Technische Universität Darmstadt

Fachbereich Informatik

Fachgebiet Knowledge Engineering

Diplomarbeit

Ansätze eines TI-Coaching-Systems zur Unterstützung von Lernenden im Bereich der Theoretischen Informatik

Autor:

Christof Kuhn

Abgabe:

11. April 2006

Betreuer**:**

Dr. Gunter Grieser

Ehrenwörtliche Erklärung:

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst habe und dass ich keine anderen als die angegeben Quellen und Hilfsmittel benutzt habe oder die Arbeit bei einem anderen Prüfungsverfahren vorgelegt habe.

Darmstadt, im April 2006

_____

Christof Kuhn

Zusammenfassung

In dieser Arbeit, werden einige Ansätze zur Implementierung eines TI-Coaching-Systems vorgestellt.

Dabei handelt es sich um ein System, welches Studenten der Informatik erlaubt, bestimmte Aufgaben aus dem Bereich der theoretischen Informatik am Rechner zu üben. Die Idee von diesem System stammt von Dr. Gunter Grieser und beruht auf einem bereits existierenden vergleichbaren System für die Mathematik.

Das System soll dabei selbst in der Lage sein, von ihm gestellte Aufgaben zu verarbeiten und gleichzeitig die Lösungen von Studenten analysieren. Durch Vergleich der studentischen Lösung mit der eigenen soll es selbstständig Fehler erkennen und den Studenten bei einer korrekten Lösung der Aufgabe unterstützen.

Das System soll ein web-basiertes Frontend besitzen und ein Backend, welches den eigentlichen Kern des Systems darstellt. Als Grundlage der internen Verarbeitung von Aufgaben und Lösungen soll der logische Ansatz des automatischen Beweisens dienen.

Die vorgestellten Ansätze beinhalten Verfahren zur logikbasierten Lösung bestimmter Aufgabenstellungen sowie eine Analyse dieser Aufgaben bezüglich potentieller Fehler von Studenten. Außerdem wird eine spezielle Sprache zur Kommunikation zwischen Frontend und Systemkern vorgestellt: Die Lösungsbeschreibungssprache LBS.

Abschließend wird noch eine grundlegende Implementierung für den Systemkern unter Verwendung der vorgestellten Ansätze und Methoden skizziert, der als Grundlage für eine Implementierung eines TI-Coaching-Systems dienen kann.

Inhaltsverzeichnis

1. Einleitung

1.1. Ein TI-Coaching System

In dieser Arbeit sollen die theoretischen Grundlagen und einige exemplarische Implementierungsansätze für ein System zur Schulung der Grundlagen der theoretischen Informatik für Studenten vorgestellt werden. Die Idee zu einem solchen System wurde von Dr. Gunter Grieser entwickelt, auf Basis eines vergleichbaren Systems für Mathematiker.

Hierbei soll der Schwerpunkt auf einem auf Methoden der künstlichen Intelligenz beruhenden System liegen. Das System soll in der Lage sein, an den Studenten gestellte Aufgaben selbstständig zu lösen und die ermittelte Lösung unter strukturellen Gesichtspunkten zu analysieren. Gleichzeitig sollte es auch die vom Studenten eingegebene Lösung analysieren und dabei verschiedene Arten von Fehlern möglichst frühzeitig erkennen. Dabei sollte es genug Flexibilität besitzen, auch solche Lösungen des Studenten als korrekt zu erkennen, die einen anderen möglichen Lösungsweg als den vom System ermittelten verwenden.

1.2. Zielstellung

Ein solches System soll im Idealfall in der Lage sein, folgende Aufgabenstellungen zu erfüllen:

1. Präsentation der Aufgabenstellung gegenüber dem lernenden Studenten
2. Lösen bestimmter klassischer Aufgabenstellungen
3. Analyse und Strukturierung des selbst ermittelten Lösungsansatzes
4. Speicherung und Abruf bereits durchgeführter Analysen und Aufgabenstellungen
5. Verarbeiten der Eingabe eines die gestellte Aufgabe lösenden Studenten
6. Analyse und Kategorisierung möglicher Fehler im studentischen Lösungsansatz
7. Interaktion mit dem Studenten in Bezug auf erkannte und potentielle Fehler

Wie bei vielen interaktiven Systemen, ist es auch hier sinnvoll, das System bezüglich Analyse und Implementierung in ein Front- und Backend aufzutrennen. Dabei ist der Grad der Kommunikation zwischen beiden Komponenten ein wichtiger Gesichtspunkt. Unter Umständen kann es auch sinnvoll sein, die Strukturierung um weitere Teilkomponenten zu verfeinern. Zunächst sollte man sich aber auf eine klare Zweiteilung in der Struktur konzentrieren. Dies ist auch hinsichtlich einer einfacheren Definition der Interaktion zwischen zwei Komponenten sinnvoll.

1.3. Frontend

Die Aufgabe des Frontend des Systems soll in der unmittelbaren Kommunikation mit dem Studenten liegen. Bezüglich der im vorherigen Abschnitt vorgestellten Zielstellungen würde es den wesentlichen Teil der Punkte 1, 5 und 7 abdecken. Eine

wichtige Fragestellung bezüglich des Frontend ist die Anwendungsform. Dabei bieten sich im wesentlichen zwei Möglichkeiten an. Das ist zum einen natürlich eine eigenständige Anwendung und zum anderen ein web-basiertes System.

Für die Form einer eigenständigen Anwendung spricht im wesentlichen die größtmögliche Freiheit bei der Implementierung eines solchen Systems. Der entscheidende Nachteil dieses Ansatzes ist jedoch, wie bei jeder Software, eine wirkungsvolle Verteilung, Aktualisierung und Lokalisierung des Systems.

Für ein web-basiertes System spricht zum einen die schnellere und einfachere Zugänglichkeit für den Benutzer und zum anderen der größere Grad der Kontrolle über das System und dessen ständige Aktualität.

1.4. Backend

Während das Frontend natürlich die größte Sichtbarkeit hat, beinhaltet das Backend den eigentlichen Kern des Systems. Es ist für alle oben als Zielstellung genannten Punkte zuständig, die nicht vom Frontend behandelt werden. Selbst diese Punkte, insbesondere der Punkt 7, müssen beim Entwurf des Backend berücksichtigt werden. Wie beim Frontend sind auch hier verschiedene grundsätzliche Ansätze denkbar.

Wie bei jedem Software-System, besteht natürlich auch hier die Möglichkeit das System im wesentlichen von Grund auf neu zu entwerfen, ohne dabei auf bereits vorhandene Systeme und Techniken zurückzugreifen. Der offensichtliche Vorteil ist dabei natürlich die größtmögliche Freiheit, die man beim Entwurf und der Implementierung besitzt. Ein entscheidender Nachteil ist der stark erhöhte Aufwand, der entsteht, wenn man auf diese Weise "das Rad neu erfindet".

Da sich das System zu einem großen Teil mit dem Lösen von Problemen befasst, stellt der Rückgriff auf erprobte Techniken aus dem Bereich der künstlichen Intelligenz, welche beispielsweise in [RuNo95] beschrieben werden, einen alternativen Ansatz dar. Hierbei bieten sich verschiedene sehr unterschiedliche Techniken an, die sich in Teilen jedoch auch überlappen.

Wie bei jedem logischen Ansatz zur Lösung eines computertechnischen Problems, bietet sich die Programmiersprache Prolog an. Ihre regelbasierte Form bietet bei richtiger Vorgehensweise eine sehr elegante Methode, um Lösungen für logische Probleme zu ermitteln beziehungsweise Fehler in diesen zu erkennen.

Planungssysteme sind ein weiteres Produkt der KI-Forschung, das sich für die gegebene Problemstellungen nutzen lässt. Dazu ist es nur nötig, die im Rahmen des Systems auftretenden und zu lösenden Probleme in Form eines Planungsproblems zu beschreiben. Das Ergebnis in Form eines Planes sollte sich dann genauso einfach als Aufgabenlösung und ähnliches interpretieren lassen.

Eine weitere Möglichkeit wäre der mehr oder weniger direkte Rückgriff auf klassische Logik und die Unterstützung eines automatischen Beweis-Systems. In gewisser Weise ist dies der "natürlichste" Ansatz, da Logik eine entscheidende Grundlage der Theoretischen Informatik darstellt, und die Lösungsansätze für die Aufgabenstellungen, mit denen sich das System befassen soll, auch von logischer Natur sind.

1.5. Zum weiteren Inhalt

Alle oben beschriebenen alternativen Ansätze auf den verschiedenen Ebenen haben ihre Vor- und Nachteile. Sicher bieten sich im Einzelnen noch unzählige weitere Möglichkeiten an.

Kern dieser Arbeit soll jedoch nur ein exemplarischer Ansatz sein. Dabei habe ich mich für die Ansätze einer Implementierung des Backend, auf Grundlage des automatischen Beweisens, entschieden. Die in dieser Einführung vorgestellten alternativen Ansätze werden allerdings an verschiedenen Stellen wieder auftauchen, beim Versuch spezifische Vergleiche zu ziehen. Am Ende werden die Grundlagen einer konkreten Implementierung skizziert.

2. Grundlagen des TI-Coaching-Systems

2.1. Die Aufgaben

Da der grundlegende Zweck des Systems das Verarbeiten von Übungsaufgaben ist, stellt sich natürlich die Frage, welche Formen von Aufgaben das System verarbeiten können soll. Dabei lohnt sich der Blick in einführende Literatur für Theoretische Informatik wie [Schö97] und [HMU01]. Die drei großen Themenbereiche, die in ihnen behandelt werden sind:

1. Automatentheorie und formale Sprachen
2. Berechenbarkeitstheorie
3. Komplexitätstheorie

Rein quantitativ betrachtet fällt auf, dass der erste Bereich in der einführenden Literatur den größten Platz einnimmt. Gleichzeitig wird deutlich, dass die Automatentheorie und formale Sprachen, verglichen zu den beiden anderen Themengebieten, eine viel stärkere rein mechanische Ausprägung besitzen. Beides spricht dafür, dass es lohnenswert ist, sich bei dem geplanten System, zumindest vorläufig, auf Aufgaben aus diesem Bereich zu konzentrieren.

2.2. Automatentheorie und formale Sprachen

Um sich einen Überblick über die denkbaren Übungsaufgaben aus diesem Bereich zu verschaffen, lohnt sich wieder ein Blick in die Literatur. [HMU01] enthält eine große Anzahl solcher Übungsaufgaben. Dabei wird natürlich schnell deutlich, dass diese sehr vielfältig sind. Der Anspruch, sämtliche Formen von Aufgaben, wie sie allein in diesem Buch zu finden sind, durch ein automatisches System zu stellen, zu verarbeiten und Lösungen auf Fehler zu überprüfen, würde zumindest den Umfang dieser Arbeit sprengen.

Es ist also wieder nötig, sich auf einen Teilbereich zu beschränken. Dies kann nun auf zwei Wegen geschehen. Zum einen lässt sich der Bereich aus dem die Aufgaben ausgewählt werden einschränken. Zum anderen kann man sich auf bestimmte Aufgabenformen konzentrieren. Beides sollte aber keine allgemeine Beschränkung des geplanten Systems mit sich bringen, sondern lediglich die Komplexität einer prototypischen Implementierung reduzieren. Die Architektur des Systems sollte in jedem Fall offen und erweiterbar bleiben. Im folgenden soll sich aber hauptsächlich auf ausgewählten Aufgabenformen aus dem Bereich der Regulären Sprachen beschränkt werden.

2.3. Reguläre Aufgaben

Wie im gesamten Bereich der formalen Sprachen, liegt auch bei den regulären, der Schwerpunkt auf den unterschiedlichen Repräsentationen entsprechender Sprachen. Der Umgang mit diesen und die Zusammenhänge zwischen ihnen stellen den Kern dessen dar, was ein Student hier lernen und üben soll. Folgende Formen bzw. *Repräsentationen* spielen dabei eine Rolle:

1. Allgemeine, formale Beschreibungen
2. Reguläre Grammatiken
3. Deterministische und nichtdeterministische endliche Automaten
4. Reguläre Ausdrücke

Hierbei nimmt die erste Form eine Sonderstellung ein, da sie eigentlich eine Vielzahl von Beschreibungsformen darstellt. Sie beinhaltet alle Arten natürlichsprachlicher Beschreibungen einer Sprache und kann somit als Rohform betrachtet werden. Allgemein lässt sich sagen, dass sie zu vielfältig ist, um sie im Rahmen dieses Systems vollständig verarbeitbar zu machen. Letztendlich sind das wohl genau die Gründe, die zur Einführung der anderen, abstrakteren *Repräsentationen* geführt haben.

Viele Übungsaufgaben befassen sich mit der Erzeugung entsprechender *Repräsentationen* für bestimmte Sprachen oder dem Umwandeln einer Form in eine andere. Im Prinzip handelt es sich beim Erzeugen auch nur um eine Umwandlung von der allgemeinen Beschreibungsform in eine der abstrakten Repräsentationen. Aber es ist sinnvoll, es als gesonderten Fall zu betrachten. Ein anderer Aufgabentyp ist der Nachweis bestimmter Eigenschaften der einzelnen Darstellungsformen oder bestimmter Sprachen. Wenn man nur diese drei Aufgabenformen bzw. Kombinationen aus diesen betrachtet, lässt sich bereits eine Vielzahl von Aufgaben finden.

Es soll nun als erstes versucht werden, diese Arten der Übungsaufgaben zu formalisieren, mit dem Ziel, eine Grundlage zur automatischen Verarbeitung zu schaffen. Dabei soll auch die Form der vom Studenten angegebenen Lösungen eine Rolle spielen. Das Ziel ist es Fehler in Lösungen möglichst früh und genau ausfindig zu machen.

2.3.1. Synthese

Der erste und direkteste Aufgabenbereich ist die Erzeugung einer bestimmten *Repräsentation* einer mehr oder weniger formal beschriebenen Sprache. Zum Beispiel: Geben sie eine reguläre Grammatik an für die Sprache aller Worte über dem Alphabet {a,b} die mit einem a beginnen und enden.

Dieses Beispiel illustriert ganz gut die Schwierigkeit für ein automatisches System, die in diesem Aufgabentyp verborgen sind. Während die Lösung dieser Aufgabe anhand der gegebenen Sprachbeschreibungen für einen Menschen, der mit regulären Grammatiken vertraut ist, sehr leicht zu finden ist, erfordert es ein sehr komplexes System, um nur die Beschreibung der Sprache zu verstehen. Es ist also klar, dass man auch hier eine formale Einschränkung finden muss. Ein Trick, um dieses Problem teilweise zu lösen, kann es auch sein, wenn das System ein Menge von Paaren aus natürlichsprachlichen

Beschreibungen und einer formaleren und abstrakteren Darstellung vieler Sprachen gespeichert und abrufbereit hält.

2.3.2. Transformation

Dieser Aufgabentyp umfasst alle Umwandlungen von einer Repräsentation in eine andere. Im Vergleich zur Synthese sollte dieser Bereich relativ leicht automatisch zu verarbeiten sein, da die üblichen Transformationen in algorithmischer Form vorliegen. Dazu muss das System, bezogen auf den Bereich der regulären Sprachen, zunächst zu den folgenden Dingen in der Lage sein:

- Transformation einer regulären Grammatik in einen NFA
- Transformation eines DFA in eine regulären Grammatik
- Transformation eines NFA in einen DFA
- Transformation eines DFA in einen regulären Ausdruck
- Transformation eines regulären Ausdruck in einen NFA

Zu jeder dieser Transformation existiert ein vergleichsweise leicht implementierbarer Algorithmus. Die Transformationen, die durch diese fünf Varianten nicht abgedeckt werden, lassen sich durch Verkettung der entsprechenden Transformationen erreichen. Im Falle der Transformation eines DFA in einen NFA ist im Prinzip nur eine Redefinition durchzuführen. Anhand dieser Transformationen lässt sich also schon eine große Fülle von Aufgaben, die theoretisch sogar zufällig erzeugt werden können, ableiten.

2.3.3. Qualifikation

Hierbei handelt es sich nun um den Aufgabentyp, bei dem es darum geht, für eine in irgendeiner Repräsentation gegebenen Sprache bzw. irgendeiner Repräsentation im Allgemeinen, bestimmte Eigenschaften nachzuweisen. Je nachdem, um welche Eigenschaften es sich dabei im einzelnen handelt, sind hier Aufgaben mit unterschiedlichstem Schwierigkeitsgrad denkbar. Gemeint ist damit, schwierig für den Studenten oder schwierig für das System oder natürlich beides.

Wie bei der Synthese, existiert hierbei ein Formulierungsproblem. Während man in natürlicher Sprache unzählige Eigenschaften beschreiben kann, ist es wiederum schwer all diese Beschreibungen für das System zugänglich zu machen. Dies spricht also auch bei dieser Art der Aufgabe dafür, sich auf ganz bestimmte Eigenschaften zu beschränken.

2.3.4. Kombination

Aus den drei genannten Aufgabentypen Synthese, Transformation und Qualifikation lassen sich natürlich eine Menge neuer Aufgabentypen kombinieren. Wenn man eine Aufgabe als Paar aus einer Eingabe und einer Ausgabe betrachtet, so kann man dadurch jedes Paar mit passenden Ein- und Ausgaben zu einer längeren Aufgabe verketten.

Dies wirft jedoch eine weitere Fragestellung auf, die mit der Präsentation der Aufgabe

selbst zusammenhängt: Wie genau sind die in der Aufgabe enthaltenen Vorgaben an den Studenten, den Lösungsweg betreffend? Eine solche kombinierte Aufgabe lässt sich ja entweder als Folge von Einzelaufgaben oder als eine große Aufgabe präsentieren. Für letztere liegt es im Ermessen des Studenten, durch die Lösung welcher Teilaufgaben die endgültige Lösung ermittelt wird. Letztendlich lässt sich jede Aufgabe so in gewissen Grenzen aufgliedern, was dann auch den Schwierigkeitsgrad der Aufgabe beeinflusst.

2.4. Fehlerquellen

Bisher wurden die Aufgaben nur von der Seite des Systems und wie dieses sie selbstständig lösen kann, betrachtet. Dies ist aber nur ein kleiner Aspekt des geplanten Systems, denn eigentlich sollte das Ziel sein, die von einem Studenten zu einer Aufgabe eingegebene Lösung auf Fehler zu überprüfen. Dazu ist es natürlich wichtig, dass das System zu einer korrekten Lösung der genannten Aufgaben in der Lage ist.

Viel wichtiger ist es aber, die Studentenlösung auf Fehler zu überprüfen, denn das ist einer der zentralen Dienste, den das System erbringen soll. Dazu ist es nötig, die einzelnen Aufgabenformen hinsichtlich aller möglichen Fehler, die ein Student bei der Lösung machen kann, zu analysieren.

Eine Fehlerart, die natürlich bei jeder Art von Aufgabe auftreten kann, sind alle Formen von Flüchtigkeitsfehlern. Diese sind die typischen "menschlichen" Fehler, die in der Regel auf Unaufmerksamkeit beruhen. Dies beinhaltet beispielsweise Schreibfehler, Verwechslungen oder das schlichte Übersehen bestimmter Schritte. Bei diesen Fehlern handelt es sich um die wohl häufigsten und einfachsten Fehler, die sich idealerweise direkt während der Eingabe erkennen lassen. Wobei sich der Vergleich mit der Rechtschreibprüfung in einem Textverarbeitungsprogramm aufdrängt. In bestimmten Fällen und je nach Aufgabentyp können diese Fehler jedoch auch sehr problematisch zu erkennen sein. Es ist also sehr wohl sinnvoll, bei jeder speziellen Aufgabe, diese Fehlerform zu berücksichtigen.

Eine andere Form von Fehlern, ist meist aufgabenspezifisch. Sie ergeben sich unmittelbar aus der speziellen Lösung für bestimmte Problemstellungen. Beispiele hierfür sind falsch angewandte Algorithmen oder auch einfach ungeeignete Lösungen für eine bestimmte Aufgabenstellung. Letztendlich führt nur die genaue Analyse bestimmter Lösungsansätze dazu, diese Fehlerform zu erkennen.

Schließlich besteht bei jedem vermeintlichen Fehler die Möglichkeit, dass es überhaupt kein wirklicher Fehler ist, sondern einfach nur eine Methode beziehungsweise ein Trick, die dem System nicht bekannt sind. Am Ende ist jeder Student ein Mensch und neigt dazu, auf neue Ideen zu kommen, die vorher niemand bedacht hat.

2.5. Fehlerbehandlung

Der nächste Schritt nach dem Ausmachen der Fehlerquellen und dem Erkennen konkreter Fehler besteht natürlich in der Behandlung solcher Ereignisse durch das System. Dazu gibt es zunächst eine passive und eine aktive Vorgehensweise. Im passiven Fall registriert das System nur den Fehler, verarbeitet die Lösung jedoch normal weiter, sofern der Fehler nicht so kritisch ist, dass alles folgende keinen Sinn

mehr ergibt. Dies hat auch im Zusammenhang mit den bereits erwähnten nur vermeintlichen Fehlern den Vorteil, dass kein Fehlalarm ausgelöst wird. Im aktiven Fall reagiert das System unmittelbar nach Erkennung auf den Fehler, indem es den Studenten darauf hinweist.

Beide Varianten haben, abhängig von der genauen Art des Fehlers, ihre Daseinsberechtigung. Während in bestimmten Fällen eine aktive Fehlermeldung für den Studenten eher störend wirken könnte, kann eine frühzeitige Warnung dem Studenten im Fehlerfall auch viel Frustration ersparen. Einen für jede Aufgabe und jeden Studenten idealen Ansatz gibt es vermutlich nicht. Es bietet sich also an, dieses Verhalten auf irgendeine Weise konfigurierbar zu machen. Es ist auch denkbar, die Unterscheidung zwischen aktiver und passiver Behandlung dynamisch zu gestalten. So könnte das System sich auf eine aktive Fehlerbehandlung umstellen, sobald die Anzahl der passiv erkannten Fehler eine bestimmte Schwelle überschritten hat.

Im Zusammenhang mit aktiver Fehlerbehandlung stellt sich auch die Frage der genauen Form der Rückmeldung für den Studenten sowie nach den Möglichkeiten, welche dieser hat um darauf zu reagieren. Sinnvoll könnte zum einen eine grobe Klassifizierung des Fehlers sein. So könnte beispielsweise zwischen Warnungen sowie einfachen und kritischen Fehlern unterschieden werden. Als alternative Reaktionen für den Studenten würden sich im wesentlichen das Ignorieren oder das Korrigieren des Fehlers anbieten, wobei letzteres unter Umständen zum Verwerfen bereits gemachter Lösungsansätze führen kann.

2.6. Aufgabenabwicklung

Während sich die bisherigen Abschnitte vor allen Dingen mit der Behandlung von Fehlern befasst hat, die während der Eingabe beziehungsweise Verarbeitung der studentischen Lösung auftreten, ist der wichtigste Aspekt natürlich die Abwicklung der vollständigen Lösung einer Übungsaufgabe. Wenn man eine Lösung als Ganzes betrachtet, gibt es im wesentlichen die folgenden Ergebnisse:

1. Die Lösung ist falsch
2. Die Lösung ist korrekt, aber der Lösungsweg ist für das System nicht nachvollziehbar
3. Die Lösung ist korrekt, aber entspringt einer alternativen Vorgehensweise
4. Die Lösung ist korrekt und entspricht der Vorgehensweise des Systems

Im Falle von korrekten Lösungen besteht zudem immer die Möglichkeit, dass trotzdem Fehler erkannt wurden, die jedoch aus irgendwelchen Gründen die endgültige Lösung nicht beeinflussen. Ein typisches Beispiel sind vorangegangene Flüchtigkeitsfehler, die der Student im späteren Verlauf jedoch bewusst oder unbewusst korrigiert hat.

In jedem Fall wird jeder Student natürlich an irgendeiner Form der Rückmeldung zu seiner Lösung interessiert sein. Der einfachste Fall ist der Hinweis, ob die Lösung falsch oder richtig ist. Deutlich interessanter ist jedoch eine genauere Bewertung der Lösung. Bei einer fehlerhaften Lösung, könnte das System die Punkte aufzuzeigen, bei denen der Student Fehler gemacht hat. Bei einer korrekten aber alternativen Lösung könnte das

System den eigenen Lösungsansatz präsentieren und ihn dem studentischen gegenüberstellen.

Aus didaktischer Sicht ist der Schritt der Aufgabenabwicklung sehr entscheidend, da dies die Situation ist, in der der Student aus seinen Fehlern lernen oder durch die Bestätigung einer richtigen Lösung motiviert wird. Je genauer die vorangegangene Analyse der Aufgabe und der Lösung war, desto besser kann man hier den Studenten über Erfolg oder Misserfolg seiner Bemühungen informieren.

2.7. Ergebnisse dieses Kapitels

In diesem Kapitel wurden die grundlegende Vorgehensweise eines TI-Coaching-Systems betrachtet. Die Schlussfolgerungen sind nun folgende:

- Es ist sinnvoll, sich zunächst nur auf ausgewählte Aufgabentypen zu beschränken: In diesem Fall sind das Aufgaben zum Bereich der regulären Sprachen.
- Aufgaben aus dem gewählten Bereich der regulären Sprachen lassen sich in bestimmte Klassen einteilen: Synthese, Transformation, Qualifikation und Kombinationen daraus.
- Ein wichtiger Aspekt jedes Aufgabentyps sind die Fehlerquellen, die ihn betreffen. Dabei kann man zwischen allgemeinen und für die Aufgabe spezifischen Fehlern unterscheiden.
- Neben der Fehlererkennung spielt die genaue Form der Fehlerbehandlung eine wichtige Rolle.
- Die abschließende Aufgabenabwicklung ist für den Lernenden von zentraler Bedeutung.

3. Analyse von Übungsaufgaben

3.1. Problemstellung

Im vorigen Kapitel wurden einige grundlegende Überlegungen dazu angestellt, welche Aspekte eines Aufgabentyps genauer betrachtet werden sollten, um diese auf sinnvolle Weise im Rahmen des Coaching-Systems zu verarbeiten. Eine vernünftige Analyse sollte folgende Schritte beinhalten:

1. Eindeutige Beschreibung des Aufgabentyps mit einer klaren Abgrenzung zu anderen Arten von Aufgaben
2. Entwickeln eines oder mehrerer automatisierter Lösungswege des durch eine entsprechende Aufgabe gestellten Problems
3. Präzise Strukturierung der einzelnen Schritte der gewählten Lösungsansätze. Falls möglich und sinnvoll eine hierarchische Gliederung in Schritte und Unterschritte
4. Betrachtung möglicher Fehlerquellen bei der Bearbeitung durch einen Studenten der betrachteten Lösungswege
5. Kategorisierung der gefundenen Fehlerquellen

Bevor man einzelne Aufgabenarten einer entsprechenden Analyse unterzieht sollte man versuchen, diese in möglichst elementare Bestandteile zu zerlegen und so gesondert zu betrachten, da es zunächst nicht sinnvoll ist, eine Aufgabenart als Ganzes zu betrachten, wenn sie eigentlich die Kombination mehrerer Teilaufgaben darstellt.

Bei der Betrachtung der drei Aufgabenarten Synthese, Transformation und Qualifikation wurde deutlich, dass die Transformationsaufgaben die am einfachsten für eine automatisches System zu verarbeitenden Aufgaben sind. Deshalb sollen zunächst diese einer genaueren Analyse unterzogen werden.

3.2. Analyse von Transformationsaufgaben

Zur Umwandlung der verschiedenen abstrakten Repräsentationen regulärer Sprachen, namentlich der regulären Grammatiken, regulären Ausdrücke sowie deterministischen und nichtdeterministischen endlichen Automaten, ineinander wurden im letzten Kapitel fünf grundlegende Umwandlungen aufgezählt. Die im folgenden verwendeten Algorithmen werden beispielsweise in [Schö97] beschrieben, wo sie hauptsächlich als Grundlage von Äquivalenzbeweisen der einzelnen Formen zueinander dienen. Alle weiteren Transformationen sind entweder trivial oder lassen sich durch Hintereinanderausführung der genannten fünf Basistransformationen realisieren.

So lässt sich beispielsweise ein DFA sehr leicht als NFA mit nur einem Startzustand darstellen, während die Umwandlung eines regulären Ausdrucks in eine Grammatik über den Umweg der Transformation in einen NFA und dann in einen DFA erreicht wird.

Im folgenden sollen nun die fünf Basistransformationen anhand der im obigen

Abschnitt Problemstellung genannten Schritte analysiert werden. Begonnen werden soll hierbei mit der wohl einfachsten Umwandlung einer Grammatik in einen NFA. Am Ende wird auch noch die besondere Aufgabe der Minimierung eines DFA analysiert.

3.3. Reguläre Grammatik in Nichtdeterministischen Endlichen Automaten

Gegeben ist eine reguläre Grammatik G für eine Sprache L. Wandeln Sie diese in einen NFA M um, welcher die selbe Sprache erkennt. Die Grammatik ist gegeben als G = (V,Σ,P,S). Dabei ist V die Menge der Variablen, Σ die Menge der Terminale, P die Menge der Produktionen und S das Startsymbol. Der NFA hat die Form M = (Z,Σ,δ,S,E). Dabei ist Z die Menge der Zustände, Σ die Menge des Eingabealphabets, δ die Überführungsfunktion, S die Menge der Startzustände und E die Menge der Endzustände.

3.3.1. Eine automatisierte Lösung

Eine einfache Beschreibung eines Algorithmus zur Umwandlung eines regulären Grammatik in einen NFA lautet wie folgt: Die Menge der Zustände Z des Automaten M entspricht der Menge der Variablen V der Grammatik G mit einem zusätzlichen Zustand E(G). Das Eingabealphabet von M ist identisch mit dem Terminalalphabet von G. Für jede Produktion der Form $X \to yZ$ der Grammatik existiert ein Übergang $\delta(X, y) \ni Z$. Die Menge der Startzustände ist die Menge die nur aus dem Startsymbol von G besteht und die Menge der Endzustände besteht nur aus dem neu eingeführten Zustand E(G)

Der folgende Lösungsansatz soll auf einem automatischen Beweiser basieren. Die Methode des automatischen Beweisens wird beispielsweise in [Bib92] sehr ausführlich beschrieben. Zu diesem Zweck soll nun eine geeignete, wenn auch auf den ersten Blick vielleicht etwas umständlich wirkende, logische Darstellung der zu verarbeitenden Objekte vorgestellt werden. Sowie logische Regeln, die die Umwandlung beschreiben.

Dazu dienen die folgenden Prädikate:

$vars_G(G, V)$: V ist die Menge der Variablen der Grammatik G
$term_G(G, \Sigma)$: Σ ist die Menge der Terminale der Grammatik G
$pro^1_G(G, x, y, z)$: $x \to yz$ ist eine Produktion für G
$pro^0_G(G, x, y)$: $x \to y$ ist eine Produktion für G
$start_G(G, S)$: S ist das Startsymbol von G
$stat_N(M, Z)$: Z ist die Menge der Zustände des NFA M
$alph_N(M, \Sigma)$: Σ ist das Eingabealphabet von M
$trans_N(M, x, y, z)$: $\delta(x, y) \ni z$ ist ein Übergang für M
$start_N(M, S)$: S ist die Menge der Startzustände von M
$end_N(M, E)$: E ist die Menge der Endzustände von M

Der logische Zusammenhang zwischen einer Grammatik G und eines äquivalenten NFA M lässt sich wie folgt beschreiben. Dabei werden zwei eindeutig bestimmte und neue Atome eingeführt: M(G) für den erzeugten Automaten sowie ein neuer Endzustand E(G):

$$vars_G(G, V) \rightarrow stat_N(M(G), V \cup \{E(G)\})$$
$$term_G(G, \Sigma) \rightarrow alph_N(M(G), \Sigma)$$
$$pro_G^1(G, x, y, z) \rightarrow trans_N(M(G), x, y, z)$$
$$pro_G^0(G, x, y) \rightarrow trans_N(M(G), x, y, E(G))$$
$$start_G(G, S) \rightarrow start_N(M(G), \{S\})$$
$$end_N(M(G), \{E(G)\})$$

Mit den Prädikaten zur Beschreibung der Grammatik G sowie obigen Regeln kann ein Beweissystem sämtliche den Automaten M(G) beschreibenden Prädikate ableiten. Dazu wird einfach versucht, folgende Aussage zu beweisen:

$$\exists M(G). \exists E(G). Definition_N^G(M(G), E(G))$$

Dabei steht $Definition_N^G$ für die Gesamtheit der die Umwandlung beschreibenden Regeln.

Es wird natürlich davon ausgegangen, dass die Prädikate für die Grammatik korrekt definiert wurden und allen Einschränkungen für eine entsprechende Definition genügen. Im einzelnen entsprechen diese Einschränkungen folgenden logischen Aussagen:

$$vars_G(G, V) \wedge term_G(G, \Sigma) \rightarrow V \cap \Sigma = \emptyset$$
$$vars_G(G, V) \wedge term_G(G, \Sigma) \wedge pro_G^1(x, y, z) \rightarrow x \in V \wedge z \in V \wedge y \in \Sigma$$
$$vars_G(G, V) \wedge term_G(G, \Sigma) \wedge pro_G^0(x, y) \rightarrow x \in V \wedge y \in \Sigma$$
$$vars_G(G, V) \wedge start_G(G, S) \rightarrow S \in V$$

Auch ein NFA M unterliegt vergleichbaren Einschränkungen:

$$stat_N(M, Z) \wedge alph_N(M, \Sigma) \rightarrow Z \cap \Sigma = \emptyset$$
$$stat_N(M, Z) \wedge alph_N(M, \Sigma) \wedge trans_N(x, y, z) \rightarrow x \in Z \wedge z \in Z \wedge y \in \Sigma$$
$$stat_N(M, Z) \wedge start_N(M, S) \rightarrow S \subseteq V$$
$$stat_N(M, Z) \wedge end_N(M, E) \rightarrow E \subseteq V$$

Das Hinzufügen dieser Regeln als Axiome stellt eine zusätzliche Konsistenzprüfung der verarbeiteten Daten sicher.

Auch wenn mit dem vorgestellten Verfahren zu jeder entsprechend definierten Grammatik ein äquivalenter NFA ableitbar ist, besitzt dieser Ansatz einen entscheidenden Mangel: Die Namen des resultierenden Zustände sind eindeutig bestimmt. Ein Student, der eine entsprechende Umwandlung durchführt, hat aber

absolute Freiheit, was die Benennung der Zustände des von ihm erzeugten Automaten angeht.

Es reicht also in keinem Fall einfach das Ergebnis des Beweisers mit dem des Studenten zu vergleichen. Es ist zumindest eine Prüfung auf Äquivalenz modulo der Zustandsnamen durchzuführen. Eine weitere Problemstellung ist die Tatsache, dass der Student theoretisch auch einen Automaten angeben kann, der nicht einmal in der Anzahl der verwendeten Zustände übereinstimmt. Trotzdem lässt sich daraus natürlich nicht schließen, dass der entsprechende Automat nicht die korrekte Sprache akzeptiert.

Es zeigt sich also schon an dieser Stelle, dass selbst bei den einfachsten Aufgaben starke Abweichungen vom Lösungsverhalten des Systems auftreten können, die jedoch nicht eindeutig falsch sind. Allerdings sind die genannten Probleme nicht unlösbar. Mittels der noch zu analysierenden Lösungen für die Umwandlung in einen DFA und die Minimierung des selbigen, steht eine Methode zur Verfügung, beliebige endliche Automaten auf Äquivalenz zu überprüfen.

Zunächst sollte aber die Analyse der Transformation einer Grammatik in einen NFA fortgesetzt werden.

3.3.2. Strukturierung der Lösungsschritte

Eine Lösung des Problems der Umwandlung einer regulären Grammatik in einen NFA besteht im wesentlichen aus folgenden Schritten:

1. Erzeugen eines Zustandes für jede Variable der Grammatik
2. Hinzufügen eines neuen und eindeutigen Endzustands E
3. Übernahme des Terminalalphabets der Grammatik als Eingabealphabet des Automaten
4. Hinzufügen einer Transition $\delta(x, y) \ni z$ für jede Produktion $x \to yz$
5. Hinzufügen einer Transition $\delta(x, y) \ni E$ für jede Produktion $x \to y$ wobei E der neue Endzustand ist
6. Festlegen der Startzustände als Menge, die nur aus dem Zustand besteht der dem Startsymbol der Grammatik entspricht
7. Festlegen der Endzustände als Menge die nur aus dem neuen Endzustand E besteht

Anzumerken ist, dass bei diesem Vorgehen die Reihenfolge der einzelnen Schritte relativ frei wählbar ist und auch vermischt werden kann. Auch die genaue Benennung der Zustände ist nicht eindeutig festgelegt. Außerdem ist zu beachten, dass der durch diese Lösung erzeugte Automat natürlich keineswegs der einzige ist, der die von der Grammatik erzeugte Sprache akzeptiert. Beispielsweise kann jede Transformation durch Analyse der erzeugten Sprache und anschließender Synthese eines passenden Automaten durchgeführt werden. Die Ergebnisse können dabei sehr vielfältig sein.

3.3.3. Fehlerquellen

Prinzipiell ist diese Transformation eine relativ einfache und sollte somit nicht übermäßig fehleranfällig sein. Typischerweise sollten, sofern dieses oder ein ähnliches Verfahren überhaupt bekannt ist, Flüchtigkeitsfehler die primäre Fehlerquelle für eine Lösung des gegebenen Problems sein.

Ein denkbarer spezifischer Fehler könnte jedoch sein, dass der Student Produktionen auf falsche Weise in Transitionen umwandelt. So könnte er beispielsweise jede Produktion der Form $X \rightarrow yZ$ umwandeln in eine Transition der Form $\delta(Z, y) \ni X$ und somit den Automaten quasi verkehrt herum erzeugen.

3.3.4. Kategorisierung der Fehler

Da Flüchtigkeitsfehler die Hauptfehlerquelle für diese einfache Transformation darstellen, sollten sie der Schwerpunkt dieser Analyse sein. Folgende Kategorien bieten sich hierfür an:

1. *Verwechslungen*: Der Student verwechselt Variablen, Zustände oder Terminalsymbole. Zum Beispiel: Für eine Produktion $S \rightarrow aB$ erzeugt er eine Transition $\delta(S, b) \ni B$.
2. *Nachlässigkeit*: Der Student übersieht bestimmte Teile der Grammatik bei der Umwandlung. Beispiel: Er vergisst bestimmte Produktionen umzuwandeln.
3. *Unvollständigkeit*: Der Student definiert den Automaten nicht vollständig. Beispielsweise gibt er nur die Transitionen explizit an.
4. *Falsche Umwandlung*: Der Student wandelt wie oben beschrieben die Produktionen auf falsche Weise in Transitionen um.

Neben der hier angegebenen Benennung und Kategorisierung ist eine weitere Unterscheidung für jede Fehlerart von Bedeutung. Während man Fehler wie die 1. und 2. Kategorie als einfache Fehler betrachten kann, handelt es sich bei den Kategorien 3. und 4. in der Regel um systematische Fehler. Der Unterschied besteht darin, dass einfache Fehler zwar immer wieder passieren können, aber nicht immer passieren müssen. Systematische Fehler dagegen sind vorhersehbar, da sie, sofern man den Verursacher nicht darauf hinweist, praktisch immer wieder passieren. In der Regel beruhen einfache Fehler auf Flüchtigkeit, während systematische Fehler das Ergebnis von Missverständnissen sind.

Im Falle der obigen Kategorie der *Unvollständigkeit* könnte das Missverständnis darin liegen, dass der Student der Meinung ist, dass die Transitionen den Kern der Definition eines Automaten beschreiben und sich der Rest daraus ja ergibt. Im Falle der *Falschen Umwandlung* hat der Student wohl schlicht das Verfahren falsch verstanden. In beiden Fällen ist es angebracht, den Studenten spätestens während der Aufgabenabwicklung auf diese Fehler hinzuweisen.

Vergleicht man den Flüchtigkeitsfehler der *Verwechslung* mit dem spezifischen Fehler der *Falschen Umwandlung* so fällt auf, dass diese nicht immer sofort klar unterscheidbar sind. Wenn der letzte Fehler nur einmal auftaucht, handelt es sich im Zweifel eigentlich nur um die 1. Kategorie. Diese Problemstellung lässt sich auch

verallgemeinern und führt zu dem Schluss, das sich die genaue Kategorie bestimmter Fehler nicht immer nach dem ersten Auftreten festlegen lässt. In gewisser Weise ist dies mit der Thematik der Phantomfehler, also der nur vermeintlichen Fehler verwandt.

3.4. Deterministischer Automat in Reguläre Grammatik

Gegeben ist ein deterministischer endlicher Automat, der die Sprache L akzeptiert. Wandeln Sie diesen in eine reguläre Grammatik G um, welche die selbe Sprache erzeugt. Der DFA ist gegeben als M = (Z,Σ,δ,S,E). Dabei ist Z die Menge der Zustände, Σ die Menge des Eingabealphabets, δ die Überführungsfunktion, S der Startzustand und E die Menge der Endzustände. Die Grammatik hat die Form G = (V,Σ,P,S). Dabei ist V die Menge der Variablen, Σ die Menge der Terminale, P die Menge der Produktionen und S das Startsymbol.

3.4.1. Automatisierte Lösung

Eine informelle Beschreibung eines Algorithmus für diese Transformation lautet wie folgt: Die Menge der Variablen von G entspricht der Menge der Zustände des DFA. Das Terminalalphabet der Grammatik ist identisch mit dem Eingabealphabet von M. Das Startsymbol von G ist der Startzustand von M. Für jede Transition $\delta(X, y)=Z$ existiert eine Produktion $X \to yZ$ und falls Z ein Endzustand ist zusätzlich die Produktion $X \to y$.

Auch für diesen Algorithmus soll wieder ein logischer Lösungsansatz vorgestellt werden, der bei Verwendung eines automatischen Beweissystems zu einem Ergebnis führt.

Zur Definition des DFA sollen folgende Prädikate verwendet werden:

$stat_D(M, Z)$: Z ist die Menge der Zustände des DFA M
$alph_D(M, \Sigma)$: Σ ist das Eingabealphabet von M
$trans_D(M, x, y, z)$: $\delta(x, y)=z$ ist ein Übergang für M
$start_D(M, S)$: S ist der Startzustand von M
$end_D(M, E)$: E ist die Menge der Endzustände von M

Diese sollen folgenden Einschränkungen unterliegen:

$stat_D(M, Z) \wedge alph_D(M, \Sigma) \to Z \cap \Sigma = \emptyset$
$stat_D(M, Z) \wedge alph_D(M, \Sigma) \wedge trans_d(x, y, z) \to x \in Z \wedge z \in Z \wedge y \in \Sigma$
$stat_D(M, Z) \wedge start_D(M, S) \to S \in V$
$stat_D(M, Z) \wedge end_D(M, E) \to E \subseteq V$

Die Prädikate und Einschränkungen für die reguläre Grammatik entsprechen denen im vorherigen Abschnitt vorgestellten.

Folgende Regeln beschreiben die entsprechende Transformation:

$$stat_D(M, Z) \rightarrow vars_G(G(M), Z)$$
$$alph_D(M, \Sigma) \rightarrow term_G(G(M), \Sigma)$$
$$start_D(M, S) \rightarrow start_G(G(M), S)$$
$$trans_D(M, x, y, z) \rightarrow pro^1_G(G(M), x, y, z)$$
$$trans_D(M, x, y, z) \wedge end_D(M, E) \wedge z \in E \rightarrow pro^0_G(G(M), x, y)$$

Mittels all dieser Regeln als Axiome und den Definitionen des Automaten lässt sich wieder eine vollständige Definition einer äquivalenten regulären Grammatik ableiten. Der zu beweisende Satz lautet hierbei, wobei $Definition^M_G$ wieder für die Gesamtheit der oben stehenden Erzeugungsregeln steht :

$$\exists G(M).\, Definition^M_G(G(M))$$

Wie schon bei der vorherigen Transformation wird hierbei jedoch nur eine von vielen möglichen Lösungen erzeugt. Zum Vergleich mit einer beliebigen anderen Lösung eines Studenten werden noch einige weitere Hilfsmittel benötigt.

Im Prinzip lässt sich das gegebene Verfahren auch zur Umwandlung eines NFA in eine reguläre Grammatik verwenden, sofern der Automat nur einen Startzustand besitzt. Zusammen mit der Möglichkeit, jeden NFA mit mehreren Startzuständen in einen äquivalenten NFA mit genau einem Startzustand zu verwandeln, ergibt sich so eine schnellere Transformation, als würde man den Umweg über einen DFA wählen. Dies ist auch ein typisches Beispiel dafür, dass das System immer damit rechnen muss, dass ein Student eine "bessere" Lösung findet als die vorgesehene.

Die beschriebene "Abkürzung" funktioniert im wesentlichen wie folgt: Man fügt dem NFA einen neuen Zustand S(M) hinzu und für jeden Übergang $\delta(X, y) \ni Z$ bei dem X ein alter Startzustand ist fügt man den Übergang $\delta(S(M), y) \ni Z$ hinzu. Danach legt man S(M) als den einzigen Startzustand des Automaten fest. Man hat nun einen NFA der dem alten entspricht, bis auf einen einzelnen, neuen und "vorgeschalteten" Startzustand S(M).

Eine logische Beschreibung dieser Methode unter Verwendung der eingeführten Prädikate und Einschränkungen für reguläre Grammatiken und nichtdeterministische endliche Automaten lautet nun:

$$stats_N(M, Z) \rightarrow stats_N(N, Z \cup \{S(M)\})$$
$$alph_N(M, \Sigma) \rightarrow alph_N(N, \Sigma)$$
$$trans_N(M, x, y, z) \rightarrow trans_N(N, x, y, z)$$
$$trans_N(M, x, y, z) \wedge start_N(M, S) \wedge x \in S \rightarrow trans_N(N, S(M), y, z)$$
$$start_N(N, \{S(M)\})$$
$$end_N(M, E) \rightarrow end_N(N, E)$$
$$stat_N(N, Z) \rightarrow vars_G(G(M), Z)$$

$$alph_N(N, \Sigma) \rightarrow term_G(G(M), \Sigma)$$
$$start_G(G(M), S(M))$$
$$trans_N(N, x, y, z) \rightarrow pro^1_G(G(M), x, y, z)$$
$$trans_N(N, x, y, z) \wedge end_N(N, E) \wedge z \in E \rightarrow pro^0_G(G(M), x, y)$$

Die obigen Regeln beschreiben eigentlich zwei vollständige Transformationen. Die vom NFA M in den äquivalenten NFA N mit nur einem Startzustand und anschließend die Umwandlung des Automaten N in eine reguläre Grammatik. Das Verfahren ließe sich etwas vereinfachen, wenn man auf die redundante zweifache Umwandlung der Teile verzichtet, die sich zwischen M und N nicht unterscheiden. Ein stark komprimiertes Verfahren sähe wie folgt aus:

$$stat_N(M, Z) \rightarrow vars_G(G(M), Z \cup \{S(M)\})$$
$$alph_N(M, \Sigma) \rightarrow term_G(G(M), \Sigma)$$
$$trans_N(M, x, y, z) \rightarrow pro^1_G(G(M), x, y, z)$$
$$trans_N(M, x, y, z) \wedge end_N(M, E) \wedge z \in E \rightarrow pro^0_G(G(M), x, y)$$
$$trans_N(M, x, y, z) \wedge start_N(M, S) \wedge x \in S \rightarrow pro^1_G(G(M), S(M), y, z)$$
$$trans_N(M, x, y, z) \wedge start_N(M, S) \wedge end_N(M, E) \wedge x \in S \wedge z \in E \rightarrow pro^0_G(G(M), S(M), y)$$
$$start_G(G(M), S(M))$$

Der zu beweisende Satz lautet hier:

$$\exists G(M). \exists S(M). Definition^N_G(G(M), S(M))$$

3.4.2. Strukturierung

Die Umwandlung eines DFA in eine reguläre Grammatik erfolgt in den folgenden Schritten:

1. Erzeugen einer Variablen für jeden Zustand des Automaten
2. Übernahme des Eingabealphabets des Automaten als Terminalalphabet der Grammatik
3. Erzeugen einer Produktion $X \rightarrow yZ$ für jede Transformation $\delta(X, y) = Z$
4. Erzeugen einer Produktion $X \rightarrow y$ für jede Transformation $\delta(X, y) = Z$ bei der Z ein Endzustand des Automaten ist
5. Übernahme des Startzustands als Startsymbol

Auch bei diesem Verfahren ist die genaue Reihenfolge der Teilschritte nicht von Bedeutung und sie können beliebig vermischt werden.

Ein leicht modifiziertes Verfahren beschreibt die Umwandlung eines NFA in eine

Grammatik, bei dem wiederum die Einzelschritte keiner bestimmten Reihenfolge gehorchen müssen:

1. Erzeugen einer Variablen für jeden Zustand des Automaten
2. Hinzufügen einer neuen und eindeutigen Variable S, die gleichzeitig Startsymbol ist
3. Übernahme des Eingabealphabets des Automaten als Terminalalphabet der Grammatiken
4. Erzeugen einer Produktion $X \rightarrow yZ$ für jede Transformation $\delta(X, y) = Z$
5. Erzeugen einer Produktion $X \rightarrow y$ für jede Transformation $\delta(X, y) = Z$ bei der Z ein Endzustand des Automaten ist
6. Erzeugen einer Produktion $S \rightarrow yZ$ für jede Transformation $\delta(X, y) = Z$ bei der X ein Startzustand des Automaten ist
7. Erzeugen einer Produktion $S \rightarrow y$ für jede Transformation $\delta(X, y) = Z$ bei der X ein Startzustand und Z ein Endzustand des Automaten ist

Beide Verfahren erzeugen wieder nur eine von beliebig vielen äquivalenten Grammatiken und stellen somit keine eindeutig richtige Lösung dar.

3.4.3. Fehlerquellen

Auch die in diesem Abschnitt vorgestellten Verfahren sind vergleichsweise einfach zu verstehen und anzuwenden. Somit besitzen wieder Flüchtigkeitsfehler die größte Wahrscheinlichkeit.

Auch bei diesem Verfahren besteht wieder die Gefahr der systematischen fehlerhaften Umwandlung von Transitionen in Produktionen, also der Umwandlung einer Transition $\delta(X, y) = Z$ in eine Produktion $Z \rightarrow yX$. Beim Sonderfall der Produktionen der Form $X \rightarrow y$ ist denkbar, dass ein Student diese für alle Transitionen der Form $\delta(X, y) = Z$, bei denen X ein Endzustand ist anstatt Z, erzeugt

Beim nichtdeterministischen Sonderfall könnte entweder die Berücksichtigung mehrere Startzustände komplett ignoriert werden, das heißt der Student macht einfach irgendeinen Startzustand zum Startsymbol und lässt die anderen unter den Tisch fallen oder aber er vergisst, eine der vier Umwandlungsregeln (Schritt 4. bis 7.) anzuwenden.

3.4.4. Fehlerkategorien

Im Bereich der Flüchtigkeitsfehler kann man die im vorherigen Abschnitt ermittelten Kategorien der *Verwechslung*, *Nachlässigkeit* und *Unvollständigkeit* auch für dieses Verfahren übernehmen. Auch die Kategorie der *Falschen Umwandlung* kann hier in leicht abgewandelter Form auftreten. Dies schließt die falsche Behandlung von Endzuständen bei der Umwandlung mit ein.

Folgende Kategorien könnten hier zusätzlich auftreten:

1. *Endzustände*: Der Student berücksichtigt bei der Erzeugung von Produktionen, die in Terminale ableiten, den falschen Zustand in einer Transition

2. *Startzustände*: Der Student ignoriert Startzustände bei einem NFA mit mehr als einem Startzustand
3. *Fehlende Fälle*: Der Student vergisst einen oder mehrere der vier Fälle bei der Umwandlung der Transitionen eines NFA

Alle drei neuen Fehlerkategorien sollten in den meisten Fällen systematische Fehler sein. Sie zeugen von einem oder mehreren Missverständnissen des Studenten bezüglich des Stoffes.

3.5. NFA in DFA

Gegeben ist ein nichtdeterministischer endlicher Automat M der die Sprache L akzeptiert. Wandeln Sie diesen in eine deterministischen Automaten N um, welcher die selbe Sprache akzeptiert. Der NFA ist gegeben als M = (Z,Σ,δ,S,E). Dabei ist Z die Menge der Zustände, Σ die Menge des Eingabealphabets, δ die Überführungsfunktion, S die Menge der Startzustände und E die Menge der Endzustände. Der DFA hat die Form als N = (Z,Σ,δ,S,E). Dabei ist Z die Menge der Zustände, Σ die Menge des Eingabealphabets, δ die Überführungsfunktion, S der Startzustand und E die Menge der Endzustände.

3.5.1. Automatisierte Lösung

Für diese Umwandlung lässt sich eine Potenzmengenkonstruktion verwenden. Die Zustandsmenge des DFA ist eine Teilmenge der Potenzmenge der Zustände des NFA. Informell beschrieben gibt ein Zustand des DFA die Menge der alternativen Zustände des NFA an, in denen sich dieser beim Abarbeiten eines Wortes befinden könnte. Da die Potenzmenge der Zustände schon bei kleiner Anzahl von Zuständen des NFA recht groß sein kann, macht es Sinn nur die Zustandsmengen zu berücksichtigen, die von den Startzuständen aus erreichbar sind.

Für die folgenden logischen Regeln die aus einem NFA einen äquivalenten DFA erzeugen, werden wieder die in den vorherigen Abschnitten definierten Prädikate und Einschränkungen für die entsprechenden Automaten verwendet:

$$stat_D(M(M), Z(M))$$

$$alph_N(M, \Sigma) \rightarrow alph_D(M(M), \Sigma)$$

$$start_N(M, S) \rightarrow S \in Z(M)$$

$$\emptyset \in Z(M) \rightarrow trans_D(T(M), \emptyset, x, \emptyset)$$

$$trans_N(M, x, y, z) \rightarrow trans_D(T(M), \{x\}, y, z)$$

$$trans_N(M, x, y, z) \wedge trans_D(N, u, y, v) \rightarrow trans_D(T(M), u \cup \{x\}, y, z \cup v)$$

$$x \in Z(M) \wedge trans_D(T(M), x, y, z) \rightarrow z \in Z(M) \wedge trans_D(M(M), x, y, z)$$

$$start_N(M, S) \rightarrow start_D(M(M), S)$$

$$x \in Z(M) \wedge end_N(M, E) \wedge y \in E \wedge y \in x \rightarrow end_D(M(M), x)$$

Die gegebenen Regeln benötigen zum Verständnis vielleicht einige Erklärungen. Auffällig ist beispielsweise das Auftauchen von Transitionen eines DFA T(M). Dabei handelt es sich um quasi temporäre Transitionen, die nicht unbedingt in den Zielautomaten übernommen werden. Der Grund für ihr Entstehen ist die rekursive Weise, mit der Transitionen für Zustandsmengen erzeugt werden.

Um die Transitionen für eine Menge der Form $X \cup \{y\}$ zu bestimmen wird zuerst das Ergebnis der ursprünglichen Transition des DFA bei Eingabe des Zustands y ermittelt und danach das Ergebnis mit demjenigen der Transition mit der Eingabe der Menge X vereinigt. Dies wird so lange durchgeführt bis die Menge X aus genau einem Element besteht, wonach das Ergebnis wieder auf die ursprüngliche Transition zurückgeführt werden kann.

Als Nebeneffekt werden dabei nicht nur Transitionen für die erreichbaren Zustandsmengen erzeugt sondern auch alle Teilmengen dieser. Um diese nicht benötigten Transitionen loszuwerden, werden diese dem Automaten T(M) zugeordnet und nur im Falle der erreichbaren Zustandsmengen auch für den Zielautomat N abgeleitet.

Mit Hilfe der gegebenen Regeln lassen sich nun also alle den DFA definierenden Prädikate ableiten beziehungsweise der folgende Satz beweisen:

$$\exists M(M).\exists Z(M).\mathit{Definition}_M^N(M(M), Z(M))$$

Dieses Beispiel zeigt jedoch auch gewisse Schwächen des gewählten Ansatzes. Während andere Implementierungen des Algorithmus im Ansatz sehr systematisch vorgehen können, hat man bei Verwendung eines automatischen Beweisers mit den gegebenen Regeln keine wirkliche Kontrolle über die Reihenfolge der Abarbeitung.

Ein Beispiel für dieses Problem ist die Art der Erzeugung der Transitionen für Zustandsmengen. Dabei werden unter Umständen Transitionen für viele Teilmengen erzeugt die nie verwendet werden.

3.5.2. Strukturierung

1. Einfügen der Menge der Startzustände des NFA als erstes Element der Zustände des DFA
2. Für alle unbearbeiteten Zustände des DFA die Transitionen für alle Eingaben erzeugen
3. Einfügen aller in Schritt 2 erreichten neuen Zustandsmengen in die Zustandsmenge des DFA
4. Falls noch unbearbeitete Zustände vorhanden zu Schritt 2, sonst fortfahren
5. Menge der Startzustände des NFA ist der Startzustand des DFA
6. Jeden Zustand des DFA, der mindestens einen der Endzustände des NFA enthält, in die Menge der Endzustände des DFA einfügen

Im Gegensatz zu den bisher vorgestellten Verfahren ist dieses aus systematischen

Gründen nicht völlig frei in der Ausführungsreihenfolge. Die Schritte 2. bis 4. müssen in genau dieser Reihenfolge ausgeführt werden, da sie eine Schleife beschreiben.

Die Ursache dafür ist die Tatsache, dass dieses Verfahren nur die wirklich notwendigen Zustände und Transitionen erzeugen soll, nämlich diejenigen, die vom Startzustand auch erreichbar sind. Aus diesem Grund ist es natürlich nötig, vom Startzustand ausgehend iterativ alle erreichbaren Zustände und die dazu gehörenden Transitionen zu erzeugen.

3.5.3. Fehlerquellen

Neben den unvermeidlichen Flüchtigkeitsfehlern existieren einige für das Verfahren spezifische Fehler. Zunächst besteht wie bei allen Verfahren, bei denen etwas nach einem bestimmten Schema umgeformt wird, das Risiko, dass man das Schema systematisch falsch anwendet.

Da es sich beim Erzeugen der Transitionen über Zustandsmengen um eine vergleichsweise komplizierte Umwandlung handelt, neigen Studenten leicht zu Missverständnissen bei der Umsetzung. Da man in vielen Fällen mehrere Zustandsübergänge des Ursprungsautomaten gleichzeitig berücksichtigen muss, führt dies auch zu einer erhöhten Gefahr etwas zu übersehen oder systematisch falsch zu machen.

Auch die Definition der Endzustände des erzeugten DFA führt leicht zu dem Missverständnis, dass man nicht alle Zustandsmengen, die mindestens ein Element mit der Menge der Endzuständen des NFA gemeinsam haben, zu Endzuständen des DFA macht, sondern nur diejenigen Zustandsmengen, die nur Endzustände des DFA enthalten.

Ein letzter Fehler, der für dieses Verfahren spezifisch ist, ist das Ignorieren der leeren Zustandsmenge bei fehlenden Übergängen für bestimmte Eingabezeichen. Dieser wird oft auch als Fehlerzustand bezeichnet. Auch wenn diese Vernachlässigung auf den ersten Blick unwichtig erscheint, so ist sie zum Erzeugen eines wirklich Deterministischen Automaten mit totaler Übergangsfunktion notwendig.

3.5.4. Fehlerkategorien

Wie schon zuvor existieren die universellen Fehlerkategorien die auf Flüchtigkeitsfehler zurückzuführen sind, auch bei diesem Verfahren. Auch die Kategorie des *Falschen Umwandlung* von Transitionen besteht hier. Sie ist sogar vermutlich deutlich häufiger, da die Umwandlung hier komplexer ist als in den vorhergehenden Verfahren

Die neuen und spezifischen Fehlerkategorien für dieses Problem sind:

1. *Endzustände*: Der Student wählt die Endzustände des DFA falsch aus. Zum Beispiel nur diejenigen, die vollständig aus Endzuständen des NFA bestehen.
2. *Fehlerzustand*: Der Student ignoriert die leere Zustandsmenge bei fehlenden Übergängen.

Bei beiden neuen Kategorien handelt es sich in der Regel um systematische Fehler.

3.6. Regulärer Ausdruck in NFA

Gegeben ist ein regulärer Ausdruck α der die Sprache L erzeugt. Wandeln Sie diesen in einen nichtdeterministischen endlichen Automaten M um, der die selbe Sprache akzeptiert. Der NFA hat die Form M = (Z,Σ,δ,S,E). Dabei ist Z die Menge der Zustände, Σ die Menge des Eingabealphabets, δ die Überführungsfunktion, S ist die Menge der Startzustände und E die Menge der Endzustände.

3.6.1. Automatisierung

Um diese Umwandlung zu beschreiben, wird zunächst eine geeignete logische Repräsentation für reguläre Ausdrücke benötigt. Folgende Prädikate sollen die rekursive Struktur eines Ausdrucks in Relation zu den Teilausdrücken beschreiben:

$leer_R(\alpha)$: Beschreibt den regulären Ausdruck α = ε

$term_R(\alpha, a)$: Beschreibt α = a

$oder_R(\alpha, \beta, \gamma)$: Beschreibt den regulären Ausdruck α = β | γ

$seq_R(\alpha, \beta, \gamma)$: Beschreibt α = β γ

$stern_R(\alpha, \beta)$: Beschreibt α = β*

Unter Verwendung dieser Darstellung lassen sich nun jeweils bestimmte NFA definieren, welche die Sprache des Ausdrucks oder eines Teilausdrucks akzeptieren. Dazu dienen folgende Regeln für die einzelnen regulären Operatoren:

$$leer_R(\alpha) \rightarrow stat_N(M(\alpha), \{S(M)\}) \wedge alph_N(M(\alpha), \emptyset)$$

$$leer_R(\alpha) \rightarrow start_M(\alpha, \{S(M)\})\ end_M(\alpha, \{S(M)\})$$

$$term_R(\alpha, a) \rightarrow stat_N(M(\alpha), \{S(M), E(M)\}) \wedge alph_N(M(\alpha), \{a\})$$

$$term_R(\alpha, a) \rightarrow trans_N(M(\alpha), S(M), a, \{E(M)\})$$

$$term_R(\alpha, a) \rightarrow start_N(M(\alpha), \{S(M)\}) \wedge end_N(M(\alpha), \{E(M)\})$$

$$oder_R(\alpha, \beta, \gamma) \wedge stat_N(M(\beta), x) \wedge stat_N(M(\gamma), y) \rightarrow stat_N(M(\alpha), x \cup y)$$

$$oder_R(\alpha, \beta, \gamma) \wedge alph_N(M(\beta), x)\ alph_N(M(\gamma), y) \rightarrow alph_N(M(\alpha), x \cup y)$$

$$oder_R(\alpha, \beta, \gamma) \wedge trans_N(M(\beta), x, y, z) \rightarrow trans_N(M(\alpha), x, y, z)$$

$$oder_R(\alpha, \beta, \gamma) \wedge trans_N(M(\gamma), x, y, z) \rightarrow trans_N(M(\alpha), x, y, z)$$

$$oder_R(\alpha, \beta, \gamma) \wedge start_N(M(\beta), x) \wedge start_N(M(\gamma), y) \rightarrow start_N(M(\alpha), x \cup y)$$

$$oder_R(\alpha, \beta, \gamma) \wedge end_N(M(\beta), x) \wedge end_N(M(\gamma), y) \rightarrow end_N(M(\alpha), x \cup y)$$

$$seq_R(\alpha, \beta, \gamma) \wedge stat_N(M(\beta), x) \wedge stat_N(M(\gamma), y) \rightarrow stat_N(M(\alpha), x \cup y)$$

$$seq_R(\alpha, \beta, \gamma) \wedge alph_N(M(\beta), x) \wedge alph_N(M(\gamma), y) \rightarrow alph_N(M(\alpha), x \cup y)$$

$$seq_R(\alpha, \beta, \gamma) \wedge trans_N(M(\beta), x, y, z) \rightarrow trans_N(M(\alpha), x, y, z)$$

$$seq_R(\alpha,\beta,\gamma) \wedge trans_N(M(\gamma),x,y,z) \rightarrow trans_N(M(\alpha),x,y,z)$$

$$seq_R(\alpha,\beta,\gamma) \wedge trans_N(M(\beta),x,y,z) \wedge end_R(M(\gamma),u) \wedge start_R(M(\gamma),v) \wedge z \in U \text{ und } w \in v \rightarrow trans_N(M(\alpha),x,y,w)$$

$$seq_R(\alpha,\beta,\gamma) \wedge start_N(M(\beta),x) \wedge end_N(M(\beta),y) \wedge x \cap y = \emptyset \rightarrow start_N(M(\alpha),x)$$

$$seq_R(\alpha,\beta,\gamma) \wedge start_N(M(\beta),x) \wedge end_N(M(\beta),y) \wedge x \cap y \neq \emptyset \wedge start_N(M(\gamma),z) \rightarrow start_N(M(\alpha),x \cup z)$$

$$seq_R(\alpha,\beta,\gamma) \wedge start_N(M(\gamma),x) \wedge end_N(M(\gamma),y) \wedge x \cap y = \emptyset \rightarrow end_N(M(\alpha),y)$$

$$seq_R(\alpha,\beta,\gamma) \wedge start_N(M(\gamma),x) \wedge end_N(M(\gamma),y) \wedge x \cap y \neq \emptyset \wedge end_N(M(\beta),z) \rightarrow start_N(M(\alpha),y \cup z)$$

$$stern_R(\alpha,\beta) \wedge stat_N(M(\beta),x) \rightarrow stat_N(M(\alpha),x \cup \{S(\beta)\})$$

$$stern_R(\alpha,\beta) \wedge alph_N(M(\beta),x) \rightarrow alph_N(M(\alpha),x)$$

$$stern_R(\alpha,\beta) \wedge trans_N(M(\beta),x,y,z) \rightarrow trans_N(M(\alpha),x,y,z)$$

$$stern_R(\alpha,\beta) \wedge trans_N(M(\beta),x,y,z)\ start_N(M(\beta),u) \wedge end_N(M(\beta),v) \wedge z \in v \wedge w \in u \rightarrow trans_N(M(\alpha),x,y,w)$$

$$stern_R(\alpha,\beta) \wedge start_N(M(\beta),x) \rightarrow start_N(M(\alpha),x \cup \{S(\beta)\})$$

$$stern_R(\alpha,\beta) \wedge end_N(M(\beta),x) \rightarrow end_N(M(\alpha),x \cup \{S(\beta)\})$$

Auf den ersten Blick fällt auf, dass die Anzahl der Regeln für die Umwandlung eines regulären Ausdrucks in einen Automaten deutlich größer ist als bei den bisherigen Transformationen. Eine Ursache dafür ist sicher die gewählte logische Repräsentation. Aber auch die Tatsache, dass jeder der fünf in Regulären Ausdrücken Operatoren getrennt behandelt werden muss, trägt zur Fülle der Regeln bei. Insgesamt handelt es sich um ein vergleichsweise einfaches Verfahren, welches nur durch die große Anzahl der Fallunterscheidung einen gewissen Aufwand in der logischen Implementierung benötigt.

$$\exists M(\alpha).\, Definition_N^E(M(\alpha))$$

Der oben stehende Satz ist für die Erzeugung eines NFA M(α) aus dem Regulären Ausdruck zu beweisen.

3.6.2. Strukturierung

Da reguläre Ausdrücke selbst einen rekursiven Aufbau haben, ist auch der Algorithmus zur Umwandlung in einen NFA rekursiv.

1. *Fall 1*: Für einen Ausdruck der Form α = ε erzeuge einen NFA, der nur das leere Wort akzeptiert

2. *Fall 2*: Für einen Ausdruck der Form α = a erzeuge einen NFA, der nur das Wort a akzeptiert
3. *Fall 3*: Für einen Ausdruck der Form α = β | γ erzeuge aus den NFA für die Ausdrücke β und γ den *Vereinigungsautomaten*
4. *Fall 4*: Für einen Ausdruck der Form α = β γ erzeuge aus den NFA für die Ausdrücke β und γ den *Sequenzautomaten*
5. *Fall 5*: Für einen Ausdruck der Form α = β* erzeuge aus dem NFA für den Ausdruck β einen *Schleifenautomaten*
6. Wiederhole die entsprechenden Schritte so lange rekursiv, bis ein Automat für den Gesamtausdruck erzeugt wurde

Für den genauen Algorithmus, der sich hinter dem Begriffen der Vereinigungs-, Sequenz- und Schleifenautomaten verbirgt, sei nur auf die im vorigen Abschnitt formulierten logischen Regeln bzw. die ausführliche Beschreibungen des Verfahrens in [Schö97] und [HMU01] verwiesen.

3.6.3. Fehlerquellen

Bei diesem rekursiven Verfahren besteht zunächst natürlich die Gefahr, dass ein Student die Rekursion nicht richtig oder unvollständig ausführt. Zusätzlich besteht für jede Teilumwandlung die Gefahr, sie systematisch falsch zu machen. Dies ist insbesondere beim 4. und 5. Fall der Konkatenation und des Stern-Operators der Fall. Schließlich macht der große Aufwand, zumindest "auf dem Papier", Flüchtigkeitsfehler sehr wahrscheinlich.

3.6.4. Fehlerkategorien

Zusammenfassend kann man für dieses Verfahren folgende spezifischen Fehlerkategorien angeben:

1. *Rekursion*: Der Student hat das Prinzip der Rekursion im allgemeinen oder für diese Aufgabe nicht richtig verstanden und angewendet.
2. *Sequenzautomat*: Der Student wendet das Verfahren des "Aneinanderhängens" falsch an oder vergisst Sonderfälle.
3. *Schleifenautomat*: Der Student wendet das Verfahren falsch oder unvollständig an.

3.7. DFA in Regulärer Ausdruck

Gegeben ist NFA M der die Sprache L akzeptiert. Wandeln Sie diesen in einen regulären Ausdruck α um, welcher die selbe Sprache erzeugt. Der NFA ist gegeben als M = (Z,Σ,δ,S,E). Dabei ist Z die Menge der Zustände, Σ die Menge des Eingabealphabets, δ die Überführungsfunktion, S ist die Menge der Startzustände und E die Menge der Endzustände.

3.7.1. Automatisierung

Auch dieses Verfahren ist bei der Umsetzung in eine logische Form sehr aufwändig. Es erfordert zusätzlich die Behandlung von natürlichen Zahlen, was die Implementierung zusätzlich erschwert. Aus diesen Gründen soll an dieser Stelle auf eine weitere lange Aufzählung von Regeln zur Lösung des Problems verzichtet werden.

3.7.2. Strukturierung

Auch bei diesem Verfahren handelt es sich um ein rekursives Verfahren. Dabei werden zunächst die Zustände ausgehend vom Startzustand durchnummeriert und bestimmte Teilsprachen $R^k_{i,j}$ der akzeptierten Sprache betrachtet. Sie beinhalten jeweils die Menge aller Wörter die den Automaten vom Zustand z_i in den Zustand z_j überführen, wobei nur die Zustände $\{z_1 \dots z_k\}$ als Zwischenzustände verwendet werden. Jeder dieser Sprachen kann man einen regulären Ausdruck $\alpha^k_{i,j}$ zuordnen.

1. *Fall 1*: Falls k = 0 hat $\alpha^k_{i,j}$ die Form einer Disjunktion aller Terminale für die Übergänge $\delta(z_i, a) = z_j$ existieren sowie dem leeren Wort, falls i = j
2. *Fall 2*: Wenn k > 0 hat $\alpha^k_{i,j}$ die Form $\alpha^{k-1}_{i,j} / \alpha^{k-1}_{k,k} * \alpha^{k-1}_{n,e}$
3. Wähle einen Endzustand $z_e \in E$
4. Erzeuge den regulären Ausdruck $\alpha_e = \alpha^n_{1,e} = \alpha^{n-1}_{1,e} / \alpha^{n-1}_{n,n} * \alpha^{n-1}_{n,e}$ wobei n die Anzahl der Zustände des DFA ist rekursiv durch Erzeugung der Teilausdrücke nach obigen Fallunterscheidungen.
5. Führe die Schritte 4. und 5. so lange aus bis jeder Endzustand bearbeitet wurde
6. Erzeuge den regulären Ausdruck α als Disjunktion der einzelnen Ausdrücke für jeden Endzustand

Strukturell handelt es sich bei diesem Verfahren also um mehrere Rekursionen in einer Schleife.

3.7.3. Fehlerquellen

Wie schon bei der Umwandlung regulärer Ausdrücke in NFA könnte die verwendete Rekursion problematisch sein. Selbst bei korrekter Ausführung der Rekursion, könnte der 2. Fall leicht systematisch falsch angewendet werden. Denkbar ist auch, dass zwar die einzelnen Ausdrücke korrekt auf die entsprechenden Teilausdrücke zurückgeführt werden, aber mit dem Falschen begonnen wird. So könnte das Missverständnis entstehen, der vollständige Ausdruck ergibt sich aus $\alpha^n_{1,n}$ wenn n die Anzahl der Zustände ist, was einer Betrachtung des Zustands mit dem höchsten Index als einzigem Endzustand gleichkäme.

3.7.4. Fehlerkategorien

Zusammengefasst könnten bei der gegebenen Aufgabe die folgenden Fehlerkategorien auftreten:

1. *Rekursion*: Der Student setzt die Rekursion nicht korrekt um.
2. *Falsche Umwandlung*: Der Student betrachtet die falschen Teilausdrücke oder vergisst Teile davon.
3. *Falscher Einstieg*: Der Student steigt falsch in die Rekursion ein.

3.8. DFA in minimalen DFA

Gegeben ist ein deterministischer endlicher Automat M, der die Sprache L akzeptiert. Wandeln sie diesen in eine deterministischen Automaten N um, welcher die selbe Sprache akzeptiert und eine minimale Anzahl von Zuständen besitzt. Der DFA ist gegeben als M = (Z,Σ,δ,S,E). Dabei ist Z die Menge der Zustände, Σ die Menge des Eingabealphabets, δ die Überführungsfunktion, S ist der Startzustand und E die Menge der Endzustände.

Diese Transformation schlägt etwas aus der Art, da sie nicht eine Repräsentation in eine andere umwandelt sondern einen DFA in einen äquivalenten mit der besonderen Eigenschaft, dass er eine minimale Anzahl von Zuständen besitzt.

3.8.1. Automatisierung

Der erste Schritt besteht darin, alle nicht vom Startzustand erreichbaren Zustände zu entfernen. Die folgenden Regeln erzeugen aus einem gegebenen DFA einen anderen, welcher nur diejenigen Zustände enthält die vom Startzustand erreichbar sind:

$$stat_D(M_1(M), Z_1(M))$$

$$alph_D(M, \Sigma) \rightarrow alph_D(M_1(M), \Sigma)$$

$$start_D(M, S) \rightarrow S \in Z_1(M) \wedge start_D(M_1(M), S)$$

$$end_D(M_1(M), E_1(E))$$

$$x \in Z_1(M) \wedge trans_D(M, x, y, z) \rightarrow z \in Z_1(M) \wedge trans_D(M_1(M), x, y, z)$$

$$x \in Z_1(M) \wedge end_D(M, E) \wedge x \in E \rightarrow x \in E_1(M)$$

Der nächste Schritt besteht nun darin, die noch verbliebenen Zustände auf Äquivalenz zu überprüfen. Dazu werden ausgehend von Paaren von Endzuständen und Nichtendzuständen iterativ alle Paare von Zuständen markiert, die nicht äquivalent sind. Dazu wird ein neues Prädikat $mark_{min}(x, y)$ eingeführt.

$$stat_D(M_1(M), Z) \wedge end_D(M_1(M), E) \wedge x \in Z \wedge x \notin E \wedge y \in E \rightarrow mark_{min}(x, y)$$

$$trans_D(M_1(M), x, y, z) \wedge trans_D(M_1(M), u, y, v) \wedge mark_{min}(z, v) \rightarrow mark_{min}(x, u)$$

$$mark_{min}(x, y) \rightarrow mark_{min}(y, x)$$

Abschließend muss nun noch ein Automat erzeugt werden, bei dem alle unmarkierten Paare zu einzelnen Zuständen zusammengefasst sind, da sie als äquivalent erkannt

wurden. Ein Problem ist, dass dabei weniger die Paare von Zuständen, die nicht äquivalent sind, von Interesse sind, sondern diejenigen, die es sind.

Es interessiert uns also gerade die Negation des abgeleiteten Markierungs-Prädikat. Da die Verarbeitung der Negation einige Probleme mit sich bringt, soll sich an dieser Stelle auf diese informelle Beschreibung beschränkt werden.

3.8.2. Strukturierung

1. Entfernen aller vom Startzustand aus nicht erreichbaren Zustände
2. Erzeugen einer Liste aller ungleichen Paare von Zuständen
3. Markieren aller Paare aus Endzuständen und Zuständen die kein Endzustand sind
4. Für alle noch unmarkierten Paare (X, Y) und alle Eingaben z überprüfen, ob das Paar $(\delta(X, z), \delta(Y, z))$ bereits markiert ist. Falls ja, dieses ebenfalls markieren
5. Schritt 4 so lange wiederholen, bis sich keine Veränderungen bei den Markierungen mehr ergeben
6. Erzeugen eines neuen Automaten bei dem alle noch unmarkierten Paare zu einem einzelnen Zustand zusammengefasst wurden

3.8.3. Fehlerquellen

Außer Flüchtigkeitsfehlern existieren für dieses Verfahren wieder eine Reihe von spezifischen Fehlern. Möglicherweise wird nur ein einziger Durchlauf der Paare durchgeführt und danach einfach das Verfahren fortgesetzt, wodurch natürlich meistens sehr viel mehr Paare als äquivalent betrachtet und dann zusammengefasst werden. Gleichzeitig besteht die Gefahr, dass zu einem unmarkierten Paar das falsche Folgepaar betrachtet wird, oder nicht für jedes mögliche Eingabezeichen verglichen wird. Schließlich kann der Fall, dass ein Folgepaar aus zwei identischen Zuständen besteht, welche natürlich in jedem Fall äquivalent und somit immer unmarkiert sind, falsch interpretiert werden.

3.8.4. Fehlerkategorien

Folgende spezifische Kategorien für Fehler existieren für das Minimierungs-Verfahren:

1. *Fehlende Wiederholung*: Es wird nur ein einziger Markierungsdurchlauf ausgeführt.
2. *Falsche Folgepaare*: Es werden nicht alle oder nicht die korrekten Folgepaare auf Markierung überprüft.
3. *Identische Folgepaare*: Paare die aus identischen Zuständen bestehen werden als markiert behandelt.

Zu diesen drei Fehlern kommen wie immer die üblichen Flüchtigkeitsfehler: *Unvollständigkeit*, *Verwechslung* und *Nachlässigkeit.*

3.9. Ergebnisse dieses Kapitels

In diesem Kapitel wurden bestimmte Aufgabenformen unter zwei Gesichtspunkten analysiert. Zum einen wurden automatische Verfahren zum Lösen der Aufgaben selbst betrachtet und der Versuch gemacht sie mittels der gewählten Methode des automatischen Beweisens zu implementieren. Zum anderen wurden sie unter dem Gesichtspunkt der möglichen Fehler durch Studenten beim Lösen von entsprechenden Aufgaben betrachtet.

Während die einfacheren Aufgaben, wie die Umwandlung von regulären Sprachen in Endliche Automaten und umgekehrt, relativ leicht mittels eines Beweisers gelöst werden können, war es bei den anderen Transformationen, denen geringfügig komplexere Algorithmen zugrunde lagen, schon deutlich schwieriger, dies auf eine elegante und effiziente Weise zu tun.

Ein großer Nachteil den der Beweiser-Ansatz besitzt ist die Tatsache, dass es nur sehr indirekt möglich ist, die Abarbeitungsreihenfolge zu steuern. Dies ist ein Problem, das in ähnlichem Maße auch die Sprache PROLOG oder ein Planungssystem aufwirft, und einen negativen Einfluss sowohl auf den Aufwand eine Lösungsmethode mit diesem Ansatz zu implementieren als auch auf die Effizienz in der Verarbeitung hat.

Ein weiteres grundlegendes Problem, welches sich bei der Gegenüberstellung einer automatisch generierten Aufgabenlösung ergeben hat, ist die Tatsache, dass die generierte Lösung nur dann von Nutzen bei der Analyse der studentischen Eingaben ist, wenn dieser einen zumindest vergleichbare Lösungsansatz verwendet. Es sollte aber in jedem Fall möglich sein, zumindest das Endergebnis des Studenten durch Abgleich mit der vom System erzeugten Lösung zu verifizieren.

Dies funktioniert jedoch nur über den Umweg der Generierung einer normalisierten Repräsentation wie dem minimalen DFA einer regulären Sprache. Ein besonders extremes Beispiel wäre der Vergleich zweier regulärer Ausdrucke auf Äquivalenz. Dazu würde das System beide zunächst in äquivalente NFA und diese wiederum in äquivalente DFA umwandeln. Danach würden beide DFA noch minimiert, um sie dann zu vergleichen, wobei auch noch das Problem unterschiedlicher Zustandsnamen berücksichtigt werden müsste. Letztendlich sollte das System, eine effiziente Implementierung der entsprechenden Transformationen vorausgesetzt, aber in der Lage sein, einen solchen Vergleich zügig durchzuführen.

Was die Analyse möglicher Fehler bei Anwendung der betrachteten Lösungsverfahren angeht, konnten einige klassische Fehlerquellen identifiziert werden. Diese lassen sich zum einen unterscheiden nach allgemeinen Fehlern, die bei allen Aufgabentypen auftreten können, dies betrifft insbesondere die verschiedenen Formen von Flüchtigkeitsfehlern, und zum anderen für das Lösungsverfahren spezifische Fehler.

Andererseits sollte die Unterscheidung zwischen einfachen und systematischen Fehlern eine wichtige Rolle spielen. Dabei können bestimmte, rein äußerlich identische Fehler, sowohl einfache als auch systematische Fehler sein, und es ist nicht immer möglich, unmittelbar beim ersten Auftreten zu entscheiden, ob es sich tatsächlich um einen systematischen Fehler handelt. Aus pädagogischer Sicht kann die klare Unterscheidung

zwischen diesen beiden Fehlerarten sehr wichtig sein, da ein systematischer Fehler ein Hinweis darauf ist, dass ein Student etwas Falsches gelernt hat.

Folgende Tabelle fasst noch einmal alle ermittelten Fehlerkategorien nach Aufgabentyp zusammen:

Aufgabentyp	***Fehlerkategorien***
Reg. Grammatik in NFA	Falsche Umwandlung
DFA (NFA) in Grammatik	Falsche Umwandlung, Endzustände (nur NFA), Fehlende Fälle (NFA)
NFA in DFA	Falsche Umwandlung, Endzustände, Fehlerzustand
Reg. Ausdruck in NFA	Rekursion, Sequenzautomat, Schleifenautomat
NFA in Reg. Ausdruck	Rekursion, Falsche Umwandlung, Falscher Einstieg
Minimierung DFA	Fehlende Wiederholung, Falsche Folgepaare, Identische Folgepaare
Gemeinsame Kategorien	Nachlässigkeit, Unvollständigkeit, Verwechslung

Tabelle 1: Fehlerkategorien nach Aufgabentypen

Natürlich erhebt diese Tabelle beziehungsweise dieses Kapitel nicht den Anspruch auf Vollständigkeit. Es sind vermutlich für jedes Verfahren eine Reihe weiterer spezifischer Fehler denkbar.

4. Interaktion mit dem Studenten

4.1. Ziele der Interaktion

Während im vorherigen Kapitel das Augenmerk auf den zu stellenden Aufgaben und deren Verarbeitung durch das System lag, wurde ein das System verwendender Student nur unter Berücksichtigung der Fehler die er machen kann, betrachtet. Bevor ein Student aber überhaupt in der Lage ist, für das System erkennbare Fehler zu produzieren, was natürlich einerseits nicht direkt erwünscht ist, aber andererseits erfüllt das System für einen Studenten, der immer alles richtig macht, keinen besonderes großen Zweck, muss dieses überhaupt einmal die Fähigkeit besitzen, mit ihm auf sinnvolle Weise zu interagieren.

Zu diesem Zweck soll sich dieses Kapitel mit der Fragestellung beschäftigen, was eine geeignete Weise der Kommunikation des Systems mit dem Studenten ist. Im einfachsten Fall präsentiert das System dem Studenten eine Aufgabe und nimmt dann dessen vollständige Lösung in Empfang, deren Korrektheit es dann zu überprüfen versucht. Im Idealfall beobachtet und analysiert das System sämtliche Eingaben des Studenten noch während dieser die Aufgabe bearbeitet und reagiert und interagiert dabei dynamisch mit dem Studenten, um eine bestmögliche Betreuung zu gewährleisten.

Beide Situationen stellen Extreme dar, zwischen denen sich ein reales System positionieren muss. Dabei gilt es, das Erwünschte und Machbare so gut wie möglich in Einklang zu bringen.

Eine weitere wichtige Rolle spielt die Form der Eingabe. In der theoretischen Informatik befasst man sich mit den unterschiedlichsten Objekten. Ein gutes Beispiel sind hierbei die unterschiedlichen Formen von Automaten. Diese lassen sich zum einen natürlich in textueller Form darstellen aber zum anderen kann man sie vollständig oder zumindest zum größten Teil in Form von Graphen darstellen. Getreu dem Motto "Ein Bild sagt mehr als tausend Worte" werden sie dabei in vielen Fällen auch deutlich leichter zugänglich gemacht. Sie lassen sich, eine effiziente Eingabemethode vorausgesetzt, auch wesentlich schneller produzieren beziehungsweise verändern.

4.2. Kommunikation

Bevor man sich mit der genauen Form der Kommunikation mit dem Studenten, als Anwender des Systems beschäftigen kann, sollte man klären, welche Informationen genau zwischen beiden Parteien ausgetauscht werden sollen. Zu diesem Zweck soll nun geklärt werden, was genau das System vom Studenten benötigt und umgekehrt.

Da das Interesse des Systems auf dem Verarbeiten einer vom Studenten angefertigten Lösung liegt, sollte das System zu jedem Zeitpunkt folgende Dinge wissen:

- *Was macht der Student?* Zum Beispiel: Der Student beschreibt einen NFA.
- *Wozu macht er es?* Beispiel: Er benötigt einen zu einer regulären Grammatik äquivalenten Automaten zur Lösung der Aufgabe.

- *Wie macht er es?* Er notiert alle zur Definition des Automaten notwendigen Teile in Textform.

Die Antworten müssen zu einem gegebenen Zeitpunkt während der Interaktion nicht immer eindeutig sein. Insbesondere spielt dabei die betrachtete Handlungsebene eine Rolle. Auf die Frage, was der Student macht, kann zum Beispiel auf mehreren Ebenen wie folgt geantwortet werden:

1. Er übt sich in der Theoretischen Informatik.
2. Er bearbeitet eine Aufgabe in der er zu einer regulären Grammatik einen minimalen DFA für die selbe Sprache ermitteln soll.
3. Er beschreibt einen NFA.
4. Er notiert die Überleitungsfunktion δ .
5. Er beschreibt den Übergang $\delta(Z_3, c)=\{Z_0, Z4\}$.
6. Er gibt den Buchstaben c ein.

Während die Frage *was* der Student macht, wie oben illustriert, in den meisten Fällen für das System anhand der Eingaben erkennbar sein sollte beziehungsweise ein direktes Ergebnis dessen ist, was das System zuvor getan hat, ergeben sich die Antworten auf die Frage nach dem *wie* und *wozu* eher selten auf einfache Weise aus den Eingaben des Studenten. In den meisten Fällen lassen sich diese durch das System nur dann eindeutig beantworten, wenn der Student es dem System ausdrücklich mitteilt.

Aus diesem Grund sollte eine Methode gefunden werden, dies dem Studenten zu ermöglichen. Während es bei der Bearbeitung einer Lösung durch einen Menschen auf einfache Weise mittels natürlichsprachlicher Anmerkungen zum gewählten Lösungsweg möglich ist, ist ein Computersystem damit sehr schnell überfordert. Das Verstehen natürlicher Sprache durch einen Rechner würde den Rahmen dieses Systems oder zumindest dieser Arbeit deutlich sprengen. Diese Problematik soll in späteren Abschnitten vertieft werden.

Während nun beschrieben wurde, was das System vom Studenten wissen muss, sollte nun auch die andere Seite betrachtet werden. Was genau erwartet ein Student von dem System. Welche Informationen und welches Feedback ist erwünscht? Hierbei gibt es eine Vielzahl von Fragen, die sich ein Student während der Bearbeitung stellen könnte:

- Habe ich den letzten Schritt richtig gemacht?
- Bin ich auf dem richtigen Weg?
- Was muss ich als nächstes machen?
- Wie kann ich die Aufgabe lösen?
- Was ist die Lösung?

Natürlich sind die genannten Fragen nur eine kleine Auswahl dessen, was sich ein Student bei der Arbeit mit dem System fragen könnte, aber die Stoßrichtung sollte klar sein. Der Student erhofft sich vom System konkrete Unterstützung beim Lösen der

Aufgabe, wenn er nicht selbst dazu in der Lage ist oder sich seiner Sache nicht sicher ist.

Der Grad der Unterstützung, die ein Student anfordern wird, ist natürlich sehr individuell. Ein guter Student wird sich eher darauf beschränken, die Aufgabe auf die Weise zu lösen, die er selbst für die richtige hält und erst am Schluss eine Bestätigung der Richtigkeit seines Tuns vom System fordern. Es wurde aber an anderer Stelle bereits darauf hingewiesen, dass sich das System nicht wirklich an den perfekten Studenten, der alles weiß und immer alles richtig macht, wendet, sondern vor allen Dingen ein Werkzeug für Studenten darstellen soll, die den Stoff zumindest teilweise noch erlernen und üben wollen.

Deshalb sollte man in der Kommunikation mit dem Studenten die typischsten Formen der Fragestellungen berücksichtigen und das System von Anfang an mit entsprechenden Möglichkeiten ausstatten. Die Fragestellung ist nun also, in welcher Form die oben angedeuteten Fragen vom Studenten an das System gestellt werden können. Auch dies soll in späteren Abschnitten vertieft werden.

4.3. Studentenprofile

Eine wichtiger Aspekt jeder effizienten Kommunikation zwischen zwei Parteien, ist eine korrekte Einschätzung des Kommunikationspartners. Das System soll mit besonders vielen Arten von Studenten, die sich in ihrem Wissen, ihrer Arbeitsweise und ihren Ansprüchen an das System stark unterscheiden, umgehen können.

Deshalb soll nun der Versuch gemacht werden, Profile typischer Arten von Studenten zu finden, die später als Grundlage genauerer Überlegungen dienen sollen. Dabei stellt sich zunächst die Frage, welche Eigenschaften eines Studenten dabei eine Rolle spielen und bei der Auswahl der Profile betrachtet werden sollen. Folgende Eigenschaften bieten sich an:

1. Wie vertraut ist der Student mit dem Stoff?
2. Wie selbstsicher ist der Student in Bezug auf das Themengebiet?
3. Wie selbstständig ist der Student beim Entwickeln von Lösungen?
4. Wie gründlich ist die Arbeitsweise des Studenten?
5. Was erwartet der Student von der Arbeit mit dem System?

Unter Berücksichtigung dieser fünf Eigenschaften, welche sozusagen die Parameter eines Studenten beschreiben, sind nun einige Studententypen möglich. Die folgenden Typen erheben nicht den Anspruch besonders realistische Charaktere zu sein und können mitunter leicht überzeichnet wirken. Sie stellen im wesentlichen die Eckpfeiler dar, zwischen denen sich das Spektrum der realen Studenten aufspannen könnte.

4.3.1. Der Musterstudent

Dieser Student ist sehr gut mit dem Stoff vertraut und verhältnismäßig selbstsicher. Beim Lösen von Aufgaben ist er nicht sehr selbstständig, sondern hält sich, sofern

möglich, an die üblichen Lösungswege. Seine Arbeitsweise ist sehr gründlich und er erwartet vor allen Dingen Bestätigung vom System.

Aus Sicht des Systems ist der Musterstudent vermutlich sehr leicht zufrieden zu stellen, da er durch seine Selbstsicherheit und sein gutes Fachwissen sehr wenige Fehler macht und sehr wenige Anfragen an das System stellt. Seine Neigung zu Standardlösungen macht ihn sehr vorhersehbar und damit die von ihm angefertigten Lösungen verhältnismäßig gut analysierbar.

4.3.2. Der Überflieger

Dieser Student ist gut mit dem Stoff vertraut und sehr selbstsicher. Er ist sehr selbstständig und neigt zu eleganten aber oft ungewöhnlichen Lösungswegen. Er ist halbwegs gründlich in seiner Arbeitsweise und betrachte das System als Zeitvertreib.

Der Überflieger wirft für das System eine Vielzahl von Schwierigkeiten auf. Während seine Vertrautheit mit dem Stoff dazu führt, dass er eher seltener systematische Fehler begeht, ist er nicht gründlich genug um nicht zu Flüchtigkeitsfehlern zu neigen. Sein Hang zu kreativen und eigenständigen Lösungen macht es dem System sehr schwer, seine Lösungen in allen Fällen nachzuvollziehen.

4.3.3. Der Zweifler

Dieser Student ist zwar relativ fleißig und mit dem Stoff vertraut, aber ist gleichzeitig sehr unsicher und hat sehr geringes Vertrauen in seine eigenen Fähigkeiten. Er ist sehr unselbständig und versucht immer alles nach Vorschrift zu lösen. Seine Nervosität beeinträchtigt seine Gründlichkeit leicht und seine er erwartet vom System sehr intensive Betreuung.

Aufgrund seiner Unsicherheit wird der Zweifler sehr häufig Anfragen an das System stellen, was einen hohen Grad an Interaktionsmöglichkeiten erfordert. Ein Nebeneffekt davon ist, dass der Zweifler dadurch relativ leicht in die für das System gewünschten Bahnen zu lenken ist, was die Analyse seiner Lösungswege erleichtert. Infolge der engen Zusammenarbeit mit dem System, seiner guten Kenntnis des Stoffes sowie seiner verhältnismäßigen Gründlichkeit, wird dieser Student relativ wenige Fehler produzieren.

4.3.4. Der Nachzügler

Dieser Student hat nur durchschnittliche Kenntnis des Stoffes und ist entsprechend selbstsicher. Er ist eigentlich eher selbstständig und versucht diesen Rückstand durch sehr hohe Gründlichkeit auszugleichen. Er erwartet vom System Unterstützung dabei, sich das ihm fehlende Wissen an zueignen.

Der Nachzügler wird in vielen Fällen zwar nicht in der Lage sein, eine gestellte Aufgabe selbstständig zu lösen, besitzt aber aufgrund seiner Gründlichkeit und der Bereitschaft zu lernen, viel Potential, die vom System angeforderte Hilfestellungen auch umzusetzen beziehungsweise aus erkannten Fehlern zu lernen.

4.3.5. Der Autodidakt

Dieser Student hat zwar nur sehr begrenzte Kenntnisse des Stoffes, ist aber einigermaßen selbstsicher und hat Vertrauen in seine Fähigkeit zu lernen. Er ist sehr selbstständig und halbwegs gründlich. Er erhofft sich vom System Unterstützung beim selbstständigen Lernen.

Auch wenn mangelnde Fachkenntnis es dem Autodidakten schwer machen, Aufgaben selbstständig zu lösen, ist er bei geeigneten Interaktionsmöglichkeiten gut dazu in der Lage in Kooperation mit dem System zur erfolgreichen Lösung einer Aufgabe zu kommen. In den Fällen, bei denen sein Fachwissen bereits ausreicht, neigt der Autodidakt zu eher kreativen Lösungen, was dem System Schwierigkeiten bereiten könnte.

4.3.6. Der Eilige

Dieser Student ist nur relativ flüchtig mit dem Stoff vertraut, ist aber trotzdem vergleichsweise selbstsicher. Er ist weder besonders selbstständig noch richtig unselbstständig. Er versucht, möglichst wenig Zeit zu investieren und ist daher sehr ungründlich. Er erwartet vom System vor allem schnelle Ergebnisse.

Für das System stellt der Eilige eine besondere Herausforderung dar, da seine Fähigkeiten zur Aufgabenlösung in starkem Gegensatz zu seinem Selbstbild stehen. Wenn etwas unklar ist, wird er oft Anfragen an das System stellen und wenn er davon ausgeht die Lösung zu wissen, wird er versuchen diese so schnell wie möglich umzusetzen, wodurch er viele Fehler macht.

4.3.6.1. Der hoffnungslose Fall

Dieser Student kennt den Stoff kaum, ist sich dessen aber auch bewusst und deshalb nicht sehr selbstsicher. Da die Grundlagen dazu fehlen, ist er auch sehr unselbstständig. In den wenigen Fällen, wo er in der Lage ist, eine gestellte Aufgabe zu lösen ist er aber relativ gründlich. Er erwartet vom System, die Grundlagen des Stoffes beigebracht zu bekommen.

Weil beim hoffnungslosen Fall eigentlich keine Basis vorhanden ist, auf der das System aufbauen kann, muss man bei ihm entweder mit sehr wenig oder aber sehr viel Interaktion rechnen. Diese wird in den meisten Fällen dazu führen, dass das System dazu verurteilt ist jeden einzelnen Schritt einer Lösung zu erklären und somit die Aufgabe gleichsam vor zurechnen.

4.3.7. Zusammenfassung

Neben den genannten sieben Studententypen sind sicher noch unzählige weitere Typen denkbar. Aber als Hilfsmittel auf dem Weg zu einer geeigneten Kommunikationsform zwischen System und Studenten sollten die vorgestellten mehr als ausreichen. Zur besseren Übersicht sind die sieben Studentenprofile zusammen mit einer Bewertung ihrer Eigenschaften in der folgenden Tabelle zusammengefasst, wobei die Erwartungen der einzelnen Typen an das System weggelassen wurden, da diese sich nicht in dieser Form bewerten lassen:

Studentenprofil	*Fachkenntnis*	*Selbstsicherheit*	*Selbstständigkeit*	*Gründlichkeit*
Musterstudent	++	+	--	++
Überflieger	++	++	++	0
Zweifler	+	--	-	0
Nachzügler	0	0	0	++
Autodidakt	-	+	++	+
Eilige	-	+	+	--
Hoffnungsloser Fall	--	--	--	+

Tabelle 2: Eigenschaften von Studentenprofilen

Diese sehr unterschiedlichen Studentenprofile illustrieren recht gut, wie vielfältig die Anforderungen an ein Coaching-System sind und wie schwer es ist, allen Arten gerecht zu werden. Man könnte nun versuchen, sich nur auf ausgewählte Profile zu konzentrieren und die problematischen einfach auszuschließen. So wäre ein System, das sich beispielsweise nur auf den bequemen Musterstudenten ausrichtet, sicher viel leichter umzusetzen. Dies würde aber letztendlich höchstens dazu führen, dass das System für den Großteil der Studenten nicht den geringsten Nutzen hat.

Bei einigen Fällen, wie beispielsweise dem Eiligen, und vor allem dem hoffnungslosen Fall kann man natürlich einwenden, dass sie bei einem System, wie dem in dieser Arbeit behandelten, schlicht an der falschen Adresse sind. Aber in der Praxis ist es auch für "normale" Studenten nicht ungewöhnlich, dass sie bei bestimmten Aufgaben in ein vergleichbares Muster verfallen.

Die Studentenprofile sind außerdem nicht ausschließlich Hilfsmittel beim Entwurf des Systems, sondern können gleichzeitig als Richtlinien für eine mögliche Konfigurierbarkeit dienen. Für viele Aspekte, wie das Verhalten bei erkannten Fehlern oder die Auswahl von Aufgaben, bieten sich verschiedene Einstellungsmöglichkeiten an. Dabei können Studentenprofile als Grundlage für bestimmte Standardkonfigurationen des Systems dienen.

4.4. Anforderungen des Systems an den Studenten

In diesem Abschnitt soll nun das Problem des Systems, herauszufinden *was* der Student *wozu* und *wie* macht, betrachtet werden. Dabei sollen diese Fragen idealerweise auf mehreren Ebenen beantwortet werden können. Zu Anfang des Kapitels wurde bereits festgestellt, dass es für das System am leichtesten ist herauszufinden, *was* der Student gerade macht, während es für das *Wie* und *Wozu* in der Regel auf Hinweise des Studenten angewiesen ist. Aber zunächst soll sich mit der ersten und einfacheren Frage befasst werden.

Dazu betrachten wir zunächst die erste Frage nach dem *Was* und die Ebenen auf denen diese gestellt und insbesondere beantwortet werden kann. Während der Verarbeitung einer Aufgabe existieren im wesentlichen folgende, für das System sichtbare Ebenen:

- *Sitzung*: Beschreibt die gesamte Zeit, in der der Student am Stück mit dem System beschäftigt ist
- *Aufgabe*: Beschreibt die derzeitige Aufgabe, die er bearbeitet
- *Teilaufgabe*: Beschreibt die bearbeitete Teilaufgabe, falls anwendbar (diese Ebene kann mehrfach vorkommen)
- *Lösungsschritt*: Beschreibt den aktuellen Schritt einer bestimmten Problemstellung auf dem Lösungsweg
- *Teilschritt*: Beschreibt einen eventuellen Teilschritt der Lösung (auch diese Ebene kann mehrfach vorkommen)
- *Eingabe*: Beschreibt die derzeitige für den Kern des Systems erkennbare Eingabe

Die vermutlich uninteressantesten Ebenen stellen hierbei die *Sitzungs-* und die *Eingabeebene* dar. Während die Antwort nach dem *Was* auf der *Sitzungsebene* in der Regel immer die gleiche sein wird, ist die Antwort auf der *Eingabeebene* vermutlich zu spezifisch. Letzteres hängt aber auch stark damit zusammen, welche Formen von Eingaben für den Kern des Systems überhaupt sichtbar ist.

Wesentlich interessanter für eine erfolgreiche Analyse von Lösungen und die Fehlererkennung sind die inneren Ebenen *Aufgabe* bis *Teilschritt.* Sind die Antworten für diese bekannt, ist das System sofort in einer viel besseren Ausgangslage, die aktuellen und bisherigen Eingaben des Studenten zu bewerten. Betrachte man beispielsweise folgendes Beispiel:

- *Aufgabe*: Umwandeln einer regulären Grammatik in einen minimalen DFA
- *Teilaufgabe*: Umwandeln eines NFA ein einen DFA
- *Lösungsschritt*: Erzeugen der Transitionen für alle erreichbaren Zustandsmengen
- *Teilschritt*: Erzeugen der Transition für eine bestimmte Zustandsmenge
- *Unterteilschritt*: Ermitteln der Folgezustände für ein bestimmtes Element der Zustandsmenge

Ist das System in der Lage all diese Informationen zu ermitteln, hat es sofort ein sehr gutes Bild davon, was der Student gerade macht. Die Frage ist nun, auf welche Weise man das System in diese Lage versetzen kann.

Dazu muss zunächst einmal eine gemeinsame Sprache existieren, in der das System die Lösung des Studenten bekommt. Diese muss nicht zwangsläufig identisch sein mit der genauen Eingabe des Studenten oder diesem überhaupt bekannt sein. Es ist ausreichend, wenn das Frontend in der Lage ist, die Eingaben des Studenten in die gewünschte Sprache für das System zu übersetzen und an das Backend zu übermitteln. In nächsten Kapitel soll ein Ansatz für eine solche Sprache vorgestellt werden. Vorher sollten aber noch die restlichen Aspekte der Interaktion betrachtet werden.

Während das *Was* die *Aktionen* des Studenten beschreibt. Gibt die Antwort auf das *Wie* und das *Wozu* eine Antwort einen Hinweis auf die *Absichten* des Studenten, sowie eine *Erklärung* für sein Verhalten. Dies ist, wie bereits mehrfach erwähnt, sehr schwer ohne

die aktive Unterstützung des Studenten. Zu diesem Zweck muss man dem Studenten im Rahmen der Kommunikation mit dem System Möglichkeiten geben, diese *Absichten* und *Erklärungen* auf geeignete Weise, also auf eine Weise die das System verstehen kann, auszudrücken. Dabei wird die genaue Form der Eingabe dieser Dinge wieder in das Frontend verlagert, aber die im vorigen Absatz angesprochene Sprache, in der der Kern des Systems die Eingaben des Studenten erhält, sollte die Möglichkeiten bieten, bestimmte *Absichten* und *Erklärungen* des Studenten zu formulieren.

4.5. Anforderungen des Studenten an das System

Neben den obligatorischen Eingaben des Studenten, die die Formulierung seiner Lösung betreffen, besteht in bestimmten Situationen der Wunsch, dem Studenten zu ermöglichen, von sich aus bestimmte Anfragen an das System zu stellen.

Ein Beispiel hierfür wäre zum Beispiel die Anforderung eines Hinweises zur Lösung der gestellten Aufgabe oder der Wunsch die bisherige Lösung auf Fehler zu überprüfen. Dies sind alles Anforderungen, die zum normalen Betrieb in dem der Student nur seine Lösung eingibt und am Schluss eine Rückmeldung zum Ergebnis erhält, zwar nicht nötig sind, aber je nach Notwendigkeit für den Lernenden sehr hilfreich sein können.

Schließlich besteht immer die Gefahr, dass ein Student bei einer bestimmten Aufgabe nicht weiter weiß oder besorgt ist, seine Zeit mit einem falschen Lösungsansatz zu verschwenden. In diesem Fall kann ein kleiner Tipp des Systems einen großen Unterschied machen. Es bestünde zwar auch die Möglichkeit, dass das System von sich aus eingreift, wenn der Student beim Lösen der Aufgabe ins Stocken gerät, aber letztendlich ist der Student selbst meist besser in der Lage zu beurteilen, ob und welche Hilfe er benötigt oder nicht.

Nur weil ein Student an einer falschen Lösung arbeitet, heißt das nicht, dass er dies nicht noch rechtzeitig bemerkt und nur weil er seine Eingabe längere Zeit unterbrochen hat, heißt das nicht, dass er nicht weiter weiß. Wenn das System in solchen Fällen von sich aus aktiv wird, würde das von den meisten Studenten wohl eher als störend empfunden.

Die Frage ist nun, welche Anforderungen den meisten Sinn machen. Zunächst lassen sich im wesentlichen zwei grundsätzliche Typen von Anfragen unterscheiden. Im einem Fall möchte der Student wissen, ob er etwas richtig macht, im anderen Fall möchte er wissen, was er tun muss. Auch hier lassen sich wieder verschiedene Ebenen betrachten.

Genau genommen kann man sogar die selben Ebenen wie im vorhergehenden Abschnitt verwenden: Aufgabe, Teilaufgabe, Lösungsschritt und Teilschritt. Die beiden Fragen können sich jeweils auf eine dieser Ebenen beziehen. Aus Sicht des Systems hat die Frage nach der Richtigkeit eine Rückschau zur Folge und die Frage danach, was zu tun ist, eine Vorschau.

Wenn man sich also auf die beiden im letzten Absatz beschriebenen Formen von Anfragen beschränkt, kann man mit ihnen trotzdem alle Anfragen, die in den einleitenden Abschnitten dieses Kapitels verwendet wurden, beschreiben. So lässt sich die Frage "Habe ich den letzten Schritt richtig gemacht?" im wesentlichen als eine

Rückschau auf Ebene der Lösungsschritte darstellen. Die Frage "Wie kann ich die Aufgabe lösen?" entspräche einer Vorschau auf der Ebene der Aufgabe.

Damit wurde eine Methode gefunden, wie man eine Vielzahl von Anliegen des Studenten auf einfache Weise darstellen und bestimmten Handlungsweisen des Systems zuordnen kann. Ob das System anschließend in der Lage ist, die Frage zu beantworten, hängt dann natürlich immer noch davon ab, ob das System ein klares Bild vom Lösungsweg des Studenten und dessen Position auf diesem hat.

4.6. Ergebnisse dieses Kapitels

In diesem Kapitel wurde versucht, die Ziele, welche bei der Interaktion und Kommunikation des Systems mit dem Studenten verfolgt werden sollen, zu ermitteln. Die wichtigsten Ziele waren dabei, dass das System möglichst zu jedem Zeitpunkt wissen sollte *was* der Student *wie* und *wozu* gerade macht.

Dabei wurde eine Betrachtungsweise, die auf mehren Ebenen beruht, zugrunde gelegt. Die für das System interessanten Ebenen sind die Ebene der Aufgabe, Teilaufgaben, Lösungsschritte und Teilschritte. Wobei die Ebenen der Teilaufgaben und Teilschritte auch mehrfach oder gar nicht auftreten können. Ein wichtiger Aspekt für das System ist die Frage nach dem *Was*. Dies ist zwar in vielen Fällen selbstständig ableitbar, für die Fragen nach dem *Wozu* und dem *Wie* ist das System jedoch auf die Mitarbeit und Hinweise durch den Studenten angewiesen ist.

Als weiteres Ergebnis des Kapitels wurden verschiedene Typen von Studenten, die sich durch ihr Fachwissen, ihre Selbstsicherheit und Selbstständigkeit sowie ihre Gründlichkeit unterscheiden, vorgestellt. Diese Typen können ein wichtiges Hilfsmittel sein, wenn man die Interaktion zwischen beliebigen Studenten und dem System analysieren möchte. Gleichzeitig können sie als Profile für verschiedenen Konfigurationen des Systems sein, die dessen Verhalten beeinflussen.

Außerdem wurde der Versuch gemacht, Fragen, die ein Student während der Verarbeitung einer Aufgabe an das System stellen könnte, in bestimmte Klassen zu unterteilen. Dabei wurde zwischen der Anforderung nach einer Rückschau zur Fehlerüberprüfung sowie einer Vorschau für Hinweise unterschieden. Beide Varianten konnten sich dabei auf jede der beschriebenen Betrachtungsebenen beziehen.

Als wichtigstes Ergebnis dieses Kapitels kann der Schluss sein, dass es sehr sinnvoll ist, eine formale Sprache zu entwickeln, welche die Eingaben des Studenten in das System auf leicht zu verarbeitende Weise beschreibt. Diese Sprache soll aber nicht als Eingabesprache für den Studenten dienen, sondern sie soll das Ergebnis eines Übersetzungsprozesses sein, den das Frontend durchführt. Sie stellt somit den Kern der Kommunikation zwischen Frontend und Kern des Systems dar und soll im folgenden Kapitel vorgestellt werden.

5. Eine Lösungsbeschreibungssprache

5.1. Zielsetzung

Eine Schlussfolgerung aus dem vorherigen Kapitel zum Thema der Interaktion mit dem Studenten war die Tatsache, dass es sich anbietet, eine geeignete Repräsentation für die Eingaben eines Studenten zu finden. Zu diesem Zweck soll in diesem Kapitel eine spezielle Sprache entwickelt werden. Da im Rahmen dieses Systems ein Großteil der Eingabe des Studenten die von ihm entwickelte Aufgabenlösung besteht, soll diese Sprache als *Lösungsbeschreibungssprache* kurz *LBS* bezeichnet werden. Daneben soll die Sprache auch Elemente beinhalten, die besondere Anfragen des Studenten, wie im vorherigen Kapitel vorgestellt, beschreiben können.

Zusammengefasst soll die LBS folgende Elemente, die ein Student im Rahmen eines Lösungsansatzes beschreibt, bereitstellen:

- *Ziele:* Ich möchte die Grammatik in einen regulären Ausdruck überführen.
- *Ideen*: Man kann diese Eigenschaft mit Vollständiger Induktion beweisen.
- *Aktionen*: Ich minimiere den DFA.
- *Definitionen*: Sei M=(Z,Σ,δ,S,E).
- *Fragen*: Wie kann ich diese Teilaufgabe lösen?

Anhand der oben aufgeführten Beispiele könnten die entsprechenden Ausdrücke zum Beispiel folgendermaßen aussehen:

- *Ziel*: `goal is transform object(grammer,G) to object(expression,ALP)`
- *Idee*: `idea is proof of property(p,M) using method(induction)`
- *Aktion*: `modify object(dfa,M0) to object(dfa,M) with property(equiv,(M,M0)) and property(minimal,M0) using method (minimize)`
- *Definition*: `define object(nfa,M) as tuple(Z,SIG,DEL,S,E)`
- *Frage*: `preview on subexercise`

5.2. Allgemeine Eigenschaften der LBS

Die Syntax der Sprache soll relativ einfach gehalten werden. Sie soll die Form einer beliebig langen Folge von klar definierten Ausdrücken haben. Semantisch beschreibt jeder dieser Ausdrücke eines der oben aufgeführten Elemente. Es gibt also Ausdrücke für Ziele, Ideen, Aktionen, Definitionen und Fragen des Studenten.

Semantisch hat jeder einzelne Ausdruck einen bestimmten Effekt auf das System. Sie dienen insbesondere dazu, die Frage *was* der Student gerade genau macht, zu beantworten beziehungsweise zu aktualisieren. Bei Definitionen kann das System intern

entsprechende Objekte erzeugen und diese zum Beispiel mit den in der eigenen Lösung entstandenen abgleichen oder sie einfach auf Konsistenz prüfen, um auf diese Weise Fehler zu erkennen.

Beispiele, wie das für die Aktion, zeigen aber, dass einzelne Ausdrücke bereits recht komplex sein können. Aber um eine Aktion wie die Minimierung eines DFA präzise zu beschreiben sind eben einige Angaben notwendig.

Im folgenden soll die Syntax der einzelnen Arten von Ausdrücken in EBNF angegeben werden:

5.2.1. Formulieren von Zielen

Im Allgemeinen hat ein Ausdruck zur Formulierung eines Ziels die folgende Form:

```
GoalExp ::= goal is Action [with Prop] [using Method]
```

Semantisch beinhaltet die Formulierung eines Zieles im Rahmen der LBS immer die Beschreibung einer Aktion. Zusätzlich lassen sich noch bestimmte Bedingungen oder eine bestimmte Lösungsmethode angeben. Die Syntax der entsprechenden Ausdrücke wird weiter unten definiert.

Zweck der Formulierung eines Ziels soll es sein, dem System leichter nachvollziehbar zu machen, was der Student beabsichtigt. Gleichzeitig dient dieser Ausdruck auch dazu eine Aufgabe selbst zu beschreiben. Dabei besteht eine Aufgabe aus der Formulierung einer oder, im Falle von Unteraufgaben, mehreren *Ziel*-Ausdrücken und der Definition der gegebenen Objekte.

5.2.2. Formulieren von Ideen

Ein Ausdruck zur Formulierung einer Idee ähnelt stark dem für Ziele und hat die Form:

```
IdeaExp ::= idea is Action [using Method]
```

Eine Idee beinhaltet also die Beschreibung einer Aktion zusammen mit der optionalen Angabe einer Methode. Wie die Zielformulierung dient sie dazu, dem System ein besseres Verständnis der Lösung des Studenten zu erlauben. Umgekehrt kann sie dazu dienen, Hinweise des Systems an den Studenten zu beschreiben.

5.2.3. Formulieren von Aktionen

Aktionen spielten bereits bei den vorherigen Ausdrücken eine Rolle. Dort dienten sie aber nur zur Beschreibung von Ideen und Zielen. Dieser Abschnitt führt nun die Ausdrücke ein, die sich mit ganz konkreten Aktionen bei der Lösung einer Aufgabe beschäftigen:

```
ActionExp ::= Action [with Prop] [using Method]
```

Welche Aktionen genau möglich sind, soll in einem späteren Abschnitt beschrieben werden. Dort soll dann auch die Semantik vertieft werden.

5.2.4. Formulieren von Definitionen

Während mit Aktionsbeschreibungen nur allgemein beschrieben wird, welche Objekte unter welchen Bedingungen verarbeitet, sind es Definitionen, welche diese Objekte konkretisieren. Definitionen haben dabei die folgende allgemeine Form:

```
DefExp ::= define Object as Content
DefExp ::= add Object to Object | remove Object from Object
DefExp ::= assign Object to Object
```

Diese Definitionen beschreiben die gängigsten Manipulationen, die an Objekten ausgeführt werden können. Dies ist zum einen die Definition dessen, was ein Objekt beschreibt, wie zum Beispiel eine Menge, zum anderen gibt es Ausdrücke um bestimmte Objekte in andere einzufügen oder etwas aus ihnen zu entfernen. Am Beispiel der Menge würde es sich dabei um die in der Menge enthaltenen Elemente handeln. Zusätzlich existiert ein Ausdruck, mit dem man begonnene Definitionen abschließen kann. Im Falle einer Menge dient das zum Beispiel dazu, zu formulieren, dass sie keine weiteren als die bereits angegebenen Elemente enthält. Auch hier soll die genaue Semantik in einem späteren Abschnitt betrachtet werden.

5.2.5. Formulieren von Fragen

Da, wie im vorherigen Kapitel beschrieben, die häufigsten Fragen auf eine Rückschau oder Vorschau auf einer bestimmten Ebene reduziert werden können, ist deren Definition im Rahmen der LBS sehr einfach:

```
QueryExp ::= review on Layer | preview on Layer
Layer ::= exercise | subexercise | step | substep
```

5.3. Objekte, Eigenschaften und Methoden

Bei praktisch jeder vorgestellten Ausdrucksform tauchten Objekte, Eigenschaften und Methoden auf. Dazu kommen noch die grundlegenden Elemente der Variablen und Konstanten vor. Ihre genaue Syntax lautet:

```
Object ::= object '(' Const ',' Var ')'
Prop ::= property '(' Const ',' Var ')' [and Prop]
Method ::= method '(' Const ')'
```

Für die Elemente der Variablen und Konstanten soll auf die Angabe der EBNF verzichtet werden. Dabei handelt es sich immer um alphanumerische Zeichenketten, die, ähnlich wie in der Sprache PROLOG, im Falle der Konstanten mit einem kleinen Buchstaben beginnen und im Falle der Variablen mit einem Großbuchstaben.

Bei Objekten handelt es sich um die theoretischen Gegenstände, die bei der Bearbeitung einer Aufgabe betrachtet werden. Dabei kann es sich um einen Automaten oder einen regulären Ausdruck oder auch einfach eine Menge und vieles mehr handeln. Syntaktisch

betrachtet sind es Paare aus einer Konstante und einer Variable. Dabei beschreibt die Konstante den Typen des Objekts und die Variable dient als Platzhalter für das Objekt selbst.

Die direkte Manipulation von Objekten geschieht mit den in Abschnitt 5.2.4. vorgestellten *Definitions*-Ausdrücken. Sie erlauben es, Objekte genau zu deklarieren, andere Objekte in sie einzufügen oder wieder zu entfernen beziehungsweise Objekte abzuschließen, was weitere Manipulationen unmöglich macht und die Objekte und ihren Inhalt endgültig definiert.

Eigenschaften sind syntaktisch betrachtet eine Folge von Paaren von Konstanten und Variablen. Für jedes Paar beschreibt die Konstante eine bestimmte Eigenschaft, während die Variable für ein bestimmtes an andere Stelle definiertes Objekt steht. Besteht die Folge aus mehr als einem Paar, beschreibt sie eine Art der zusammengesetzten Eigenschaft, die genau dann erfüllt ist, wenn jede einzelne Eigenschaft erfüllt. Im logischen Sinne sind solche Folgen also Konjunktionen.

Bei Methoden handelt es sich schließlich um durch eine Konstante eindeutig beschriebene Lösungsmethoden für bestimmte Aktionen, wie Transformationen oder Beweisformen. Sie dienen vor allen Dingen als Hinweise, die dem System die Analyse erleichtern sollen.

5.4. Arten von Aktionen

Nicht nur weil die Beschreibung einer Aktion in immerhin drei von fünf Ausdrucksformen eine Rolle spielt, nehmen diese offensichtlich eine wichtige Position ein. Im Prinzip ist jeder Lösungsschritt eine Aktion. Es soll nun geklärt werden, welche Formen von Aktionen benötigt werden.

Eine sehr allgemeine Aktion stellt die Umwandlung eines Objekt in ein anderes dar, dies kann auf sehr viele Weisen geschehen. Eine konkretere Beschreibung einer solchen Aktion könnte zum Beispiel das Verändern eines gegebenen Objekts sein. Eine andere Variante lässt sich durch das Erzeugen eines neuen Objekt auf Basis eines alten beschreiben. Damit haben wir schon drei verschiedene Arten, die wie folgt definiert werden:

```
Action ::= transform Object to Object
Action ::= modify Object to Object
Action ::= create Object from Object
```

Dabei beschreibt die *Transform*-Aktion im wesentlichen eine Verallgemeinerung der beiden spezifischen Aktionen *Modify* und *Create*. Sie beschreibt einen, Vorgang an dessen Beginn ein Objekt und an dessen Ende ein anderes steht. Dabei wird keine Festlegung getroffen, auf welche Weise der Übergang stattfindet. Sie beschreibt immer eine Aktion auf der Ebene von Aufgaben oder Teilaufgaben.

Die *Modify*-Aktion beschreibt einen Vorgang bei dem ein Objekt durch bestimmte Veränderungen in ein anderes überführt wird. Sie hat die besondere Einschränkung, dass sich der Typ des Objektes dabei nicht verändert, nur dessen Inhalt. Diese Aktion

kann sich wie auch die folgende *Create*-Aktion sowohl auf die Arbeitsebene der Aufgabe und Teilaufgabe als auch auf die der Lösungsschritte und Teilschritte beziehen.

Eine *Create*-Aktion schließlich dient zur Beschreibung des Vorgangs, dass ein neues Objekt erzeugt wird, wobei Eigenschaften eines Ursprungsobjektes, das nicht vom selben Typ sein muss, als Grundlage dienen können. Ein Beispiel hierfür ist die in Abschnitt 3.3 vorgestellte Transformation einer regulären Grammatik in einen NFA.

Natürlich ist die Überführung einzelner Objekte ineinander nicht die einzige Form von Aktion, die in einer Lösung eine Rolle spielen kann. Eine andere wichtige Aktion ist die des Beweises einer Eigenschaft eines Objektes. Sie ist wie folgt definiert:

```
Action ::- proof of Prop [using method]
```

Auch die *Proof*-Aktion kann sich wieder auf jede Arbeitsebene beziehen.

In der Praxis sind sicher auch noch viele andere Arten von Aktionen denkbar, aber wie vieles in dieser Arbeit soll hier nur ein Ansatz vorgestellt. Die vorgestellte erhebt nicht nicht den Anspruch allumfassend und vollständig definiert zu sein, sondern soll vor allem als Basis zu einer konkreten Implementierung dienen und ist somit leicht erweiterbar gestaltet.

5.5. Arten von Objekten

Neben den Aktionen, die in der LBS beschrieben werden sollen, spielt natürlich auch die genaue Form der Objekte eine Rolle. In diesem Abschnitt sollen nun die Syntax und Semantik einiger dieser Objekte vorgestellt werden.

Das wichtigste und häufigste Objekt im Rahmen der in diesem System bearbeiteten Aufgaben und Lösungen ist natürlich die Menge. In praktisch jeder Aufgabe spielt irgendeine Menge eine Rolle. Die anderen grundlegenden Objekte sind Atome. Bei ihnen handelt es sich um die einzigen Objekte, die keine anderen Objekte beinhalten. Sie werden in der Sprache, im Rahmen eines Definitions-Ausdrucks auf folgende denkbar einfache Weise definiert.

```
Content ::= set | atom '(' Const ')'
```

Mittels der restlichen Arten von Definitions-Ausdrücken können andere Objekte hinzugefügt und entfernt werden, sowie die Menge als vollständig festgelegt werden. Im Zusammenhang mit Atomen sind die selben Ausdrücke jedoch wirkungslos. Als Beispiel der Definition einer Menge $\Sigma = \{a, b\}$ dient diese Folge von Ausdrücken:

```
define object(alphabet,SIGMA) as set
define object(terminal,X) as atom(a)
add object(terminal,X) to object(alphabet,SIGMA)
define object(terminal,X) as atom(b)
add object(terminal,X) to object(alphabet,SIGMA)
finish object(alphabet,SIGMA)
```

Dieses Beispiel illustriert zwar zum einen, dass die Beschreibung eines einfachen Sachverhalts durch sie, schon verhältnismäßig länglich sein kann. Zum anderen erlaubt das aber auch eine relativ genaue Analyse. Da es sich in der Praxis bei der LBS nur um eine interne Sprache des Systems handeln soll, spielt die Länge der Darstellung auch keine zu große Rolle.

Eine Fragestellung, die das Beispiel vielleicht aufwirft, ist die Bedeutung des letzten Ausdrucks und ob es einen Unterschied machen würde, wenn sie fehlt. Die Antwort auf diese Frage muss Ja lauten. So lange kein *Abschluss*-Ausdruck zu einem *Mengen*-Objekt aufgetreten ist, gilt die Menge als offen. Das heißt, sie enthält zwar alle Objekte, die ausdrücklich in sie eingefügt wurden, schließt aber nicht aus, dass sich noch beliebige andere Objekte in ihr befinden. Dazu betrachte man auch folgenden alleinstehenden Ausdruck:

```
define object(set, M) as set
```

Dieser definiert keineswegs die leere Menge $M = \emptyset$, sondern eine Menge, über deren Inhalt nichts bekannt ist. Erst durch hinzufügen eines *Abschluss*-Ausdrucks ohne vorheriges Hinzufügen irgendwelcher Objekte wäre die Menge vollständig beschrieben und somit die leere Menge.

Neben den Mengen und Atomen sind Funktionen und und Tupel ein häufiges Element jeder Lösung. Auf eine getrennte Definition dieser Objektarten soll hier verzichtet werden, da Funktionen eine bestimmte Teilmenge von Mengen von Tupeln darstellen. Ein Tupel wird, wieder im Rahmen eine *Definitions*-Ausdrucks, mit folgender Syntax definiert:

```
Content ::= tuple '(' Var ',' Tuple ')'
Tuple ::= Var ',' Tuple
```

Dabei kann man die Variablennamen als Referenzen betrachten. Verändert man Objekte mit diesen Variablennamen, verändert man auch den Inhalt des Tupels. Dies ist ein Unterschied zu dem Verhalten bei Mengen. Wenn Objekte einer Menge hinzugefügt werden, dann nur deren Inhalt. Die Referenz geht dabei verloren. Der Inhalt einer Menge ändert sich nur durch Hinzufügen oder Entfernen. Nachträgliches Verändern der dabei verwendeten Objekte, beeinflusst die Menge nicht.

Als Beispiel der Verwendung eines Tupel-Objekts soll folgende Definition einer einfachen Übergangsfunktion dienen:

```
define object(function, DELTA) as set
define object(transition, T) as tuple(TX, TY, TZ)
define object(state, TX) as atom(z0)
define object(terminal, TY) as atom(a)
define object(state, TZ) as atom(z1)
add object(transition, T) to object(function, DELTA)
define object(state, TX) as atom(z0)
```

```
define object(terminal, TY) as atom(b)
define object(state, TZ) as atom(z2)
add object(transition, T) to object(function, DELTA)
define object(state, TX) as atom(z1)
define object(terminal, TY) as atom(a)
define object(state, TZ) as atom(z2)
add object(transition, T) to object(function, DELTA)
finish object(function,DELTA)
```

Neben Mengen, Atomen und Tupeln sind sicher noch weitere grundlegende Objektformen denkbar. Aber für den Moment soll sich auf diese drei wichtigen beschränkt werden, da es sich bei der LBS nur um einen Lösungsansatz handelt. Ihr Hauptzweck liegt in der einfacheren Analyse von Lösungswegen und nicht in der möglichst eleganten Formulierung mathematischer und anderer Objekte.

5.6. Besonderes Konstanten

Sowohl Eigenschaften als auch Methodenangaben beinhalten immer eine Konstante, die sie beschreibt. Beispiel für solche Konstanten und ihre Bedeutung sind die folgenden:

- *equiv*: Beschreibt die Eigenschaften, dass zwei Repräsentationen äquivalent bezüglich der erkannten/erzeugten Sprache sind
- *minimal*: Beschreibt die Eigenschaft eines DFA, dass die Anzahl der Zustände minimal ist
- *dfa2grammer*: Beschreibt das Verfahren zum Umwandeln eines DFA in eine reguläre Grammatik
- *nfa2dfa*: Umwandlung eines NFA in einen DFA
- *minimize*: Minimierung eines DFA

Diese Konstanten sind nicht Teil der Sprache selbst, aber sie haben für das System eine Bedeutung. Mit ihrer Hilfe lässt sich das System auch sehr leicht erweitern ohne die LBS selbst zu verändern.

Auch für gängige Objekte sind und können solche Konstanten definiert werden:

- *dfa*: Das Objekt beschreibt einen DFA
- *transition*: Das Objekt beschreibt eine Transition

Weitere Beispiel lassen sich den Code-Beispielen entnehmen. Dabei muss aber nicht jede vorkommende Konstante für das System eine Bedeutung haben. Dies sollte nur für Objekte, Eigenschaften und Verfahren der Fall sein, mit denen das System auch auf andere Weise umgehen kann.

5.7. Ein größeres Beispiel

Es soll nun am Beispiel der Umwandlung eines einfachen DFA in eine Reguläre Grammatik. Der Automat ist wie folgendermaßen definiert und akzeptiert die Sprache aller Worte über {a,b} die mit a enden:

$$M=(Z,\Sigma,\delta,S,E)\ mit\ Z=\{z_0,z_1\},\Sigma=\{a,b\},S=z_0,E=\{z_1\}$$
$$\delta(z_0,a)=z_0,\delta(z_0,b)=z_1,\delta(z_1,a)=z_0,\delta(z_1,a)=z_1$$

Zunächst soll diese Aufgabe und der Automat in LBS beschrieben werden:

```
goal is transform object(dfa, M) to object(grammer, G)
     with property(equiv,(M,G))
define object(dfa, M) as tuple(ZM, SIGM, DELM, SM, EM)
define object(states, ZM) as set
define object(alphabet, SIGM) as set
define object(function, DELM) as set
define object(start, SM) as atom(z0)
define object(end, EM) as set
define object(state, Z0) as atom(z0)
define object(state, Z1) as atom(z1)
define object(terminal, A) as atom(a)
define object(terminal, B) as atom(b)
add object(state, Z0) to object(states, ZM)
add object(state, Z1) to object(states, ZM)
finish object(states, ZM)
add object(terminal, A) to object(alphabet, SIGM)
add object(terminal, B) to object(alphabet, SIGM)
finish object(alphabet, SIGM)
define object(transition, T) as tuple(TX,TY,TZ)
define object(state, TX) as atom(z0)
define object(terminal, TY) as atom(a)
define object(state, TZ) as atom(z0)
add object(transition, T) to object(function, DELM)
define object(state, TX) as atom(z0)
define object(terminal, TY) as atom(b)
define object(state, TZ) as atom(z1)
```

```
add object(transition, T) to object(function, DELM)
define object(state, TX) as atom(z1)
define object(terminal, TY) as atom(a)
define object(state, TZ) as atom(z0)
add object(transition, T) to object(function, DELM)
define object(state, TX) as atom(z1)
define object(terminal, TY) as atom(b)
define object(state, TZ) as atom(z1)
add object(transition, T) to object(function, DELM)
finish object(function, DELM)
add object(state, Z1) to object(end, EM)
finish object(end, EM)
```

Damit ist der DFA M und die Aufgabe vollständig beschrieben. Ein korrektes Lösungsprotokoll könnte darauf dann so aussehen:

```
goal is transform object(dfa,M) to object(grammer,G)
     with property(equiv,(M,G))
create object(grammer,G) from object(dfa,M)
     using method(dfa2grammer)
define object(grammer, G) as tuple(VG, SIGD, P1D, P0D
SD)
assign object(states,ZM) to object(variables,VG)
assign object(alphabet, SIGM) to object(alphabet, SIGD)
define object(productions, P1D) as set
define object(productions, P0D) as set
define object(production, P1) as tuple(PX,PY,PZ)
define object(production, P0) as tuple(PX,PY)
define object(variable, PX) as atom(z0)
define object(terminal, PY) as atom(a)
define object(variable, PY) as atom(z1)
add object(production,P1) to object(productions,P1D)
add object(production,P0) to object(productions,P0D)
define object(terminal, PY) as atom(b)
define object(variable, PY) as atom(z0)
```

```
add object(production,P1) to object(productions,P1D)
define object(variable, PX) as atom(z1)
define object(terminal, PY) as atom(a)
define object(variable, PY) as atom(z1)
add object(production,P1) to object(productions,P1D)
add object(production,P0) to object(productions,P0D)
define object(terminal, PY) as atom(b)
define object(variable, PY) as atom(z0)
add object(production,P1) to object(productions,P1D)
assign object(start,SM) to object(start,SD)
review on exercise
```

Damit ist die gewünschte Grammatik vollständig beschrieben. Das entscheidende sind jedoch die ersten beiden Zeilen. Durch den *Ziel*-Ausdruck ist für das System erkennbar, welche Eigenschaft bei der Umwandlung erhalten bleiben soll, nämlich die Äquivalenz beider Repräsentationen. Durch den folgenden *Aktions*-Ausdruck ist erkennbar, was konkret im folgenden geschieht, da die verwendete Methode der Umwandlung eines DFA in eine Grammatik dort beschrieben ist.

Wenn man das Beispiel für die Aufgabenstellung mit dem für die Lösung vergleicht, erkennt man, dass beide mit einem identischen *Ziel*-Ausdruck beginnen. Dies ist keineswegs redundant, sondern ein gewünschtes Verhalten. Man muss beachten, dass sich die Aufgabe an den Studenten richtet und die Lösung an das System.

Der Rest des Beispiels besteht nur aus den eigentlichen Eingaben der Lösung, wobei dies teilweise schon in komprimierter Form geschieht. Mit dem letzten Ausdruck wird dem System mitgeteilt, dass der Student die Aufgabe beendet hat und wissen möchte ob er sie richtig gelöst hat.

5.8. Ergebnisse des Kapitels

In diesem Kapitel wurde die LBS, eine Sprache zur Beschreibung von Lösungen und Anfragen des Benutzers vorgestellt. Es hat sich gezeigt, dass die gewählte Repräsentation zwar einerseits zu recht langen Beschreibungen für einfache Lösungen führt, aber andererseits besteht dadurch eine feinere Auflösung auf die einzelnen Schritte einer Lösung.

Da die Sprache sich nicht an den Studenten als Endanwender, richtet sondern, abhängig von dessen Eingaben, vom Frontend erzeugt wird, um dann vom Kern des Systems verarbeitet zu werden. Ein wichtiger Aspekt ist auch, dass die Sprache direkt Ausdruck für Ausdruck verarbeitet wird. Es handelt sich dabei nur um eine Beschreibungssprache, und es spielt keine Rolle ob jeder einzelne Ausdruck auch tatsächlich vom System berücksichtigt wird.

Beim Blick auf die Definitionen und Beispiele könnte man sich die Frage stellen,

weshalb die Sprache so "blumig" ist und nicht deutlich knapper formuliert ist, da es sich ja nur um eine programminterne Sprache handelt. Ein Grund dafür ist die Tatsache, dass sie nebenbei auch als Mittel zur Analyse und Fehlerkorrektur beim Entwurf von Frontend und Backend dienen soll. Je leichter verständlich sie ist, desto schneller erkennt man eine fehlerhafte Kommunikation zwischen beiden Komponenten. Sie erlaubt dadurch auch eine unabhängige Entwicklung von Frontend und Backend.

Sollte die LBS, wie sie hier vorgestellt wurde, in der Praxis nicht ausreichend sein, so bietet sie beispielsweise durch das Hinzufügen neuer Arten von Aktionen und Objekten eine leichte Möglichkeit, ihren Sprachumfang zu erweitern. Neben einer Erweiterung der Sprache selbst bietet auch die Festlegung neuer besonderer Konstanten, wie in Abschnitt 5.6 beschrieben eine Möglichkeit, die die Syntax der Sprache nicht verändert.

6. Zusammenführung

6.1. Rückblick auf die vorherigen Kapitel

In den Kapiteln 2 bis 5 wurden sehr unterschiedliche Ansätze zur Lösung bestimmter Teilprobleme des Coaching-Systems eingeführt. Diese sollen an dieser Stelle noch einmal zusammengefasst werden.

Das Kapitel 2 beschäftigte sich mit einer genaueren Betrachtung möglicher Übungsaufgaben, die das System verarbeiten und dem Studenten anbieten könnte. Dabei wurde zunächst eine Teilmenge aller möglichen Arten von Übungsaufgaben, wie sie zum Beispiel in [HMU01] zu finden sind, in bestimmte Gruppen eingeteilt. Darüber hinaus wurde das Thema der Fehlerquellen in Aufgaben angesprochen und Möglichkeiten der Verarbeitung erkannter Fehler diskutiert.

Im Kapitel 3 wurden nun exemplarische einige Standardaufgaben vorgestellt und von zwei Seiten betrachtet. Zum einen wurden Ansätze einer Implementierung eines Lösungsverfahrens für jede Aufgabe unter Verwendung eines automatischen Beweissystems vorgestellt. Zum anderen wurden die Lösungsverfahren dahingehend untersucht, welche Fehler ein Student bei ihrer Umsetzung machen könnte.

Der Inhalt des Kapitels 4 bestand aus einer genaueren Betrachtung der Kommunikation und Interaktion zwischen Student und Coaching-System. Dabei wurde zum einen herausgearbeitet, welche Informationen das System vom Studenten benötigt und auf welche Weise das System diese während der Eingaben des Studenten ermitteln kann. Umgekehrt wurde analysiert, welche Arten von Anfragen ein Student während des Lösens einer Aufgabe und der Arbeit mit dem System im Allgemeinen haben könnte. Außerdem wurden in dem Kapitel verschiedene Studentenprofile vorgestellt, die als Grundlage weiterer Betrachtungen dienen können.

Im Kapitel 5 wurde schließlich, auch als Konsequenz der Ergebnisse des vorhergehenden Kapitels, eine eigene Sprache zur Beschreibung von Lösungen vorgestellt, die Lösungsbeschreibungssprache LBS. Eine solche Sprache kann als Werkzeug für eine effektive Kommunikation zwischen dem Kern des Systems und dem Frontend sowie dadurch auch mit dem Studenten dienen.

Zusammengefasst haben wir nun als Ergebnis der vorhergehenden Kapitel unter anderem folgende Ansätze:

- Ansätze zur Verarbeitung von Fehlern
- Logikbasierte Lösungsverfahren für bestimmte Aufgaben
- Die Struktur von Lösungsverfahren bestimmter Aufgaben
- Genaue Fehlerquellen und Fehlerkategorien für bestimmte Aufgaben
- Verschieden Studentenprofile
- Eine Lösungsbeschreibungssprache

Ziel dieses Kapitels solle es nun sein, die erzielten Ergebnisse sinnvoll zusammenzuführen.

6.2. Zusammensetzen der Teile zu einem Ganzen

Es stellt sich nun die Frage, wie man unter Verwendung der vorgestellten Ansätze ein funktionsfähiges System zusammensetzen kann. Dazu sollen nun die beiden Teilkomponenten des Front- und Backend betrachtet werden sowie einige weitere grundlegende Aspekte.

6.2.1. Aktive und passive Fehlerbehandlung

Während die Erkennung von Fehlern in jedem Fall Aufgabe des Kerns des Systems und somit des Backend sein soll, stellt sich die Frage nach der als Reaktion auf eine Erkennung erfolgenden Fehlerbehandlung. Dabei ergab sich in Kapitel 2 die Unterscheidung zwischen aktiver und passiver Behandlung.

Die Entscheidung, ob für einzelne Fehler eine aktive, also für den Studenten sichtbare Behandlung, stattfindet, kann man entweder dem Backend oder dem Frontend überlassen. Im letzten Fall würde das Frontend Informationen über jeden vom Kern erkannten Fehler bekommen und diese dann entweder ignorieren oder an den Studenten weitergeben. Dabei sollte, um das Frontend nicht unnötig kompliziert zu gestalten, das Verhalten relativ statisch sein.

Der andere Fall der Entscheidung über die Fehlerbehandlung im Backend selbst hätte den Vorteil, dass dieses im Idealfall bereits sämtliche Fehler protokolliert und so besser in der Lage ist, unter Berücksichtigung der gesamten Fehler-Historie zu entscheiden, wie jeder Fehler im einzelnen zu behandeln ist.

Beide Ansätze haben ihre Vor- und Nachteile. In beiden Fällen können jedoch auch spezielle Konfigurationen das Verhalten steuern. Für ein System, dass den Anspruch erhebt, diesen Bereich auf "intelligente" Weise zu behandeln, sollte man sich wohl für die zweite Variante entscheiden. Das Frontend hätte dann nur noch die Aufgabe, alle vom Kern gemeldeten Fehler an den Studenten in geeigneter Weise weiterzugeben und dessen Reaktion dem Kern zu melden.

6.2.2. Konfiguration des Systems

Ein weiterer wichtiger Aspekt, den man, vor dem Angeben einer genaueren Implementierung des Gesamtsystems betrachten sollte, ist die Konfigurierbarkeit. Dabei soll entschieden werden, welche Teile des Systems auf welche Weise konfigurierbar sein sollen. Auch die Frage, ob das System als Ganzes oder die beiden Teilkomponenten Front- und Backend getrennt konfigurierbar gemacht werden sollen, spielt eine wichtige Rolle.

Zunächst stellt sich aber die Frage, bei welchen Eigenschaften und Verhalten des Systems es sich anbietet es mittels Konfigurationen variabel zu gestalten. Zunächst trifft dies natürlich, wie bereits mehrfach erwähnt, auf den Bereich der Fehlerbehandlung zu. Hierbei bieten sich Einstellungen für jede Fehlerkategorie an, die besagen, ob entsprechende Fehler aktiv oder passiv behandelt werden. Zusätzlich kann man sowohl

für einzelne Kategorien oder für die Menge aller erkannten Fehler einen Schwellenwert angeben, der steuert wann von einer passiven zu einer aktiven Behandlung übergegangen wird.

Ein weiterer Bereich, der je nach Studententyp und vielleicht auch Aufgabentyp unterschiedliches Verhalten aufweisen könnte, ist die Aufgabenabwicklung. Hierzu sind Einstellungen denkbar, welche die Ausführlichkeit in der Darstellung der vom Studenten erzielten "Ergebnisse" steuern.

Etwas, das bisher noch gar nicht thematisiert wurde, ist der Bereich der Aufgabenauswahl. Dabei handelt es sich um einen relativ wichtigen Aspekt. Denn eigentlich sollte das System mehr tun als einfach nur zufällig eine Aufgabe aus dem eigenen Repertoire auszuwählen und dem Studenten vorzusetzen. Idealerweise sollten dabei zum einen die Wünsche des Studenten, und zum anderen dessen bisherige Leistungen eine Rolle spielen.

Beide Dinge lassen sich auch durch Konfigurationen steuern. Zum einen sollte die Wünsche des Studenten konfigurierbar sein, also beispielsweise welche Aufgabengruppen und welchen Schwierigkeitsgrad der Student bevorzugt. Zum anderen soll auch einstellbar sein, in wie weit bisher bearbeitete Lösungen und die dabei erbrachten Leistungen den Schwierigkeitsgrad beeinflussen.

Wir haben nun also zumindest drei Bereiche, welche sinnvollerweise konfigurierbar sein sollten:

1. Fehlerverhalten
2. Aufgabenabwicklung
3. Aufgabenauswahl

Je nachdem wie genau man diese Dinge nun konfigurieren kann, stellt sich natürlich die Frage, ob sich ein Student überhaupt die Mühe machen soll, diese Einstellungen vorzunehmen. Eigentlich will der Student Aufgaben lösen und üben und nicht erst lange alles einstellen. Deswegen spielen hier, neben einer geeigneten Standardeinstellung, unterschiedliche Konfigurationsprofile eine Rolle.

Will ein Student sich gar nicht mit der Konfiguration beschäftigen, verwendet er die Standardeinstellungen. Möchte er nicht viel einstellen, kann er einfach ein für ihn geeignetes Profil auswählen. Und schließlich hat er, falls er es genau wissen will, die Möglichkeit, jeden konfigurierbaren Aspekt nach seinen Wünschen einzustellen.

Die Konfigurationsprofile stellen offenbar einen guten Mittelweg dar und als Grundlage können hierzu die in Kapitel 4 vorgestellten Studentenprofile dienen. Die bei den Studentenprofilen betrachteten Eigenschaften lassen sich relativ einfach in entsprechende Konfigurationen übersetzen.

Natürlich sind die erwähnten Einstellungen nicht die einzig sinnvollen oder auch einzig nötigen Parameter, die das Verhalten des Systems steuern. Neben den genannten, am ehesten als Benutzereinstellungen, zu verstehenden Aspekten, könnten auch noch einige interne Parameter des Systems konfigurierbar sein. Diese sollten aber nicht für den Studenten als Anwender des Systems sichtbar und manipulierbar sein.

6.2.3. Persistente Informationen über Studenten

Ein weiterer Aspekt, der bisher nicht thematisierte wurde, ist die Fragestellung was passiert, wenn ein Student seine Sitzung beendet und der selbe Student zu einem späteren Zeitpunkt erneut mit dem System arbeitet. Ein Student der beispielsweise das System ausführlich für seine Bedürfnisse konfiguriert hat, ist natürlich daran interessiert, dass er beim nächsten Zugriff nicht wieder alles neu einstellen muss. Zu diesem Zweck kann es sinnvoll sein, gewisse Informationen über den Studenten zu speichern.

Neben den Konfigurationseinstellung, könnte beispielsweise eine Historie der bisher bearbeiteten Aufgaben sowie den dabei erzielten Ergebnissen für einen Studenten gespeichert bleiben. Beim web-basierten Ansatz gibt es im Prinzip zwei Methoden, diese Informationen zu speichern. Eine besteht darin, diese Einstellungen auf dem vom Studenten verwendeten Rechner zu speichern, die andere darin, diese Information zentral für alle Studenten zu speichern.

Im letzteren Fall besteht natürlich die Notwendigkeit, dass der Student in irgendeiner Weise für das System identifizierbar ist, indem er sich beispielsweise bei jeder Sitzung mit einer eindeutigen Kennung anmeldet. Dies erhöht natürlich den Verwaltungsaufwand beträchtlich und kann für manchen Studenten störend wirken, da sie lieber halbwegs anonym bleiben wollen. Andererseits hat es für den Studenten den Vorteil, dass er von überall Zugriff auf seine Einstellungen hat und nicht nur von einem bestimmten Rechner.

Ein Vorteil des zweiten Ansatzes aus Sicht des Systems ist die zusätzliche Analysemöglichkeit von bearbeiteten Aufgaben aller Studenten auch unter statistischen Gesichtspunkten. Allerdings ließen sich entsprechende Informationen auch unabhängig sammeln, ohne dabei einen direkten Bezug zu einzelnen studentischen Benutzern herzustellen.

6.2.4. Spielerisches

Ein bisher ebenfalls ignorierter Punkt ist die Frage des Spaßes. Es sollte klar sein, dass den meisten Menschen das Lernen leichter fällt, wenn sie dabei Spaß haben. Deswegen ist die Frage, wie man die Arbeit mit dem System unterhaltsamer gestalten kann, auch für das betrachtete TI-Coaching-System sehr berechtigt. Natürlich sollte dies nicht so weit führen, dass das System vom Studenten nicht mehr als ernsthaftes Werkzeug betrachtet wird.

Es sollten also Versuche gemacht werden, gewisse spielerische Komponenten in das System einzubauen. Eine einfache Möglichkeit besteht zum Beispiel in Statistiken und entsprechenden Bewertungen des Studenten. So könnte ein Student für jede gelöste Aufgabe eine bestimmte Punktzahl erhalten, in der zum Beispiel auch ein Zeitfaktor eine Rolle spielt.

Es sind sicher unzählige Ansätze denkbar, die dazu dienen können, den Studenten zur weiteren Arbeit mit dem System zu motivieren um zu vermeiden, dass er sich aufgrund einer etwas nüchternen Arbeitserfahrung vom System abwendet, bevor dieses irgendeinen echten Lerneffekt entfalten kann.

6.3. Funktionsweise des Frontend

Auch wenn der Entwurf des Frontend nicht der Schwerpunkt dieser Arbeit sein sollte, sollen an dieser Stelle doch einige grundsätzliche Dinge geklärt werden. Dabei spielen die Anforderung des Backend als Systemkern eine entscheidende Rolle. Im vorigen Kapitel wurde zur Kommunikation zwischen beiden Komponenten die Lösungsbeschreibungssprache eingeführt. Diese stellt dabei den Schlüssel zur gewünschten Funktionsweise des Frontend dar.

Das Frontend soll in der Lage sein, die Eingaben des und die Interaktion mit dem Studenten in diese Sprache zu übersetzen und an den Kern weiterzugeben. Gleichzeitig soll sie Aufgaben, die ebenfalls durch die Sprache LBS beschrieben werden, vom Backend entgegen nehmen und dem Studenten in geeigneter Weise präsentieren.

Allerdings kann die LBS nicht die einzige Form der Kommunikation zwischen beiden Komponenten sein, da sie einige Aspekte wie die Kommunikation über erkannte Fehler oder andere Rückmeldungen des Kerns an den Benutzer nicht beinhaltet. Dabei handelt es sich aber um spezifischere Problemstellungen einer Implementierung, die an dieser Stelle nicht genauer behandelt werden.

6.4. Ansätze zur Implementierung des Backend und Systemkerns

In diesem Abschnitt soll, quasi als Endergebnis der in dieser Arbeit gemachten Überlegungen und Ansätze, die Implementierung des Kerns eines Coaching-Systems skizziert werden. Dazu soll zunächst ein typischer Durchlauf des Systems, in dem ein Student eine Aufgabe bearbeitet, im Allgemeinen beschrieben werden.

6.4.1. Die Aufgabenverarbeitung

An erster Stelle steht hierbei die Auswahl eines Aufgabentyps, anhand der für den aktiven Studenten gewählten Konfiguration. Je nach Typ wird dann, falls dies für eine entsprechende Aufgabe möglich und sinnvoll ist, eine zufällige Instanz einer dieser Aufgabe generiert oder eine Aufgabe aus einer Liste mit Standardaufgaben ausgewählt. Das Erzeugen einer Zufallsinstanz zum Beispiel für Automaten, Grammatiken sollte verhältnismäßig leicht möglich sein. Danach wird für die gewählte Aufgabe eine Darstellung in LBS entwickelt und die Aufgabe wird dann in dieser Form an das Frontend geschickt, damit dieses sie dem Studenten präsentieren kann.

Als nächster Schritt beginnt das System, die Aufgabe selbst zu lösen und speichert alle dabei entstehenden Objekte intern ab. Damit besitzt das System eine korrekte Lösung. Zusätzlich können, je nach Aufgabentyp, weitere Dinge, die nicht in unmittelbarem Zusammenhang mit der gestellten Aufgabe stehen, berechnet werden. Ziel ist es, dass das System möglichst viele Informationen über den Gegenstand der Aufgabe hat. So könnte es zu diesem Zeitpunkt zum Beispiel alle abstrakten Repräsentationen einer regulären Sprache ermitteln. Einschließlich der eines minimalen DFA, welcher als gute Grundlage für Vergleiche dienen kann.

Nachdem oder noch während das System interne Lösungen entwickelt, sollte es die ersten LBS-Ausdrücke vom Frontend geliefert bekommen, welche die Lösung des Studenten repräsentiert. Diese kann natürlich auch immer Anfragen des Studenten an

das Systems enthalten. Das System kann nun also damit beginnen, die empfangenen LBS-Ausdrücke zu verarbeiten. Dabei handelt es sich um den eigentlichen Kern der Verarbeitung und diese sollte so lange weiterlaufen, bis der Student seine Lösung vervollständigt hat, oder aber aus irgendeinem Grund entscheidet, die Aufgabe nicht weiter zu bearbeiten.

Den letzten Schritt eines Zyklus der Aufgabenverarbeitung stellt die Aufgabenabwicklung dar. Dabei wird versucht, sofern die Aufgabe nicht abgebrochen wurde, die Lösung vollständig zu analysieren, auf Korrektheit zu prüfen und gegebenenfalls zu bewerten. Das Ergebnis dieses Vorgangs wird dann dem Frontend übermittelt, damit dieses wieder seine Präsentationsaufgabe gegenüber dem Studenten wahrnehmen kann.

Diese vier Schritte beschreiben die wichtigste Funktion des Systems, aber daneben gibt es noch einige andere Dienste, welche es ausführt. Die aktive Ausgabenauswahl durch den Studenten könnte ein Beispiel darauf sein. Aber der beschriebene Zyklus soll Mittelpunkt der Betrachtung bleiben. Im folgenden sollen nun alle vier Schritte genauer beschrieben werden:

1. Ausgabenauswahl
2. Interne Aufgabenverarbeitung
3. Verarbeiten der Lösung des Studenten
4. Aufgabenabwicklung

6.4.2. Ausgabenauswahl

Wie oben beschrieben, soll die automatische Ausgabenauswahl, im Sinne dass der Student nicht eine ganz bestimmte Aufgabe anfordert, durch bestimmte Einstellungen gesteuert werden. Diese könnten sich zum Beispiel auf die gewünschten Aufgabentypen, auf die Länge der Aufgaben oder aber den Schwierigkeitsgrad der Aufgaben beziehen. Dazu muss das System natürlich das eigene Aufgabenrepertoire nach diesen Gesichtspunkten klassifiziert haben.

Während eine Klassifizierung nach Aufgabentypen trivial ist, da diese Unterscheidung bereits bei der Entwicklung getroffen wurde beziehungsweise getroffen werden musste, sind die beiden anderen Klassifizierungen schwerer. Im Falle der Aufgabenlänge, sollte dies aber auch kein Problem darstellen, da diese sich unmittelbar aus dem Aufgabentyp und dem Umfang der gegebenen Objekte ermitteln lässt, wenn auch nur grob. Auch der Schwierigkeitsgrad ist im Prinzip eine Funktion von Aufgabentyp und Umfang, aber zusätzlich unterliegt er sehr subjektiven Maßstäben. Letztendlich sollte man sich dabei beim Entwurf eines bestimmten Aufgabentyps auf irgendein Maß festlegen. Eine subjektive Einschätzung ist immer besser als gar keine. Denn dann müsste man ganz auf das Konzept der Schwierigkeit einer Aufgabe im Rahmen des Systems verzichten.

Wenn das Problem der Ausgabenauswahl, bezüglich Typ, Umfang und Schwierigkeit, gelöst ist, stellt sich als nächstes die Frage nach der Aufgabenerzeugung. Dies kann zum einen durch Auswahl aus einer Datenbank von Aufgaben erreicht werden, zum anderen durch zufällige Generierung von Objekten wie Regulären Ausdrücken und so weiter.

Am sinnvollsten ist die Kombination beider Varianten. Nicht für alle Aufgaben ist die Zufallsgenerierung möglich, und für bestimmte ist sie nicht immer sinnvoll um eine gute Aufgabe zu erzeugen. Ein Beispiel wäre die zufällige Erzeugung eines DFA, der sich durch Minimierung wirklich verkleinern lässt. Gleichzeitig erhöhen zufällige Aufgaben natürlich das Repertoire an unterschiedlichen Aufgaben ungemein.

Sobald die gewählte Aufgabe feststeht, besteht der letzte Schritt in der Generierung der Aufgabenbeschreibung in LBS. Diese besteht im wesentlichen in der Erzeugung eines *Ziel*-Ausdrucks für die Aufgabenstellung und allen nötigen *Definitions*-Ausdrücken zur Beschreibung der behandelten Gegenstände.

6.4.3. Interne Aufgabenverarbeitung

Nachdem die Aufgabenbeschreibung an das Frontend gesendet wurde, kann das System sofort mit der internen Verarbeitung der Aufgabe beginnen. Dies kann zum einen durch das Anstoßen der entsprechenden Verfahren geschehen, zum anderen durch erneuten Zugriff auf die Aufgabendatenbank, welche unter Umständen auch interne Repräsentationen der gespeicherten Aufgaben enthält. Zusätzlich kann auch noch ein Art Cache für Aufgabenlösungen existieren, der zuvor verarbeitete Lösungen enthält. Der Unterschied zwischen der Aufgabendatenbank und dem Cache besteht darin, dass die Aufgabendatenbank bei der Systementwicklung mit bestimmten Aufgaben und Lösungen gefüllt wurde, während der Cache dynamisch vom System selbst verwaltet wird.

Ziel dieses Schritts ist es, möglichst viele Informationen über die Aufgabe, ihre Lösungen und die dabei verwendeten Objekte zu erfahren, um die Grundlage für eine besonders wirkungsvolle Analyse der Studentischen Lösung zu entwickeln. Idealerweise sollt die interne Verarbeitung schnell genug sein, sodass sie abgeschlossen ist, noch bevor das System die ersten Eingaben des Studenten empfängt und verarbeitet.

Aber selbst wenn das System gerade keine internen Aufgaben mehr hat, ist die interne Verarbeitung nicht abgeschlossen. Sinnvollerweise sollten die interne Aufgabenverarbeitung und die Lösungsverarbeitung parallel ablaufen. Wobei die internen Ergebnisse zum einen bei Bedarf bei der Lösungsanalyse abgefragt werden können und zum anderen bestimmte Aktionen des Studenten den Anstoß für neue Berechnungen geben können.

6.4.4. Verarbeitung der Lösung des Studenten

Wie im letzten Abschnitt beschrieben, ist die Verarbeitung der empfangenen LBS-Ausdrücke und damit der Eingaben des Studenten einer von zwei parallelen Abläufen. Die Lösungsverarbeitung läuft dabei sozusagen Hand in Hand mit der internen Aufgabenverarbeitung. Ziel der Lösungsverarbeitung ist es, ein möglichst genaues internes Abbild dessen zu schaffen, was der Student in seiner Lösung beschreibt.

Ein solches Abbild besteht aus verschiedenen in vorherigen Kapiteln schon beschriebenen Komponenten. Dies sind zunächst Antworten auf die Fragen *Was*, *Wie* und *Wozu* der Student gerade auf den verschiedenen Betrachtungsebenen der *Aufgabe*, *Teilaufgabe*, *Lösungsschritt* und *Teilschritt* macht. Darüber hinaus versucht das System

anhand der Eingaben des Studenten ein internes Abbild aller von ihm beschriebenen Objekte zu erstellen.

Während die Antworten auf die Frage, was der Student macht, dazu dienen können, neue interne Verarbeitungen anzustoßen, vervollständigen sie auch das Bild der vom Studenten definierten Objekte. Wenn das System zum Beispiel weiß, dass der Student einen Automaten mit einer ganz bestimmten Eigenschaft konstruiert, kann er mit dem Wissen über diese Eigenschaften und der internen Repräsentation des vom Studenten entwickelten Automaten versuchen, diese auf Fehler zu überprüfen.

Das Stichwort der Fehler ist auch einer der wichtigsten Aspekte der Verarbeitung, denn diese zu erkennen ist einer der Hauptgründe dafür, warum es überhaupt Sinn macht, ein internes Abbild der vom Studenten verwendeten Objekte sowie interne Lösungen zu entwickeln. Durch das Abgleichen dieser beiden lassen sich sehr wirkungsvoll Fehler finden. Aber nicht für alle Fehler ist eine interne vom Programm entwickelte Lösung überhaupt notwendig. Sofern das System ermitteln kann, welche Art von Objekten der Student in seiner Lösung benutzt, kann es diese auch auf einfache Fehler überprüfen.

Beispiele für solch einfache Fehler könnte zum Beispiel die Verwendung von nicht im Alphabet oder der Zustandsmenge enthaltenen Terminalen oder Zuständen in der Übergangsfunktion eines DFA sein. Oder die Definition einer Grammatik als 5-Tupel und so weiter. Diese einfachen Fehler lassen sich sehr leicht erkennen, sofern die Objekte in der LBS mit entsprechenden Konstanten bezeichnet werden, was für das Frontend in den meisten Fällen kein Problem darstellen sollte.

Wenn nun entsprechende Fehler erkannt wurden, kann das System sehr leicht, anhand der Konfiguration des genauen Fehlerverhaltens, entsprechend reagieren, indem es zum Beispiel Fehlermeldungen an das Frontend weiter gibt und auf die Reaktion des Studenten wartet. Unabhängig von der genauen Konfiguration sollte das System jedoch sämtliche erkannten Fehler protokollieren.

Ein weiterer Aspekt der Lösungsverarbeitung, verarbeitet eigentlich gar keine Lösung, sondern bestimmte Anfragen des Studenten. Diese können sich auf die eigene Lösung beziehen, oder auch Hilfeanfragen sein. Anfragen zur eigenen Lösung sind die besprochenen Rückschauen, während Hilfeanfragen Vorschauen sind. Antworten auf eine Rückschau lassen sich durch die Fehlerüberprüfung der bisherigen Lösung durchführen, während die Vorschau die intern berechnete Lösung verwenden, um dem Studenten eine befriedigende Antwort zu ermöglichen.

Die Lösungsverarbeitung wird beendet, sobald ein Student die Aufgabe als abgeschlossen erklärt oder die Bearbeitung der Aufgabe abbricht. In jedem Fall führt das zur Ausführung der Aufgabenabwicklung. Die interne Aufgabenverarbeitung wird dadurch jedoch nicht mit beendet. Auch die internen Abbilder der vom Studenten erzeugten Objekte werden nicht verworfen, insbesondere nicht das Objekt, welches das Endergebnis darstellt.

6.4.5. Aufgabenabwicklung

Sobald der Student seine Eingabe für beendet erklärt, startet der Prozess der Aufgabenabwicklung. Hierbei wird die abgegebene Lösung, falls vorhanden, als Ganzes

analysiert und das Endergebnis auf Korrektheit überprüft. Als Ergebnis dieser Analyse wird eine Bewertung der Lösung vorgenommen und das Ergebnis mit allen vom Studenten gewünschten Informationen an das Frontend gesendet. Welche Informationen den Studenten genau interessieren, wird dabei wieder durch entsprechende Einstellungen gesteuert. Trotz ihrer Wichtigkeit für den Lerneffekt stellt die Aufgabenabwicklung den wohl am einfachsten zu implementierenden der vier betrachteten Schritte dar.

6.5. Datenspeicherung

Um eine effizientere interne Verarbeitung zu ermöglichen, bietet es sich an, bestimmte Dinge zu speichern, sodass das System nicht immer alles selbst berechnen muss. Dies erlaubt auch die Aufnahme bestimmter Aufgaben, die sich nicht einfach automatisch lösen lassen. Solche Aufgaben kann man als Teil der Implementierung zusammen mit einer für die Analyse geeigneten Darstellung einer Lösung in einer bestimmten Datenbank speichern.

Eine weitere Variante der Datenspeicherung zur Verbesserung der Ausführungszeit interner Vorgänge stellt das Caching dar. Dabei werden bereits berechnete Lösungen zusammen mit den zugehörigen Aufgaben abgespeichert, um sie später bei Bedarf schnell abrufen zu können.

6.6. Abschluss und Ausblick

Die in diesem Kapitel beschrieben Ansätze einer Implementierung zusammen mit denen zur Lösung ganz bestimmter Probleme, die in den vorigen Kapiteln entwickelt wurden, sollten einen bescheidenen ersten Schritt auf dem Weg zu einer tatsächlichen Implementierung eines entsprechenden Systems darstellen.

Eine Frage, die sich noch stellt ist, ob die gewählte Methode des automatischen Beweisens tatsächlich sinnvoll zur Lösung der im Rahmen des Coaching-Systems auftretenden Probleme ist, oder ob ein anderer Ansatz nicht vielleicht geeigneter sein könnte. Bei der Verwendung des Ansatzes zur Beschreibung der Algorithmen für Reguläre Sprachen hat sich gezeigt, dass die geringe Kontrolle über die Ausführungsreihenfolge ein nicht unbeträchtliches Problem darstellt. Gleichzeitig ließen sich gewisse Zusammenhänge, die Grundlage der Verfahren waren, in fast natürlicher Weise logisch formulieren.

Letztendlich hat jede in der Einleitung vorgestellte Variante zur Implementierung, ihre Vor- und Nachteile. Die Betrachtung dieser Varianten und vielleicht auch eine Antwort darauf, welche Vorteile für die Probleme, die das System aufwirft, die Besten sind, und welche Nachteile am wenigsten störend sind, könnte Teil einer anderen Arbeit sein.

Der Versuch, zunächst einige Grundlagen zur Implementierung zu betrachten, zu analysieren und zu diskutieren, hat dazu geführt, dass diese Arbeit über die reinen Ansätze zur Lösung der einzelnen Probleme nicht hinaus gekommen ist. Aber es bleibt die Hoffnung, dass die hier vorgestellten Ansätze, zumindest in Teilen, als Grundlage für eine tatsächliche Implementierung eines entsprechenden Systems dienen können.

Literaturverzeichnis

[RuNo95] ***Russell, Stuart; Norvig, Peter***: Artificial Intelligence A Modern Approach. Upper Saddle River, New Jersey 1995.

[Schö97] ***Schöning, Uwe***: Theoretische Informatik - kurzgefaßt (3. Auflage). Heidelberg, Berlin 1997.

[HMU01] ***Hopcroft, John E.; Motwani, Rajeev; Ullman Jeffrey D.***: Introduction to Automata Theory, Languages, and Computation (Second Edition). Boston, San Francisco, New York 2001.

[Bib92] ***Bibel, Wolfgang***: Deduktion - Automatisierung der Logik. München 1992.