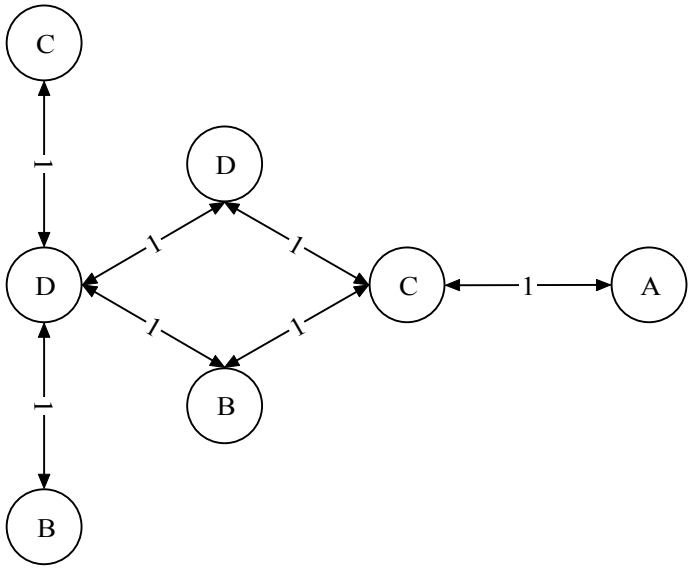



**Abbildung 5.22: Graph 1**

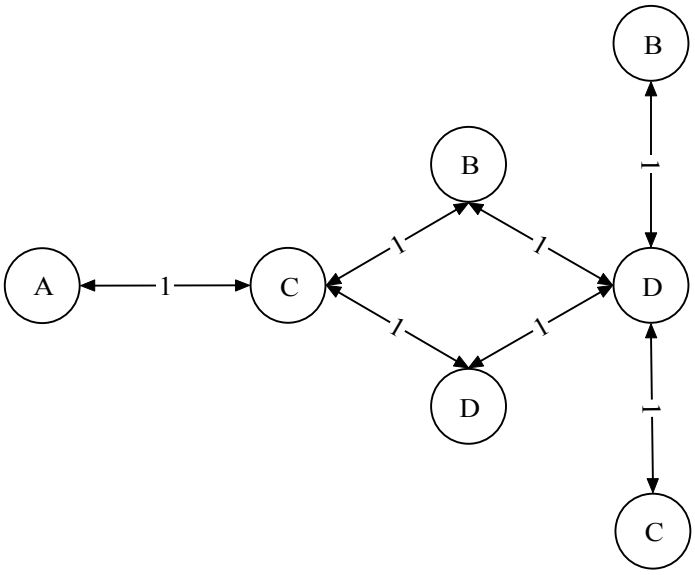



**Abbildung 5.23: Graph 2**

Bei beiden Graphen, die in Abbildung 5.22, 5.23 auf Graphisomorphie getestet werden sollen, werden als erstes die Werte der Knoten berechnet.

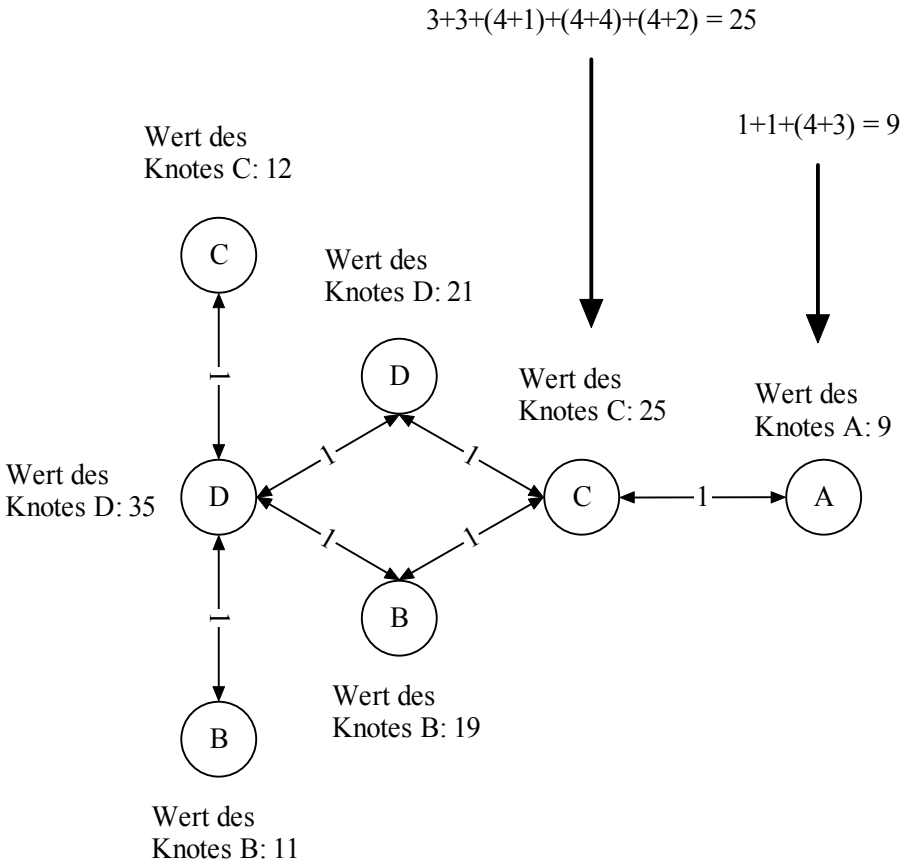



**Abbildung 5.24: „Werte der Knoten“ für Graphen 1**

In Abbildung 5.24 sind die Werte für jeden Knoten aus Graph 1 dargestellt. Diese Werte dienen als Grundlage, um den Wert des Graphen zu bestimmen.
Als nächstes muss der „Startknoten“ für die Breitensuche gefunden werden, als Grundlage dient $\sum = \{A, B, C, D, 1\}$. Da die Menge $\sum$ in einem Array gespeichert wurde (Array[0] = A; Array[1] = B; …; Array[4] = 1) und auf das erste Element des Arrays zugegriffen wird, dient Knotenlabel „A“ als Auswahl für den Startknoten, da Label A in dem Graphen vorhanden ist.

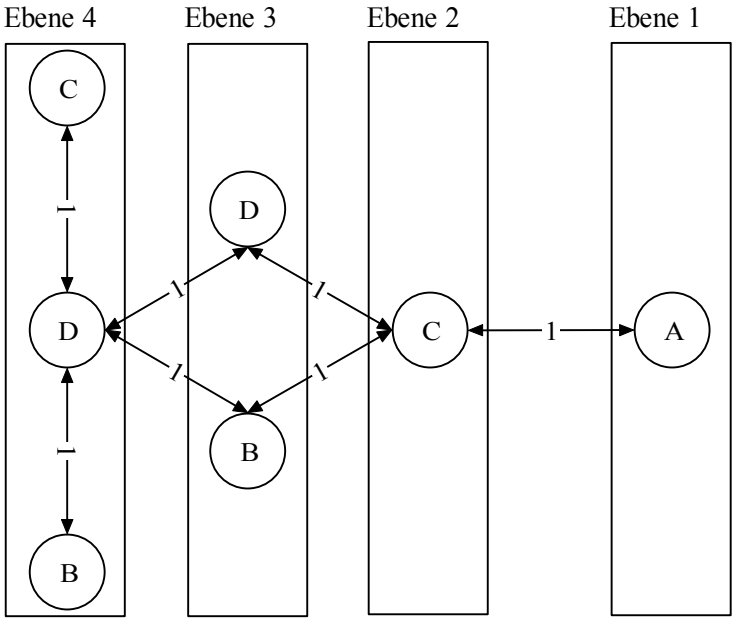



**Abbildung 5.25: Ebenen, die durch die Breitensuche entstehen**

Der Wert des Graphen berechnet sich also wie folgt:

9 * 1 + // alle Werte der Knoten * Ebene
25 * 2 +
21 * 3 + 19 * 3 +
12 * 4 + 35 * 4 + 11 * 4
= 411

Der Graph aus Abbildung 5.22 würde somit durch die Zahl 411 repräsentiert.
Der Graph aus Abbildung 5.23 würde ebenfalls durch die Zahl 411 repräsentiert, somit wären beide Graphen sehr wahrscheinlich isomorph.

### 5.4.4 Zusammenfassung

Der Wert eines Graphen setzt sich aus den Werten seiner Knoten und deren Entfernung zu einem/mehreren bestimmten Knoten zusammen. Dadurch wird es möglich, Graphisomorphie sehr schnell zu testen (vergleich zweier Integer Zahlen). Der Nachteil liegt bei der Berechnung, die den Graphen auf eine Zahl abbildet. Diese sollte sehr exakt sein, da das Abbilden zweier unterschiedlicher Graphen auf ein und denselben Wert das Löschen eines Lösungsweges und aller möglichen Lösungen die draus resultieren, bedeutet.
Da nun jeder Graph durch eine Integer Zahl abgebildet werden kann, können mehr „Graphen“ als vorher gespeichert werden, allerdings bleibt das Problem, dass nicht alle Graphen gespeichert werden können, bestehen. Es muss also eine Datenstruktur geschaffen werden, die „alte“ Werte, die nicht mehr benötigt werden, löscht. Die Technik „Least Recently Used“ bietet sich in diesem Fall an [8].
Hierbei werden alle Werte verdrängt, die am Längsten nicht mehr benutzt wurden. Dies lässt sich durch eine Liste realisieren, in die nur am Anfang eingefügt werden darf. Dadurch rutschen Werte, die lange nicht mehr benutzt wurden, automatisch nach hinten. Falls nun Platzbedarf entsteht, werden die Werte, die „hinten“ in der Liste stehen, gelöscht.
Ein erneutes Aufrufen eines Wertes sortiert diesen automatisch an den Anfang der Liste ein.

# 6 Die Strategy Database

In diesem Kapitel wird die Aufgabe und Funktionsweise der Strategy Database beschrieben.

## 6.1 Aufgabe der Strategy Database

Die Strategy Database enthält alle Informationen wie eine Design-Aufgabe gelöst werden kann. Sie enthält also eine Strategie zur Lösung der Design-Aufgabe.

Unter Strategie wird in diesem Zusammenhang ein Lösungsweg verstanden, der vom Anfang des Problems bis zu einem fertigen Design führt.
Dabei kann es verschiedene Strategien geben, die zum gewünschten Design führen.

Eine Strategie kann sich hierbei in disjunkten Zuständen befinden, es existieren fünf Zustände für jede Strategie.

- Strategie arbeitet: Die Strategie wird benutzt, sie hat vollen Zugriff auf das Blackboard und kann dort Objekte anlegen/löschen/ändern.
- Strategie gestoppt: Die Strategie hat keinen Zugriff auf das Blackboard und deren Objekte.
- Strategie pausiert: Die Strategie wurde vom Benutzer pausiert, angeforderte Blackboardobjekte bleiben im Besitz der Strategie.
- Strategie bearbeiten: Die Strategie wird vom Benutzer geändert, sie muss zuerst gestoppt werden.
- Strategie gelöscht: Die Strategie wurde vom Benutzer gelöscht, sie muss zuerst gestoppt werden.

Der Zustand „Strategie arbeitet“ bedeutet, dass die Strategie Objekte vom Blackboard angefordert hat und mit diesen arbeitet. Sie arbeitet entweder bis zu einem fest definierten Ende, dieses Ende wird in der Strategie selber definiert oder sie wird vom Benutzer gestoppt oder pausiert.

Der Zustand „Strategie gestoppt“ bedeutet, dass der Benutzer die Strategie ändern darf. Nachdem er seine Änderungen vorgenommen hat, kann er die Strategie erneut starten, diese wird dann in ihren Anfangszustand versetzt und beginnt zu arbeiten.
Eine gestoppte Strategie kann auch gelöscht werden. Die gestoppte Strategie gibt alle angeforderten Blackboardobjekte wieder frei.

Der Zustand „Strategie pausiert“ bedeutet, dass die Strategie derzeit nicht arbeitet, sie hat weiterhin Zugriff auf das Blackboard. Dabei behält sie alle angeforderten Blackboardobjekte und kann vom Benutzer jederzeit gestartet werden.
Wenn der Benutzer die Strategie startet, wird diese an der Stelle weiter arbeiten an der sie pausiert wurde, sie wird nicht am Anfangszustand der Strategie neu beginnen.
Eine pausierte Strategie kann jederzeit gestoppt werden.

Der Zustand „Strategie bearbeiten“ bedeutet, dass die Strategie gestoppt wurde und derzeit vom Benutzer geändert wird. Wenn der Benutzer die Strategie neu startet, wird die bearbeitete Strategie am Anfangszustand der Strategie starten.

Eine Strategie kann sich im Zustand „gelöscht“ befinden, wenn sie vom Benutzer gestoppt wurde und anschließend gelöscht wird. Die Strategie verliert durch das stoppen jeden Zugriff auf das

Blackboard und kann vom Benutzer nur erneut geladen werden, um dann in den Zustand „Strategie arbeitet“ versetzt zu werden.

Aufgrund dieser Zustände ist es möglich, dass der Benutzer fehlerhafte Strategien zur Laufzeit des Programms erkennen und stoppen kann. Nachdem die Strategie gestoppt wurde, kann der Benutzer den Fehler beheben und die geänderte Strategie erneut starten.

## 6.2 Möglichkeiten eine Strategie zu beschreiben

Eine Strategie ist ein Lösungsweg, der vom Benutzer vorgegeben wird. Allerdings stellt sich somit die Frage, wie der Benutzer „der Maschine“ mitteilen will, wie seine Strategie aussieht.
Es muss also geklärt werden, wie eine Strategie vom Benutzer formuliert wird, um dann anschließend von „der Maschine“ verarbeitet werden zu können.

Es existieren sicher viele Bücher mit dem Thema „Mensch Maschine Kommunikation“, es werden hier allerdings nur exemplarisch drei mögliche Lösungen vorgestellt

### 6.2.1 Beschreibung einer Strategie mit Alltagssprache

Eine denkbare Lösung wäre, dass der Benutzer einfach in seiner Alltagssprache eine Strategie formuliert und diese in einer Textdatei ablegt.
Diese Textdatei müsste dann von „der Maschine“ verarbeitet werden können.

Bsp.: „Suche aus allen Motoren die Motoren heraus, die mindestens 100 PS haben und weniger als 50 kg sollten sie auch wiegen, das wäre ganz toll.“

Die Aufgabe der Maschine wäre die Analyse der menschlichen Sprache. Aufgrund dieser Analyse würde dann ein lauffähiges Programm erzeugt.
Ein Problem ist, dass sich die Alltagssprache der Menschen unterscheidet. Jeder Benutzer formuliert seine Sätze etwas anderes und es gibt sicherlich einen großen Unterschied zwischen der Formulierung eines Laien und der eines Experten.
Die Hauptaufgabe wäre es, die menschliche Sprache in allen möglichen Formen zu analisieren und daraus lauffähige Programme zu erzeugen. Eine verbale Beschreibung in Alltagssprache bietet sich also nicht an, da diese Aufgabe einfach zu komplex ist und den Rahmen dieser Diplomarbeit sprengen würde.

### 6.2.2 Beschreibung einer Strategie mit einer Skriptsprache

Eine andere Möglichkeit eine Strategie zu beschreiben bestände darin, eine Skriptsprache zu implementieren, mit der eine Strategie beschrieben werden kann.

Bsp.:
gesuchteMotoren = „alleMotoren/(PS < 100)“ (geschnitten mit) „alleMotoren/(Gewicht > 50)“

Als erstes müsste eine Skriptsprache mit Regeln festgelegt werden. Diese Skriptsprache könnte allerdings von keiner gängigen Programmiersprache direkt ausgeführt werden. Es wäre also nötig einen Interpreter zu implementieren, der aus der Skriptsprache ein lauffähiges Programm der jeweiligen Sprache erzeugt.

Erst nach diesem Schritt ist es möglich eine Strategie durch eine Programmiersprache ausführen zu lassen.

Ein großer Vorteil der Skriptsprache ist, dass die Formulierung der Strategie fest definiert ist und dass sich ein Laie und ein Experte, von der Form her, gleich ausdrücken müssen.
Auch sprachliche Hindernisse spielen keine Rolle, da die Skriptsprache eindeutig ist, egal welche Muttersprache der Bediener benutzt.

### 6.2.3 Beschreibung einer Strategie mit einer Programmiersprache

Bisher traten zwei Probleme auf:

- Eine klar definierte Sprache, die Strategien beschreiben kann, muss vorhanden sein
- Es wird ein Interpreter benötigt, um formulierte Strategien in ein lauffähiges Programm umzuwandeln

Eine andere Möglichkeit diese beiden Probleme zu lösen, ist die Verwendung einer Programmiersprache zur Beschreibung der Strategie.

Was sind die Vorteile einer Strategie, die direkt in einer Programmiersprache formuliert wird?

Eine Strategie wird klar definiert, egal welche Sprache der Benutzer spricht oder welches technische Wissen er besitzt. Der Benutzer muss sich an die sprachlichen Gegebenheiten der Programmiersprache halten.

Es ist nicht mehr nötig einen Interpreter zu schreiben, der die formulierte Strategie in ein lauffähiges Programm umwandelt.

Die syntaktische/lexikalische Analyse wird vom Compiler der jeweiligen Programmiersprache übernommen.

Für diese Diplomarbeit wurde der Weg gewählt, die Strategien mit Hilfe der Programmiersprache Java zu beschreiben.

## 6.3 Probleme beim Beschreiben einer Strategie mit einer Programmiersprache

Die Vorteile, die eine Beschreibung einer Strategie mit einer Programmiersprache mitbringt, wurden bereits genannt. Allerdings treten hierbei auch einige technische Probleme auf.

Eine Strategie wird als Klasse implementiert, diese Klasse wird dann vom ausführenden Programm benutzt.

Es soll aber dem Benutzer jederzeit die Möglichkeit geboten werden, Strategien zu ändern und diese erneut auszuführen.
Dies bereitet keine Schwierigkeiten, wenn die Strategie und das Programm zum Ausführen der Strategie, zusammen kompiliert werden. Die Strategie liegt dem ausführenden Programm als Klasse vor, somit sind alle Methoden der Strategieklasse bekannt.

Was würde aber passieren, wenn die Strategie geändert werden muss?
Der gesamte Quellcode müsste erneut kompiliert und gestartet werden, da die Strategie geändert wurde.

Ein weiteres Problem ist, dass eine Strategie dem Benutzer, der das Programm bedient, zum Startzeitpunkt des Programms noch gar nicht bekannt sein muss.
Der Benutzer lädt lediglich seine Nutzdaten auf das Blackboard und will dann eine Strategie neu erzeugen.
Ist dies der Fall, dann ist dem Programm zum Zeitpunkt der Kompilierung die Klasse, der neuen Strategie, nicht einmal bekannt.

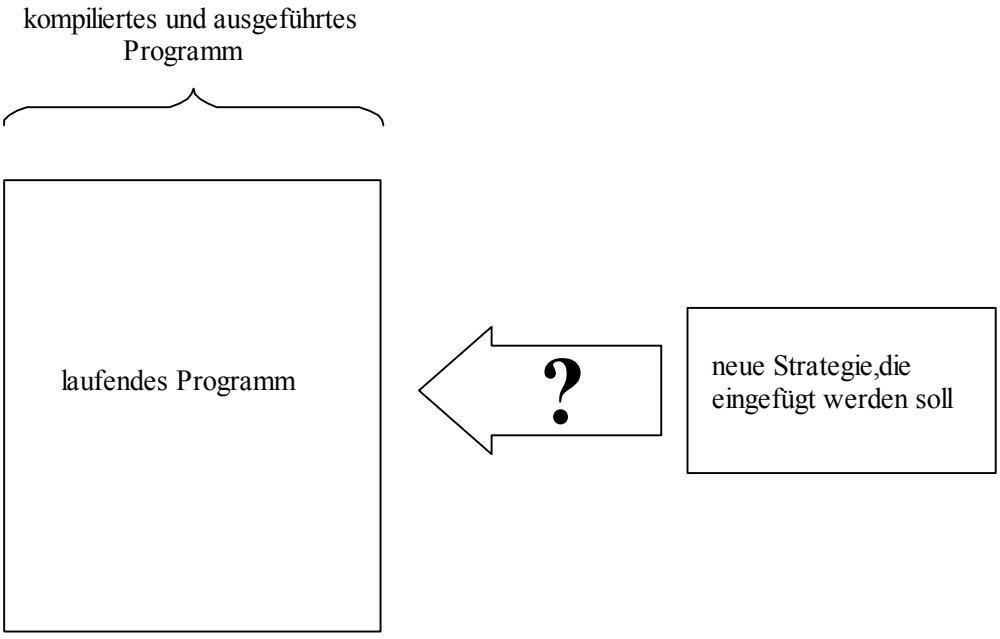



**Abbildung 6.1: Ein Programm, dass abgearbeitet wird, soll eine neue Strategie einbinden**

Die eben genannten Probleme lassen sich zu einem Problem zusammenfassen. Es muss der Programmiersprache möglich sein, eine Klasse nachträglich zu benutzen, obwohl das eigentliche Programm schon kompiliert wurde und die Klasse, die benutzt werden soll, zum Zeitpunkt der Kompilierung völlig unbekannt ist.

Das Hauptproblem ist, dass der Compiler zum Zeitpunkt der Kompilierung, alle Informationen über die Klasse benötigt, die Informationen werden aber erst vom Benutzer erzeugt, wenn das Programm bereits kompiliert wurde und arbeitet.

Es muss also ein Weg gefunden werden, eine Art Platzhalter zu schaffen, mit dem der Compiler zwar arbeiten kann, der es aber ermöglicht, dem Benutzer diesen Platzhalter durch seine Strategie zu ersetzen.

### 6.3.1 Vererbung

Die Idee der Vererbung ist es, Informationen/Methoden einer Klasse zu zuschreiben, diese Klasse wird Vaterklasse genannt.
Alle Klassen die ebenfalls die Informationen/Methoden der Vaterklasse benötigen, erben dann von der Vaterklasse diese Informationen/Methoden. Durch die Vererbung von

Informationen/Methoden ist es nicht mehr nötig, diese explizit in den Klassen, die von der Vaterklasse erben, zu erwähnen.

Ein Beispiel hierfür wäre, wenn verschiedene Autotypen verwalten werden müssten.
Eine Möglichkeit bestände darin, für jeden Autotyp eine Klasse zu erzeugen. In dieser Klasse könnte das Gewicht des Fahrzeugs, die Farbe und Fahrzeugtyp gespeichert sein.
Bei 10 Autotypen müssten somit 30 verschiedene Variablen verwaltet werden.

Die Vererbung erlaubt es, dass eine Vaterklasse „Fahrzeug“ erzeugt wird, und nur die Vaterklasse erhält die Variablen, die bei allen Klassen vorhanden sein soll (Gewicht, Farbe, PS, usw.). Die Unterklassen erben dann von der Vaterklasse „Fahrzeug“ und erhalten somit die Variablen der Vaterklasse, ohne dass diese explizit in der Unterklasse angegeben werden müssen.

Die gleiche Technik funktioniert auch bei Methoden. Wenn eine Vaterklasse eine Methode hat, dann kann diese von einer Unterklasse, die von der Vaterklasse erbt überladen werden.

Bsp.:
Die Vaterklasse enthält die Funktion „Ausgabe“, eine andere Klasse erbt von der Vaterklasse und enthält auch die Methode „Ausgabe“. Wird die Methode „Ausgabe“ benutzt, dann wird die Methode der Unterklasse aufgerufen, die Methode der Vaterklasse wird überladen.

Es ist möglich die Methode einer Vaterklasse zu überschreiben. Dies kann nun ausgenutzt werden, um eine Klasse nachträglich einzuladen.

In dem Hauptprogramm wird eine „leere Strategie“ benutzt, diese hat nur eine Methode „Strategie ausführen“. Mit dieser Methode kann der Compiler arbeiten, da sie zum Zeitpunkt des Kompilierens bekannt ist. Alle neuen Strategien erben von dieser „leeren Strategie“ und überschreiben die Methode „Strategie ausführen“. In der Methode, „Strategie ausführen“, der neuen Strategie steht dabei der auszuführende Quellcode.

Die neue Strategie kann natürlich jederzeit neu kompiliert werden ohne, das ausführende Programm zu unterbrechen. Ist die Klasse kompiliert, wird sie einfach eingeladen und überschreibt die Methode der Vaterklasse und steht somit zur Verfügung.
Dadurch ist es möglich eine Strategie zu ändern, ohne das Programm, das die Strategie ausführen soll, zu beenden und neu zu kompilieren.

Bisher wurde das Konzept eine Klasse nachträglich zu laden und zu benutzen nur sehr allgemein beschrieben, in diesem Abschnitt soll verdeutlicht werden, wie diese Technik mit der Programmiersprache Java verwendet werden kann.

Die Grundidee ist, dass eine Vaterklasse kompiliert wird. Nach der Kompilierung wird eine neue Klasse geladen, diese neue Klasse überschreibt die Methode der Vaterklasse.

Somit sind 2 Fragen zu klären:

- Wie funktioniert Vererbung in Java und wie werden Methoden der Vaterklasse überladen
- Wie kann eine Klasse zur Laufzeit eines Programms geladen werden

Das erste Problem, das gelöst werden muss, ist dass der Compiler zum Zeitpunkt der Kompilierung alle Methodennamen und Variablen einer Klasse kennen muss, damit diese im Quellcode benutzen werden können.

Da der Benutzer aber Methoden und Variablen erst noch erstellt, muss dem Compiler eine Art Platzhalter angeboten werden, mit dem er arbeiten kann. Dieser Platzhalter wird dann überladen und so kann der neue Quellcode genutzt werden.
Dies geschieht mit der Technik „Vererbung und Überladen von Methoden“, die hier nun kurz erläutert wird.

Als erstes muss eine Vaterklasse existieren, die dem Compiler zum Zeitpunkt des kompilieren bekannt ist. Im Folgenden wird diese Vaterklasse auch „Basisstrategie“ genannt.
Die Klasse „Basisstrategie“ enthält nur eine Methode namens „Ausgabe“.
Die Methode „Ausgabe“ hat als einzige Aufgabe den Text, „Ich bin eine Methode der Basisstrategie“, auf dem Bildschirm zu erzeugen.

Eine zweite Klasse wird im Folgenden als „bestimmte Strategie“ bezeichnet. Diese erbt von „Basisstrategie“ und enthält ebenfalls die Methode „Ausgabe“. Allerdings ist ihre Bildschirmausgabe anders. Die Ausgabe lautet diesmal: „Ich bin eine Methode der bestimmten Strategie“.

Wird nun ein Objekt vom Typ „bestimmte Strategie“ erzeugt und die Methode Ausgabe aufgerufen, dann wird auf dem Bildschirm „Ich bin eine Methode der bestimmten Strategie“ erscheinen. Die Methode der Vaterklasse wird überladen und nicht mehr ausgeführt.

An dieser Stelle verzichte ich absichtlich auf ausführlichen Quellcode, da Vererbung in jedem Java-Lehrbuch ausführlich beschrieben und in aller Ausführlichkeit dargestellt wird.

Das zweite Problem, welches gelöst werden muss, ist das Einladen einer Klasse mit Hilfe von Java. Die Programmiersprache Java bietet hierbei die Möglichkeit, eine Klasse nachträglich zu laden. Diese Technik wird anhand eines kleinen Beispiels erläutert.

Als erstes wird eine Klasse vom Typ „Class“ erzeugt.

```
Class neueKlasse = Class.forName(“Name”);
```

Der Befehl Class.forName(„Name“) legt dabei fest, welche Klasse eingeladen werden soll. Die Klasse muss als „class“ Datei vorliegen.

Zu diesem Zeitpunkt ist der Typ der Variablen „neueKlasse“ „class“.

Als nächstes wird eine Instanz der Klasse erzeugt.

```
Object instanz = neueKlasse.newInstance();
```

Der Typ der Variablen „instanz“ ist nun vom Typ der angegebenen Klasse, in diesem Beispiel war die Klasse also „Name“.

Zu diesem Zeitpunkt ist die neue Klasse geladen worden. Allerdings kann auf keine Variable oder Methode der neu geladenen Klasse zugegriffen werden, da zum Zeitpunkt der Kompilierung keine Information über die geladene Klasse vorliegt.

An dieser Stelle werden „Vererbung und Überladen von Methoden“ und „dynamisches Laden von Klassen“ zusammen geführt.

### 6.3.2 Kombination von Vererbung und dynamisches Laden von Klassen

Bisher können Klassen dynamisch eingeladen werden. Diese können aber nicht benutzen werden, da ihre Methoden/Variablen zum Zeitpunkt der Kompilierung unbekannt sind.

Eine Kombination von „Überladen einer Vaterklasse“ und „dynamisches Laden einer neuen Klasse“ würde dieses Problem lösen.

Die Vaterklasse enthält leere Methoden, die dem Compiler bekannt sind. Die neuen Klassen werden zur Laufzeit dynamisch eingeladen und überschreiben die leeren Methoden der Vaterklasse.

An dieser Stelle wird sehr explizit auf den Java Quellcode eingegangen, da diese Technik keine Standarttechnik ist und auch nicht in Lehrbüchern zu finden ist.

Bsp.:

In dem Programm, das die neue Strategie ausführen soll, existiert eine leere Strategie, die „Basisstrategie“ heißt. Sie hat als einzige Methode „Strategie ausführen“. In dieser Methode steht kein Quelltext.

Eine zweite Methode liegt nur als .class Datei vor. Sie ist dem anderen Programm völlig unbekannt, ihr Name ist „bestimmte Strategie“. Sie enthält auch die Methode „Strategie ausführen“, in dieser Methode ist allerdings eine Strategie zur Lösung einer Design-Aufgabe implementiert.
Es soll nun die „bestimmte Strategie“ in das laufende Programm eingebunden werden, damit die Strategie, die der Benutzer implementiert hat, benutzt werden kann.

```
// Es wird eine Klasse „class“ erzeugt, diese verweißt schon auf die Klasse „bestimmte
// Strategie“
Class neueKlasse = Class.forName("bestimmte Strategie");

// Eine Instanz wird erzeugt
Object instanz = neueKlasse.newInstance();

// Bisher wurde nur eine Klasse dynamisch eingeladen, an dieser Stelle wird nun die
// Vererbung benutzt

// Es wird eine neue Klasse erzeugt und ein Cast auf Basisstrategie ausgeführt, dabei wird die
// Methode der Basisstrategie überladen
Basisstrategie NeueStrategie = (Basisstrategie) instanz;

// Hier wird die überschriebene Methode aufgerufen
NeueStrategie. Strategie ausführen();
```

Es würde nun der Quellcode der Klasse "bestimmte Strategie" ausgeführt, obwohl dem Hauptprogramm zum Zeitpunkt des Kompilierens der Quellcode völlig unbekannt war.

### 6.3.3 Java spezifische Probleme

Bisher wurde gezeigt, dass es problemlos möglich ist eine Klasse einmal zur Laufzeit des Programms einzuladen.
Was passiert aber, wenn eine Strategie fehlerhaft ist und sie geändert werden muss?
Die Antwort ist einfach, die Klasse muss erneut geladen werden.

Ein kleines Beispiel soll die Problematik darlegen.

Die Strategie, die eingeladen wird, hat nur eine Methode, die„Ausgabe“ heißt, und gibt auf dem Bildschirm folgenden Text aus: „Ich bin eine Schtrategie“.

```
// Die Klasse wird geladen
Class Strategie = Class.forName("Beispielstrategie");
Object instanz = Strategie.newInstance();
Basisstrategie NeueStrategie = (Basisstrategie) instanz;
NeueStrategie.Ausgabe();

// An dieser Stelle bemerkt der Benutzer die Ausgabe und seinen Rechtschreibfehler. Er wird
// die Methode ändern und die Strategie erneut laden lassen, ohne das Programm zu
// beenden und neu zu kompilieren.
.
.
.
// Der Benutzer lädt die geänderte Strategie
Class Strategie = Class.forName("Beispielstrategie ");
Object instanz = Strategie.newInstance();
Basisstrategie NeueStrategie = (Basisstrategie) instanz;
NeueStrategie.Ausgabe();
```

An dieser Stelle sollte nun als Bildschirmausgabe der Text „Ich bin eine Strategie“ erfolgen.
Aber anstelle des richtigen Textes erscheint erneut „Ich bin eine Schtrategie“.
Die geänderte Klasse wird völlig ignoriert, die alte Klasse, obwohl sie geändert wurde, wird erneut geladen.

Mit Java ist es also nicht möglich, eine Klasse die eingeladen wurde, einfach zu überschreiben. Es ist auch nicht möglich eine Klasse einfach zu „entladen“ oder zu „vergessen“.

Dies alles macht es nötig, dass eine Strategie, die neu geladen wird, intern umbenannt werden muss, damit Java die geänderte Klasse laden kann.

Das Beispiel müsste also folgendermaßen geändert werden:

```
.
.
.
Class Strategie = Class.forName("Beispielstrategie ");
.
.
.
// Der Fehler wird bemerkt und geändert
Class Strategie = Class.forName("Beispielstrategie_2");
.
.
.
```

Dadurch wird eine „neue“ Klasse erzeugt und die Änderung übernommen.

# 7 Das Blackboard

In diesem Kapitel werden die Aufgaben und Funktionen des Blackboards beschrieben

## 7.1 Aufgaben des Blackboards

Das Blackboard hat die Aufgabe, Objekte, die von unterschiedlichen Datentypen sein können, zu verwalten. Es muss aber auch gleichzeitig Methoden zur Verfügung stellen damit Objekte, die auf dem Blackboard abgelegt sind, verarbeitet werden können.
Auf das Blackboard kann von verschiedenen Stellen aus zugegriffen werden, eine Beschränkung existiert hierbei nicht. Es kann also durchaus vorkommen, dass mehrere Strategien, die als Threads realisiert wurden, auf dem Blackboard arbeiten.

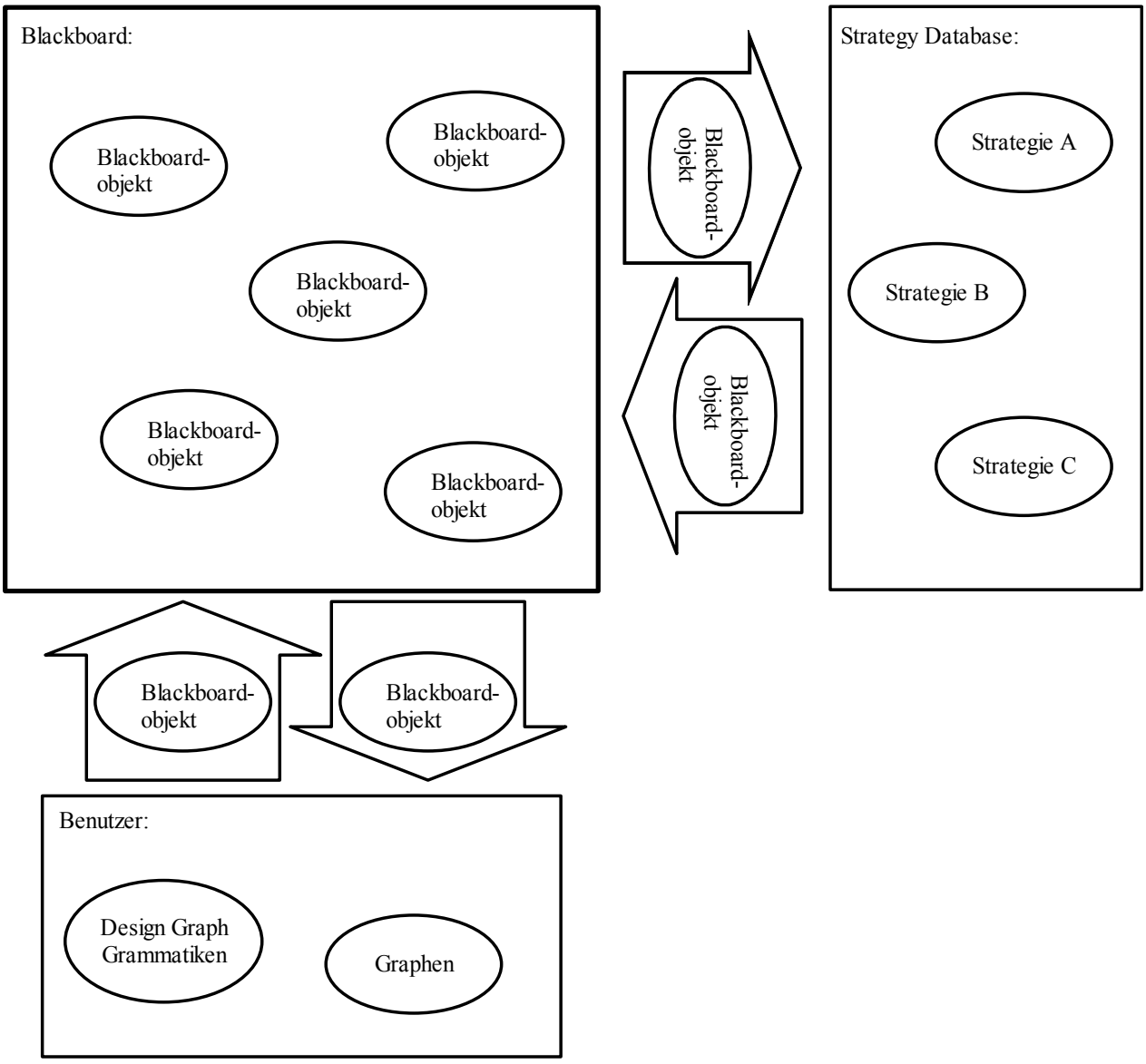



**Abbildung 7.1: Prozesse, die mit dem Blackboard arbeiten**

## 7.2 Methoden des Blackboards

Das Blackboard muss den Prozessen, die auf ihm arbeiten, unterschiedliche Methoden zur Verfügung stellen. Folgende Methoden sind hierbei nötig:

- Blackboard anlegen: Ein Blackboard wird erzeugt und kann von Prozessen benutzt werden. Es existiert nur ein Blackboard für alle Prozesse.

- Objekt auf dem Blackboard ablegen: Ein beliebiges Datenobjekt wird auf dem Blackboard abgelegt und kann dann mit den Methoden des Blackboard bearbeitet werden.

- Datentypen auf dem Blackboard suchen: Da auf dem Blackboard unterschiedliche Objekte mit unterschiedlichen Datentypen gespeichert werden, gibt diese Methode die Möglichkeit, alle Objekte eines bestimmten Datentyps zu suchen. Dadurch wird es möglich, die gefundenen Objekte eindeutig zu identifizieren und somit mit ihnen weiter zu arbeiten.

- Objekt vom Blackboard anfordern: Ein bestimmtes Objekt wird von einer Strategie angefordert. Das Blackboardobjekt wird dann für andere Strategien gesperrt, verbleibt aber auf dem Blackboard und ist für den Benutzer zugänglich.

- Objekt anzeigen: Jedes Blackboardobjekt hat eine Methode um wichtige Daten und Fakten des Objektes auf dem Bildschirm darzustellen.

## 7.3 Prozesse, die mit dem Blackboard arbeiten

Das Blackboard kann von mehreren Strategien gleichzeitig benutzt werden, somit muss sichergestellt werden, dass bei einem Zugriff auf das Blackboard von zwei Strategien kein unvorhersehbarer Zustand entsteht.
Wenn mehr als eine Strategie auf das Blackboard zugreift, entstehen folgende Probleme:

- Zwei Prozesse haben mit demselben Objekt gearbeitet und möchten „ihre“ Arbeit nun auf das Blackboard schreiben, welche Arbeit ist die „aktuelle“ Arbeit.
- Ein Prozess fordert ein Blackboardobjekt an, ein anderer Prozess arbeitet aber gerade mit diesem Objekt, wann ist das Blackboardobjekt bereit gelesen zu werden.
- Ein Prozess arbeitet mit einem Blackboardobjekt und gerät in eine Endlosschleife. Was passiert nun mit dem Blackboardobjekt.
- Usw.

Nur zwei „Prozessarten“ bekommen Zugriff auf das Blackboard:

- Strategien
- Benutzer

Die Strategien benutzen das Blackboard um Informationen auszutauschen und sich mit anderen Strategien zu koordinieren.
Der Benutzer überwacht das Blackboard und hat Zugriff auf alle Objekte des Blackboards.

### 7.3.1 Strategien

Die Aufgabe einer Strategie wurde in Kapitel 6.1 beschrieben. Damit sie diese Aufgabe erfüllen kann, muss sie Informationen erhalten und austauschen können.
Das Blackboard dient der Strategie als eine Art externer Speicher auf dem sie Daten ablegen/anfordern kann. Da es durchaus möglich ist, dass mehrere Strategien arbeiten, muss ein Mechanismus existieren, der den gleichzeitigen Zugriff auf ein und dasselbe Blackboardobjekt verhindert.
Sobald eine Strategie ein Blackboardobjekt anfordert, wird dieses Objekt für alle anderen Strategien gesperrt. Der Zugriff auf ein Objekt ist exklusiv.
Wenn eine Strategie beendet wird, muss sie alle angeforderten Objekte wieder freigeben.
Eine Strategie kann durch den Benutzer beendet werden oder durch das „gewollte Ende“ der Strategie. Falls eine Strategie fehlerhaft arbeitet, zum Beispiel durch eine Endlosschleife, kann der Benutzer die Strategie beenden und die benutzten Blackboardobjekte von Hand freigeben.

### 7.3.2 Benutzer

Der Benutzer kann jederzeit neue Blackboardobjekte auf das Blackboard laden. Dadurch stehen die Objekte den Strategien zur Verfügung. Unter anderem kann er auch Objekte löschen, solange diese nicht gesperrt sind. Der Benutzer kann die Sperre eines Blackboardobjekts auch wieder aufheben. Als letztes hat der Benutzer noch die Möglichkeit sich ein Blackboardobjekt, auf dem Bildschirm, anzeigen zu lassen.
Mit diesen Funktionen hat der Benutzer jederzeit die volle Kontrolle über alle Blackboardobjekte. Diese Kontrolle ist nötig da Strategien, die fehlerhaft arbeiten, Blackboardobjekte sperren können. Somit kann keine andere Strategie diese Objekte nutzen. Auch eine fehlerhafte Strategie die geändert wurde, hätte keine Möglichkeit mehr „ihre“ gesperrten Objekte anzufordern.

## 7.4 Die Aufgabe des Blackboards aus der Sicht einer beliebigen Programmiersprache

Das Blackboard stellt also eine Sammlung von unterschiedlichen Datentypen dar, die mit Hilfe der Blackboardfunktionen verarbeitet werden können.
Gleichzeitig verwaltet das Blackboard aber auch die Objekte, die auf ihm gespeichert werden und stellt sicher, dass nur konsistente Daten verarbeitet werden.

Jedes Objekt in einer Programmiersprache, gehört einem gewissen Typ an und besitzt dadurch verschiedene Eigenschaften, die fest definiert sind.
Dadurch ist es zum Beispiel nicht möglich in ein Array von „Integer“ Werten einen „String“ Wert zu speichern, da es dem Array nur möglich ist, „Integer“ Werte zu speichern.

Das Blackboard benötigt aber eine Datenstruktur, die Objekte unterschiedlichsten Typs verwalten und verarbeiten kann. Es muss auch problemlos möglich sein, neue Objekte unbekannten Typs auf das Blackboard zu schreiben, ohne dass dafür der Programmcode des Blackboards stark geändert werden muss.

Das Ziel ist also eine Datenstruktur, die beliebige Objekttypen verwalten kann und die leicht erweiterbar ist.

### 7.4.1 Die Aufgabe des Blackboards aus der Sicht der Programmiersprache Java

Die Programmiersprache Java ist eine objektorientierte Programmiersprache, das bedeutet unter anderem, dass jedes Objekt zu einer Klasse gehört und dadurch Methoden und Variablen der jeweiligen Klasse besitzt und sie benutzen kann.

Das Blackboard wird also eine Klasse sein, die Methoden zur Verwaltung ihrer Daten zur Verfügung stellt.

Die Daten sind allerdings von unterschiedlichem Typ, das bedeutet, dass keine Standartdatenstruktur (Array, Liste,...) verwendet werden kann, da diese Datenstrukturen nur einen Typ zulassen. Das Blackboard soll aber unterschiedliche Datentypen verwalten.

Damit dieses Problem gelöst werden kann, benötigt man die Techniken der Vererbung und der Typkonvertierung (Cast) von Objekten.

### 7.4.2 Vererbung in Java

Dieses Thema wurde bereits in Kapitel 6.3.1 erläutert.

### 7.4.3 Typkonvertierung/Cast von Objekten in Java

Der Begriff „Cast“ bedeutet, dass ein Objekt eines Typs in einen anderen Objekttyp umgewandelt wird. Dabei verliert das alte Objekt die alten Eigenschaften und gewinnt die Eigenschaften des neuen Objekttyps.
Ein Beispiel wäre das Umwandeln einer „Float“ Zahl in eine „Integer“ Zahl.

```
float f = 2.3;
int i = (int) f;
```

Die Variable “i” enthält nun den Wert 2, der Wert wurde abgerundet, da eine Integer Variable keine Zahlen mit Kommawerten speichern kann.

### 7.4.4 Die Kombination von Vererbung und „Cast“, um das Blackboard zu realisieren

Die Idee beruht auf dem Blackboard. Dieses verwaltet lediglich „Container“ eines bestimmten Typs, in welchem sich die richtigen Typen befinden.

Die Container wären in Abbildung 7.2 das so genannte Blackboardobjekt. Die Typen die verwaltet werden sollen, heißen „Nutzdaten“. Die Blackboardobjekte sind mit einer doppelt verketteten Liste verbunden, somit ist es möglich das Blackboard beliebig zu erweitern.

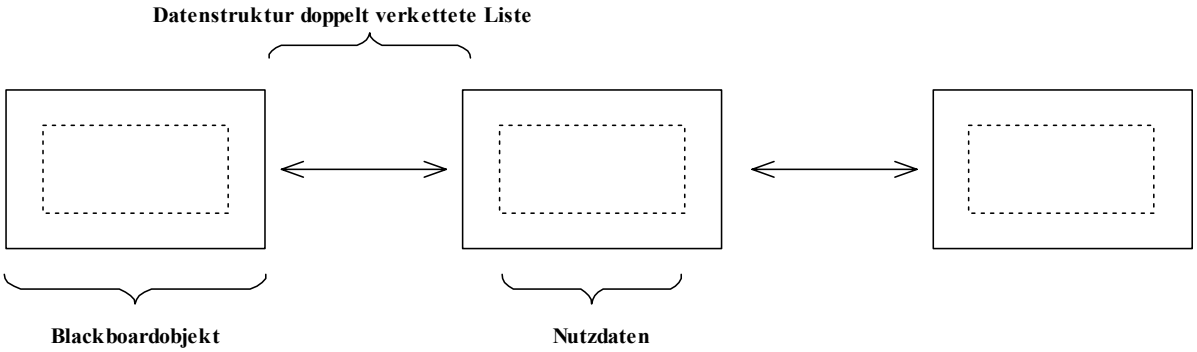



**Abbildung 7.2: Das Blackboard als doppelt verkettete Liste**

Eine „Löschoperation“ bezieht sich folglich auf das Blackboardobjekt und nicht auf die Nutzdaten selber.

Das Blackboardobjekt wäre in Java eine Klasse von der alle Objekte erben, die auf das Blackboard gespeichert werden müssen.
Damit anschließend auf die Nutzdaten des Blackboardobjekts zugegriffen werden kann, muss auf das neue Objekt eine Typkonvertierung (cast) angewendet werden. Erst danach stehen die Nutzdaten zur Verfügung, diese wurden vorher durch den Typ „Blackboardobjekt“ verdeckt.

# 8 Ressourcenorientierte Konfigurierung

Die meisten technischen Systeme, die eingesetzt werden, um Leistung zu erbringen oder Funktionen anzubieten, benötigen Ressourcen, um diese Aufgabe erfüllen zu können.
Ein technisches System seinerseits besteht wiederum aus Komponenten. Diese Komponenten benötigen auch Ressourcen, damit diese ihre Funktionen anbieten können.
Somit werden Ressourcen auf der einen Seite von Komponenten gefordert, auf der anderen Seite werden Ressourcen von Komponenten angeboten.
Ein technisches System ist also nur dann in der Lage seine Leistung zu erbringen oder Funktionen anzubieten, wenn die dafür benötigten Ressourcen vorhanden sind.
Eine Möglichkeit wäre, alle Ressourcenbedürfnisse zu sammeln und solange Komponenten in das technische System einzubinden, bis alle Ressourcenbedürfnisse befriedigt sind.
Diese Methode wird Bilanzverfahren genannt. Eine globale Bilanz überwacht alle Ressourcenbedürfnisse, erst wenn diese globale Bilanz ausgeglichen ist, d.h. alle Bilanzeinträge sind positiv oder null, kann das technische System Leistung erbringen.
Allerdings hat die globale Bilanz einen Nachteil. Einige Ressourcen, die zwar global verfügbar sind, stehen lokal nicht zur Verfügung.

Aus diesem Grund wird in dem folgenden Kapitel das Konzept „Lokale Konzepte in der ressourcenorientierten Konfigurierung“ genauer erläutert.

## 8.1 Lokale Konzepte in der ressourcenorientierten Konfigurierung

Das bisher vorgestellte Konzept der globalen Bilanz weißt einen Fehler auf. Es ist durchaus möglich eine globale Bilanz zu erfüllen, aber lokal steht die Ressource nicht zur Verfügung.
Ein Beispiel hierfür wäre der Buchdruck. Wenn 100 Bücher mit 100 Seiten erstellt werden sollen, dann reicht es nicht aus 10000 Seiten zur Verfügung zu stellen. Die globale Bilanz wäre sicherlich ausgeglichen, aber jedes Buch muss Seite 1 bis 100 aufweisen, damit es ein lesbares Buch darstellt. Der globalen Bilanz reicht dabei 10000-mal Seite 1.

Der strukturelle Aspekt wird nicht berücksichtigt, Heinrich [6] schlägt vor, jeder Komponente eine lokale Bilanz zur Verfügung zu stellen. Erst wenn alle Bilanzen ausgeglichen sind, ist auch das Ursprungsproblem gelöst.

In diesem erweiterten Konzept wird zwischen globalen und lokalen Ressourcen unterschieden. Eine globale Ressource ist im ganzen System wirksam, die Komponenten müssen nicht miteinander in Verbindung stehen, um sie nutzen zu können. Als lokale Ressource wird eine Verbindung zwischen zwei Komponenten bezeichnet, diese Ressource ist nicht von allen Komponenten zugreifbar. Jede Komponente verfügt hierbei über eine eigene lokale Bilanz ihrer Ressourcen. Da jede Komponente über eine eigene Bilanz verfügt und das technische System aus Komponenten besteht, kann eine hierarchische Struktur aufgebaut werden, welche als „Bilanzbaum“ bezeichnet wird. Dieser Baum bildet sofort die kompositionelle Struktur des Systems ab.

Die Wurzel stellt dabei die globale Bilanz da, an ihr werden alle lokalen Bilanzen angehängt.
Die lokalen Ressourcen finden sich in der lokalen Bilanz wieder, sie sind für andere Komponenten nicht zu erreichen. Die globalen Ressourcen der Komponente werden an die nächst höhere Ebene weiter gereicht und dort verrechnet. Dies wird so lange fortgesetzt, bis die Wurzel erreicht wird. Wird die Bilanz der Wurzel ebenfalls ausgeglichen ist das Problem gelöst.

Es seien Folgende Komponenten gegeben:
Komponente C bietet global Ressource 1,2,3 an, lokal verfügt sie über Ressource 4
Komponente B bietet global Ressource 2,3 an, lokal verfügt sie über Ressource 1
Komponente A bietet global Ressource 3 an, lokal verfügt sie über Ressource 2

Somit entsteht folgender Bilanzbaum:

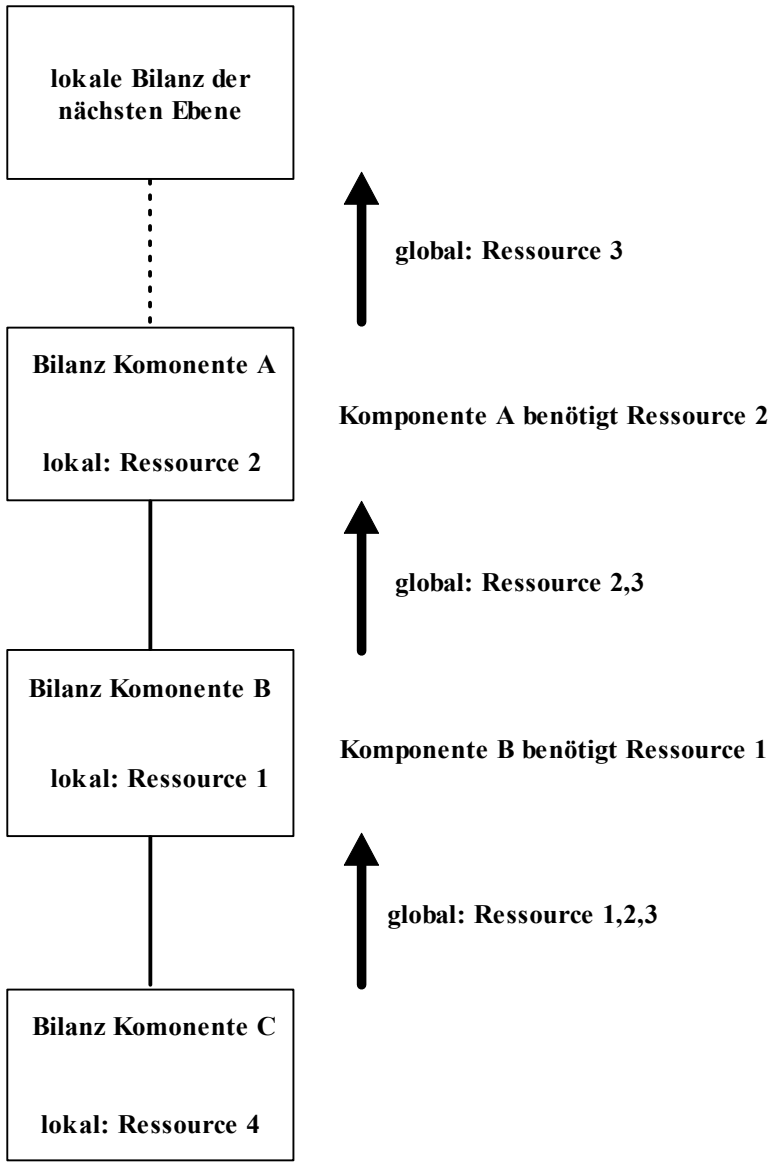



**Abbildung 8.1: Bilanzbaum mit Teilbilanzen der einzelnen Komponenten**

# 9 Anwendungsbeispiel

In diesem Kapitel soll an einem praktischen Beispiel gezeigt werden, wie das Design/Konfigurationssystem arbeitet. Als eine Art Startkonfiguration sei eine leere Fabrikhalle gegeben. Die Zielkonfiguration ist eine mit Maschinen gefüllte Fabrikhalle, die versandfertige Kettenräder herstellt. Damit eine gültige Endkonfiguration erreicht werden kann, müssen mehrere Schritte durchlaufen werden. Für die richtige Abarbeitung der Schritte ist die Strategie zuständig.

Der erste Schritt wäre die Darstellung des Arbeitsprozesses, dies geschieht mit Hilfe der in Kapitel 5.2 beschriebenen Design-Graphgrammatik. Hierbei wird der nötige Arbeitsprozess aus den Design-Graphgrammatik Regeln hergeleitet.
Die Design-Graphgrammatik erzeugt dabei alle möglichen Ergebnisse, wobei viele davon nicht zu gebrauchen sein werden. Damit die brauchbaren Lösungen von den nicht brauchbaren Lösungen unterschieden werden können, muss eine Simulation (Schritt 2) die Lösung der Design-Graphgrammatik testen. Erst wenn die Simulation erfolgreich verläuft, wurde ein Arbeitsprozess gefunden.
Aufgrund des hergeleiteten Arbeitprozesses sind die benötigten Maschinentypen bekannt, diese müssen im nächsten Schritt (Schritt3) instanziiert werden.
Nachdem Schritt 3 ausgeführt wurde stehen konkrete Maschinen bereit um das geforderte Werkstück zu fertigen. Die gerade getroffene Maschinenauswahl muss dann in die Struktur, die im ersten Schritt erzeugt wurde, platziert werden (Schritt 4). Dabei kann sofort geprüft werden, ob die gewählten Maschinen die geforderten Restriktionen erfüllen (herstellbare Menge je Stunde, benötigtes Personal, usw.).
Als nächstes müssen die real existierenden Maschinen in der Fabrikhalle aufgestellt werden. Dabei ist das Hauptziel, die Transportwege zwischen den einzelnen Maschinen so kurz wie möglich zu halten (Schritt 5).
Erst nachdem Schritt 5 beendet wurde ist die Design-Aufgabe gelöst und dem Auftraggeber kann eine Lösung des Problems geliefert werden. In Abbildung 9.1 ist die Idee, wie das Problem zu lösen ist, dargestellt. Die Abbildung ist etwas vereinfacht um das Grundprinzip zu verdeutlichen. Wenn zum Beispiel in Schritt 3 Maschinen ausgewählt wurden, diese aber in Schritt 4 nicht platziert werden können, dann wird nicht sofort ein neuer Arbeitsablauf angefordert, sondern es werden so lange weitere Lösungen angefordert, bis Schritt 3 keine Lösungen mehr erstellen kann. Erst dann wird der nächste Arbeitsablauf erstellt.

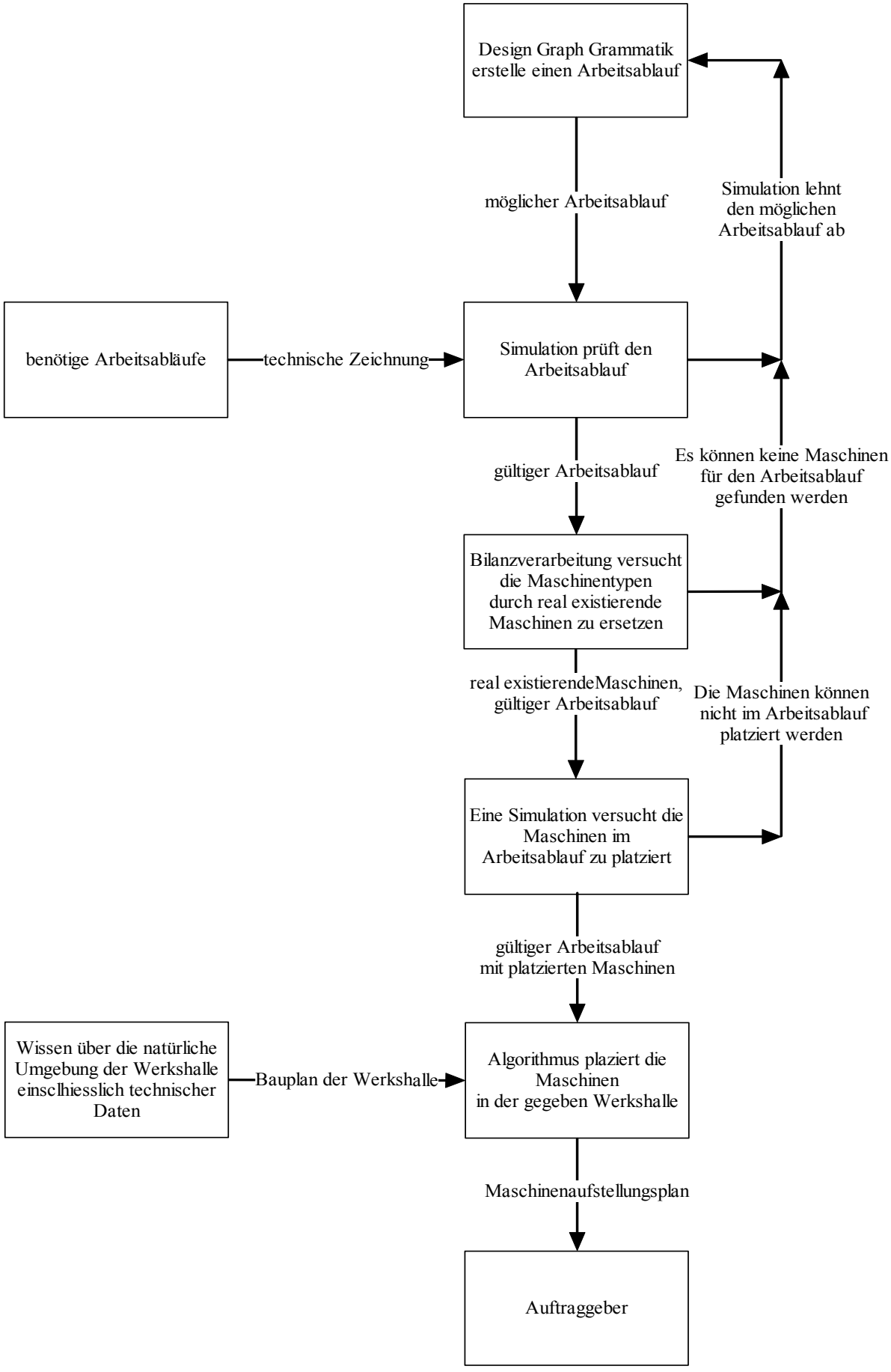



**Abbildung 9.1: Vom Problem zur Lösung, die Strategie**

Als nächstes soll der Produktionsprozess genauer betrachtet werden, aus dieser Betrachtung soll die Design-Graphgrammatik entstehen.

Der Produktionsprozess wird in diesem Beispiel mehrstufig sein [10], somit muss ein Produkt mehrere Produktionsanlagen durchlaufen bis ein fertiges Endprodukt entstanden ist.

Ein weiteres Kriterium ist die Produktionsmenge, für dieses Beispiel wird die „Serienproduktion“ gewählt. Laut Definition der Serienproduktion [10] wird dabei eine größere Menge von Produkteinheiten einer oder mehrerer Produktarten innerhalb einer Planungsperiode gefertigt.

Die Anordnung der Maschinen wird hierbei von der Entwicklungsumgebung festgelegt.

Der Vorteil der Serienproduktion liegt bei einer hohen Flexibilität mit gleichzeitiger Spezialisierung der Arbeitsabläufe.

## 9.1 Der reale Betrieb

Bisher wurde nur theoretisch über die Design-Aufgabe und über den Produktionsprozess, sowie die zu erstellenden Produkte geredet. In diesem Kapitel soll genauer beschrieben werden was die Beispielfirma erzeugen soll und wie dies geschieht.

Die Produktionsstätte in diesem Beispiel ist ein Metall verarbeitender Betrieb. Die Belegschaft besteht aus etwa ca. 30 Leuten. Die Endprodukte sind Zahn- und Kettenräder, die aus allen geeigneten Materialien (Eisen, Aluminium, Kunststoffe, usw.) bestehen.
Die meisten Menschen werden ein Zahnrad vom Fahrrad her kennen. Diese Zahn-/Kettenräder gibt es in unterschiedlichen Größen, mit unterschiedlicher Zahnzahl und mit Bohrungen.
Die Zahn-/Kettenräder, die gefertigt werden, unterliegen strengen Auflagen bezüglich ihrer Ausmaße. Die Toleranz befindet sich fast immer im tausendstel Millimeter Bereich. Wird gegen eine fest vorgegebene Toleranzgrenze verstoßen, muss das gefertigte Teilprodukt entsorgt werden, egal in welchem Schritt des Arbeitsprozesses sich das Teilprodukt befindet.

Die Beispielfirma, die hier beschrieben werden soll, ist ein typisches Beispiel für die Serienproduktion. Es werden zwar auch größere Produktserien hergestellt, Serien von über 2000 Teilen kommen durchaus vor, aber größtenteils sind die Serien unter 100 Endprodukten.
Auch die Art der zu fertigenden Endprodukte wechselt ständig, somit ist ein hohes Maß an Flexibilität Pflicht.

Da nicht davon auszugehen ist, dass jedem Menschen bekannt ist, wie ein Zahn-/Kettenrad produziert wird, soll im Folgenden kurz der Arbeitsablauf beschrieben werden.

Der erste Schritt ist die Auftragsbeschaffung, hierbei äußert der Kunde seine Wünsche und die Bestellmenge. Die Kundenwünsche werden mit Hilfe einer technischen Zeichnung dargestellt, auf dieser Zeichnung sind alle Informationen vorhanden die benötigt werden, um das Endprodukt herzustellen. Die typischen Informationen, die sich auf der Zeichnung befinden sind:

- Materialart
- Menge
- Dimensionen des Werkstückes
- Einzuhaltende Toleranzen

- Fertigungstermin
- Sonderwünsche (Bsp.: spezielle Legierungen die nach dem Fertigungsprozess aufgebracht werden sollen, spezielle Kennzeichnungen des Endproduktes (Seriennummer, Firmenlogo, usw.))

Der zweite Schritt ist die Beschaffung des Rohmaterials, dies kann Eisen, Stahl, Aluminium oder Kunststoff sein. Das Rohmaterial befindet sich in einem Lager und ist in langen Stangen mit unterschiedlichen Durchmessern vorhanden.

**Abbildung 9.2a: Das Rohmaterial ist in Stangen mit 5 Meter Länge im Lager vorhanden. Der Durchmesser der Stangen ist unterschiedlich.**

Wenn nun ein Auftrag 100 Zahnräder aus Stahl mit einem Durchmesser von 10 cm und einer Breite von 8 cm erfordert, wird aus dem Lager eine Stahlstange mit einem Durchmesser von mindestens 10 cm angefordert. Diese Stahlstange wird dann mit einer extra dafür angefertigten Säge zerkleinert und es werden 100 Rohlinge mit einer Breite von mindestens 8 cm hergestellt. Ein Rohling ist hierbei eine runde „Stahlscheibe“, die sowohl den richtigen Durchmesser als auch die richtige Breite hat.

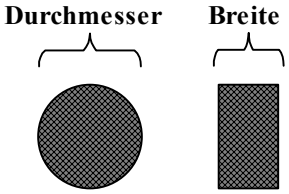



**Abbildung 9.2b: Ein Rohling**

Nachdem nun 100 Rohlinge erzeugt wurden, müssen diese zu einer Maschine transportiert werden, jeder Rohling wiegt dabei etwa 500 gr.

Die Rohlinge werden vor der Maschine gelagert, es lagern mehrere Aufträge vor einer Maschine. Sobald der Auftrag an der Reihe ist, wird die Maschine darauf programmiert den Rohling auf die richtige Größe zu bringen. Alle Informationen, die für die Programmierung der Maschine nötig sind, befinden sich auf einer technischen Zeichnung, die dem Auftrag immer beiliegt.

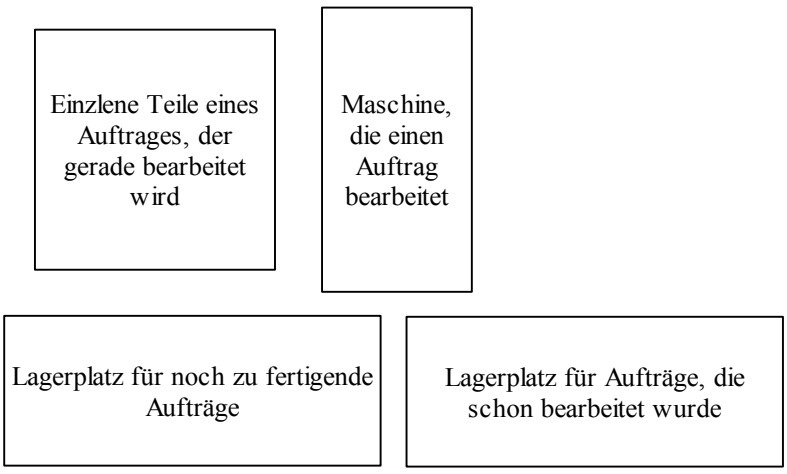



**Abbildung 9.3: Lagerung von Rohlingen und Werkstücken an einer Maschine**

Für die Bearbeitung wird der Rohling in die Maschine eingelegt und sehr schnell gedreht, ein Elektromotor erledigt diese Aufgabe. An den rotierenden Rohling werden nun die Werkzeuge der Maschine herangeführt. Da der Rohling sich mit ca. 2000-3000 Umdrehungen in der Minute dreht, entsteht bei der Bearbeitung Reibungshitze. Daher ist eine durchgehende Kühlung des Rohlings und der Werkzeuge unbedingt notwendig, da sonst der Rohling und die Werkzeuge beschädigt werden.
Die Werkzeuge entfernen hierbei überschüssiges Material und bringen den Rohling in die gewünschte Form und auf die geforderten Maße. Bei diesem Vorgang fällt das Material, das entfernt wurde, auf ein Fliessband, welches sich in der Maschine befindet. Dieses Fliessband fördert den Materialabfall (Späne) in einen Container, der an der Maschine steht.
Für kleine Serien sind Maschinen vorhanden, die nicht computergesteuert sind. Diese werden von Hand bedient, allerdings werden auch diese Maschinen mit Strom betrieben und eine Kühlung mit Kühlwasser ist unbedingt notwendig.

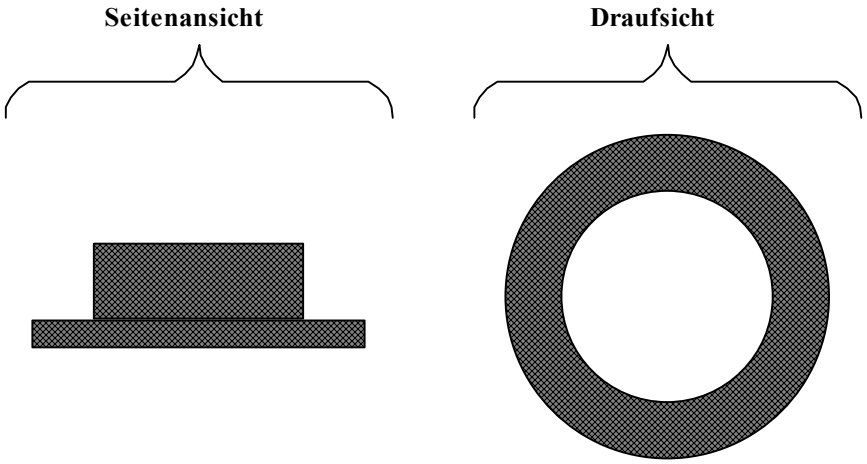



**Abbildung 9.4: Rohling nach der Bearbeitung mit einer Drehbank**

Der beschriebene Arbeitsschritt wird für alle Produkte vollzogen. Erst danach findet eine Trennung statt.

Nachdem der Rohling durch die Drehbank in die geforderte Form gebracht wurde, wird der Rohling als Werkstück bezeichnet. Mit diesem Werkstück kann es nun zu unterschiedlichen Folgeprozessen kommen.

Ein möglicher Folgeprozess wäre das Bohren von Löchern. Dabei werden an fest vorgebenden Stellen Löcher in das Werkstück gebohrt, diese dienen später als Befestigung.
Auch diese Arbeit wird von einer Maschine erledigt, die man als Bohrzentrum bezeichnet. Die Maschinen sind dabei wieder computergesteuert, diesmal dreht sich das zu bearbeitende Teil nicht selber, sondern der Bohrer dreht sich. Dabei spielt die Kühlung wieder eine große Rolle. Es ist absolut erforderlich, das der Bohrer beim Bohren mit Wasser gekühlt wird, da er sonst abbricht und das Werkstück beschädigt. Auch diese Maschinen werden mit Strom betrieben.
Der Abfall, der durch den Prozess entsteht, wird in der Maschine gesammelt und anschließend von Hand in einen Container geworfen. Die dabei entstehenden Abfallmengen sind aber gering. Wie auch bei der Drehbank sind alle relevanten Daten auf der technischen Zeichnung zu finden, die dem Auftrag beiliegt.

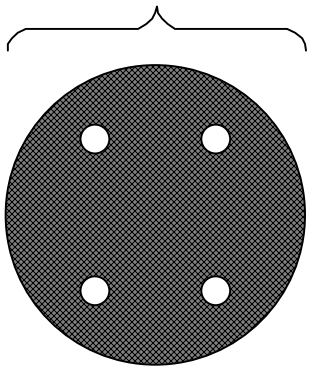



**Abbildung 9.5: Werkstück mit Bohrungen, die durch ein Bohrzentrum erzeugt wurden**

Ein weiterer Folgeprozess wäre das Erstellen der „Zähne“. Bisher ist das Werkstück ein kreisförmiges Gebilde. Auch dieser Vorgang wird teilweise durch computergesteuerte Maschinen durchgeführt. Die Position der Zähne und deren Anzahl, sowie die Zahntiefe kann der technischen Zeichnung entnommen werden.
Die Werkstücke werden in die Maschine eingelegt, danach wird eine rotierende Walze (Fräser) an dem Werkstück vorbeigefahren. Durch die Rotation des Fräsers wird eine halb Runde Öffnung erzeugt, dies wird so oft wiederholt, bis das Werkstück die geforderte Anzahl an Zähnen besitzt. Auch diese Maschinen werden mit Strombetrieben, die Kühlung wird durch spezielle Öle gewährleistet. Der Abfall, der entsteht, sammelt sich in der Maschine und muss von Hand entnommen werden.

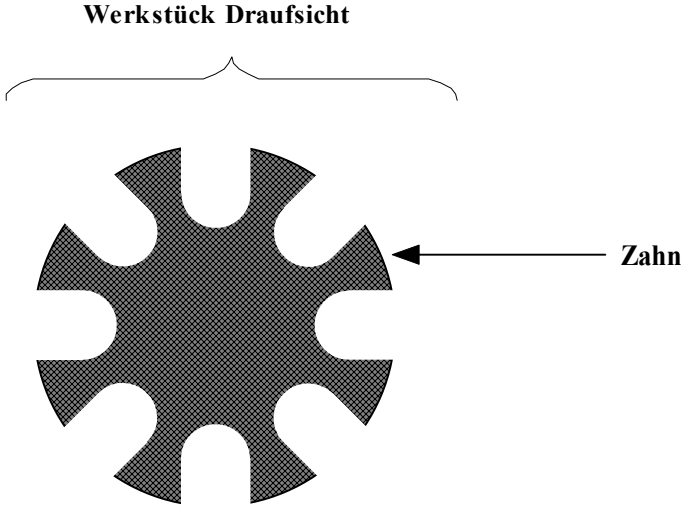



**Abbildung 9.6: Eine Fräse hat Zähne in das Werkstück gefräst**

Ein Folgeprozess, der am Ende der Fertigung steht, ist das Einbringen einer Nut in das Werkstück. Ein langer Metallstab wird durch das Werkstück geführt und eine Einbuchtung (Nut) entsteht. Dieser Arbeitsgang wird maschinell aber ohne Computerunterstützung vorgenommen. Auch in diesem Arbeitsschritt dient die technische Zeichnung als Grundlage.
Die zuständige Maschine wird mit Strom betrieben. Auch bei diesem Arbeitsschritt ist eine Kühlung mit Spezialölen nötig, da sonst das Werkzeug, das die Nut erzeugt, Schaden nehmen würde. Der entstehende Abfall fällt dabei in die Maschine und wird von Hand entsorgt, die Abfallmenge ist gering.

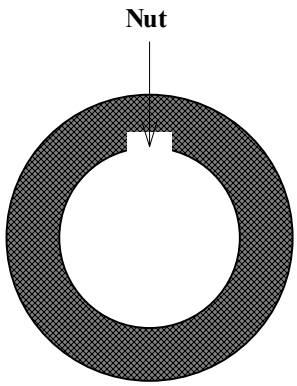



**Abbildung 9.7: Werkstück nachdem eine Nutenbank eine Nut eingebracht hat**

Nachdem diese Arbeitsschritte erledigt wurden, findet eine Endkontrolle statt.
Hierbei wird überprüft ob das fertige Endprodukt den Spezifizierungen des Kunden entspricht, ist dies der Fall, werden die Endprodukte gereinigt und verpackt, um dann zum Kunden geschickt zu werden.

Wie schon angedeutet ist eine durchgängige Kühlung bei allen Arbeitsgängen erforderlich. Es ist nicht tolerierbar, dass die Kühlung durch Wasser, spezielle Öle, Wasser/Öl-Gemische auch nur für Sekundenbruchteile unterbrochen wird. Tritt dieser Fall dennoch ein kann ein hoher Schaden entstehen. Fast alle Werkzeuge sind Spezialanfertigungen und kosten mehrere tausend Euro. Die Gefahr, dass die Maschine beschädigt wird, ist auch sehr groß.
Ein Werkstück, das sich mit 3000 Umdrehungen dreht und 5 kg wiegt und in dem ein Bohrer abbricht, kann große Schäden hervorrufen.

Ein weiteres Problem stellt die Abfallbeseitigung dar. Die anfallenden Metallspäne werden in großen Containern gesammelt. Dieser „Abfall“ kann noch verkauft werden, da er wiederverwertet werden kann. Natürlich ist ein Container mit reinen Materialien mehr Wert, als ein Container, in dem alle Materialsorten gesammelt wurden.
Es ist also durchaus üblich, Aluminiumspäne gesondert aufzufangen, damit diese dann anschließend verkauft werden können.

Jede Maschine wird mit Strom betrieben, also muss auch ein entsprechender Stromanschluss vorhanden sein. Dabei reicht ein 220 Volt Anschluss natürlich nicht aus, ein Starkstromnetz muss vorhanden sein, zu dem jede Maschine Zugang hat.
Natürlich hat jede Maschine einen anderen Stromverbrauch, der gedeckt werden muss, damit die Maschine einsatzfähig bleibt.

In der Beschreibung der Arbeitsabläufe wurde es zwar nicht explizit erwähnt, aber an jedem Arbeitsplatz ist ein Schlauch mit Druckluft vorhanden. Mit diesem können Metallspäne entfernt werden, manchmal wird auch ein Bohrer, der mit Druckluft angetrieben wird, benötigt, um kleinere Schrauben in das Werkstück zu drehen.

Da die Arbeitsprozesse alle sehr unterschiedlich sind, gibt es keine Maschine, die alle Arbeitsschritte auf einmal durchführen kann. Für jeden Arbeitsschritt wird eine Maschine eines bestimmten Typs benötigt. Der Arbeitsschritt „Drehen“ benötigt zum Beispiel eine Drehbank. Dabei benötigen die unterschiedlichen Maschinen alle unterschiedlichen Platz damit sie aufgestellt werden können. Dabei ist darauf zu achten, dass ein Materialfluss zwischen den Maschinen möglich ist. Ein weiterer Punkt sind Sicherheitsabstände zwischen den Maschinen, die eingehalten werden müssen.

Alle Maschinen benötigen einen Menschen für die Bedienung. Sei es nun um die Maschine auf einen Auftrag vorzubereiten (Einrichten) oder um den Arbeitsprozess zu überwachen. Jeder Beschäftigte hat dabei eine Qualifikation und bestimmte Maschinen benötigen eine bestimmte Qualifikation damit diese bedient werden kann.

Wenn nun eine Maschine einzeln betrachtet wird, dann benötigt diese zum fehlerfreien bearbeiten eines Werkstückes folgende Dinge:

- Strom (gemessen in Ampere)
- Personal
- Stellplatz (gemessen in $m^2$)
- Kühlmittel (gemessen in Liter pro Stunde)
- Abfallentsorgung (gemessen in kg je Stunde)
- Druckluft (gemessen in $m^3$ je Stunde)
- Maximale/Minimale Größe des zu bearbeitenden Werkstücks

## 9.2 Eine Design-Graphgrammatik

Der erste Schritt, der nötig ist damit die Design-Aufgabe gelöst werden kann, ist die Erstellung einer Design-Graphgrammatik, die den Arbeitprozess abbilden kann.
Sobald der Arbeitsprozess abgebildet wurde, kann daraus gefolgert werden, welche Maschinentypen nötig sind, um den Arbeitsprozess zu ermöglichen. Diese Maschinentypen können im zweiten Schritt dann instanziiert werden.

### 9.2.1 Startsymbol

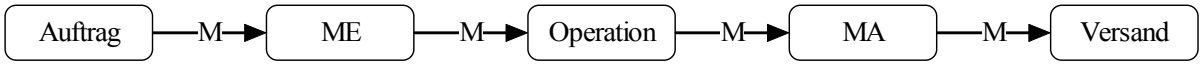



**Abbildung 9.8: Startsymbol**

In Abbildung 9.8 ist das Startsymbol der Design-Graphgrammatik zu sehen. Ein Auftrag löst einen Materialfluss zu einem Objekt aus, dieses Objekt verwendet den Materialfluss um eine unbekannte Aktion (Operation) damit auszuführen. Danach entsteht wieder ein Materialfluss, der an die Abteilung „Versand“ weiter geleitet wird. Die Abteilung „Versand“ verarbeitet den Materialfluss und schickt das fertige Werkstück an den Auftraggeber. Der Auftrag ist somit abgeschlossen.

### 9.2.2 Regel 1: Arbeitsabläufe seriell darstellen

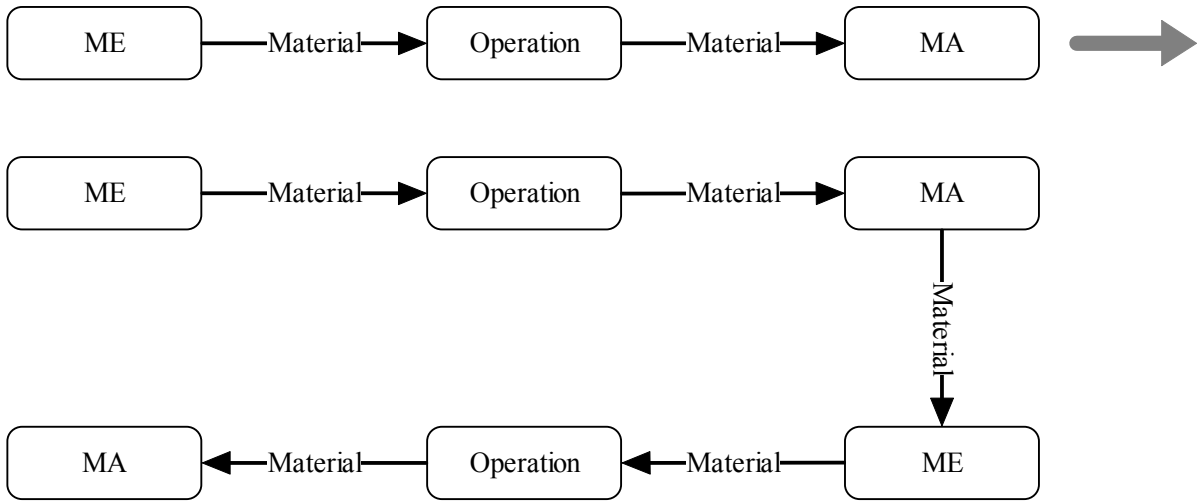



**Abbildung 9.9: Regel 1, serielle Bearbeitung von Arbeitsprozessen**

In Abbildung 9.9 ist eine Design-Graphgrammatik Regel angegeben, die es erlaubt aus einer Operation zwei Operationen zu machen. Dadurch ist es möglich mehrere Schritte nacheinander abzuarbeiten. Für diese Regel fehlen noch die Einbettungsregeln. Diese definieren, was mit den Kanten des Host-Graphen geschieht, die den Zielgraph Graphen mit dem Host-Graphen verbinden. Im Folgenden soll die Entstehung der Einbettungsregeln für diese Regel erläutert werden.

Zielgraph $T = <V_T,E_T,o_T> = <\{1,2,3\},\{(1,2),(2,3)\},\{(1,ME),(2,Operation),(3,MA)\}>$
Ersetzungsgraph $R = <V_R,E_R,o_R> = <\{4,5,6,7,8,9\},\{(4,5),(5,6),(6,7),(7,8),(8,9)\},$
$\{(4,ME),(5,Operation),(6,MA),(7,ME),(8,Operation),(9,MA)\} >$

Der Zielgraph T ist die „linke Seite“ der Graph Grammatik Regel in Abbildung 5.3.
Der Ersetzungsgraph R stellt die „rechte Seite“ der Graph Grammatik Regel in Abbildung 5.3 dar.
Als nächstes werden die Einbettungsregeln dargestellt und erläutert.

**Einbettungsregeln E = {((Auftrag, ME, Material), (Auftrag, ME, Material)),**
**(Versand, MA, Material), (Versand, MA, Material)}**

Die erste Einbettungsregel ((Auftrag, ME, Material), (Auftrag, ME, Material)) soll mögliche Kanten, die von dem Knoten „Auftrag“, der sich im Host-Graphen befindet, auf den neuen Knoten „ME“, der sich im Ersetzungsgraph befindet, umleiten.

Die zweite Einbettungsregel ((Versand, MA, Material),( Versand, MA, Material)) soll mögliche Kanten, die von dem Knoten „Versand“, der sich im Host-Graphen befindet, auf den neuen Knoten „ME“, der sich im Ersetzungsgraph befindet, umleiten.
In Abbildung 9.10 ist der Host-Graph vor der Anwendung der ersten Regel dargestellt, sowohl der Zielgraph, der gelöscht wird, wie auch der Host-Graph sind gekennzeichnet.

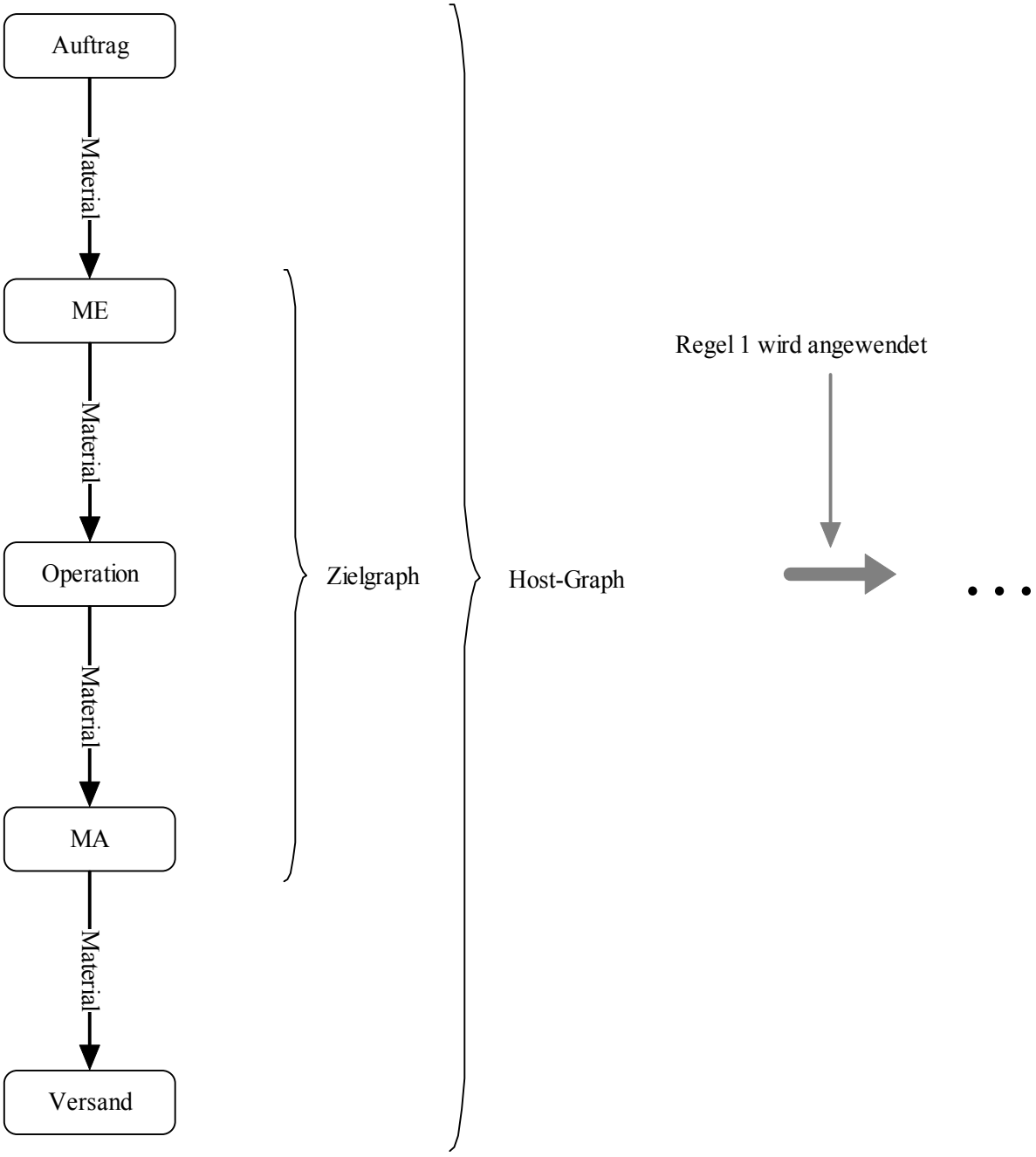
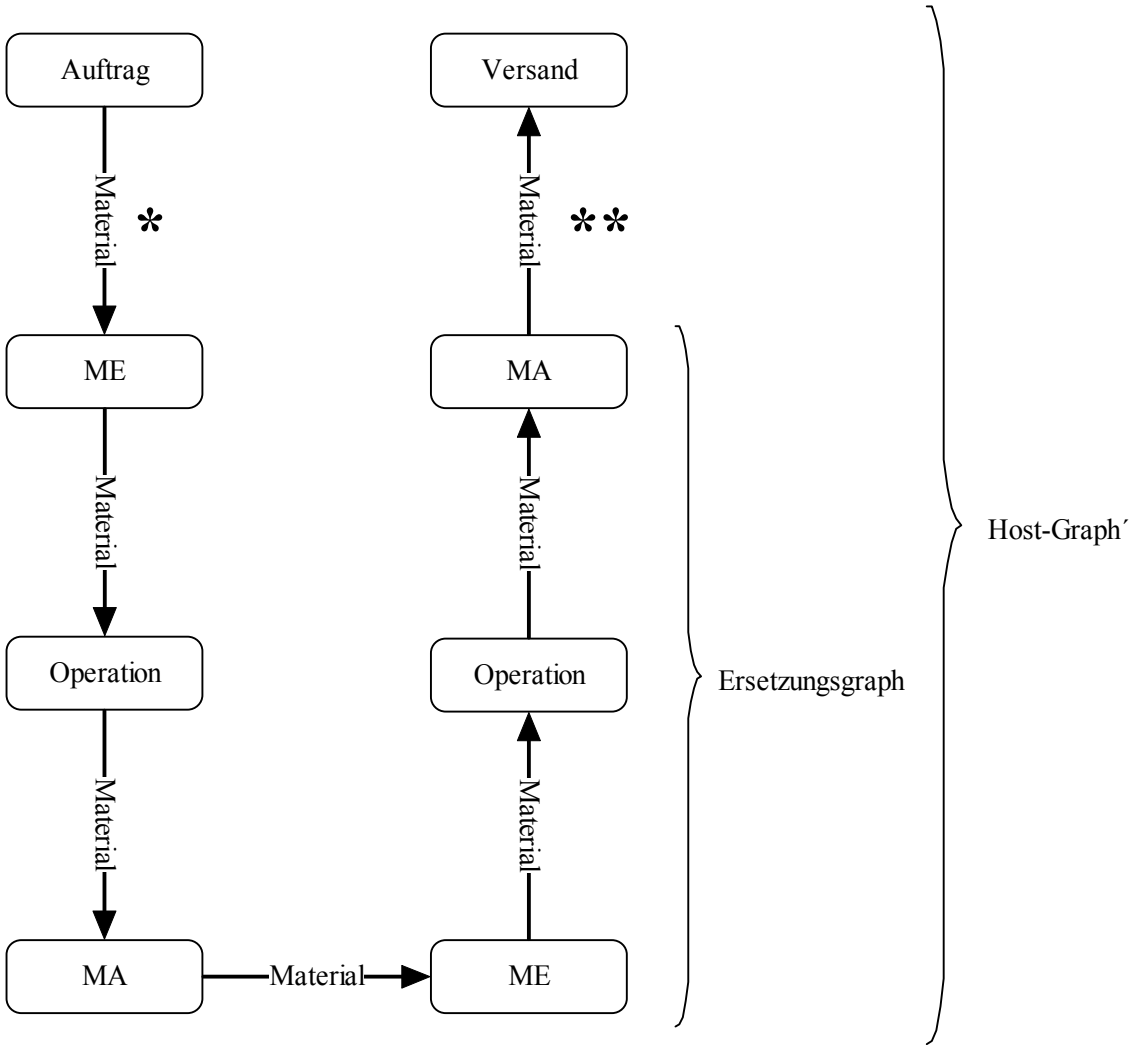



**Abbildung 9.10: Regel 1 wird auf einen Host-Graphen angewendet**




* = Diese Kante ist durch die erste Einbettungsregel enstanden
** = Diese Kante ist durch die zweite Einbettungsregel enstanden

**Abbildung 9.11: Der neue Host-Graph nach Anwendung der ersten Regel**

In Abbildung 9.11 ist der entstandene Host-Graph dargestellt. Als erstes wurde der Zielgraph gelöscht und durch den Ersetzungsgraph ersetzt, danach wurde die Kanten *, ** mit Hilfe den eben beschriebenen Einbettungsregeln eingefügt.

Im Nachfolgenden sind alle Einbettungsregeln, die zu Regel 1, gehören aufgezählt.

**Einbettungsregeln E ={**
**((Auftrag, ME, Material),(Auftrag, ME, Material)),**
**((MA,ME,Material),(MA,ME,Material)),**

**((Versand, MA, Material),(Versand, MA, Material)),**
**((ME, MA, Material),(ME, MA, Material))}**

### 9.2.3 Regel 2: Arbeitsabläufe parallel darstellen

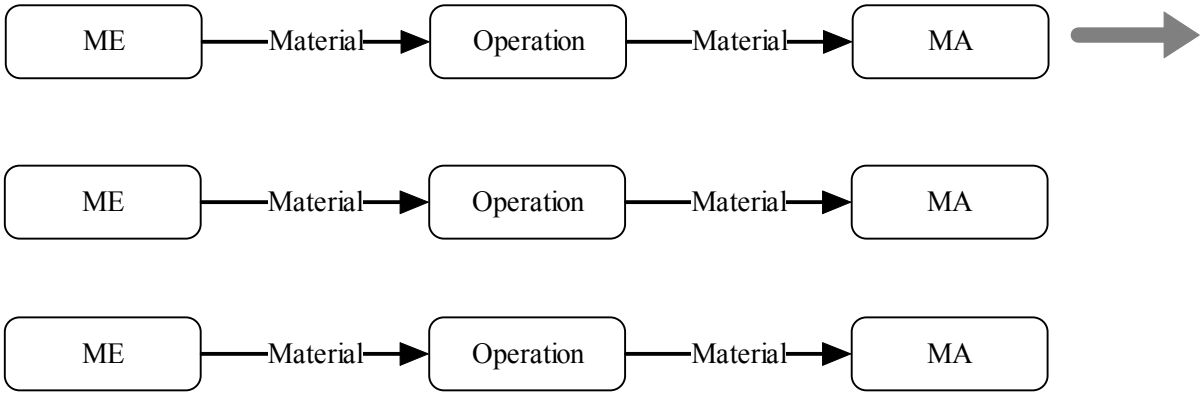



**Abbildung 9.12: Regel 2, parallele Bearbeitung von Arbeitsprozessen**

In Abbildung 9.12 wird eine Regel dargestellt, die es erlaubt einen Arbeitsschritt parallel zu bearbeiten. Dadurch entsteht die Möglichkeit einen Arbeitsschritt durch zwei identische Maschinentypen ausführen zu lassen oder aber einen identischen/alternativen Arbeitsablauf zu erzeugen.
Dies hat den Vorteil, dass ein Auftrag schneller abgearbeitet werden kann, da zwei/mehrere Maschinentypen an einem Auftrag arbeiten. Sobald der Auftrag auf allen Maschinentypen abgearbeitet wurde, wird das Material der parallelen Arbeitsprozesse zusammen geführt.
Im Folgenden wird die Regel und ihre Einbettungsregeln beschrieben.

Zielgraph $T = <V_T,E_T,o_T> = <\{1,2,3\},\{(1,2),(2,3)\},\{(1,ME),(2,Operation),(3,MA)\}>$
Ersetzungsgraph $R = <V_R,E_R,o_R> = <\{4,5,6,7,8,9\},\{(4,5),(5,6),(7,8),(8,9)\},$
$\{(4,ME),(5,Operation),(6,MA),(7,ME),(8,Operation),(9,MA)\} >$

**Einbettungsregeln E ={**
**((Auftrag, ME, Material),(Auftrag, ME, Material)),**
**((Auftrag, ME, Material),(Auftrag, ME, Material)),**
**((MA, ME, Material),(MA, ME, Material)),**
**((MA, ME, Material),(MA, ME, Material)),**

**((Versand, MA, Material),(Versand, MA, Material)),**
**((Versand, MA, Material),(Versand, MA, Material)),**
**((ME, MA, Material),(ME, MA, Material)),**
**((ME, MA, Material),(ME, MA, Material))}**

Jede Einbettungsregel ist zweifach vorhanden. Der Grund dafür ist, dass der Algorithmus, der die Regeln abarbeitet, eine Regel doppelt benötigt, wenn aus einer Kante zwei neue Kanten entstehen sollen.
Der Algorithmus betrachtet jede Kante, die einen Knoten aus dem Host-Graphen mit einem Knoten aus dem Zielgraphen Graphen verbindet. Es wird dann in der Einbettungsregelmenge nach einer Regel gesucht, die für diese spezielle Kante angewendet werden kann. Wurde eine Regel gefunden wird eine neue Kante eingefügt, danach wird weiter überprüft, ob noch weitere Regeln für diese Kante anwendbar sind.
Aus diesem Grund muss jede Regel doppelt vorhanden sein, Abbildung 9.13 und 9.14 soll dies verdeutlichen.

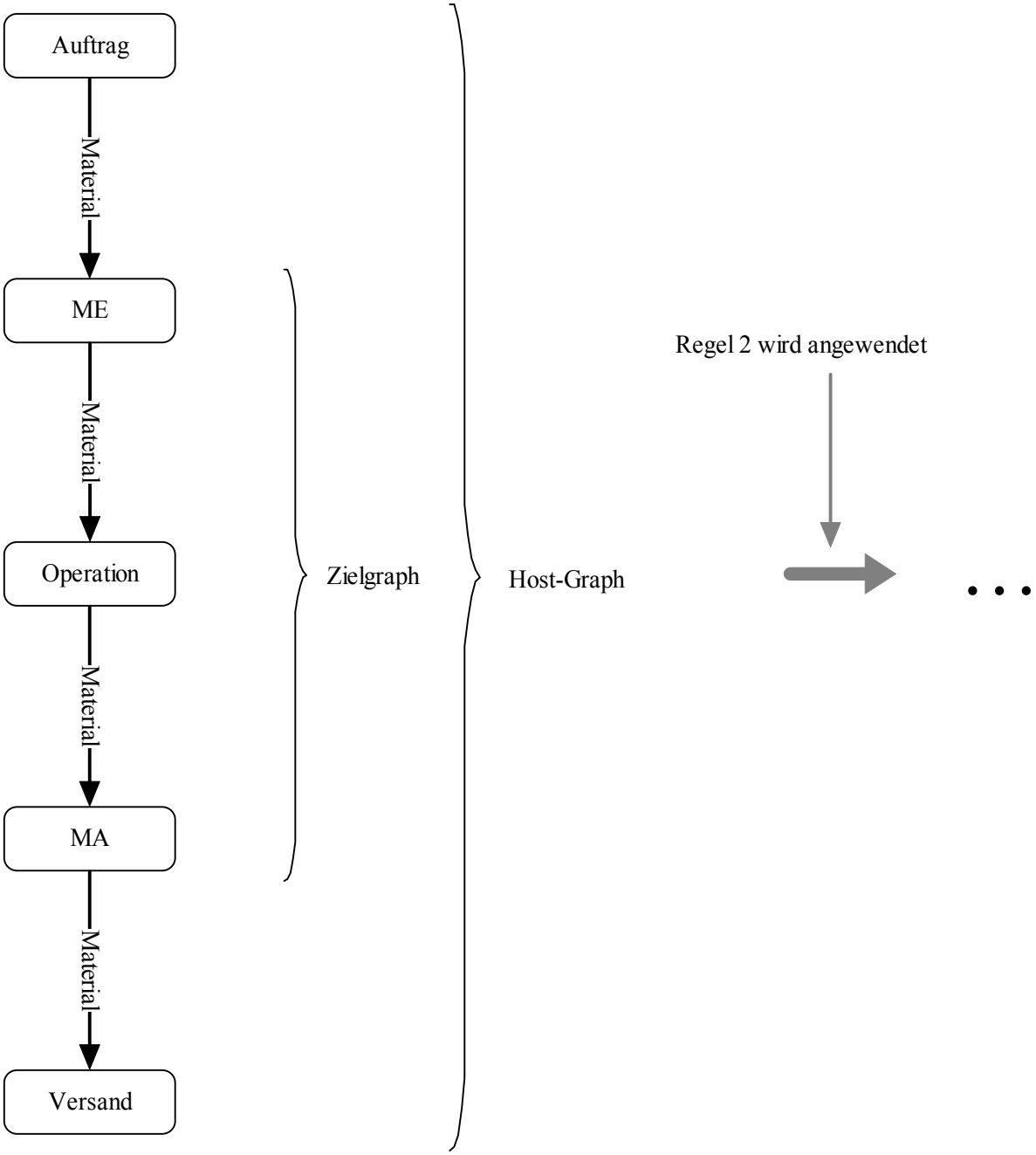
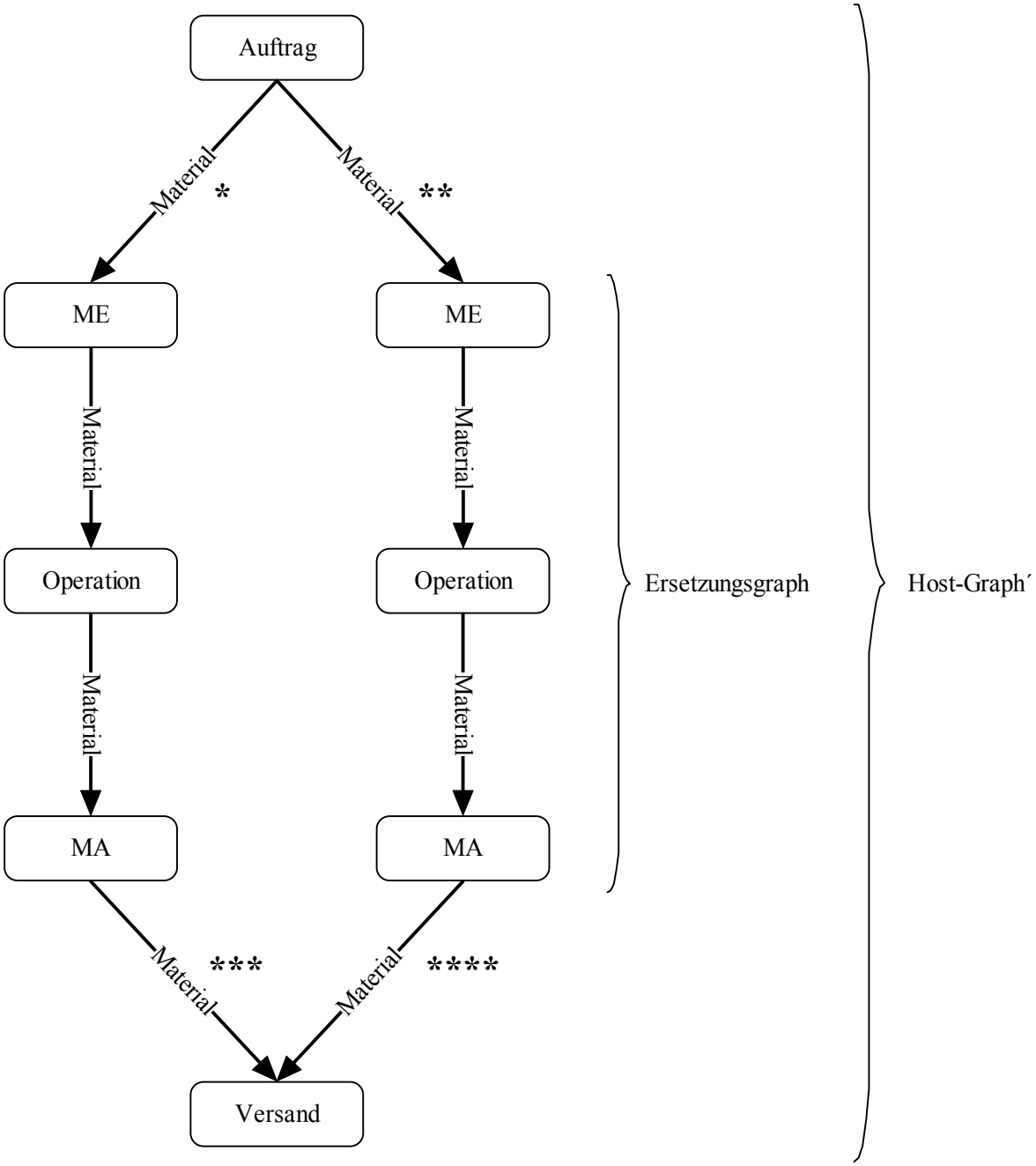



**Abbildung 9.13: Regel 2 wird auf einen Host-Graphen angewendet**




*  = Diese Kante entsteht durch Einbettungsregel 1
**  = Diese Kante entsteht durch Einbettungsregel 2
***  = Diese Kante entsteht durch Einbettungsregel 5
**** = Diese Kante entsteht durch Einbettungsregel 6

**Abbildung 9.14: Der neue Host-Graph nach Anwendung der zweiten Regel**

### 9.2.4 Regel 3-7: Operation spezialisieren

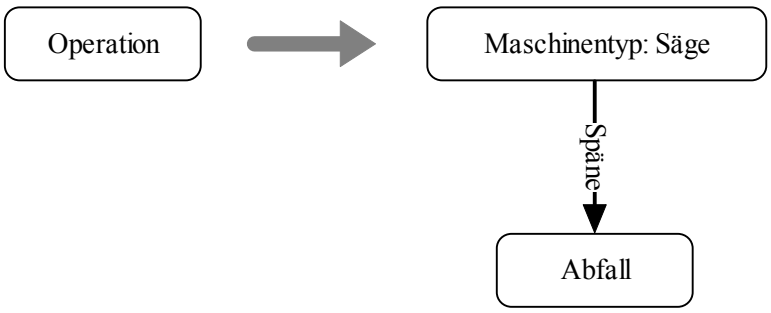



**Abbildung 9.15: Regel 3, Spezialisierung einer Operation in einen Maschinentyp und Abfall**

In Abbildung 9.15 wird der Knoten „Operation“ in einen Maschinentyp umgewandelt. Bei dieser Regel wird der Knoten „Maschinentyp: Säge“ und der Knoten „Abfall“ erzeugt. Die beiden Knoten sind mit der Kante „Späne“ verbunden. Auch diesmal sind wieder mehrere Einbettungsregeln notwendig, um den Ersetzungsgraphen in den Host-Graphen einzubinden.

Zielgraph $T = <V_T,E_T,o_T> = <\{1\},\{\},\{(1,\text{Operation})\}>$
Ersetzungsgraph $R = <V_R,E_R,o_R> = <\{2,3\},\{(2,3)\},\{(2,\text{Maschinentyp: Säge}),(3,\text{Abfall})\}>$

**Einbettungsregeln I = {**
**((ME, Operation, Material),(ME, Maschinentyp: Säge, Material)),**
**((MA, Operation, Material),(MA, Maschinentyp: Säge, Material))}**

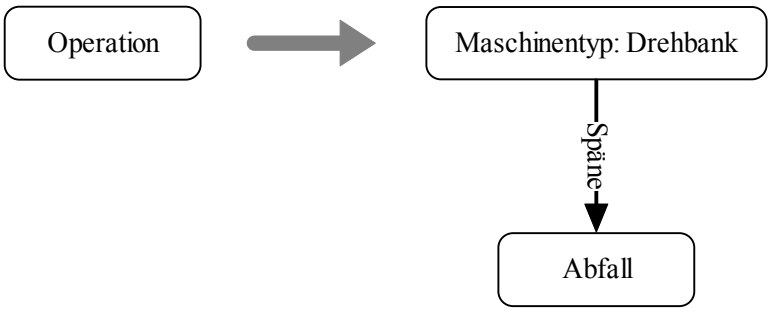



**Abbildung 9.16: Regel 4, Spezialisierung einer Operation in einen Maschinentyp und Abfall**

Regel 4, die in Abbildung 9.16 dargestellt wird, ersetzt den Knoten „Operation“ durch die beiden Knoten „Maschinentyp: Drehbank“ und „Abfall“. Wie auch in Regel 4 müssen Einbettungsregeln dafür sorgen, dass die beiden neuen Knoten in den Host-Graphen eingefügt werden.

Zielgraph $T = <V_T,E_T,o_T> = <\{1\},\{\},\{(1,\text{Operation})\}>$
Ersetzungsgraph $R = <V_R,E_R,o_R> = <\{2,3\},\{(2,3)\},\{(2,\text{Maschinentyp: Drehbank}),(3,\text{Abfall})\}>$

**Einbettungsregeln I = {**
**((ME, Operation, Material),(ME, Maschinentyp: Drehbank, Material)),**
**((MA, Operation, Material),(MA, Maschinentyp: Drehbank, Material))}**

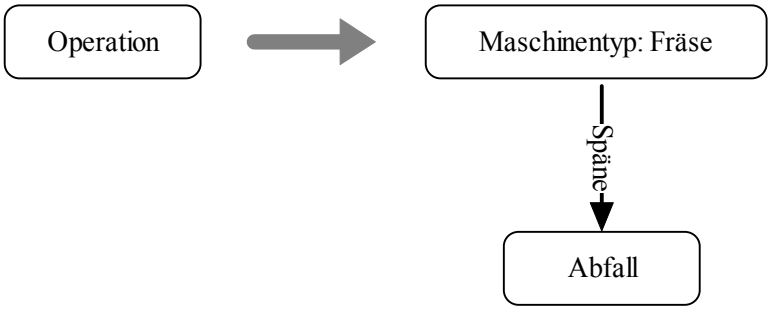



**Abbildung 9.17: Regel 5, Spezialisierung einer Operation in einen Maschinentyp und Abfall**

Regel 5, die in Abbildung 9.17 dargestellt wird, ersetzt den Knoten „Operation“ durch die Knoten „Maschinentyp: Fräse“ und „Abfall“.

Zielgraph $T = <V_T,E_T,o_T> = <\{1\},\{\},\{(1,\text{Operation})\}>$
Ersetzungsgraph $R = <V_R,E_R,o_R> = <\{2,3\},\{(2,3)\},\{(2,\text{Maschinentyp: Fräse}),(3,\text{Abfall})\}>$

**Einbettungsregeln I = {**
**((ME, Operation, Material),(ME, Maschinentyp: Fräse, Material)),**
**((MA, Operation, Material),(MA, Maschinentyp: Fräse, Material))}**

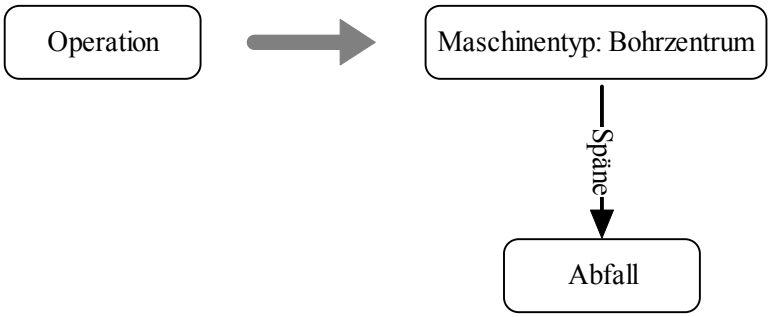



**Abbildung 9.18: Regel 6, Spezialisierung einer Operation in einen Maschinentyp und Abfall**

In Abbildung 9.18 wird Regel 6 dargestellt. Diese ersetzt ebenfalls den Knoten mit Label „Operation“ durch zwei Knoten mit Label „Maschinentyp: Bohrzentrum“ und Label „Abfall“.

Zielgraph $T = <V_T,E_T,o_T> = <\{1\},\{\},\{(1,\text{Operation})\}>$
Ersetzungsgraph $R = <V_R,E_R,o_R> = <\{2,3\},\{(2,3)\},\{(2,\text{Maschinentyp: Bohrzentrum}),(3,\text{Abfall})\}>$

**Einbettungsregeln I = {**
**((ME, Operation, Material),(ME, Maschinentyp: Bohrzentrum, Material)),**
**((MA, Operation, Material),(MA, Maschinentyp: Bohrzentrum, Material))}**

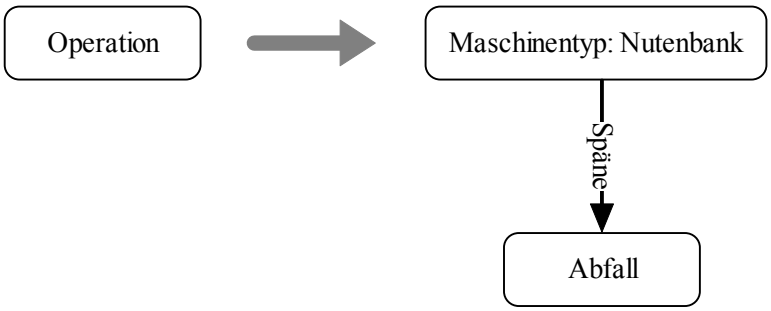



**Abbildung 9.19: Regel 7, Spezialisierung einer Operation in einen Maschinentyp und Abfall**

Die letzte Regel wird in Abbildung 9.19 dargestellt. Auch hier wird eine Operation durch zwei neue Knoten ersetzt.

Zielgraph $T = <V_T,E_T,o_T> = <\{1\},\{\},\{(1,\text{Operation})\}>$
Ersetzungsgraph $R = <V_R,E_R,o_R> = <\{2,3\},\{(2,3)\},\{(2,\text{Maschinentyp: Nutenbank}),(3,\text{Abfall})\} >$

**Einbettungsregeln I = {**
**((ME, Operation, Material),(ME, Maschinentyp: Nutenbank, Material)),**
**((MA, Operation, Material),(MA, Maschinentyp: Nutenbank, Material))}**

### 9.2.5 Der Strukturgraph

Durch die Anwendung der gerade beschriebenen Regeln kann ein Graph erzeugt werden, dieser wird als Strukturgraph bezeichnet. Seine Aufgabe besteht darin, die Bearbeitungsreihenfolge abzubilden und eine Anordnung der Maschinen darzustellen. Die Maschinentypen, die durch Regel 3-7 eingesetzt wurden, müssen dann später durch konkrete Maschinen ersetzt werden.

## 9.3 Simulation 1

Nachdem mehrere Design-Graphgrammatik Regeln angewendet wurden, steht ein Strukturgraph zur Verfügung, der durch keine Regel mehr verändert werden kann.
Der entstandene Graph ist zwar ungerichtet, aber es findet ein Materialfluss zwischen den Maschinen statt. Aufgrund des Materialflusses, von dem Knoten „Auftrag“ zu dem Knoten „Versand“, wird ein Werkstück in einer bestimmten Reihenfolge durch die Maschinen bearbeitet. Diese Bearbeitungsreihenfolge ist manchmal vorgeschrieben.

Zu jedem Werkstück, das gefertigt werden soll, gehört eine technische Zeichnung. Auf ihr wird vermerkt, wie das Werkstück am Ende des Arbeitsprozesses aussehen soll (Kapitel 9.1).
Aufgrund der Zeichnung sind einige Arbeitsabläufe fest vorgeschrieben. Eine Abweichung von dem vorgeschriebenen Arbeitsablauf ist zwar manchmal möglich, allerdings wird der Arbeitsablauf dadurch viel komplizierter. Ein Beispiel hierfür wären Löcher, die an einer bestimmten Stelle des Werkstückes zu bohren sind. Die Position des Loches ist durch die Position der gefrästen Zähne vorbestimmt.
Es ist einfacher ein Loch an einem gefrästen Zahn auszurichten, als umgekehrt.

Manche Arbeitsabläufe schließen sich auch einfach aus.
Es ist zum Beispiel nicht möglich, eine „Materialstange“ zuerst mit der Drehbank zu bearbeiten und anschließend mit einer Säge zu zersägen, da das Material, welches in die Drehbank eingelegt

werden muss, einfach zu groß ist. Die Stangen, die gesägt werden, haben eine Länge von 5 Metern, diese Stangen können unmöglich mit einer Drehbank bearbeitet werden.

Es ist auch nicht möglich, eine Nut in ein Werkstück zu ziehen, ohne dass vorher eine Drehbank ein Loch in das Werkstück gedreht hat.

Es gibt noch viele Beispiele, die belegen, dass manche Werkstücke nur mit einem fest vorgegebenen Arbeitsablauf hergestellt werden können.
In der Realität sieht ein Mensch diese Abhängigkeiten aufgrund der technischen Zeichnung oder seine Erfahrung hilft ihm weiter.
In dieser Diplomarbeit wird das Erkennen von nötigen Arbeitsabläufen durch eine Simulation ersetzt. In der technischen Zeichnung, die dem Programm zur Verfügung steht, werden alle möglichen Arbeitsabläufe als Pfade dargestellt.
Diese Pfade müssen auch in dem Graphen, den die Design-Graphgrammatik erzeugt hat, wieder gefunden werden. Erst wenn alle Knoten/Kanten des Graphen in einem Pfad gefunden werden, gilt ein Graph als gültiger Arbeitsablauf, der weiter verarbeitet werden kann.

Somit kann eine technische Zeichnung als eine Anordnung von Pfaden angesehen werden.

Beispiel 1:

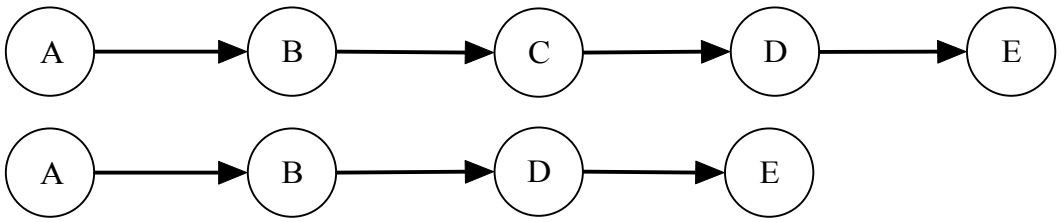



**Abbildung 9.20: Eine technische Zeichnung**

In Abbildung 9.20 werden zwei mögliche Arbeitsabläufe als Pfade dargestellt, der obere Pfad benötigt fünf Maschinen um das geforderte Werkstück herzustellen, der untere Pfad stellt dasselbe Werkstück her, allerdings mit vier Maschinen.
Beide Pfade wären auf der technischen Zeichnung zu finden.

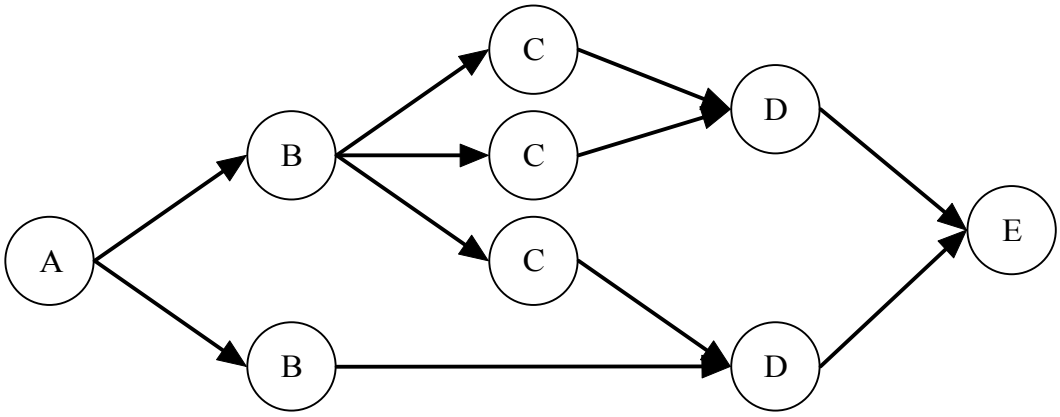



**Abbildung 9.21: Ein Graph der mit einer Design-Graphgrammatik erstellt wurde**

In Abbildung 9.21 wird eine mögliche Maschinenanordnung dargestellt, die mit einer Design-Graphgrammatik erzeugt wurde. Die Simulation würde diesen Graphen als Lösung akzeptieren.

Es ist der Simulation möglich jeden Knoten/Kante des Graphen aus Abbildung 9.21 zu markieren, dabei hilft der Simulation die „technische Zeichnung“ aus Abbildung 9.20.

Das Beispiel zeigt, dass es problemlos möglich ist, komplexe Arbeitsabläufe mit wenigen Pfaden darzustellen. Die erste Simulation prüft also, ob der erzeugte Strukturgraph nur korrekte Arbeitsabläufe enthält. Ist dies der Fall, dann kann der Strukturgraph weiter verarbeitet werden.

## 9.4 Ressourcenorientierte Konfiguration

In Kapitel 9.3 wurde die strukturelle Anordnung der Maschinentypen überprüft, ist diese Überprüfung positiv, müssen die Maschinentypen in konkrete Maschinen umgewandelt (instanziiert) werden. Hierbei wird die Technik der globalen Bilanzverarbeitung aus Kapitel 8 benutzt.

Ein Programm, dass unter anderem die globale Bilanzverarbeitung zur Verfügung stellt, heißt „Local Balance“ und wurde von Serdar Özalp im Rahmen seiner Diplomarbeit geschrieben [9].

Damit das Programm „Local Balance“ eine Lösung berechnen kann, müssen sowohl die Maschinen, als auch die Arbeitsaufgabe abgebildet werden.
Es werden dabei zwei Typen unterschieden, einmal die Ressourcen und die Komponenten.
Eine Ressource kann angeboten und verbraucht werden, alle Ressourcen sind global verfügbar.
Eine Komponente stellt Ressourcen zur Verfügung und verbraucht diese.
Eine Problemstellung muss also in eine Ressourcenanfrage/Startbilanz umgewandelt werden, damit das Programm „Local Balance“ mögliche Komponenten anbieten kann, die die geforderten Ressourcen zur Verfügung stellen.

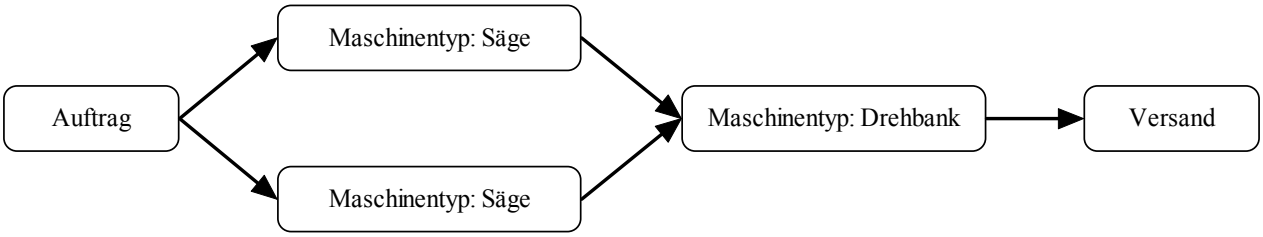



**Abbildung 9.22: strukturelle Anordnung dreier Maschinentypen**

In Abbildung 9.22 wird die strukturelle Anordnung dreier Maschinen gezeigt, zwei Sägen, bisher unbekannten Typs, bearbeiten einen Auftrag parallel. Das Erzeugnis der Sägen wird mit einer Drehbank weiter verarbeitet und anschließend versandt. Diese Struktur wurde mit Hilfe der Design-Graphgrammatik Regeln erzeugt und muss nun in eine Ressourcenanforderung umgewandelt werden. Die Ressourcenanforderung wird anschließend mit der Bilanzverarbeitung gelöst. Eine Möglichkeit ist es, die geforderten Maschinen als eine Ressource anzusehen. Eine Säge wäre somit eine Ressource „sägen“, die gefordert wird. Als nächstes muss eine Komponente diese Ressource anbieten, zum Beispiel eine Komponente „Säge 100“, die mehrere Ressourcen anbietet und fordert. Unter anderem wird aber auch die Ressource „sägen“ einmal angeboten.

Die in Abbildung 9.22 dargestellte Struktur könnte in folgende Ressourcenanforderung umgewandelt werden:

2 mal Ressource „sägen“
1 mal Ressource „drehen“

Die Aufgabe der Bilanzverarbeitung wäre also, alle Komponenten zu prüfen, ob diese die Ressourcen „drehen“ und „sägen“ zur Verfügung stellen.
Danach würden alle Komponenten, die die geforderten Ressourcen anbieten, ausgegeben.

Damit eine Maschine ihre Aufgabe erfüllen kann, benötigt sie Menschen für die Bedienung, Energie für die Motoren der Maschine, Kühlwasser zum kühlen der Werkzeuge und zu guter Letzt einen Stellplatz, der groß genug ist, dass die Maschine aufgestellt werden kann.
Dies alles muss mit Hilfe von Ressourcen dargestellt werden, es existiert also eine Ressource „Meister“ und eine Komponente „Meister zur Verfügung stellen“, die eine Ressource „Meister“ anbietet. Unter anderem gibt es noch die Ressourcen Kühlwasser, Strom und „Platz“. Natürlich gibt es Komponenten, die diese Ressourcen zur Verfügung stellen.

Damit ein bestimmtes Werkstück erzeugt werden kann, müssen unterschiedliche Arbeitsprozesse ablaufen. Diese Arbeitsprozesse werden von den Maschinen angeboten. Dabei können die Arbeitsprozesse unterschiedliche Schwierigkeitsgrade besitzen, es ist zum Beispiel einfacher ein Loch mit einem Durchmesser von 2 cm zu bohren, als ein Loch mit einem Durchmesser von 30 cm. Das Loch mit dem Durchmesser von 30 cm benötigt viel mehr Vorarbeiten. Zuerst wird ein kleines Loch gebohrt, dieses wird dann immer weiter aufgebohrt. Dieses Vorgehen verlängert den Arbeitsprozess, ist aber unumgänglich. Ein weiteres Beispiel wären Maschinen mit unterschiedlichen Bearbeitungsmöglichkeiten. Eine Maschine, die sich auf dem neusten Stand der Technik befindet, kann andere Produktionsprozesse anbieten, als eine ältere Maschine, die diese Produktionsprozesse gar nicht oder nur sehr langsam anbieten kann.
Damit dieser Sachverhalt dargestellt werden kann, bieten Maschinen unterschiedliche Arbeitsprozesse in verschiedenen Schwierigkeitsgraden an, diese werden im Folgenden Komplexitätsgrade genannt.
Der Einfachheit halber besitzt jede Maschine zwei Komplexitätsgrade, Komplexitätsgrad A und Komplexitätsgrad B. Der Komplexitätsgrad A ist hierbei der Einfachere von den beiden.
Jede Maschine kann pro Stunde nur eine gewisse Menge von Werkstücken herstellen, diese Menge ist dabei vom Komplexitätsgrad abhängig. Je komplexer die Aufgabe, desto weniger Werkstücke können erzeugt werden.
In der technischen Zeichnung wird der Komplexitätsgrad, der vorhanden sein muss, um das Werkstück zu fertigen, angegeben.

Eine Maschine würde Folgende Ressourcen anbieten:

- Arbeitsprozess (sägen, bohren, fräsen, usw.), (1 mal)
- Herstellungsmenge je Stunde für Komplexitätsgrad A, (X mal)
- Herstellungsmenge je Stunde für Komplexitätsgrad B, (Y mal)

Eine Maschine würde Folgende Ressourcen fordern:

- Arbeiter mit nötiger Qualifikation (Lehrling, Facharbeiter, Meister), (1 mal)
- Benötigte Leistung (kW/h)
- Benötigte Kühlwassermenge (l/h)
- Platzbedarf der Maschine ($m^2$)

Die Ressourcenanfrage, die zu Abbildung 9.22 gehört müsste somit um die Komplexitätsgrade des „Maschinentyp: Säge“ und des „Maschinentyp: Drehbank“ erweitert werden.

Falls der Auftrag 500 Werkstücke umfasst und die Werkstücke einen Komplexitätsgrad der Säge von A benötigen und einen Komplexitätsgrad B für den „Maschinentyp: Drehbank“, so würde die Ressourcenanfrage wie folgt aussehen:

- 2 mal „sägen“
- 1 mal „drehen“
- 500 Einheiten/Stunde Komplexitätsgrad A „sägen“
- 500 Einheiten/Stunde Komplexitätsgrad B „drehen“

Bisher wurde immer davon Ausgegangen, dass in einer Lösung immer genau passend viele Maschinen enthalten waren. Diese Annahme ist auch richtig, solange nur die minimal benötigten Maschinen betrachtet werden. In Abbildung 9.22 wird zwei Mal die Ressourcen "sägen" gefordert, sobald 2 Sägen gefunden wurde ist diese Bedingung erfüllt. Was passiert aber, wenn diese beiden Sägen die geforderten 500 Einheiten/Stunde vom Komplexitätsgrad A nicht leisten? Das Programm „Local Balance“ würde eine weitere Säge suchen und diese Benutzen. Eine Lösung für Abbildung 9.22 wären somit auch drei Sägen, die zusammen 500 Einheiten/Stunde vom Komplexitätsgrad A herstellen. In Abbildung 9.22 können aber nur zwei Sägen platziert werden. Aus der Sicht der globalen Bilanzverarbeitung ist die Lösung mit drei Sägen richtig, da die globale Bilanz positiv ist. Aber aus der Sicht des Strukturgraphen ist die Lösung völlig unsinnig.
Dieses Dilemma kann mit einem kleinen Trick gelöst werden. Es wird eine weitere Ressource erzeugt, diese wird aber von keiner Komponente angeboten, sondern nur verbraucht.
Für den Graphen aus Abbildung 9.22 würde eine Ressource „maximale Anzahl Sägen = 2“ mit in die Startbilanz geschrieben. Jede Säge verbraucht genau eine Ressource „maximale Anzahl Sägen“. Sobald zwei Sägen eingesetzt wurde, steht die Ressource mit einer „Null“ in der globalen Bilanz. Wenn eine weiter Säge eingesetzt werden soll, weil zum Beispiel die geforderte Menge je Stunde nicht erreicht wird, dann wird der Eintrag „maximale Anzahl Sägen“ in der globalen Bilanz negativ. Da es keine Komponenten gibt, die diese Ressource erzeugen kann, wird es niemals zu einer Lösung kommen.

Mit diesen Informationen ist es dem Bilanzverarbeitungsprogramm „Local Balance“ möglich eine Liste mit Maschinen auszugeben, die alle Anforderungen erfüllt.

## 9.5 Simulation 2

In Kapitel 9.4 wurde mit Hilfe der globalen Bilanzverarbeitung eine Liste von Komponenten erstellt, die eine vorgegebene Ressourcenanforderung erfüllt. Diese Komponenten entsprechen Maschinen, die Werkstücke bearbeiten. Die Aufgabe der zweiten Simulation ist es, die Maschinenauswahl zu überprüfen und anschließend die ausgewählten Maschinen im Strukturgraph zu platzieren. Bei der Maschinenauswahl achtet die globale Bilanzverarbeitung nicht auf Hallenspezifische oder Firmenspezifische Gegebenheiten. Es kann also durchaus passieren, dass die globale Bilanzverarbeitung Maschinen auswählt für die gar kein Personal vorhanden ist oder aber der Stellplatz der Halle reicht nicht mehr aus. Diese Restrektionen werden im ersten Schritt der Simulation geprüft, erst wenn die gewählten Maschinen alle Restriktionen erfüllen, wird versucht die gewählten Maschinen im Strukturgraph zu platzieren. Bei der Platzierung kann es geschehen, dass die geforderte Herstellungsmenge pro Stunde nicht erfüllt werden kann. Dieser Fehler tritt auf, da die Bilanzverarbeitung die lokale Struktur des Strukturgraphen bei der Lösungsfindung nicht beachtet.

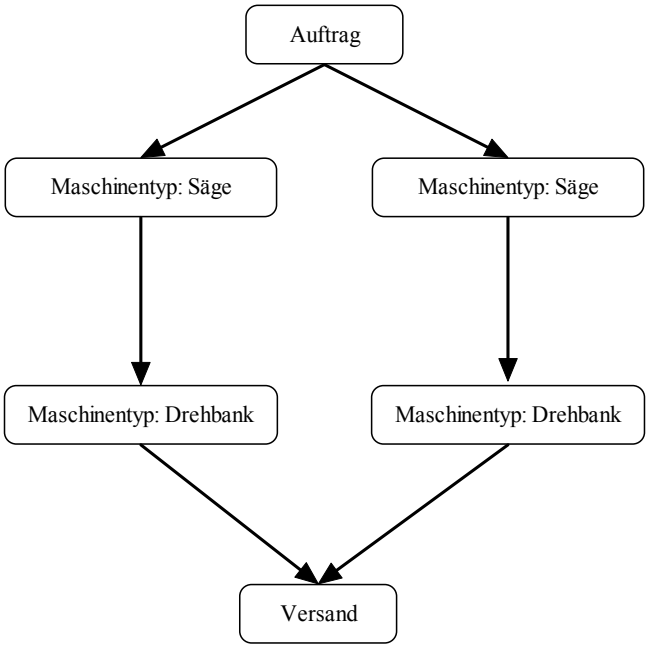



**Abbildung 9.23: Ein Strukturgraph**

In Abbildung 9.23 ist ein Strukturgraph dargestellt, in dem Werkstücke parallel durch eine Säge und eine Drehbank bearbeitet werden. Dieser Strukturgraph würde in eine Startbilanz für die Bilanzverarbeitung umgewandelt.

- 2 X sägen
- 2 X drehen
- 500 Werkstücke pro Stunde (Säge)
- 500 Werkstücke pro Stunde (Drehbank)

Das Ergebnis der globalen Bilanzverarbeitung könnte wie folgt aussehen.

- Säge Modell 250
- Säge Modell 250
- Drehbank Modell 300
- Drehbank Modell 200

Eine Säge Modell 250 stellt 250 Werkstücke in einer Stunde her, eine Drehbank Modell 200 stellt 200 Werkstücke pro Stunde her und eine Drehbank Modell 300 kann 300 Werkstücke pro Stunde fertigen. Die Lösung der Bilanzverarbeitung ist korrekt, die beiden Sägen bieten beide die Ressource „sägen“ an und zusammen 500 Werkstücke pro Stunde. Auch die beiden Drehbänke erfüllen die geforderten 500 Werkstücke je Stunde und bieten ebenfalls die Ressource„drehen“ an. Aus Sicht der Bilanzverarbeitung ist diese Lösung korrekt. Sollen nun aber die gewählten Maschinen in dem Strukturgraphen platziert werden tritt ein Problem auf.

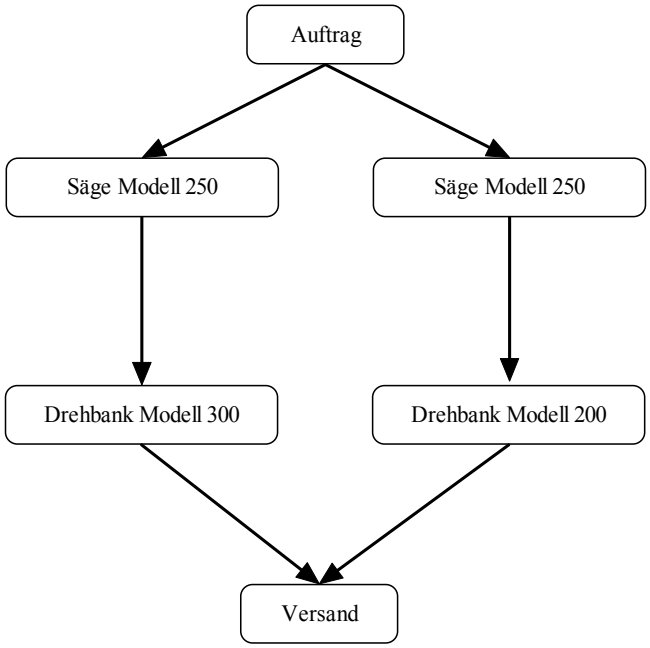



**Abbildung 9.24: Strukturgraph mit platzierten Maschinen**

In Abbildung 9.24 wurden alle Maschinen im Strukturgraphen platziert. Wird der Strukturgraph genauer betrachtet, so fällt auf, dass die geforderte Menge von 500 Werkstücken je Stunde nicht erfüllt werden kann. Dies liegt an der lokalen Struktur des Strukturgraphen. Die beiden Sägen vom Modell 250 beliefern jeweils eine Drehbank, somit stehen jeder Drehbank 250 Werkstücke zur Verfügung. Die Drehbank vom Modell 200 kann aber nur 200 Werkstücke verarbeiten, die restlichen 50 werden vor der Maschine liegen bleiben und nicht bearbeitet. Die Drehbank Modell 300 kann 300 Werkstücke verarbeiten, bekommt aber nur 250 Werkstücke geliefert. Somit stellen beide Drehbänke zusammen 450 Werkstücke in einer Stunde her. Die geforderte Menge von 500 Werkstücken pro Stunde wird nicht erreicht. Die globale Bilanzverarbeitung geht von global verfügbaren Ressourcen aus, jede Ressource ist überall verfügbar. Aber genau dies ist nicht der Fall, der Strukturgraph legt genau fest wo welche Ressource vorhanden sein muss. Aus diesem Grund kann die Lösung der Bilanzverarbeitung fehlerhaft sein und muss überprüft werden.

Bei der Platzierung der Maschinen müssen alle möglichen Maschinenanordnungen überprüft werden. Dies ist nötig, weil eine Maschine, die einen Maschinentypen ersetzt, an dieser Stelle nicht geforderte Leistung erbringt. Dieselbe Maschine kann aber, an einer anderen Stelle, sehr wohl die geforderten Leistungen erbringen.

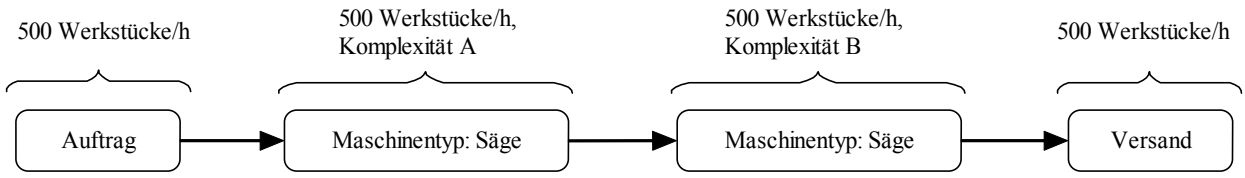



**Abbildung 9.25: seriell Schaltung von Sägen und deren geforderte Kapazitäten**

In Abbildung 9.25 werden 2 Sägen gefordert, die erste Säge muss 500 Werkstücke mit Komplexität A zur Verfügung stellen. Die zweite Säge muss 500 Werkstücke vom Komplexitätsgrad B zur Verfügung stellen. Dabei muss der Arbeitsprozess, der den Komplexitätsgrad A erfordert, zuerst ausgeführt werden. Danach kann der zweite Arbeitsvorgang, der den Komplexitätsgrad B fordert, ausgeführt werden.

Die Bilanzverarbeitung bietet zwei mögliche Sägen an. Das Modell „Säge A 500“ stellt 500 Werkstücke in einer Stunde mit Komplexitätsgrad A her und ein Werkstück pro Stunde vom Komplexitätsgrad B. Die zweite Säge wird als „Säge B 500“ bezeichnet, ihre Kapazität liegt bei 500 Werkstücken in der Stunde vom Komplexitätsgrad B und einem Werkstück pro Stunde für den Komplexitätsgrad A. Dadurch ergeben sich zwei Möglichkeiten, wie die Maschinen in die Struktur eingefügt werden können.

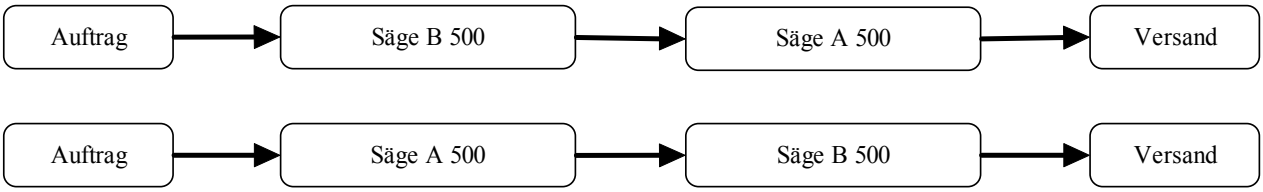



**Abbildung 9.26: Alle möglichen Anordnungen der beiden Sägen**

In Abbildung 9.26 werden alle möglichen Anordnungen der beiden Sägen dargestellt. Die obere Abbildung ist dabei keine gültige Lösung, da Säge B nur ein Werkstück in der Stunde mit dem geforderten Komplexitätsgrad erzeugen kann. Die untere Abbildung hingegen ist eine gültige Lösung, da beide Maschinen die geforderten Kapazitäten erreichen können.
Dies zeigt, dass alle möglichen Maschinenkombinationen getestet werden müssen.
Sobald alle Maschinentypen durch konkrete Maschinen ersetzt wurden, muss die Kapazität der einzelnen Maschinen überprüft werden.

Wie Abbildung 9.26 zeigt, besitzt der Graph eine Quelle und eine Senke. Die Quelle ist der Knoten mit dem Label „Auftrag“ und die Senke der Knoten mit dem Label „Versand“.
Da eine Quelle und eine Senke vorhanden sind, findet ein Materiellfluss statt. Der Fluss wird durch die Kapazität der Maschinen begrenzt. Der maximale Fluss des Graphen entspricht dabei der maximalen Menge von Werkstücken, die in einer Stunde produziert werden können.
Eine Möglichkeit, den maximalen Fluss eines Graphen zu berechnen, ist ein „Max flow“ Algorithmus. Die Kapazitäten der Maschinen, können dabei als Kapazitäten der Kanten angesehen werden.

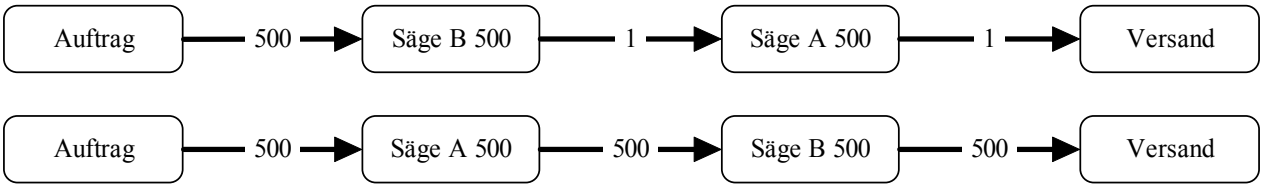



**Abbildung 9.27: Maschinenkapazitäten werden an den Kanten dargestellt**

In Abbildung 9.27 werden die Kapazitäten der Maschinen an den Kanten dargestellt, die Quelle sendet 500 „Werkstücke“ in den Graphen.
In der oberen Abbildung kann Säge B 500 aber nur ein Werkstück fertigen, somit kann Säge A 500 nur ein Werkstück herstellen. Der Versand bekommt ein Werkstück von Maschine Säge A 500. Der maximale Fluss wäre somit ein Werkstück. Die geforderte Menge von 500 Werkstücken kann somit nicht erreicht werden.

Die untere Abbildung in Abbildung 9.27 hingegen, hat einen maximalen Fluss von 500. Dies entspricht der geforderten Menge von 500 Werkstücken, die in einer Stunde hergestellt werden müssen.

Wenn der maximale Fluss eines Graphen ausreicht, wurde eine gültige Lösung gefunden.
Der Graph enthält nun konkrete Maschinen, die ein Werkstück erzeugen können.

## 9.6 Innerbetriebliche Standortplanung

Nachdem in Kapitel 9.5 die Maschinenauswahl der Bilanzverarbeitung in dem Strukturgraphen platziert wurde und anschließend überprüft wurde, ob die gegebenen Maschinen ausreichen um die geforderte Produktmenge zu erzeugen, müssen nun die Maschinen in der Werkhalle aufgestellt werden. Dabei soll eine Materialflussorientierte Sicht angestrebt werden. Als Konsequenz daraus sind innerbetriebliche Transporte und die innerbetriebliche Lagerung von Materialien genauer zu betrachten. Dies wird in der Literatur als innerbetriebliche Standortplanung bezeichnet [5,10,11]. Da die innerbetriebliche Standortplanung mehr umfasst, als das anordnen von Maschinen in Werkshallen, sei hier darauf hingewiesen, dass nur dieser Teil betrachtet werden soll.

Das Ziel eines jeden wirtschaftlich geführten Unternehmens ist die Gewinnmaximierung. Da das Hauptaugenmerk auf innerbetriebliche Transporte und Lagerung liegt folgt daraus, dass sowohl die Transportkosten, als auch die Lagerkosten, so gering wie möglich sein sollten. Je höher die eben genannten Kosten, desto niedriger fällt der Gewinn aus. Im Folgenden ist das primäre Ziel, die Senkung der Transportkosten. Die Lagerkosten werden dabei nicht näher beachtet.

Es gibt mehrere Modelle, die versuchen bei der Auswahl von Stellplätzen für Maschinen die Transportwege zu berücksichtigen und somit möglichst niedrige Transportkosten zu erzeugen. Bei dem Modell, dass hier vorgestellt wird, sind Folgende Prämissen zu beachten [10,11]:

1. Produktionsprogramm und Absatzprogramm sind gegeben und werden durch die Layoutplanung nicht beeinflusst
2. Die Reihenfolge der Arbeitsgänge ist für jede zu fertigende Produktart bekannt
3. Es gibt eine endliche Zahl M von anzuordnenden Elementen
4. Es gibt eine endliche Zahl S von möglichen Standorten
5. Die Transportwege, die die S möglichen Standorte verbinden, sind gegeben
6. Die Transportmittel sind gegeben
7. Die Transportkosten sind proportional zur Transportmenge und Transportentfernung

Alle Punkte, die als Prämisse genannt wurden, sind in dem hier betrachteten Fall erfüllt. Punkt 1) ist durch die technische Zeichnung gegeben. Auf ihr werden das Produktionsprogramm und die Absatzmenge festgelegt. Auch die Reihenfolge der Bearbeitung (Punkt 2) ist bekannt. Eine Bearbeitungsreihenfolge wird mit dem Strukturgraphen festgelegt. Des Weiteren existiert eine endliche Anzahl von M anzuordnenden Elementen (Maschinen). Diese Menge wird durch den Strukturgraphen, der durch die Design-Graphgrammatik erzeugt wurde, festgelegt und durch die globale Bilanzverarbeitung instanziiert.
Auch die möglichen Standorte der Maschinen sind beschränkt (Punkt 4), ein Koordinatensystem dient als eindeutige Zuordnung.
Bei der Maschinenauswahl wurde bereits darauf geachtet, dass ein Materialfluss zwischen den Maschinen entstehen kann. Es ist genügend Platz vorhanden und die Transportwege sind durch die Bearbeitungsreihenfolge festgelegt. Somit ist Punkt 5 ebenfalls erfüllt. Als Transportmittel stehen Gabelstapler, Fliessbänder und bei Einzelfertigung die Muskelkraft der Arbeiter zur Verfügung. Auch Punkt 7) ist zu realisieren, je länger ein Mitarbeiter mit einem zu transportierenden Gut unterwegs ist, desto teuerer wird es (Stundelohn des Mitarbeiters). Die Transportmenge bestimmt

dabei auch die Transportkosten (ein Gabelstapler ist teuerer als ein Mitarbeiter, der ein Werkstück mit der Hand transportiert).
Da alle Restriktionen erfüllt werden kann ein Quadratisches Zuordnungsproblem bei gleichem Platzbedarf formuliert werden.

Ein exaktes Verfahren, dass bei gleichem Platzbedarf der Maschinen, die optimale Aufstellung der Maschinen berechnet wäre in der Praxis nicht anwendbar [5].

Bei 15 anzuordnenden Maschinen war die Laufzeit bereits 2947,320 Sekunden, in der Literatur wurde bisher noch keine Zeitangabe für mehr als 15 Maschinen gefunden.

Aus diesen Zahlen kann gefolgert werden, dass ein Heuristisches Verfahren das Problem lösen muss, da eine exakte Berechnung einfach zu lange dauert. Im Folgenden soll das „Umlaufverfahren von Kiehne“ beschrieben werden.

Das Verfahren von Kiehne unterteilt sich in zwei Schritte, als erstes wird ein Eröffnungsverfahren angewandt. Die Aufgabe dieses Schrittes ist es, eine erste Lösung zu produzieren und diese dann mit dem zweiten Schritt zu verbessern.

Schritt 1: Eröffnungsverfahren

1. Bestimme das erste anzuordnende Element als dasjenige Element, von dem und zu dem insgesamt die größte Menge transportiert wird
2. Ordne das im ersten Schritt ermittelte Element in der Mitte der Standortfläche an. Es bildet den Kern oder das Zentrum des Layouts
3. Bestimme unter den noch nicht angeordneten Elementen dasjenige Element, zwischen dem und den bereits angeordneten Elementen insgesamt die größten Mengen zu transportieren sind
4. Prüfe für jeden freien Standort, der am Rande des Zentrums liegt, wie hoch die Kosten des Transports zwischen dem anzuordnenden Element und den bereits angeordneten Elementen sind, wenn das ausgewählte Element dem betrachteten Standort zugeordnet wird. Als Standort für das Element wird derjenige Standort ausgewählt, der die insgesamt niedrigsten zusätzlichen Transportkosten verursacht. Das neue Zentrum umfasst alle bereits besetzten Standortflächen und die neu belegten Flächen.

Sind die Auswahlkriterien nicht eindeutig, so wird unter den Elementen mit gleichen Transportmengen bzw. den Standorten mit gleichen Transportkosten willkürlich ausgewählt. Werden Schritt 3 und 4 genau M-1-Mal wiederholt, so sind alle Elemente angeordnet und eine zulässige Lösung wurde ermittelt.

Sind mehr als drei Elemente anzuordnen, werden nach Anordnung des vierten und aller folgenden Elemente jeweils Verbesserungsschritte durchgeführt, die im Anschluss an den vierten Schritt des Eröffnungsverfahrens zu vollziehen sind.

5. Prüfe, ob sich die Kosten für den Transport zwischen den angeordneten Elementen verringern, wenn das zuletzt angeordnete Element seinen Standort mit dem Standort eines benachbarten Elements tauscht. Findet man eine solche Vertauschungsmöglichkeit, so ist ein entsprechender Standorttausch vorzunehmen.

Schritt 5 wird für das neu angeordnete Element so oft durchgeführt, bis Vertauschungsmöglichkeiten mit benachbarten Elementen, die zu einer Kostenreduzierung führen, nicht mehr zu finden sind. Danach wird mit Schritt 3 weiter gemacht.

Nach M-1 Wiederholungen der Schritte 3-5 wurde eine zulässige Lösung für das Layoutproblem gefunden. Eine Aussage über die Güte der Lösung lässt sich nicht treffen.

Ein weiteres Verfahren zur Lösung des Layoutproblems wird CRAFT (Computerized Relative Allocation of Facilities Technique) genannt. Bei diesem Verfahren muss bereits eine Startlösung vorhanden sein, diese wird dann weiter verbessert, CRAFT ist somit ein „Verbesserungsverfahren“. Auf CRAFT soll nicht weiter eingegangen werden, es soll nur als mögliche alternative zu Kiehne erwähnt werden. Die innerbetriebliche Standortplanung wurde im Rahmen der Diplomarbeit nicht mehr implementiert.

## 9.7 Die Strategie

Bisher wurde eine Design-Graphgrammatik (Kapitel 9.2) und mehrere Simulationen und Verfahren (Kapitel 9.3, Kapitel 9.4, Kapitel 9.5, Kapitel 9.6) vorgestellt. Dies alles war die Basis für eine Strategie, die die Design-Aufgabe lösen soll. Alle diese Bauteile werden im Folgenden benutzt um eine Strategie zu erzeugen.
Die Strategie muss eine Lösung des Problems erarbeiten, hierfür hat sie verschiedene Tools/Werkzeuge zur Verfügung. Als erstes kann sie, mit Hilfe der Design-Graphgrammatik, einen Strukturgraphen erzeugen. Dieser Graph enthält Maschinentypen und eine Struktur wie ein Werkstück bearbeitet werden muss.
Da bei der Erzeugung des Strukturgraphen nicht auf notwendige Reihenfolgen bei der Bearbeitung des Werkstückes geachtet wurde, muss dies nachgeholt werden. Für diesen Schritt ist die erste Simulation zuständig. Sie überprüft ob die Struktur des Arbeitsablaufes, den die Design-Graphgrammatik erzeugt hat, überhaupt gültig ist.

Ist die Struktur gültig, muss im nächsten Schritt der Strukturgraph mit real existierenden Maschinen gefüllt werden. Dies geschieht in zwei Schritten. Als erstes wird mit Hilfe der in Kapitel 9.4 beschriebenen Bilanzverarbeitung eine Menge von Maschinen erzeugt. Der Strategie stehen nach dem ersten Schritt zwei Informationen zur Verfügung. Der Strukturgraph, der alle Informationen enthält in welcher Reichenfolge die Maschinen das Werkstück bearbeiten müssen und die Lösung der Bilanzverarbeitung. In der Lösung der Bilanzverarbeitung stehen lediglich Maschinen, die eingesetzt werden müssen. Der nächste Schritt besteht darin, den Strukturgraph mit den Maschinen der Bilanzverarbeitung zu füllen. Dies geschieht mit der zweiten Simulation.

Die zweite Simulation setzt die Maschinen, unter Berücksichtigung ihrer Kapazitäten, in den Strukturgraphen ein. Wenn es möglich ist alle Maschinen zu platzieren und die geforderte Werkstückmenge je Stunde erreicht wird, kann der letzte Schritt erfolgen.
Jeder Maschine wird ein Stellplatz in der Werkhalle zugeordnet. Nach dem dies geschehen ist, wurde die Design-Aufgabe gelöst, es können Werkstücke produziert werden.

Somit besteht die Strategie aus Folgenden Schritten:

1. Erzeugen des Strukturgraphen mit einer Design-Graphgrammatik
2. Testen ob der Strukturgraph korrekt ist
3. Maschinenauswahl mit Hilfe der Bilanzverarbeitung erzeugen
4. Maschinen aus Schritt 3. in den Strukturgraphen aus Schritt 1. einsetzen
5. Die in Schritt 4 angeordneten Maschinen werden in der Werkhalle aufgestellt

## 9.8 Zusammenfassung

Die Design-Aufgabe bestand darin, Zahn und Kettenräder zu produzieren. Die Startsituation war dabei eine leeren Fabrikhalle und einer technischen Zeichnung, die das Werkstück abbildete. Es wurden Werkzeuge ausgewählt, die beim Lösen der Design-Aufgabe behilflich sein sollten. Ein Werkzeug war dabei die Design-Graphgrammatik, sie wurde dafür benutzt die Struktur des Arbeitsablaufs darzustellen. Eine Simulation im Anschluss kontrollierte diese Struktur. Ein weiteres Tool/Werkzeug ist die Bilanzverarbeitung, ihre Aufgabe bestand darin, Komponenten auszuwählen. Auch diese Auswahl wurde mit einer Simulation geprüft. Anschließend wurden die Lösung der Bilanzverarbeitung und der Strukturgraph zusammengefügt um einen Strukturgraph mit konkreten Maschinen zu erhalten. Im letzten Schritt wurde das letzte Werkzeug/Tool benutzt, seine Aufgabe bestand darin, die Maschinen, unter bestimmten Bedingungen, in der Werkhalle zu platzieren. Die eben genannten Werkzeuge/Tools arbeiteten nicht zufällig zusammen, sondern erzeugten Zwischenlösungen, die benutzt wurde um das nächste Werkzeug mit den Zwischenlösungen zu benutzen. Die Reihenfolge in der die Werkzeuge eingesetzt wurden, wurde dabei durch die Strategie bestimmt.
Die Design-Aufgabe wurde also durch eine Kombination von Strategie, welches Werkzeug wird wie eingesetzt, und Werkzeugen gelöst.
Wird nun mit der realen Entwicklungsumgebung für Design-Aufgaben gearbeitet, so muss lediglich jedes benötigte Werkzeug auf dem Blackboard vorhanden sein. Dann muss eine Strategie erstellt werden, diese fordert dann vom Blackboard alle benötigten Werkzeuge an und kann dann die Werkzeuge benutzen um die Design-Aufgabe zu lösen. Dabei sind sowohl Strategie, als auch Werkzeuge, auf dem Blackboard gespeichert.

# 10 Zusammenfassung und Ausblick

Das Konzept einer Entwicklungsumgebung für Design-Aufgaben konnte in dieser Diplomarbeit realisiert werden. Es existiert nun ein Programm, das in der Lage ist, beliebige Design-Aufgaben zu unterstützen. Dies geschieht mit Hilfe von Strategien, welche die Vorgehensweise beim Design beschreiben unter Zuhilfenahme von Modulen wie Design-Graphgrammatiken und Bilanzverarbeitung. Das Programm enthält unter anderem auch eine grafische Benutzeroberfläche, die dem Benutzer bei der Entwicklung seiner Strategien unterstützt und eine Beobachtung des Blackboards ermöglicht, individuell nach Benutzerwünschen. Natürlich gibt es noch einige Dinge, die zu verbessern sind. Die Datenstruktur des Blackboards könnte verbessert werden. Ein Zugriff über Hash Funktionen könnte den Zugriff beschleunigen. Allerdings sollte zunächst in weiteren Beispielaufgaben das Datenvolumen auf dem Blackboard beobachtet werden. Ein gleichzeitiger Zugriff auf Elemente des Blackboards durch verschiedene Module oder Strategien wäre sicher auch sinnvoll, dies könnte mit temporären Kopien ermöglicht werden.

Die Strategien könnten über eine Script-Sprache realisiert werden. Dadurch könnten auch Laien ohne Java Kenntnisse Strategien erstellen. Sobald eine Script-Sprache festgelegt ist, könnte ein professioneller Editor erstellt werden, der Fehler bei der Eingabe sofort erkennt, und ein passender Interpreter/Debugger.

Für die Module können weitere Hilfsmittel bereitgestellt werden. So wäre beispielsweise ein Editor für Design-Graphgrammatiken beim Erstellen von Regeln sehr hilfreich. Es müssen auch weitere Module eingebunden werden, die Wissen darstellen und verarbeiten, zum Beispiel Constrained Netze.

Die Oberfläche könnte noch stärker im Ablauf einer Strategie eingebunden werden. Derzeit existiert ein Logbuch, das nur Benutzereingaben protokolliert. Dieses Logbuch könnte auch für Strategien benutzt werden, somit wäre das Auffinden versteckter Fehler leichter.

Alle Funktionen, die bisher angeboten werden, basieren auf der bisher, auf wenige Beispiele, beschränkten Nutzung des Programms. Erst die Praxis wird zeigen, welche Funktionen wirklich sinnvoll sind und welche Funktionen noch angeboten werden müssen. Eine gründliche Evaluierung wird nötig sein, um dies herauszufinden.

Es kann eine Menge weiterer Funktionen eingefügt werden, aber es ist jetzt schon möglich Design-Aufgaben zu lösen. Dies sollte mit dem Beispiel aus Kapitel 9 gezeigt werden. Anstatt viele neue Funktionen einzufügen, die sicherlich ihren Sinn gehabt hätten, wurde an einem ausführlichen Beispiel die Arbeit mit der Entwicklungsumgebung für Design-Aufgaben dargestellt. Mit diesem Wissen sollte es möglich sein, Design-Aufgaben zu lösen. Das Programm ist eine funktionsfähige Umgebung, in der bereits nicht-triviale Aufgaben gelöst werden können. Die Erweiterung sollte problemlos möglich sein, da alle neuen Funktionen nur mit dem Blackboard zusammenarbeiten müssen.

Das Beispiel aus Kapitel 9 zeigt, dass die Laufzeit sehr stark von der Strategie abhängig ist. Die Strategie bestimmt, welche Module (Design-Graphgrammatik, Bilanzverarbeitung, usw.) benutzt werden und mit welchen Informationen diese Module arbeiten. Wenn die Design-Graphgrammatik so modelliert wurde, dass viele Regelanwendungen möglich sind und viele Graphen doppelt erzeugt werden können, dann wird dieses Modul sehr lange arbeiten, bis eine Lösung ausgegeben werden kann. Das gleiche gilt für die Bilanzverarbeitung, je besser mögliche Lösungen eingeschränkt werden, desto schneller wird eine Lösung erzeugt. Der Schlüssel zur Effizienz bei

der Lösung von Design-Aufgaben liegt damit weiterhin in der Geschicklichkeit des Modellierers. Dies war aber auch nicht anders zu erwarten.

# Anhang A

In diesem Kapitel werden alle Dateien beschrieben, die nötig sind um das Programm zu benutzen.

## *A.1 Dateiformat für eine Design-Graphgrammatik*

Damit eine Design-Graphgrammatik aus einer Datei eingelesen werden kann, muss ein bestimmtes Format eingehalten werden. Dieses Format soll in diesem Unterkapitel näher erläutert werden. Eine gültige Design-Graphgrammatik besteht aus einem Startgraphen und einer Regelmenge.

### A.1.1 Definition eines Graphen

Ein Graph besteht aus einer Knotenmenge und einer Kantenmenge. Eine Knotenmenge besteht wiederum aus Knoten, eine Kantenmenge besteht aus Kanten.

Ein Knoten hat folgende Darstellung:

(Knoten ID, Knotenlabel, Knotenname)

Die ID ist eine Integer Zahl, die den Knoten eindeutig identifiziert. In jedem Graph ist die ID einmalig.
Der Knotenlabel ist ein String, ein Label kann mehrfach in einem Graphen vorhanden sein.
Der Knotenname ist ein String, der Name kann auch mehrfach in einem Graphen vorhanden sein. Er kann zum auffinden von Fehlern benutzt werden. Für den Ablauf des Programms hat der Knotenname keine Auswirkungen.

Eine Kante hat die Darstellung:

(Kantenlabel, Knoten ID1,Knoten ID2)

Das Kantenlabel einer Kante ist ein Integer Wert, jeder Wert kann mehrfach in einem Graphen vorhanden sein.
Die beiden Knoten ID´s sind Integer Werte. Sie dienen dazu Knoten zu identifizieren. Sind beide ID´s im Graphen vorhanden, können diese mit einer Kante verbunden werden.

Eine Knotenmenge wird mit:

<Koten>
<%Knoten>

definiert. Eine Kantenmenge wird mit:

<Kanten>
<%Kanten>

definiert.

Ein Graph hat somit die Form:

<Graph>
<Koten>
<%Knoten>

<Kanten>
<%Kanten>
< %Graph >

Beispiel 1:

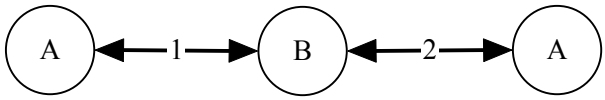



**Abbildung A.1: Ein einfacher Graph**

Der in Abbildung A.1 dargestellt Graph müsste also folgende Definition in der Textdatei besitzen.

<Graph>

<Koten>
(1,A, „beliebiger Text“)
(2,B, „beliebiger Text“)
(3,A, „beliebiger Text“)
<%Knoten>

<Kanten>
(1,1,2)
(2,2,3)
<%Kanten>

< %Graph >

## A.1.2 Definition einer Regelmenge

Eine Regelmenge besteht aus Regeln. Eine Regel wiederum besteht aus der „linken Seite“ einer Regel und der „rechten Seite“ einer Regel. Sowohl die linke, als auch die rechte Seite einer Regel, ist wiederum ein Graph (siehe A.1.1).

<Regelmenge>
<Regel>
<linkeSeiteDerRegel>
„Graph“
<%linkeSeiteDerRegel>

<rechteSeiteDerRegel>
„Graph“
<%rechteSeiteDerRegel>
<%Regel>
<%Regelmenge>

Zu jeder Regel können Einbettungsregeln vorhanden sein, diese werden im Folgenden definiert:

(((Knoten A), (Knoten B), („Label einer Kante, die die Knoten mit Label A,B verbindet“)),
((Knoten A), (Knoten C), („Label einer Kante, die die Knoten mit Label A,C verbindet“)))

Knoten A besitzt die Definition eines Knotens (A.1.1), dabei kann die ID des Knotens beliebig gewählt werden, da nur das Label des Knotens Auswirkungen hat.

Knoten B besitzt ebenfalls die Definition eines Knotens, bei diesem Knoten werden alle Informationen verarbeitet.

Knoten C hat ebenfalls die Definition eines Knotens, alle Informationen des Knotens werden bei der Regelanwendung verarbeitet.

Die Einbettungsregeln werden mit den Schlüsselwörtern „Einbettungsregeln“ und „%Einbettungsregeln“ gekennzeichnet.

Beispiel 2:

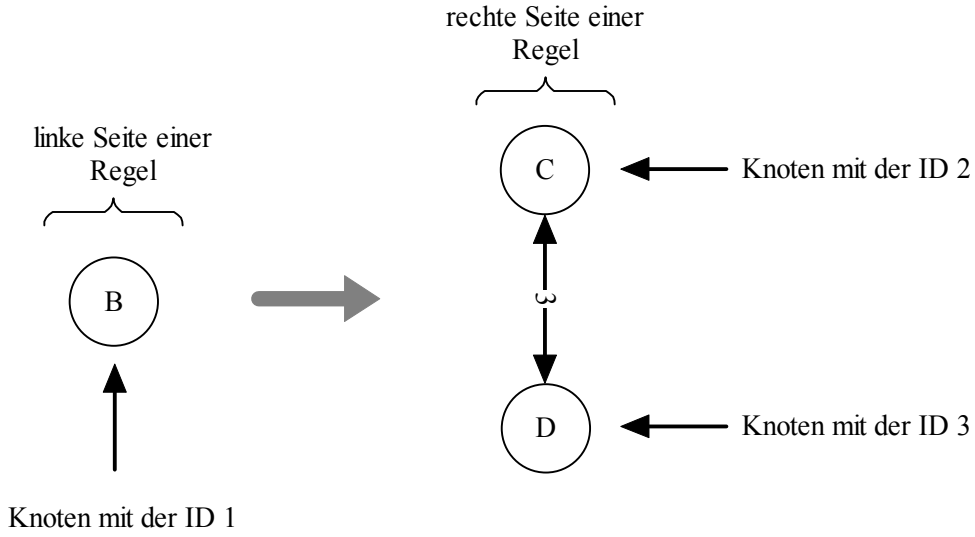



**Abbildung A.2: Eine Regel**

In Abbildung A.2 wird eine Graph Grammatik Regel gezeigt, die einen Knoten mit dem Label B, durch einen Graphen mit zwei Knoten ersetzt. Diese Regel soll auf den Graphen aus Abbildung A.1 angewendet werden.
Die Regel besitzt die Form:

```
<Regelmenge>
        <Regel>
                <linkeSeiteDerRegel>
                        <Graph>
                                <Koten>
                                        (1,B, „beliebiger Text“)
                                <%Knoten>
                        < %Graph >
                <%linkeSeiteDerRegel>
```

```
            <rechteSeiteDerRegel>
                        <Graph>
                                    <Koten>
                                                (2,C, „beliebiger Text“)
                                                (3,D, „beliebiger Text“)
                                    <%Knoten>
                                    <Kanten>
                                                (3,2,3)
                                    <%Kanten>
                        < %Graph >
            <%rechteSeiteDerRegel>
            <Einbettungsregeln>
                        (999,A, „Knoten aus Abbildung 1“),(1,B, „Knoten aus Abbildung A2“),1
                        (999,A, „Knoten aus Abbildung 1“),(2,C, „Knoten aus Abbildung A2“),5

                        (999,A, „Knoten aus Abbildung 1“),(1,B, „Knoten aus Abbildung A2“),1
                        (999,A, „Knoten aus Abbildung 1“),(3,D, „Knoten aus Abbildung A2“),6

                        (999,A, „Knoten aus Abbildung 1“),(1,B, „Knoten aus Abbildung A2“),2
                        (999,A, „Knoten aus Abbildung 1“),(2,C, „Knoten aus Abbildung A2“),7

                        (999,A, „Knoten aus Abbildung 1“),(1,B, „Knoten aus Abbildung A2“),2
                        (999,A, „Knoten aus Abbildung 1“),(2,D, „Knoten aus Abbildung A2“),8
            <%Einbettungsregeln>
      <%Regel>
<%Regelmenge>
```

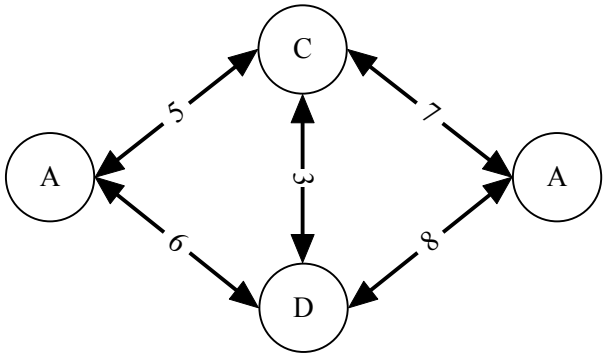



**Abbildung A.3: Graph aus Abbildung A1, nach der Anwendung einer Regel und den dazugehörigen Einbettungsregeln**

In Abbildung A.3 wird der Graph dargestellt, der entsteht, wenn die eben beschriebene Regel auf den Graphen aus Abbildung A.1 angewendet wird. Die neuen Kanten entstehen durch die Einbettungsregeln.

## *A.2 Benötigte Dateien für Simulation 1 und 2*

Damit die beschriebenen Simulationen arbeiten können, benötigen sie verschiedene Dateien. Als erstes sollen die Dateien beschrieben werden, die von dem Programm „Local Balance“ benötigt werden. Die Datei LBKB.TXT (**L**ocal **B**alance **K**nowledge **B**ase) enthält die Definition aller Funktionen und Ressourcen. Die zweite Datei heißt Task.TXT, sie wird von der ersten Simulation erzeugt. Beide Dateien werden benötigt damit „Local Balance“ eine Lösungsdatei erstellen kann.

Die Lösungsdatei heißt LBLösung.TXT und wird gleichzeitig von der zweiten Simulation benötigt. Eine weitere Datei die für die zweite Simulation benötigt wird heißt Umwelt.TXT. Diese Datei enthält alle relevanten Daten, die die Werkhalle betreffen:

- Verfügbare Mitarbeiter und Ausbildungsgrad der Mitarbeiter
- Vorhandene Pumpenanschlüsse (Typ und Anzahl)
- Vorhandene Stromanschlüsse (Typ und Anzahl)
- Größe der Werkhalle ($m^2$)

Sowohl die Task.TXT als auch die LBLösung.TXT werden automatisch durch die Simulationen/Local Balance erzeugt. Die anderen Dateien müssen vom Benutzer zur Verfügung gestellt werden.

# Literatur

[1] Aho, A.; Sethi, R.; Ullmann, J.D.: Compilerbau Teil 1, zweite Auflage, Oldenburg Verlag

[2] Cunis, R.; Günter, A.; Strecker, H.: Das Plakon-Buch, Springer Verlag, 1991

[3] Dangelmaier, W.: Produktion und Logistik, Systemkonzepte und Modelle (University of Paderborn, Germany)

[4] Dangelmaier, W.: Fertigungsplanung (University of Paderborn, Germany), 1999

[5] Domschke; Drexl: Einführung in Operations Research, 1990

[6] Heinrich, M.: Ressourcenorientierte Modellierung als Basis modularer Technischer Systeme. In: Beiträge zum 5. Workshop Planen und Konfigurieren, 1991

[7] Kleine Büning: Vorlesung „Wissensbasierte Systeme Teil 1+2“, Vorlesungsunterlagen des Fachbereiches 17 der Universität Paderborn, Dozent: Prof. Dr. Kleine Büning, 2000

[8] Nehmer, J.; Sturm, P.: Systemsoftware, Grundlagen moderner Betriebssysteme, 1998

[9] Özalp, S.: Lokale Konzepte in der ressourcenorientierten Konfigurierung, Diplomarbeit des Fachbereiches 17 der Universität Paderborn, Prüfer: Prof. Dr. Kleine Büning

[10] Rosenberg, O.: Produktionsvollzugsplanung. Vorlesungsscript Hauptstudium Betriebswirtschaftslehre Universität Paderborn, 2001

[11] Rosenberg, O.: Produktionslogistik. Vorlesungsscript Hauptstudium Betriebswirtschaftslehre Universität Paderborn, 2000

[12] Stein, B.M.: Functional Models in Configuration Systems, from the Department of Mathematics and Computer Science of the University of Paderborn, Germany.
The Accepted Dissertation of Benno Maria Stein, in order to Obtain the Academic Degree of Dr. rer. Nat. 1995

[13] Stein, B.M: Model Construction in Analysis and Synthesis Tasks (Habilitation Theses, University of Paderborn, Germany)

Literatur zum Thema Java:

[14] http://www.galileocomputing.de/openbook/javainsel2/java_010000.htm
Allgemeine Java Informationen (Threads, GUI, Reflection)

[15] http://www.ibiblio.org/javafaq/javatutorial.html
Allgemeine Java Informationen (Variablen, Programmierung)

[16] http://www.pi.informatik.tu-darmstadt.de/joop98/Folien/06oo/
Datentypen in Java (Keller, Listen, usw.)

[17] http://java.sun.com/docs/books/tutorial/reflect/index.html

Reflection Tutorial: Wie können nachträglich bekannte Klassen eingeladen werden

[18] http://java.sun.com/docs/books/tutorial/uiswing/index.html
GUI Tutorial: Wie funktioniert Swing

[19] http://java.sun.com/docs/books/tutorial/essential/threads
Thread Tutorial: Was sind Threads und wie benutzt man sie.