Universität
Gesamthochschule Paderborn

Fachbereich 17 · Mathematik - Informatik

Wissensbasiertes Analysieren und Optimieren von Systemspezifikationen

Diplomarbeit

für den integrierten Studiengang Informatik im Rahmen des Hauptstudiums II

von
Thomas Bölting

Vorgelegt bei
Prof. Dr. F. J. Rammig
und
Prof. Dr. W. Hauenschild

September 1996

Diese Arbeit wurde im C-LAB, einer Kooperation der Universität-Gesamthochschule Paderborn und der Siemens-Nixdorf-Informationssysteme AG, erstellt. Mein Dank gilt Herrn Dr. B. Kleinjohann für die interessante Aufgabenstellung und die tatkräftige Unterstützung, sowie Herrn Prof. Dr. F.–J. Rammig für die Betreuung und Herrn Prof. Dr. W. Hauenschild für die Zweitkorrektur der Arbeit. Außerdem möchte ich Herrn Dipl.–Inf. J. Tacken für seine wertvollen Hinweise und Anregungen danken und Herrn Dipl.–Inf. Oliver Kluge, der mir hilfreiche Hinweise bezüglich MuPAD gab. Schließlich danke ich allen Mitarbeitern des C-LAB, die mir im Verlauf dieser Arbeit zur Seite standen und meinen Eltern, die mir dieses Studium ermöglichten.

Ich versichere, die vorliegende Diplomarbeit selbständig angefertigt, alle benutzten Hilfsmittel vollständig angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer direkt oder mit Abänderungen entnommen wurde.
Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
Paderborn, August 1996

Thomas Bölting

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Heutzutage wird man in vielen Bereichen mit Optimierungsproblemen konfrontiert, da immer mehr darauf geachtet werden muß, daß Ressourcen eingespart oder besser genutzt werden. Die Reduzierung anfallender Kosten steht hierbei meistens im Vordergrund.
Beispielsweise wird der Konkurrenzkampf auf dem Wirtschaftsmarkt immer härter. Deshalb muß jede Firma darauf bedacht sein, daß sie für die Probleme, die sich bei der Entwicklung und Produktion ergeben, optimale Lösungen findet, um sich auch langfristig am Markt behaupten zu können. Dies betrifft nicht nur den Ablauf des Arbeitsprozesses, sondern auch den optimalen Einsatz der Werkstoffe und des Personals. Es muß also das Ziel sein, durch den Einsatz der Methoden des *Operation-Research* einen maximalen Gewinn mit minimalen Kosten zu erzielen.
Zu diesen Optimierungsproblemen gehören beispielsweise aus dem Bereich des *Workflow-Managements* die Produktions-, die Vertriebs-, die Personal- und die Fertigungsplanung. Zu einer guten Fertigungsplanung gehört, daß die Firma ihre Maschinen und Werkzeuge optimal auslastet und der Materialverbrauch so niedrig wie möglich gehalten wird. Um den optimalen Einsatz der Ressourcen zu erreichen, werden beispielsweise Heuristiken oder *Branch and Bound*-Verfahren eingesetzt.
Aber nicht nur im Bereich der Wirtschaft treten Optimierungsprobleme auf. Ein weiteres Szenario, auf das auch in [aviv 95] eingegangen wird, wäre die optimale Nutzung unseres Verkehrsnetzes. Deutschland ist mittlerweile die Drehscheibe des europäischen Transitverkehrs. Da die Ausbaubarkeit des Straßen- und Schienennetzes nicht beliebig fortsetzbar ist, muß nach Wegen gesucht werden, wie man diese Streckennetze optimal nutzen kann. Es stehen also nur begrenzte Verkehrskapazitäten zur Verfügung, die nach betriebswirtschaflichen Gesichtspunkten optimal genutzt werden sollen bzw. müssen, um ein Zusammenbrechen des Verkehrs zu verhindern. Die sonst folgenden erheblichen wirtschaftlichen Einbußen betroffener Firmen dürfen auch nicht außer acht gelassen werden.
Auch im Bereich der Hardware-Entwicklung ergeben sich diverse Probleme, die einer optimalen Lösung bedürfen. Hierzu gehören das Plazieren und Verbinden der Bauteile. Je nach Schaltung kann es beispielsweise wesentlich sein, daß die Leiterbahnen möglichst kurz oder die benötigte Fläche einer Schaltung und die Anzahl der verwendeten Bauteile möglichst gering sind. Möglicherweise steht eine bestimmte Bibliothek mit Bauteilen, die zur Produktion der Schaltung verwendet werden sollen, zur Verfügung. Ist beispieslweise die Geschwindigkeit einer Schaltung der wichtigste Aspekt, so können

Bausteine zum Einsatz kommen, die zu hohen Produktionskosten führen.
Die gerade beschriebenen Beispiele sind nur ein sehr kleiner Ausschnitt aus der großen Menge der Optimierungsprobleme. In [litt 92] wird ein Überblick über eine große Anzahl von Optimierungsproblemen gegeben. Mögliche Lösungsmethoden werden dort auch angegeben. Wie das Beispiel der Hardware-Entwicklung gezeigt hat, muß Optimierung nicht immer unbedingt Reduktion bedeuten. Es kann sich auch um eine Expansion handeln, die nötig sein kann, damit bestimmte existierende Ressourcen eingesetzt oder andere Kriterien erfüllt werden können.

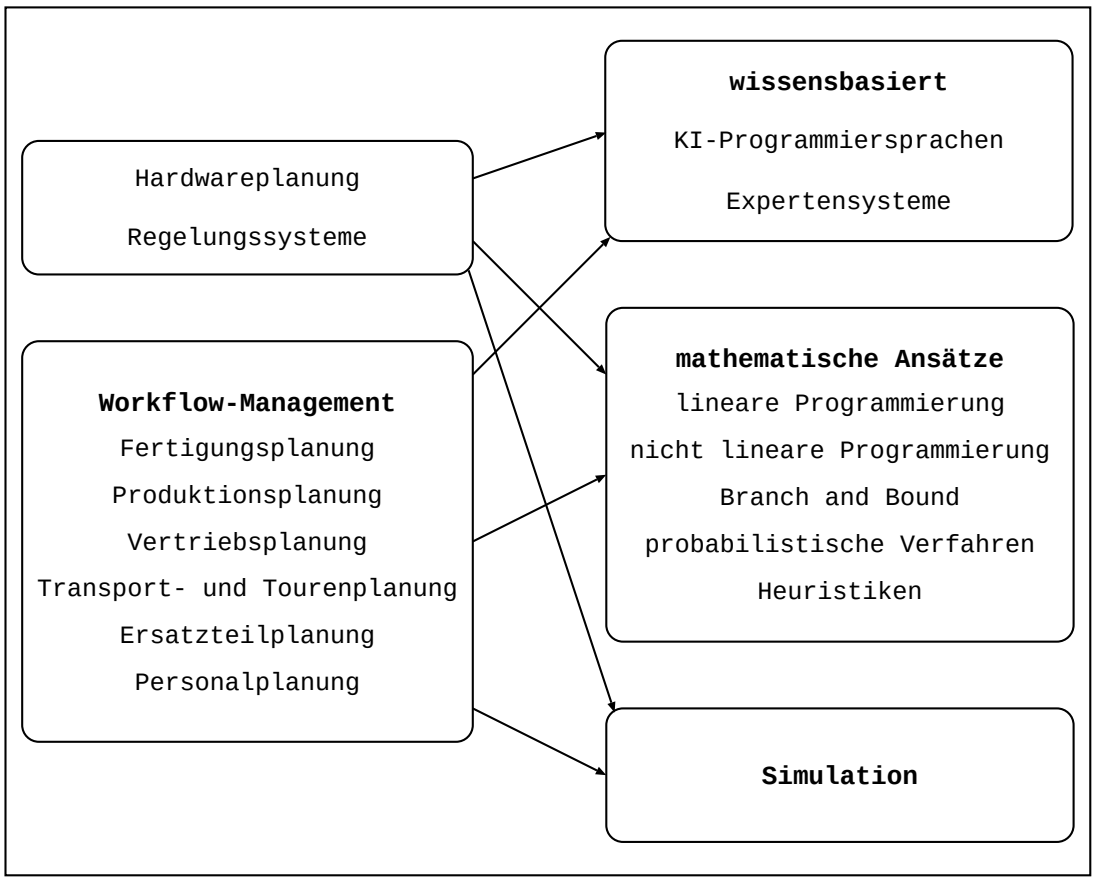

Abbildung 1.1: Optimierungsprobleme und mögliche Lösungsverfahren

Ausgehend von dem Prozeß, den man optimieren möchte, muß man eine geeignete Methode wählen, die auch zu einer Optimierung führt. Die Wahl des Verfahrens ist mit ausschlaggebend für das Optimierungergebnis und legt auch fest, wie das System modelliert werden muß.
Wenn wir noch einmal das Problem der Verkehrsplanung betrachten, so muß man sich überlegen, wie eine mögliche Modellierung aussehen müßte, damit ein Optimierungprozeß in Gang gesetzt werden könnte. Beispielsweise wäre die Abbildung auf eine lineares Gleichungssystem vorstellbar. Durch die lineare Optimierung könnte dann für die einzelnen Parameter bestimmt werden, wo Gewinne bzw. Verbesserungen zu erwarten sind. Man hätte den entscheidenden Nachteil, daß mit einem linearen Gleichungssystem keine exakte Abbildung der Realität möglich ist. Ein weiterer Nachteil ist, daß

die Modellierung des Gleichungssystems schon fest in einem Werkzeug vorgegeben werden müßte, da sonst die Handhabbarkeit stark in Frage gestellt wäre. Es wäre auch möglich, das Optimierungsproblem analytisch zu lösen. Da sich analytische Ansätze i. allg. nur für Systeme geringer Komplexität eignen, würde man die Anzahl der in Betracht kommenden Systeme stark einschränken. Die feste Vorgabe einer Heuristik bietet auch keine Alternative, da diese sich meistens nur für eine spezielle Art von Problemen eignet. Werden mehrere Heuristiken bereitgestellt, so muß zusätzlich noch entschieden werden, welche eingesetzt werden soll.
Auf der linken Seite der Abbildung 1.1 auf Seite 6 sind einige Optimierungsprobleme und auf der rechten Seite mögliche Methoden, die zur Optimierung verwendet werden können, zu sehen. Die Pfeile, die von den Problemen zu den Lösungsmethoden verlaufen, sollen andeuten, daß in vielen Fällen die Möglichkeit besteht, für ein Problem verschiedene Verfahren zur Optimierung einzusetzen, wobei sich für ein gegebenes Problem ein Verfahren mehr oder weniger gut eignen kann. Zusätzlich zu den mathematischen Verfahren existiert die Möglichkeit, durch eine Simulation Schwachstellen in einem System aufzudecken. Beispielsweise stellt man durch die Simulation eines Fertigungsprozesses leicht fest, an welchen Stellen Ressourcen fehlen oder überflüssig sind.
In dieser Arbeit soll eine Alternative zu den existierenden Verfahren vorgestellt werden. Für die Modellierung des Systems, das optimiert werden soll, stehen erweiterte Prädikat/Transitions-Netze zur Verfügung. Der Ansatz zur Optimierung basiert darauf, daß ein Experte, der über eine große Erfahrung verfügt, die Möglichkeit hat, die Identifikation und Behebung der Schwachstellen eines Systems von Hand vorzunehmen. Da sich das Expertenwissen nicht eindeutig definieren läßt, wird es auch als unscharfes Wissen bezeichnet, welches sich mit Methoden aus der künstlichen Intelligenz (KI) beschreiben läßt. Hier finden Expertensysteme ihre Anwendung. Mit solchen Werkzeugen hat auch ein Anwender, der nicht über die Fähigkeiten eines hochspezialisierten Experten verfügt, die Möglichkeit für sein spezielles Problem, das er natürlich vorher in geeigneter Notation formulieren muß, eine Lösung zu finden.
Um die bereits beschriebenen Probleme zu umgehen, müssen sowohl die Modellierung und Simulation des Systems als auch die Analyse und Optimierung so flexibel und effektiv wie möglich gestaltet werden. Hierzu wird die im C-LAB entwickelete Prädikat/Transitions-Netz-Entwicklungsumgebung, die auch in [taklkl 96] beschrieben wird, verwendet und um eine wissensbasierte Komponente erweitert.
Auf dem Gebiet der KI existieren mehrere Möglichkeiten, um Wissen in einer geeigneten Form abzulegen und abzufragen. Es werden einerseits die KI-Programmiersprachen, die am häufigsten verwendeten sind Prolog und Lisp, und andererseits die Expertensysteme verwendet. In dieser Arbeit kommen die KI-Programmiersprachen nicht in Frage, da die Eigenschaften, die Expertensysteme bieten, benötigt werden. Prolog und Lisp werden häufig zur Realisierung solcher Expertensysteme eingesetzt. Nun stellt sich noch die Frage, welches Expertensystem die geforderten Aufgaben erfüllen kann. Im Laufe der Entwicklung der Expertensysteme hat man sich von den Spezialentwicklungen gelöst, die ersten Expertensysteme waren nur für ein Spezialgebiet geeignet, und ist dazu übergegangen, sogenannte Expertensystem-Shells zu entwickeln, wobei auch diese auf ganz bestimmte Probleme oder Problemklassen zugeschnitten sein können. I. allg. sind die Expertensystem-Shells aber flexibel für mehrere Bereiche einsetzbar. Die Begriffe Expertensystem und Expertensystem-Shell werden in der Arbeit gleichbedeutend verwendet, da auch in der Literatur keine exakte Trennung vollzogen wird.

Das Expertensystem, daß für die hier geforderte Anwendung in Frage kommt, muß in die existierende Prädikat/Transitions-Netz-Einwicklungsumgebung integrierbar sein. Wesentlich ist auch, daß der gegebene Funktionsumfang des Expertensystems erweiterbar ist, da spezielle Aufgaben durchführbar sein müssen, die nicht unmittelbar zu den Aufgaben eines Expertensystems gehören. Weiterhin muß sich die Form der Wissensdarstellung und die Problemlösungsstrategie, die das Expertensystem einsetzt, zur Lösung von Optimierungsproblemen eignen. Detailliert wird auf diese Aspekte im Kapitel 2.2.3 und den folgenden Kapiteln eingegangen.
Im C-LAB wird im Rahmen des Projekts Intelligent Framework Services (IFS) die *hybrid knowledge description language* HyKL (siehe auch [lauf 95] und [drkr 95]) entwickelt, mit der unter anderem die Modellierung eines Expertensystems möglich ist. Da HyKL noch in der Entwicklung ist und benötigte Mechanismen zum Zeitpunkt dieser Arbeit noch nicht zur Verfügung standen, wird es nicht eingesetzt. HyKL hätte den Vorteil gehabt, daß die Daten direkt in der im C-LAB entwickelten Datenbank (Siframe/OMS) hätten abgelegt werden können. Die Daten der Prädikat/Transitions-Netz-Entwicklungsumgebung werden mit Hilfe des COM-Moduls abgelegt. Es handelt sich hierbei um einen C++-Objekt-Manager (siehe auch [push 93]), der eine Bibliothek von C++ Klassen und zugehörigen Methoden zur Verfügung stellt, die es ermöglichen, persistente Daten strukturiert zu verwalten. Mit einer entsprechenden Schnittstelle, die die Umsetzung zwischen OMS und COM-Modul realisiert hätte, hätte der Datentransfer, der durch das für die Arbeit verwendete Expertensystem nötig wird, wesentlich verringert werden können.

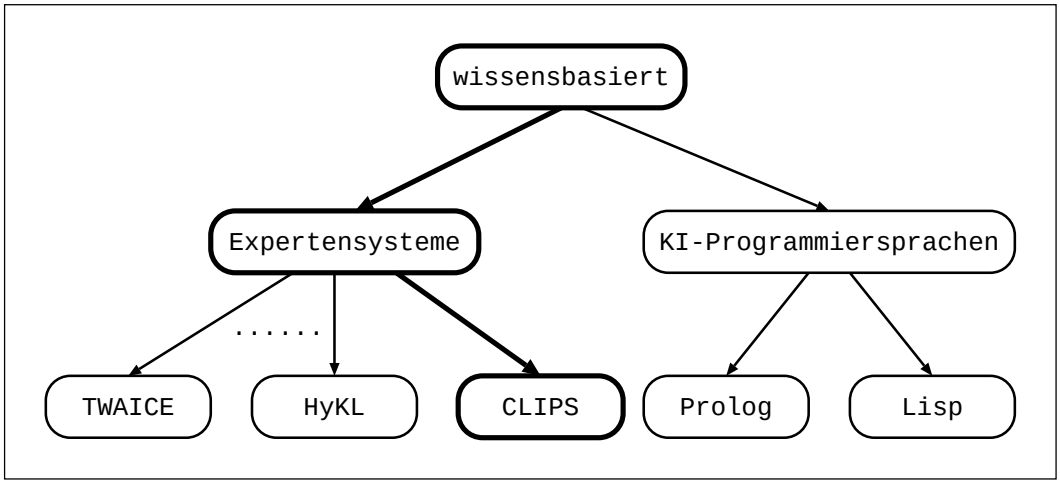

Abbildung 1.2: Die Wahl des wissensbasierten Werkzeugs

Es existieren diverse Expertensystem, die sich für unterschiedliche Aufgaben eignen. Bei TWAICE (siehe auch [savo 85]) handelt es sich um eine Expertensystem-Shell, die von der Nixdorf AG entwickelt wurde und für die Bereiche Klassifikation und Diagnose eingesetzt wird. Ein weiteres Beispiel ist MYCIN (siehe auch [jack 87]), es handelt sich um eines der ersten Expertensystem, das für die Diagnose im Gesundheitswesen eingestzt wird. Die Wahl ist letztendlich auf das Expertensystem CLIPS (C Language Integrated Production System) gefallen, da es sich für das Problem der Optimierung gut eignet. Es wird im Kapitel 2.3 vorgestellt. CLIPS, das von der NASA entwickelt

wurde, hat den Vorteil, daß es als C-Quellcode vorliegt und somit von den Schnittstelleneigenschaften und bezüglich der Erweiterbarkeit gute Eigenschaften bietet. Man hat beispielsweise die Möglichkeit, den Funktionsumfang durch eigene C-Funktionen zu erweitern. Diese Erweiterbarkeit ist für die Arbeit wesentlich, da zusätzliche Funktionen, die beispielsweise zur Verwaltung von Mengen benötigt werden, zu CLIPS hinzugefügt werden müssen.
Da beim Mustervergleichsprozeß eines Expertensystems im Normalfall, eine exakte Übereinstimmung der Muster gegeben sein muß, dies betrifft sowohl das Muster an sich als auch die Position, existiert keine Möglichkeit mathematische Ausdrücke zu vergleichen. Dies gilt auch für CLIPS. Beispielsweise wäre $2 * x$ mathematisch identisch mit $x + x$, ein normaler Mustervergleich würde dies aber nicht erkennen. Auch $x + y$ wäre bei einem normalen Mustervergleich verschieden von $y + x$, da die Position der Variablen vertauscht wurde. Um dieses Problem zu beheben, ist das an der Universität-GH Paderborn entwickelte MuPAD (Multi Processing Algebra Data Tool) mit in die Architektur eingebunden worden. MuPAD wird in dieser Arbeit für Termersetzungen, Berechnungen und das Normieren von Ausdrücken eingesetzt. Eine Einführung zu MuPAD ist beispielsweise im Benutzerhandbuch [fuch 93] und im Tutorial [fuch 94] zu finden.
Wird ein System optimiert, so bedeutet dies, daß das mit dem Prädikat/Transitions-Netz-Editor modellierte System nach der Optimierung verändert vorliegt und auch dargestellt wird. Bei der Analyse werden mögliche Optimierungen angezeigt. Die Ergebnisse können beim Erstellen von Transformationen eingesetzt werden, um beispielsweise zu überprüfen, ob die Transformationen einwandfrei arbeiten.

1.1 Zielsetzung

Der Prädikat/Transitions-Netz-Editor bietet alle Möglichkeiten, um ein System als Prädikat/Transitions-Netz zu modellieren. Bevor das System mit dem Editor modelliert werden kann, muß feststehen, wie die Umsetzung in ein Prädikat/Transitions-Netz aussieht. Die im Kapitel 2.1 beschriebenen Grundlagen sollen als Einführung in die Thematik der Prädikat/Transitions-Netze dienen. Wie eine Umsetzung eines Systems oder einer Klasse von Systemen in ein Prädikat/Transitions-Netz aussieht, muß für den jeweiligen Fall untersucht werden und wird in dieser Arbeit nicht behandelt.
Ausgehend davon, daß das System mit dem Prädikat/Transitions-Netz-Editor modelliert wurde, wird ein Formalismus benötigt, mit dem Transformationen formulierbar sind. Hierzu wird im Verlauf der Arbeit eine Sprache entwickelt, die zum Formulieren geeigneter Transformationen verwendet werden kann. Die Notation der Transformationssprache ist so nah wie möglich an die Notation der Prädikat/Transitions-Netze angelehnt, um die Formulierung der Transformationen zu erleichtern. Eine Transformation kann aus mehreren Regeln bestehen, die sich jeweils in einen Bedingungsblock und einen Aktionsblock unterteilen. Im Bedingungsblock wird beschrieben, welche Eigenschaften das Prädikat/Transitions-Netz oder ein Teilnetz erfüllen muß, damit die Optimierung im Aktionsblock ausgeführt werden kann. Es können beispielsweise Verbindungsstrukturen oder Annotationen abgefragt werden. Als Optimierung ist z. B. das Löschen oder bei einer Expansion das Hinzufügen von Objekten möglich. Details zu den Transformationen findet man im Kapitel 3.

Damit die Aktionen, die letztendlich zu einem optimierten System führen sollen, ausgeführt werden können, wird die Prädikat/Transitions-Netz-Entwicklungsumgebung und insbesondere der Prädikat/Transitions-Netz-Editor um die entsprechende Funktionalität erweitert. Hierzu gehört z. B. das Erweitern des Menüs, über das die einzelnen Punkte angewählt werden können, sowie das Einbinden der benötigten Werkzeuge und deren Erweiterungen. Zusätzlich werden mehrere Parser benötigt, die eine Konvertierung der Daten durchführen müssen, damit der Informationsaustausch zwischen dem Prädikat/Transtions-Netz-Editor, CLIPS und MuPAD möglich wird. Detailliert werden die Erweiterungen und der Ablauf des Optimierungsprozesses im Kapitel 4 erläutert.

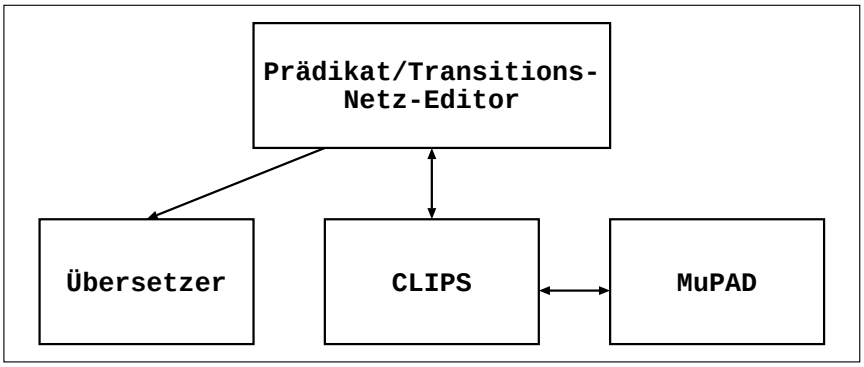

Abbildung 1.3: Die beteiligten Werkzeuge

Die Kernpunkte der Arbeit sind die Koppelung des Prädikat/Transitions-Netz-Editors mit dem wissensbasierten Werkzeug und das Einbinden von MuPAD. Weitere wichtige Punkte sind die Einführung der Sprache, mit der die Transformationen formuliert werden können, sowie die Realisierung der benötigten Konvertierungsmechanismen, die den Informationsaustausch zwischen den verschiedenen Werkzeugen erst ermöglichen. Zusätzlich wird beschrieben, welche Kriterien beim Formulieren der Transformationen beachtet werden müssen, damit die Optimierung in einer akzeptablen Zeit durchgeführt werden kann.

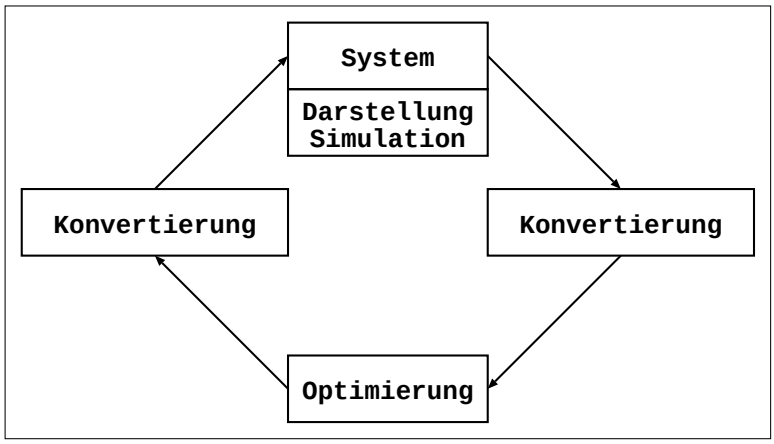

Abbildung 1.4: Der Optimierungsprozeß

In Abbildung 1.3 sind die wichtigsten beteiligten Werkzeuge und der Informationsfluß

zwischen diesen Werkzeugen dargestellt. Die Architektur wird im Kapitel 4.1 erläutert. Damit das Expertensystem die Transformationen verarbeiten kann, müssen sie in eine für CLIPS verständliche Notation überführt werden. Hierzu wird der Prädikat/Transitions-Netz-Editor um den dafür benötigten Übersetzer erweitert.
Die Abbildung 1.4 auf der Seite 10 zeigt den Ablauf, der zur Optiemierung eines Systems führt. Zuerst wird das System als Prädikat/Transitions-Netz dargestellt. Anschließend bietet der Prädikat/Transtitions-Netz-Editor die Möglichkeit der Simulation. Soll das System optimiert werden, so muß eine Konvertierung der Daten in ein für CLIPS verständliches Format durchgeführt werden. Weiterhin werden Die Transformationen, die mit der Transformationssparche erstellt und anschließend übersetzt wurden, benötigt. Ist der Optimierungsprozeß abgeschlossen, so werden die Daten, die sich ergeben haben, wieder konvertiert und dargestellt. Eine Simulation des resultierenden Systems ist anschließend wieder möglich.

1.2 Struktur der Arbeit

Um die Prädikat/Transitions-Netz-Entwicklungsumgebung um eine wissensbasierte Optimierungskomponente zu erweitern, wird wie folgt vorgegangen. Im Kapitel 2 werden zuerst die Modellierungsgrundlagen beschrieben. Hierbei handelt es sich zum einen um die Prädikat/Transitions-Netze, zum anderen werden die grundlegendsten Begriffe der Expertensysteme erläutert. Das Kapitel wird mit einer Beschreibung des Expertensystems CLIPS, das in dieser Arbeit verwendet wird, abgeschlossen. Das Kapitel 3 befaßt sich mit der Beschreibung der Transformationssprache. Es wird zuerst auf die benötigten Datentypen eingegangen und anschließend werden die einzelnen Konstrukte erklärt. Zusätzlich wird noch auf die Termersetzung und die Berechnungen eingegangen, die bei Optimierungen, bei denen Annotationen "verschmolzen" werden, benötigt werden. Den Abschluß des Kapitels bildet die Beschreibung eines Kontrollmeachnismus, mit dem die Abarbeitungsreihenfolge der Transformationen beeinflußt werden kann. Im Kapitel 4 wird auf die Punkte eingegangen, die zusätzlich zur Transformationssprache realisiert wurden. Es geht um die Erweiterung des Prädikat/Transitions-Netz-Editors und um die Architektur, zu der auch die Koppelung des Expertensystems an den Prädikat/Transitions-Netz-Editor gehört. Weiterhin werden Erweiterungen bezüglich CLIPS und der implizite Regelsatz, der unter anderem die Aufgabe hat das Prädikat/Transitions-Netz in einem konsistenten Zustand zu halten, beschrieben. Dieser Regelsatz wird zusätzlich zu den Transformationen, die zur Optimierung des Systems führen sollen, benötigt. Da er immer benötigt wird, wird er bei der Initialisierung von CLIPS geladen. Sowohl die Erweiterungen von CLIPS als auch der implizite Regelsatz werden benötigt, um eine Umsetzung aller Eigenschaften der Transformationssprache zu ermöglichen. Zwei Anwendungsbeispiele, werden im Kapitel 5 beschrieben. Eine Zusammenfassung und ein Ausblick im Kapitel 6 schließen die Arbeit ab.
Im Anhang findet man die bei der Transformationssprache verwendeten Datentypen in der CLIPS-Notation und die formale Grammatik der Transformationssprache in EBNF. Weiterhin werden der implizite Regelsatz und die Transformationen zu den Anwendungsbeispielen, die im Kapitel 5 beschrieben werden, angegeben.

Kapitel 2

Modellierungsgrundlagen

Die Grundlagen der Prädikat/Transitions-Netze, die zur Modellierung eines Systems eingesetzt werden, werden im ersten Abschnitt dieses Kapitels erläutert. Im zweiten Abschnitt wird ein Einblick in die Arbeitsweise der Expertensysteme gegeben, damit ihre Eigenschaften und Fähigkeiten bei der Formulierung der Transformationen richtig eingeschätzt und damit auch sinnvoll eingesetzt werden können. Dies betrifft im wesentlichen die effektive Nutzung des Kontrollmechanismus zur Beeinflussung der Abarbeitungsreihenfolge der Transformationen, der Teil der Transformationssprache ist. Abschließend werden noch benötigte Grundkenntnisse bezüglich CLIPS beschrieben.
Bei der Beschreibung der Prädikat/Transitions-Netze wird aufgezeigt, welche Mächtigkeit zur Modellierung der Systeme zur Verfügung steht. Es wird nicht auf den eigentlichen Umsetzungprozeß „Wie wandele ich das System, das ich optimieren möchte, in ein Prädikat/Transitions-Netz um?“ eingegangen.
Der Entwurf von Systemen wird in Abhängigkeit von deren Größe immer komplizierter, so daß der Einsatz von unterstützenden Werkzeugen erforderlich wird. Beispielsweise wird in [klei 93] und [tack 92] auf die Steuerung und Überwachung von Entwurfsystemen auf der Basis erweiterter Prädikat/Transitions-Netze eingegangen. Der systematische Entwurf digitaler Systeme (Hardware-Synthese) wird unter anderem in [ramm 89] und [klei 94] beschrieben. In [kirs 92] wird ein Sprache zum schnellen Entwurf komplexer Systeme zur Transformation hierarchischer Petrinezte entwickelt. Ihre Anwendung findet diese Sprache im Bereich der Hardware-Synthese. Die Umsetzung von Differentialgleichungen in Prädikat/Transitions-Netze, die bei einem Anwendungsbeispiel im Kapitel 5.2 noch zum Tragen kommt, wird in [brkl 93] behandelt.
Auf welche Aspekte bei der Umsetzung eines Systems geachtet werden muß, wird im Kapitel 2.2 erläutert. Ist das Modellierungsproblem gelöst, so müssen noch geeignete Transformationen formuliert werden. Hierzu ist es von Nutzen, wenn man den Ablaufzyklus des Optimierungsprozesses kennt. Zur Beschreibung dieser Grundkenntnisse, dienen im allgemeinen Kapitel 2.2 und im speziellen Kapitel 2.3.

2.1 Erweiterte Prädikat/Transitions-Netze

Der folgende Abschnitt soll als Einführung in die Thematik der erweiterten Prädikat/Transitions-Netze dienen. Im Rahmen dieser Einführung wird zuerst allgemein auf Petrinetze und deren Entwicklungsphasen sowie deren Einsatzgebiete eingegangen und

anschließend auf die Erweiterungen, die ein Petrinetz zu einem Prädikat/Transitions-Netz machen.

2.1.1 Entwicklung und Anwendung

Die Petrinetze, die im Rahmen der Dissertation [petr 62] von Dr. Petri im Jahre 1962 definiert wurden, eignen sich zur graphischen Darstellung von Systemen. Sie eignen sich besonders gut, um interagierende, nebenläufige Komponenten von Systmen zu modellieren. Sie sind Anwendungsunabhängig wie auch State Charts [hare 78] und Daten-Fluß-Graphen [denn 85]. Alle diese Formalismen haben gemeinsam, daß sie jeweils einen vordefinierten Satz von Symbolen bieten, mit dem es möglich ist, komplexe Strukturen zu konstruieren. Da es keine universelle Sprache zur graphischen Beschreibung von Systemen gibt, muß man sich je nach gegebenem Problem für eine Variante entscheiden. Die Einsatzgebiete der Petrinetze sind vielfältig. Sie werden beispielsweise beim Software Engineering [reis 91], bei Datenbanken [voss 87] und bei der Spezifikation und Verifikation von Produktsystemen [vale 87] eingesetzt. Eine grundlegende Einführung in die Thematik der Petrinetze ist in [reis 91] und [pete 81] zu finden. Während in [reis 91] das Thema von der theoretischen Seite aus angegangen wird, wird in [pete 81] mehr die praktische Seite beleuchtet. In beiden Büchern sind diverse Beispiele zu finden. In [pete 81] wird beispielsweise die Modellierung von Hardware erläutert. Es finden sich dort auch Beispiele zu den Gebieten der Nuklear-Physik, Soziologie, Biologie und Astronomie. Eine zusammenfassende Darstellung der Einsatzmöglichkteiten ist in [ager 79] zu finden. Die erweiterte Prädikat/Transitions-Netze, die zur Modellierung der Systeme bzw. Systemspezifikationen in dieser Arbeit verwendet werden, basieren auf den Petrinetzen. Die grundlegenden Definitionen zum Thema der Prädikat/Transitions-Netze wurden von Genrich und Lautenbach in [gela 81] eingeführt. Ein formaler Ansatz zur Beschreibung von Prädikat/Transitions-Netzen ist auch in [klei 93] und[tack 92] zu finden. Die bei der Prädikat/Transitions-Netz-Entwicklungs-Umgebung und im speziellen beim Prädikat/Transitions-Netz-Editor zum Einsatz kommenden Annotationen werden detailiert in [rust 95] beschrieben.

2.1.2 Petrinetze

Die Notation der Petrinetze, die im folgenden beschrieben werden, ist [pete 81] entnommen. Ein Petrinetzgraph ist ein gerichteter bipartiter Graph, der aus Knoten und Kanten besteht. Der Begriff Netz wird im folgenden synonym mit Graph verwendet. Man unterscheidet bei den Knoten zwischen Stellen (*places*) und Transitionen (*transitions*). Die Menge der Stellen wird mit P und die Menge der Transitionen mit T bezeichnet. Es gilt $P \cap T = \emptyset$. Bei einer Kante (*edge*) unterscheidet man zwischen Eingangs- und Ausgangskante, wobei der Bezugspunkt die Transitionen sind. Die Eingangskanten werden mit I und die Ausgangskanten mit O bezeichnet. Es gilt $I \subseteq P \times T$ und $O \subseteq T \times P$. Bei den Mengen I und O handelt es sich um Multimengen, d. h. Elemente können mehrfach vorhanden sein.
Die Marken (*token*), die benötigt werden, um die Semantik eines Netzes verändern zu können, werden in den Stellen abgelegt. Eine Marke kann entweder als Punkt oder die Anzahl aller Marken einer Stelle als Zahl dargestellt werden. Die Punkte bzw. die Zahl wird innheralb des Kreises der entsprechenden Stelle plaziert.

Die Transitionen werden zum kontrollierten Fluß der Marken im Netz benötigt. Für das Schalten einer Transition t ist es erforderlich, daß für alle Stellen p mit $(p,t) \in I$ gilt, daß p markiert ist. Verlaufen mehrere Kanten von p nach t, so muß p pro Kante eine Marke enthalten. Ist diese Bedingung erfüllt, so wird t als aktivierbar (*enabled*) bezeichnet. Ist t aktiv, so werden von jeder Stelle p mit $(p,t) \in I$ soviele Marken, wie Kanten zwischen p und t existieren, genommen, und zu jeder Stelle p' mit $(t,p') \in O$ werden soviele Marken hinzugfügt, wie Kanten von t nach p' verlaufen.
Bei der graphischen Darstellung eines Petrinetzes werden die Stellen als Kreise, die Transitionen als Rechtecke und die Kanten als Pfeile gezeichnet. Die Richtung wird durch die Pfeilspitze angezeigt.
Der Zustand des Netzes, der vor dem ersten Schaltvorgang existiert, wird durch die initiale Markierung m und den Status der Transitionen beschrieben. Die initiale Markierung beschreibt, in welchen Stellen sich wieviele Marken befinden. Wenn mindestens eine aktivierbare Transition existiert, so kann das Netz durch einen Schaltvorgang in eine Markierung m' überführt werden. Die Ausführung mehrerer Schaltvorgänge wird als Schaltfolge bezeichnet. Ist nach einer endlichen Anzahl von Schaltvorgängen kein weiterer Schaltvorgang mehr möglich ist, so ist die Schaltfoge endlich, sonst unendlich.
Das dynamische Verhalten muß von einem Simulationsalgorithmus koordiniert werden, der nach einem bestimmten Schema überprüft, welche Transitionen aktivierbar sind und somit schalten können. Ist die Auswahl auf eine bestimmte Transition gefallen, so wird diese aktiviert, und es muß noch für die Demarkierung der Eingangs-Stellen und die Markierung des Ausgangs-Stellen gesorgt werden. Diese Schritte können bei einer unendlichen Schaltfolge beliebig oft wiederholt werden. Bei einer endlichen Schaltfolge terminiert dieser Prozeß, nachdem alle möglichen Schaltvorgänge durchgeführt wurden. Es hängt nun vom verwendeten Simulationsalgorithmus ab, ob die Simulation schrittweise oder nur komplett durchgeführt werden kann. Bei der Schrittweisen Simulation hat man den Vorteil, das man auch die Zwischenschritte betrachten kann. Der Simultationsalgorithmus, der bei dem Prädikat/Transitions-Netz-Entwicklung-Umgebung verwendet wird, erlaubt beide Möglichkeiten. Auch das Abbrechen einer laufenden Simulation ist möglich.

Die Punkt-Operatoren

Um auf dem Graphen eines Petrinetzes navigieren zu können, werden die Punkt-Operatoren benötigt. Mit deren Hilfe können für ein Element p aus P oder t aus T alle Elemente bestimmt werden, von denen aus eine Kante zu p bzw. t hinführt und alle Elemente zu denen eine Kante von p bzw. t hinführt. Es gilt ${}^\bullet p = \{t \in T | (t,p) \in O\}$ ist die Menge der Eingangs-Transitionen von p und $p^\bullet = \{t \in T | (p,t) \in I\}$ ist die Menge der Ausgangs-Transitionen von p. Weiterhin gilt ${}^\bullet t = \{p \in P | (p,t) \in I\}$ ist die Menge der Eingangs-Stellen von t und $t^\bullet = \{p \in P | (t,p) \in O\}$ ist die Menge der Ausgangs-Stellen von t.

Das Beispiel in Abbildung 2.1 auf der Seite 16 soll zur Verdeutlichung der auf den vorherigen Seiten eingeführten Notation dienen. Für das dargestellte Petrinetz gilt $P = \{p_1, p_2, p_3, p_4, p_5\}$, $T = \{t_1, t_2\}$, $I = \{(p_1,t_1), (p_2,t_1), (p_2,t_2), (p_3,t_2)\}$ und $O = \{(t_1,p_4), (t_1,p_5), (t_2,p_5)\}$.
Auf der linken Seite der Abbildung ist das Petrinetz in einem Zustand, bei dem t_1 und t_2

aktivierbar sind. Da aber nur eine der beiden Transitionen letztendlich schalten kann, handelt es sich um einen sogenannten Vorwärtskonflikt. Der Simulationsalgorithmus muß also nichtdeterministisch eine dieser beiden aktivierbaren Transitionen auswählen.

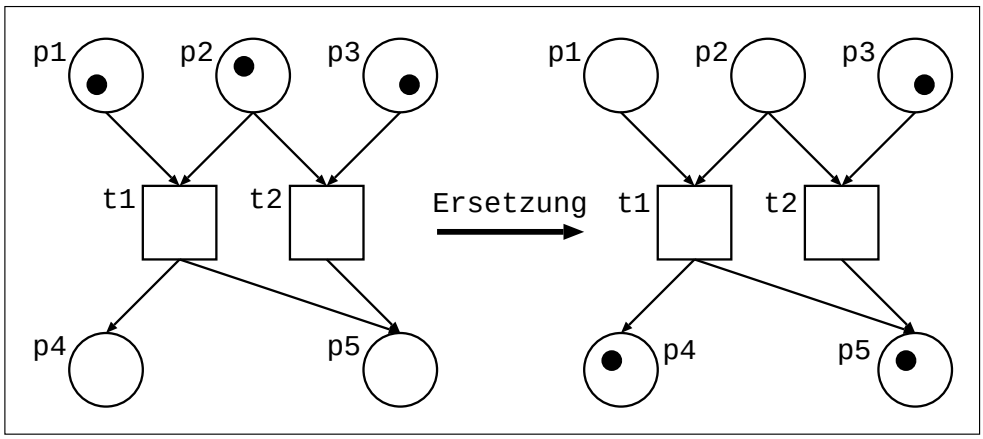

Abbildung 2.1: Petrinetz

Im Beispiel ist die Wahl auf t_1 gefallen. Die Eingangstellen p_1 und p_2 werden demarkiert und anschließend die Ausgangsstellen p_4 und p_5 markiert. Weitere Schaltvorgänge sind bei dem gegebenen Netz nicht möglich, da nicht mehr genug Marken auf den Eingangsstellen der Transitionen liegen.
Für die Menge der Eingangs- und Ausgangs-Stellen von t_1 gilt ${}^\bullet t_1 = \{p_1, p_2\}$ und $t_1^\bullet = \{p_4, p_5\}$. Weiterhin gilt für t_2 ${}^\bullet t_2 = \{p_2, p_3\}$ und $t_2^\bullet = \{p_5\}$.

2.1.3 Erweiterungen

Als Grundlage für die erweiterten Prädikat/Transitions-Netze dienen die im Abschnitt 2.1.2 beschriebenen Petrinetze. Der Hauptunterschied, der zwischen den Petrinetzen und den erweiterten Prädikat/Transitions-Netzen besteht, ist, daß die Marken bei den Petrinetzen nicht unterscheidbar sind, wohingegen es sich bei den Marken der erweiterten Prädikat/Transitions-Netze um individuelle unterscheidbare hierarchische Tupel handelt.
Im einzelnen betreffen die Erweiterungen folgende Punkte:

- Die Marken der Stellen werden durch formale Summen von Konstantenausdrücken beschrieben.
- Die Transitionen werden um eine Schaltbedingung und Schaltaktionen erweitert. Zusätzlich können noch Schaltzeiten festgelegt werden.
- Die Kanten besitzen Annotationen, die durch formale Summen von Variablenausdrücken beschrieben werden.

Es gibt einen formalen Ansatz zur Beschriftung von Prädikat/Transitions-Netzen, der ausführlich in [tack 92] im Kapitel 3 (Formale Grundlagen) abgehandelt wird. In diesem gerade zitierten Kapitel, das einen Abschnitt über die Prädikatenlogik erster Ordnung beinhaltet, wird eine Sprache $\mathcal{L} = \mathcal{L}^1 \cup \mathcal{L}_{edge} \cup \mathcal{L}_{mark} \cup \mathcal{L}_{assign}$ zur Beschriftung von

Prädikat/Transitions-Netzen definiert. Mit der Sprache $\mathcal{L}_{mark}$ können formale Summen von Konstantenausdrücken und damit die Marken, die in den Stellen abgelegt werden, beschrieben werden. Die Sprache $\mathcal{L}_{edge}$ dient zur Beschreibung formaler Summen von Variablenausdrücken und damit zur Beschreibung der Annotationen an den Kanten. $\mathcal{L}^1$ wird als Sprache der Prädikatenlogik erster Ordnung definiert, mit der auch die Schaltbedingung (*condition*) einer Transition beschrieben wird. Die Sprache $\mathcal{L}_{assign}$ dient zur Beschreibung von Anweisungsketten, die bei den Schaltaktionen der Transitionen angegeben werden können.
Die Syntax und Semantik, die im praktischen Einsatz bei der Prädikat/Transitions-Netz-Enwicklungsumgebung und im speziellen beim Prädikat/Transitions-Netz-Editor zum Tragen kommt, wird in [rust 95] (Annotationen von Pr/T-Netzen) ausführlich beschrieben. Die wesentlichsten Aspekte werden im folgenden erläutert, um einen Einblick zu vermitteln, welche Möglichkeiten für die Beschriftung eines Prädikat/Transitions-Netzes zur Verfügung stehen.

Stellen

Bei den Marken der Stellen handelt sich um hierarchische Tupel von Konstanten, die durch eckige Klammern umschlossen werden. Beispielsweise sind $[1], 2[3,2], [1,[2,3]]$ und $4[\text{“}2.113500\text{“} : float]$ solche Tupel. Es ist möglich dem Tupel eine Kardinalität und den Konstanten einen Typ zuzuordnen. Die Kardinaltität drückt aus, wie oft die angegebene Marke in der Stellen enthalten sein soll. Mögliche Typen, die den Konstanten zugeordent werden können, sind *char*, *int*, *float*, usw. Details findet man in [rust 95] im Abschnitt 3. Konstanten sind beispielsweise 'x' entspricht “x“ : *char*, 4 entspricht “4“ : *int* und 1.41 entspricht “1.410000“ : *float*. Anhand der Beispiele kann man schon erkennen, daß eine explitzite Typzuweisung nur in Ausnahmefällen nötig ist, nämlich dann, wenn man einer Konstanten einen Typ zuordnen möchte, der mit der automatischen Zuordnung nicht übereinstimmt.

Kanten

Die Tupel zur Beschreibung der Marken der Stellen und der Annotationen an den Kanten sind strukturell identisch. Zusätzlich zu der Kardinalität können noch die von den regulären Ausdrücken her bekannten Symbole $*$, $+$ und ? verwendet werden. Die Typzuordnung erfolgt automatisch, wenn keine Angabe gemacht wird. Daher ist eine explizite Typevorgabe nur zur Typumwandlung sinnvoll. Mögliche Annotationen sind beispielsweise $[x], 3[x], [x,z]$ und $[x,[y,z]]$. Die Kardinalität vor einem Tupel, die bei erweiterten Prädikat/Transitions-Netzen als abkürzende Schreibweise erlaubt ist, wird bei den Petrinetzen (siehe Abschnitt 2.1.2) durch eine entsprechende Anzahl von Kanten ausgedrückt.

Transitionen

Schaltbedingung Bei der Schaltbedingung einer Transition handelt es sich um einen boolschen Ausdruck, der aus Konstanten und Variablen bestehen kann. Die Variablen müssen an den Eingangskanten definiert sein. Damit eine Transition aktivierbar ist, muß die Schaltbedingung erfüllt sein. Dazu muß gelten, daß die Eingangs-Stellen der Transition mit Marken belegt sind, die strukturell konform mit den Annotationen an

den Kanten sind. Weiterhin müssen die Werte der Konstanten eine Belegung der Variablen ermöglichen, die konsitent ist, und für die gilt, daß die logische Auswertung der Schaltbedingung den Wert '*wahr*' liefert. Konsistent bedeutet, wird eine Variable an einer Eingangskante mit einem Wert belegt, so muß sie auch bei jedem weiteren Auftreten an einer Eingangskante mit dem gleichen Wert belegt werden. Das Beispiel, das die Abbildung 2.2 auf Seite 19 zeigt, soll unter anderem diesen Sachverhalt verdeutlichen. Die Verknüpfung der Werte wird in [rust 95] im Abschnitt 5 beschrieben. Im Abschnitt 4 werden die erlaubten Operatoren, die zum größten Teil mit denen von C++ identisch sind, beschrieben. Es werden auch die Punkte Verknüpfung, Vergleich und Mengenoperationen angesprochen. Wird keine Klammerung verwendet, so gelten die üblichen Bindungsstärken. Beispiele für Schaltbedingungen sind: $a < b$, $a + b <= c$ und $(x + 3 > y)\&\&(x > 2)$. Tritt eine Variable an den Eingangs- und an den Ausgangskanten auf, so ist ihre Belegung identisch. D.h die Variable an den Ausgangskanten wird mit dem gleichen Wert belegt, der ihr an der Eingangskante zugewiesen wurde.

Schaltaktionen Die Schaltaktionen der Transitionen unterteilen sich in eine Schaltaktion zum Schaltbeginn (*preaction*), daß ist der Zeitpunkt nach der Demarkierung der Eingangs-Stellen, und eine Schaltaktion zum Schaltende (*postaction*), daß ist der Zeitpunkt vor der Markierung der Ausgangs-Stellen. Hat die Transition Ausgangskanten, an denen Variablen verwendet werden, die nicht an den Eingangskanten auftreten, so dienen die Schaltaktionen zur Berechnung neuer Werte, die diesen Variablen dann zugewiesen werden können. Die Anweisungen aus denen sich die Schaltaktionen zusammensetzen, werden durch ein Semikolon voneinander getrennt. Sie umfassen Zuweisungen und Kommandos, die ausführlich in [rust 95] im Abschnitt 6 beschrieben werden. Es stehen für die Beschreibung von Kommandoprametern und Variablenwerten die gleichen Ausdrücke wie bei den Bedingungen zur Verfügung. Der Unterschied zu den Bedingungen ist, daß beliebige Variablen verwendbar sind. Der Gültigkeitsbereich einer Variable ist auf die Anweisungskette beschränkt, in der sie definiert wird. Es existiert keine Variablendeklaration und feste Typzuordnung. Beispiele für Schaltaktionen sind: $x = a + b$ und $z = 2 * y + 3$.
Werden Werte an Funktionen übergeben oder bei Operationen verwendet, so wird wenn nötig eine Typumwandlung durchgeführt. Dies geschieht ohne Rückmeldung. Details hierzu sind auch in [rust 95] zu finden.
Das Beispiel, das die Abbildung 2.2 auf der Seite 19 zeigt, soll zur Verdeutlichung der vorher beschriebenen Punkte dienen. Wie bei dem Beispiel 2.1 auf Seite 16 läßt sich auch hier der Ablauf der Schaltvorgänge beschreiben. Damit die Transition t aktivierbar ist, muß die Bedingung von t erfüllt sein. Dies bedeutet, es müssen Marken in den Stellen p mit $p \in {}^\bullet t$ liegen für die gilt, daß sie eine Belegung der Variablen, die in der Bedingung verwendet werden, liefern, so daß ein Auswertung den Wert '*wahr*' liefert.
Entsprechende Annotationen an den Kanten werden vorausgesetzt. Verwendet werden die Variablen a und b. Es wird gefordert, daß $((2 * a) >= b)$ gilt. Über die Kante (p_1, t) gelangt ein Konstantentupel der Form $[a, b]$ und über die Kante (p_2, t) gelangen zwei Konstantentupel der Form $[b]$ zu t. Es wird also gefordert, daß die entsprechende Anzahl von Marken in den Stellen vorhanden sind. Die Gleichheit der strukturellen Form von Marken und Annotationen an den Kanten muß auch gegeben sein. Weiterhin muß gelten, daß die Belegung der Variablen b bei beiden Kanten gleich ist. Damit gelangt man zu dem Ergebnis, daß die Transition mit der Marke $[2, 3]$ von p_1 und den

zwei Marken der Form [3] von p_2 aktivierbar ist. Die verbleibenden Marken in p_1 bzw. p_2 sorgen für eine weitere Aktivierung.

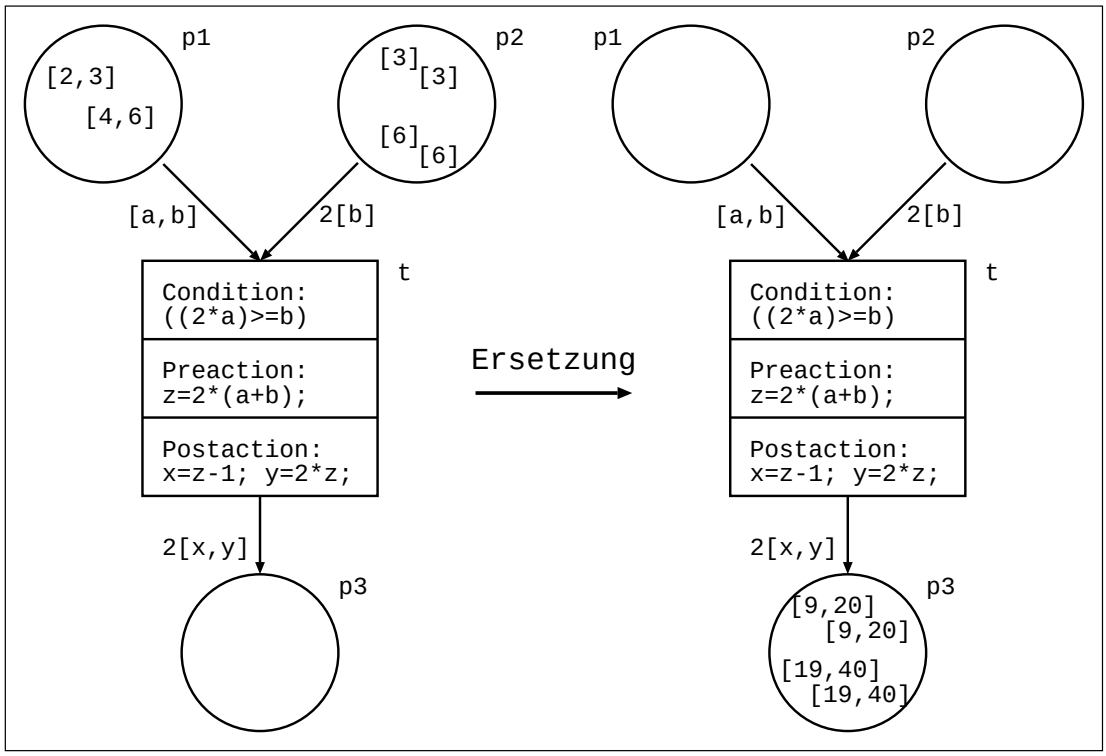

Abbildung 2.2: Erweitertes Prädikat/Transitions-Netz

Wird nun t aktiv, so werden die entsprechenden Marken aus p_1 und p_2 entfernt. Jetzt wird die Anweisung der *preaction* ausgeführt. Vor der Markierung der Ausgangs-Stellen wird die *postaction* ausgeführt. Es werden wie im Beispiel zu sehen jeweils zwei Marken mit den entsprechenden Werten in p_3 abgelegt.

Verzögerungszeiten Ein weiterer wichtiger Punkt ist, daß einer Transition Schaltzeiten zuordnen werden können, um auch zeitbehaftete Systeme modellieren zu können. Zu den Zeitangaben gehören die Aktivierungsverzögerung (*enable-delay*) und die Schaltverzögerung (*firing-delay*). Es wird jeweils noch unterteilt in minimal (*min*), durchschnittlich (*avg*) und maximal (*max*). Für die Wertangaben muß $0 \leq min \leq avg \leq max$ gelten. Die durchschnittlichen Verzögerungszeiten können für Analysezwecke verwendet werden. Für die jeweilige Verzögerungszeit muß ein Intervall angegeben werden, daß für den Simulationsalgorithmus bindend ist. Dies bedeutet, daß die entsprechende Aktion innerhalb dieses Intervalls durchgeführt werden muß. Während des Schaltens wird eine Transition als aktiv bezeichnet. Für den eigentlichen Schaltvorgang gilt nun, daß eine Transition, die aktivierbar ist, in den Zustand aktiv überführt wird. Das Entfernen der Marken von den Eingangs-Stellen dieser Transition wird als Demarkierungphase, und das Hinzufügen von Marken zu den Ausgangs-Stellen wird als Markierungsphase bezeichnet. Die Aktivierungsverzögerung gibt an, welche Zeit vergehen darf, bis eine aktivierbare Transition in den Zustand aktiv überführt wird. Die Demarkierung der

Eingangs-Stellen wird erst anschließend durchgeführt. Dies hat zur Folge, daß eine Transition mit einer anderen Ersetzung aktiviert werden kann, als mit der, die für die Aktivierbarkeit ausschlaggebend war, da während der Verzögerungszeit Veränderungen auftreten können. Mit der Schaltverzögerung wird festgelegt, wie lange die Transition aktiv ist. Erst anschließend wird die Markierung der Ausgangs-Stellen durchgeführt.

Schnittstellen

Weiterhin ist es möglich eine Stelle oder Transition als *Port* zu definieren. Hier gibt es die Möglichkeiten des Eingabe-*Ports* (P_i), Ausgabe-*Ports* (P_o) oder Eingabe-Ausgabe-*Ports* (P_{io}). Diese *Ports* werden benötigt, um eine Schnittstelle eines Netzes für die "Außenwelt" zur Verfügung zu stellen.

2.2 Expertensysteme

Der folgende Abschnitt unterteilt sich in die Unterabschnitte KI und Darstellung von Expertensystemen. Da das große Gebiet der KI, das wesentlich mehr umfaßt als nur die Expertensysteme, und das den Oberbegriff für alle Systeme, die sich mit der Erfassung und Verarbeitung der menschlichen Intelligenz befassen, bildet, beginnt der Abschnitt mit ein paar einleitenden Worten zu diesem Thema. Die Beschreibung der Expertensysteme und ihrer Eigenschaften ist der zweite wichtige Punkt dieses Abschnitts. Eine detaillierte Beschreibung der Punkte KI, Expertenwissen, Wissensrepräsentation und Aufbau eines Expertensystems wird in [goer 95], hauptsächlich im Kapitel 7, gegeben. Abschließend folgen wesentliche Aspekte bezüglich CLIPS.

2.2.1 Künstliche Intelligenz

In allen Entwicklungsphasen der KI, die ihren Ursprung in der Konferenz von 1956 am Dartmouth College findet, wo zum ersten Mal eine Untersuchung angeregt wurde, um festzustellen, wie man das Lernen und die Eigenschaften der Intelligenz beschreiben kann, um sie maschinell verarbeiten zu können, wurden unterschiedliche Ansätze und Ziele verfolgt. Hierzu gehören die absolut verschiedenen Ansätze der strikten Formalisierung und im Gegensatz dazu das exemplarische Realisieren durch Implementierung. Die zentrale Aufmerksamkeit galt aber schon immer der Repräsentation und Verarbeitung von Symbolen als wichtigste Basis interner Prozesse, von denen man annimmt, daß sie rationales Denken nachbilden können. Es müssen also die Punkte der Charakterisierung und Repräsentation von Wissen geklärt werden. Weiterhin muß man sich überlegen, was ein System, das über Wissen verfügt, auszeichnet.
Das große Ziel der KI ist die Operationalisierung der Kognitions- und Verstandesleistung des Menschen. Damit wäre eine Abbildung der Information, die wir als Wissen und Wahrnehmungsvermögen beschreiben, auf ein System der Informationsverarbeitung möglich, was eine Nachbildung der menschlichen Fähigkeiten durch einen Rechner nach sich ziehen würde. Die KI umfaßt unter anderem die Gebiete Wissensrepräsentation und Logik, KI-Programmiersprachen und Methoden und das Planen von Expertensystemen sowie das Problemlösen mit Expertensystemen. Weitere aktuelle Gebiete sind die Sprachverarbeitung, das Bildverstehen und die neuronalen Netze.

Der Unterschied zur klassischen Informatik liegt bei allen Themen darin, daß die Schwerpunkte die Wahrnehmung, das Schlußfolgern und das Handeln sind. Im allgemeinen muß also eine Operationalisierung der Intelligenz des Menschen erreicht werden. Zur Intelligenz gehören unter anderem auch das Erkenntnisvermögen, die Urteilsfähigkeit, das Erfassen von Möglichkeiten und auch das Begreifen von Zusammenhängen. Aus allen diesen Punkten zieht der Mensch letztendlich seine Schlußfolgerungen. An der Vielfalt der Informationen, die ein Mensch zur Urteilsfindung verwendet, erkennt man bereits, daß hier auch das Hauptproblem der KI liegt. Da sich Intelligenz nicht eindeutig messen oder beurteilen läßt, ist auch nicht die Möglichkeit der Operationalisierung gegeben, woraus folgt, daß keine Beschreibung durch formale Systeme möglich ist und damit auch keine Berechnungen auf Rechnern durchgeführt werden können. Der Schritt vom personalen Handeln zum schematischen ist nicht so einfach durchführbar.

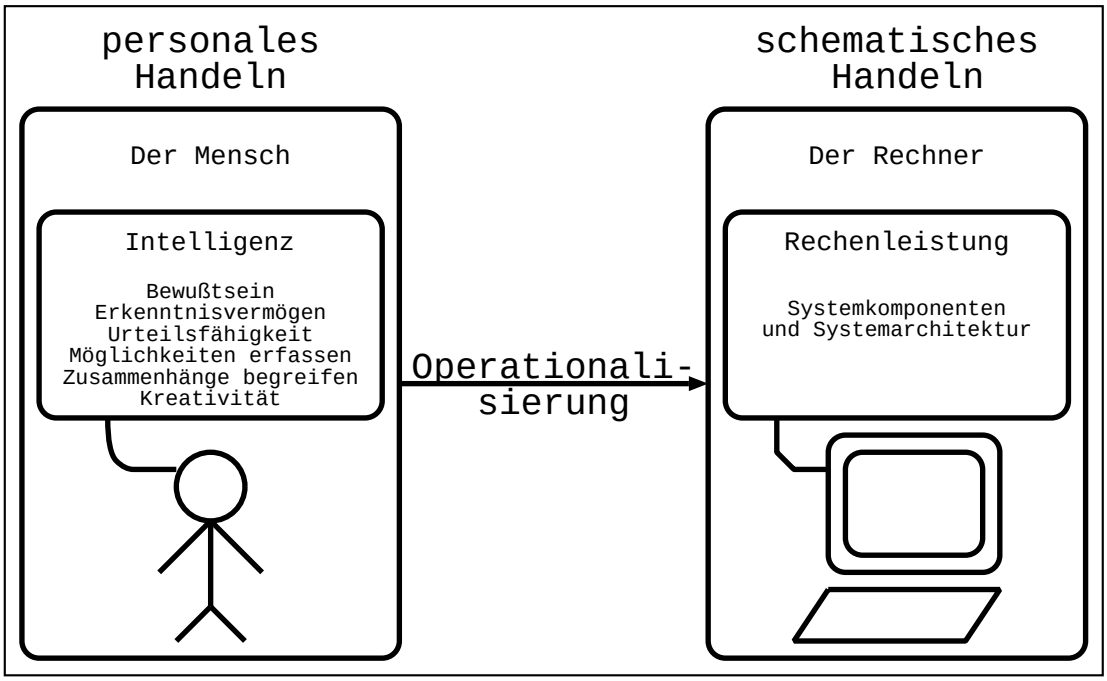

Abbildung 2.3: Ziel der KI

Dieses noch nicht erreichte Ziel der KI wird in Abbildung 2.3 in graphischer Form verdeutlicht. Auch für das Operationalisieren der Kreativität oder des Bewußtseins existieren keine Verfahren oder überhaupt Ansätze, die auf Verfahren schließen lassen würden. Würde ein allgemein gültiges Verfahren zur Beschreibung der Intelligenz gefunden, so würden sich Lösungsansätze zu vielen bestehenden Problemen auf dem Gebiet der KI finden lassen.
Das Ziel kann also nur sein, sich auf einzelne Aspekte der Intelligenz zu beschränken, und für diese eine Repräsentation und Verarbeitung des Wissens durch den Aufbau von Systemen zu finden. Unter den Forschern auf dem Gebiet der KI existiert auch noch kein Konsens, wie sich Intelligenz überhaupt darstellen läßt. Bei verschiedenen theoretischen Fassungen des Intelligenzbegriffs geht man davon aus, daß sich Intelligenz durch die Interaktion vieler einfacher Prozesse nachbilden läßt. Hier taucht dann auch noch das Problem auf, daß die Organisation und das Zusammenwirken, also die

Systemarchitektur, eine wesentliche Rolle spielen. Man geht davon aus, daß sich durch viele einfache Komponenten das komplexe Verhalten nachbilden läßt.
Für die auf der Konferenz von 1956 gestellte Frage „Wie kann Wissen modelliert und repräsentiert werden?“ existieren mehrere Ansätze. Logik kann zur Analyse auf der Wissensebene und zur Implementation von Logikformalismen als Repräsentationsmittel für Wissen dienen. Ein Ansatz ist nun das mathematisieren bestimmter Aspekte der Intelligenz. Hierzu gehören unter anderem das rationale Handeln und das logische Schlußfolgern. Wenn nun ein konkretes Problem gelöst werden soll, so kann man mit den vorher entwickelten Logikformalismen die für wahr gehaltenen Aussagen mit einer Repräsentationssprache beschreiben. Auf diesem Wege wird die Wissensbasis erstellt. Wie bei allen naturwissenschaftlichen Problemen versucht man auch hier einen Formalismus zur Repräsentation zu finden, der auf den allgemeinen Fall anwendbar ist. Dies hätte die Vorteile, daß man Bibliotheken und Aggregation nutzen könnte. Aufgrund der stetig steigenden Anzahl von wissensbasierten Systemen, ist es schwierig, einen einheitlichen Formalismus zu finden.
Die Informationskomplexität und Fülle unterscheidet sich bei den einzelnen Gebieten sehr stark. Für Expertensysteme kann ein relativ enger Rahmen abgesteckt werden, wobei man natürlich erwähnen muß, daß das zu erwartende Ergebnis um so exakter ist, desto mehr Informationen als Wissen eingebracht wurden, während man bei der natürlichen Spracheingabe wesentlich umfangreichere Informationen abdecken muß. Aber auch bei Expertensystemen hat man festgestellt, daß es sich bei dem Wissen, das ein Experte benutzt, um unscharfes Wissen handelt. Weiterhin ist der Experte in der Lage zu lernen, er nutzt seine Erfahrungen und weiß auch wie er sich in Ausnahmesituationen zu verhalten hat. Ein weiteres großes Problem ist auch, daß das Wissen nie im vollständigen Umfang vorliegt oder sich nicht vollständig modellieren läßt.
Ungeachtet der ganzen Probleme, die gerade aufgeführt wurden, existieren diverse praktische Anwendungen, bei denen Expertensysteme von Nutzen sind. Es handelt sich hauptsächlich um Probleme der Klassifikation (Diagnostik), der Konfiguration (Synthese) und Simulation. Ein ausschlaggebender Punkt für alle Probleme des täglichen Lebens, die man mit einem Expertensystem lösen möchte, ist, daß man die Domäne ausreichend beschreiben kann. Davon ausgehend, daß man für den vorher beschriebenen Weltausschnitt eine bestimmte Aufgabe lösen möchte, muß man noch die benötigte Funktionalität zur Verfügung stellen. Die Domäne und die auf sie anwendbare Funktionalität bilden zusammen ein Modell des zu lösenden Problems.

2.2.2 Expertenwissen

Bevor auf das eigentliche Expertenwissen eingegangen wird, kann man sich die Frage „Wozu benötige ich ein Expertensystem und welchen Nutzen bringt es mir?“ stellen. Das Hauptziel ist, dem Anwender eine Applikation anzubieten, mit der er die Möglichkeit hat, auf die Problemlösungskompetenz eines Experten zurückzugreifen, ohne daß dieser anwesend ist. Hierzu ist es nötig die Gedankengänge und Erfahrungen eines Experten so zu formalisieren, daß sie von einer Maschine verarbeitet werden können.
Vorteile, die durch das Formalisieren von Expertenwissen entstehen, sind, daß das Wissen in gespeicherter Form vorliegt, und man beliebig darauf zurückgreifen kann, und, daß die Möglichkeit der Überprüfbarkeit gegeben ist, und man das Wissen zur Unterstützung des eigenen Fachwissens verwenden kann. Die Umsetzung des Experten-

wissens in ein geeignetes Modell gestaltet sich schwierig, da es sich im wesentlichen um implizites Wissen handelt; Anmerkungen hierzu findet man auch im Kapitel 2.2.1. Um einen Wissensbereich zu modellieren, muß man sich über die Kategorien zur Einordnung von Objekten, Eigenschaften der Objekte und Annahmen über die Objekte, Beziehungen, die zwischen den Objekten möglich sein sollen, und möglichen Folgerungen, die man ableiten möchte, Gedanken machen.
Hier kommt ein Wissensingenieur (*knowledge engineer*), der die Aufgabe hat ein Fachgebiet so darzustellen, daß es maschinell bearbeitbar ist, zum Einsatz. Das Hauptproblem, das der Wissensingenieur lösen muß, ist, daß er ein Modell finden muß, das sich mit dem mentalen Modell des Experten deckt. Dies zeigt unmittelbar auf, daß die Lösung dieses Problems nicht auf der programmiertechnischen Ebene lösbar ist. Für dieses Problem gibt es auch keine allgemeinen formalen Methoden, so daß dem Wissensingenieur eine sehr wichtige Rolle zukommt, da man erkennen kann, daß die Lösung einer bestimmten Aufgabe nicht unwesentlich von ihrer Modellierung bzw. der Modellierung der dazu benötigten Daten abhängt. Hinzu kommt auch noch, daß verschiedene Experten bei ein und demselben Problem unterschiedlich gut qualifiziert sein können. Es können auch Differenzen bezüglich der Exaktheit und Differenziertheit vorhanden sein. Ein weiterer Punkt, der nicht vernachläßigt werden darf, ist, daß der Experte sein Expertenwissen in sein Allgemeinwissen eingebettet hat, daß aber die Modellierung der Informationen für die Wissensbasis nur einen Ausschnitt bilden kann, der sich stark auf das spezielle Aufgabengebiet bezieht und dadurch die Freiheitsgrade, die ein Experte nutzen kann, bei der Verwendung eines Expertensystems nicht zur Verfügung stehen. Details zu diesem sogenannten „Kliff- und Plateau-Effekt“ sind in [pupp 91] zu finden.
Es fallen einem ein Teil der Aufgaben des Wissensingenieurs zu, wenn man mit der Prädikat/Transitions-Netz-Entwicklungsumgebung ein Optimierungsproblem angeht. Daher muß man sich zur Wissensmodellierung über die zwei folgenden Punkte, zu deren Verdeutlichung die Abbildung 2.4 auf der Seite 24 dienen soll, Gedanken machen. Die Modellierung ist, wie vorher schon öfter bemerkt, mit am wesentlichsten für das Optimierungsergebnis.

1. Wie setzt man ein System in ein Prädikat/Transitions-Netz um?

   Die grundlegende Frage der Modellierung richtet sich entweder an einen sehr erfahrenen Anwender oder eher noch an einen Experten, der die grundlegenden Module erstellen sollte. Demjenigen, der das System einsetzen möchte, sollte die Aufgabe der Verbindung dieser Module zufallen. Die Module im Zusammenhang mit der Verbindungsstruktur sollten dann das zu optimierende System repräsentieren. Es ist auch vorstellbar, daß eine bestehende Bibliothek so erweitern wird, daß sie bestimmten Erfordernissen gerecht wird. Problemlos lassen sich auch Systeme modellieren, für die ein allgemeingültiges Verfahren zur Modellierung existiert. Hierzu gehört beispielsweise das Umsetzen von Hardware in Prädikat/Transitions-Netze. Eine Beschreibung des Verfahrens ist in [pete 81] gegeben.

2. Welche Transformationen können für die Optimierung verwendet werden?

   Auch hier sollte die Vorarbeit von einem Experten geleistet worden sein, so daß man sich auf die Auswahl der für den entsprechenden Fall geeigneten Transformationen und die Beeinflussung des Ablaufzyklusses der Optimierung durch die Wahl einer geeigneten Konfliktlösungsstrategie beschränken kann.

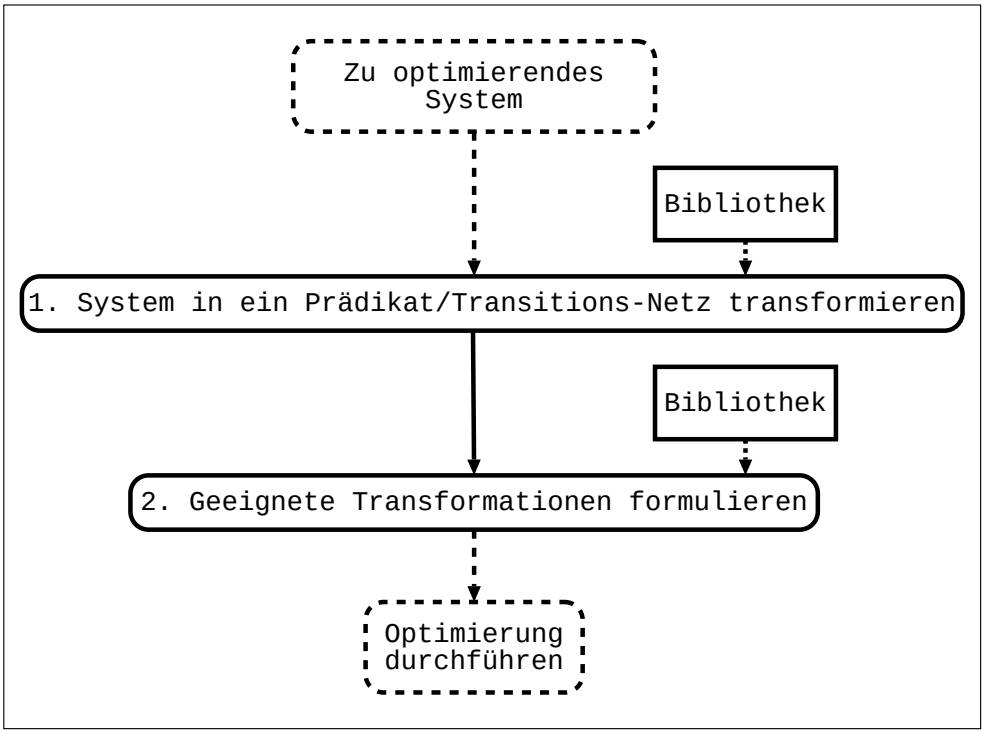

Abbildung 2.4: Der Umsetzungsprozeß

Nachdem ein Einblick in die zu beachtenden Problem der KI und der Wissensmodellierung gegeben wurden, wird nun im Kapitel 2.2.3 konkret auf die Wissensrepräsentation und im Kapitel 2.2.4 auf die Architektur eines Expertensystems eingegangen.

2.2.3 Wissensrepräsentation

Das Ziel der Wissensrepräsentation ist, das Expertenwissen so darzustellen und zu strukturieren, daß es von informationsverarbeitenden Systemen gut weiterverarbeitet werden kann. Dazu stehen im allgemeinen Objekte, Frames, Constraints und Regeln zur Repräsentation des Wissens zur Verfügung. Die Darstellung von prozeduralem Wissen ist auch möglich. Es hängt nun von dem eingesetzten Expertensystem ab, welche Wissensrepräsentation im konkreten Fall unterstützt wird. Hinzu kommt auch noch, daß sich bestimmte Darstellungsformen für bestimmte Problemlösungsmethoden besonders gut eignen. Zum Beispiel eignet sich für das konkrete Problem der Optimierung, daß sich dadurch charakterisieren läßt, daß man von einem Anfangszustand (das zu optimierende System) über diverse Zwischenzustände in einen Endzustand (das optimierte System) gelangt, besonders gut die Darstellung des Wissens durch Objekte, Regeln und Frames in Verbindung mit der Problemlösungsstrategie (siehe Kapitel 2.2.4) des monotonen Schließens. Bei dem Optimierungsproblem handelt es sich um ein Konstruktionsproblem (Synthese), so daß sich dieses Muster auf alle Anwendungen dieses Gebiets ausdehnen läßt.
Im folgenden wird nur allgemein auf die Oberbegriffe eingegangen. Die für die Arbeit wesentlichen Aspekte werden im Kapitel 2.3 besprochen. In [crhejue 91] und [goer 95] ist eine genaue Beschreibung der Repräsentationsformen zu finden.

Ausgehend von der einfachsten Methode Wissen darzustellen, es handelt sich hierbei um eine ungeordnete Menge von Fakten (i. allg. Textmuster), läßt sich eine Strukturierung erreichen, wenn das Expertensystem die Verwendung von Templates erlaubt. Ein Template ist mit einer *record*-ähnlichen Struktur zu vergleichen, wie man sie aus den Programmiersprachen C oder Pascal her kennt.
Eine Strukturierungsstufe höher liegen die Frames, die mit den in C++ bekannten Klassen vergleichbar sind. Die Objektdarstellung wird bei den Frames z. B. durch solche Punkte wie Vererbungshierachie, zu dem Objekt gehörige Prozeduren und Methoden sowie Initialisierung durch Defaultwerte erweitert. Es ist auch ein Mechanismus zur automatischen Klassifikation unbekannter Objekte vorstellbar.
Eine weitere Repräsentationsform sind die Constraints, durch die beliebige Relationen zwischen Variablen dargestellt werden können. Um eine gültige Belegung für alle Variablen zu bekommen, startet man mit einer Belegung für einen Teil der Variablen und bestimmt dann für weitere Variablen deren Belegung. Das dabei verwendete iterative Verfahren wird als Propagierung bezeichnet.

2.2.4 Der Aufbau eines Expertensystems

Der Aufbau der Expertensysteme hat sich im Laufe der Zeit gewandelt. Während in den Anfängen zur Erstellung eines Expertensystems hauptsächlich die KI-Programmiersprachen Lisp und Prolog verwendet wurden, wird heutzutage hauptsächlich C verwendet, um eine Integration in bestehende Umgebungen zu ermöglichen. Im Laufe der Zeit ist auch eine Trennung zwischen den Kontrollstrukturen, der sogenannten Shell, und dem Wissen vollzogen worden.

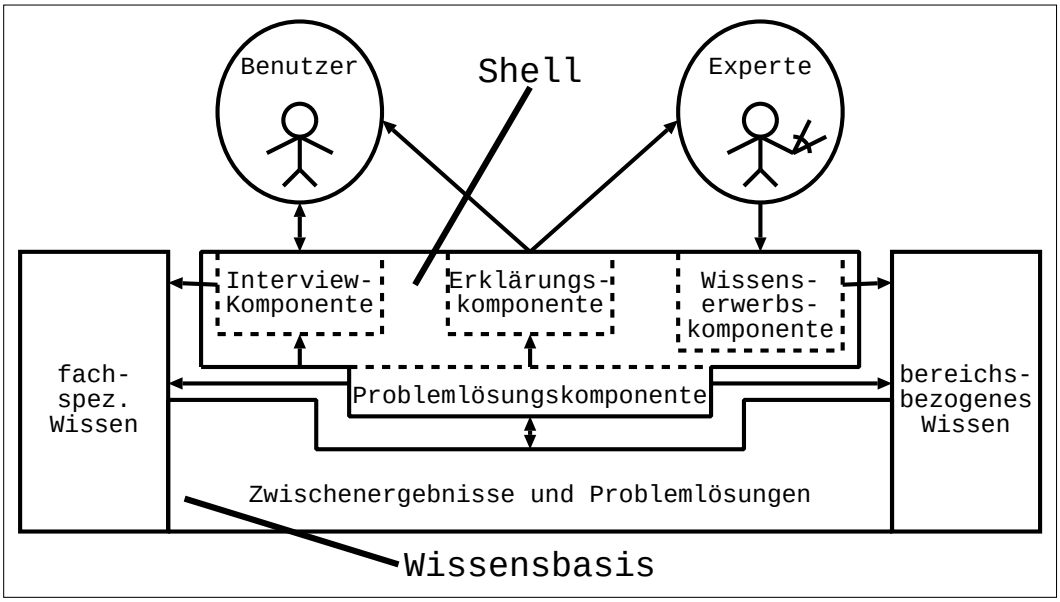

Abbildung 2.5: Der Aufbau eines Expertensystems

Man unterscheidet nun zwischen den KI-Programmiersprachen, den Expertensystemwerkzeugen, die einen Zugriff auf die verschiedenen Komponenten und streckenweise eine Beeinflussung der Wissensrepräsentation erlauben, und den Expertensystem-Shells,

die sich für diverse Anwendungen eines speziellen Problemlösungstyps eignen. Die Art des dargestellten Wissens und deren Verarbeitung liegen bei den Expertensystem-Shells fest. Das konkrete Wissen muß aber noch eingebracht werden.
Die Shell besteht aus einer Dialog-, einer Erklärungs-, einer Wissenserwerbs- und einer Problemlösungskomponente. Die Komponenten, die bereichsneutral sind, gehören zur Shell, und den bereichsspezifischen Teil stellt die Wissensbasis dar. Die Shell ist also der Teil des Expertensystems, der für verschiedene Probleme verwendbar ist, während die Wissensbasis immer für ein spezielles Problem erstellt werden muß.
In der Abbildung 2.5 werden die einzelnen Komponenten und ihr Zusammenhang dargestellt. Die Abbildung wurde [pupp 91] entnommen und wurde nur leicht graphisch aufbereitet.
Es folgt eine Beschreibung der einzelnen Komponenten:

- Die **Interviewkomponente**

  wird benötigt, um bei der Urteilsfindung Rückfragen an den Anwender stellen zu können. Es handelt sich also um ein Modul, das zur Unterstützung der Problemlösungkomponente benötigt wird. Der Informationsfluß sieht so aus, daß im Falle fehlender Informationen, die für eine Schlußfolgerung benötigt werden, die Interviewkomponente aktiviert wird, und der Anwender die entsprechenden Informationen eingeben muß. Die eingegebenen Daten gelangen in die Wissensbasis und stehen dann dort zum weiteren Abruf bereit, so daß die Problemlösungskomponente auf sie zugreifen kann.

- Die **Erklärungskomponente**

  dient dazu, dem Anwender bzw. dem Experten Ergebnisse in einer ansprechenden graphischen Form darzustellen, damit eine einfache Interpretation möglich ist. Für die Bereitstellung dieser Ergebnisse ist die Problemlösungskomponente zuständig.

- Mit der **Wissenserwerbskomponente**

  wird das bereichsbezogene Expertenwissen, das einen Teil der Wissensbasis bildet, eingebracht.

- Bei der **Problemlösungskomponente**

  handelt es sich um den Teil des Expertensystems, der für die Verwaltung der feuerbereiten Regeln und für die Auswahl der Regel, die als nächste feuern darf, zuständig ist. Sie besteht dazu unter anderem aus der Agenda und einem Inferenzmechanismus. Die Agenda, die man als Abarbeitungsreihenfolge betrachten kann, besteht aus einer geordneten Menge von feuerbereiten Regeln. Das Einfügen einer Regel in diese Menge geschieht nach einer bestimmten Konfliktlösungsstrategie, die der Anwender aus einer Menge von Vorgaben vorab bestimmen kann.

  - Bei der einfachsten Strategie werden die Regeln der Reihe nach abgearbeitet.
  - Die Regeln werden nach ihrer Spezifität unterschieden. Dies bedeutet, wenn bei einer Regel zwei Muster in ihrer Bedingung stehen und bei einer anderen nur ein Muster, so wird die Regel mit den zwei Mustern ausgewählt, da sie spezieller ist als die andere. Genaueres zu den Regeln ist im Kapitel 2.3.1 zu finden.

- Es ist auch möglich die Regeln mit einem Zeitstempel zu versehen und die Auswahl abhängig von ihrer Aktualität zu machen.
- Weiterhin ist auch die Verwendung von Meta-Wissen vorstellbar. Man kann den Regeln z. B. Prioritäten zuordnen oder auf strukturelle Eigenschaften eingehen. Beispielsweise kann man einem Expertensystem, das für Behandlungsvorschläge verwendet wird, angeben, daß man den Patienten zuerst mit Behandlungmethoden, die bedenkenlos sind, therapieren möchte, bevor man zu Methoden mit erhöhtem Risiko übergeht. Bei Meta-Wissen handelt es sich letztendlich auch nur um Wissen, das aber mehr auf strukturelle Gegebenheiten eingeht.

Die Konfliktlösungsstrategien, die beim Einsatz von CLIPS gewählt werden können, werden im Kapitel 2.3.5 erläutert.

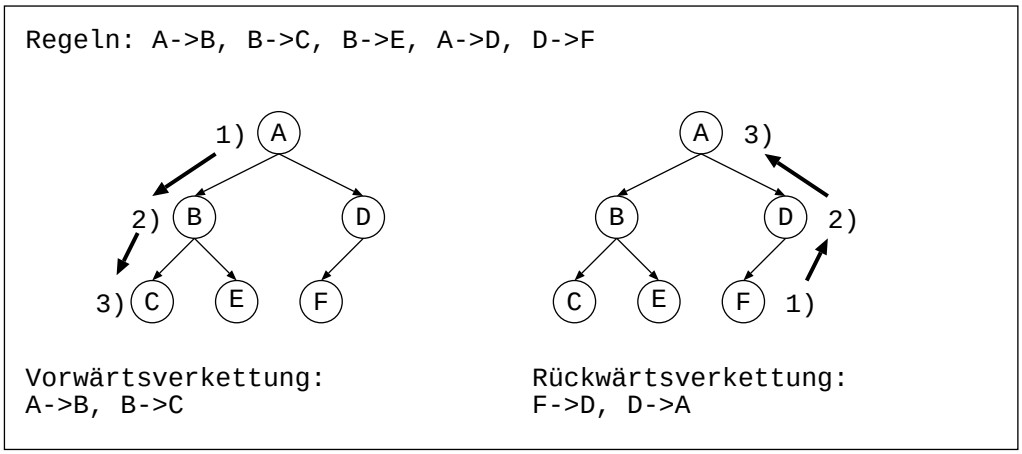

Abbildung 2.6: Vorwärts- und Rückwärtsverkettung

Der Inferenzmechanismus ist der Teil der Problemlösungskomponente, der für die Herleitungen unter Verwendung einer Ableitungsstrategie zuständig ist. Die Herleitungen werden entweder durch eine Vorwärtsverkettung oder eine Rückwärtsverkettung erreicht. Die Abbildung 2.6 zeigt hierzu ein Beispiel.

Die Bäume zeigen bei den gegebenen Regeln jeweils alle möglichen Ableitungen an. $X \rightarrow Y$ ist so zu verstehen, daß wenn die Bedingung X erfüllt ist, so kann die Aktion Y ausgeführt werden. Genaueres hierzu findet man im Kapitel 2.2.3.

Eine Regel feuert bei einem gegebenen Faktensatz genau einmal. Voraussetzung ist natürlich, das der Faktensatz die Fakten enthält, die für eine Aktivierung der Regel verantwortlich sind. Soll die Regel erneut aktiviert werden, so geschieht dies nur durch die Aktualisierung des Faktensatzes und im speziellen durch die Aktualisierung der Fakten, die in der Bedingung der Regel abgefragt werden, und damit letztendlich für die Aktivierung verantwortlich sind.

Bei der Vorwärtsverkettung wählt der Regelinterpreter aus der Datenbasis eine Regel aus; die bereits vorher erwähnten Konfliktlösungsstrategien kommen hier zum Tragen; und führt deren Aktion aus. Dieser Vorgang wird solange wiederholt,

bis keine Regel mehr anwendbar ist. Die Vorwärtsverkettung kommt z. B. bei CLIPS zur Anwendung.

Bei der Rückwärtsverkettung wird ausgehend von einer Hypothese versucht, diese zu beweisen. Es werden daher alle Regeln überprüft, deren Aktion diese Hypothese enthält. Ist die Aussage der Vorbedingung dieser Regeln unbekannt, so werden rekursiv Unterziele bestimmt, um die Wahrheitswerte dieser Aussagen zu generieren. Entweder sind diese durch andere Regeln ableitbar oder sie müssen vom Benutzer erfragt werden. Ein Beispiel für eine Expertensystem-Shell, die die Rückwärtsverkettung verwendet, ist das von der Nixdorf Computer AG entwickelte TWAICE, das in [savo 85] beschrieben wird.

Ausgehend von der einfachsten Ableitungsstrategie, dem monotonen Schließen, bei dem einfach alle vorhandenen Muster mit den Bedingungen der Regeln verglichen werden, und dadurch die feuerbereiten Regeln bestimmt werden, gibt es auch komplexere Verfahren. Hierzu gehören beispielsweise das probabilistische Schließen, das mit Wahrscheinlichkeiten arbeitet, das nicht-monotone Schließen und das temporale Schließen. Beim nicht-monotonen Schließen wird berücksichtigt, daß die Informationen nicht immer vollständig vorliegen, so daß man teilweise aus mehr Informationen weniger schließen kann. Beim temporalen Schließen wird die Zeitinformation mit einbezogen. Da bei CLIPS das monotone Schließen zur Anwendung kommt, werden die anderen Verfahren hier nicht weiter erläutert. Detaillierte Informationen zu diesen Verfahren findet man in [pupp 91] und [crhejue 91]. Die Verfahren werden teilweise für Problemlösungen auf dem Gebiet der Klassifikation und der Simulation eingesetzt.

- Die **Wissensbasis**

  setzt sich aus dem fallspezifischen, dem bereichsbezogenen Wissen sowie den Zwischenergebnissen und Problemlösungen zusammen.

  Bei der konkreten Anwendung, die in der Arbeit vorgestellt wird, handelt es sich bei dem Prädikat/Transitions-Netz, das aus dem zu optimierenden System entwickelt wurde, und den zugehörigen Transformationen um das fallspezifische Wissen. Als bereichsbezogenes Expertenwissen könnte man die Regeln bezeichnen, die benötigt werden, um den Faktensatz in einem konsistenten Zustand zu halten. Es handelt sich zum Beispiel um Regeln, mit denen die Mengen der Eingangs- und Ausgangs-Stellen einer Transition auf den neusten Stand gebracht werden, wenn beispielsweise eine Kante, die von dieser Transition ausgeht oder zu ihr hinführt, gelöscht wird. Details zu diesem Regelsatz sind im Kapitel 4.2.2 zu finden. Als Problemlösung ist das optimierte Prädikat/Transitions-Netz zu sehen.

2.3 Das Expertensystem CLIPS

Im folgenden wird auf die Eigenschaften und die Arbeitsweise von CLIPS eingegangen. Detaillierte Informationen zu CLIPS sind in den CLIPS-User-Manuales [curu 93] und [cuob 93] sowie in den Reference-Manuales [crba 93], [crad 93] und [crin 93] zu finden.

2.3.1 Einleitung und Grundlagen

Das von der NASA entwickelte CLIPS (C Language Integrated Production System) ist zu den Expertensystem-Shells zu zählen. Zur Zeit des Entwicklungsbeginns, 1984, war Lisp die Basissprache, mit der Expertensysteme entwickelt wurden. Um die Nachteile, zu denen zur damaligen Zeit z. B. die geringe Verfügbarkeit auf einer großen Menge von Computern, die hohen Kosten der aktuellen Lisp-Werkzeuge und Hardware sowie die schlechte Integrationsfähigkeit von Lisp in andere Sprachen, was eine Einbettung in andere Applikationen erschwerte, gehörten, zu umgehen, wurde im Johnson Space Center von der Abteilung, die für die KI zuständig war, ein eigenes C-Expertensystem-Werkzeug entwickelt. Die wesentlichsten Punkte, die beachtet werden sollten, waren, daß das Werkzeug in einem angemessenen Zeitrahmen und mit geringem Kostenaufwand erstellt werden sollte. Es folgte eine rasche Entwicklung eines Prototyps, dessen Konzept als sinnvoll erachtet wurde, und der demzufolge weiterentwickelt wurde. Im Sommer 1986 wurde die Version 3.0, zu der erstmals auch Handbücher existierten, auch für Personen, die nicht bei der NASA arbeiteten, zugänglich. Es folgte eine stetige Weiterentwicklung und Erweiterung, es werden mittlerweile auch Templates und Frames (siehe Kapitel 2.2.3) unterstützt, bis zur heutigen Version 6.02, die in dieser Arbeit zum Einsatz kommt.

CLIPS, das mittlerweile über 4000 Anwender gefunden hat, wird z. B. in Regierungskreisen, in der Industrie und von Akademikern genutzt. Die Dokumentation und CLIPS selber sind über das Internet verfügbar. Detaillierte Informationen erhält man über die WWW-Homepage http://www.jsc.nasa.gov/~clips/CLIPS.html.

Wenn man mit CLIPS arbeiten möchte, so gibt es die Möglichkeit, eine der beiden eigenständigen Varianten zu wählen, oder man kann CLIPS in eine Applikation integrieren. Zu den eigenständigen Varianten gehört zum einen eine einfache Kommandozeilen-Version und zum anderen eine Version, die die Fenstertechnik verwendet. Bei der Verwendung einer der eigenständigen Varianten ist die mit Fenstertechnik vorzuziehen, da die Informationen wesentlich übersichtlicher dargestellt werden können. Es existieren Versionen für den Macintosh, Windows 3.1 und für X-Windows, das als GUI bei Unix-Systemen zum Einsatz kommt.

Da bei dieser Arbeit der Prädikat/Transitions-Editor um eine wissensbasierte Komponente erweitert wird, kam nur die Integration von CLIPS in die bestehende Applikation in Frage. CLIPS liegt als C-Quellcode vor. Daher ist die Integration in eine bestehende Applikation sehr leicht realisierbar. Ein genaue Beschreibung der Vorgehensweise findet man in den Reference-Manuales [crba 93] und [crad 93]. Ein weiterer wesentlicher Punkt, den CLIPS bietet, ist das Hinzufügen selbstprogrammierter Funktionen, so daß einer Erweiterung des bereits bestehenden Funktionsumfangs nichts im Wege steht. Für diese Arbeit werden z. B. Operationen benötigt, die Mengenoperationen auf Mustern nachbilden. Details der dazu benötigten Vorgehensweise werden nicht beschrieben. Man findet aber bei der Beschreibung der Transformationssprache im Kapitel 3 die für die Anwendung wesentlichen Aspekte. Das Hinzufügen von Funktionen wird in [crad 93] erklärt.

Um das Expertensystem im Rahmen der Anwendung einsetzen zu können, muß man sich nur mit grundlegenden Punkten der Wissensrepräsentation und im speziellen mit Fakten und Regeln auseinandersetzen. Hinzu kommt, daß Verständnis über den Ablaufzyklus von CLIPS und die Handhabung von Prioritäten, die verwendet werden können,

um bestimmten Regeln eine Vorzugsstellung zu verschaffen, benötigt wird. Grundkenntnisse bezüglich der von CLIPS gebotenen Konfliktlösungsstrategien kommen auch noch hinzu. Auf die einzelnen Punkte wird in der angegebenen Reihenfolge eingegangen.

2.3.2 Fakten und Regeln

Die einfachste Darstellung von Wissen ist ein einfaches Textmuster, das als Objekt oder Fakt bezeichnet wird.

```
Beispiele für Fakten:
(Frosch gruen), (Tier Frosch)
```

Wird eines oder beide dieser Muster in der Bedingung einer Regel entdeckt, und sind alle weiteren Bedingungen in dieser Regel erfüllt, so wird die Aktion dieser Regel ausgeführt.

```
Beipiel für eine Aktion:
(printout t "Das Tier" ?tier "ist" ?farbe "." crlf)
```

Es handelt sich bei der verwendeten Notation um die CLIPS-Notation, die für die Nutzung der Optimierungskomponente aber nicht benötigt wird, da hier, die Transformationssprache (siehe Kapitel 3) verwendet wird. Daher wird im folgenden die CLIPS-Notation verwenden, ohne sie genauer zu erläutern. Die Notation wird detailliert in den User- und Reference-Manuales [curu 93], [cuob 93], [crba 93] und [crad 93] erklärt.
Um die Informationen, die in Form von Fakten vorliegen, verwenden zu können, muß man beschreiben, welche Aktion bei einer erfüllten Bedingung ausgeführt werden soll. Dies geschieht durch ein Regel, deren allgemeinste Form wie folgt aussieht:

```
Bedingung(en) => Aktion(en)
```

Formuliert man also eine Regel, die bei gegebener Wissensbasis aktivierbar ist, so erreicht man eine Aktion, die sich auf eine Ausgabe beschränken kann, die aber auch eine Veränderung der Wissensbasis nach sich ziehen kann. Das Aktivieren einer Regel wird als feuern bezeichnet. Eine Verwendung der Fakten könnte also wie folgt aussehen:

```
Beispiel:
(defrule tier_farbe
  (Tier ?tier)
  (?tier ?farbe)
  =>
  (printout t "Das Tier" ?tier "ist" ?farbe "." crlf)
  (assert (Tier ?tier ?farbe))
)
```

Als Ergebnis würde man die Ausgabe “Das Tier Frosch ist grün“ bekommen, und die Wissensbasis würde um den Fakt `(Tier Frosch gruen)` erweitert. Die Voraussetzung für diese beiden Aktionen wäre, daß die Regel `tier_farbe` ausgewählt wurde und die Fakten `(Tier Frosch)` und `(Frosch gruen)` vorher in der Wissensbasis abgelegt wurden.

Bei den gerade verwendeten Fakten handelt es sich um geordnete Fakten, d. h. die Position der einzelnen Muster ist wesentlich. Würde also in der Bedingung einer Regel auf `(?tier Tier)` überprüft, so würde diese Regel nie aktiv werden. Eine weitere Strukturierung erhält man durch die Verwendung von Templates. Es ist z. B. vorstellbar, daß man ein Struktur zum Verwalten von Tierinformationen erstellt, die beispielsweise wie folgt aussieht.

```
Beispiel:
(deftemplate tier
  (slot typ
        (type SYMBOL))
  (slot farbe
        (type SYMBOL))
  (multislot nahrung
             (type SYMBOL))
 )
```

Man hat bei dieser Form der Wissensrepräsentation die Möglichkeit, gezielt auf einzelne Informationen zuzugreifen. Die Bedingung einer Regel könnte wie folgt aussehen.

```
Beispiel:
(defrule frist_gurke
  (tier (typ ?typ)(nahrung $? gurke $?))
  =>
  (printout t "Das Tier" ?typ "frißt Gurken.")
 )
```

Ein Fakt der die Regel zum Feuern bringen würde, sähe wie folgt aus.

```
Beispiel:
(tier (typ maus)
      (nahrung körner gurke apfel sonnenblumenkerne)
 )
```

Bei einer Regel hat man also generell eine Menge von Mustern, die zusammen die Bedingung bilden, und man hat Aktionen, die ausgeführt werden, wenn die Bedingung erfüllt ist.

2.3.3 Ablaufzyklus

Um zu bestimmen, welche Regel wann feuert, besitzt CLIPS einen sogenannten Inferenzmechanismus (siehe auch Kpitel 2.2.4), der die Aufgabe hat, zu entscheiden, welche Regel anwendbar ist. Sind bei einer gegebenen Wissensbasis mehrere Regeln feuererbereit, so muß der Inferenzmechanismus mit Hilfe einer Konfliktlösungsstrategie eine eindeutige Entscheidung finden. Die von CLIPS zur Verfügung gestellten Konfliktlösungsstrategien werden im Kapitel 2.3.5 beschrieben. Der Zyklus endet dann, wenn keine Regel mehr feuerbereit ist. Eine Regel kann also mit einem *if-then*-Konstrukt einer prozeduralen Sprache, z. B. C oder Pascal, verglichen werden. Der entscheidende Unterschied ist aber, daß ein Programm, das in C geschrieben wurde, die Bedingung immer

nur dann prüft, wenn der Programmzähler an der entsprechenden Stelle ist. Im Gegensatz hierzu gilt für die Regeln, daß es sich eigentlich um ein *whenever-then*-Konstrukt handelt, und die Ausführungsreihenfolge von der Konfliktlösungsstrategie abhängt.

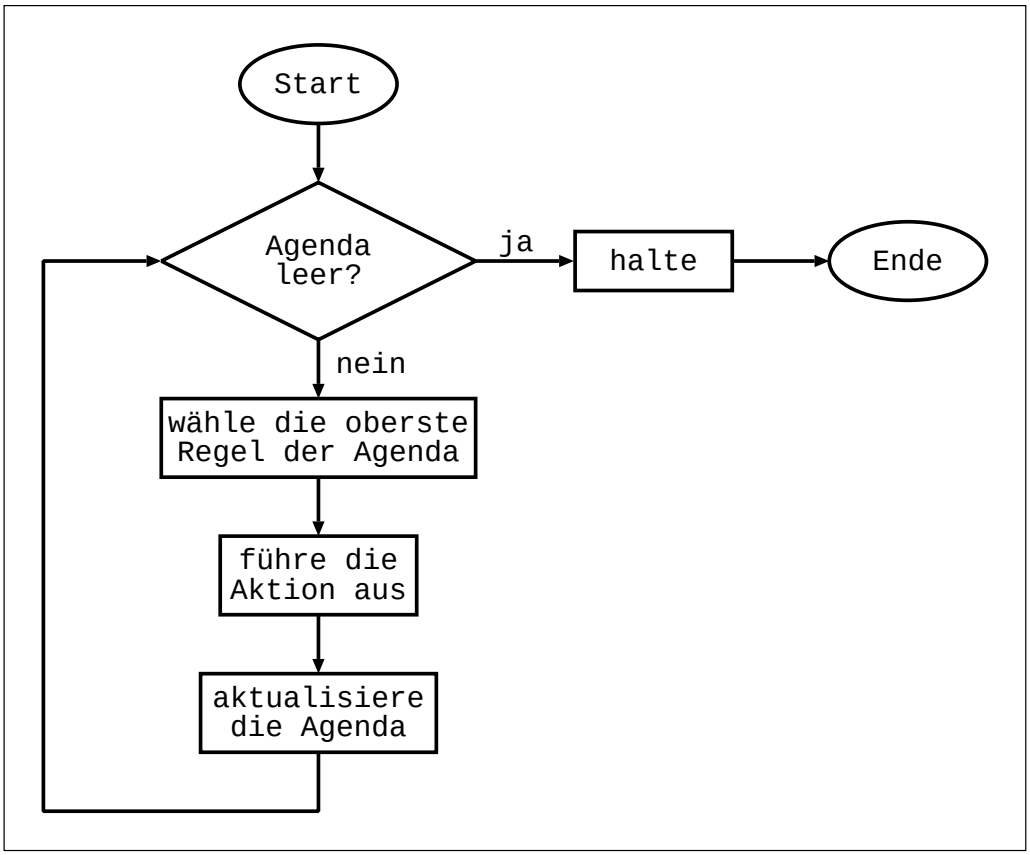

Abbildung 2.7: Ablaufzyklus der Regelausführung

Die von CLIPS verwendete Ableitungsstrategie ist das monotone Schließen, und zur Herleitung wird die Vorwärtsverkettung verwendet. Details zu den Verfahren findet man im Kapitel 2.2.4 und in [crhejue 91].
Wenn wir noch einmal das Beispiel, bei dem die Farbe zu einem Tier zugeordnet wird, betrachten, so sieht die Vorgehensweise beim Mustervergleich so aus, daß nach einer Instanz des Musters `(Tier X)` gesucht wird, wobei `X` bei gefundenem Muster mit einem Wert identifiziert wird. Weiterhin wird eine Instanz eines Musters `(X Y)` gesucht, wobei die Bindung von `X` durch das vorherige Muster der einzige Anhaltspunkt ist. Würde diese Einschränkung durch das erste Muster nicht existieren, so würde jedes Muster, das die entsprechende Form hat, verwendet werden können. Womit `Y` identifiziert wird, ist bei der hier gegebenen Situation belanglos. Da die Reihenfolge der Bestimmung der Instanzen der Muster nicht festgelegt ist, kann keine Aussage gemacht werden, ob tatsächlich zuerst nach einer Instanz des Musters `(Tier X)` gesucht wird. Hinzu kommt auch noch, daß jeweils die Konsistenz bei der Bindung einer Konstanten an eine Variable überprüft werden muß, da dies konform mit einer möglichen früheren Bindung laufen muß. Man erkennt, daß das Verfahren zum Mustervergleich eine hohe Komplexität hat, die es zu verringern gilt.

Für den Mustervergleich verwendet CLIPS den Rete-Algorithmus, der das sehr aufwendige Durchsuchen des Speichers nach der geforderten Instanz, beschleunigt. Der Rete-Algorithmus, der detailliert in [jack 87] beschrieben wird, nutzt, daß die Muster der Bedingungen verschiedener Regeln teilweise identisch sind, und, daß sich der Arbeitsspeicher beim Ausführen einer Aktion einer Regel nur gering verändert.
Im Gegensatz zu einem C-Programm, bei dem der Start- und Zielpunkt sowie der genaue Ablauf vorgegeben ist, ist bei CLIPS der Ablauf nicht fest vorgegeben. Wird durch eine Aktion die Wissensbasis verändert, so kann dies Einfluß auf die Agenda haben, da es sein kann, daß feuerbereite Regeln hinzukommen und andere nicht mehr feuerbereit sind. Die möglichen Prioritäten einzelner Regeln, und die Konfliktlösungsstrategie haben Einfluß auf das Einsortieren der Regeln. Ist die Agenda nicht leer, so wird die oberste Regel bzw. deren Aktion ausgeführt, was wiederum Einfluß auf die Wissensbasis hat und damit zu einem Aktualisieren der Agenda führt. Die Abbildung 2.7 auf Seite 32 verdeutlicht den Ablauf.

2.3.4 Prioritäten

Prioritäten können verwendet werden, um Regeln hervorzuheben oder zurückzustufen. Man sollte darauf achten, daß man sie nur selten verwendet, da es sonst passieren kann, daß man einen mehr oder weniger sequentiellen Ablauf vorgibt, was natürlich dem Verwendungszweck eines Expertensystems widerspricht.

2.3.5 Konfliktlösungsstrategien

Mit Hilfe der Konfliktlösungsstrategie werden die Regeln, die feuerbereit sind, in die Agenda eingetragen. Für Abarbeitungsreihenfolge gilt, daß die oberste Regel zuerst ausgeführt wird. Wird eine Regel aktiviert, so hängt ihre zu erlangende Position in der Agenda von mehreren Faktoren ab. Das größte Gewicht hat die Priorität, falls der Regel eine zugeordnet wurde. Existieren in der Agenda mehrere Regeln gleicher Priorität, so kommt die Konfliktlösungsstrategie zum Tragen. Werden mehrere Regeln gleichzeitig aktiviert, und weder durch die Prioritäten noch durch die Konfliktlösungsstrategie ist eine eindeutige Position in der Agenda zu bestimmen, so werden die Regeln willkürlich eingefügt. In diesem Zusammenhang hat die Reihenfolge, in der die Regeln definiert werden eine große Bedeutung. Dabei hängt die Reihenfolge von der Implemtierung und nicht von der Reihenfolge, in der ein Benutzer die Regeln und Fakten eingegeben hat, ab.
CLIPS bietet sieben verschiedene Konfliktlösungsstrategien, die nun im einzelnen erklärt werden. Die einzelnen Strategien, werden auch in [crba 93] beschrieben.

- Depth Strategy

  Neu aktivierte Regeln werden oberhalb aller Regeln, die die gleiche Priorität haben, plaziert. Werden mehrere Regeln gleichzeitig aktiviert, so ist ihre Reihenfolge untereinander willkürlich. Diese Strategie ist die *default*-Strategie.

- Breadth Strategy

  Werden Regeln neu aktiviert, so werden diese unterhalb aller Regeln, die die gleiche Priorität haben, plaziert. Auch hier ist die Reihenfolge der Regeln untereinander willkürlich, falls mehrere Regeln gleichzeitig aktiviert werden.

- Simplicity Strategy

  Bei einer Menge von Regeln mit gleicher Priorität, werden neu aktivierte Regeln oberhalb von Regeln mit gleicher oder höherer Spezifität plaziert. Die Spezifität wird durch die Anzahl der Vergleiche, die zur Auswertung einer Bedingung einer Regel benötigt werden, bestimmt. Für jeden Vergleich und jeden Funktionsaufruf erhöht sich die Spezifität um 1. Die logischen Operationen *and*, *or* und *not* sowie Funktionsaufrufe innerhalb von Funktionsaufrufen haben keine Auswirkung auf die Spezifität.

- Complexity Strategy

  Bei einer Menge von Regeln gleicher Priorität, werden neu aktivierte Regeln oberhalb der Regeln mit gleicher oder niedrigerer Spezifität plaziert. Die Spezifität wird wie bei der *simplicity*-Strategie bestimmt.

- LEX Strategy

  Jedem Fakt wird ein Zeitstempel zugeordnet, mit dessen Hilfe beim Vergleich zweier Fakten entschieden werden kann, welcher der neuere ist. Die Musterinstanzen werden nun in absteigender Reihenfolge, ausschlaggebend ist der Zeitstempel, angeordnet, um sie zu einem späteren Zeitpunkt mit den in den Bedingungen der Regeln verwendeten Mustern in Verbindung bringen zu können. Werden nun zwei Regeln neu aktiviert, so werden die Zeitstempel der korrespondierenden Musterinstanzen verglichen. Es werden also jeweils die Zeitstempel, die zu den Musterinstanzen der Bedingungen gehören, verglichen. Dies wird solange durchgeführt, bis ein Zeitstempel größer als sein korrespondierender Zeitstempel ist. Es wird nun die Regel, deren Musterinstanz in der Bedingung den größeren Zeitstempel bestitzt vor der anderen Regel in der Agenda plaziert. Differiert die Anzahl der Muster in den Bedingungen zweier Regeln, und die korrespondierenden Zeitstempel sind identisch, so wird die Regel, die mehr Zeitstempel besitzt, vor der anderen Regel in der Agenda plaziert. Besitzen die Regeln genau die gleiche Anzahl an Mustern, so wird die Regel, die die höhere Spezifität besitzt, vor der anderen Regel plaziert. Bei CLIPS werden den *not*-Bedingungselementen Pseudozeitstempel zugeordnet, die immer niedriger sind als die einer Instanz eines Musters, aber größer als die eines *not*-Bedingungselements, das innerhalb einer Abfrage instanziiert wurde.

- MEA Strategy

  Bei der MEA-Strategie werden die Zeitstempel der Musterinstanzen, die mit dem ersten Muster in der Bedingung der beiden zu vergleichenden Regeln identifiziert werden können, verglichen, um zu entscheiden, wo die Regeln zu plazieren sind. Ist der Zeitstempel des ersten Musters einer Regel größer als der einer anderen Regel, so wird diese Regel vor der anderen Regel auf der Agenda plaziert. Sind die Zeitstempel beider Regeln gleich groß, so wird die LEX-Strategie verwendet, um zu entscheiden welche Regel wo plaziert werden muß. Wie bei der CLIPS-LEX-Strategie haben negierte Muster Pseudozeitstempel.

- Random Strategy

  Das Einsortieren der Regeln in die Agenda wird bei der *random*-Strategie dadurch erreicht, daß jeder Regel eine Zufallszahl zugewiesen wird. Wird die Strategie

zwischendurch geändert, so wird die Zufallszahl gesichert, so daß nach einem erneuten Wechsel zurück zur *random*-Strategie die gleiche Reihenfolge der Regeln gegeben ist.

Welche Strategie man im konkreten Fall wählt, hängt davon ab, welche Kriterien man bei der Optimierung in den Vordergrund stellt.
Für den Fall, daß man eine Konfliktlösungsstrategie wählt, die die Regeln, die feuerbereit werden, vor den bereits feuerbereiten Regeln einordnet, so kann man davon ausgehen, daß verstärkt lokal optimiert wird, da nur eine Veränderung der Fakten zu einer Aktivierung neuer Regeln führen kann. Es würde sich die Wahl der *Depth*-Strategie anbieten. Bestimmt man nicht den Startpunkt durch eine bestimmte strukturelle Eigenschaft des Systems und entsprechend angepaßte Transformationen bevor der Optimierungsprozeß gestartet wird, so kann nicht vorausgesagt werden, welcher Teil des Prädikat/Transitions-Netzes zu Beginn gewählt wird.
Wird eine Konfliktlösungsstrategie gewählt, bei der die Regeln, die feuerbereit werden, nach den bereits feuerbereiten Regeln eingeordnet werden, so wird verstärkt global optimiert. Um dies zu erreichen, müßte man die *Breadth*-Strategie wählen.
Eine weitere Möglichkeit ist, daß man die Spezifität der Transformationen in den Vorder- oder Hintergrund stellen möchte. So könnte man durch die Wahl der *Simplicity*- oder *Complexity*-Strategie bestimmen, ob Transformationen bzw. Regeln, die eine geringe bzw. hohe Spezifität besitzen, bevorzugt werden sollen oder nicht. Wählt man beispielsweise die *Complexity*-Strategie, so könnte man dafür sorgen, daß Regeln, in deren Bedingung ein Ausschnitt des Prädikat/Transitions-Netzes präziser bzw. ein größerer Ausschnitt beschrieben wird, bevorzugt werden.
Es ist auch möglich, daß je nach System und Transformationen, die Wahl der Konfliktlösungsstrategie keinerlei Auswirkungen auf das Ergebnis hat. Dies ist dann der Fall, wenn nur ein eindeutiges Ergebnis existiert. Die Konflitklösungsstrategie hätte dann nur Auswirkungen auf die Zwischenergebnisse.

Kapitel 3

Die Transformationssprache

Der Kernpunkt dieses Kapitels ist die Beschreibung einer Sprache, mit der sich die Transformationen, die zur Analyse und Optimierung eines Systems dienen sollen, formulieren lassen. Da das System als Prädikat/Transitions-Netz modelliert werden muß, ist die Sprache so nah wie möglich an die Prädikat/Transitions-Netz-Notation angelehnt. Die im Kapitel 2 beschriebenen Grundlagen sowie die dort verwendeten Bezeichnungen werden im folgenden Verwendung finden.
Im ersten Abschnitt werden die Überlegungen beschrieben, die die Sprache geprägt haben. Es folgt im zweiten Abschnitt eine Beschreibung der benötigten Datentypen. Der dritte und umfangreichste Abschnitt befaßt sich mit der Syntax und Semantik der Sprache. Der anschließende Abschnitt geht auf die Berechnungen und die Termersetzungen ein, die beim Anlegen neuer oder beim Verändern bestehender Transitionen zur Bildung der *condition*, *preaction* und *postaction* verwendet werden können. Der letzte Abschnitt befaßt sich mit einem Kontrollmechanismus, der es erlaubt, die Abarbeitungsreihenfolge der Transformationen bzw. der Regeln dieser Transformationen zu beeinflussen. Die Grundlagen zu diesem Thema werden in den Kapiteln 2.3.3, 2.3.4 und 2.3.5 behandelt. Auf die Umwandlung der Transformationen in die CLIPS-Notation wird im Kapitel 4 eingegangen.

3.1 Grundlegende Überlegungen

Bevor auf die benötigten Datentypen und die Syntax und Semantik der Sprache eingegangen werden kann, muß festgelegt werden, welche Anforderungen die Sprache erfüllen muß und welche Aufgben mit ihr gelöst werden sollen.

1. Welche gegebenen Voraussetzungen müssen beachtet werden?
2. Welche Aufgaben sollen mit der Sprache gelöst werden?

zu 1. Eine wichtige Voraussetzung ist, daß das Modul zur Analyse und Optimierung in die bestehende Prädikat/Transitions-Netz-Entwicklungsumgebung eingepaßt werden soll. Das System, das mit dem Prädikat/Transitions-Netz-Editor modelliert wird, liegt anschließend als Prädikat/Transitions-Netz vor. Die komplexen Datentypen der Sprache spiegeln daher die Objekte, aus denen sich eine Prädikat/Transitions-Netz zusammensetzt, wider. Es handelt sich dabei um die

Stellen, Transitionen und Kanten. Weiterhin werden elementare Datentypen benötigt, um die Attribute dieser Objeke beschreiben zu können. Zusätzlich werden noch Mengen benötigt, damit man beispielsweise die Menge der Stellen oder der Transitionen direkt modifizieren kann, aber auch, damit man Objekte mit gleichen Eigenschaften gruppieren und somit bei späteren Aktionen verwenden kann. Der Einsatz der Mengen wird im Verlauf dieses Kapitels noch an Beispielen verdeutlicht werden.

Weiterhin müssen die beim Prädikat/Transitions-Netz-Editor verwendeten Annotationen, die zur Beschriftung des Netzes verwendet werden, auch bei den Transformationen zur Verfügung stehen. D. h. die bereits existierende Grammatik muß mit eingebunden werden. Um die bestehende Grammatik verwenden zu können, mußte an einigen wenigen Stellen vom Grundkonzept abgewichen werden. An den entsprechenden Stellen wird auf die Abweichungen, die keinen Einfluß auf die Funktionalität, sondern nur auf die Syntax haben, eingegangen.

Der dritte wichtige Punkt ist, daß eine Beschreibung, die in der Notation der Transformationssprache erstellt wurde, in die Lisp-ähnliche Syntax von CLIPS (siehe auch Kapitel 2.3) übersetzt werden muß, damit sie anschließend von dem Expertensystem weiterverarbeitet werden kann. Hieraus resultiert, daß die Beschreibung bereits so aufgebaut werden muß, daß sie in einen Regelsatz und in Fakten umgewandelt werden kann. Eine Transformation wird daher aus einer oder mehreren Regeln bestehen.

Im Bedingungs- bzw. Aktionsblock einer Regel beschreibt man die Objekte mit ihren Attributen. Falls beispielsweise für Vergleiche oder Berechungen Funktionen benötigt werden, so muß ihr Ergebnis im Bedingungsblock unmittelbar ausgewertet bzw. im Aktionsblock unmittelbar zugewiesen werden. Es werden also keine Variablen benötigt, da immer nur die Objekte des Systems verändert werden können. Eine Ausnahme stellen die Mengen dar, deren Elemente Objekte des Systems sind.

Die beiden letzten Punkte waren ausschlaggebend dafür, daß ein funktionaler Ansatz gewählt wurde.

zu 2. Es ist das Ziel, eine Beschreibung anzugeben, die aus Transformationen besteht, die eine Modifizierung eines bestehenden Prädikat/Transitions-Netzes ermöglichen sollen. Die einzelnen Transformationen sollen aus mehreren Regeln bestehen können, um auch komplexe Transformationen formulieren zu können.

Abgesehen von allgemeinen Kontrollstrukturen, konzentriert sich alles auf die Regeln und im speziellen auf den Bedingungs- und Aktionsblock dieser Regeln.

Im Bedingungsblock muß man einen Teil des Systems exakt beschreiben, um es zu identifizieren und anschließend mit Hilfe der im Aktionsblock zur Verfügung stehenden Operationen zu modifizieren. Für Vergleiche werden zusätzliche Funktionen benötigt.

Im Aktionsblock einer Regel wird auch wieder exakt beschrieben, welche Objekte mit welchen Attributen hinzugefügt, verändert oder gelöscht werden sollen. Auch hier können Funktionen zur Modifikation zum Einsatz kommen.

Da aus den vorherigen Fragen und deren Antworten bereits ersichtlich ist, daß eine Transformation aus mehreren Regeln bestehen kann, ergeben sich zwei weitere Fragen.

4. Werden Prioritäten benötigt, um Regeln hervorzuheben?
5. Wird ein Mechanismus benötigt, um Regeln zu gruppieren?

zu 4. Es muß die Möglichkeit gegeben sein, daß man verschiedenen Regeln verschiedene Wertigkeiten zuordnen kann. Einerseits, wenn eine Regel einer Transformation wichtiger als eine andere erachtet wird, andererseits, wenn eine komplexe Transformation formuliert werden soll, die in mehrere Regeln aufgesplittet werden muß, und die Abarbeitungsreihenfolge der Regeln wesentlich ist. Dies gilt z. B. für Transformationen, die eine Vorinitialisierungs- eine Aktions- und eine Nachbereitungsphase benötigen. Ein konkretes Beispiel ist im Kapitel 5 zu finden.

zu 5. Ein zusätzlicher Mechanismus zum Gruppieren muß unbedingt existieren, auch wenn es auf den ersten Blick so aussieht, als würde man die Freiheitsgrade des Expertensystems einschränken und einen sequentiellen Ablauf vorbestimmen. Dies ist bei fehlerhafter Verwendung auch sicherlich der Fall. Bei einer restrektiven Handhabung entstehen keinerlei Nachteile. Der Mechanismus bietet die Möglichkeit, mehrere Regeln so zu gruppieren, daß sie zusammen eine Transformation bilden. Während ein Expertensystem im Normalfall mit einer Menge von gleichberechtigten Regeln arbeitet, wird durch den Mechanismus die Möglichkeit gegeben, daß Transformationen als gleichberechtigt angesehen werden können. Anwendung findet dieser Mechanismus bei komplexen Transformationen, die in einzelne Teile (Regeln) gesplittet werden müssen. Wesentlich für diese Transformationen ist, daß nur alle Regeln dieser Transformation zusammen das gewünschte Ergebnis erzielen können, und auch nur dann, wenn nicht zwischendurch eine Regel einer andern Transformation feuert. Transformationen müssen also atomar abgearbeitet werden. Ein genauere Erläuterung ist im Kapitel 3.5 zu finden. Ein Beispiel ist im Kapitel 5 zu finden.

Bevor mit den einzelnen Abschnitten begonnen wird, wird noch festgelegt, was im folgenden unter einer *Anfrage* zu verstehen ist. Von einer Anfrage wird gesprochen, wenn eine einzelne Bedingung des Bedingungsblocks gemeint ist. Wird von einer Bedingung gesprochen, so ist die einzelne Bedingung gemeint. Alle Anfragen bzw. Bedingungen, die innerhalb eines Bedingungsblocks formuliert werden, müssen erfüllt sein, damit die Aktionen des Aktionsblocks ausgeführt werden. In diesem Zusammenhang bedeutet: Eine Anfrage bzw. Bedingung liefert das Resultat '*wahr*', daß diese einzelne Bedingung erfüllt ist und '*falsch*', daß diese einzelne Bedingung nicht erfüllt ist. Gilt für jede Anfrage innerhalb eines Bedingungsblocks, daß sie das Resultat '*wahr*' liefert, so werden die Aktionen des Aktionsblocks ausgeführt.

3.2 Datentypen

Im Kapitel 2.1 wurden die Objekte beschrieben, aus denen sich ein Prädikat/Transitions-Netz zusammensetzt. Diese Objekte liefern die Datentypen, die zur Formulierung der Transformationen benötigt werden. Zum einen sind dies die primär sichtbaren

Objekte, zu denen die Stellen, Transitionen und Kanten gehören, und zum anderen die Mengen, zu denen unter anderem P_i, P_o, P_{io}, P und T (siehe Kapitel 2.1) sowie die Mengen, die man durch die Punkt-Operatoren erhält, gehören. Mengen werden zusätzlich für Transformationen benötigt. Hierzu ist ein Beispiel im Kapitel 5.1 zu finden.
Welche Attribute bei den einzelnen Datentypen modelliert werden müssen, ist aus Kapitel 2.1 ersichtlich. Zusätzlich sind noch Attribute modelliert worden, die die Formulierung der Transformationen erleichtern sollen. Weiterhin werden noch Attribute für Verwaltungszwecke beim Analyse- und Optimierungsprozeß benötigt. In diesem Abschnitt werden die Datentypen mit allen Attributen vorgestellt, da in den folgenden Kapiteln, insbesondere im Kapitel 4, noch auf sie Bezug genommen wird. Welche Attribute konkret zur Verfügung stehen, ist davon abhängig, ob ein Objekt im Bedingungs- oder im Aktionsblock einer Regel beschrieben werden soll. Im Kapitel 3.3, in dem auf die Syntax und Semantik der Sprache eingegangen wird, wird ersichtlich werden, welche Attribute wo verwendet werden dürfen. Einige Bezeichnungen, die im Kapitel 3.3 besprochen werden, werden im folgenden schon vorab verwendet. Zuerst werden die Punkte besprochen, die alle Datentypen betreffen.

- Bei den Attributen wird unterschieden, ob es sich um ein Feld handelt, das genau ein Element aufnehmen kann, es handelt sich dabei im CLIPS-Sprachgebrauch um einen *slot*, oder, ob es sich um ein Feld handelt, das beliebig viele Elemente aufnehmen kann, ein sogenannter *multislot*. Wenn es sich um einen *slot* handelt so bedeutet dies, daß dem Attribut genau ein Wert zugeordnet werden kann. Bei *multislots* sind mehrere Werte erlaubt. Beispielsweise kann das Attribut Marke einer Stelle mehrere Marken aufnehmen, es ist daher als *multislot* definiert.

- Weiterhin wird auf den Typ eines Feldes eingehen. Die verwendeteten Typen sind INTEGER, es handelt sich um ganze Zahlen, und SYMBOL, wobei es sich um beliebige Zeichenketten handelt.

- Auf *default-Werte*, mit denen ein Feld vorbesetzt werden kann, wird auch eingegangen. Die *default*-Initialisierung erfolgt nur bei einem Objekt, das neu angelegt wird, und auch nur dann, wenn die Attribute nicht gesetzt werden. Es wird die *default*-Initialisierung, die auch in den *Dialog-Boxen* des Prädikat/Transitions-Netz-Editors beim Anlgen neuer Objekte vorgegeben wird, verwendet.

- Abschließend wird die Verwendung der einzelne Attribute erläutert. Dies gilt insbesondere für Attribute, die nicht unmittelbar zu dem Objekt gehören, sondern die zur leichteren Formulierbarkeit einer Transformation beitragen sollen. Wenn bei einem Attribut angegeben wird, das es abgefragt und gesetzt werden kann, so bedeutet dies, daß zum Abfragen und Setzen sowohl konkrete Werte vom vorgegebenen Typ als auch Bezeichner angegeben werden dürfen. Beim Anlegen eines neuen Objekts darf kein Attribut mit einem Bezeichner inititalisieren, der nicht vorher an einen Wert gebunden wurde. Die Verwendung von Bezeichnern ist bei der Abfrage von Attributen nur dann sinnvoll, wenn der Bezeichner bei einer späteren Operation auch benötigt wird. Dies ist beispielsweise dann der Fall, wenn zwei Attribute miteinander verglichen werden sollen oder bestehende Attribute zum Setzen der Attribute eines neuen Objekts verwendet werden sollen. Sonst bietet es sich an, direkt einen konkreten Wert abzufragen oder das Attribut

bei der Abfrage wegzulassen. Wird z. B. eine Optimierung gewünscht, die sich nur auf die strukturellen Eigenschaften des Netzes bezieht, so können Angaben bezüglich der Annotationen entfallen. Je weniger externe Vergleiche benötigt werden, um so schneller kann der Inferenzmechanismus entscheiden, welche Regeln zur Agenda hinzugefügt werden müssen. Extern bedeutet, daß eine zusätzlich Vergleichsoperation verwendet wird (siehe auch Kapitel 3.3.5.4).

- Stellen, Transitionen und Kanten besitzen die Attribute *id*, *name* und *info*. Die *id* wird benötigt, um ein Objekt eindeutig identifizieren zu können. Durch das Attribut *name* kann auf ein Objekt, das im Bedingungsblock an einen Bezeichner gebunden wurde, im Aktionsblock wieder zugegriffen werden. Das Attribut *info* wird für interne Verwaltungszwecke benötigt (siehe auch Kapitel 4.2.2). Mengen werden über den Bezeichner, der an das Attribut *name* gebunden wird, identifiziert. Sie besitzen keine *id*.

Im Anhang A werden die verwendeten Datentypen in der CLIPS-Notation gezeigt. Die Beschreibung der einzelnen Datentypen wird so aussehen, daß zuerst der Datentyp in tabellarischer Form mit allen Attributen angegeben wird, und anschließend noch einmal auf wesentliche Aspekte, die nicht aus der Tabelle ersichtlich sind, eingegangen wird. Zusätzlich wird für jedes Attribut angegeben, ob es abgefragt und gesetzt werden darf. Beispiele zu den Datentypen werden im Kapitel 3.3 angegeben.

3.2.1 Stellen

Stelle					
Attribut	Feldtyp	Datentyp	Inititalisierung	Abfragen	Setzen
id	slot	SYMBOL	-	nein	nein
name	slot	SYMBOL	-	-	-
in_nodes	multislot	SYMBOL	-	ja	nein
out_nodes	multislot	SYMBOL	-	ja	nein
in	slot	INTEGER	-	ja	nein
out	slot	INTEGER	-	ja	nein
lower_bound	slot	INTEGER	-1	ja	ja
upper_bound	slot	INTEGER	-1	ja	ja
predicate	multislot	SYMBOL	-	ja	ja
mark_number	slot	INTEGER	1	ja	ja
mark	multislot	SYMBOL	[1]	ja	ja
info	slot	SYMBOL	-	nein	nein

Der Datentyp Stelle mit den zugehörigen Attributen

Für den Namen muß sowohl im Bedingungs- als auch im Aktionsblock einen Bezeichner vergeben werden. Die Syntax eines Bezeichners wird im Kapitel 3.3.3 beschrieben. Der Name muß vergeben werden, da sonst kein Zugriff auf das Objekt mehr möglich wäre. Beispielsweise kann im Aktionsblock nur eine Kante zwischen einer Stelle und einer Transition angelegt werden, wenn die beiden Objekte identifiziert werden können. Dies bedeutet, daß sowohl für die Stelle als auch für die Transition ein Bezeichner vergeben worden sein muß. Da das Anlegen von Kanten zwischen Stellen und Transitionen erlaubt ist, die unmittelbar vorher im gleichen Aktionsblock angelegt wurden; wie dies realisiert wurde, wird im Kapitel 4.2.2 beschrieben; müssen auch im Aktionsblock Namen für die

Stellen und Transitionen vergeben werden. Abgesehen von dieser Ausnahme können nur Objekte im Aktionsblock modifiziert werden, die im Bedingungsblock an einen Bezeichner gebunden wurden, d.h. für die ein Name vergeben wurde.
Die Attribute *in_nodes* und *out_nodes*, es handelt sich jeweils um Mengen, werden intern mit den Eingangs- bzw. Ausgangs-Transitionen einer Stelle besetzt. Wird beispielsweise der Bezeichner `?p` an eine Stelle gebunden, so kann die Menge der Eingangs- bzw. Ausgangs-Transitionen durch die Angabe von `?*p` bzw. `?p*` bei den Mengen-Funktionen, die im Kapitel 3.3.5.5 beschrieben werden, verwendet werden. Würden diese Mengen nicht zur Verfügung gestellt, so müßte man zusätzliche Transformationen formuliern, um auf sie zugreifen zu können. Die Mengen dürfen für Anfragen und zum Löschen von Objekten verwendet werden. Es dürfen aber auf keinen Fall explizit Elemente zu diesen Mengen hinzugefügt werden, da sie intern verwaltet werden und eine Veränderung vom Setzen oder Löschen der Objekte bzw. der Kanten abhängt. Wird beispielsweise eine Eingangs-Transition einer Stelle gelöscht, so wird das Attribut *in_nodes* der betroffenen Stelle automatisch angepaßt. Automatische Hintergrundaktionen werden im Kapitel 4.2.2 beschrieben.
Die Attribute *in* und *out* werden mit der Anzahl der Eingangs- bzw. Ausgangs-Transitionen besetzt. Ihre Verwaltung erfolgt intern zusammen mit den Attributen *in_nodes* und *out_nodes*, da sie unmittelbar zusammengehören. Daher gilt auch für diese beiden Attribute, daß nur das Abfragen erlaubt ist. Sie können beispielsweise eingesetzt werden, um zu überprüfen, wieviele Eingangs- bzw. Ausgangs-Transitionen eine Stelle besitzt.
Das Attribut *mark_number* wird intern mit der Anzahl der Marken besetzt. Da der Wert vom Attribut *mark* abhängt, ist es nur möglich, die Anzahl der Marken abzufragen. Die Marken einer Stelle können sowohl gesetzt als auch abgefragt werden. Bei Anfragen bzgl. der Marken sind Besonderheiten zu beachten, die im Kapitel 3.3.5.4 erläutert werden.

3.2.2 Transitionen

Transition					
Attribut	Feldtyp	Datentyp	Inititalisierung	Abfragen	Setzen
id	slot	SYMBOL	-	nein	nein
name	slot	SYMBOL	-	-	-
in_nodes	multislot	SYMBOL	-	ja	nein
out_nodes	multislot	SYMBOL	-	ja	nein
in	slot	INTEGER	-	ja	nein
out	slot	INTEGER	-	ja	nein
enable_delay_min	slot	INTEGER	0	ja	ja
enable_delay_avg	slot	INTEGER	0	ja	ja
enable_delay_max	slot	INTEGER	0	ja	ja
firing_delay_min	slot	INTEGER	0	ja	ja
firing_delay_avg	slot	INTEGER	0	ja	ja
firing_delay_max	slot	INTEGER	0	ja	ja
condition	multislot	SYMBOL	-	ja	ja
preaction	multislot	SYMBOL	-	ja	ja
postaction	multislot	SYMBOL	-	ja	ja
info	slot	SYMBOL	-	nein	nein

Der Datentyp Transition mit den zugehörigen Attributen

Für den Namen einer Transition gilt gleiches, wie für den Namen der Stellen.
Mit Hilfe der Attribute *in_nodes* und *out_nodes* kann auf die Menge der Eingangs- bzw. Ausgangs-Stellen einer Transition zugegriffen werden. Für die Verwaltung dieser Attribute und der Attribute *in* und *out* gilt das bereits bei den Stellen gesagte. Im Kapitel 3.3 werden einige Beispiele angegeben, die die Einsatzmöglichkeiten dieser Mengen verdeutlichen werden.
Alle *delay*-Attribute können unabhängig voneinander abgefragt und gesetzt werden.
Die *condition*, *preaction* und *postaction* können abgefragt und gesetzt werden. Bei Anfragen sind Besonderheiten zu beachten, die im Kapitel 3.3.5.4 beschrieben werden. Zum Setzen der Attribute existiert auch eine erweiterte Funktionalität, die im Kapitel 3.4 erläutert wird. Es existiert beispielsweise die Möglichkeit, Berechnungen durchzuführen.

3.2.3 Kanten

Kante					
Attribut	Feldtyp	Datentyp	Inititalisierung	Abfragen	Setzen
id	slot	SYMBOL	-	nein	nein
name	slot	SYMBOL	-	-	-
from	slot	SYMBOL	-	ja	ja
to	slot	SYMBOL	-	ja	ja
annotation	multislot	SYMBOL	1[x]	ja	ja
info	slot	SYMBOL	-	nein	nein

Das Attribut Kante mit den zugehörigen Attributen

Für das Attribut *name* muß nur im Bedingungsblock ein Bezeichner vergeben werden. Die Namensangabe im Aktionsblock kann entfallen, da auf eine neue Kante nicht im gleichen Aktionsblock referiert werden darf.
Die Attribute *from* und *to* werden benötigt, damit der Start- und Zielkonten der Kante angegeben werden kann. Es muß sich hierbei jeweils um einen Bezeichner handeln, der schon vorher an eine Stelle oder Transition vergeben wurde. Es ist zu beachten, daß immer nur Kanten von einer Stelle zu einer Transition bzw. von einer Transition zu einer Stelle erlaubt sind. Es können auch Kantenbündel angelegt werden. Beispiele folgen im Verlauf dieses Kapitels.
Die Annotation einer Kante kann sowohl gesetzt als auch abgefragt werden. Welche Besonderheiten bei Anfragen zu beachten sind, wird im Kapitel 3.3.5.4 erläutert.

3.2.4 Mengen

Menge					
Attribut	Feldtyp	Datentyp	Inititalisierung	Abfragen	Setzen
name	slot	SYMBOL	-	-	-
object	multislot	SYMBOL	-	ja	ja
info	slot	SYMBOL	-	nein	nein

Der Datentyp Menge mit den zugehörigen Attributen

Die Mengen sind zur Verwaltung der Stellen und Transtionen gedacht. Möchte man beispielsweise Stellen oder Transitionen, deren Attribute eine bestimmte Eigenschaft besitzen, ermitteln, so kann man diese Objekte in einer Menge ablegen. Vorstellbar ist,

daß man ein Teilnetz aus dem System entfernen möchte, in das nur Kanten hineinführen aber aus dem keine Kanten herausführen. Es würde sich dabei um das Entfernen einer sogenannten Senke handeln. Der Einsatz von Mengen wird an einem Beispiel im Kapitel 5.1 verdeutlicht.
Für den Namen muß eine Bezeichner vergeben werden, damit auf die Menge im weiteren Verlauf zugegriffen werden kann.
Hinter dem Attribut *object* verbergen sich die Elemente einer Menge, die abgefragt und gesetzt werden können. Wird beispielsweise eine Menge angelegt, der der Bezeichner `?s` zugeordnet wurde, so erfolgt der Zugriff auf die eigentlichen Elemente der Menge auch über diesen Bezeichner. Dies ist möglich, da nur ein Attribut existiert, auf das der Zugriff erlaubt ist.

3.3 Syntax und Semantik

Wie bereits erwähnt ist die verwendete Syntax Lisp-ähnlich. Dies bedeutet, daß die gesamte Beschreibung durch geschachtelte Klammerausdrücke beschrieben wird. Jedes Konstrukt wird von einem Paar runder Klammern umschlossen. Nach der öffnenden Klammer folgt eine Schlüsselwort, welches für ein Konstrukt eindeutig identifizierend ist. Nachfolgend können die Parameter oder weitere Konstrukte folgen, so daß ein geschachtelter Klammerausdruck entstehen kann. Den Schluß eines jeden Konstrukts bildet die schließende Klammer. Schlüsselworte sind nicht als Bezeichner erlaubt.
Die Sprache wird im folgenden anhand von Beispielen eingeführt. Im Anhang B ist die formale Grammatik der Sprache in EBNF so dargestellt, daß sie sich strukturell mit der Gliederung der folgenden Abschnitte deckt. Die Überschriften sind mit in die Grammatik übernommen worden, damit ersichtlich ist, welche Konstrukte zu welchem Abschnitt gehören. Für die Darstellung der Konstrukte sind verschiedene Schriftarten gewählt worden, um die Übersichtlichkeit zu wahren. Elementare Elemente, zu denen die Zahlen und Bezeichner gehören, werden in *italic* dargestellt. Schlüsselworte werden **fett** gedruckt. Für Hilfskonstrukte, wurde die Schriftart `typewriter` gewählt.
Die Kernpunkte der Transformationssprache und damit auch die wesentlichen Blöcke einer Beschreibung sind die Regeln, die durch einen Kontrollmechansismus zu Transformationen zusammengefaßt werden können, und der Bedingungs- bzw. Aktionsblock dieser Regeln, der für eine Beschreibung bzw. Modifikation eines Ausschnitts eines Prädikat/Transitions-Neztes zur Verfügung stehen. Die Abbildung 3.1 auf der Seite 45 soll den Aufbau einer Beschreibung verdeutlichen.

- globale Informationen

  Es handelt sich um Informationen, die für jede Regel sichtbar sind. Dies gilt sowohl für die Mengen P_i, P_o, P_{io}, P und T als auch für neu angelegte Mengen. Globale Bezeichner können initialisiert werden, um sie beispielsweise beim Setzen der Prioritäten einer Regel zu verwenden. Die Informationen, die für die Steuerung der Zugriffsrechte benötigt werden, müssen für jede Regel sichtbar sein.

- Transformationen und Regeln

  Das Transformation-Konstrukt dient dazu, die Lesbarkeit einer Beschreibung zu verbessern. Die eigentliche Gruppierung der Regeln muß mit dem Kontrollmechanismus, der im Kapitel 3.5 erläutert wird, realisiert werden.

Die Regeln bilden den eigentlichen Kernpunkt einer Beschreibung. Sie werden in einen Bedingungs- und einen Aktionsblock unterteilt.

- Im Bedingungsblock wird ein Teil des System so exakt wie nötig beschrieben, um es anschließend im Aktionsblock zu modifizieren. Identifizierte Objekte können an Bezeichner gebunden werden, um auf sie im Aktionsblock oder bei Vergleichsoperationen zugreifen zu könnnen.
- Im Aktionsblock werden die Objekte, die im Bedingungsblock identifiziert und an Bezeichner gebunden wurden, modifiziert oder gelöscht. Weiterhin können neue Objekte angelegt werden.

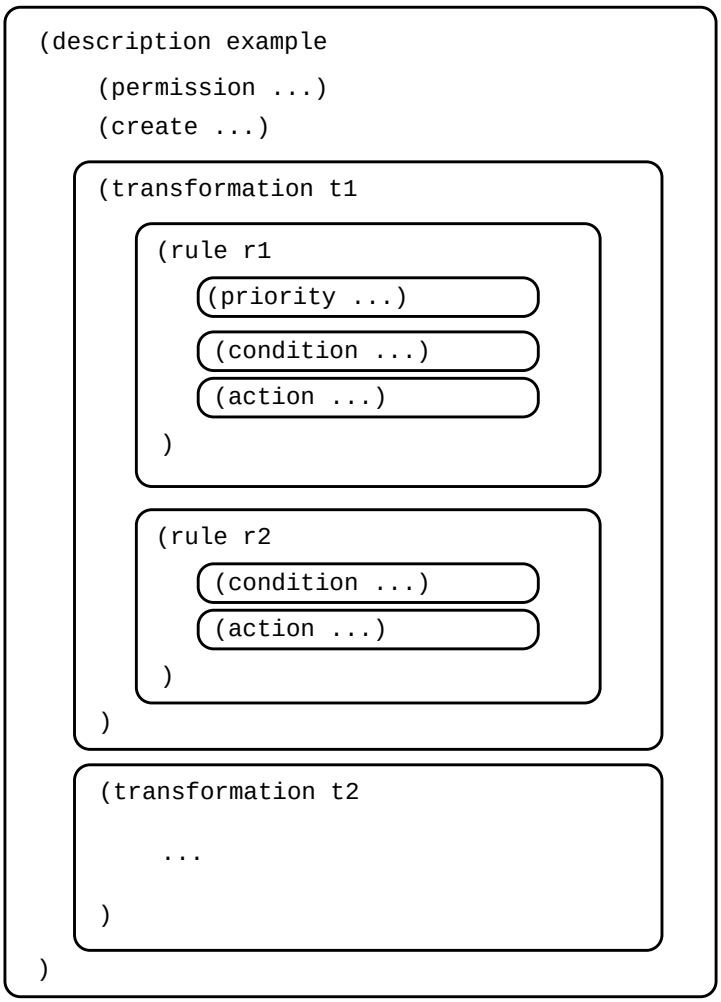

Abbildung 3.1: Aufbau einer Beschreibung

3.3.1 Elementare Konstrukte

Kommentare, die durch '//' eingeleitet werden und bis zum Zeilenende reichen, können zur Dokumentation der Transformationen genutzt werden. Die Verwendung der C++-

üblichen Kommentare resultiert daraus, daß die Grammatik zur Beschreibung der Annotationen mit eingebunden wurde, welche diese Kommentare erlaubt. Im Normalfall würde ein Kommentar durch ';' (Lisp-Kommentar) eingeleitet. Das Semikolon wird aber zur Trennung mehrerer Ausdrücke bei der *preaction* und *postaction* verwendet und konnte daher nicht mehr zur Einleitung der Kommentare eingesetzt werden.
Es werden mehrere Arten von Bezeichnern unterschieden. Dies resultiert zum einen darauß, daß die Grammatik zum Beschreiben der Annotationen vom Prädikat/Transitions-Netz-Editor vorgegeben wird. Anderseits muß beachtet werden, daß eine Beschreibung in die CLIPS-Notation übersetzt werden muß. Die Notation der Bezeichner wird im Anhang B gezeigt.

```
(description example

 (transformation t1
  (rule r1
   (condition
    (transition ...
                preaction x=x+1; y=2*y;)
   )
   ...
```

Die Bezeichner der Annotationen müssen sich an eine syntaktische Form halten, die im folgenden als *c_ident* (siehe auch Anhang B) bezeichnet wird. Weiterhin werden diese Bezeichner noch für die Namen der gesamten Beschreibung, der Transformationen und der Regeln verwendet. Bei `t1, r1 x` und `y` handelt es sich also jeweils um einen *c_ident*.

```
(description ...
 ?*x* = 7
 ?*y* = 8
  ...
```

Ein *global_ident* beginnt mit "?*" gefolgt von einem *c_ident* gefolgt von einem "*". Der Gültigkeitsbereich dieser Bezeichner umfaßt die gesamte Beschreibung.

```
(description ...

 (transformation ...
  (rule ...
   (condition
    (place ?p1
           in ?p1_in
           out ?p1_out)
    (transition ?t1
                in ?t1_in)
    (= 2 ?p1_in ?p1_out)
    (> 1 ?t1_in)
```

Bei einem *ident* handelt es sich um einen *c_ident*, dem ein "?" vorangestellt wird. Diese Bezeichner werden innerhalb der Regeln zum Identifizieren von Objekten und Attributen und im globalen Block für die Mengen benötigt.

```
(condition
 (place ?p)
)
(action
 (remove ?*p ?p*)
 ...
```

Die Bezeichner, die unmittelbar hinter dem "?" einen "$*$" besitzen sind an die Eingangs-Transitionen bzw. Eingangs-Stellen einer Stelle bzw. Transition gebunden. Für die Bezeichner, die mit einem "$*$" enden gilt, daß sie an die Ausgangs-Transitionen bzw. Ausgangs-Stellen einer Stelle bzw. Transition gebunden sind. Die Bindung erfolgt, sobald eine Stelle oder Transition im Bedingungsblock an einen Bezeichner gebunden wird (siehe auch Kapitel 3.2). Bei dem Beispiel würden alle Eingangs- und Ausgangs-Transitionen einer Stelle, die im Bedingungsblock an den Bezeichner `?p` gebunden wurde, gelöscht.
Ein Bezeichner darf innerhalb einer Regel nur einmal vergeben werden. Wird ein Bezeichner für eine Menge vergeben, so ist zu beachten, daß auf diese Menge von jeder Regel aus zugegriffen werden kann. Dies bedeutet, daß die Bezeichner, die für Mengen vergeben werden, für die gesamte Beschreibung eindeutig sein müssen, damit man unerwünschte Seiteneffekte vermeidet.
Man sollte die einzelnen Objekte nicht mit Variablen vergleichen, da man sowohl im Bedingungs- als auch im Aktionsblock immer Teile des Systems, das man zuvor als Prädikat/Transtions-Netz modelliert hat, beschreibt oder modifiziert. Im folgenden werden die einzelenen Blöcke einer Beschreibung erläutert.

3.3.2 Die Beschreibung

```
(description example
 ...
```

Jede Beschreibung beginnt mit dem Schlüsselwort **description** gefolgt von einem Dateinamen. Beim Übersetzen des Quellcodes wird eine Datei mit dem Namen *c_ident*.clp erzeugt, die dann zur Optimierung geladen werden kann. Hat man beispielsweise als Bezeichner für die Beschreibung `example` angegeben, so hat die übersetzte Datei, den Namen `example.clp`. Diese Datei wird für den Optimierungsprozeß geladen.

3.3.3 Globale Konstrukte

Nach dem Dateinamen können entweder globale Informationen, zu denen das Setzen von Zugriffsrechten (siehe auch Kapitel 3.5), das Erzeugen leerer Mengen und das Binden von Werten an globale Bezeichner gehören, oder die eigentlichen Transformationen folgen. Die Reihenfolge der Blöcke ist beliebig. Die entsprechende Zuordnung und Umsetzung der Blöcke erfolgt bei der Übersetzung in die CLIPS-Notation.

Permission

```
(permission t1:r1 t2:r1)
(permission t3:r1)
...
```

Das **permission**-Konstrukt im globalen Block wird benötigt, um die Voraussetzung für die Überprüfbarkeit der Zugriffsrechte zu schaffen. Mit dem ersten *c_ident* muß die Transformation und mit dem zweiten *c_ident* die Regel, die zu dieser Transformation gehört, benannt worden sein. Gilt also beispielsweise, daß die Regel `r1` zur Transformation `t1` gehört, so kann durch die Angabe von `(permission t1:r1)` im globalen Block im Bedingungsblock der Regel `r1`, die zur Transformation `t1` gehört, überprüft werden, ob die Zugriffsrechte gesetzt sind. Wenn eine Transformation aus mehreren Regeln besteht, so wird es nie der Fall sein, daß man den Zugriff für alle Regeln gleichzeitig erlauben möchte, da man dann nicht mehr voraussagen kann, welche Regel dieser Transformation wann feuert. Eine Transformation besteht aber gerade dann aus mehreren Regeln, wenn man eine komplexe Transformation beschreibt, die ein kontrolliertes Feuern der Regeln benötigt, um das gewünschte Ergebnis zu bekommen. Man wird also immer nur für eine dieser Regeln den Zugriff erlauben. Wurde diese Regel dann aktiviert, so können die Rechte für die anderen Regeln der Transformation gesetzt werden. Sollen alle Regeln gleichberechtigt sein, so kann man auf die Zugriffsrechete verzichten. Im Kapitel 5.1 wird der Einsatz des Zugriffsrechte an einem Beispiel verdeutlicht. Weiterhin wird bei der Beschreibung des Kontrollmechanismus im Kpaitel 3.5 auf den Einsatz der Zugriffsrechte eingegangen.

Create

```
(create ?Sp ?St)
...
```

Mit dem **create**-Konstrukt können zusätzlich zu den bereits implizit vorhandenen Mengen neue Mengen, die nach dem Erzeugen leer sind, angelegt werden. Beispielsweise werden mit `(create ?Sp ?St)` die beiden leeren Mengen `?Sp` und `?St` angelegt. Auf eine Menge, die einmal angelegt wurde, kann in allen Regeln zugegriffen werden. Ein umfangreiches Anwendungsbeispiel ist im Kapitel 5 zu finden.

Globale Bezeichner

Es können globale Bezeichner gesetzt werden, die in jeder Regel verwendet werden können. Diese Bezeichner können beispielsweise zum Setzen von Prioritäten verwendet werden.

3.3.4 Transformationen und Regeln

Transformationen

```
(transformation t1
...

(transformation t2
...
```

Mit dem Schlüsselwort **transformation** wird die eigentliche Transformation eingeleitet. Eine Transformation besteht aus einer oder mehreren Regeln. Der Transformation muß ein Name zugeordnet werden, der beim Setzen der Zugriffsrechte Verwendung

findet. Das **transformation**-Konstrukt hat nur die Aufgabe die Lesbarkeit des Quelltextes zu erhöhen. Es ist wesentlich, daß keinerlei Gruppierungseffekt erzielt wird. Alle Regeln aller Transformationen sind gleichberechtigt, sofern dies nicht mit den im Kapitel 3.5 beschriebenen Möglichkeiten verändert wird.

Regeln

```
(rule r1
 ...

(rule r2
 ...
```

Jede Regel beginnt mit dem Schlüsselwort **rule** gefolgt von einem Namen, der der Regel gegeben werden muß. Der Anwender hat optional die Möglichkeit der Regel eine Priorität (siehe auch Kapitel 3.5) zuzuordnen. Die eigentliche Regel wird durch einen Bedingungs- und einen Aktionsblock gebildet (siehe auch Kapitel 2.3.2).

Prioritäten

```
(priority 10)
(priority -20)
(priority ?*x*)
```

Als Priorität wird entweder eine ganze Zahl im Bereich [-5000, 5000] zugelassen oder ein globaler Bezeichner. Der Wert, der an den Bezeichner gebunden wurde, muß ebenfalls im angegebenen Bereich liegen. Für den Fall, daß keine Priorität vergeben wird, hat die Regel die Priorität 0.

Aufgaben des Bedingungsblocks

Mit Hilfe der Konstrukte des Bedingungsblocks (siehe auch Kapitel 2.3.2) kann präzise beschrieben werden, wie ein Ausschnitt des Prädikat/Transitions-Netzes aussehen muß, damit die Bedingung der Regel erfüllt ist. Da jedes Konstrukt im Kapitel 3.3.5 erläutert wird, wird hier nur kurz auf die verschiedenen Gruppen von Konstrukten eingegangen.

- Es kann überprüft werden, ob die Zugriffsrechte für eine Regel gesetzt sind. Details hierzu findet man auch im Kapitel 3.5.
- Es können detailliert Stellen, Transitionen, Kanten und Menge sowie deren Attribute beschrieben werden.
- Es können die Eigenschaften von Mengen überprüft werden, und es können Mengenoperationen durchgeführt werden. Die Mengenoperationen sowie weitere Eigenschaften von Mengen werden im Kapitel 3.3.5.5 beschrieben.
- Zur Verknüpfung der Anfragen können die logischen Operationen **and**, **or** und **not** verwendet werden, Die *default*-Verknüpfung ist *and*.
- Es stehen diverse Vergleichsoperationen zur Verfügung mit deren Hilfe Objekte und deren Attribute verglichen werden können.

Aufgaben des Aktionsblock

Ist die Bedingung einer Regel erfüllt, so werden die Aktionen, die im Aktionsblock (siehe auch Kapitel 2.3.2) beschrieben werden können, ausgeführt. Auch hier ist wieder eine Einteilung der Konstrukte in Gruppen möglich.

- Die Zugriffsrechte können neu gesetzt oder geändert werden. Details hierzu sind im Kapitel 3.5 zu finden.
- Es können neue leere Mengen erzeugt werden. Das Konstrukt wurde bereits im Kapitel 3.3.3 erklärt.
- Es können Stellen, Transitionen, Kanten und Mengen neu angelegt werden.
- Existierende Objekte können verändert werden. Beispielsweise können die Attribute eines existierenden Objekts neu gesetzt werden.
- Objekte können gelöscht werden.

Ein Beispiel soll zur Verdeutlichung der auf den vorangegangnen Seiten beschriebenen Konstrukten dienen. Nachfolgend wird auf den Bedingungs- und Aktionsblock näher eingegangen.

```
(description example

 (create ?St ?Sp)
 (global ?*x*=4)

 (permission t1:r1 t2:r2)

 (tranformation t1

  (rule r1
   (priority 10)
   (condition ... )
   (action ... )
  )

  (rule r2
   (condition ... )
   (action ... )
  )
 )

 (transformation t2 ... )
)
```

Der Beschreibung wurde der Name `example` zugeordnet. Die übersetzten Transformationen werden daher unter `example.clp` abgespeichert. Im Beispiel werden zuerst globale Informationen beschrieben. Hierzu gehört das Anlegen der leeren Mengen `?St`

und `?Sp`, das Zuordnen der Zahl 4 zu dem globalen Bezeichner `?*x*` und das Setzen der Zugriffsrechte für die Regeln `r1` und `r2` der Transformation `t1`. Für die Regel `r1` wird eine erhöhte Priorität von 10 vereinbart. Anschließend können die Bedingungen, die erfüllt sein müssen, damit die Regel feuert, beschrieben werden. Die anschließend auszuführenden Aktionen werden im Aktionsblock beschrieben. Es folgt eine weitere Transformation `t2`. Die Reihenfolge, in der die Transformationen angegeben werden, hat keinerlei Einfluß auf die Abarbeitungsreihenfolge.

3.3.5 Der Bedingungsblock

Im Bedingungsblock kann detailliert ein Ausschnitt eines Prädikat/Transitions-Neztes beschrieben werden. Zusätzlich stehen noch Funktionen für Anfragen zur Verfügung. Es sind sowohl strukturelle Eigenschaften des Neztes als auch Annotationen beschreibbar. Im folgenden werden die einzelnen Konstrukte des Bedingungsblocks beschrieben. Es muß gelten, daß alle Anfragen das Resultat '*wahr*' liefern, damit eine Regel feuern kann. Möchte man im Aktionsblock ein Attribut eines Objekts mit einem Wert belegen, der dem Wert eines Attributs eines Objekts, das im Bedingungsblock beschrieben wurde, entspricht, so gibt es zwei Möglichkeiten. Einerseits kann es sein, daß man den Wert direkt angeben kann, so ist dies sicherlich auch im Aktionsblock beim Setzen des entsprechenden Attributs möglich. Andererseits ist es möglich, daß man ein Attribut nicht direkt beschreiben kann, den Wert aber im Aktionsblock einem Attribut zuweisen möchte. In diesem Fall kann man den Wert an einen Bezeichner binden und diesen beim Setzen des entsprechenden Attributs angeben. Der Bezeichner muß also für den Fall, daß man den Wert im Aktionsblock benötigt, angegeben werden, auch wenn der Wert zu keiner Vergleichsoperation herangezogen werden soll. Es folgen die Konstrukte, die im Bedingungsblock eingesetzt werden können.

3.3.5.1 Zugriffsrechte abfragen

```
(pemission)
```

Durch die Angabe von `(permission)` wird überprüft, ob im globalen Block der Zugriff auf die Regel erlaubt wurde. Wird beispielsweise in der Regel `r1`, die zur Transformation `t1` gehört die Anfrage `(permission)` gestellt, so ist deren Resultat '*wahr*', wenn im globalen Block `(permission t1:r1)` angegeben wurde. Weiterhin muß gelten, daß beim Abarbeiten der Zugriffsrechte keine weiteren Angaben vorher abgearbeitet werden müssen (siehe Kapitel 3.5).

3.3.5.2 Objekte beschreiben

Zu den Objekten, die im Bedingungsblock beschrieben werden können, gehören die Stellen, Transitionen, Kanten und Mengen. Da im Kapitel 3.2 die zur Verfügung stehenden Datentypen ausführlich beschrieben wurden, werden im folgenden nur Besonderheiten hervorgehoben. Für alle Objekte gilt, daß unmittelbar nach dem Schlüsselwort ein Bezeichner angegeben werden muß, durch den das Objekt im weiteren Verlauf identifiziert wird.

Stellen

```
(place ?p1)
(place ?p2 in 1 out 1 mark 2[2])
```

Bei dem Attribut **mark** kann entweder ein Bezeichner vergeben werden, um später im Bedingungsblock einen Vergleich durchzuführen, oder die Marke kann direkt beschrieben werden. Wie eine Marke aussehen muß, wurde bereits im Kapitel 2.1.3 beschrieben. Beispielsweise wird durch `(place ?p1 in 1 out 1 mark 2[2])` gefordert, daß eine Stelle existieren muß, die genau eine Eingangs- und eine Ausgangs-Transition und zwei Marken der Form `[2]` besitzt.

Transitionen

```
(transition ?t1)
(transition ?t2 in 2 out 2
                preaction x=x+1; postaction ?t2_post)
```

Bezüglich der Beschreibung der **condition**, **preaction** und **postaction** gilt gleiches, wie für die Beschreibung der Marken einer Stelle. Durch die Anfrage `(transition ?t1)` wird beispielsweise gefordert, daß eine Transition existieren muß. Bezüglich der Attribute wurden keinerlei Einschränkungen angegeben, so daß jede existierende Transition die Anforderung erfüllt.

Kanten

```
(place ?p)
(transition ?t)
(edge ?e from ?p to ?t)
```

Zusätzlich zu dem Bezeichner für die Kante muß auf jeden Fall der Start- und Zielknoten angegeben werden. Für beide gilt, daß sie vorher im gleichen Bedingungsblock an einen Bezeichner gebunden worden sein müssen. Für die Beschreibung einer Annotation gilt das Gleiche, wie für die Beschreibung der Marken einer Stelle.
Im vorherigen Beispiel wird gefordert, daß eine Kante zwischen der Stelle `?p` und der Transition `?t` existieren muß. Bezüglich einer möglichen Annotation wurde keine Angabe gemacht. Dies bedeutet, daß die Annotation beliebig sein kann.

Mengen

```
(set ?s1)
```

Durch die Angabe von `(set ?s1)` wird gefordert, daß die Menge `?s1` existieren muß. Falls man außer den implizit vorhandenen Mengen (siehe Kapitel 3.2) weitere Mengen benötigt, so müssen diese vor ihrer Benutzung durch ein **create** erzeugt werden. Der Zugriff auf die implizit vorhandenen Mengen ist immer möglich. Es muß also beispielsweise nicht `(set ?P)` angegeben werden, wenn man auf die Elemente der Menge zugreifen möchte. Zur Überprüfung weiterer Eigenschaften der Mengen stehen die im Kapitel 3.3.5.5 beschriebenen Konstrukte zur Verfügung. Ein möglicher Einsatz von Mengen wird im Kapitel 5.1 an einem Beispiel verdeutlicht.

3.3.5.3 and, or und not Verknüpfungen

Wie bereits im Kapitel 3.3.4 erwähnt, kann die Einstellung, daß alle Anfragen *and*-Verknüpft werden, verändert werden. Es ist eine beliebige rekursive Schachtelung von **and**, **or** und **not** möglich. Die doppelte Negation `(not (not ...))` ist nicht erlaubt. Es stehen aber weiterhin alle Möglichkeiten, die der Bedingungsblock bietet, für die Anfragen zur Verfügung.

```
(place ?p_v
       out ?p_v_out)
(place ?p_n
       in ?p_n_in
       mark_number ?p_n_mark_number)
(neq ?p_v ?p_n)
(or
 (and
  (= 1 ?p_n_in)
  (not (in ?Po ?p_n))
  (= 0 ?p_n_mark_number))
 (and
  (= 1 ?p_v_out)
  (not (in ?Pi ?p_v)))
 )
```

Das vorherige Beispiel soll die Verwendung der Konstrukte verdeutlichen. Es wird gefordert, daß die beiden Stellen `?p_v` und `?p_n` existieren müssen. Ein Teil der Attribute der beiden Stellen werden Bezeichner gebunden. Es wird anschließend überprüft, ob ein Kante in die Stelle `?p_n` hineinführt, ob es sich bei der Stelle nicht um einen Ausgabe-*Port* handelt und ob die Stelle keine Marke enthält. Alternativ kann gelten, daß die Stelle `?p_v` eine Ausgangskante besitzt und kein Eingabe-*Port* ist.
Bei der Verwendung von **or** ist zu beachten, daß CLIPS intern aus dieser Regel soviel Regeln generiert, wie Anfragen innerhalb des **or**-Konstrukts gestellt werden. Man hat also lediglich weniger Schreibarbeit, die Anzahl der Regeln wird nicht verringert. Weiterhin ist es nicht möglich, innerhalb eines **or**-Konstrukts Bezeichner zu binden.

3.3.5.4 Vergleiche

Um Attribute vergleichen zu können, werden diverse Vergleichsfunktionen benötigt, die sich wie folgt unterteilen lassen:

- =, <>, >, <, >=, <=

  Die Attribute, denen der Datentyp INTEGER zugeordnet wurde, können mit diesen Vergleichsfunktionen überprüft werden.

  Zusätzlich hat man noch die Möglichkeit, die Anzahl der Elemente einer Menge zu überprüfen. Hierzu kann die **card**-Funktion verwendet werden (siehe auch Kapitel 3.3.5.5). Als Argument erwartet die **card**-Funktion ein Objekt vom Datentyp Menge, dessen Elementanzahl dann bestimmt wird.

Weiterhin ist es bei den Vergleichsfunktionen $=, <>, >, <, >=, <=$ möglich, daß man mehrere Werte auf einmal miteinander vergleicht.

```
(place ?p1
       in ?p1_in out ?p1_out)
(place ?p2
       in ?p2_in out ?p2_out)
(>= ?p1_in ?p2_in ?p2_out 3)
```

Im vorherigen Beispiel wurden die Bezeichner `?p1` und `?p2` jeweils an eine Stelle gebunden und für die Anzahl der Eingangskanten die Bezeichner `?p1_in` und `?p2_in` vergeben. Weiterhin wurde für die Anzahl der Ausgangskanten die Bezeichner `?p1_out` und `?p2_out` vergeben. Durch `(>= ?p1_in ?p2_in ?p2_out 3)` kann man überprüfen, ob für die Stellen `?p1` und `?p2` die Anzahl der Eingangs- und für die Stelle `?p2` die Anzahl der Ausgangskanten größer oder gleich 3 ist. Da bei den Vergleichsfunktionen $>, <, >=$ und $<=$ jeweils die gesamte Folge überprüft wird, muß zusätzlich `?p1_in` $>=$ `?p2_in` $>=$ `?p2_out` $>= 3$ gelten, damit man als Resultat der Anfrage '*wahr*' erhält.

Bei $=$ und $<>$ werden alle Elemente mit dem **ersten** Element verglichen. Die Anfrage `(= 3 (card ?P))` würde beispielsweise '*wahr*' liefern, wenn das System aus 3 Stellen besteht.

- **eq, neq**

  Möchte man Attribute vom Typ SYMBOL (siehe Kapitel 3.2) vergleichen, so stehen unter anderem diese beiden Operationen zum Test auf Gleichheit oder Ungleichheit zur Verfügung.

  Auch hier werden alle Elemente mit dem **ersten** Element verglichen.

  Diese beiden Funktionen eignen sich in erster Linie zum Vergleichen von Attributen, die den Feldtyp *slot* besitzen. Für den Fall, daß man Attribute, die den Feldtyp *multislot* besitzen, vergleichen möchte, sollte man beachten, daß dies nur positiv verlaufen kann, wenn exakte Gleicheit bezüglich der Position der Symbole und der eigentlichen Symbole gegeben ist.

  Stellt man beispielsweise die Anfragen `(place ?p1) (place ?p2)`, so liefert die Anfrage `(eq ?p1 ?p2)` '*falsch*', falls es sich um zwei verschiedene Stellen handelt und '*wahr*', wenn es sich um dieselbe Stelle handelt. Es wird also keine Aussage über die Gleichheit der Attribute getroffen, sondern nur, ob es sich um ein oder um zwei Objekte handelt. Die Attribute müssen separat überprüft werden.

  Welches Resultat erhält man nun, wenn z. B. die *preaction* (`x=(2*a)+b;`) einer Transition an den Bezeichner `?p1_preaction` und die *preaction* (`x=b+(a+a);`) einer anderen Transition an den Bezeichner `?p2_preaction` gebunden ist? In diesem Fall würde der Vergleich `(eq ?p1_preaction ?p2_preaction)` das Ergebnis '*falsch*' liefern, obwohl die Argumente unter mathematische Gesichtspunkten gleich sind. Dies liegt daran, daß die normalen Funktionen des Expertensystems nur für einen einfachen Mustervergleich ausgelegt sind.

Daher sollte man für den Vergleich von Annotationen immer die im folgenden Unterpunkt beschriebenen Vergleichsfunktionen verwenden, es sei denn, man kann die zu überprüfende Annotation exakt beschrieben. Dieser Fall tritt öfter auf als man im ersten Moment meinen könnte. Dies zeigen beispielsweise die Anwendungsbeispiele im Kapitel 5. Verzichtet man auf die Vergleichsfunktionen und gibt die Annotation direkt beim entsprechenden Attribut an, so stehen einem alle Möglichkeiten, die die Grammatik des Prädikt/Transitions-Netz-Editors zur Beschreibung der Annotationen bietet, zur Verfügung (siehe auch [rust 95]).

- **meq, aeq, ceq, peq**

  Das vorherige Beispiel hat gezeigt, daß man zum Vergleichen der Annotationen zusätzliche Funktionen benötigt, die für die spezielle Aufgabe ausgelegt sind. Mit **meq** können die Marken der Stellen und mit **aeq** die Annotationen an den Kanten verglichen werden. Um die *condition* einer Transition zu vergleichen, kann **ceq** verwendet werden. Mit **peq** können sowohl die *preaction* als auch die *postaction* einer Transition verglichen werden. Alle Funktionen verlangen genau zwei Argumente, wobei das erste Argument ein Bezeichner sein muß, der überprüft werden soll. Das zweite Argument kann entweder ein Bezeichner sein oder es kann direkt die Annotation angegeben werden, mit der das erste Argument verglichen werden soll.

  An die zu vergleichenden Annotationen werden einige Bedingungen gestellt. So muß für die Marken der Stellen und die Annotationen an den Kanten gelten, daß für die einzelnen Tupel gilt, daß sie jeweils die Kardinalität 1 haben müssen. Dies resultiert aus der Implementierung der Vergleichsfunktionen **meq** und **aeq**. Abgesehen von dem erhöhten Schreibaufwand wird die Funktionalität nicht eingeschränkt.

  Möchte man beispielsweise für eine Annotation an einer Kante überprüfen, ob sie die Form `1[x],1[y,z]` hat, so ist dies durch `(aeq ?annotation 1[x],1[y,z])` möglich. Voraussetzung ist, daß die zu überprüfende Annotation vorher an den Bezeichner `?annotation` gebunden wurde. Die Anfrage liefert auch '*wahr*', wenn `?annotation` die Form `1[y,z],1[x]` hat, da die Reihenfolge nicht ausschlaggebend ist.

  Soll diese Überprüfung z. B. für die Annotation `2[x],1[y]` durchgeführt werden, so ist dies entweder durch die bereits erwähnte direkte Angabe bei der Annotation der entsprechenden Kante möglich, oder falls die Annotation an einen Bezeichner gebunden wurde, so muß sie und das zweite Argument die Form `1[x],1[x],1[y]` besitzen, damit man das Resultat '*wahr*' erhält. Die Reihenfolge der Tupel kann auch hier wieder beliebig und bei beiden Argumenten unterschiedlich sein.

  Für die Marken der Stellen gilt gleiches wie für die Annotationen der Kanten.

  Noch etwas komplizierter wird das Ganze, wenn man die *condition*, *preaction* und *postaction* einer Transition überprüfen möchte. In diesem Fall kommt zusätzlich zur möglichen Positionsvielfalt der einzelnen Bezeichner, Zahlen und Operatoren noch hinzu, daß mit unterschiedlichen Symbolen mathematisch gleiche Ausdrücke beschrieben werden können, wie dies auch das Beispiel beim Unterpunkt **eq, neq** auf der vorherigen Seite zeigt, bei dem zwei *preactions* verglichen werden sollen. Um dieses Problem zu lösen, wurde das Computer-Algebra-System MuPAD

in den Analyse- und Optimierungsprozeß mit eingebunden (siehe auch Kapitel 4.1). MuPAD bietet die Möglichkeit Ausdrücke zu normalisieren, so daß sich der Vergleich der resultierenden Ausdrücke wieder auf einen reinen Mustervergleich beschränkt.

Für die *condition* gilt, daß die Vergleichsoperationen $<, <=, >, >=, !=, ==$ verwendet werden können und die einzelnen Vergleiche durch &&, ||, ! verknüpft werden können. Die Klammerung der Ausdrücke ist auch möglich.

Wenn z.B. die *condition* `((2*a<b) && (e>=f))` an den Bezeichner `?condition` gebunden ist, so liefert die Anfrage `(ceq ?condition ((f<=e) && a+a<b))` das Resultat '*wahr*'.

Wenn die *preaction* und *postaction* mit **peq** überprüft werden sollen, so dürfen beide nur aus mathematischen Ausdrücken bestehen. Diese Einschränkung resultiert aus der Implementierung von **peq**.

Sei beispielsweise die *preaction* `x=a+b+b; y=a*a;` an den Bezeichner `?preaction` gebunden. Die Anfrage `(peq ?preaction y=2*a; x=a+2*b;)` würde das Resultat '*wahr*' liefern.

Es ist jeweils darauf zu achten, daß man sich bei allen Annotationen an die Grammatik des Prädikat/Transitions-Netz-Editors halten muß. Die erweiterte Funktionalität, zu der z. B. das Öffnen von Dateien und verschiedene andere Operationen gehören (siehe auch [rust 95]) kann nur verwendet werden, wenn die Annotation direkt beim entsprechenden Attribut angegeben wird und keine Überprüfung durch die Vergleichsoperationen durchgeführt werden soll.

3.3.5.5 Mengen und die auf sie anwendbaren Funktionen

Die Mengen und die auf sie anwendbaren Funktionen, bilden einen großen Block der Sprache. Zusätzlich zu der Konstruktorfunktion **create**, die bereits im Kapitel 3.3.3 erläutert wurde, existieren noch Zugriffsfunktionen, die nur im Bedingungsblock verwendet werden dürfen, und Transformationsfunktionen, die sowohl im Bedingungsblock als auch im Aktionsblock verwendbar sind.

Die Zugriffsfunktionen

Die Zugriffsfunktionen **empty, in**, **subset** und **card** können verwendet werden, um Mengen auf bestimmte Eigenschaften zu überprüfen. Die Mengen können entweder direkt durch den entsprechenden Bezeichner angegeben werden, oder das Argument kann eine Menge sein, die als Resultat von einer Transformationsfunktionen geliefert wird.
Mit der **subset**-Funktion kann überprüft werden, ob eine Menge Teilmenge einer anderen Menge ist. Es wird überprüft, ob das zweite Argument Teilmenge des ersten Arguments ist. Die Funktion **empty** erwartet genau ein Argument, für das überprüft wird, ob die Eigenschaft der leeren Menge erfüllt ist. Die Funktion **in** dient zum Überprüfen, ob ein Element in einer Menge enthalten ist. Das erste zu übergebende Argument ist die Menge, die überprüft werden soll, und für das zweite Argument, bei dem es sich um ein einzelnes Objekt handelt, soll überprüft werden, ob es in der Menge enthalten ist. **Card** ist eine weitere Zugriffsfunktion, mit der die Elementanzahl einer Menge

bestimmt werden kann. Die Funktion kommt im Rahmen der Vergleiche zum Einsatz (siehe Kapitel 3.3.5.4).

```
(place ?p)

(subset ?Sx ?Sy)
(empty ?Sx)
(in ?Pi ?p)
```

Seien beispielsweise `?Sx` und `?Sy` Mengen, so wird mit `(subset ?Sx ?Sy)` überprüft, ob die Menge `?Sy` Teilmenge von `?Sx` ist. Duch `(empty ?Sx)` wird beispielsweise überprüft, ob die Menge `?Sx` leer ist. Wenn man also beispielsweise für eine vorher im Bedingungsbock identifizierte Stelle, die an den Bezeichner `?p` gebunden wurde, wissen möchten, ob `?p` in der Menge der Eingangs-Ports enthalten ist, so wird dies durch die Anfrage `(in ?Pi ?p)` realisiert.

Die Transformationsfunktionen

Mit den Transformationsfunktionen **union** ($\cup$), **intersection** ($\cap$) und **difference** ($\setminus$) können aus existierenden Mengen neue veränderte Mengen gewonnen werden. Jede dieser Funktionen erwartet genau zwei Argumente. Bei den Argumenten muß es sich entweder um einen Bezeichner handeln, der an eine Menge oder ein Objekt gebunden wurde, oder es handelt sich um eine Menge, die sich als Resultat einer Transformationsfunktion ergibt. Im Gegensatz zu den Zugriffsfunktionen können die Transformationsfunktionen sowohl im Bedingungs- als auch im Aktionsblock verwendet werden. Im Bedingungsblock muß gelten, daß eine Transformationsfunktion nur innerhalb einer Zugriffsfunktion verwendet wird, da alle Anfragen im Bedingungsblock das Resultat '*wahr*' oder '*falsch*' liefern müssen. Auf die Verwendung der Funktionen im Aktionsblock wird im Kapitel 3.3.6 eingegangen.
Mit der Funktion **union** wird die Vereinigung zweier Mengen durchgeführt. Falls Elemente existieren, die sowohl in der einen als auch in der anderen Menge vorkommen, so enthält die resultierende Menge dieses Element genau einmal. Es handelt sich also nicht um eine Vereinigung von Multimengen. Die Funktion **intersection** wird verwendet, um die Schnittmenge zweier Mengen zu bestimmen. Die Menge, die sich beim Anwenden der Funktion **difference** ergibt, enthält alle Elemente der Menge, die mit dem ersten Argument identifiziert wird, ohne alle Elemente der Menge, die mit dem zweiten Argument identifiziert wird.
Das Beispiel auf der nächsten Seite soll einen Einblick in die Verwendung von Mengen geben. Die vor den Zeilen stehenden Buchstaben werden nur als Referenzen benötigt und gehören nicht zur Beschreibung. Bei `a)` wird überprüft, ob die Stelle `?p_1` zu der Menge der Eingangs-Stellen der Transition `?t` gehört. Mit der Abfrage `b)` kann festgestellt werden, ob die Stellen `?p_1` und `?p_2` keine gemeinsame Eingangs-Transition besitzen. D. h. es existiert keine Transition, von der eine Kante zu `?p_1` und `?p_2` verläuft. Mit der Abfrage `c)` kann sichergestellt werden, daß es sich bei den Eingangs- und Ausgangs-Stellen der Transition `?t` weder um Eingangs- noch um Ausgangs-Ports handelt. Bei `d)` wird überprüft, ob die Menge der Ausgangs-Transitionen von der Stelle `?p_2` Teilmenge der Ausgangs-Transitionen der Stelle `?p_1` ist. Bei `e)` wird die Anfrage

von d) umgekehrt. Die Anfragen d) und e) zusammen überprüfen, ob die Menge der Ausgangs-Transitionen beider Stellen identisch sind.

```
Beispiele:
   (condition
    (place ?p_1)
    (place ?p_2)
    (neq ?p_1 ?p_2)
    (transition ?t)

a)  (in ?*t ?p_1)
b)  (empty (intersection ?*p_1 ?*p_2))
c)  (empty (intersection (union ?*t ?t*) (union ?Pi ?Po)))
d)  (subset ?p_1* ?p_2*)
e)  (subset ?p_2* ?p_1*)
   )
```

Wie bereits erwähnt wird ohne die explizite Verwendung von *and*, *or* oder *not* immer die *and*-Verknüpfung verwendet. Um die Bedingung der Regel zu erfüllen, müßten daher alle Anfragen das Resultat '*wahr*' liefern. Ein umfangreiches Beispiel, bei dem der Einsatz von Mengen verdeutlicht wird, ist im Kapitel 5 zu finden.

3.3.6 Der Aktionsblock

Nachdem die Konstrukte des Bedingungsblocks beschrieben sind, müssen noch die Konstrukte des Aktionsblocks erläutert werden. Die wesentlichen Aufgaben, die innerhalb des Aktionsblocks beschrieben werden, sind das Anlegen, Verändern und Löschen von Stellen, Transitionen, Kanten und Mengen. Wie bereits bei der Beschreibung des Bedingungsblocks erwähnt, müssen Attribute, die beim Anlegen oder Modifizieren von Objekten eingesetzt werden sollen, im Bedinungsblock an einen Bezeichner gebunden werden. Alternativ existiert die Möglichkeit, die Werte direkt zu setzen.

3.3.6.1 Zugriffsrechte setzen

```
(permission t1:r2 t1:r3)
(permission save t1:r2 t1:r3)
```

Um die Zugriffsrechte während des Optimierungsprozesses zu ändern, kann das **permission**-Konstrukt verwendet werden. Die Notation zum Setzen der neuen Zugriffsrechte ist mit der, die auch im globalen Block verwendet wird, identisch. Dies bedeutet, daß das Setzen der Zugriffsrechte durch die Angabe des Transformationsnamens gefolgt von einem Doppelpunkt gefolgt vom Regelnamen gesetzt wird. Möchte man die Zugriffsrechte nicht komplett neu setzen, sondern nur kurzzeitig eine neue Regelgruppe aktivieren und danach zum alten Zustand zurückkehren, so kann dies durch die Angabe des Schlüsselworts **save** direkt nach dem Schlüsselwort **permission** erreicht werden. Durch `(permission t1:r2 t1:r3)` werden die Zugriffsrechte für die Regeln `r2` und `r3` der Transformation `t1` gesetzt. Die alten Zugriffsrechte werden überschrieben. Bei `(permission save t1:r2 t1:r3)` werden die alten Zugriffsrechte vorab gesichert. Die

Verwendung der Konstrukte, die die Zugriffsrechte betreffen, werden ausführlich im Kapitel 3.5 erläutert.

3.3.6.2 Objekte hinzufügen

Im Aktionsblock können neue Stellen, Transitionen, Kanten und Mengen angelgt werden. Das Setzen der zugehörigen Attribute ist auch möglich. Bei Stellen, Transitionen und Mengen muß jeweils nach dem entsprechenden Schlüsselwort ein Bezeichner angegeben werden. Es muß gelten, daß der Bezeichner noch nicht vergeben wurde. Für den Fall, daß für ein Objekt eines Typs bereits der gleiche Bezeichner im Bedingungsblock vergeben wurde, wird davon ausgegangen, daß das betreffende Objekt verändert werden soll. Dies ist nur mit dem **modify**-Konstrukt, das im nächsten Abschnitt beschrieben wird, möglich. Der gleiche Bezeichner darf aber auch nicht im Aktionsblock für verschiedene Objekte vergeben werden. Es würde sich das Problem der Mehrdeutigkeit ergeben.

Stellen

```
(add (place ?p))
(add (place ?p mark 2[3]))
```

Der Bezeichner direkt hinter dem Schlüsselwort **place** wird benötigt, falls im gleichen Aktionsblock eine Kante angelegt werden soll, bei der die Stelle Start- oder Zielknoten sein soll. Wird ein Attribut nicht gesetzt, so wird es mit den im Kapitel 3.2 beschriebenen *default*-Werten initialisiert. Auf die Syntax, die zur Beschreibung der Marken verwendet werden kann, wurde bereits im Kapitel 2.1 eingegangen. Beispielsweise wird durch `(add (place ?p mark 2[3]))` eine Stelle angelegt, die zwei Marken der Form `[3]` enthält.

Transitionen

```
(add (transition ?t
                 condition (a<b)
                 preaction a=a+1;
                 postaction z=2*a;))
```

Generell gilt für die Transitionen gleiches wie für die Stellen. Hinzu kommen aber noch einige Besonderheiten, die die *condition*, die *preaction* und die *postaction* betreffen. Es existiert die Möglichkeit, Berechnungen beim Anlegen der Annotationen mit einzubeziehen. Ausführlich wird das Thema „Termersetzung und Berechnungen“ im Kapitel 3.4 erläutert. Da bei der Sprache zur Beschreibung der Annotationen, die vom Prädikat/Transitions-Netz-Editor vorgegeben wird, auch runde Klammern zur Vergabe der Präferenzen eingesetzt werden, mußte an dieser Stelle von der allgemeinen Notation abgewichen werden. Es kommen statt dessen eckige Klammern zum Einsatz. Durch `(add (transition ?t condition (a<b) preaction a=a+1; postaction z=2*a;))` wird beispielsweise eine neue Transition mit der angegebenen *condition*, *preaction* und *postaction* hinzugefügt.

Die *condition* einer Transition kann aus Bezeichnern, der direkten Angabe der Annotation und einer neu berechnenten Annotation zusammengesetzt werden. Zur Verknüpfung können &&, || und ! verwendet werden.
Die *preaction* und *postaction* kann durch die direkte Angabe einer Annotation oder die Verwendung von Bezeichnern gesetzt werden. Soll die Annotation neu berechnet werden, so kommt MuPAD zum Einsatz. Beispiele bei denen das **calculate**-Konstrukt, das zur Berechnung der *preaction* und *postaction* eingesetzt werden kann, zum Einsatz kommt, sind im Kapitel 3.4 zu finden.

Kanten

```
(add (edge form ?p to ?t annotation [y]))
(add (edge from ?*p to ?t*))
```

Mit dem **edge**-Konstrukt ist es möglich Kanten zwischen Stellen und Transtionen anzulegen. Hinter dem Schlüsselwort **edge** muß kein Bezeichner angegeben werden, da Kanten im Gegensatz zu Stellen und Transitionen nicht unmittelbar im gleichen Aktionsblock verändert werden können. Es ist auch möglich, ganze Kantenbündel anzulegen. Anstatt also die Bezeichner einzelner Objekte bei den Attributen **from** und **to** anzugeben, können Bezeichner angegeben werden, die an Mengen gebunden sind. Man muß sicherstellen, daß man die Eigenschaften eines Prädikat/Transitions-Netzes nicht verletzt. Falls ein Kantenbündel angelegt werden soll, so ist dies nur möglich, wenn die Annotation für alle Kanten gleich sein soll. Durch `(add (edge from ?*p to ?t*))` wird beispielswiese eine Kante von jeder Eingangs-Transition von der Stelle, die an den Bezeichner `?p` gebunden wurde, zu jeder Ausgangs-Stelle von der Transition, die an den Bezeichner `?t` gebunden wurde, angelegt. Die Annotation ist für alle Kanten `[x]`.
Als mit der Beschreibung des Aktionsblocks begonnen wurde, wurde erwähnt, daß die Attribute nur mit Bezeichnern besetzt werden können, die bereits im Bedingungsblock gebunden wurden. Um das Anlegen von Kanten zu vereinfachen, wurde ein interner Regelsatz (siehe Kapitel 4.2.2) implementiert, der im Hintergrund arbeitet und das Anlegen von Kanten zwischen Objekten, die unmittelbar vor der Kante im gleichen Aktionsblock erzeugt wurden, ermöglicht.

Mengen

```
(add (set ?S ?P ?T))
```

Wie für die Stellen und Transitionen gilt auch für die Mengen, daß ein neues Objekt erzeugt wird, wenn ein Bezeichner angegeben wird, der noch nicht im Bedingungsblock an ein Objekt vom Typ Menge gebunden wurde. Durch den Bezeichner, der beim Anlegen vergeben wird, ist die Menge dann auch in anderen Regeln verwendbar. Möchte man beispielsweise eine Menge mit dem Namen `?S` anlegen, die alle Objekte aus P und T enthalten soll, so erreicht man dies durch `(add (set ?S ?P ?T))`. Auch die Verwendung der Transformationsfunktionen ist möglich.

3.3.6.3 Objekte verändern

Der Unterschied von **modify** gegenüber **add** ist, daß überprüft wird, ob das Objekt schon existiert. Dies bedeutet, eine Stelle, Transition, Kante oder Menge kann nur

verändert werden, wenn sie im Bedingungsblock an einen Bezeichner gebunden wurde. Außerdem gilt für Stellen, Transitionen, Kanten und Mengen, möchte man ein Attribut nur erweitern, so muß der schon gesetzte Wert im Bedingungsblock an einen Bezeichner gebunden worden sein, und dieser Bezeichner muß beim Setzen mit angegeben werden, sonst ist der "alte" Wert überschrieben. Natürlich kann der Wert auch direkt angegeben werden.
Nun kann man sich fragen, warum existiert nicht die Möglichkeit neue Werte zu den bestehenden hinzugefügt? Um dies zu ermöglichen, müßte man im Bedingungsblock alle Attribute an einen Bezeichner binden. Beispielsweise könnte man für eine Stelle, die an den Bezeichner `?p` gebunden wurde, alle Attribute automatisch an `?p_attribut` binden, wobei `attribut` durch den Namen des entsprechenden Attributs ersetzt würde. Dies hätte zur Folge, daß man keine Werte bei den Attributen mehr angeben könnte, sondern alle Anfragen bezüglich der Attribute über die Vergleichsoperationen realisieren müßte. Einerseits ist oft die Situtation gegeben, daß der Wert direkt angegeben werden kann und andererseits, hierbei handelt es sich um die wesentliche Gegebenheit, kann durch die Möglichkeit der direkten Angabe von Werten die Anzahl der Anfragen verringert werden und damit die Geschwindigkeit bei der Erkennung, ob die Regel feuern kann, erhöht werden.

Stellen

```
(modify (place ?p mark ?p_mark ?p1_mark))
```

Die Syntax ist absolut identisch mit der bei **add** verwendeten. Beispieslweise wird durch `(modify (place ?p mark ?p_mark ?p1_mark))` das Attribut **mark** der Stelle, die im Bediungsblock an den Bezeichner `?p` gebunden wurde, neu gesetzt. Wurde der Bezeichner `?p_mark` für die Marken dieser Stelle und der Bezeichner `?p1_mark` für die Marken einer weiteren Stelle vergeben, so werden diese Marken zu der Stelle hinzugefügt.

Transitionen

```
(modify (transition ?t preaction [rename x y]))
```

Bei den Transitionen gibt es zusätzlich zu den Möglichkeiten, die einem bei **add** zur Verfügung stehen, noch die Erweiterung, daß man Bezeichner, die bei Annotationen verwendet werden, umbenennen kann. Der Bezeichner, der als erstes Argument angegeben wird, wird durch den Bezeichner, der als zweiten Argument angegeben wird, ersetzt. Durch `(modify (transition ?t preaction [rename x y]))` wird beispielsweise dafür gesorgt, daß bei der Transition, die im Bedinungsblock an den Bezeichner `?t` gebunden wurde, bei der *preaction* jedes x durch ein y ersetzt wird (siehe auch Kapitel 3.4).

Kanten

```
(modify (edge ?e annotation [y,z]))
```

Wie bei Transitionen kann auch bei Kanten eine Umbenennung der Bezeichner der Annotation durchgeführt werden. Die Annotation kann auch neu gesetzt werden. Beispielsweise wird durch `(modify (edge ?e annotation [y,z]))` die Annotation der Kante, die im Bedingungsblock an den Bezeichner `?e` gebunden wurde, neu gesetzt.

Mengen

```
(modify (set ?St ?St ?t))
(modify (set St ?t))
(modify (?St (union ?St ?t)))
```

Existiert eine Menge bereits, und man möchte zu den in der Menge enthaltenen Objekten neue hinzufügen, so muß man die schon in der Menge enthaltenen Objekte auch mit angeben. Hat man beispielsweise an eine Menge den Bezeichner `?St` gebunden und man möchte eine Transition, die an den Bezeichner `?t` gebunden wurde, hinzufügen, so wird dies durch `(modify (set ?St ?St ?t))` realisiert. Im Gegensatz zu den Kanten muß die Bindung von `?t` an die Transition bereits im Bedingungsblock durchgeführt worden sein. Gibt man beispielsweise nur `(modify (set St ?t))` an, so enthält `?St` hinterher genau ein Element, nämlich `?t`. Es kommen auch die Transformationsfunktionen, die bereits im Kapitel 3.3.5.5 erläutert wurden, zum Einsatz. Möchte man beispielsweise sicherstellen, daß die Transition `?t` nur einmal in `?St` enthalten ist, so erreicht man dies durch `(modify (?St (union ?St ?t)))`.

3.3.6.4 Objekte löschen

Den letzten Block bilden die Konstrukte, mit denen Stellen, Transitionen, Kanten und Mengen gelöscht werden können. Es können nur Objekte gelöscht werden, die im Bedingungsblock an einen Bezeichner gebunden wurden.

```
(remove ?t ?*t ?t*)
(remove (edge from ?p to ?t))
(remove ?e)
(remove (difference ?P ?Sp))
```

Wurde beispielsweise im Bedingungblock eine Transition an den Bezeichner `?t` gebunden, so wird durch `(remove ?t ?*t ?t*)` die Transition sowie die Menge der Eingangs- und Ausgangs-Stellen dieser Transition gelöscht. Wird ein Objekt gelöscht, so werden automatisch alle Kanten, die dadurch nicht mehr verbunden sind, gelöscht. Zusätzlich ist die Möglichkeit gegeben, Kanten direkt zu löschen. Wurde beispielsweise im Bedingungsblock eine Stelle an den Bezeichner `?p` und eine Transition an den Bezeichner `?t` gebunden, so kann durch `(remove (edge form ?p to ?t))` die Kante zwischen diesen Objekten gelöscht werden. Da diese Kante im Bedingungsblock nicht an einen Bezeichner gebunden worden sein muß, kommt auch hier der Regelsatz, der im Hintergrund arbeitet, zum Einsatz (siehe auch Kapitel 4.2.2). Wurde die Kante beispieslweise vorher schon an den Bezeichner `?e` gebunden und damit schon im Bedingungsblock identifiziert, so kann sie durch `(remove ?e)` gelöscht werden. Beim Löschen von Kanten kann auch eine Annotation angegeben werden. Wird dies nicht getan, so kann die Annotation der Kante beliebig sein. Die Transformationsfunktionen der Mengen können auch zum Löschen von Objekten eingesetzt werden. Wurde z. B. eine Teil aller Stellen in eine Menge `?Sp` aufgenommen, so werden durch `(remove (difference ?P ?Sp))` alle Stellen, die nicht in `?Sp` enthalten sind, gelöscht.

3.4 Termersetzung und Berechnungen

In diesem Abschnitt wird an einem Beispiel erläutert, wie die Termersetzung und die Berechnungen, die die *condition*, *preaction* und *postaction* der Transitionen und die Annotationen an den Kanten betreffen, bei den Transformatinonen eingesetzt werden können.

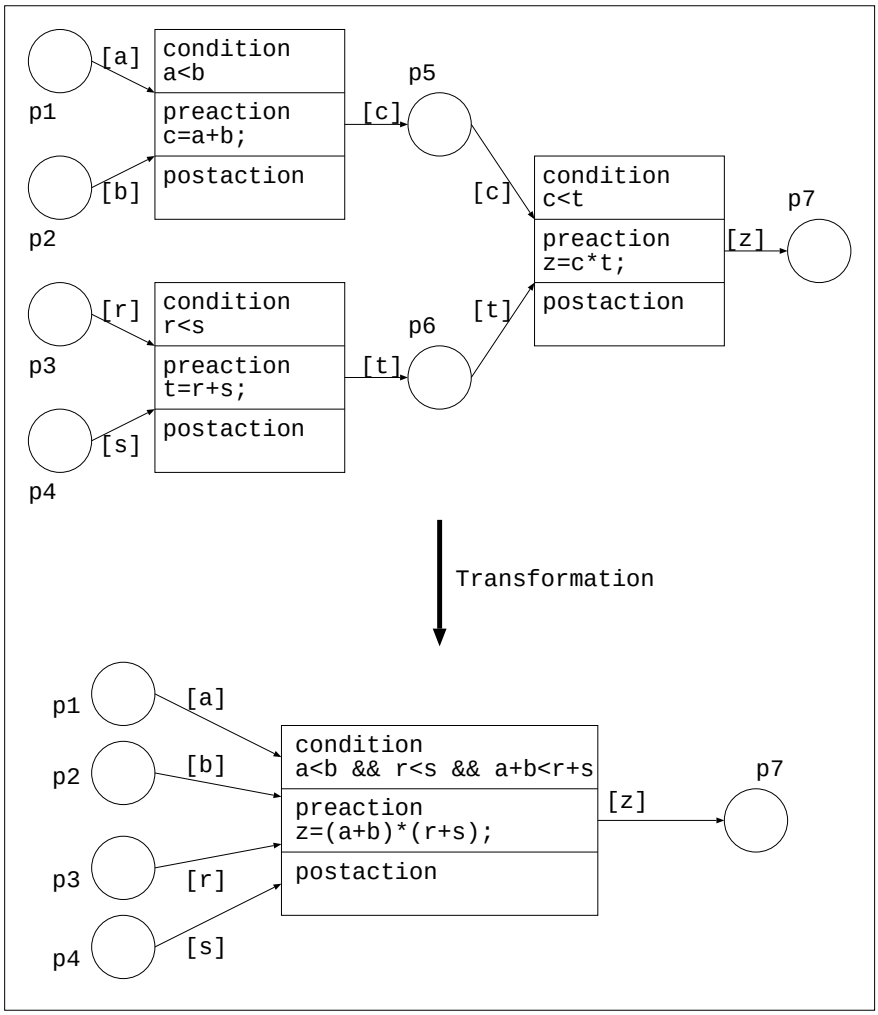

Abbildung 3.2: Ersetzung von Transitionen

Die eigentlichen Ersetzungen und Berechnungen werden von MuPAD im Hintergrund während des Optimierungsprozesses durchgeführt (siehe auch Kapitel 4).
Für alle Transformationen, die man formuliert, kann man voraussetzen, daß jedes Detail der Module, mit denen das System modelliert wird, bekannt ist, und man daher alle Details der Module exakt beschreiben kann.
Möchte man einen Teil eines Prädikat/Transitions-Neztes, zu dem beispielsweise mehrere Transitionen gehören, durch eine Transition ersetzen, so kann man eine geeignete Transformation formulieren. Zuerst beschreibt man im Bedingungblock das Teilnetz, um dann im Aktionsblock die benötigte Ersetzung durchzuführen. Die Vorgehensweise wird an dem Beispiel, das die Abbildung 3.2 auf der Seite 63 zeigt und der zugehörigen

Beschreibung in der Notation der Transformationssprache, erläutert.

```
(description example

 (permission t1:r1)

 (transformation t1
  (rule r1
   (condition
    (permission)
    (place ?p1)
    (place ?p2)
    (transition ?t1
                condition ?t1_cond
                preaction ?t1_pre)
    (edge ?e1 from ?p1 to ?t1 annotation [a])
    (edge ?e2 from ?p2 to ?t1 annotation [b])
    (place ?p3)
    (place ?p4)
    (transition ?t2
                condition ?t2_cond
                preaction ?t2_pre)
    (edge ?e3 from ?p3 to ?t2 annotation [r])
    (edge ?e4 from ?p4 to ?t2 annotation [s])
    (place ?p5)
    (place ?p6)
    (edge ?e5 from ?t1 to ?p5 annotation [c])
    (edge ?e6 from ?t2 to ?p6 annotation [t])
    (transition ?t3
                condition ?t3_cond
                preaction ?t3_pre)
    (edge ?e7 from ?p5 to ?t3 annotation [c])
    (edge ?e8 from ?p6 to ?t3 annotation [t])
    (place ?p7)
    (edge ?e9 from ?t3 to ?p7 annotation [z])
   ) // end condition
```

Im Bedingungsblock wird das Teilnetz exakt beschrieben. Die Formeln, die für die Ersetzung benötigt werden, werden an Bezeichner gebunden. Um sicherzustellen, daß man das gewünschte Teilnetz identifiziert hat, könnte man noch Vergleichsoperationen hinzufügen, um eine Überprüfung der Annotationen durchzuführen.
Um eine korrekte Ersetzung zu erhalten, muß sichergestellt werden, daß die Vergabe der Bezeichner dies zuläßt. Dies ist auf jeden Fall dann sichergestellt, wenn jeder Bezeichner nur einmal auftritt. Ist dies nicht gegeben, so kann das **rename**-Konstrukt zur Umbenennung von Bezeichnern eingesetzt werden. Im Kapitel 5.2 wird ein Beispiel erläutert, bei dem zuerst eine Umbenennung der Bezeichner durchgeführt werden muß. Nachdem die Voraussetzungen für eine korrekte Berechnung geschaffen wurden, kann diese mit Hilfe des **calculate**-Konstrukts durchgeführt werden.

Das Beispiel, das die Abbildung 3.2 auf der Seite 63 zeigt, wurde so gewählt, daß keine Umbenennung nötig ist.
Um die Bedingung einer Transition neu zu bestimmen, gibt man zuerst entweder direkt die Annotation oder einen Bezeichner, der an eine Bedingung einer Transition gebunden wurde, an. Anschließend können mehrere Bezeichner angegeben werden, die im Bedinungsblock an die *preaction* oder *postaction* einer Transition gebunden wurde. Bei der eigentlichen Ersetzung wird jeder Bezeichner in der Bedingung ersetzt, wenn eine Formel gegeben ist, deren linke Seite mit diesem Bezeichner identisch ist. Der Bezeichner wird dann durch die rechte Seite der Formel ersetzt. Die Reihenfolge der Formeln ist hierbei wesentlich, da ein Term in einer Formel nur ersetzt werden kann, wenn er bereits in einer vorherigen Formel definiert wurde.
Soll die *preaction* oder *postaction* einer Transition berechnet werden, so gibt man zuerst den Bezeichner, der auf der linken Seite der resultierenden Formel stehen soll, anschließend die Formel, die zur Berechnung verwendet werden soll, und abschließend mehrere Bezeichner, die im Bedingungsblock an die *preaction* oder *postaction* einer Transition gebunden wurden, an. Auch hier ist die Reihenfolge der Formeln für daß Ergebnis wesentlich. Der Ersetzungsvorgang wird wie bei der *condition* durchgeführt.

```
   (action
    (remove ?t1 ?t2 ?p5 ?p6)
    (modify (transition ?t3
                condition ?t1_cond && ?t2_cond &&
                          [calculate ?t3_cond ?t1_pre ?t2_pre]
                preaction [calculate z ?t3_pre ?t1_pre ?t2_pre]))
    (add (edge from ?p1 to ?t3 annotation [a]))
    (add (edge from ?p2 to ?t3 annotation [b]))
    (add (edge from ?p3 to ?t3 annotation [r]))
    (add (edge from ?p4 to ?t4 annotation [s]))
   ) // end action
  ) // end r1
 ) // end t1
) // end description
```

Im hier gezeigten Aktionsblock wird nun eine neue *condition* und *preaction* berechnet. Damit für die Bedingung der neuen Transition gilt, daß sie äquivalent mit den Bedingungen der drei Transitionen ist, muß sie aus diesen Bedingungen zusammengesetzt werden. Die einzelnen Bedingungen werden *and*-verknüpft. Die Bedingungen der Transitionen `t1` und `t2` können unverändert übernommen werden. Bei der Bedingung der Transition `t3` müssen vor der Übernahme erst einige Terme ersetzt werden. Es stehen die Formeln `c=a+b` und `t=r+s` zur Verfügung, um in die Bedingung `c<t` eingesetzt zu werden. Das Resultat ist `a+b<r+s`. Die Vorgehensweise bei der Berechnung der *preaction* ist ähnlich. Zuerst gibt man den Bezeichner an, der auf der linken Seite der resultierenden Formel stehen soll. Hier handelt es sich um den Bezeichner `z`. Anschließend folgt die Formel, die zur Berechnung verwendet werden soll. Zum Schluß gibt man die Formeln, die zur Ersetzung verwendet werden sollen, an. Bei dem Beispiel werden `c=a+b` und `t=r+s` in die Formel `z=c*t` eingesetzt. Abgesehen von einer einfachen Ersetzung kann auch eine wirkliche Berechnung durchgeführt werden. Im Kapitel 5.2 wird hierzu ein Beispiel angegeben.

Die Annotationen können nicht direkt bei den Transformationen beschrieben werden, da durch mehrfache Zusammenfassungen, wie sie beispielsweise in dem gerade erwähnten Beispiel durchgeführt werden, vorab keine Aussage mehr über die resultierende Formel gemacht werden kann.

3.5 Kontrollmechanismus zur Ablaufsteuerung

Möchte man ein System optimieren, so ist der erste Schritt die Umsetzung des Systems in ein Prädikat/Transitions-Netz und die Modellierung mit dem Prädikat/Transitions-Netz-Editor. Daraus folgt, daß man sich die eigentlichen Transformationen, die letztendlich zu der Optimierung führen sollen, zuerst in der Notation der Prädikat/Transitions-Netze überlegen wird, bevor man mit der Umsetzung in eine Beschreibung der Transformationssprache beginnen kann. Die Beispiele im Kapitel 5 verdeutlichen diese Vorgehensweise. Nun kann man davon ausgehen, daß die Transformationen, die man sich überlegt hat, atomar ausgeführt werden müssen, damit man ein korrektes Ergebnis erhält. Es ist zu beachten, daß eine Beschreibung, die mit der Transformationsprache erstellt wurde, nach der Übersetzung in die CLIPS-Notation, zu einer Menge von Regeln führt, die ohne den Einsatz eines Kontrollmechnismus, alle gleichberechtigt sind. Dies hätte zur Folge, daß nur Transformationen formulierbar sind, die eine gewünschte Optimierung durch das einmalige Feuern einer Regel herbeiführen können. Da dies aber eine starke Einschränkung möglicher Transformationen wäre, wird ein entsprechender Mechanismus benötigt, der es erlaubt, Regeln zu Transformationen zu gruppieren. Während im Normalfall bei einem Expertensystem alle Regeln gleichberechtigt sind, ist mit Hilfe des Kontrollmechanismus eine Menge von gleichberechtigten Transformationen realisierbar. Die Eigenschaften des Expertensystems kommen dadurch weiter zum Tragen. Zusätzlich kann es sein, daß einige Transformationen herausgehoben werden sollen, was bedeutet, daß diese Transformationen zuerst greifen sollen, falls die Wissensbasis dies erlaubt. Würde man sich auf die Möglichkeit beschränken, Prioritäten zu vergeben, so müßte man einen sequentiellen Ablauf vorgeben, um das gewünschte Ergebnis zu erhalten. Dies widerspricht aber der Grundidee eines Expertensystem, das sich gerade dadurch auszeichnet, daß kein sequentieller Ablauf vorgegeben wird, sondern der Inferenzmeachanismus mit Hilfe der gewählten Konfliktlösungsstrategie die Regeln in die Agenda einordnet. Folglich werden die Prioritäten auch nur unterstützend eingesetzt, um beispielsweise eine Regel einer Transformation hervorzuheben.
Die Transformationen können in drei Gruppen unterteilt werden, wobei die Unterteilung zustande kommt, wenn man eine Tranformation aus der Sicht der Transformationssprache betrachtet.

1. Der einfachste Fall ist, daß sich eine Transformation mit einer einzigen Regel formulieren läßt und diese Regel auch nur genau einmal feuern muß, um die gewünschte Optimierung zu erhalten. Bei diesen Transformationen werden keinerlei Kontrollmechanismen benötigt. Bestehen alle Transformationen, die zur Optimierung eingesetzt werden sollen, nur aus solchen Regeln, so wird der Kontrollmechanismus nicht benötigt.

2. Auch die Transformationen der zweiten Gruppe lassen sich mit einer Regel formulieren. Es muß aber gelten, daß diese Regel mehrfach unmittelbar hintereinander

feuert, um das gewünschte Ergebnis zu erhalten. Es muß also die Möglichkeit geben, daß diese Regel, nachdem sie einmal gefeuert hat, solange weiter feuern kann, wie dies aufgrund der gegebenen Fakten möglich ist.

3. Zur dritten Gruppe gehören die Transformationen, zu deren Formulierung mehrere Regeln benötigt werden. Beispieslweise benötigt man eine Regel, die den Einstieg in die Transformation darstellt und Inititalisierungsaufgaben übernimmt. Eine solche Regel wird im folgenden als Einstiegsregel bezeichnet. Diese Regel kann zu Beginn gleichberechtigt mit allen Einstiegsregeln aller anderen Transformationen sein. Feuert diese Einstiegsregel, so muß dafür gesorgt werden, daß alle Regeln, die nicht zu dieser Transformation gehören, ausgeblendet werden, und nur noch die zu der Transformation gehörigen Regeln, die Möglichkeit, bekommen zu feuern. Es kann zusätzlich sein, daß die Transformation eine Regel benötigt, die „Aufräumarbeiten“ übernimmt und nur ganz zum Schluß feuern darf. Eine solche Regel wird im folgenden als Abschlußregel bezeichnet. Kann keine Regel dieser Transformation mehr feuern, so muß der Zustand wieder hergestellt werden, der vor dem Ausblenden der anderen Transformationen aktuell war.

   Eine Transformation dieser Gruppe besteht also aus einer Einstiegsregel, einer Menge von gleichberechtigten Regeln für die eigentliche Aktion und einer Abschlußregel. Wird keine Einstiegs- oder Abschlußregel benötigt, so ändert dies nichts an dem benötigten Kontrollmechanismus.

Es wird im folgenden eine Beispiel angegeben, daß zur Erklärung des Kontrollmechanismus verwendet wird. Die Aufgaben, die die Transformationen ausführen sollen, sind für die Erklärung unwesentlich und werden daher weggelassen. Der wesentliche Aspekt ist der Einsatz der **permission**-Konstrukte.
Beispielsweise bestehe eine Beschreibung aus drei Transformationen `t1, t2` und `t3`.

```
(description example

 (permission t1:g1 t2:g2 t3:g3_a)

 (transformation t1
  (rule g1
   (condition
    (permission) ... )
   (action ... ))
 ) // end t1
```

Transformation `t1` soll aus einer einzigen Regel `g1` bestehen und zur Gruppe 1 gehören. Die gewünschte Optimierung wird also durch das einmalige Feuern der Regel erreicht.

```
 (transformation t2
  (rule g2
   (condition
    (permission) ... )
   (action
    (permission save t2:g2) ... ))
 ) // end t2
```

Die Transformation t2 soll auch nur aus einer einzigen Regel g2 bestehen und zur Gruppe 2 gehören. D. h. die Regel muß also mehrfach unmittelbar hintereinander feuern, um das gewünschte Ergebnis zu realisieren.

```
 (transformation t3
  (rule g3_a
   (condition
    (permission) ... )
   (action
    (permission save t3:g3_b t3:g3_c t3:g3_d) ... )) // end g3_a

  (rule g3_b
   (conditon
    (permission) ... )
   (action ... )) // end g3_b

  (rule g3_c
   (condition
    (permission) ... )
   (action ... )) // end g3_c

  (rule g3_d
   (priority -10)
   (condition
    (permission ... )
   (action ... )) // end g3_d
 ) // end t3
) // end description
```

Die Transformation t3 besteht aus den Regeln g3_a, welche die Einstiegsregel darstellen soll, zwei gleichberechtigten Regeln g3_b und g3_c und einer Abschlußregel g3_d. Für t3 gilt damit, daß die Regel g3_a genau einmal zu Beginn feuern soll, anschließend sollen g3_b und g3_c solange feuern, wie dies die Wissensbasis zuläßt, und den Abschluß soll die Regel g3_d bilden. Im Kapitel 5.1 wird ein Transformation, die strukturell identisch mit t3 ist, detalliert erläutert.
Alle Transformationen sollen zu Beginn gleichberechtigt sein. Dies bedeutet, daß die Regeln g1, g2 und g3_a feuern dürfen, falls die Wissensbasis dies erlaubt und eine der Regeln ganz oben auf der Agenda eingeordent wurde.
Die Realisierung und Arbeitsweise des Kontrollmechanismus sieht wie folgt aus:
Um den Gruppierungeffekt zu erzielen, muß Wissen zur Wissensbasis hinzugefügt werden. Dieses zusätzliche Wissen wird durch den Regelsatz (siehe auch 4.2.2), der im Hintergrund arbeitet und unter anderem dafür zuständig ist, daß das Prädikat/Transitions-Netz in einem konsistenten Zustand gehalten wird, verwaltet. Die Verwaltung der Zugriffsrechte ist *stack*-orientiert. Dies bedeutet, daß immer die Zugriffsrechte, die das oberste Element des Stack bilden, aktuell sind. Weiterhin können von dem Regelsatz, der im Hintergrund arbeitet, Zugriffsrechte vom Stack entfernt werden. Das oberste Element wird vom Stack entfernt, wenn keine der Regeln, für die der Zugriff erlaubt

ist, mehr feuern kann. Voraussetzung ist, daß weitere Zugriffsrechte auf dem Stack liegen. Ist dies nicht der Fall, so kann keine Regel mehr feuern und CLIPS hält. Es ist aber auch möglich, neue Zugriffsrechte auf dem Stack abzulegen und dadurch die aktuellen Zugriffsrechte zu verdecken. Dies wird benötigt, um den Zugriff exklusiv für die Regeln einer Transformation zu setzen. Kann keine dieser Regeln mehr feuern, so ist die Transformation abgearbeitet und die Regeln, die vorher verdeckt wurden, können wieder greifen, falls die Wissensbasis dies erlaubt.
Durch die Angabe von mehreren *permission*-Zeilen kann bereits vorab ein Stack angelegt werden, auf dem mehrere Elemente liegen. Die Anordnung der Elemente auf dem Stack ist mit den angegebenen Zeilen identisch. Dies bedeutet, daß die erste Zeile das oberste Element darstellt. Durch die Angabe mehrerer Zeilen kann man also die Transformationen in einer bevorzugten Reihenfolge anordnen und dadurch erreichen, daß Transformationen bevorzugt vor anderen Transformationen greifen, falls dies die Wissensbasis ermöglicht.
Der Stack wird einmal zu Beginn des Optimierungsprozesses initialisiert. Dies geschieht in dem Beispiel durch die Zeile `(permission t1:g1 t2:g2 t3:g3_a)`, die zu Beginn der Beschreibung angegeben wurde. Alle Regeln die nach **permission** folgen, können als Einstiegsregeln betrachtet werden, wobei eine Einstiegsregel auch andere Aufgaben als reine Initialisierungsaufgaben übernehmen kann.

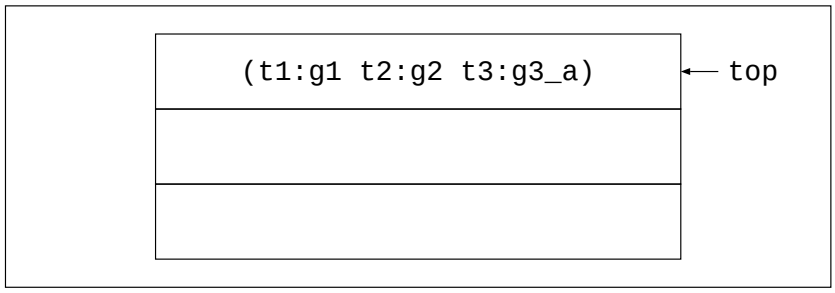

Abbildung 3.3: Zugriffsrechte zu Beginn

Die Abbildung 3.3 zeigt den Stack mit den Zugriffsrechten, in seinem initialen Zustand. Die drei Regeln sind alle gleichberechtigt. Welche Regel zuerst feuern kann hängt zum einen von der gegebenen Wissensbasis und zum anderen von der Wahl, die der Inferenzmechanismus trifft, ab.
Durch die Angabe von `(permission)` im Bedingungsblock einer Regel, wird überprüft, ob der Zugriff erlaubt ist. Da bei allen sechs Regeln diese Zeile eingefügt wurde, können die betroffenen Regeln nur feuern, wenn im globalen Block die Voraussetzungen geschaffen wurden. Bei dem gegebenen Beispiel können also die Regeln `g3_b, g3_c` und `g3_d` nicht feuern, was auch beabsichtigt ist.
Erscheint die Regel `g1` der Transformation `t1` ganz oben auf der Agenda, so kann die Regel feuern, und die entsprechende Aktion wird ausgeführt. Da diese Regel den Stack nicht verändert, können anschließend wieder dieselben Regeln feuern, wie vor der Aktivierung von `g1`.
Falls die Regel `g2` der Transformation `t2` ausgewählt wird, so muß im Aktionsblock dieser Regel dafür gesorgt werden, daß nur diese Regel als nächste feuern darf, vorausgesetz die Wissensbasis erlaubt dies. Damit dies möglich ist, muß in den Aktionsblock

die Zeile `(permission save t2:g2)` aufgenommen wirde. Diese Zeile sorgt dafür, daß die alten Zugriffsrechte gesichert werden und nur die Regel `g2` der Transformation `t2` feuern darf.

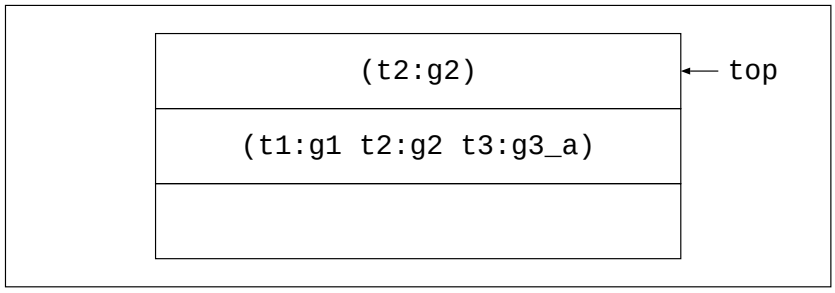

Abbildung 3.4: Zugriffsrechte nachdem die Regel `g2` gefeuert hat

Die Abbildung 3.4 zeigt den Stack mit den Zugriffsrechten nachdem die Regel `g2` gefeuert hat. Ist aufgrund der gegebenen Wissensbasis kein weiteres Feuern dieser Regel mehr möglich, so wird von dem Regelsatz der im Hintergrund arbeitet, das oberste Element vom Stack entfernt. Dadurch werden die alten Zugriffsrechte wieder gesetzt, und der ursprüngliche Zustand der Zugriffsrechte ist wieder hergestellt. Der Stack befindet sich dann wieder in dem Zustand, den die Abbildung 3.3 zeigt.
Der Ablauf bei der Regel `g3_a` ist annähernd identisch. Im Aktionsblock der Regel wird Durch die Zeile `(permission save t3:g3_b t3:g3_c t3:g3_d)` der Zugriff für diese drei Regeln erlaubt, und die alten Zugriffsrechte werden vorher gesichert.

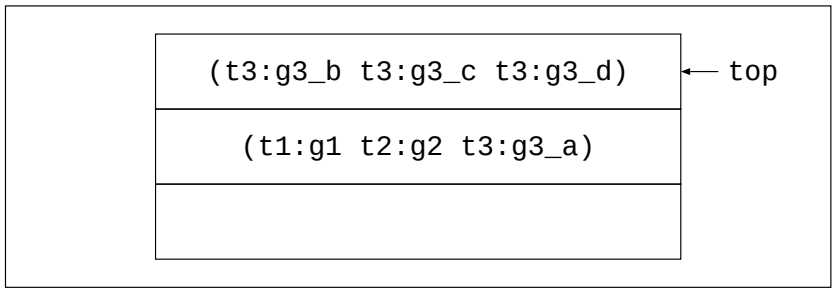

Abbildung 3.5: Zugriffsrechte nachdem die Regel `g3_a` gefeuert hat

Die Abbildung 3.5 zeigt den Stack mit den Zugriffsrechten, nachdem die Regel `g3_a` gefeuert hat. Ein weiterer geforderter Punkt war, daß die Regel `g3_d` nur zum Schluß feuern darf. Dies erreicht man dadurch, daß man dieser Regel eine verringerte Priorität zuordnet, beispielsweise `(priority -10)`. Kann keine der drei Regeln der Transformation `t3` mehr feuern, so erhält man wieder den Stack in seiner ursprünglichen Form (siehe Abbildung 3.3). Entweder kann eine der angegebenen Regeln noch feuern oder CLIPS hält.

Kapitel 4

Realisierung

Im folgenden wird beschrieben, welche Module zum Prädikat/Transitions-Netz-Editor hinzugefügt worden sind. Der erste Abschnitt befaßt sich mit der Beschreibung der Architektur. Im Zweiten Abschnitt wird erläutert, welche Erweiterungen bezüglich CLIPS benötigt wurden.

4.1 Die Architektur

In diesem Abschnitt werden sowohl die neu hinzugekommenen Module als auch der Datenaustausch zwischen diesen Modulen und dem Prädikat/Transitions-Netz-Editor beschrieben.

4.1.1 Status des übernommenen Editors und Erweiterungen

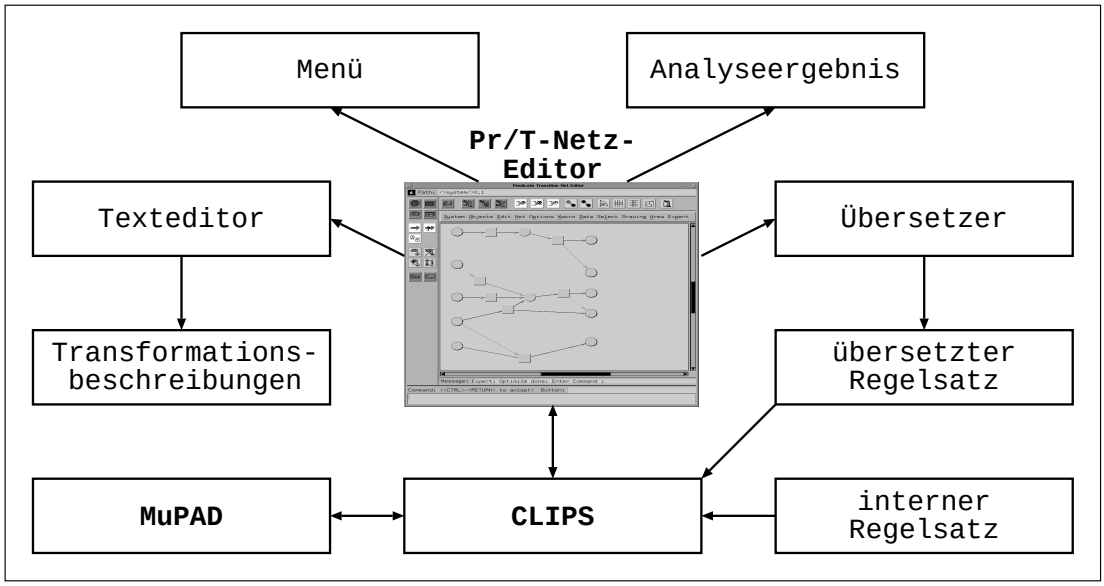

Abbildung 4.1: Architektur

Die Abbildung 4.1 auf Seite 71 zeigt alle wesentliche Komponenten, um die der Prädikat/Transitions-Netz-Editor erweitert wurde. Die Verbindungen geben zum einen an, welche Komponente von wo aufgerufen wird, andereseits ist zu erkennen, zwischen welchen Modulen Informationen ausgetauscht werden.
Zur Implementierung der Module wurden die Programmiersprache C++ ([stro 92], [lipp 95], [meye 92]) sowie die X-Window- und OSF/Motif-Bibliotheken ([brjoleloe 94]) verwendet. Weiterhin wurde der Scannergenerator flex sowie der Parsergenerator LLgen eingesetzt.
Der übernommene Editor bietet alle Voraussetzungen, die benötigt werden, um ein System als Prädikat/Transitions-Netz zu modellieren. Die Simulation des Prädikat/Transitions-Netzes ist auch möglich. Die Erweiterungen betreffen daher die Module, die zur wissensbasierten Analyse und Optimierung benötigt werden.
Das Menü des Prädikat/Transitions-Netz-Editors wurde um den Menüpunkt *Expert* erweitert. Über das *Expert*-Menü können die Aktionen, die zum Analysieren und Optimieren benötigt werden, aufgerufen werden. Die einzelnen Menüpunkte werden im folgenden erläutert.

- Strategy

  Die Konfliktlösungstrategie von CLIPS (siehe auch 2.3.5) kann gesetzt werden. Sie wird benötigt, damit der Inferenzmechansimus entscheiden kann, welche Regeln an welche Position zur Agenda hinzugefügt werden müssen.

- Initialize

  Bevor Analysiert oder Optimiert werden kann, muß das Expertensystem initialisiert werden. Es wird zuerst die gesamte Wissensbasis gelöscht. Anschließend wird die Datei, die die *templates* enthält, die die Datentypen (siehe auch 3.2 und Anhang A) in der CLIPS-Notation beschreiben, geladen. Weiterhin wird der implizite Regelsatz, der benötigt wird, um das Netz konsistent zu halten, geladen. Würde diese Initialisierung nicht erfolgen, so könnte nicht korrekt analysiert bzw. optimiert werden, da das Wissen des vorherigen Analyse- bzw. Optimierungsprozesses das Ergebnis verfälschen würde. Die Dateien für die Datentypen bzw. den impliziten Regelsatz werden durch die Umgebungsvaribalen `PTNE_EXPERT_TEMPLATES` bzw. `PTNE_EXPERT_ACTION` bestimmt. Auf den impliziten Regelsatz wird im Kapitel 4.2.2 eingegangen.

- Load

  Bei der Wahl dieses Menüpunkts wird eine *File-Selection-Box* zur Verfügung gestellt, um einen übersetzten Transformationssatz zu laden. Dieser Transformationssatz, es handelte sich bei der übersetzten Form um eine Menge von Regeln, wird zum Regelsatz, der bei *Initialize* geladen wurde, hinzugefügt. Die Übersetzung ein Beschreibung wird durch den Menüpunkt *Compile* gestartet. Werden mehrere Transformationssätze benötigt, so kann man diese durch wiederholtes Ausführen von Load hinzuladen. Eine Übersetzte Datei hat die Endung `.clp`. Die Initialisierung muß vor dem Laden ausgeführt werden, damit der Optimierungsprozeß korrekt arbeiten kann.

- Edit

  Um die Transformationen einzugeben, wird ein Editor benötigt. Welcher Editor gestartet werden soll, wird durch das Setzen der Umgebungsvariable `EDITOR` bestimmt. Durch `setenv EDITOR 'emacs -font 10x20'` (c-Shell) wird beispielsweise veranlaßt, daß beim Auswählen von *Editor* der Emacs mit der gewählten Schrift gestartet wird. Nachdem die Transformationen eingegeben wurden, muß die gesamte Beschreibung abgespeichert und anschließend übersetzt werden.

- Compile

  Bei *Compile* wird eine *File-Selection-Box* geöffnet, um eine Transformationsdatei, die übersetzt werden soll, zu selektieren. Kann die Datei fehlerfrei übersetzt werden, so wird an die übersetzte Datei die Endung `.clp` angehängt. Falls ein Fehler aufgetreten ist, so wird in einem Fenster angezeigt in welcher Zeile der Fehler aufgetreten ist. Über die Umgebungsvaribale `LANG_ERROR_LIMIT` kann festgelegt werden, wieviel Fehler maximal angezeigt werden sollen. Diese Variable sollte mit einem Wert aus dem Bereich 1 bis 3 belegt werden, da nach einem Fehler keine weitere korrekte Übersetzung mehr möglich ist. Der Fehler muß zuerst behoben werden, bevor der Übersetzungvorgang wiederholt werden kann.

- Analyse

  Wählt man *Analyse*, so wird der Analyseprozeß gestartet. Vorab muß die Initialisierung durchgeführt werden, und die Transformationen, die das System optimieren sollen, müssen geladen worden sein. Das Ergebnis wird dann in einem Fenster angezeigt. Es wurde das bereits vorhandene *Message-Window* des Prädikat/Transitions-Netz-Editors verwendet, das den Vorteil hat, daß Zeitangaben gemacht werden. Man kann dadurch überprüfen, wie lange ein Optimierungsprozeß dauern würde. Die Analyse ist für denjenigen gedacht, der Transformationen entwickelt, da nur angezeigt wird, welche Regeln welcher Transformationen in welcher Reihgenfoge aufgerufen werden.

- Optimize

  Mit *Optimize* wird der eigentliche Optimierungsprozeß gestartet. Wie bei der Analyse gilt, daß vorab die Initialisierung durchgeführt und die benötigten Transformationen geladen worden sein müssen. Das Ergebnis des Optimierungsprozesses, wird durch das modifizierte System widergespiegelt. Um auf das Original zurückgreifen zu können, kann vorab ein Backup angelegt werden.

4.1.2 Interaktion zwischen dem Editor, CLIPS und MuPAD

In diesem Abschnitt wird der Ablauf des Optimierungsprozesses erläutert. Der Ablauf beim Analyseprozeß ist identisch.
Die Abb. 4.2 auf Seite 74 zeigt alle Details, des Optimierungsprozesses. Die Nummern an den Kanten geben die Reihenfolge, in der die einzelnen Aktionen aufgerufen werden müssen, an. Zuerst muß CLIPS initialisiert werden. Anschließend müssen die übersetzten Transformationen, die zur Analyse oder Optimierung verwendet werden sollen, geladen werden. Als nächstes kann über das *Expert*-Menü gewählt werden, ob analysiert oder optimiert werden soll.

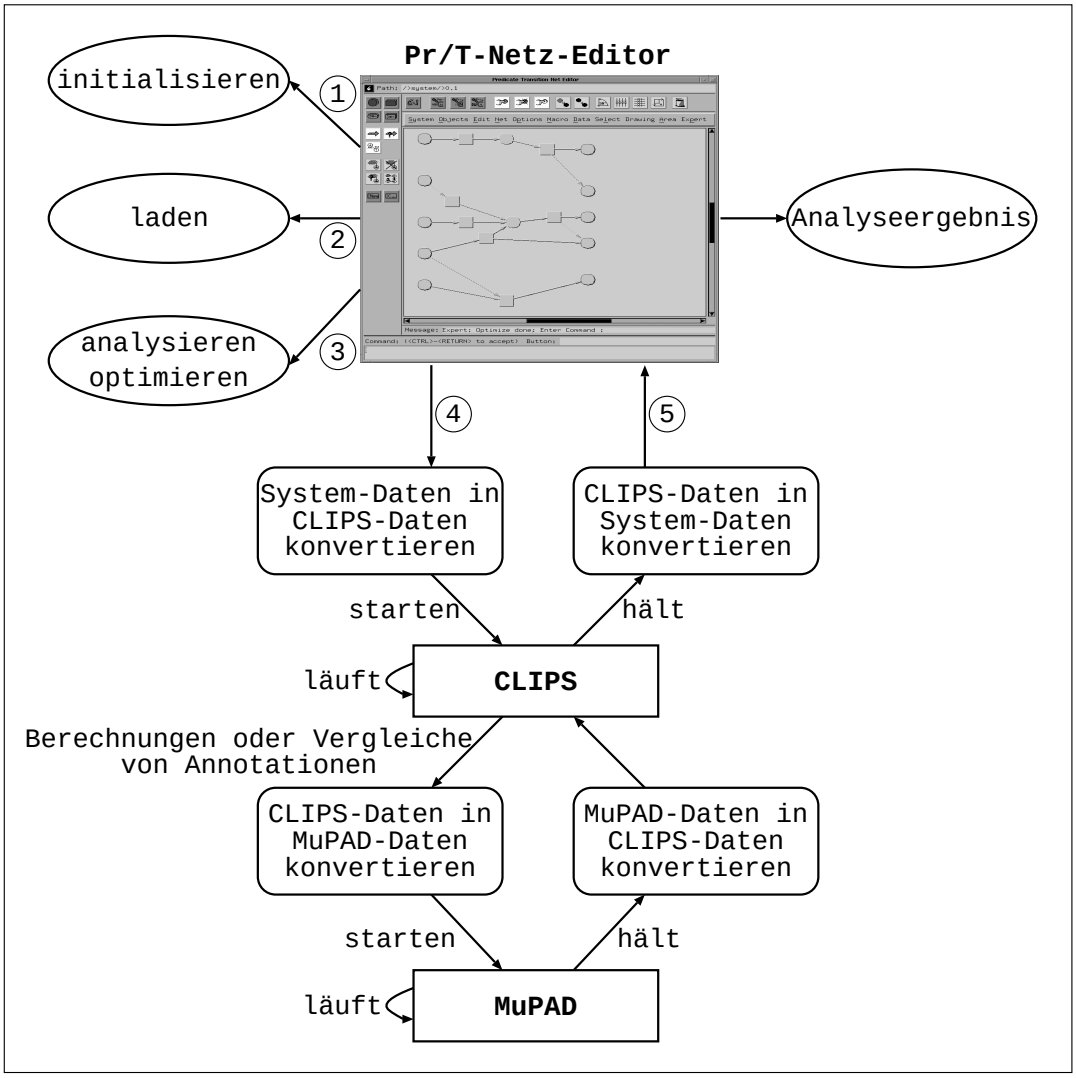

Abbildung 4.2: Optimierungsprozeß

Der Ablauf des eigentliche Optimierungsprozesses beginnt bei 4. und endet bei 5. Zuerst müssen die Daten, die das System darstellen, in Wissen gewandelt werden, das von CLIPS verarbeitet werden kann. Anschließend kann CLIPS gestartet werden. CLIPS arbeitet dann, solange noch feuerbereite Regeln auf der Agenda eingetragen sind. Sollen Berechnungen durchgeführt werden oder Annotationen verglichen werden, so wird MuPAD benötigt. Bei welchen Annotationen MuPAD benötigt wird und warum wurde im Kapitel 3.3.5.4 erläutert. Da die Syntax von MuPAD und CLIPS nicht identisch ist, müssen die Annotationen konvertiert werden. Anschließend kann MuPAD seine Arbeit verrichten, und das Ergebnis kann, nachdem es wieder konvertiert wurde, von CLIPS weiterverarbeitet werden. Sind keine feurerbereiten Regeln mehr vorhanden, so hält CLIPS an. Der Optimierungsprozeß ist abgeschlossen.
Anschließend muß noch das Ergebnis dargestellt werden. Entweder wird es bei der Optimierung durch das modifizierte System oder bei der Analyse durch ein Fenster, in dem die aufgerufenen Regeln angezeigt werden, dargestellt.

Obwohl die Daten mehrfach konvertiert werden müssen, wirkt sich dies nur gering auf die Performance aus, da daß Expertensystem die meiste Zeit benötigt um festzustellen, welche Regeln zur Agenda hinzugefügt oder von der Agenda entfernt werden müssen. Auf die Arbeitsweise des Inferenzmechansismus wurde im Kapitel 2.3.3 eingegangen.

4.2 CLIPS

Es wird im folgenden erläutert welche Erweiterungen des Funktionsumfangs benötigt wurden. Anschließend wird auf den internen Regelsatz, der unter anderem benötigt wird, um das Prädikat/Transitions-Netz in einem konsistenten Zustand zu halten, eingegangen.

4.2.1 Erweiterungen des Funktionsumfangs

Der von CLIPS zur Verfügung gestellte Funktionssatz umfaßte eine große Anzahl von Funktionen, die beispielsweise für Vergleiche von INTEGER-Werten oder zur Manipulation von *multisolts* eingesetzt werden können (siehe auch [crba 93] und [crad 93]).
Im Kapitel 3 wurde bereits erwähnt, das Mengen ein wichtiger Bestandteil der Transformationssprache sind. Um die Funktionen zur Verwaltung dieser Mengen zu realisieren, mußte der Funktionsumfang von CLIPS erweitert werden.
Es sind die Funktionen *empty*, *card*, *in*, *subset*, *union intersection* und *difference* implementiert worden. Die Syntax der Funktionen wurde bereits im Kapitel 3.3.5.5 erläutert. Ein Anwendungsbeispiel ist im Kapitel 5.1 zu finden.
Weiterhin sind Funktionen zum Anlegen und Löschen von Kanten implementiert worden. Diese Funktionen werden benötigt, da es möglich ist, ganze Kantenbündel anzulegen oder zu löschen. Bei dem Start- und Zielpunkt einer Kante kann es sich also auch um Mengen handeln. Intern müssen diese Mengen durchlaufen werden, und für jeden Start- und Zielpunkt wird eine Kante angelegt. Die Annotationen müssen bei Kantenbündeln für alle Kanten identisch sein.
Im Kapitel 3.3.5.4 sind bereits die Vergleichsfunktionen, um die CLIPS erweitert wurde, erläutert worden. Diese speziellen Funktionen sind nur für den Vergleich der Annotatione ausgelegt. Für Normierungsaufgaben, die bei diesen Vergleichen benötigt werden, wird MuPAD eingesetzt.
Weiterhin wurden Funktionen hinzugefügt, die es ermöglichen, daß die *condition*, *preaction* und *postaction* einer Transition neu berechnet werden können. Auch hier wird MuPAD für die Termersetzungen und Berechnungen eingesetzt.

4.2.2 Implizit benötigte Regeln

In den anderen Kapiteln wurde mehrfach darauf hingewiesen, daß zusätzlich zu dem Regelsatz, der benötigt wird, um die Analyse oder Optimierung durchzuführen, ein weiterer Regelsatz benötigt wird, der im Hintergrund arbeitet und hauptsächlich die Aufgabe hat, das Prädikat/Transitions-Netz in einem konsistenten Zustand zu halten. Man kann diesen Regelsatz als fachspezifisches Wissen betrachten, das speziell für die Modifikation von Prädikat/Transitions-Netzen ausgelegt ist. Einerseits müssen diverse Verwaltungsarbeiten im Hintergrund durchgeführt werden, andererseits werden auch

Regeln zur Verfügung gestellt, die die Erstellung einer Beschreibung mit der Transformationssprache erleichtern.
Bei der Transformationssprache kann an einigen Stellen der Eindruck entstehen, daß es möglich ist, Operationen mit Objekten auszuführen, die nicht im Bedingungsblock an einen Bezeichner gebunden wurden. Dieser Eindruck täuscht, da an diesen Stellen die Regeln des impliziten Regelsatzes diese Arbeit übernehmen. Der Regelsatz wird in der Arbeit als impliziter Regelsatz bezeichnet, da er automatisch bei der Auswahl des Menüpunkts *Initialize*, der zur Inititalisierung von CLIPS dient, geladen wird. Er ist für denjenigen, der die Optimierungskomponente einsetzt, nicht sichtbar.

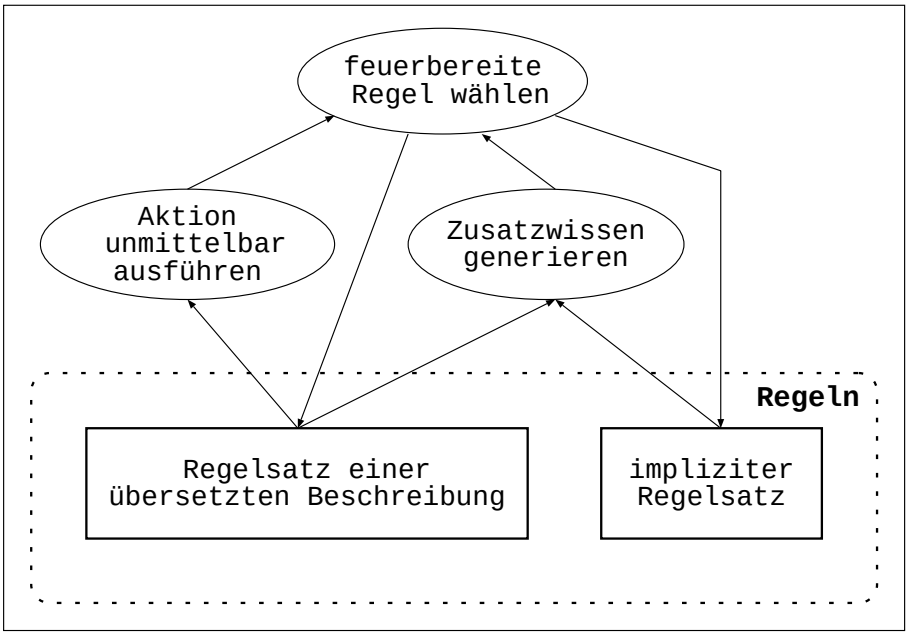

Abbildung 4.3: Einbindung des impliziten Regelsatzes

Die Abbildung 4.3 zeigt, wie die Zusammenarbeit der Regeln des impliziten Regelsatzes mit den Regeln, die durch das Übersetzen einer Beschreibung entstanden sind, aussieht. Wird der Optimierungsprozeß gestartet, so wird zuerst eine feuerbereite Regel ausgewählt. Zu Beginn des Optimierungsprozesses wird im Normalfall ein Regel feuern, die aus dem Regelsatz stammt, der zur Optimierung dient. Kann keine dieser Regeln feuern, so kann nur eine der Regeln aus dem impliziten Regelsatz, die für die Verwaltung des Stacks, der für die Zugriffsrechte zuständig ist, feuern. Falls auch dies nicht möglich ist, so hält CLIPS.
Jetzt hängt es von der Aktion der feuerenden Regel ab, ob eine unmittelbare Modifikation des Neztes durchgeführt werden kann oder ob erst Zusatzwissen erzeugt werden muß. Falls Zusatzwissen erzeugt wird, so greift auf jeden Fall eine Regel des impliziten Regelsatz, da die Prioritäten, die diesen Regeln zugeordnet wurden, höher sind als die, die einer Regel mit der Beschreibungssprache zugeordent werden darf. Die Regeln des impliziten Regelsatzes führen dann die notwendigen Aktionen aus. Wird kein Zusatzwissen benötigt, so beginnt der Prozeß von Neuem, vorausgesetzt es sind noch feuerbereite Regeln vorhanden.
Den Regeln des impliziten Regelsatzes sind unterschiedliche Prioritäten zugeordnet

worden, da die Abarbeitungsreihenfolge dieser Regeln wesentlich ist. Die Regeln lassen sich in mehrere Gruppen einteilen, die verschiedene Aufgabengebiete abdecken. Der komplette Regelsatz ist im Anhang C zu finden. Im folgenden werden die Aufgaben der einzelnen Gruppen erläutert.

- Zugriffsrechte

  Es werden drei Regeln benötigt, die den Stack für die Zugriffsrechte (siehe auch Kapitel 3.5) verwalten. Die Regel `set_permission` hat die niedrigste Priorität aller Regeln. Dies bedeutet, daß diese Regel nur feuern kann, wenn keine weitere feuerbereite Regel vorhanden ist und wenn der Stack nicht leer ist. Die Regel `save_permission` hat ein hohe Priorität und die Regel `new_permission` eine etwas niedrigere Priorität. Die `save_permission`-Regel muß einerseits überprüfen, ob die zu sichernden Zugriffsrechte und die auf dem Stack liegenden Zugriffsrechte verschieden sind. Ist dies nicht der Fall, so kann das Abspeichern entfallen. Andererseits müssen bei identischen Zugriffsrechten die überflüssigen Fakten gelöscht werden. Sollen die Zugriffsrechte nicht gesichert, sondern nur neu gesetzt werden, so feuert die Regel `new_permission`. Zusätzlich müssen alle Regeln noch Aufgaben übernehemen, die für die interne Organisation des Stacks benötigt werden. An den Regeln für die Zugriffsrechte kann man erkennen, daß eine Vergabe von Prioritäten unerläßlich ist, da man sonst falsche Ergebnisse erhalten würde. Beispielsweise könnte es sein, daß der Inferenzmechanismus die Regel `new_permission` vor der Regel `save_permission` auf der Agenda plazieren würde, da die Fakten, die die Bedingung der Regel `new_permission` erfüllen, eine Teilmenge der Fakten sind, die die Bedingung der Regel `save_permission` erfüllen. Die alten Zugriffsrecht würden in diesem Fall nicht gesichert, obwohl dies vielleicht gefordert wird.

- zusätzliche Attribute aktualisieren

  Im Kapitel 3.2 wurde bereits erwähnt, daß die Mengen, die über die Punkt-Operatoren ermittelt werden können, nur abgefragt werden können, da sie intern verwaltet werden. Hierzu werden zwei Regeln benötigt, die mit der Hilfe von zusätzlichen Informationen, die bei einer Veränderung des Prädikat/Transitions-Neztes erzeugt werden, eine Aktualisierung dieser Mengen durchführen. Auch die Attribute *in* und *out*, über die die Anzahl der Eingangs- bzw. Ausgangskanten abgefragt werden können, werden mit aktutalisiert. Zusätzlich wird eine Regel benötigt, die die zusätzlichen Fakten, die bei diesem Prozeß anfallen, wieder entfernt.

- Objekte anlegen und Attribute setzen oder verändern

  Möchte man neue Objekte anlegen oder Attribute bestehender Objekte setzen, so wird dies jeweils durch eine Regel realisiert. Dies bedeutet, daß für Stellen, Transitionen, Kanten und Mengen jeweils eine Regel zum Anlegen von Objekten oder zum Setzen von Attributen benötigt wird. Zusätzlich werden Regeln benötigt, mit denen die Bezeichner der *condition*, *preaction* und *postaction* einer Transition und die Annotation einer Kante umbenannt werden können. Auch bei diesen Operationen fallen zusätzliche Fakten an, die gelöscht werden müssen, bevor der normale Zyklus fortgesetzt werden kann.

- Kanten anlegen

  Während das Anlegen von Stellen und Transitionen unabhängig von anderen Aktionen durchgeführt werden kann, ist es vorteilhaft, wenn eine Kante zwischen Objekten angelegt werden kann, die unmittelbar vorher in dem selben Aktionsblock erzugt wurden. Sonst müßte man jeweils zusätzliche Regeln zu einer Transformation hinzufügen, die die Objekte wieder identifizieren und anschließend die Kante anlegt. Die Objekte müßten entsprechend gekennzeichnet werden, da sonst keine Identifikation möglich wäre. Generell ist es aber nur möglich auf Objekte im Aktionsblock zuzugreifen, die vorab im Bedingungsblock an einen Bezeichner gebunden wurden. Es werden zum Anlegen der Kanten mehrere Regeln benötigt. Dies resultiert daraus, daß die Transformationssprache das Anlegen von einfachen Kanten und von Kantenbündeln erlaubt. Weiterhin müssen CLIPS-spezifische Restriktionen beachtet werden. Es ist nicht erlauben, daß bei Anfragen die *or*-Verknüpft werden, Variablen gebunden werden.

- „virtuelle“ id's vergeben

  Eine Regel wird benötigt, die den neu erzeugten Stellen, Transitionen und Kanten virtuelle id's zuordnet, da diese zur eindeutigen Identifikation der Objekte benötigt werden.

- Löschen von Objekten

  Zum Löschen von Objekten werden mehrere Regeln benötigt, da beispielsweise Mengenfunktionen eingesetzt werden können, um die Objekte, die gelöscht werden sollen, zu bestimmen. Das eigentliche Löschen übernimmt in diesem Fall eine Regel des internen Regelsatzes. Auch wenn ein Objekt durch die Angabe des Bezeichners gelöscht werden soll, so wird dies von einer Regel des internen Regelsatzes ausgeführt. Kanten, von denen der Start- oder Zielpunkt gelöscht wurde, müssen auch aus dem Faktensatz entfernt werden.

Zur Identifikation der Objekte werden die im Kapitel 3.2 erwähnten Attribute *id* und *name* benötigt. Das Attribut *info* dient zur Übermittlung der durchzuführenden Aufgabe. Es wird beispielsweise dort vermerkt, ob ein Objekt angelegt, gelöscht oder verändert werden soll. Diese Information wertet dann der implizite Regelsatz aus und führt die gewünschte Aktion durch.

Kapitel 5

Anwendungsbeispiele

In diesem Kapitel werden zwei Anwendungsbeispiele zur Verdeutlichung des in den vorherigen Kapiteln beschriebenen dienen. Dies betrifft insbesondere die im Kapitel 3 beschriebene Transformationssprache. Es wird gezeigt, wie ein System, das als Prädikat/Transitions-Netz modelliert wurde, durch eine bzw. mehrere Transformationen optimiert wird. Für beide Beispiele gilt, daß schon vorab festgelegt wurde, wie das System als Prädikat/Transitions-Netz modelliert werden kann und wie geeignete Transformationen aussehen. Der Kernpunkt ist die Umsetzung dieser Transformationen in die Notation der Transformationssprache.
Beim ersten Beispiel wird bei der Optimierung auf die Struktur des Prädikat/Transitions-Netzes eingegangen. Zusätzlich werden beim zweiten Beispiel die Annotationen mit einbezogen.
Wird im folgenden der Begriff Petrinetz verwendet, so handelt es sich um ein Prädikat/ Transitions-Netz ohne Annotationen (siehe auch Kapitel 2.1.2).

5.1 Transformation auf Basis der Petrinetzstruktur

Das erste Beispiel wurde aus der Dissertation [klei 94], die das Thema „Synthese von zeitinvarianten Hardware-Modulen“ behandelt, entnommen. Kernpunkt der Arbeit ist die Umsetzung der Verhaltensbeschreibung in eine Struktur auf Gatterebene. Um diese Umsetzung zu realisieren, werden unter anderem Netztransformationen ([klei 94] Kapitel 3.2, Seite 127 ff.) eingeführt, die sich auf die Struktur eines Petrinetzes beziehen. Die Transformationen dienen einerseits zum Verkleinern des Neztes und andererseits, um eine Überdeckung zu finden, die es ermöglicht, das Netz aus einer Menge von vorgegebenen Teilnetzen (Bibliothek) zusammenzusetzen.
Aus der Menge der in der Arbeit angegebenen Transformationen ist nun eine Transformation ausgewählt worden, bei deren Umsetzung in die Notation der Transformationssprache diverse Eigenschaften der Sprache verdeutlicht werden können. Die algorithmische Umsetzung steht hier nicht im Vordergrund. Für die ausgewählte Transformation existieren bereits diverse effiziente Algorithmen. Die Aufgabe der Transformation 3.2.12 ([klei 94], Seite 141) ist das Entfernen aller Senken aus dem Petrinetz. Eine Senke ist ein Teilnetz, in das nur Kanten hineinführen, aus dem aber keine Kanten herausführen. Von einer Quelle spricht man, wenn aus einem Teilnetz nur Kanten herausführen und keine Kanten hineinführen.

Das Petrinetz, das in Abbildung 5.1 dargestellt wird, enthält eine Senke und eine Quelle. Bei dem Teilnetz, das innerhalb des oberen gestrichelten Rahmens gezeigt wird, handelt es sich um die Senke. Innerhalb des unteren gestrichelten Rahmens ist die Quelle zu sehen. Das Entfernen der Quelle ist mit dem einer Senke nahezu identisch.
Die Transformation zum Entfernen der Senke wird im folgenden formal Angegeben, um einen Vergleich der ursprünglichen Transformation und der Transformation, die in der Notation der Transformationssprache formuliert wurde, zu ermöglichen. Die Transformation wurde dabei an die im Kapitel 2.1 eingeführte Notation angepaßt.

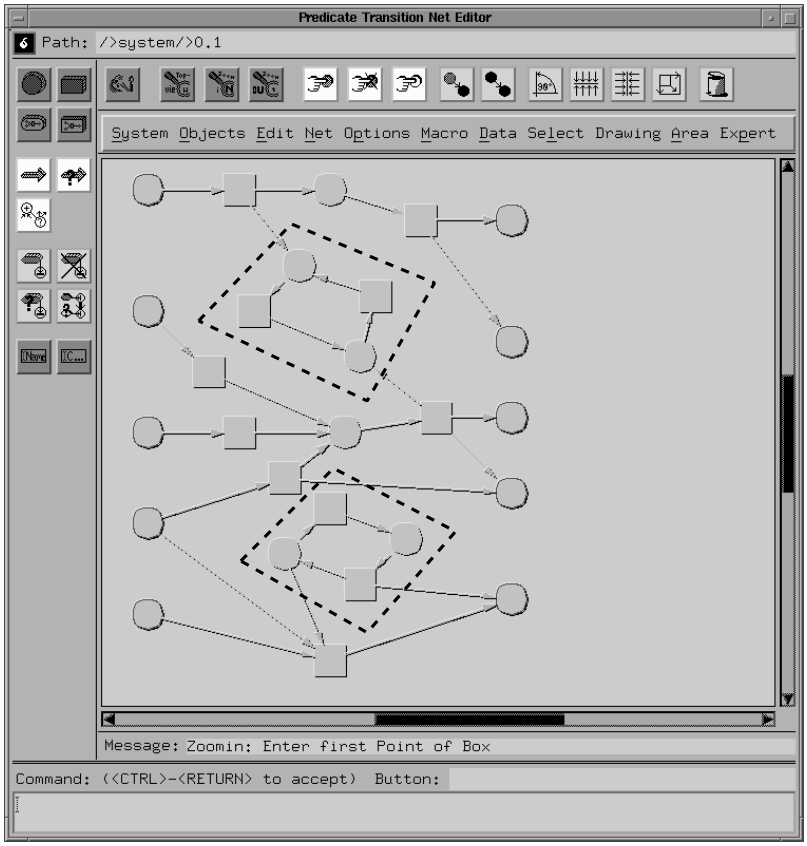

Abbildung 5.1: Petrinetz mit einer Senke und Quelle

Transformation: Sei eine Stellenmenge $P_d \subseteq P \setminus (P_i \cup P_o)$ und eine Transitionsmenge $T_d \subseteq T$ eines Petrinetzes gegeben, existiere keine Eingangskante $(p,t) \in I$ mit $p \in P_d$ und $t \in T \setminus T_d$, und existiere keine Ausgangskante $(t,p) \in O$ mit $p \in P \setminus P_d$ und $t \in T_d$, so wird durch $P' = P \setminus P_d, T' = T \setminus T_d, I' = I \setminus \{(u,v) \mid u \in P_d$ oder $v \in T_d\}, O' = O \setminus \{(v,u) \mid u \in P_d$ oder $v \in T_d\}$ ein Petrinetz ohne Senke konstruiert.

Bei den Stellen auf der linken Seite des Prädikat/Transitions-Netzes, das in der Abbildung 5.1 zu sehen ist, handelt es sich um Eingabe-*Ports* und bei den Stellen auf der rechten Seite um Ausgabe-*Ports*. Man muß sich nun überlegen, wie man eine Senke identifizieren kann, um sie anschließend zu entfernen. Der gewählte Ansatz ist mit einer Tiefensuche vergleichbar. Man benötigt zwei Hilfsmengen `?Sp` und `?St`, die zur Aufnahme der erreichbaren Stellen bzw. Transitionen dienen sollen. Die Namen dieser Mengen

sind wie alle anderen Bezeichner beliebig wählbar. Ausgehend von den Ausgabe-*Ports*, mit denen `?Sp` vorbesetzt wird, werden alle Transitionen, von denen aus eine Kante zu einem Element in `?Sp` verläuft, zur Menge `?St` hinzugefügt. Nun kann man von der Menge `?St` ausgehend alle Stellen, von denen eine Kante zu einem Element in `?St` verläuft, zu `?Sp` hinzufügen. Wiederholt man diesen Prozeß mehrfach, so gelangt man zu den Eingabe-*Ports*. Alle Objekte, die auf diesem Weg nicht zu `?Sp` oder `?St` hinzugefügt wurden, müssen zu einer Senke gehören. Ausgehend von dieser Idee muß man sich nun die Umsetzung in die Notation der Transformatinssprache überlegen. Möchte man alle Quellen entfernen, so muß man die Suche bei den Eingabe-*Ports* beginnen.

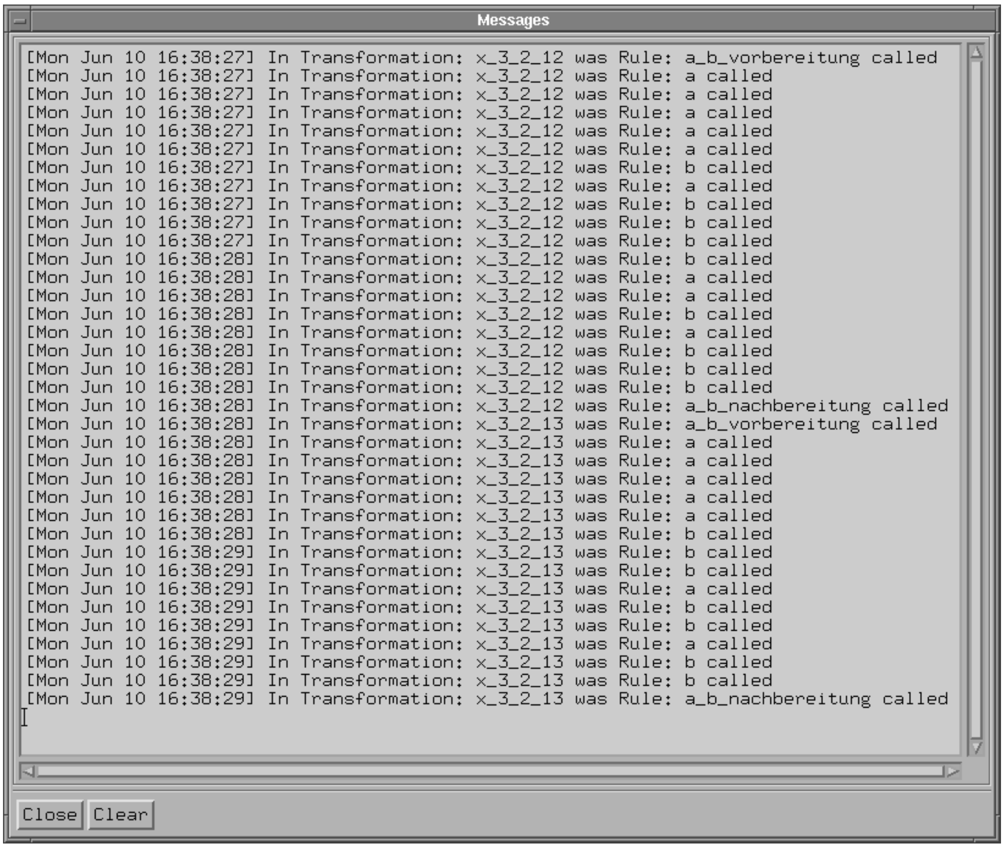

Abbildung 5.2: Analyseergebnis

Es folgt auf den nächste Seiten die Beschreibung der Transformation 3.2.12 in der Notation der Transformationssprache. Wie alle Beschreibungen beginnt auch diese mit einer öffnenden runden Klammer gefolgt vom Schlüsselwort `description` und dem Namen, den die Datei nach dem Übersetzungprozeß erhalten soll. Hier würden die Regeln, die zur Optimierung verwendet werden können, in der Datei `optimize.clp` abgelegt. Anschließend werden die Voraussetzungen geschaffen, die das Überprüfen der Zugriffsrechte ermöglichen. Wie bereits im Kapitel 3.5 erläutert, existieren mehrere Möglichkeiten, um die Abarbeitungsreihenfolge der Transformationen zu beeinflußen. Vorstellbar wäre, daß man unter Verwendung einer bevorzugten Reihenfolge vorab andere Transformationen zuläßt. Hier werden nun exklusiv nur die Transformationen zum Entfernen von Senken und Quellen erlaubt. Voraussetzung ist immer, daß auch eine entsprechen-

de Überprüfung im Bedingungsblock der Regeln der entsprechenden Transformation durchgeführt wird.

```
(description optimize
 (permission x_3_2_12:a_b_vorbereitung   // Senke
             x_3_2_13:a_b_vorbereitung)  // Quelle
 (create ?Sp ?St)
 (transformation x_3_2_12
  (rule a_b_vorbereitung
   (condition
    (permission)
    (set ?Sp ?Po)
    (empty ?Sp))
   (action
    (permission save x_3_2_12:a x_3_2_12:b
                     x_3_2_12:a_b_nachbereitung)
    (modify (set ?Sp ?Po)))
   )
```

Es ist wesentlich, daß kein sequentieller Ablauf vorgegeben wird. Es handelt sich bei dem Beispiel nur um eine Transformation aus vielen. Diese benötigt mehrere Regeln, um ihr Ziel zu erreichen. Die Transformationen sind i. allg. alle gleichberechtigt. Der größte Teil der Transformationen, die in [klei 94] beschrieben werden, lassen sich in der Notation der Transformtionssprache, mit einer Regel realisieren.

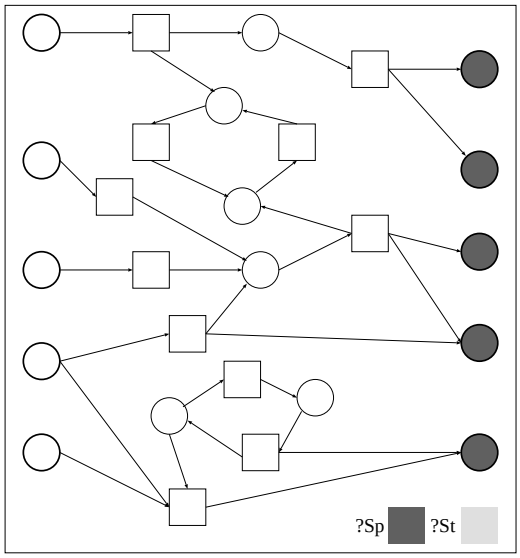

Abbildung 5.3: Die Mengen `?Sp` und `?St` nach dem Feuern der Einstiegsregel

Bevor die Beschreibung der eigentlichen Transformationen beginnt, werden die benötigten leeren Mengen `?Sp` und `?St` erzeugt. Da die Transformation nicht mit einer einzelnen Regel realisierbar ist, muß eine Aufteilung erfolgen. Den Einstieg in die Transformation bildet die Regel `a_b_vorbereitung`, die die Voraussetzungen schafft. Mit `(permission)` wird überprüft, ob der Zugriff erlaubt ist. Anschließend wird durch `(set ?Sp ?Po)` und

`(empty ?Sp)` überprüft, ob die Mengen `?Sp` und `?Po` existieren und die Menge `?Sp` leer ist. Bei `?Po` handelt es sich um die Menge der Ausgabe-*Ports*, die immer existiert. Sind diese Voraussetzungen erfüllt, so wird die Aktion ausgeführt. Zuerst werden die aktuellen Zugriffsrechte gerettet, und anschließend werden die Zugriffsrechte für die restlichen Regeln der Transformation gesetzt (siehe auch Kapitel 3.5) . Damit können auf der Agenda nur diese drei Regeln erscheinen. Mit `(modify (set ?Sp ?Po))` werden zusätzlich alle Stellen, die Ausgabe-*Ports* sind, zu `?Sp` hinzugefügt.
Die Abbildung 5.3 auf der Seite 82 zeigt, welche Elemente nach dem Feurern der Regel `a_b_vorbereitung` zu `?Sp` hinzugefügt wurden. Die Menge `?St` ist noch leer.

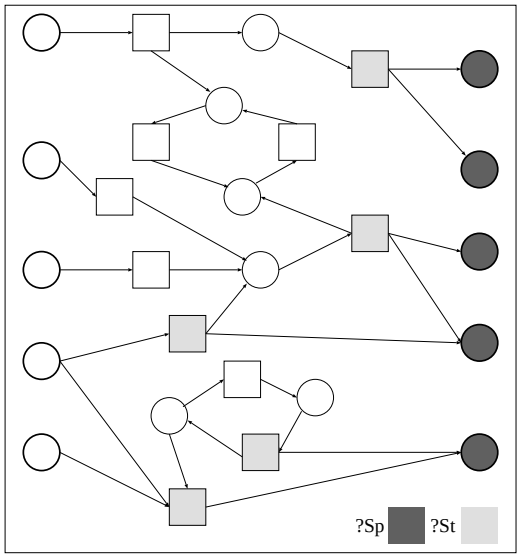

Abbildung 5.4: Die Mengen `?Sp` und `?St` nach mehrfachem Feuern der Regel `a`

Wie bei der Beschreibung der Idee bereits erwähnt, werden nun noch Regeln benötigt, die ausgehend von den Elementen in den Mengen `?Sp` bzw. `?St` überprüfen, von welchen Objekten aus eine Kante zu einem Element in `?Sp` bzw. `?St` existiert.

```
(rule a
 (condition
  (permission)
  (set ?Sp ?St)
  (place ?p)
  (transition ?t)
  (edge ?e from ?t to ?p)
  (in ?Sp ?p)
  (not (in ?St ?t)))
 (action
  (modify (set ?St ?St ?t)))
 )
```

Die Regel `a` überprüft für die Elemente in `?Sp`, ob Transitionen existieren, von denen aus eine Kante zu einem Element in `?Sp` verläuft. Ist dies der Fall, so wird die Transition zur Menge `?St` hinzugefügt, falls sie noch nicht in `?St` enthalten war.

Im Detail sieht dies nun so aus, daß zuerst überprüft wird, ob der Zugriff erlaubt ist. Dies kann nur dann der Fall sein, wenn vorab die Regel `a_b_vorbereitung` der Transformation `x_3_2_12` gefeuert hat. Anschließend wird überprüft, ob die Mengen `?Sp` und `?St` existieren. Weiterhin muß eine Stelle `?p` eine Transition `?t` und eine Kante von `?t` nach `?p` existieren. Es muß zusätzlich noch gelten, daß `?p` in der Menge `?Sp` und `?t` nicht in der Menge `?St` enthalten ist. Sind alle Bedingungen erfüllt, so wird die Transition `?t` zur Menge `?St` hinzugefügt.
Die Abbildung 5.4 auf der Seite 83 zeigt, welche Elemente `?Sp` und `?St` enthalten, wenn ausgehend von der Situation, die die Abbildung 5.3 auf der Seite 82 zeigt, die Regel `a` so oft feuert, wie möglich.

```
(rule b
 (condition
  (permission)
  (set ?Sp ?St)
  (place ?p)
  (transition ?t)
  (edge ?e from ?p to ?t)
  (in ?St ?t)
  (not (in ?Sp ?p)))
 (action
  (modify (set ?Sp ?Sp ?p)))
 )
```

Die Regel `b` arbeitet wie die Regel `a`, mit dem Unterschied, daß überprüft wird, ob eine Kante zwischen einer Stelle `?p` und einer Transition `?t` existiert und `?t` bereits in `?St` und `?p` nicht in `?Sp` enthalten ist. Ist dies der Fall, so wird `?p` zu `?Sp` hinzugefügt. Die Regeln `a` und `b` sind gleichberechtigt und feuern in Abhängigeit der Wissensbasis und der gewählten Konfliktlösungsstrategie.

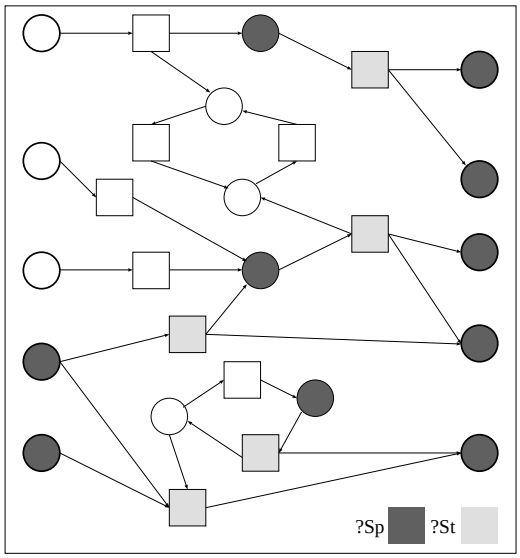

Abbildung 5.5: Die Mengen `?Sp` und `?St` nach mehrfachem Feuern der Regel `b`

Können die Regeln `a` und `b` nicht mehr feuern, so sind alle erreichbaren Stellen in `?Sp` und alle erreichbarchen Transitionen in `?St` aufgenommen worden. Es müssen jetzt nur noch alle nicht erreichten Stellen bzw. Transitionen aus `?P` bzw. aus `?T` entfernt werden. Hierzu wird die Regel `a_b_nachbereitung` benötigt. Die Priorität dieser Regel ist niedriger als die der Regeln `a` und `b`, da sie erst feuern darf, wenn alle erreichbaren Stellen und Transitionen in `?Sp` bzw. in `?St` enthalten sind.

```
(rule a_b_nachbereitung
 (priority -10)
 (condition
  (permission)
  (set ?P ?T ?Sp ?St)
  )
 (action
  (remove (difference ?P ?Sp))
  (remove (difference ?T ?St))
  (remove ?Sp ?St))
 )
) // end transformation
) // end description
```

Zuerst wird wieder überprüft, ob der Zugriff erlaubt ist. Anschließend wird gefordert, daß die benötigten Mengen existieren. Im Aktionsblock werden dann die nicht mehr benötigten Objekte aus `?P` und `?T` sowie die Mengen `?Sp` und `?St` gelöscht. Es werden also alle Elemente gelöscht, die in der Abbildung 5.6 'weiß' dargestellt werden.

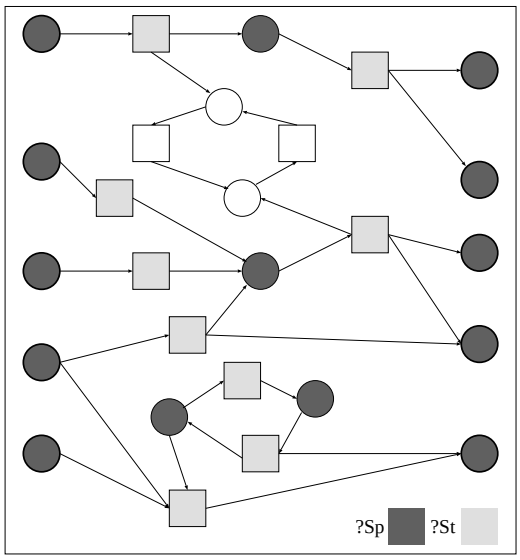

Abbildung 5.6: Das Petrinetz bevor die Senke von der Abschlußregel entfernt wird

Wenn die Regeln `a` und `b` nicht mehr feuern können, enthalten die Mengen `?Sp` und `?St` die in der Abbildung 5.6 gezeigten Elemente.
Für die Abbildung 5.4 auf der Seite 83 sowie für die Abbildung 5.5 auf der Seite 84 gilt, daß sie jeweils ein mögliches Zwischenergebnis anzeigen. Die Regeln `a` und `b` sind

beide gleichberechtigt. Welche Regel wann feuert bestimmt der Inferenzmechanismus in Abhängigkeit der gewählten Konfliktlösungsstrategie. Die Abbildung 5.2 auf der Seite 81 zeigt bereits, daß bei der realen Optimierung eine andere Reihenfolge gewählt wurde. Das dort gezeigte Ergebnis erhält man, wenn man beim Editor den Menüpunkt *Analyse* auswählt. Für denjenigen, der die Transformationen entwickelt, sind die Informationen, die das Formular liefert, hilfreich, da man schnell überprüfen kann, ob die Transformationen so arbeiten, wie dies gefordert wird.
Man kann in der Abbildung erkennen, daß nicht nur die Regeln der Transformation 3.2.12 aufgerufen wurden, sondern auch die der Transformation 3.2.13. Diese Transformation entfernt Quellen aus einem Prädikat/Transitions-Netz. Beide Transformationen sind auch im Anhang D.1 zu finden.

5.2 Strukturbildoptimierung einer Differentialgleichung

Im Bereich der Regelungstechnik kommen dynamische Systeme, die sich aus zeitveränderlichen Größen zusammensetzen und durch funktionale Beziehungen miteinander verknüpft werden, zum Einsatz. Zur Beschreibung der Ausgangsgröße in Abhängigkeit von der Eingangsgröße werden Differentialgleichungen verwendet. Durch die Anwendung der *Laplace*-Transformation erhält man Gleichungen, bei denen die Abhängigkeit zwischen der Eingangs- und der Ausgangsgröße wesentlich einfacher dargestellt wird. Man spricht auch von einer Transformation in den s-Bereich. Die *Laplace*-Transformation ist nicht beliebig anwendbar, kann aber bei linearen Differentialgleichungen mit linearen Koeffizienten eingesetzt werden. Zusätzlich werden Strukturbilder verwendet, um die Zusammenhänge anschaulicher zu gestalten.

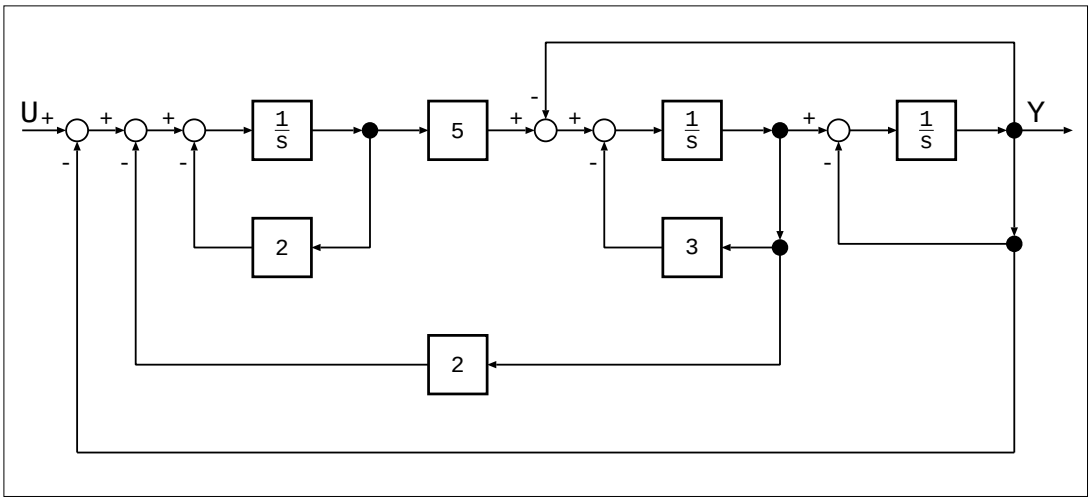

Abbildung 5.7: Strukturbild einer Differentalgleichung

Vor der Weiterverarbeitung kann ein Strukturbild in den meisten Fällen noch in eine einfachere oder übersichtlichere Form gebracht werden.
Die Abbildung 5.7 zeigt eine Differentialgleichung, die als Strukturbild im s-Bereich modelliert wurde. Es handelt sich dabei um ein Übertragungsystem mit der Eingangsgröße U(s) und der Ausgangsgröße Y(s). Zusätzlich zu der Möglichkeit, das Struktrubild in

eine übersichtlichere Form zu bringen, kann auch die Übertragungsfunktion T(s) mit Y(s) = T(s) U(s) ermittelt werden. Es gibt zwei verschiedene Lösungsmöglichkeiten. Zum einenen könnte man Gleichungen aufstellen und diese lösen, andererseits kann man die Reduktion im Strukturbild durchführen, die im folgenden beschrieben wird. Vorab werden erst noch die einzelnen Glieder, aus denen das Strukturbild zusammengesetz wurde, und die Umsetzung in ein Prädikat/Transitions-Netz kurz erläutert.

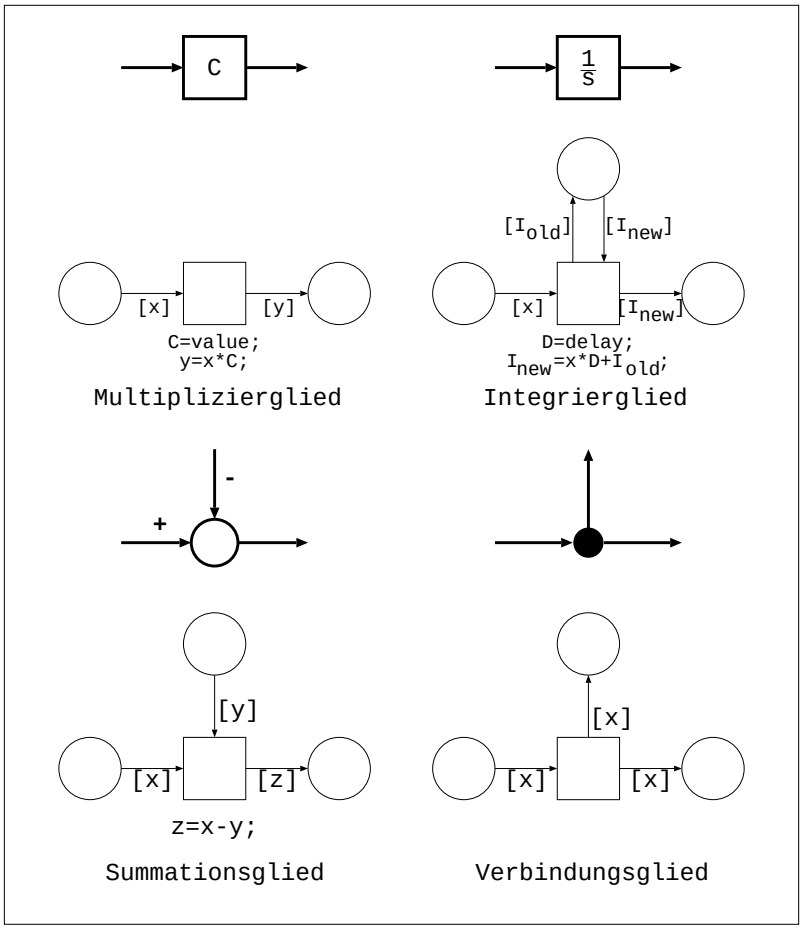

Abbildung 5.8: Module einer Differentialgleichungbibliothek

Die Abbildung. 5.8 zeigt zum einen die Elemente, aus denen das Strukturbild aufgebaut wurde, und andererseits, wie diese Elemente als Prädikat/Transitions-Netz modelliert werden können (siehe auch [brkl 93]). Es werden Multiplizierglieder benötigt, die den eingehenden Wert mit einer Konstanten multiplizieren und an den Ausgang weitergeben. Weiterhin werden Integrierglieder benötigt, die mit einem Startwert vorbesetzt werden können. Summationsglieder werden für Mit- oder Gegenkopplungen benötigt, und die Modellierung von Verzweigungen muß auch möglich sein.
Für die eigentliche Optimierung des Strukturbildes existiert ein Satz von Transformationen, der in [foel 94] im Kapitel 2.8 beschrieben wird. Diese Transformationen lassen sich in eine Gruppe von Zusammenfassungstransformationen und in eine Gruppe von Vertauschungtransformationen unterteilen. Zu den Zusammenfassungstransformationen gehört das Vereinfachen von Parallelschaltungen, Reihenschaltungen, Gegen-

kopplungen und Mitkopplungen. Die Vertauschungstransformationen können zum Vertauschen zweier Blöcke oder zum Verlegen eines Blocks vor bzw. hinter ein s-Glied oder eine Verzweigungsstelle eingesetzt werden. Wesentlich ist, daß es sich jeweils um zeitinvariante Übertragungsglieder handelt. Daher lassen sich diese Transformationen nicht anwenden, falls beispielsweise Kennlinienglieder beteiligt sind.

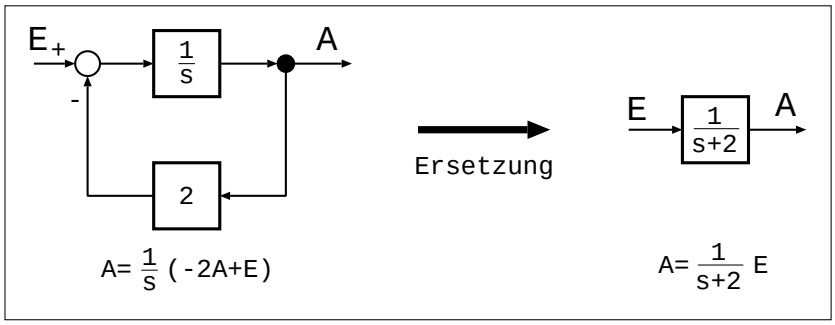

Abbildung 5.9: Schleife bestehend aus Integrier- und Multiplizierglied

In der Abbidlung 5.9 ist ein Teilsystem aus dem Strukturbild von Seite 86 mit der Eingangsgröße E und der Ausgangsgröße A zu sehen. Dieses Teilsystem, es handelt sich um eine Gegenkopplung, kann durch die ein einfacheres System ersetzt werden. Das Ergebnis ist rechts in der Abbildung zu sehen. Welche Berechnung durchgeführt werden muß, ist auch zu erkennen.

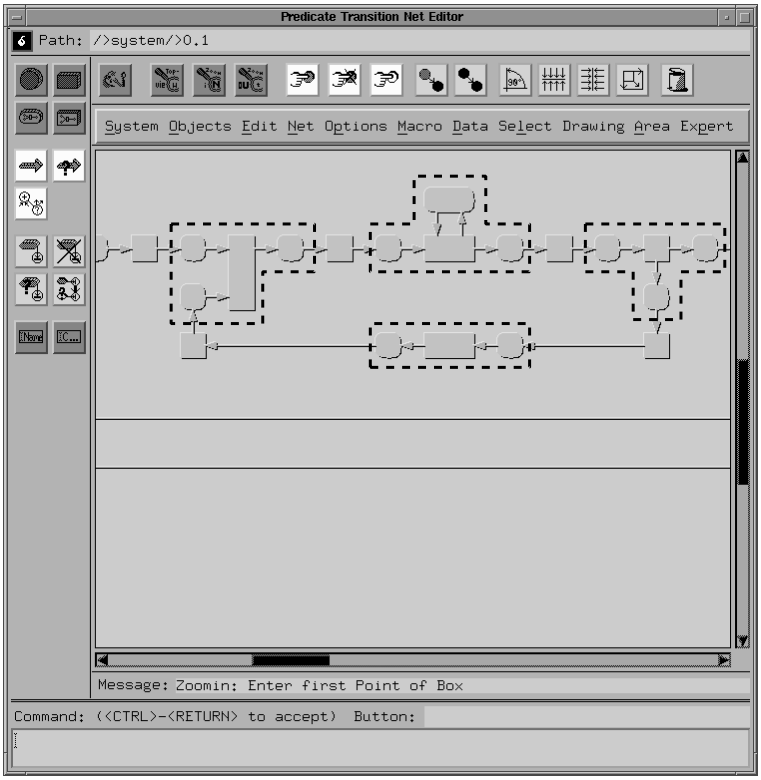

Abbildung 5.10: Gegenkopplung bestehend aus Integrier- und Multiplizierglied

Die Abbildung 5.10 zeigt die mit dem Editor modellierte Gegenkopplung. Die gestrichelten Rahmen sind zusätzlich eingezeichnet worden, damit man die Module, aus denen diese Schleife zusammengesetzt wurde, besser wiedererkennt.

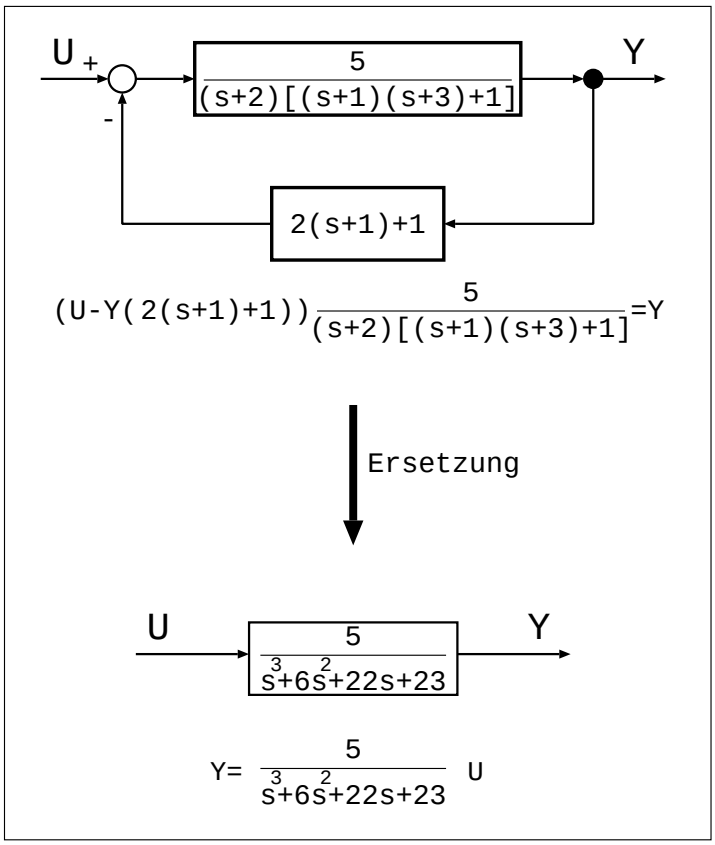

Abbildung 5.11: Schleife bestehend aus zwei Integrierglieder

Die Abbildung 5.11 ist ein weiteres Beispiel für ein Teilsystem, das durch ein einfacheres System ersetzt werden kann. Es handelt sich bei der Ersetzung um den letzten Schritt der Vereinfachung des Strukturbildes von Seite 86 (Abbildung 5.7). Zusätzlich zu der Vereinfachung des Strukturbildes kann man unmittelbar die Übertragungsfunktion T(s) ablesen.
Um die Übertragungsfunktion zu bestimmen und die Optimierung des Strukturbildes zu erreichen, werden fünf Transformationen, von denen eine auf den folgenden Seiten besprochen wird, benötigt. Durch das `permission`-Konstrukt wird ausgedrückt, daß alle Transformationen gleichberechtigt sind. Die Transformation `dgln_2` und `dgln_5` bestehen jeweils aus zwei Regeln, da bei diesen beiden Transformationen vor der Neuberechnung der Annotation, die Variablen umbenannt werden müssen, damit eine korrekte Berechung durchführbar ist. Bei den Transformationen 1, 2, 3 und 5 handelt es sich um Zusammenfassungstransformationen und bei Transformation 4 um eine Vertauschungstransformationen. Zur Berechnung der Übertragungsfunktion wird MuPAD eingesetzt. Im folgenden wird die Transformation `dgln_2` beschrieben. Bei dieser Transformation wird das Prädikat/Transitions-Netz zum einen strukturell verändert, und anderseits wird die *preaction* mit MuPAD neu berechnet. Alle Transformationen sind im Anhang

D.2 zu finden.
Die Transformation `dgln_2` besteht aus zwei Regeln. Die erste Regel wird benötigt, damit vor der eingentlichen Zusammenfassung die Bezeichner umbenannt werden können, um anschließend mit der zweiten Regel die Zusammenfassung durchzuführen.

```
(description
 dgln

(permission dgln_1:loop_integ_mult
            dgln_2:loop_integ_integ_rename
            dgln_3:integ_mult
            dgln_4:pos
            dgln_5:integ_integ_rename)

// Bei der zweiten Transformation werden zwei in einer Schleife
// zusammengeschaltete Integrierglieder zu einem zusammengefaßt.
// Vorher müssen die Bezeichner umbenannt werden, damit eine
// korrekte Ersetzung möglich ist.

 (transformation
  dgln_2
  (rule
   loop_integ_integ_rename
   (condition
    (permission)

// Addierglieder in allen Einzelheiten beschreiben
    ...
// Integrierglied 1
    ...
    (transition ?t_integ_1
                preaction ?t_integ_1_pre)
    ...
// Verzweigung
    ...
// Integrierglied 2
    ...
    (transition ?t_integ_2
                preaction ?t_integ_2_pre)
    ...
// Die Verbindung der Module ist bereits integriert
// Beschreibung des Ein- und Ausgangs fehlt noch
    ...
    ) // end condition
```

Die durch die Punkte angedeuteten Passagen, werden an dieser Stelle zur Erklärung nicht benötigt, sie sind aber im Anhang D.2 enthalten. Im Bedingungsblock werden die strukturellen Eigenschaften beschrieben, die gegeben sein müssen. D.h die einzelenen

Module aus denen sich die Schleife zusammensetzt und ihre Verbindung wird detailliert beschrieben. Für die beiden Transitionen der Integrierglieder werden Bezeichner vergeben, damit eine Identifikation im Aktionsblock möglich ist.

```
(action
 (modify (transition ?t_integ_1
                     preaction [rename i_new i_new_1]))
 (modify (transition ?t_integ_2
                     preaction [rename i_new i_new_2]))
 (permission save dgln_2:loop_integ_integ)
 ) // end action
) // end rule loop_integ_integ_rename
```

Ist die Schleife im Bedingungsblock identifiziert worden, so muß im Aktionsblock darfür gesorgt werden, daß die Bezeichner umbenannt werden. Zusätzlich werden die aktuellen Zugriffsrechte gerettet, und es werden die Zugriffsrechte für die zweite Regel, die die eigentliche strukturelle Optimierung und die Neuberechnung durchführt, der Transformation gesetzt.

```
(rule
 loop_integ_integ
 (condition
  (permission)
 // Identisch mit vorheriger Regel bis auf folgende Zeile
  ...
  (not (peq (?t_integ_1 ?t_integ_2)))
  ...
  ) // end condition
```

Der Bedingungsblock der zweiten Regel ist mit dem der ersten Regel nahezu identisch. Der einzige Unterschied ist, daß gefordert wird, daß die Annotationen der beiden beteiligten Transitionen verschieden sind. Gerade diese Situation hat das Feuern der vorherigen Regel herbeigeführt. Somit wird sichergestellt, daß auch die richtige Schleife ausgewählt wird.

```
 (action
 // Löschen nicht mehr benötigter Objekt
  ...
  (modify (transition ?t_integ_1
                      preaction [calculate i_new
                            i_new=(1+i_new_1*i_new_2)/i_new_1;
                            ?t_integ_2_pre ?t_integ_1_pre]))

 // Die Verbindungen des Eingangs und Ausgangs
  ...
  ) // end action
 ) // end rule loop_integ_integ
) // end transformation dgln_2
) // end description
```

Im Aktionsblock können nun die nicht mehr benötigten Objekte gelöscht werden, und mit MuPAD kann die Berechnung der *preaction* der Transition, die erhalten bleibt, durchgeführt werden. Nachdem diese Regel gefeuert hat, sorgt der interne Regelsatz dafür, daß wieder die alten Zugriffsrechte gesetzt werden und somit wieder alle Transformationen gleichberechtigt sind (siehe auch Kapitel 3.5).
Durch die einzelenen Transformationen wird das Strukturbild nach und nach zusammengefaßt. Zusätzlich werden die notwendigen Berechnungen durchgeführt. Letztendlich erhält man das in der Abbildung 5.11 auf der Seite 5.11 dargestellte Ergebnis.

5.3 Zeit- und Platzbedarf bei der Optimierung

Möchte man ein System mit der Optimierungskomponente des Prädikat/Transitions-Netz-Editors optimieren, so ist der Zeitbedarf ein wesentlicher Aspekt. Während die im Kapitel 4.1.2 angesprochene Konvertierung der Daten kaum Zeit benötigt, ist der Zeit- und Platzbedarf des Expertensystems wesentlich höher und auch ausschlaggebend. Zwei voneinander abhängige Punkte, die anschließend an den beiden Anwendungsbeispielen verdeutlicht werden, sind wesentlich:

1. Die Größe des Systems bzw. des Prädikat/Transitions-Neztes

   Je größer das Prädikat/Transitions-Netz ist, desto mehr Fakten werden bei der Konvertierung generiert. Dies führt dazu, daß die Anzahl der zu überprüfenden Muster, je nach Art der Regeln (siehe 2.) stark steigt. Dies erfordert zum einen viel Zeit aber auch viel Platz. Es ist zu beachten, daß sich der Faktensatz nach jedem Optimierungsschritt verändert, so daß auch jedes mal die Bedingungen der Regeln überprüft werden müssen. Der Vergeleichsprozeß führt dazu, daß entweder neue Regeln zur Agenda hinzugefügt oder Regeln von der Agenda entfernt werden.

2. Die Anzahl und Art der Regeln, die zur Optimierung eingesetz werden

   Ein Faktor für die benötigte Zeit ist die Anzahl der benötigten Regeln, da die Anzahl der zu überprüfenden Bedingungen natürlich mit der Anzahl der Regeln steigt. Der wesentlichere Aspekt ist die Art der Regeln. Dies bedeutet, daß die Formulierung einer Regel und ihr Einsatzgebiet wesentlich ist. Beispielsweise kann der Transformationssatz so gestalltet sein, daß sehr viele Muster existieren, die die Bedingungen der Regeln erfüllen. Dies führt dazu, daß der Vergleichsprozeß viel Zeit benötigt. Wird im Bedingungsblock einer Regel ein größerer Ausschnitt eines Prädikat/Tranitions-Netzes beschrieben, so kann schnell entschieden werden, ob die Bedingung erfüllt ist. Der Vergleichsprozeß kann dadurch beschleunigt werden. Besteht der Teil des Prädikat/Transitions-Neztes, der in der Bedinung der Regel gefordert wird, aus mehreren Modulen, die miteinander verbunden sind, so ist es wesentlich, daß nach der Beschreibung eines Moduls unmittelbar die Verbindung zum nächsten Modul beschrieben wird. Werden beispielsweise einzelne Module bei der Modellierung des Systems mehrfach verwendet, so kann es sein, daß bestimmte Verbindungen zwischen diese Modulen selten auftreten.

Bei dem Beispiel, das im Kapitel 5.1 besprochen wird, wurde nur auf eine von vielen Regeln, die zur Optimierung der Struktur eines Petrinetzes eingesetzt werden können, eingegangen. Modelliert man weitere der in [klei 94] angegebenen Transformationen, so

stellt man fest, daß bei vielen Transformationen jeweils nur ein sehr kleiner Ausschnitt eines Petrinetzes benötigt wird, um die Bedingung einer Regel zu erfüllen. Dies hat zur Folge, daß bereits bei einem kleinen Petrinetz sehr viele Regeln feuern könnten. Wobei natürlich der Fall auftritt, daß dieselben Fakten die Bedingungen mehrerer Regeln erfüllen. Dies zieht unmittelbar nach sich, daß nach dem Feuern einer Regel viele Bedingungen erneut überprüft werden müssen und die Anzahl der Regeln, die zur Agenda hinzugefügt oder von der Agenda entfernt werden, nach jedem Feuern einer Regel, groß ist. Der Zeit- und Platzbedarf ist daher bei diesem Transformationssatz sehr groß.
Die Optimierung des Strukturbildes der Differentialgleichung dauert nur wenige Minuten. Dies liegt zum einen daran, daß die Transformationen sehr exakt einen Teil des Prädikat/Transitions-Netzes beschreiben. Der wesentlichste Faktor ist, daß bei den Transformationen unmittelbar nach der Beschreibung eines Moduls die Verbindung zum nächsten Modul beschrieben wird. Dies hat zur Folge, daß die Anzahl der in Frage kommenden Muster sich schnell verringert. Der Optimierungsprozeß wird dadurch wesentlich beschleunigt.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde untersucht, wie ein wissensbasiertes Werkzeug zur Optimierung von Systemspezifikationen eingesetzt werden kann. Hierzu wurde die im C-LAB entwickelte Prädikat/Transitions-Netz-Entwicklungsumgebung um eine wissensbasierte Optimierungskomponente erweitert. Die Systeme, die analysiert oder optimiert werden sollen, müssen als Prädikat/Transitions-Netz modelliert werden, da zur Darstellung und Simulation der zur Entwicklungsumgebung gehörende Prädikat/Transitions-Netz-Editor verwendet wird. Die Schwerpunkte der Arbeit waren,

- die Einführung einer Transformationssprache, um die Formulierung von Transformationen, die zur Optimierung eingesetzt werden sollen, zu ermöglichen.
- die Integration des wissensbasierten Werkzeugs (CLIPS) in die Entwicklungsumgebung.
- die Umsetzung einer mit der Transformationssprache erstellten Beschreibung in einen Regelsatz, der von CLIPS verarbeitet werden kann.

  Da die Basisversion von CLIPS nicht die benötigte Funktionalität bietet, die die Umsetzung aller Sprachkonstrukte der Transformationssprache erlaubt, wurde

  - der Funktionsumfang von CLIPS erweitert.
  - das Computer-Algebra-System MuPAD in die Architektur integriert.
  - ein Regelsatz erstellt, der speziell an die Anforderungen, die sich durch die Transformationssprache in Verbindung mit den Prädikat/Transitions-Nezten ergeben haben, angepaßt ist und notwendige Operationen während des Optimierungsprozesses durchführt.
- die exemplarische Untersuchung des Laufzeitverhaltens

Hat man das System, das man optimieren möchte als Prädikat/Transitions-Netz modelliert, die notwendigen Grundlagen wurden im Kapitel 2.1 eingeführt, so muß man sich anschließend geeignete Transformationen überlegen, die zur Optimierung des Systems führen sollen. Da das System nun als Prädikat/Transitions-Netz vorliegt, wird man sich die Tranformationen, die i. allg. atomar ausgeführt werden müssen, in der Notation der Prädikat/Transitions-Netze überlegen.

Um die Optimierungskomponente einsetzen zu können, muß man die Transformationen in eine Beschreibung umsetzen, die die Notation der Transformationssprache einhält. Diese Beschreibung muß letztendlich in einen Fakten- und Regelsatz, der von CLIPS verarbeitet werden kann, umgesetzt werden.
Eine Beschreibung kann aus mehreren Transformationen bestehen, die aus mehreren Regeln bestehen können. Eine Regel besteht aus einem Bedingungsblock, in dem exakt ein Ausschnitt des Prädikat/Transitions-Neztes beschrieben wird, und einem Aktionsblock, in dem Aktionen zur Modifikation des Prädikat/Transitions-Netzes durchgeführt werden können. Zusätzlich werden Funktionen zur Verfügung gestellt, mit deren Hilfe im Bedingungsblock Attribute verglichen und im Aktionsblock Modifikationen vorgenommen werden können (siehe auch Kapitel 3.3).
Da die Regeln eines Regelsatzes eines Expertensystems im Normalfall geleichberechtigt sind, eine Ausnahme kann nur durch die Vergabe von Prioritäten erreicht werden, besitzt die Transformationssprache einen Kontrollmechanismus, der es erlaubt, mehrere Regeln zu einer Transformation zusammenzufassen (siehe auch 3.5).
In der Arbeit kommen nur nicht hierarchische Prädikat/Transitions-Netze zum Einsatz. Daher besteht hier eine Möglichkeit zur Erweiterung, die die Modellierung der Systeme und die Formulierung der Transformationen vereinfachen würde. Eine entsprechende Erweiterung der Transformationssprache müßte durchgeführt werden.
Der wichtigste Baustein der Optimierungskomponente ist CLIPS, das gewählt wurde, da es alle Eigenschaften, die für die zu lösende Aufgabe benötigt werden, besitzt.
Hierzu gehört die Möglichkeit, Wissen durch Objekte und Regeln darzustellen. Weiterhin verwendet CLIPS die Problemlösungsstrategie des monotonen Schließens. Diese Art der Wissensdarstellung zusammen mit der zum Einsatz kommenden Problemlösungsstrategie eignet sich besonders gut, um Optimierungsprobleme zu lösen (siehe auch Kapitel 2.2.3).
Ein weiterer wichtiger Punkt war, daß CLIPS in C geschrieben wurde und daher die Integration in die bestehende Architektur problemlos möglich war.
Hinzu kommt, daß der Funktionsumfang von CLIPS beliebig erweiterbar ist. Dies war für die Arbeit wesentlich, da sonst wesentliche Teile der Transformationssprache nicht realisierbar gewesen wären. Die Funktionen, die zu CLIPS hinzugefügt wurden, haben die Aufgabe, die Umsetzung von Operationen, die die Transformationssprache anbietet, zu ermöglichen. Beispielsweise wurden Funktionen für Vergleiche und Berechnungen von Annotationen sowie für die Realisierung von Mengen benötigt (siehe auch Kapitel 4.2.1). Zusätzlich wurde für die Berechnungen, die beim Setzen der Annotationen der Transitionen verwendet werden könnnen, und die Normierung von Ausdrücken, die für die Vergleiche benötigt wird, das Computer-Algebra-System MuPAD in die Architektur integriert. Sollen mit den Funktionen, um die CLIPS erweitert wurde, Annotationen der Transitionen verglichen werden, so dürfen diese nur aus mathematischen Ausdrücken bestehen (siehe auch Kapitel 3.3.5.4). Hier wäre eine Erweiterung auf die volle Funktionalität (siehe [rust 95]), die der Prädikat/Transitions-Netz-Editor bietet, sinnvoll.
Da die Transformationssprache Operationen zur Verfügung stellt, die nicht unmittelbar mit einer Regel realisiert werden können, wurde ein zusätlicher Regelsatz, der in der Arbeit als impliziter Regelsatz bezeichnet wird, benötigt. Dieser Regelsatz übernimmt Aufgaben, die nicht unmittelbar im Aktionsblock einer Regel ausgeführt werden können. Weiterhin muß der Regelsatz Verwaltungsaufgaben während des Optimierungsprozesses übernehmen (siehe auch Kapitel 4.2.2).

An zwei Beispielen wurde gezeigt, wie gegebene Transformationen mit der Transformationssprache formuliert werden können. Zusätzlich wurde an diesen Beispielen gezeigt, daß die Art der Transformationen und die größe des Systems wesentlich sind, wenn die Optimierung in einer akzptablen Zeit durchgeführt werden sollen. Beschreibt man im Bedingungsblock einer Regel einen Ausschnitt eines Systems, so ist es wesentlich, daß die Verbindungen zwischen den Modulen unmittelbar nach der Beschreibung des Moduls erfolgt (siehe auch Kapitel 5.3).
Um auch größere Systeme optimieren zu können, müßte ein Algorithmus implementiert werden, der das Prädikat/Transitions-Netz stückweise in Wissen wandelt und die so entsthenden Ausschnitte optimiert. Voraussetzung hierfür wäre, daß die Transformationen lokal arbeiten.
Eine Überlegung zu Beginnn der Arbeit war, daß die Optimierungskomponente auch für einen Anwender, der nicht gleichzeitg ein Experte ist, einsetzbar sein soll. Für den Fall, daß sich ein Experte mit der Modellierung des Systems und der Erstellung geeigneter Transformationen auseinandergesetzt hat, und die Ergebnisse in Form von Bibliotheken vorliegen, kann sich ein möglicher Anwender auf das Zusammensetzen seines Systems und den globalen Ablaufprozeß der Optimierung beschränken. In diesem Fall, der eigentlich der wünschenswerte ist, kommen auf ihn nur steuernde Aufgaben zu.

Literaturverzeichnis

[ager 79] T. Agerwala *Putting Petri Nets to Work*. Computer, 12(12), 1979

[aviv 95] *Analyse von integrierten Verkehrsystemen*. internes Papier, C-LAB, 1995

[brjoleloe 94] H.-J. Brede, N. Josuttis, S. Lemberg, A. Lörke, *Programmieren mit OSF/Motif*. Addison-Wesley, 1994

[brkl 93] M. Brielmann und B. Kleinjohann. *A Formal Model for Coupling Computer Based Systems and Physical Systems*. Aus Proceedings of the European Design Automation Conference, Hamburg, 1993.

[buesc 88] Kleine Büning/Schmitgen, *PROLOG*. Teubner, 1988

[crba 93] *CLIPS Volume I - Basic Programming Guide*. Reference Manual. NASA, 1993

[crad 93] *CLIPS Volume II - Advanced Programming Guide*. Reference Manual. NASA, 1993

[crin 93] *CLIPS Volume III - Interface Guide*. Reference Manual. NASA, 1993

[curu 93] *CLIPS Volume I - Rules*. User Guide. NASA, 1993

[cuob 93] *CLIPSVolume II - Objects*. User Guide. NASA, 1993

[crhejue 91] Cremers/Heinz/Jünemann, *Expertensysteme für die Planung der Produktion*. TÜV Rheinland, 1991

[denn 85] J. B. Dennis , *Models of Data Flow Computation in M. Bory (ed.), Control Flow and Data Flow: Concepts of Distributed Programming*. Springer, 1985

[drkr 95] P. Drescher, R. Kruschinski, *Detailed Functional Specification, IFS, Basic Services IFS 1-2*. C-LAB, 1995

[foel 94] Otto Föllinger, *Regelungstechnik, Einführung in die Methoden und ihre Anwendung*. Hüthig, 1994

[fuch 94] Benno Fuchssteiner et al. *Tutorial; MuPAD (Multi Processing Algebra Data Tool) Version 1.2*. Birkhäuser Verlag Basel, 1994

[fuch 93] Benno Fuchssteiner et al. *Benutzerhandbuch; MuPAD (Multi Processing Algebra Data Tool) Version 1.1*. Birkhäuser Verlag Basel, 1993

[gela 81] H. J. Genrich, K. Lautenback, *System Modelling with High-Level Petri Nets*. Theoretical Computer Science, 13, North Holland, 1981.

[goer 95] G. Görz, *Einführung in die künstliche Intelligenz* . Addison-Wesley, 1995, 2. Auflage

[hare 78] D. Harel *Statecharts: A visual formalism for complex systems*. Science of Computer Programming, 8, 1978

[haki 89] P. Harmon/D.King, *Exptertensysteme in der Praxis*. Oldenbourg, 1989

[hoku 91] D. Hofbauer/R. D. Kutsche, *Grundlagen des maschinellen Beweisens*. Vieweg, 1991

[jack 87] P. Jackson, *Expertensysteme*. Addison-Wesley, 1987

[kast 90] U. Kastens, *Übersetzerbau*. Oldenbourg, 1990

[kirs 92] D. Kirstein *TransNet - Ein Interpreter zur mengentheoretischen Transformation hierarchischer Petrinetze*. Diplomarbeit, Universität GH Paderborn, 1992.

[klei 94] B. J. Kleinjohann, *Synthese von zeitinvarianten Hardware-Modulen*. Dissertation, Universität GH Paderborn, 1994

[klei 93] E. Kleinjohann, *Integrierte Entwurfsberatung auf der Basis erweiterter Prädikat/Transitions-Netze*. Dissertation, Universität GH Paderborn, 1993

[lauf 95] G. Laufkötter, *Detailed Functional Specification, IFS, Basic Services IFS 1-2, Knowledge Evaluation Component*. C-LAB, 1995

[litt 92] Kurt Littger, *Optimierung, Eine Einführung in rechnergestützte Methoden und Anwendungen*. Springer-Verlag, 1992

[lipp 95] S. B. Lippman, *C++, Einführung und Leitfaden*. Addison-Wesley, 1995

[maye 88] O. Mayer, *Programmieren in COMMON LISP*. B.I. Wissenschaftsverlage, 1988

[meye 92] S. Meyers, *Effektiv C++ Programmieren*. Addison-Wesley, 1992

[obvo 89] W. Oberschelp/G. Vossen *Rechneraufbau und Rechnerstrkturen*. Oldenbourg, 1989

[pete 81] J. L. Peterson, *Petri Net Theory And The Modeling of systems*. Prentice-Hall, 1981

[pupp 91] F. Puppe, *Einführung in Expertensysteme* 2. Auflage, Studienreihe Informatik. Springer, 1991

[push 93] T. Push, *Der C++-Objekt-Manager*. internes Papier, C-LAB, 1993

[petr 62] C. A. Petri, *Kommunikation mit Automaten*. Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, Bonn, 1962

[ramm 89] F. J. Rammig *Systematischer Entwurf digitaler Systeme*. Teubner Stuttgart, 1989.

[reis 91] W. Reisig, *Petrinetze*. Springer, 1991

[rust 95] C. Rust, D. Stichling *Annotationen an Prädikat/Transitions-Netzen*. internes Papier, C-LAB, 1995.

[scle 86] P. Schnupp/U. Leibrandt, *Expertensysteme*. Springer-Verlag, 1985

[savo 85] S. Savory, *Künstliche Intelligenz und Expertensysteme*. Oldenbourg, 1985

[savo 90] S. Savory, *Grundlagen von Expertensystemen*. Oldenbourg, 1990

[stro 92] B. Stroustrup, *Die C++ Porgrammiersprache*. Addison-Wesley, 1992

[tack 92] J. Tacken *Steuerung und Überwachung von Entwurfssystemen mit Hilfe von Prädikat/Transitions-Netzen*. Diplomarbeit, Universität GH Paderborn, 1992.

[taklkl 96] J. Tacken, E. Kleinjohann, B. Kleinjohann *The SEA Language for System Engineering and Animation*. C-LAB, 1996.

[vale 87] R. Valette. Nets in Production Systems. Aus W. Brauer, W. Reisig und G. Rozenberg (Hrsg.) *Advances in Petri Nets 1986, Petri Nets: Application and Relationships to Ohter Models of Concurrency, Ausgabe 255 von Lecture Notes in Computer Science*. Springer, 1987. Part II.

[voss 87] K.Voss Nets in Data Bases. Aus W. Brauer, W.Reisig und G. Rozenberg (Hrsg.); *Advances in Petri Nets 1986, Petri Nets: Application and Relationships to Ohter Models of Concurrency, Ausgabe 255 von Lecture Notes in Computer Science*. Springer, 1987, Part II.

Anhang A

Datentypen in der CLIPS-Notation

Im folgenden werden, die Datentypen in der CLIPS-Notation mit allen ihren Attributen gezeigt. Die Mengen, Kanten Stellen und Transitionen und ihre Attribute werden im Kapitel 3.2 erläutert.

```
; Der Datentyp Menge

(deftemplate set
  (slot name
        (type SYMBOL))
  (multislot object)
  (slot info
        (type SYMBOL))
)

; Der Datentyp Kante

(deftemplate edge
  (slot id
        (type SYMBOL))
  (slot name
        (type SYMBOL ))
  (slot from
        (type SYMBOL))
  (slot to
        (type SYMBOL))
  (multislot annotation
             (default 1 lbracket ident x rbracket))
  (slot info
        (type SYMBOL))
)

; Der Datentyp Stelle
```

```
(deftemplate place
  (slot id
        (type SYMBOL))
  (multislot in_nodes)
  (multislot out_nodes)
  (slot name
        (type SYMBOL))
  (slot in
        (type INTEGER)
        (default 0))
  (slot out
        (type INTEGER)
        (default 0))
  (slot lower_bound
        (type INTEGER)
        (default -1))
  (slot upper_bound
        (type INTEGER)
        (default -1))
  (multislot predicate)
  (slot mark_number
        (type INTEGER)
        (default 1))
  (multislot mark
             (default lbracket 1 rbracket))
  (slot info
        (type SYMBOL))
)

; Der Datentyp Transition

(deftemplate transition
  (slot id
        (type SYMBOL))
  (multislot in_nodes)
  (multislot out_nodes)
  (slot name
        (type SYMBOL))
  (slot in
        (type INTEGER)
        (default 0))
  (slot out
        (type INTEGER)
        (default 0))
  (slot enable_delay_min
             (type SYMBOL INTEGER)
```

```
            (default 0))
  (slot enable_delay_avg
            (type SYMBOL INTEGER)
            (default 0))
  (slot enable_delay_max
            (type SYMBOL INTEGER)
            (default 0))
  (slot firing_delay_min
            (type SYMBOL INTEGER)
            (default 0))
  (slot firing_delay_avg
            (type SYMBOL INTEGER)
            (default 0))
  (slot firing_delay_max
            (type SYMBOL INTEGER)
            (default 0))
  (multislot delay_semantic)
  (multislot marking_mode)
  (multislot demarking_mode)
  (multislot condition)
  (multislot preaction)
  (multislot postaction)
  (slot info
        (type SYMBOL))
)
```

Anhang B

Grammatik der Transformationssprache

Die Beschreibung

'(' '**description**' *c_ident* [set_permission|create_set|global|transformation]∗ ')'

Globale Konstrukte

set_permission ::= '(' '**permission**' [*c_ident*':'*c_ident*]+ ')'

create_set ::= '(' '**create**' [*ident*]+ ')'

global ::= '(' '**global**' [*global_ident* '=' *number*]+ ')'

Transformationen

transformation ::= '(' '**transformation**' *c_ident* [rule]+ ')'

Regeln

rule ::= '(' '**rule**' *c_ident* [priority]? [condition] [action] ')'

Prioritäten

priority ::= '(' '**priority**' ['+' , '-']? *number* | *global_ident* ')'

Bedingungsblock

```
condition ::= '(' 'condition'
                [ permission_cond | fact_cond | logic_cond
                  compare_cond | set_func ]* ')'
```

Aktionsblock

```
action ::= '(' 'action'
             [ permission_act | create | add_act
               modify_act | remove_act ]* ')'
```

Der Bedingungsblock

Zugriffsrechte abfragen

`permission_cond` ::= '(' '**permission**' ')'

Objekte beschreiben

`fact_cond` ::= [`place_cond` | `transition_cond` | `edge_cond` | `set_cond`]+

Stellen

```
place_cond ::= '(' 'place'          ident
                   ['in'            number | ident]?
                   ['out'           number | ident]?
                   ['lower_bound'   number | ident]?
                   ['upper_bound'   number | ident]?
                   ['predicate'     number | ident]?
                   ['mark_number'   number | ident]?
                   ['mark'          tpl_start | ident]? ')'
```

Transitionen

```
transition_cond ::= '(' 'transition'          ident
                        ['in'                 number | ident]?
                        ['out'                number | ident]?
                        ['enable_delay_min'   number | ident]?
                        ['enable_delay_avg'   number | ident]?
                        ['enable_delay_max'   number | ident]?
                        ['firing_delay_min'   number | ident]?
                        ['firing_delay_avg'   number | ident]?
                        ['firing_delay_max'   number | ident]?
                        ['condition'          cond_start | ident]?
                        ['preaction'          cmd_start | ident]?
                        ['postaction'         cmd_start | ident]? ')'
```

Kanten

```
edge_cond ::= '(' 'edge'         ident
                  'from'         ident
                  'to'           ident
                  ['annotation'  var_fmt_start | ident]? ')'
```

Mengen

`set_cond` ::= '(' '**set**' [*ident*]+ ')'

and, or und not Verknüpfungen

```
logic_cond ::= '(' [ 'and' | 'or' | 'not' ]
                   [ logic_cond | permission_cond | fact_cond |
                     compare_cond | set_func ]+ ')'
```

Vergleiche

```
compare_cond ::= '('  [ [ '=' | '>' | '<' | '>=' | '<=' | '<>' ]
                        [ [ '(' 'card' set_block ')' | number | ident ]
                          [ '(' 'card' set_block ')' | number | ident ]+ ] ]
                    | [ [ 'eq' | 'neq' ]
                        [ ident [ ident ]+ ] ]
                    | [ [ 'meq' ]
                        [ ident [ ident | tpl_start ] ] ]
                    | [ [ 'aeq' ]
                        [ ident [ ident | var_fmt_start ] ] ]
                    | [ [ 'ceq' ]
                        [ ident [ ident | cond_start ] ] ]
                    | [ [ 'peq' ]
                        [ ident [ ident | cmd_start ] ] ] ')'
```

Die Zugriffsfunktionen

```
set_func ::= [  [ '(' 'subset' set_block set_block ')' ]
              | [ '(' 'empty' set_block ')' ]
              | [ '(' 'in' set_block ident ')' ] ]
```

Die Transformationsfunktionen

```
set_tmp ::= '(' [ union | intersection | difference ] set_block set_block ')'
```

```
set_block ::= [  [ ident ]
               | [ in_ident | ident_out ]
               | [ set_tmp ] ]
```

Der Aktionsblock

Zugriffsrechte setzen

```
permission_act ::= '(' 'permission' [ 'save' ]? [ c_ident':'c_ident ]+ ')'
```

Objekte hinzufügen

```
add_act ::= '(' 'add' [ place_act | transition_act | edge_act | set_act ] ')'
```

Stellen

```
place_act ::= '(' 'place'          ident
              ['lower_bound' [ number | ident ]+ ]?
              ['upper_bound' [ number | ident ]+ ]?
              ['predicate'   [ number | ident ]+ ]?
              ['mark'        [ tpl_start ]? [ ident]* ')'
```

Transitionen

```
transition_act ::= '(' 'transition'          ident
                   ['enable_delay_min' [ number | ident ]+ ]?
                   ['enable_delay_avg' [ number | ident ]+ ]?
                   ['enable_delay_max' [ number | ident ]+ ]?
                   ['firing_delay_min' [ number | ident  ]+ ]?
                   ['firing_delay_avg' [ number | ident  ]+ ]?
                   ['firing_delay_max' [ number | ident  ]+ ]?
                   ['condition'        cond_act ]?
                   ['preaction'        cmd_act ]?
                   ['postaction'       cmd_act ]? ')'

cond_act ::= [  [ [ '[' cond_act ']' ] | cond_act ]
                [ [ '&&' | '||' | '!' ] [ [ '[' cond_act ']' ] | cond_act ] ]? ]
             | [  [ '[' 'calculate' [ cond_start | ident ] [ ident ]+ ']' ]
                | [ cond_start ]
                | [ ident ] ] ]

cmd_act ::= [ [ '[' 'calculate' c_ident [ cmd_start | ident ] [ident]+ ']' ]*
              [ cmd_start ]? [ident]* ]
```

Kanten

```
edge_act ::= '(' 'edge'
             'from'       [ ident | in_ident | ident_out ]
             'to'         [ ident | in_ident | ident_out ]
             ['annotation' [ var_fmt_start ]? ident]* ')'
```

Mengen

```
set_act ::= '(' 'set' ident [ set_block ]+ ')'
```

Objekte verändern

```
modify_act ::= '(' 'modify' [ place_act | transition_mod | edge_mod | set_act ] ')'
```

Transitionen

```
transition_mod ::= '(' 'transition'          ident
                   ['enable_delay_min' [ number | ident ]+ ]?
                   ['enable_delay_avg' [ number | ident ]+ ]?
```

['**enable_delay_max**' [*number* | *ident*]+]?
['**firing_delay_min**' [*number* | *ident*]+]?
['**firing_delay_avg**' [*number* | *ident*]+]?
['**firing_delay_max**' [*number* | *ident*]+]?
['**condition**' ['[' '**rename**' *c_ident c_ident* ']'] | [`cond_act`]]?
['**preaction**' ['[' '**rename**' *c_ident c_ident* ']'] | [`cmd_act`]]?
['**postaction**' ['[' '**rename**' *c_ident c_ident* ']'] | [`cmd_act`]]? ')'

Kanten

`edge_mod` ::= '(' '**edge**' *ident*
'**annotation**' [['[' '**rename**' *c_ident c_ident* ']']
| [[`var_fmt_start`]? [*ident*]∗] ')'

Objekte löschen

`remove_act` ::= [[*ident* | *in_ident* | *ident_out*]+
| [`edge_act`]
| [`set_tmp`]]

elementare Konstrukte

number ::= ['-']?[*digit*]+ | ['-']?[*digit*][*digit*]∗.[*digit*][*digit*]∗

global_ident ::= '?∗'[*c_ident*]∗'∗'

ident ::= '?'[*c_ident*]∗

in_ident ::= '?∗'[*c_ident*]∗

ident_out ::= '?'[*c_ident*]∗'∗'

c_ident ::= [*l_letter* | *u_letter* | '_'][*digit* | *l_letter* | *u_letter* | '_']∗

digit ::= ['0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9']

l_letter ::= ['a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' |
'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' |
's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z']

u_letter ::= ['A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z']

Anhang C

Der implizite Regelsatz

In diesem Anhang werden die Regeln des impliziten Regelsatzes, die im Kapitel 4.2.2 besprochen wurden, gezeigt. Die Aufgabe einer Regel oder einer Gruppe von Regeln wird jeweils kurz beschrieben. Bevor die eigentlichen Regeln folgen, werden Werte an globale Bezeichner gebunden, die für die Vergabe der Prioritäten verwendet werden.

```
(defglobal MAIN
; Werden neue Objekte angelegt, so muß diesen eine virtuelle id
; zugeordnet werden, damit ein Objekt im weitern Veralauf der
; Optimierung eindeutig identifizierbar ist.
  ?*virtual_id* = 0

; Den Regeln werden verschiedene Prioritäten zugeordent, da die Regeln
; des impliziten Regelsatzes in einer ganz bestimmten Reihenfolge
; greifen müssen.
  ?*permission_set_salience* = -6000
  ?*permission_save_salience* = 9000
  ?*permission_new_salience* = 8990

  ?*remove_update_salience* = 6010
  ?*remove_object_salience* = 6000
  ?*remove_edge_salience* = 7600
  ?*remove_trash_salience* = 5900

  ?*set_rename_salience* = 7520
  ?*remove_rename_salience* = 7515
  ?*set_object_salience* = 7510
  ?*set_edge_salience* = 7500
  ?*set_set_salience* = 7500

  ?*update_salience* = 7030
  ?*star_salience* = 7020
  ?*circle_salience* = 7010
)
```

```
; ---------------------------------------------------------------------
; Es folgen zwei Regeln, mit denen der permission-Stack verwaltet
; wird.

; Die set_permission-Regel sorgt dafür, daß das oberste Element vom
; permission-Stack entfernt wird. Voraussetzung ist, daß keine
; feuerbereite Regel mehr vorhanden war und der Stack nicht leer
; ist. Die Proirität der Regel ist daher sehr niedrig gewählt.

(defrule MAIN::set_permission
  (declare (salience ?*permission_set_salience*))
  ?queue <- (permission_queue ?number $?rest)
  ?permission <- (permission ?number $?rules)
  ?active <- (permission active $?)
  =>
  (retract ?queue ?permission ?active)
  (assert (permission_queue $?rest))

  (assert (permission active $?rules))
  )

; Mit der save_permission-Regel werden 2 verschiedene Aufgaben
; erledigt.
; 1. Wenn die aktuelle Permission gesichert werden soll, so muß
;    ueberprüft werden, ob ein Unterschied zu der neu zu setzenden
;    Permission besteht.
;    Ist dies nicht der Fall, so kann das Abspeichern entfallen.
; 2. Sind die neu zu speichernde Permission und die aktive Permission
;    identisch, so müssen die entsprechenden Fakten gelöscht werden,
;    da sie nutzlos sind.

(defrule MAIN::save_permission
  (declare (salience ?*permission_save_salience*))
  ?save <- (permission save)
  ?new <- (permission active_new $?new_rules)
  ?permission_queue <- (permission_queue $?queue)
  ?permission <- (permission active $?active_rules)
  =>
  (retract ?save ?new)
  (if
      ; Abfrage A!=B <=> A\B != leere Menge oder B\A != leere Menge
      (or (not (empty$ (difference$ $?new_rules $?active_rules)))
          (not (empty$ (difference$ $?active_rules $?new_rules))))
      then
    (if (> (card$ $?queue) 0)
        then
      (bind ?number (nth$ 1 $?queue))
```

```
      (bind $?rest (rest$ $?queue))
      (retract ?permission_queue ?permission)
      (assert (permission_queue (- ?number 1) ?number $?rest))
      (assert (permission (- ?number 1) $?active_rules))
      (assert (permission active $?new_rules))
    else
    (if (= (card$ $?queue) 0)
        then
      (retract ?permission_queue ?permission)
      (assert (permission_queue 1))
      (assert (permission 1 $?active_rules))
      (assert (permission active $?new_rules))
      )
    )
  )
  )

; Wenn die alte Permission nicht gespeichert werden soll, so greift
; die folgenden Regel, die nur eine neue Permission setzt.

(defrule MAIN::new_permission
  (declare (salience ?*permission_new_salience*))
  ?old <- (permission active $?rules)
  ?new <- (permission active_new $?new_rules)
  =>
  (retract ?old ?new)
  (assert (permission active $?new_rules))
  )

; ---------------------------------------------------------------------
; Es folgen drei Regeln, die für das Aktualisieren der Menge der
; Eingangs- bzw. Ausgangs-  Stellen bzw. Transitionen benötigt
; Zusätzlich werden die Attribute in und out angepaßt. Die Mengen
; müssen sowohl beim Anlegen als auch beim Löschen von Objekten
; angepaßt werden.

(defrule MAIN::update_place_set
  (declare (salience ?*star_salience*))
  ?used <- (update (id ?update_id)(new_id ?new_id)
                   (info ?info&set_left_add|set_right_add|
                         set_left_remove|set_right_remove))
  ?place <- (place (id ?update_id)(star_set $?star_set)
                   (set_star $?set_star)(in ?in)(out ?out))
  (transition (id ?new_id))
  =>
  (retract ?used)
  (if (eq set_left_add ?info) then
```

```
    (if (not (in$ $?star_set ?new_id)) then
      (modify ?place
              (star_set $?star_set ?new_id)(in (+ ?in 1))))

    else (if (eq set_right_add ?info) then
           (if (not (in$ $?set_star ?new_id)) then
             (modify ?place (set_star $?set_star ?new_id)
                            (out (+ ?out 1))))

           else (if (eq set_left_remove ?info) then
                  (if (in$ $?star_set ?new_id) then
                    (modify ?place (star_set
                 (difference$ $?star_set ?new_id))(in (- ?in 1))))

                  else (if (eq set_right_remove ?info) then
                         (if (in$ $?set_star ?new_id) then
                           (modify ?place (set_star
                            (difference$ $?set_star ?new_id))
                            (out (- ?out 1))))
                         )
                  )
           )
    )
  )

(defrule MAIN::update_transition_set
  (declare (salience ?*star_salience*))
  ?used <- (update (id ?update_id)(new_id ?new_id)
                   (info ?info&set_left_add|set_right_add|
                         set_left_remove|set_right_remove))
  ?transition <- (transition (id ?update_id)(star_set $?star_set)
                             (in ?in)(out ?out)
                             (set_star $?set_star))
  (place (id ?new_id))
  =>
  (retract ?used)
  (if (eq set_left_add ?info) then
    (if (not (in$ $?star_set ?new_id)) then
      (modify ?transition
              (star_set $?star_set ?new_id)(in (+ ?in 1))))

    else (if (eq set_right_add ?info) then
           (if (not (in$ $?set_star ?new_id)) then
             (modify ?transition (set_star $?set_star ?new_id)
                                 (out (+ ?out 1))))

           else (if (eq set_left_remove ?info) then
```

```
                (if (in$ $?star_set ?new_id) then
                  (modify ?transition (star_set
              (difference$ $?star_set ?new_id))(in (- ?in 1))))

                else (if (eq set_right_remove ?info) then
                       (if (in$ $?set_star ?new_id) then
                         (modify ?transition (set_star
                          (difference$ $?set_star ?new_id))
                          (out (- ?out 1))))
                       )
                )
          )
    )
  )

; Um das Aktualisieren der Attribute zu ermöglichen, muß zusätzliches
; Wissen erzeugt werden, das anschließend wieder gelöscht werden muß.

(defrule MAIN::update_clean_up
  (declare (salience ?*update_salience*))
  ?trash <- (update (id ?update_id))
  (not (place (id ?update_id)))
  (not (transition (id ?update_id)))
  =>
  (retract ?trash)
  )

; ---------------------------------------------------------------------
; Es werden Regeln benötigt, um die Attribute der Objkte zu
; setzen. Zusätzlich kommen noch Regeln hinzu, die für das Umbenennen
; von Annotationen benötigt werden.

; Neue Stelle anlegen bzw. Attribute modifizieren.

(defrule MAIN::set_place
  (declare (salience ?*set_object_salience*))
  ?place <- (place (id ?id)(in ?in)(out ?out)(info set|nil))
  ?p_set <- (set (name P)(object $?p))
  (ident (name ?ident)(id ?id))
  ?used <- (place (name ?ident)
                  (info ?info&lower_bound|upper_bound|predicate|mark)
                  (lower_bound ?new_lower_bound)
                  (upper_bound ?new_upper_bound)
                  (predicate $?predicate)
                  (mark $?mark))
  =>
  (retract ?used)
```

```
  (if (not (in$ $?p ?id)) then
    (modify ?p_set (object $?p ?id)))

  (if (eq ?info lower_bound) then
    (modify ?place (lower_bound ?new_lower_bound)(info set)))
  else (if (eq ?info upper_bound) then
    (modify ?place (upper_bound ?new_upper_bound)(info set)))
  else (if (eq ?info predicate) then
    (modify ?place (predicate $?predicate)(info set)))
  else (if (eq ?info mark) then
    (modify ?place (mark $?mark)(info set)))
  )

; Bezeichner einer Transition umbenennen (condition)

(defrule MAIN::set_rename_transition_condition
  (declare (salience ?*set_rename_salience*))
  ?transition <- (transition (id ?id)(condition
                     $?left ?old_ident $?right)(info set|nil))
  (ident (name ?ident)(id ?id))
  (transition (name ?ident)(info condition))
  (rename (name ?ident)(info condition)(old_ident ?old_ident)
                                       (new_ident ?new_ident))
  =>
  (modify ?transition (condition $?left ?new_ident $?right)(info set))
  )

; Bezeichner einer Transition umbenennen (preaction)

(defrule MAIN::set_rename_transition_preaction
  (declare (salience ?*set_rename_salience*))
  ?transition <- (transition (id ?id)(preaction
                     $?left ?old_ident $?right)(info set|nil))
  (ident (name ?ident)(id ?id))
  (transition (name ?ident)(info preaction))
  (rename (name ?ident)(info preaction)(old_ident ?old_ident)
                                       (new_ident ?new_ident))
  =>
  (modify ?transition (preaction $?left ?new_ident $?right)(info set))
  )

; Bezeichner einer Transition umbenennen (postaction)

(defrule MAIN::set_rename_transition_postaction
  (declare (salience ?*set_rename_salience*))
  ?transition <- (transition (id ?id)(postaction
```

```
                    $?left ?old_ident $?right)(info set|nil))
  (ident (name ?ident)(id ?id))
  (transition (name ?ident)(info postaction))
  (rename (name ?ident)(old_ident ?old_ident)(new_ident ?new_ident))
  =>
  (modify ?transition (postaction $?left ?new_ident $?right)(info set))
  )

; Ueberflüßige Fakten nach dem rename löschen (transition)

(defrule MAIN::remove_rename_trash_transition
  (declare (salience ?*remove_rename_salience*))
  ?transition <- (transition (name ?ident)
                    (info ?info&condition|preaction|postaction))
  ?rename <- (rename (name ?ident)(info ?info))
  =>
  (retract ?transition ?rename)
  )

; Neue Transition anlegen bzw. Attribute modifizieren

(defrule MAIN::set_transition_id
  (declare (salience ?*set_object_salience*))
  ?transition <- (transition (id ?id)(info set|nil))
  ?t_set <- (set (name T)(object $?t))
  (ident (name ?ident)(id ?id))
  ?used <- (transition (name ?ident)
                       (info ?info&enable_delay|firing_delay|
                                 delay_semantic|marking_mode|
                                 demarking_mode|condition|
                                      preaction|postaction)
                       (enable_delay_min ?enable_delay_min)
                       (enable_delay_avg ?enable_delay_avg)
                       (enable_delay_max ?enable_delay_max)
                       (firing_delay_min ?firing_delay_min)
                       (firing_delay_avg ?firing_delay_avg)
                       (firing_delay_max ?firing_delay_max)
                       (delay_semantic $?delay_semantic)
                       (marking_mode $?marking_mode)
                       (demarking_mode $?demarking_mode)
                       (condition $?condition)
                       (preaction $?preaction)
                       (postaction $?postaction))
  =>
  (retract ?used)

  (if (not (in$ $?t ?id)) then
```

```
    (modify ?t_set (object $?t ?id)))

  (if (eq ?info enable_delay) then
    (modify ?transition (enable_delay_min ?enable_delay_min)
                        (enable_delay_avg ?enable_delay_avg)
            (enable_delay_max ?enable_delay_max)(info set)))
  else (if (eq ?info firing_delay) then
    (modify ?transition (firing_delay_min ?firing_delay_min)
                        (firing_delay_avg ?firing_delay_avg)
            (firing_delay_max ?firing_delay_max)(info set)))
  else (if (eq ?info delay_semantic) then
    (modify ?transition (delay_semantic $?delay_semantic)(info set)))
  else (if (eq ?info marking_mode) then
    (modify ?transition (marking_mode $?marking_mode)(info set)))
  else (if (eq ?info demarking_mode) then
    (modify ?transition (demarking_mode $?demarking_mode)(info set)))
  else (if (eq ?info condition) then
    (modify ?transition (condition $?condition)(info set)))
  else (if (eq ?info preaction) then
    (modify ?transition (preaction $?preaction)(info set)))
  else (if (eq ?info postaction) then
    (modify ?transition (postaction $?postaction)(info set)))
  )

; Bezeichner einer Kante umbenennen (annotation)

(defrule MAIN::set_rename_edge_annotation
  (declare (salience ?*set_rename_salience*))
  ?edge <- (edge (id ?id)(annotation $?left ?old_ident $?right)
                         (info set|nil))
  (ident (name ?ident)(id ?id))
  (edge (name ?ident)(info annotation))
  (rename (name ?ident)(info annotation)(old_ident ?old_ident)
                                        (new_ident ?new_ident))
  =>
  (modify ?edge (annotation $?left ?new_ident $?right)(info set))
  )

; Ueberflüßige Fakten nach dem rename löschen (edge)

(defrule MAIN::remove_rename_trash_edge
  (declare (salience ?*remove_rename_salience*))
  ?edge <- (edge (name ?ident)(info annotation))
  ?rename <- (rename (name ?ident)(info annotation))
  =>
  (retract ?edge ?rename)
  )
```

```
; Annotation einer Kante neu setzen

(defrule MAIN::modify_edge
  (declare (salience ?*set_object_salience*))
  ?edge <- (edge (id ?id)(info set|nil))
  (ident (name ?ident)(id ?id))
  ?used <- (edge (name ?ident)
                 (info annotation)
                 (annotation $?annotation))
  =>
  (retract ?used)
  (modify ?edge (annotation $?annotation)(info set))
  )

; ---------------------------------------------------------------------
; Es werden neun Regeln zum Anlegen von Kanten benötigt, da es die
; Transformationssprache erlaubt, daß innerhalb des selben
; Aktionsblocks sowohl Stellen und Transitionen als auch Kanten
; zwischen diesen Stellen und Transitionen angelegt werden dürfen. Die
; Kanten können nicht direkt angelegt werden, da die id's nicht
; vorhanden sind. Dies bedeutet, daß zu jedem Bezeichner erst die id
; ermittelt werden muß. Da bei einer or-Verknüpfung keine Varibalen
; gebunden werden können, müssen alle Fälle durch einzelne Regeln
; abgedeckt werden.
; (wäre bei or auch der Fall (Clips-intern))
; Da zum Anlegen der Kanten auch Mengen als Start- oder Zielknoten
; angegeben werden können, ergeben sich die Kombinationen
; (p,p)(p,t)(p,s)(t,t)(t,p)(t,s)(s,s)(s,t)(s,p), die abgedeckt werden
; müssen.
; Da alle Regeln nahezu identisch sind, ist nur die Regel
; set_edge_place_transition komplett angegeben worden.

; (p,p)
(defrule MAIN::set_edge_place_place
  ...
  )

; (p,t)
(defrule MAIN::set_edge_place_transition
  (declare (salience ?*set_edge_salience*))
  ?edge <- (set_edge (from ?from_ident)(from_info ?from_info)
                     (to ?to_ident)(to_info ?to_info)
                     (annotation $?annotation))
  (ident (name ?from_ident)(id ?from_id))
  (ident (name ?to_ident)(id ?to_id))
  (place (id ?from_id)
```

```
        (star_set $?from_star_set)(set_star $?from_set_star))
  (transition (id ?to_id)
              (star_set $?to_star_set)(set_star $?to_set_star))
  =>
  (retract ?edge)
  (if (eq object ?from_info) then
    (bind $?from_set ?from_id))
  else (if (eq star_set ?from_info) then
         (bind $?from_set $?from_star_set))
  else (if (eq set_star ?from_info) then
         (bind $?from_set $?from_set_star))

  (if (eq object ?to_info) then
    (bind $?to_set ?to_id))
  else (if (eq star_set ?to_info) then
         (bind $?to_set $?to_star_set))
  else (if (eq set_star ?to_info) then
         (bind $?to_set $?to_set_star))

  (add_edge$ $?from_set $?to_set $?annotation)
  )

; (p,s)
(defrule MAIN::set_edge_place_set
  ...
  )

; (t,t)
(defrule MAIN::set_edge_transition_transition
  ...
  )

; (t,p)
(defrule MAIN::set_edge_transition_place
  ...
  )

; (t,s)
(defrule MAIN::set_edge_transition_set
  ...
  )

; (s,s)
(defrule MAIN::set_edge_set_set
  ...
  )
```

```
; (s,t)
(defrule MAIN::set_edge_set_transition
  ...
  )

; (s,p)
(defrule MAIN::set_edge_set_place
  ...
  )

; Werden Kanten neu angelegt, so muß ihnen nachträglich ein id
; zugeornet werden, damit sie bei modify-Aktionen verwendet werden
; können.

(defrule MAIN::set_edge_virt_id
  (declare (salience ?*set_edge_salience*))
  ?edge <- (edge (id nil))
  =>
  (modify ?edge
          (id (sym-cat idxx ?*virtual_id*))
          (name (sym-cat idxx ?*virtual_id*)))
  (bind ?*virtual_id* (+ ?*virtual_id* 1))
)

; Die ident-Fakten können auch wieder gelöscht werden, da sie nach
; dem Beenden einer Regel nicht mehr gebraucht werden.

(defrule MAIN::remove_ident
  (declare (salience ?*remove_trash_salience*))
  ?ident <- (ident)
  =>
  (retract ?ident)
  )

; ---------------------------------------------------------------------
; Die folgende Regel verwaltet die Objekte einer Menge.
; Die Vorarbeit wird im action-Block geleistet, da dort schon
; überprüft wird, ob die Menge bereits existiert. Wenn dies der Fall
; ist, so wird kein Fakt mit (info set) erzeugt, so daß auch kein
; clean-up erforderlich wird. Das Verfahren ist also ähnlich wie bei
; den Stellen und Transitionen.

(defrule MAIN::set_object
  (declare (salience ?*set_set_salience*))
  ?used <- (set (name ?name)(object $?object)(info object))
  ?set <- (set (name ?name)(info set|nil))
  =>
```

```
  (retract ?used)
  (modify ?set (object $?object))
  )

; ---------------------------------------------------------------------
; remove

; remove ist so aufgebaut, daß die Fakten sofort aus dem Faktensatz
; gelöscht werden, wenn im info-Feld remove steht (Faktensatz klein).
; Das hat zur Folge, daß beim Einlesen der Fakten aus dem Editor
; Listen für die Kanten, Stellen und Transitionen geführt werden
; müssen, um beim Auslesen der Fakten aus CLIPS entscheiden zu können,
; welche Fakten gelöscht werden müssen.

; Mit der Regel remove_ids werden Mengen gelöscht, die bei den
; Transformationsfunktionen entstehen können. Es müssen jeweils
; separate Fakten erzeugt werden, die dann von remove_place oder
; remove_transition gelöscht werden. Ist die ids-Menge leer, so kann
; der Fakt gelöscht werden, da er abgearbeitet ist.
; Es muß darauf geachtet werden, daß zuerst die Menge der Eingangs-
; Stellen bzw. Transitionen und die Menge der Ausgangs- Stellen
; bzw. Transitionen aktualisiert werden. Dies wird durch die letzten
; beiden Fakten eingeleitet. Handelt es sich nicht um Mengen, so
; werden die benötigten Fakten zum updaten gesetzt.

(defrule MAIN::remove_ids
  (declare (salience ?*remove_update_salience*))
  ?remove <- (remove (info ids)(ids $?ids))
  =>
  (if (empty$ $?ids)
      then (retract ?remove)
      else
      (bind ?first (nth$ 1 $?ids))
      (bind $?rest (rest$ $?ids))
      (modify ?remove (ids $?rest))

      (assert (remove (name no_name)(id ?first)))

      (assert (edge (from ?first)(to no_name)(info remove)))
      (assert (edge (from no_name)(to ?first)(info remove)))
      )
  )

; Mengen können ohne jegliche Nebeneffekte gelöscht werden.

(defrule MAIN::remove_set
  (declare (salience ?*remove_object_salience*))
```

```
  ?used <- (remove (name ?name)(id no_id)(info set))
  ?set <- (set (name ?name))
  =>
  (retract ?used ?set)
  )

; Wird eine Kante über ihren Bezeichner gelöscht, so ist dies
; problemlos möglich.

(defrule MAIN::remove_edge_ident
 (declare (salience ?*remove_object_salience*))
 ?used <- (remove (name no_name)(id ?id))
 ?edge <- (edge (id ?id)(from ?from_id)(to ?to_id))
 =>
 (retract ?used ?edge)
 (assert (update (id ?from_id)(info set_right_remove)(new_id ?to_id)))
 (assert (update (id ?to_id)(info set_left_remove)(new_id ?from_id)))
 )

; Löschen von Stellen als Fakt und aus P und gegebenenfalls aus Pi, Po
; und Pio

(defrule MAIN::remove_place
  (declare (salience ?*remove_object_salience*))
  ?used <- (remove (name no_name)(id ?id))
  ?place <- (place (id ?id))
  ?set <- (set (name P)(object $?object))
  ?pi <- (set (name Pi)(object $?Pi))
  ?po <- (set (name Po)(object $?Po))
  ?pio <- (set (name Pio)(object $?Pio))
  =>
  (retract ?used ?place)
  (modify ?set (object (difference$ $?object ?id)))
  (if (in$ $?Pi ?id)
      then
    (modify ?pi (object (difference$ $?Pi ?id))))
  (if (in$ $?Po ?id)
      then
    (modify ?po (object (difference$ $?Po ?id))))
  (if (in$ $?Pio ?id)
      then
    (modify ?pio (object (difference$ $?Pio ?id))))
  )

; Löschen von Transitionen als Fakt und aus T und gegebenenfalls
; aus Pi, Po und Pio
```

```
(defrule MAIN::remove_transition
  (declare (salience ?*remove_object_salience*))
  ?used <- (remove (name no_name)(id ?id))
  ?transition <- (transition (id ?id))
  ?set <- (set (name T)(object $?object))
  ?pi <- (set (name Pi)(object $?Pi))
  ?po <- (set (name Po)(object $?Po))
  ?pio <- (set (name Pio)(object $?Pio))
  =>
  (retract ?used ?transition)
  (modify ?set (object (difference$ $?object ?id)))
  (if (in$ $?Pi ?id)
      then
    (modify ?pi (object (difference$ $?Pi ?id))))
  (if (in$ $?Po ?id)
      then
    (modify ?po (object (difference$ $?Po ?id))))
  (if (in$ $?Pio ?id)
      then
    (modify ?pio (object (difference$ $?Pio ?id))))
  )

; Die beiden folgenden Regeln löschen Kanten, deren Anfangs- oder
; Endpunkt gelöscht wurde, d.h. es ist eine Stelle oder Transition
; gelöscht worden. Es werden noch Fakten für das aktualisieren der
; Menge der Eingangs- (Ausgangs-) Stellen bzw. Transitionen
; gesetzt. Es werden aus technischen Gründen erst die Kanten gelöscht
; und dann die Stellen bzw. Transitionen. Sonst wäre das Aktualisieren
; der Menge der Eingangs- (Ausgangs-) Stellen bzw. Transitionen nicht
; mehr möglich.
; Die vorbereitenden Informationen werden von der Regel remove_ids
; gesetzt, falls es sich um Mengen handelt. Sonst werden die
; Informationen bereits in der Grammatik bei remove_ident gesetzt.

(defrule MAIN::remove_edge_no_begin
  (declare (salience ?*remove_update_salience*))
  (edge (from ?del_id)(to no_name))
  ?trash <- (edge (from ?del_id)(to ?update_id&~no_name))
  =>
  (retract ?trash)
  (assert (update (id ?update_id)(new_id ?del_id)
                                  (info set_left_remove)))
  )

(defrule MAIN::remove_edge_no_end
  (declare (salience ?*remove_update_salience*))
  (edge (from no_name)(to ?del_id))
```

```
  ?trash <- (edge (from ?update_id&~no_name)(to ?del_id))
  =>
  (retract ?trash)
  (assert (update (id ?update_id)(new_id ?del_id)
                                  (info set_right_remove)))
  )

; Die folgende Regel löscht Fakten, die für das Löschen überflüssiger
; Kanten zuständig waren und jetzt nicht mehr benötigt werden. Der
; Fakt kann nicht bei einer der vorherigen Regeln gelöscht werden, da
; nicht absehbar ist, wie viele Kanten existieren.
; Es müssen auch Kanten in Betracht gezogen werden, bei denen die
; Annotationen keine Übereinstimmung mit den vorhandenen Kanten
; ergeben haben.

(defrule MAIN::remove_edge_trash
  (declare (salience ?*remove_trash_salience*))
  ?trash <- (edge (info remove))
  =>
  (retract ?trash)
  )

; Ein Kante wird dann geloescht, wenn sie real existiert (info
; ~remove) und  ein entsprechender Fakt zum Löschen gesetzt wurde
; (info remove)
; Hinzu kommt, daß ueberprüft wird ob die Annotationen identisch
; sind. Dies ist der Fall, wenn
; 1. $?annotation_1 == empty; d.h. keine Angaben
; 2. $?annotation_1 == $?annotation_2
; Sonst können die Kanten nicht als identisch anerkannt werden.

(defrule MAIN::remove_edge
  (declare (salience ?*remove_edge_salience*))
  ?edge1 <- (edge (from ?from_id)(to ?to_id)
                  (annotation $?annotation_1)(info remove))
  ?edge2 <- (edge (from ?from_id)(to ?to_id)
                  (annotation $?annotation_2)(info ~remove))
  =>
; Es wurden keine Angaben zu den Annotationen gemacht
; oder Identsich sind z.B. [x],[y,z] und [y,z],[x]
  (if (or (empty$ $?annotation_1)
          (annoteq$ $?annotation_2 $?annotation_1))
      then (retract ?edge1 ?edge2)
      )
  )
```

Anhang D

Transformationen für die Anwendungsbeispiele

D.1 Senke und Quelle

```
(description optimize

 (permission x_3_2_5:a x_3_2_6:a
             x_3_2_7:a x_3_2_8:a x_3_2_8:b x_3_2_9:a
             x_3_2_10:a x_3_2_11:a x_3_2_12:a_b_vorbereitung)

 (create ?Sp ?St)

// Senke
 (transformation x_3_2_12

  (rule a_b_vorbereitung
   (condition
    (permission)
    (set ?Sp ?Po)
    (empty ?Sp)
    )
   (action
    (permission save x_3_2_12:a x_3_2_12:b
                     x_3_2_12:a_b_nachbereitung)
    (add (set ?Sp ?Po))
    )
   )

  (rule a
   (condition
    (permission)
    (set ?Sp ?St)
    (place ?p)
    (transition ?t)
    (edge ?e from ?t to ?p)
    (in ?Sp ?p)
    (not (in ?St ?t))
    )
   (action
    (add (set ?St ?St ?t))
    )
   )

  (rule b
   (condition
    (permission)
    (set ?Sp ?St)
    (place ?p)
    (transition ?t)
```

```
    (edge ?e from ?p to ?t)
    (in ?St ?t)
    (not (in ?Sp ?p))
    )
   (action
    (add (set ?Sp ?Sp ?p))
    )
   )

  (rule a_b_nachbereitung
   (priority -10)
   (condition
    (permission)
    (set ?P ?T)
    (set ?Sp ?St)
    )
   (action
    (remove (difference ?P ?Sp))
    (remove (difference ?T ?St))
    (remove ?Sp ?St)
    )
   )
  )

 (create ?Sp1 ?St1)

// Quelle
 (transformation x_3_2_13

  (rule a_b_vorbereitung
   (condition
    (permission)
    (set ?Sp1 ?Pi)
    (empty ?Sp1)
    )
   (action
    (permission save x_3_2_13:a x_3_2_13:b
                     x_3_2_13:a_b_nachbereitung)
    (add (set ?Sp1 ?Pi))
    )
   )

  (rule a
   (condition
    (permission)
    (set ?Sp1 ?St1)
    (place ?p)
```

```
    (transition ?t)
    (edge ?e from ?p to ?t)
    (in ?Sp1 ?p)
    (not (in ?St1 ?t))
    )
   (action
    (add (set ?St1 ?St1 ?t))
    )
   )

  (rule b
   (condition
    (permission)
    (set ?Sp1 ?St1)
    (place ?p)
    (transition ?t)
    (edge ?e from ?t to ?p)
    (in ?St1 ?t)
    (not (in ?Sp1 ?p))
    )
   (action
    (add (set ?Sp1 ?Sp1 ?p))
    )
   )

  (rule
   a_b_nachbereitung
   (priority -10)
   (condition
    (permission)
    (set ?P ?T)
    (set ?Sp1 ?St1)
    )
   (action
    (remove (difference ?P ?Sp1))
    (remove (difference ?T ?St1))
    (remove ?Sp1 ?St1)
    )
   )
  )
 )
```

D.2 Strukturbildoptimierung einer Differentialgleichung

```
(description dgln

(permission dgln_1:loop_integ_mult
            dgln_2:loop_integ_integ_rename
            dgln_3:integ_mult dgln_4:pos
            dgln_4:pos
            dgln_5:integ_integ_rename)


// Die erste Transformation transformiert eine Schleife, die aus
// einem Addierglied, einem Integrierglied, einer Verzweigung und
// einem Multiplizierglied besteht zu einem Integrierglied.
 (transformation dgln_1
  (rule loop_integ_mult
   (condition
    (permission)

// Addierglied
    (place ?p_x)
    (place ?p_y)
    (transition ?t_z
                preaction z=x-y;)
    (edge ?ea1 from ?p_x to ?t_z annotation 1[x])
    (edge ?ea2 from ?p_y to ?t_z annotation 1[y])
    (place ?p_z_out)
    (edge ?ea3 from ?t_z to ?p_z_out annotation 1[z])

    (transition ?t_z_out)
    (edge ?emo1 from ?p_z_out to ?t_z_out annotation 1[x])

// Integrierglied
    (place ?p_integ_in)
    (edge ?emo2 from ?t_z_out to ?p_integ_in annotation 1[x])

    (transition ?t_integ
                preaction ?t_integ_pre)
    (edge ?ei1 from ?p_integ_in to ?t_integ annotation 1[x])
    (place ?p_integ
           mark_number 1)
    (edge ?ei2 from ?t_integ to ?p_integ annotation 1[i_new])
    (edge ?ei3 from ?p_integ to ?t_integ annotation 1[i_old])
    (place ?p_integ_out)
    (edge ?ei4 from ?t_integ to ?p_integ_out annotation 1[i_new])

    (transition ?t_integ_out)
```

```
    (edge ?emo3 from ?p_integ_out to ?t_integ_out annotation 1[x])

// Verzweigung
    (place ?p_fork_in)
    (edge ?emo4 from ?t_integ_out to ?p_fork_in annotation 1[x])

    (transition ?t_fork)
    (edge ?ef1 from ?p_fork_in to ?t_fork annotation 1[x])
    (place ?p_fork_out_1)
    (place ?p_fork_out_2)
    (neq ?p_fork_out_1 ?p_fork_out_2)
    (edge ?ef2 from ?t_fork to ?p_fork_out_1 annotation 1[x])
    (edge ?ef3 from ?t_fork to ?p_fork_out_2 annotation 1[x])

    (transition ?t_fork_out)
    (edge ?emo5 from ?p_fork_out_2 to ?t_fork_out annotation 1[x])

// Multiplizierglied
    (place ?p_mult_in)
    (edge ?emo6 from ?t_fork_out to ?p_mult_in annotation 1[x])
    (transition ?t_mult
                preaction ?t_mult_pre)
    (edge ?em1 from ?p_mult_in to ?t_mult annotation 1[x])
    (place ?p_mult_out)
    (edge ?em2 from ?t_mult to ?p_mult_out annotation 1[y])

    (transition ?t_mult_out)
    (edge ?emo7 from ?p_mult_out to ?t_mult_out annotation 1[x])
    (edge ?emo8 from ?t_mult_out to ?p_y annotation 1[x])

// Die Verbindung der Module ist bereits integriert
    (transition ?t_modul_in)
    (edge ?emo9 from ?t_modul_in to ?p_x annotation 1[x])
    (transition ?t_modul_out)
    (edge ?emo10 from ?p_fork_out_1 to ?t_modul_out annotation 1[x])

    ) // end condition

   (action
    (remove ?p_x ?p_y ?t_z ?p_z_out ?t_z_out)
    (remove ?t_integ_out)
    (remove ?p_fork_in ?t_fork ?p_fork_out_1 ?p_fork_out_2 ?t_fork_out)
    (remove ?p_mult_in ?t_mult ?p_mult_out ?t_mult_out)
    (modify (transition ?t_integ
                        preaction [calculate
                                    i_new new=(1+i_new*y)/i_new;
                                    ?t_mult_pre ?t_integ_pre]))
```

```
// Die Schnittstelle
    (add (edge from ?t_modul_in to ?p_integ_in annotation 1[x]))
    (add (edge from ?p_integ_out to ?t_modul_out annotation 1[x]))

    ) // end action
   ) // end rule loop_integ_mult
  ) // end transformation dgln_1

// Bei der zweiten Transformation werden zwei in einer Schleife
// zusammengeschaltete Integrierglieder zu einem zusammengefaßt.
// Vorher müssen die Bezeichner umbenannt werden, damit eine korrekte
// Ersetzung möglich ist.
 (transformation dgln_2
  (rule loop_integ_integ_rename
   (condition
    (permission)

// Addierglied
    (place ?p_x)
    (place ?p_y)
    (transition ?t_z
                preaction z=x-y;)
    (edge ?ea1 from ?p_x to ?t_z annotation 1[x])
    (edge ?ea2 from ?p_y to ?t_z annotation 1[y])
    (place ?p_z_out)
    (edge ?ea3 from ?t_z to ?p_z_out annotation 1[z])

    (transition ?t_z_out)
    (edge ?emo1 from ?p_z_out to ?t_z_out annotation 1[x])

// Integrierglied
    (place ?p_integ_1_in)
    (edge ?emo2 from ?t_z_out to ?p_integ_1_in annotation 1[x])

    (transition ?t_integ_1
                preaction ?t_integ_1_pre)
    (edge ?ei11 from ?p_integ_1_in to ?t_integ_1 annotation 1[x])
    (place ?p_integ_1
           mark_number 1)
    (edge ?ei12 from ?t_integ_1 to ?p_integ_1 annotation 1[i_new])
    (edge ?ei13 from ?p_integ_1 to ?t_integ_1 annotation 1[i_old])
    (place ?p_integ_1_out)
    (edge ?ei14 from ?t_integ_1 to ?p_integ_1_out annotation 1[i_new])

    (transition ?t_integ_1_out)
    (edge ?emo3 from ?p_integ_1_out to ?t_integ_1_out annotation 1[x])
```

```
// Verzweigung
    (place ?p_fork_in)
    (edge ?emo4 from ?t_integ_1_out to ?p_fork_in annotation 1[x])

    (transition ?t_fork)
    (edge ?ef1 from ?p_fork_in to ?t_fork annotation 1[x])
    (place ?p_fork_out_1)
    (place ?p_fork_out_2)
    (neq ?p_fork_out_1 ?p_fork_out_2)
    (edge ?ef2 from ?t_fork to ?p_fork_out_1 annotation 1[x])
    (edge ?ef3 from ?t_fork to ?p_fork_out_2 annotation 1[x])

    (transition ?t_fork_out)
    (edge ?emo5 from ?p_fork_out_2 to ?t_fork_out annotation 1[x])

// Integrierglied
    (place ?p_integ_2_in)
    (edge ?emo6 from ?t_fork_out to ?p_integ_2_in annotation 1[x])
    (transition ?t_integ_2
                preaction ?t_integ_2_pre)
    (edge ?ei21 from ?p_integ_2_in to ?t_integ_2 annotation 1[x])
    (place ?p_integ_2_out)
    (edge ?ei22 from ?t_integ_2 to ?p_integ_2_out annotation 1[i_new])
    (place ?p_integ_2
           mark_number 1)
    (edge ?ei23 from ?t_integ_2 to ?p_integ_2 annotation 1[i_new])
    (edge ?ei24 from ?p_integ_2 to ?t_integ_2 annotation 1[i_old])

    (transition ?t_integ_2_out)
    (edge ?emo7 from ?p_integ_2_out to ?t_integ_2_out annotation 1[x])
    (edge ?emo8 from ?t_integ_2_out to ?p_y annotation 1[x])

// Die Verbindung der Module ist bereits integriert
    (transition ?t_modul_in)
    (edge ?emo9 from ?t_modul_in to ?p_x annotation 1[x])
    (transition ?t_modul_out)
    (edge ?emo10 from ?p_fork_out_1 to ?t_modul_out annotation 1[x])

    ) // end condition

   (action
    (modify (transition ?t_integ_1
                        preaction [rename i_new i_new_1]))
    (modify (transition ?t_integ_2
                        preaction [rename i_new i_new_2]))
    (permission save dgln_2:loop_integ_integ)
    ) // end action
```

```
   ) // end rule loop_integ_integ_rename

  (rule
   loop_integ_integ
   (priority 10)
   (condition
    (permission)

// Addierglied
    (place ?p_x)
    (place ?p_y)
    (transition ?t_z
                preaction z=x-y;)
    (edge ?ea1 from ?p_x to ?t_z annotation 1[x])
    (edge ?ea2 from ?p_y to ?t_z annotation 1[y])
    (place ?p_z_out)
    (edge ?ea3 from ?t_z to ?p_z_out annotation 1[z])

    (transition ?t_z_out)
    (edge ?emo1 from ?p_z_out to ?t_z_out annotation 1[x])

// Integrierglied 1
    (place ?p_integ_1_in)
    (edge ?emo2 from ?t_z_out to ?p_integ_1_in annotation 1[x])

    (transition ?t_integ_1
                preaction ?t_integ_1_pre)
    (edge ?ei11 from ?p_integ_1_in to ?t_integ_1 annotation 1[x])
    (place ?p_integ_1
           mark_number 1)
    (edge ?ei12 from ?t_integ_1 to ?p_integ_1 annotation 1[i_new])
    (edge ?ei13 from ?p_integ_1 to ?t_integ_1 annotation 1[i_old])
    (place ?p_integ_1_out)
    (edge ?ei14 from ?t_integ_1 to ?p_integ_1_out annotation 1[i_new])

    (transition ?t_integ_1_out)
    (edge ?emo3 from ?p_integ_1_out to ?t_integ_1_out annotation 1[x])

// Verzweigung
    (place ?p_fork_in)
    (edge ?emo4 from ?t_integ_1_out to ?p_fork_in annotation 1[x])

    (transition ?t_fork)
    (edge ?ef1 from ?p_fork_in to ?t_fork annotation 1[x])
    (place ?p_fork_out_1)
    (place ?p_fork_out_2)
    (neq ?p_fork_out_1 ?p_fork_out_2)
```

```
    (edge ?ef2 from ?t_fork to ?p_fork_out_1 annotation 1[x])
    (edge ?ef3 from ?t_fork to ?p_fork_out_2 annotation 1[x])

    (transition ?t_fork_out)
    (edge ?emo5 from ?p_fork_out_2 to ?t_fork_out annotation 1[x])

// Integrierglied 2
    (place ?p_integ_2_in)
    (edge ?emo6 from ?t_fork_out to ?p_integ_2_in annotation 1[x])
    (transition ?t_integ_2
                preaction ?t_integ_2_pre)
    (edge ?ei21 from ?p_integ_2_in to ?t_integ_2 annotation 1[x])
    (place ?p_integ_2_out)
    (edge ?ei22 from ?t_integ_2 to ?p_integ_2_out annotation 1[i_new])
    (place ?p_integ_2
           mark_number 1)
    (edge ?ei23 from ?t_integ_2 to ?p_integ_2 annotation 1[i_new])
    (edge ?ei24 from ?p_integ_2 to ?t_integ_2 annotation 1[i_old])

    (transition ?t_integ_2_out)
    (edge ?emo7 from ?p_integ_2_out to ?t_integ_2_out annotation 1[x])
    (edge ?emo8 from ?t_integ_2_out to ?p_y annotation 1[x])

// Die Verbindung der Module ist bereits integriert
    (transition ?t_modul_in)
    (edge ?emo9 from ?t_modul_in to ?p_x annotation 1[x])
    (transition ?t_modul_out)
    (edge ?emo10 from ?p_fork_out_1 to ?t_modul_out annotation 1[x])

    ) // end condition

   (action
    (remove ?p_x ?p_y ?t_z ?p_z_out ?t_z_out)
    (remove ?t_integ_1_out)
    (remove ?p_fork_in ?t_fork ?p_fork_out_1 ?p_fork_out_2
                                             ?t_fork_out)
    (remove ?p_integ_2_in ?t_integ_2 ?p_integ_2 ?p_integ_2_out
                                                ?t_integ_2_out)
    (modify (transition ?t_integ_1
                        preaction [calculate
                                i_new new=(1+i_new_1*i_new_2)/i_new_1;
                                ?t_integ_2_pre ?t_integ_1_pre]))
// Die Schnittstelle
    (add (edge from ?t_modul_in to ?p_integ_1_in annotation 1[x]))
    (add (edge from ?p_integ_1_out to ?t_modul_out annotation 1[x]))

    ) // end action
```

```
   ) // end rule loop_integ_integ
  ) // end transformation dgln_2

// Die dritte Transformation wandelt eine Kette, bestehend aus einem
// Integrierglied und einem Multiplizierglied in ein Integrier-
// glied um.
 (transformation dgln_3
  (rule integ_mult
   (condition
    (permission)

// Integrierglied
    (transition ?t_modul_in)
    (place ?p_integ_in)
    (edge ?emo1 from ?t_modul_in to ?p_integ_in annotation 1[x])
    (transition ?t_integ
                preaction ?t_integ_pre)
    (edge ?ei1 from ?p_integ_in to ?t_integ annotation 1[x])
    (place ?p_integ)
    (edge ?ei2 from ?t_integ to ?p_integ annotation 1[i_new])
    (edge ?ei3 from ?p_integ to ?t_integ annotation 1[i_old])
    (place ?p_integ_out)
    (edge ?emo2 from ?t_integ to ?p_integ_out annotation 1[i_new])

// Multiplizierer
    (transition ?t_mult_in)
    (edge ?emo3 from ?p_integ_out to ?t_mult_in annotation 1[x])
    (place ?p_mult_in)
    (edge ?emo4 from ?t_mult_in to ?p_mult_in annotation 1[x])
    (transition ?t_mult
                preaction ?t_mult_pre)
    (edge ?em1 from ?p_mult_in to ?t_mult annotation 1[x])
    (place ?p_mult_out)
    (edge ?em2 from ?t_mult to ?p_mult_out annotation 1[y])

    (transition ?t_modul_out)
    (edge ?emo5 from ?p_mult_out to ?t_modul_out annotation 1[x])

    ) // end condition

   (action
    (remove ?t_mult_in ?p_mult_in ?t_mult ?p_mult_out)
    (modify (transition ?t_integ
                        preaction [calculate i_new new=i_new*y;
                                   ?t_mult_pre ?t_integ_pre]))
    (add (edge from ?p_integ_out to ?t_modul_out annotation 1[x]))
    ) // end action
```

```
   ) // end rule integ_mult
  ) // end transformation dgln_3

// Bei der vierten Transformation wird ein Zweig umgehängt,
// damit wieder eine der anderen Transformationen greifen kann.
 (transformation dgln_4
  (rule pos
   (condition
    (permission)

// Integrierglied 1
    (place ?p_integ_1_in)
    (transition ?t_integ_1)
    (edge ?ei11 from ?p_integ_1_in to ?t_integ_1 annotation 1[x])
    (place ?p_integ_1_out)
    (edge ?ei12 from ?t_integ_1 to ?p_integ_1_out annotation 1[i_new])
    (place ?p_integ_1)
    (edge ?ei13 from ?t_integ_1 to ?p_integ_1 annotation 1[i_new])
    (edge ?ei14 from ?p_integ_1 to ?t_integ_1 annotation 1[i_old])

// Verzweigung 1
    (transition ?t_fork_1_in)
    (edge ?ef11 from ?p_integ_1_out to ?t_fork_1_in annotation 1[x])
    (place ?p_fork_1_in)
    (edge ?ef12 from ?t_fork_1_in to ?p_fork_1_in annotation 1[x])
    (transition ?t_fork_1)
    (edge ?ef13 from ?p_fork_1_in to ?t_fork_1 annotation 1[x])
    (place ?p_fork_11_out)
    (edge ?ef14 from ?t_fork_1 to ?p_fork_11_out annotation 1[x])
    (place ?p_fork_12_out)
    (edge ?ef15 from ?t_fork_1 to ?p_fork_12_out annotation 1[x])
    (neq ?p_fork_11_out ?p_fork_12_out)
    (transition ?t_fork_11_out)
    (edge ?ef16 from ?p_fork_11_out to ?t_fork_11_out annotation 1[x])
    (transition ?t_fork_12_out)
    (edge ?ef17 from ?p_fork_12_out to ?t_fork_12_out annotation 1[x])
    (neq ?t_fork_11_out ?t_fork_12_out)

// Integrierglied 2
    (place ?p_integ_2_in)
    (edge ?ei20 from ?t_fork_11_out to ?p_integ_2_in annotation 1[x])
    (transition ?t_integ_2
                preaction ?t_integ_2_pre)
    (edge ?ei21 from ?p_integ_2_in to ?t_integ_2 annotation 1[x])
    (place ?p_integ_2_out)
    (edge ?ei22 from ?t_integ_2 to ?p_integ_2_out annotation 1[i_new])
    (place ?p_integ_2)
```

```
    (edge ?ei23 from ?t_integ_2 to ?p_integ_2 annotation 1[i_new])
    (edge ?ei24 from ?p_integ_2 to ?t_integ_2 annotation 1[i_old])

// Multiplizierglied
    (place ?p_mult_in)
    (edge ?em0 from ?t_fork_12_out to ?p_mult_in annotation 1[x])
    (transition ?t_mult)
    (edge ?em1 from ?p_mult_in to ?t_mult annotation 1[x])
    (place ?p_mult_out)
    (edge ?em2 from ?t_mult to ?p_mult_out annotation 1[y])

// Verzweigung 2
    (transition ?t_fork_2_in)
    (edge ?ef21 from ?p_integ_2_out to ?t_fork_2_in annotation 1[x])
    (place ?p_fork_2_in)
    (edge ?ef22 from ?t_fork_2_in to ?p_fork_2_in annotation 1[x])
    (transition ?t_fork_2)
    (edge ?ef23 from ?p_fork_2_in to ?t_fork_2 annotation 1[x])
    (place ?p_fork_21_out)
    (edge ?ef24 from ?t_fork_2 to ?p_fork_21_out annotation 1[x])
    (place ?p_fork_22_out)
    (edge ?ef25 from ?t_fork_2 to ?p_fork_22_out annotation 1[x])
    (neq ?p_fork_21_out ?p_fork_22_out)
    (transition ?t_fork_21_out)
    (edge ?ef26 from ?p_fork_21_out to ?t_fork_21_out annotation 1[x])
    (transition ?t_fork_22_out)
    (edge ?ef27 from ?p_fork_22_out to ?t_fork_22_out annotation 1[x])
    (neq ?t_fork_21_out ?t_fork_22_out)

// Verzweigung 3
    (place ?p_fork_3_in)
    (edge ?ef32 from ?t_fork_21_out to ?p_fork_3_in annotation 1[x])
    (transition ?t_fork_3)
    (edge ?ef33 from ?p_fork_3_in to ?t_fork_3 annotation 1[x])
    (place ?p_fork_31_out)
    (edge ?ef34 from ?t_fork_3 to ?p_fork_31_out annotation 1[x])
    (place ?p_fork_32_out)
    (edge ?ef35 from ?t_fork_3 to ?p_fork_32_out annotation 1[x])
    (neq ?p_fork_31_out ?p_fork_32_out)
    (transition ?t_fork_31_out)
    (edge ?ef36 from ?p_fork_31_out to ?t_fork_31_out annotation 1[x])
    (transition ?t_fork_32_out)
    (edge ?ef37 from ?p_fork_32_out to ?t_fork_32_out annotation 1[x])
    (neq ?t_fork_31_out ?t_fork_32_out)

    ) // end condition
```

```
   (action
    (remove (edge from ?p_integ_1_out to ?t_fork_1_in))
    (remove (edge from ?t_fork_11_out to ?p_integ_2_in))
    (add (transition ?t_new))
    (add (edge from ?p_integ_1_out to ?t_new annotation 1[x]))
    (add (edge from ?t_new to ?p_integ_2_in annotation 1[x]))
    (remove ?t_fork_21_out)
    (add (edge from ?p_fork_21_out to ?t_fork_1_in annotation 1[x]))
    (add (edge from ?t_fork_11_out to ?p_fork_3_in annotation 1[x]))
    (remove (edge from ?t_fork_12_out to ?p_mult_in))

    (add (place ?p_integ_new_in))
    (add (edge from ?t_fork_12_out to ?p_integ_new_in annotation 1[x]))
    (add (transition ?t_integ_new
                    preaction [calculate
                                  i_new new=1/i_new; ?t_integ_2_pre]))
    (add (edge from ?p_integ_new_in to ?t_integ_new annotation 1[x]))
    (add (place ?p_integ_new_out))
    (add (edge from ?t_integ_new to ?p_integ_new_out
                                   annotation 1[i_new]))
    (add (place ?p_integ_new))
    (add (edge from ?t_integ_new to ?p_integ_new annotation 1[i_new]))
    (add (edge from ?p_integ_new to ?t_integ_new annotation 1[i_old]))
    (add (transition ?t_integ_new_out))
    (add (edge from ?p_integ_new_out to ?t_integ_new_out
                                        annotation 1[x]))
    (add (edge from ?t_integ_new_out to ?p_mult_in annotation 1[x]))

    ) // end action
   ) // end rule pos
  ) // end transformation dgln_4

// Zwei hintereinandergeschaltete Integrierglieder werden zu einem
// zusammengefaßt.
// Die Bezeichner müssen vorher umbenannt werden, damit eine
// eindeutige Ersetzung möglich ist.
 (transformation dgln_5
  (rule integ_integ_rename

   (condition
    (permission)

// Integierglied 1
    (transition ?t_integ_1_in)
    (place ?p_integ_1_in)
    (edge ?ei10 from ?t_integ_1_in to ?p_integ_1_in annotation 1[x])
    (transition ?t_integ_1)
```

```
    (edge ?ei11 from ?p_integ_1_in to ?t_integ_1 annotation 1[x])
    (place ?p_integ_1_out)
    (edge ?ei12 from ?t_integ_1 to ?p_integ_1_out annotation 1[i_new])
    (place ?p_integ_1)
    (edge ?ei13 from ?t_integ_1 to ?p_integ_1 annotation 1[i_new])
    (edge ?ei14 from ?p_integ_1 to ?t_integ_1 annotation 1[i_old])
    (transition ?t_integ_1_out)
    (edge ?ei15 from ?p_integ_1_out to ?t_integ_1_out annotation 1[x])

// Integrierglied 2
    (place ?p_integ_2_in)
    (edge ?ei21 from ?t_integ_1_out to ?p_integ_2_in annotation 1[x])
    (transition ?t_integ_2)
    (edge ?ei22 from ?p_integ_2_in to ?t_integ_2 annotation 1[x])
    (place ?p_integ_2_out)
    (edge ?ei23 from ?t_integ_2 to ?p_integ_2_out annotation 1[i_new])
    (place ?p_integ_2)
    (edge ?ei24 from ?t_integ_2 to ?p_integ_2 annotation 1[i_new])
    (edge ?ei25 from ?p_integ_2 to ?t_integ_2 annotation 1[i_old])

    ) // end condition

   (action
    (modify (transition ?t_integ_1
                        preaction [rename i_new i_new_1]))
    (modify (transition ?t_integ_2
                        preaction [rename i_new i_new_2]))
    (permission save dgln_5:integ_integ)
    ) // end_action
   ) // end rule integ_integ_rename

   (rule
    integ_integ

    (priority 10)
    (condition
     (permission)

// Integrierglied 1
    (transition ?t_integ_1_in)
    (place ?p_integ_1_in)
    (edge ?ei10 from ?t_integ_1_in to ?p_integ_1_in annotation 1[x])
    (transition ?t_integ_1
                preaction ?t_integ_1_pre)
    (edge ?ei11 from ?p_integ_1_in to ?t_integ_1 annotation 1[x])
    (place ?p_integ_1_out)
    (edge ?ei12 from ?t_integ_1 to ?p_integ_1_out annotation 1[i_new])
```

```
    (place ?p_integ_1)
    (edge ?ei13 from ?t_integ_1 to ?p_integ_1 annotation 1[i_new])
    (edge ?ei14 from ?p_integ_1 to ?t_integ_1 annotation 1[i_old])
    (transition ?t_integ_1_out)
    (edge ?ei15 from ?p_integ_1_out to ?t_integ_1_out annotation 1[x])

//Integrierglied 2
    (place ?p_integ_2_in)
    (edge ?ei21 from ?t_integ_1_out to ?p_integ_2_in annotation 1[x])
    (transition ?t_integ_2
                preaction ?t_integ_2_pre)
    (edge ?ei22 from ?p_integ_2_in to ?t_integ_2 annotation 1[x])
    (place ?p_integ_2_out)
    (edge ?ei23 from ?t_integ_2 to ?p_integ_2_out annotation 1[i_new])
    (place ?p_integ_2)
    (edge ?ei24 from ?t_integ_2 to ?p_integ_2 annotation 1[i_new])
    (edge ?ei25 from ?p_integ_2 to ?t_integ_2 annotation 1[i_old])

    ) // end condition

   (action
    (remove ?p_integ_1_in ?p_integ_1_out ?p_integ_1)
    (remove ?t_integ_1 ?t_integ_1_out)
    (add (edge from ?t_integ_1_in to ?p_integ_2_in annotation 1[x]))
    (modify (transition ?t_integ_2
                        preaction [calculate
                                      i_new new=i_new_1*i_new_2;
                                      ?t_integ_1_pre ?t_integ_2_pre]))

    ) // end_action
   ) // end rule integ_integ
  ) // end transformation dgln_5
 ) // end description
```