Universität Karlsruhe (TH)
Institut für Algorithmen und
Kognitive Systeme
der Fakultät für Informatik

Prototypische Umsetzung einer natürlichsprachlichen Verhaltensbeschreibung in eine unscharfe metrisch temporale Logikdarstellung

Diplomarbeit
von
M. Arens

Betreuer: Dipl. Inform. M. Middendorf, Prof. Dr. H.–H. Nagel

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Karlsruhe, den 18. Mai 2001

Danksagung

Mein Dank gilt in erster Linie meinen Betreuern Herrn Prof. Dr. H.–H. Nagel sowie Herrn Dipl. Inform. M. Middendorf. Herr Prof. Nagel fand neben seinen diversen anderen Aufgaben die Zeit, frühere Fassungen dieser Arbeit gewissenhaft zu durchleuchten und mich auf thematische und terminologische Ungenauigkeiten hinzuweisen. Von seinem Drängen zu einem systematischen Vorgehen und einer klaren Benennung der Dinge habe ich sehr profitiert. Herr Middendorf hat mir bei der Lösung so mancher technischer Frage mit großem Einsatz geholfen.
Ein großes Dankeschön gilt auch Herrn Artur Ottlik. Zahlreiche Diskussionen mit ihm haben mir immer wieder neue Anregungen gebracht und mich gleichzeitig dazu gezwungen, bisherige Ideen immer wieder neu zu überprüfen.
Dank auch den aktuellen bzw. ehemaligen Assistenten am Institut, Herrn Dr. R. Gerber, Herrn Dr. H. Leuck und Herrn Dr. Th. Müller. Auf ihre Hilfe bei kleinen oder nicht ganz so kleinen Problemen konnte man stets bauen.
Schließlich gilt mein Dank auch Herrn J. Dieterich, Herrn D. Eisenhardt sowie Herrn T. Rath, die immer wieder die Geduld aufbrachten, sich meine Geschichten über den aktuellen Stand der Diplomarbeit anzuhören.

M. Arens

Kurzfassung

Am Institut für Algorithmen und Kognitive Systeme der Universität Karlsruhe (TH) wurde mit `XTRACK` [Koller 92; Kollnig 95; Haag 98; Gerber 2000; Leuck 2000] ein System entwickelt, um Videobildfolgen von Straßenverkehrsszenen auszuwerten, die darin erkannten Fahrzeuge zu verfolgen und deren Aktionen in Form von natürlichsprachlichen Texten auszugeben. Dies setzt zunächst leistungsfähige und robuste signalnahe Bildauswertungskomponenten voraus. Mit Hilfe von geeignet repräsentiertem physikalischem und geometrischem Wissen über die beobachtete Szene können die Ergebnisse dieser Bildauswertungskomponenten zu abstrakteren Begriffen aggregiert werden. Weiteres Wissen über die möglichen Aktionen, die Absichten und somit das zu *erwartende* Verhalten von Verkehrsteilnehmern – wiederum in geeigneter Weise repräsentiert – ermöglichen einerseits die Rückkopplung auf die signalnäheren Schichten des Bildauswertungssystems zur Steigerung der Robustheit, andererseits auch die natürlichsprachliche Beschreibung des Beobachteten.
Zur Modellierung von Situationen und Handlungen schlägt [Krüger 91] den Formalismus der Situationsgraphenbäume vor, der gerade die Repräsentation komplexer begrifflicher Beschreibungen ermöglicht. [Schäfer 96] erweitert diesen Formalismus um besondere unscharfe und zeitliche Aspekte und zeigt, wie Situationsgraphenbäume in unscharfe, metrisch temporale Logik–Programme übersetzt werden können. Trotz der dort vorgestellten Beschreibungssprache `SIT++` für Situationsgraphenbäume ist der Implementierungsaufwand jedoch schon bei einfachen Verhaltensbeschreibungen beträchtlich.
Wünschenswert wäre es, natürlichsprachliche Verhaltensbeschreibungen automatisch in einen entsprechenden Situationsgraphenbaum und die zugehörige Logikrepräsentation zu überführen. Diese Verhaltensbeschreibungen könnten dann auch von Menschen verfaßt werden, die zwar mit dem betrachteten Diskursbereich, nicht aber notwendigerweise mit dem internen Aufbau des Bildauswertungssystems – sowie der zur Umsetzung der Resultate der Bildfolgenauswertung in eine natürlichsprachliche Beschreibung herangezogenen Programmsysteme – vertraut sind. Die dazu notwendigen Schritte zu untersuchen, ist Ziel dieser Arbeit.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Heutige Bildfolgenauswertungssysteme beschränken sich nicht mehr nur auf das bloße Berechnen von Bildbereichshinweisen wie Kantensegmenten oder Änderungen in aufeinander folgenden Bildern. Die Zunahme der verfügbaren Rechnerleistung, vor allem aber die Integration von *Wissen* über die betrachtete Szene in das Auswertungssystem, ermöglichen es vielmehr, die von einer Kamera aufgenommenen bewegten Objekte zu erkennen und auch unter schwierigen Bedingungen wie etwa in Verdeckungssituationen zu verfolgen [Haag 98].
Das integrierte Wissen stammt dabei aus unterschiedlichen Quellen, wird in unterschiedlichen Teilprozessen des Bildauswertungssystems verwendet und muß dementsprechend auch unterschiedlich repräsentiert werden [Haag 98; Haag & Nagel 2000]. Ein Kameramodell etwa beinhaltet Wissen über den Abbildungsvorgang der Kamera in Form von Matrizen und Vektoren und ermöglicht so den Schluß von Merkmalen im Bild auf Merkmale in der Szene. Objektmodelle in Form von Polyedermodellen und Bewegungsmodelle in Form von Differenzengleichungen stellen Wissen über die Form und die Art der Bewegung von zu beobachtenden Objekten dar und erleichtern so deren Auffinden und Verfolgen im Bild. Beleuchtungsmodelle in Form von Gleichungen erleichtern weiter das Auffinden von Objekten durch Wissen über den zu erwartenden Schattenwurf. Szenenmodelle in Form von Polyedermodellen ermöglichen eine Objektverfolgung, die auch in Verdeckungssituationen robust bleibt. *Situationsmodelle* in Form von unscharfen, metrisch temporalen Logikprogrammen schließlich erlauben es, Wissen über das *Verhalten* der zu beobachtenden Objekte in das Bildfolgenauswertungssystem einzubringen. Dadurch können primitive Ergebnisse des Auswertungsprozesses – wie etwa die Positionsänderung eines Objekts innerhalb einer bestimmten Zeitspanne – zu komplexeren Begriffen – wie etwa Bewegungsverben – abstrahiert und auf dieser Abstraktionsebene dann Voraussagen über kommende Aktionen der Objekte getroffen werden. Dies verbessert einerseits wiederum den Auswertungsprozeß, andererseits erleichtert es die Inspektion der Auswertungsergebnisse des Systems durch den

menschlichen Benutzer. Waren vor diesem Schritt ggf. lange Zahlenkolonnen zu kontrollieren, geben nun automatisch erzeugte, begriffliche [Kollnig 95] oder gar natürlichsprachliche [Gerber 2000] Beschreibungen Auskunft über das erkannte Geschehen in der Szene.
Zur Modellierung von Wissen über das Verhalten von zu beobachtenden Objekten, die dann Akteure (*agents*) genannt werden, führt [Nagel 88] den Begriff der "generisch beschriebenen Situation" (*generically described situation*, *gd_situation*) ein. Ein solches Situationsschema beinhaltet sowohl Angaben über den Zustand des Akteurs und seiner Umwelt, als auch Angaben über dessen Handlungen und Absichten. [Krüger 91] verwendet zur Darstellung von Situationsschemata Situationsgraphenbäume. Darin wird in Form von Graphen die zeitliche Abfolge von Situationen modelliert. Diese Graphen sind zusätzlich in eine Baumstruktur eingebettet, wodurch Situationen auch hierarchisch je nach ihrer Spezialisiertheit angeordnet sind. [Schäfer 96] zeigt, daß solche Situationsgraphenbäume in Programme einer unscharfen, metrisch temporalen Logiksprache überführt werden können. Der Autor erweitert den Formalismus um unscharfe und zusätzliche zeitliche Aspekte und stellt die Beschreibungssprache `SIT++` vor, in der sich Situationsgraphenbäume formulieren lassen, die dann automatisch in Logikprogramme übersetzt werden.

Trotz der oben geschilderten Fortschritte im Bereich der Wissensrepräsentation ist der Aufwand zur Erstellung schon eines einfachen Situationsgraphenbaumes recht hoch, wie [Haag & Nagel 2000] exemplarisch zeigen. Desweiteren muß der Benutzer, der sein Wissen über den betrachteten Diskursbereich dem Bildauswertungssystem zur Verfügung stellen will, noch in sehr hohem Maße mit der Arbeitsweise des Systems und den verwendeten Formalismen vertraut sein. Wünschenswert wäre es, den Benutzer in die Lage zu versetzen, *mit seinen eigenen Worten*, also in Form einer natürlichsprachlichen Verhaltensbeschreibung, dem System sein Wissen zu vermitteln.

1.2 Ziel der Arbeit

[Kamp & Reyle 93] bieten mit der von ihnen vorgestellten Diskurs–Repräsentationstheorie (DRT) ein sehr detailliertes Verfahren, um englischsprachige Texte zu verarbeiten und die Semantik des Textes in eine rechnerinterne Struktur – die Diskurs–Repräsentationsstruktur (DRS) – zu überführen. [Arens & Ottlik 2000] entwickelten auf dieser Basis ein Werkzeug, das es erlaubt, durch Angabe einer Grammatik (etwa für ein Fragment des Englischen) und entsprechender Konstruktionsregeln automatisch ein System erzeugen zu lassen, das Texte, die dieser Grammatik genügen, in DRSen umformt.
Ausgehend von diesen Grundlagen soll im Rahmen der vorliegenden Arbeit zunächst untersucht werden, inwieweit sich die von [Kamp & Reyle 93] entwickelte Diskurs–

Repräsentationstheorie und das dazu von [Arens & Ottlik 2000] erstellte Werkzeug dazu eignen, *Verhaltensbeschreibungen* zu verarbeiten und das darin enthaltene Wissen über Akteure sowie deren Handlungen und Absichten in Form von DRSen zu repräsentieren. Darauf aufbauend wird dann eine automatische Übersetzung von DRSen in Situationsgraphenbäume und die zugehörigen Logikprogramme zu erarbeiten und zu implementieren sein. Zusammenfassend soll untersucht werden, inwieweit sich natürlichsprachliche Texte zur Bereitstellung von Wissen über das Verhalten von beobachteten Akteuren in der Bildfolgenauswertung eignen und wie dieses Wissen in das bestehende Bildfolgenauswertungssystem `XTRACK` am Institut für Algorithmen und Kognitive Systeme der Universität Karlsruhe (TH) automatisch integriert werden kann.

1.3 Aufbau der Arbeit

Kapitel 1 wird abgeschlossen mit einem kurzen Überblick über einige Arbeiten, welche sich mit ähnlichen Zielen wie die hier angestrebten beschäftigen. Kapitel 2 stellt die Grundlagen vor, welche zum Erreichen der hier angestrebten Ziele notwendig sind. Im einzelnen sind dies die Grundzüge der von [Schäfer 96] entwickelten Logiksprache UMTL, die Vorgehensweise bei der Modellierung von Verhalten durch Situationsgraphenbäume sowie eine kurze Vorstellung der von [Kamp & Reyle 93] entwickelten Diskurs–Repräsentationstheorie.
Kapitel 3 beschäftigt sich mit der Erstellung von DRSen speziell aus natürlichsprachlichen Verhaltensbeschreibungen. In Kapitel 4 werden einige Verfahren zur Repräsentation sowie zur grafischen Darstellung von Situationsgraphenbäumen untersucht.
Die Überführung von DRSen in Situationsgraphenbäume ist das Thema von Kapitel 5, in dem ein Verfahren hierzu entwickelt wird. Kapitel 6 beschreibt die Implementierung diese Verfahrens und führt die Transformation einer DRS in ein Verhaltensschema beispielhaft vor.
Kapitel 7 schließt die Arbeit mit einer Bewertung des Erreichten sowie einem Ausblick auf noch ausstehende Aufgaben ab.

1.4 Literaturüberblick

[Haag 98] und [Gerber 2000] geben einen Überblick über aktuelle Bild– und Bildfolgenauswertungssysteme. Viele der genannten Systeme verwenden Modellwissen, um die Robustheit des Auswertungsprozesses zu erhöhen oder um begriffliche bzw. natürlichsprachliche Beschreibungen des vom System Beobachteten zu erzeugen. Nur wenige Systeme (vgl. [Schirra 94]) betrachten bisher auch die umgekehrte Richtung, also die Erzeugung von Bildern oder Bildfolgen aus natürlichsprachlichen Texten, etwa zur Kontrolle des Bildauswertungssystems. Eine weitergehende Nutzung natürlichsprachlicher Texte zur Erzeugung des Modellwissens selbst, wie in der vorliegenden Arbeit geplant, ist in keinem der bekannten Systeme anzutreffen. Im Folgenden soll deshalb zunächst

exemplarisch ein eigenständiges System vorgestellt werden, das natürlichsprachliche Texte verarbeitet und in eine Wissensbasis überführt. Ein weiterer Abschnitt stellt dann eine Arbeit vor, die sich mit der Wissenserlangung (*knowledge acquisition*) speziell durch Endbenutzer beschäftigt und damit ähnliche Ziele verfolgt wie die vorliegende Arbeit. In einem dritten Abschnitt wird dann eine Arbeit behandelt, die sich mit dem Verstehen gesprochener Sprache beschäftigt. Das dort vorgestellte System generiert aus ihm übergebenen Sprachdefinitionen ein Sprachverständnis–System und kann aufgrund dieser Vorgehensweise als Vergleich zu dem in der vorliegenden Arbeit angestrebten System dienen.

1.4.1 Das System *Spaceprobe*

[Dinsmore 91] stellt mit *Spaceprobe* ein logikbasiertes System zur Repräsentation komplexer natürlichsprachlicher Diskurse vor. Sein Hauptanliegen ist hierbei, auf möglichst direktem Weg von der linguistischen Analyse eines Textes hin zu einer logik–basierten Darstellung des Textinhalts zu gelangen, die vor allem das Schliessen aus dieser Darstellung heraus erlaubt. *Spaceprobe* basiert auf dem Prinzip der Partitionierten Repräsentation (*partitioned representation*), bei der ein Gesamtdiskurs in mehreren kleineren sog. mentalen Räumen (*mental spaces*) repräsentiert wird. Dies ermöglicht es dem System, sowohl zeitlich begrenzte Zusammenhänge als auch hypothetische oder fiktive Teildiskurse zu repräsentieren.
Die Überführung eines Textes gliedert sich dabei in drei Teilschritte:
Das Zerteilen (*parsing*) übersetzt den Text satzweise in eine Zwischendarstellung, die einer prädikatenlogischen Beschreibung des Satzinhaltes gleicht. Zusätzlich wird in diesem Schritt zu dem gerade zu bearbeitenden Satz ein sog. Raumhinweis (*space cue*) erzeugt. Anhaltspunkte für diesen Raumhinweis können dabei sowohl zeitliche Adverbialphrasen des Satzes bzw. die Zeitform des Prädikats als auch bestimmte Verben und Verbkonstrukte sein, die eine Überzeugung (*belief*) oder eine Fiktion ausdrücken. Ein zweiter Schritt – Kontextualisierung (*contextualization*) genannt – wandelt die so erzeugte vorläufige Repräsentation aller Sätze in eine Folge von Fakten der Form

$$S \; ! \; P$$

um, wobei S den Raum und P die prädikatenlogische Form eines Satzes des Textes bezeichnet. Hierbei wird gleichzeitig versucht, Mehrdeutigkeiten, etwa durch das Auftreten von Pronomen oder bestimmten Artikeln aus dem Kontext heraus aufzulösen. Im dritten und letzten Schritt (*digestion*) schließlich wird die oben erzeugte Faktenliste anhand von Restrukturierungsregeln (*restructuring rules*) in ihre endgültige und meist einfacherer Form gebracht. Restrukturierung findet vor allem bei der Behandlung von Überzeugungen, Hypothesen und Fiktionen Verwendung, da diese Sprachkonstrukte zunächst in Fakten resultieren, die implizit wiederum auf andere mentale Räume verweisen. Diese implizit gegebenen Räume werden durch die Restrukturierungsregeln dann explizit in Form von weiteren Fakten erzeugt. Der implizite Verweis in den ur-

sprünglichen Fakten wird durch einen expliziten Verweis auf diese neuen Fakten vereinfacht. Ergebnis des Verarbeitungsprozesses ist somit eine Liste von Fakten in Form prädikatenlogischer Ausdrücke, von denen jeder einem mentalen Raum zugeordnet ist. Der Autor verweist auf die Ähnlichkeit seiner mentalen Räume zu den von [Kamp & Reyle 93] eingeführten DRSen. Während jene aber von der Seite der Linguistik motiviert seien, sieht [Dinsmore 91] sein System als von der Logik ausgehend. Er demonstriert seinen Ansatz an einem einfachen Beispiel. Dies läßt weder eine Behandlung von unscharfen oder vagen Ausdrücken erkennen noch kann sein System zeitliche Aspekte über rein qualitative Aussagen hinaus repräsentieren. Aus diesem Grund scheint sein Ansatz für die Bereitstellung von Situationsmodellen für die Bildfolgenauswertung ungeeignet.

1.4.2 *EMeD – EXPECT* Method Developer

Viele bisherige Ansätze zur Wissenserlangung setzen neben dem Bereichsexperten, der das Wissen besitzt, welches in einer Wissensbasis repräsentiert werden soll, vor allem auch einen sog. Wissensingenieur (*knowledge engineer*) voraus, der als Mittler zwischen Mensch und Rechner fungiert. [Kim & Gil 2000] beschäftigen sich mit der Frage, wie Problemlösungswissen direkt von Endbenutzern erlangt werden kann. Ihr System *EMeD* (*EXPECT Method Developer*) baut auf dem System *EXPECT* (s. [Gil & Melz 96]) auf, mit dem Wissen über Objekte, vor allem aber Wissen über Methoden, mit denen die Manipulation dieser Objekte zum Lösen bestimmter Aufgaben möglich ist, repräsentiert werden kann. Diese Methoden beschreiben, wie bestimmte Aufgaben erreicht werden können. Aufgaben werden hierbei als Zielhierarchien spezifiziert, bei denen jedes Ziel in einfacherer Unterziele aufgeteilt wird. Das Wissen wird dazu in einer systemeigenen Beschreibungssprache formuliert, die sich durch verschachtelte Methodenaufrufe und Kontrollstrukturen (*if-then-else*) auszeichnet. Genau an diesem Punkt setzt nun *EMeD* an. Es ist als Schnittstelle zwischen dem Bereichsexperten und der Wissenbasis gedacht, soll also – wie vorher der Wissensingenieur – den Endbenutzer beim Einbringen seines Wissens in das System unterstützen. Dies geschieht, indem *EMeD* ein von *EXPECT* bereitgestelltes Abhängigkeitsmodell (*Interdependency Model*) auswertet. Dieses Abhängigkeitsmodell beinhaltet Informationen über schon vorhandenes und noch fehlendes Wissen sowie Informationen über Abhängigkeiten innerhalb der Wissensbasis. Der Endbenutzer kann so auf drei verschiedene Arten bei der Wissenseingabe unterstützt werden:

1. Fehlende Teile der Wissensbasis, wie zwar verwendete, aber noch nicht definierte Methoden, werden angezeigt.

2. Relationen zwischen Wissensteilen werden angezeigt und beschrieben.

3. Auf Inkonsistenzen, wie etwa einen falschen Rückgabetyp einer Methode, wird hingewiesen.

Weitere Hilfen, wie eine syntaktische Hilfe bei der Eingabe von Methoden, erleichtern darüber hinaus die Benutzung des Systems.
Zur Wissensbereitstellung werden bei *EMeD* keine natürlichsprachlichen Texte ausgewertet. Die Modellierung von Wissen erfolgt durch die oben erwähnte Beschreibungssprache. [Kim & Gil 2000] zeigen jedoch, daß durch die Ausnutzung des Abhängigkeitsmodells zur Unterstützung des Endbenutzers dessen Arbeit sehr erleichtert und seine Leistungsfähigkeit bei der Wissensmodellierung fast verdoppelt werden kann. Die Autoren weisen allerdings daraufhin, daß sie eine zu geringe Zahl an Testpersonen zur Verfügung hatten, als daß ihre Aussage statistisch gesichert sei. Die qualitative Aussage, daß nämlich dem Endbenutzer als Bereichsexperten durch geeignete, ggf. wie hier interaktive Eingabesysteme direkt die Wissensmodellierung ermöglicht werden kann, bleibt jedoch bestehen.

1.4.3 Das *Gemini*–System

Das von [Dowding et al. 93] vorgestellte System *Gemini* ist ein System zum Verstehen natürlicher Sprache. Entwickelt wurde es speziell für die Anwendung im Bereich der Verarbeitung gesprochener Sprache. Gesprochene Sprache unterscheidet sich von natürlichsprachlichen Texten in vielen Punkten. Neben rein durch die Verarbeitung des Sprachsignals auftretenden Fehlern sind gesprochene Äußerungen oft unvollständig. Ein Sprecher unterbricht sich selbst, bricht angefangene Äußerungen ab und setzt sie neu fort. Aus diesen Gründen legten die Autoren bei der Entwicklung des Systems sehr viel Wert auf dessen Robustheit. Einschränkungen durch die Spezialisierung auf gesprochene Sprache können jedoch nicht festgestellt werden, wodurch die in [Dowding et al. 93] vorgestellten Methoden sich auch auf textuell vorliegende natürlichsprachliche Äußerungen anwenden lassen.
Gemini ist jedoch nicht als festes System zu sehen, also insbesondere nicht festgelegt auf eine bestimmte Sprache. Vielmehr werden dem Grundsystem in Form von Textdateien etliche Sprachdefinitionen übergeben, aus denen dann in Verbindung mit einen System–Kern das eigentliche Sprachverarbeitungssystem generiert wird. Wesentlicher Bestandteil dieser *variablen* Sprachdefinitionen ist neben dem Lexikon zunächst die sogenannte Element–Grammatik (*constituent grammar*). Diese gibt an, auf welche Weise Wörter der zu verarbeitenden Sprachäußerung Teilphrasen bilden können. Das System erkennt diese Teilphrasen dann unabhängig von ihrer Position in der Gesamtäußerung. Neben diesen rein syntaktischen Regeln enthält die Element–Grammatik auch semantische Regeln, welche die erkannten Teilphrasen schon in diesem frühen Stadium der Verarbeitung mit logischen Ausdrücken assoziieren. Interessant ist hierbei vor allem, daß zwar jede semantische Regel eine entsprechende (also dieselbe Teilphrase beschreibende) syntaktische Regel voraussetzt, diese Zuordnung aber nicht ein–eindeutig ist. Zu einer syntaktischen Regel können durchaus mehrere semantische Regeln definiert werden. Die dadurch entstehenden Mehrdeutigkeiten werden später behandelt.
Neben der Element–Grammatik beinhaltet die Sprachdefinition auch eine sogenannte

Äußerungs–Grammatik (*utterance grammar*). Diese gibt an, wie sich Gesamtäußerungen aus den von der Element–Grammatik gebildeten Teilphrasen zusammensetzen. Da diese Analyse aller Teilphrasen–Kombinationen wiederum nicht eindeutig sein muß, findet der aus der Äußerungs–Grammatik gebildete Zerteiler zunächst alle zulässigen Äußerungen, von denen dann entsprechend vorgegebener Bewertungskriterien die *beste* ausgewählt wird. Bewertungskriterien ergeben sich zum einen aus den Grammatikregeln selbst. Diese können mit bestimmten Nebenbedingungen ausgestattet werden, die sowohl syntaktische als auch semantische Bedingungen widerspiegeln. Als Beispiel für eine syntaktische Nebenbedingung sei hier die Übereinstimmung von Subjekt und Prädikat eines Satzes im Numerus genannt. Eine semantische Nebenbedingung kann beispielsweise aus der Forderung bestehen, daß das direkte Objekt eines Verbs von einer bestimmten *Sorte* ist (die Autoren nennen hier das Beispiel eines Systems zur Flug–Information. In diesem Diskursbereich wurden an direkte Objekte des englischen Verbs *depart* (verlassen, hier: abfliegen von) die Forderung gestellt, ein Flughafen zu sein.). Sorteninformationen, wie sie hierfür notwendig sind, müssen daher für jedes Wort der Sprache im Lexikon vorhanden sein.
Bleiben nach der Betrachtung dieser Nebenbedingungen und entsprechender Aussortierung von in diesem Sinne ungültigen Interpretationen immer noch mehrere Möglichkeiten für die Struktur und Bedeutung der untersuchten Äußerung, werden diese mit Hilfe von Heuristiken bewertet und die beste Interpretation wird als zu treffende ausgewählt. Als Beispiele für eine solche Heuristik nennen die Autoren zum einen die der Minimalen Anhängung (*Minimal Attachment*), welche Strukturen als besser bewertet, die aus wenigen Knoten bestehen. Zum anderen stellen sie die Heuristik der Rechts–Assoziation (*Right Association*) vor, welche diejenigen Strukturen bevorzugt, die gleiche Teilstrukturen sehr weit rechts und damit später in die Gesamtstruktur integriert. Man kann sich leicht klar machen, daß sich diese Heuristiken in vielen Fällen widersprechen können. Die Autoren weisen hierauf auch hin, halten die bisher damit erzielten Ergebnisse aber für in den meisten Fällen akzeptabel.
Der letzte Schritt der Verarbeitung einer Äußerung besteht in der endgültigen Kombination der von den semantischen Regeln gelieferten logischen Ausdrücken einzelner Teilphrasen zu einem logischen Ausdruck für die erkannte Äußerung. Hierbei geht es vor allem darum, den Gültigkeitsbereich von in den logischen Ausdruck zu integrierenden Quantoren zu bestimmen (*quantifier scoping*). Die dabei auftretenden Mehrdeutigkeiten werden wiederum mit Hilfe bestimmter Vorzugsregeln, Bewertungskriterien und Heuristiken aufgelöst. Ergebnis dieses letzten Schrittes ist ein logischer Ausdruck, der den Inhalt der Äußerung widergibt und zur weiteren Verarbeitung – etwa durch ein Anfragesystem – bereit gestellt wird.

Kapitel 2

Grundlagen

2.1 Unscharfe metrisch temporale Logik

Logikformalismen, wie etwa die Prädikatenlogik erster Stufe (PL1), werden wegen ihrer vergleichsweise langen Geschichte und damit verbundenen Ausgereiftheit häufig als Mittel zur Wissensrepräsentation eingesetzt. Andere Formalismen lassen sich oft in eine Logikform überführen und ausgereifte Verfahren zur Inferenz lassen Logik darüber hinaus als vorteilhaft erscheinen. Beim Einsatz von Logikformalismen zur begrifflichen Bildfolgenauswertung treten jedoch zunächst einige Probleme auf:

- Bildfolgen stellen gegenüber Einzelbildern *zeitveränderliche* Zusammenhänge bildlich dar. Ein Formalismus zur begrifflichen Auswertung dieser Zusammenhänge muß dieser zeitlichen Komponente Rechnung tragen.

- Ein System zur Bildfolgenauswertung ist allein schon aufgrund von technischen Aspekten einem Rauschen unterworfen. Die daraus resultierende *Unsicherheit* der Auswertungsergebnisse sollte in dem herangezogenen Formalismus zur begrifflichen Auswertung repräsentierbar sein.

- Von Menschen benutzte Begriffe sind in den meisten Fällen *vage*. Sollen also für den Menschen verständliche Begriffe die Auswertungsergebnisse des Systems beschreiben, sollte sich die *Vagheit* dieser Begriffe im Repräsentationsformalismus ausdrücken lassen.

Ein Formalismus wie die oben erwähnte PL1 scheint diesen Aufgaben nicht gerecht zu werden. Weder zeitliche Aspekte noch *Unschärfe* aufgrund von Unsicherheit oder Vagheit können in diesem Formalismus direkt repräsentiert werden. [Schäfer 96] erweitert deshalb die PL1 um eine zeitliche und unscharfe Komponente.

2.1.1 Metrisch temporale Logik MTL

[Schäfer 96] benutzt zunächst die von [Brzoska 94] entwickelte Metrisch Temporale Logik (*MTL*) als Grundlage für weitere Schritte. Diese legt zusätzlich zur PL1 eine Zeitstruktur $(\mathcal{T}, t_0, <)$ zugrunde, wobei $\mathcal{T}$ eine Menge von Zeitpunkten, $t_0 \in \mathcal{T}$ einen Startzeitpunkt und "$<$" eine irreflexive, transitive und asymmetrische Vorgängerrelation auf $\mathcal{T}$ bezeichnet. [Brzoska 94] erweitert die PL1 um die zeitlichen Operatoren $\Diamond_S\mathcal{F}$, der die zeitlich–existentielle Gültigkeit (*irgendwann innerhalb* des Zeitintervalls $S \subseteq \mathcal{T}$) der Aussage $\mathcal{F}$ ausdrückt, und $\Box_S\mathcal{F}$, der entsprechend die zeitlich–universelle Gültigkeit (*immer innerhalb* des Zeitintervalls $S \subseteq \mathcal{T}$) der Aussage $\mathcal{F}$ ausdrückt. Als verkürzende Schreibweise für die Spezialfälle $\Diamond_{\{1\}}\mathcal{F} \equiv \Box_{\{1\}}\mathcal{F}$ (bzw.$\Diamond_{\{-1\}}\mathcal{F} \equiv \Box_{\{-1\}}\mathcal{F}$) führt [Brzoska 94] die temporalen Operatoren $\circ\mathcal{F}$ (bzw. $\bullet\mathcal{F}$) ein. Diese werden als "*next*" (bzw. "*previous*") bezeichnet und treffen entsprechend Aussagen über den nächsten (bzw. vorangegangenen) Zeitpunkt. Im Gegensatz zur PL1 hängt bei der MTL dann die temporallogische Interpretationsfunktion $\mathcal{I}$ auch vom jeweiligen Zeitpunkt $t \in \mathcal{T}$ ab. [Schäfer 96] zeigt, wie Formeln der MTL in solche der PL1 überführt werden können. Für einen effizienten Einsatz dieses Logikformalismus schränkt der Autor die MTL auf ihr hornlogisches Fragment (MTHL) ein. Er erweitert den Tableaukalkül für die PL1 zu einem Tableaukalkül für die MTHL und weist dessen Vollständigkeit und Korrektheit nach.

2.1.2 Unscharfe Logik FL1

Die oben vorgestellte MTL ist in der Lage, zeitveränderliche Zusammenhänge zu repräsentieren. Eine Repräsentation von Unschärfe ist mit diesem Formalismus jedoch nicht möglich. [Schäfer 96] generalisiert deshalb zunächst die zweiwertige Prädikatenlogik erster Stufe durch die Zulassung von reellwertigen Wahrheitswerten im Intervall $[0, 1] \subset \mathbb{R}$ zur *Unscharfen (fuzzy) Prädikatenlogik erster Stufe* (FL1).
Er erweitert die PL1 um die einstelligen Operatoren $\downarrow_\kappa \mathcal{F}$ mit $0_{\mathbb{R}} < \kappa \leq 1_{\mathbb{R}}$, der die *unscharfe* Abschwächung des Wahrheitswertes von $\mathcal{F}$ bezeichnet, und $\uparrow_\lambda \mathcal{F}$ mit $0_{\mathbb{R}} < \lambda \leq 1_{\mathbb{R}}$, der analog die *unscharfe* Verstärkung des Wahrheitswertes von $\mathcal{F}$ bezeichnet. Dabei ist $\downarrow_\kappa \mathcal{F}$ bereits *absolut wahr* (also $\mathcal{I}(\downarrow_\kappa \mathcal{F}) = 1_{\mathbb{R}}$), wenn der Wahrheitswert von $\mathcal{F}$ selbst den Wahrheitswert κ erreicht oder überschreitet (also $\mathcal{I}(\mathcal{F}) \geq \kappa$). Im Gegensatz dazu erreicht $\uparrow_\lambda \mathcal{F}$ gerade noch den Wahrheitswert λ (also $\mathcal{I}(\uparrow_\lambda \mathcal{F}) = \lambda$), wenn $\mathcal{F}$ als absolut wahr anzusehen ist ($\mathcal{I}(\mathcal{F}) = 1_{\mathbb{R}}$). Darüberhinaus verallgemeinert [Schäfer 96] die boole'schen Logikoperatoren der PL1 auf den unscharfen Fall. Die unscharfe Negation wird definiert als Funktion $\nu : [0, 1] \to [0, 1]$ mit den Forderungen $\nu(0) = 1$, $\nu(1) = 0$ (Grenzbedingungen) und $x < y \Rightarrow \nu(x) \geq \nu(y)$ (monotones Fallen). Die unscharfe Konjunktion ist definiert als Funktion $\kappa : [0, 1] \times [0, 1] \to [0, 1]$ mit $\kappa(1, x) = x$ (neutrales Element), $\kappa(x, y) = \kappa(y, x)$ (Kommutativität), $x \leq y \Rightarrow \kappa(x, z) \leq \kappa(y, z)$ (Monotonie) und $\kappa(\kappa(x, y), z) = \kappa(x, \kappa(y, z))$ (Assoziativität). Die unscharfe Disjunktion ist entsprechend definiert als eine Funktion $\delta : [0, 1] \times [0, 1] \to [0, 1]$ mit den Eigenschaften $\delta(0, x) = x$ (neutrales Element), $\delta(x, y) = \delta(y, x)$ (Kommutativität),

	weak	*medium*	*strong*
$\neg(x)$	$1-x^2$	$1-x$	$1-\sqrt{x}$
$\wedge(x,y)$	$min(x,y)$	$x*y$	$max(0,x+y-1)$
$\vee(x,y)$	$min(1,x+y)$	$x+y-x*y$	$max(x,y)$
$\leftarrow(x,y)$	$\vee_w(\neg_m(y),x)$	$\vee_m(\neg_m(y),x)$	$\vee_s(\neg_m(y),x)$

Tabelle 2.1: Drei alternative Semantiken für die unscharfe Negation $\neg$, Konjunktion $\wedge$ und Disjunktion $\vee$ jeweils mit den Argumenten x bzw. x und y. Die Semantik der Subjunktion ergibt sich aus der Definiton: $\leftarrow_\nu (x,y) \equiv \vee_\nu(\neg_m(y),x)$. Hierbei wird also stets die Standard–Negation $\neg_m(x) = 1-x$ verwendet.

$x \leq y \Rightarrow \delta(x,z) \leq \delta(y,z)$ (Monotonie) und $\delta(\delta(x,y),z) = \delta(x,\delta(y,z))$ (Assoziativität). [Schäfer 96] schränkt die durch die oben gegebenen Definitionen entstehenden Freiheitsgrade der boole'schen Operatoren dahingehend ein, daß er für jeden Operator nur drei feste Semantiken (*weak*, *medium* und *strong*) durch entsprechend gewählte Funktionen zuläßt (siehe. Tabelle 2.1). Diese Semantiken der Operatoren entsprechen den in der klassischen unscharfen Logik gebräuchlichen (vgl. etwa [Menzel 2000]). [Schäfer 96] läßt auch die gleichzeitige und gemischte Verwendung dieser Semantiken zu.
Wiederum schränkt der Autor wie bei der MTL auch die FL1 auf ihr hornlogisches Fragment (FHL) ein. Er erweitert den Tableaukalkül für die PL1 zu einem Tableaukalkül für die FHL und weist dessen Vollständigkeit und Korrektheit nach.

2.1.3 Vereinigung von MTL und FL1

Mit Hilfe der MTL ist man in der Lage, zeitveränderliche Zusammenhänge zu repräsentieren. Die FL1 kann dazu verwendet werden, Unschärfe aufgrund von Unsicherheit oder Vagheit formal auszudrücken. In einem weiteren Schritt vereint [Schäfer 96] diese beiden Logiken zur *unscharfen metrisch temporalen Logik* (UMTL). Neben den oben beschriebenen Eigenschaften der beiden einzelnen Logikformalismen bietet die UMTL auch die Möglichkeit, die Vagheit und damit die Unschärfe *zeitlicher* Begriffe auszudrücken. Dazu wird die Semantik der temporalen Operatoren $\Box_S$ und $\Diamond_S$ unscharf modelliert. Eine Aussage wie etwa $\Box_{[0,1000]}$`p` (das Prädikat `p` ist wahr im Zeitraum $[0,1000]$) wird demnach auch dann als *annähernd* absolut wahr betrachtet, wenn `p` lediglich für die Zeiträume $[0,399]$ und $[401,1000]$ wahr ist. Diese Modellierung der Semantik führt zu einer geordneten Menge zeitlich unscharfer Operatoren wie etwa {`absolut immer`, `fast immer`, `zumeist`, `oft`, `manchmal`, `selten`, `irgendwann`}, die dann jeweils mit einer bestimmten Häufigkeit der Wahrheit innerhalb eines bestimmten Zeitraumes in Verbindung gebracht werden.
Auch die UMTL schränkt [Schäfer 96] auf ihr hornlogisches Fragment UMTHL ein. Er erweitert entsprechend den Tableaukalkül der PL1 zu einem Tableaukalkül der UMTHL

	F-Limette–Ausdruck	UMTL–Formel
Faktum	$\lambda \mid t_a : t_e \; ! \; p$	$\downarrow_\lambda \Box_{[t_a,t_e]} p$
Regel	$\lambda \mid t_a : t_e \; ! \; (p \; :- \; q)$	$\downarrow_\lambda \Box_{[t_a,t_e]} (p \leftarrow q)$
Anfrage	$\lambda \mid t_a : t_e \; ! \; ? \; p$	$\downarrow_\lambda \Box_{[t_a,t_e]} p$

Tabelle 2.2: Allgemeiner Aufbau von UMTHL–Fakten, –Regeln und Anfragen in *F-Limette*–Syntax sowie als korrespondierende UMTL–Formel. p und q bezeichnen Prädikate, t_a bzw. t_e bezeichnen Zeitpunkte. λ bezeichnet einen Zusicherungsgrad. Der Unterschied zwischen Faktum und Anfrage besteht in der unterschiedlichen Behandlung durch *F-Limette*. Fakten werden dem bestehenden Logikprogramm angefügt, während Anfragen die Suche nach der Ableitbarkeit eines Faktums bewirken.

und weist wiederum dessen Vollständigkeit und Korrektheit nach.

2.1.4 Anwendung der unscharfen metrisch temporalen Logik in der Bildfolgenauswertung

[Schäfer 96] operationalisiert den Tableaukalkül für die UMTHL und implementiert diese operationalisierte Fassung in Form der Logik–Programmiersprache *F-Limette* (*fuzzy logic programming integrating metric temporal extensions*). Als Zeitstruktur wird dabei $(\mathbb{Z}, 0_{\mathbb{Z}}, <_{\mathbb{Z}})$, also die Menge der ganzen Zahlen mit 0 als Referenzzeitpunkt und der Standard–Ordnungsrelation, verwendet. Ein Logikprogramm besteht dann aus *UMTHL–Regeln* und *UMTHL–Fakten*. UMTHL–Regeln bestehen aus einem Prädikat `p` (dem Kopf der Regel) und einer Bedingung (dem Rumpf der Regel), die die Ableitbarkeit des Prädikats `p` angibt. Ein Prädikat `p` kann dabei Kopf mehrerer Regeln und somit auf mehrere Arten ableitbar sein. Eine *UMTHL–Anfrage* stellt eine Frage nach der Ableitbarkeit eines bestimmten Prädikats `p` dar. Das Logiksystem versucht daraufhin, rekursiv eine Ableitung eben dieses Prädikats aus den gegebenen UMTHL–Fakten anhand der vorliegenden UMTHL–Regeln zu finden. Eine Übersicht über den allgemeinen Aufbau von UMTHL–Fakten, –Regeln und –Anfragen gibt Tabelle 2.2.
Mit Hilfe der UMTHL kann nun Wissen aus unterschiedlichen Quellen repräsentiert werden. Modellwissen kann in Form von UMTHL–Regeln und –Fakten formalisiert werden. Eine Anbindung des Logiksystems an die Ergebnisse der geometrischen Bildfolgenauswertung geschieht durch begriffliche Abstraktion und Formulierung dieser Ergebnisse in Form von UMTHL–Fakten. Die folgenden Abschnitte zeigen, wie auch Wissen über das Verhalten von zu beobachtenden Akteuren in Form von UMTHL–Programmen formuliert werden kann.

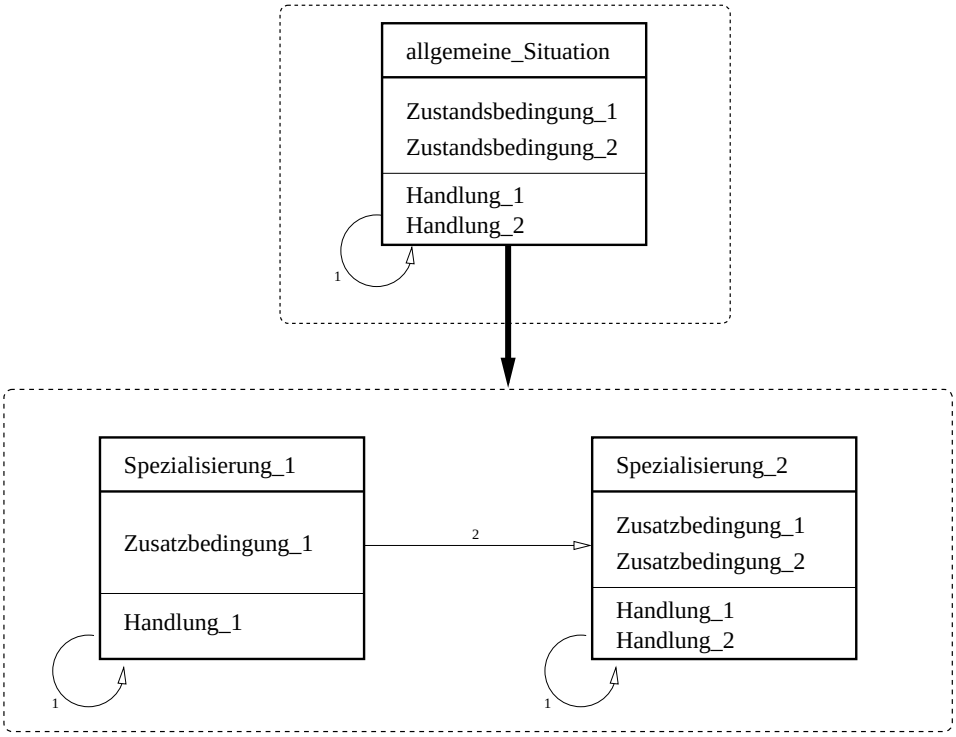


Abbildung 2.1: Allgemeiner Aufbau eines Situationsgraphenbaums. Die durchgezogenen Kästen enthalten neben dem Situationsnamen auch die Situationsbeschreibung in Form von Zustands– und Handlungsschema. Nachfolgebeziehungen sind mit dünnen Pfeilen angedeutet. Die Nummern neben diesen Pfeilen geben die Prioritäten der einzelnen Beziehungen an. Eine *Spezialisierung* der oberen Situation durch den im unteren gestrichelten Kasten gegebenen Situationsgraphen wird durch den dicken Pfeil angezeigt.

2.2 Situationsmodellierung

Bei der Situationsmodellierung geht es darum, *Wissen* über das zu *erwartende* Verhalten von zu beobachtenden Akteuren in einer für die Verwendung in der Bildfolgenauswertung geeigneten Form zu repräsentieren. [Nagel 88] schlägt zur Modellierung von Verhalten den Begriff der “generisch beschriebenen Situation” (*generically described situation*) vor. Als solches *Situationsschema* wird dabei stets die Einheit aus einem *Zustandsschema* und einem *Handlungsschema* bezeichnet. Ein *Zustandsschema* beinhaltet sowohl die Zustandsbeschreibung des Akteurs selbst als auch die Beschreibung des Zustands seiner Umwelt. Diese können in Form von einzelnen Zustandsbedingungen formuliert werden. Das *Handlungsschema* beinhaltet die mit dem Zustand des Akteurs verbundene Handlung oder Handlungserwartung. Diese werden in Form von Handlungsprozeduren formuliert. Die Interpretation dieses Handlungsschemas hängt davon

ab, ob die stattfindende Situationsanalyse in ein *reaktives* System eingebettet ist, etwa als Teil eines Regelungssystems. In einem solchen Fall ist gerade das System der Akteur. Seine Handlungen werden modelliert und die Handlungsprozeduren können direkt die auszuführenden Regelbefehle beinhalten. Findet die Situationsanalyse dagegen in einem lediglich *beobachtenden* System statt, so lassen sich trotzdem die beschriebenen Handlungen mit den tatsächlich beobachteten Handlungen eines Akteurs vergleichen. Dadurch ist eine Erkennung der Absichten des Akteurs und damit eine Prädiktion späterer Situationen möglich.

2.2.1 Von Situationsschemata zu Situationsgraphenbäumen

Als *Situationsgraph* [Krüger 91] wird ein Graph bezeichnet, dessen Knoten durch Situationsschemata gegeben sind. Zeitliche Nachfolgebeziehungen zwischen Situationen werden durch die gerichteten Kanten eines solchen Situationsgraphen modelliert. Diese Kanten sind nummeriert, wodurch die hierüber erreichbaren Nachfolgesituationen je nach der vom Anwender *geschätzten* apriori–Wahrscheinlichkeit ihres Eintreffens geordnet werden können.
Einzelne Situationsschemata können durch untergeordnete Situationsgraphen spezialisiert werden, wodurch eine zunächst allgemein gehaltene Situationsbeschreibung sowohl zeitlich als auch begrifflich feiner aufgelöst wird. Dies geschieht, indem das zu spezialisierende Situationsschema eines Graphen durch eine *Spezialisierungskante* mit einem anderen Situationsgraphen verbunden wird. Dieser stellt dann die genauere Beschreibung der übergeordneten Situation dar. Die dadurch entstehende hierarchische Struktur aus Situationsgraphen, deren Situationsschemata durch andere Situationsgraphen spezialisiert werden, wird *Situationsgraphenbaum* genannt (siehe Abb. 2.1). Das in einem solchen Situationsgraphenbaum enthaltene Wissen in Form von Situationsschemata und deren Beziehungen untereinander stellt das schematische Wissen über das Verhalten eines Akteurs dar und wird im folgenden als *Verhaltensschema* bezeichnet.

2.2.2 Situationsgraphenbäume als UMTHL–Programm

Zur Denotation von Situationsgraphenbäumen entwickelt [Schäfer 96] die Sprache `SIT++`. Diese erlaubt es, Situationen, deren Abfolge in Situationsgraphen sowie deren Spezialisierung in Situationsgraphenbäumen zu formulieren. Darüberhinaus erlaubt `SIT++` die Verwendung vordefinierter sowie die Definition neuer Variablentypen. Begriffe sowie Begriffshierarchien, die zur Beschreibung von Situationsschemata nötig sind, können als sog. *Terminologie* ebenso in `SIT++`–Notation formuliert werden. Die Semantik eines solchen `SIT++`–Programms wird definiert durch die Übersetzung in ein äquivalentes UMTHL–Programm. Hierdurch geschieht auch die Anbindung des Ver-

haltensschemas an das Gesamtsystem zur Bildfolgenauswertung.
Die *Ablauferkennung* erfolgt durch die sogenannte *Graphtraversierung*, die ebenfalls als UMTHL–Programm formuliert ist. Ausgehend von der Wurzel eines Situationsgraphenbaums wird anhand der primitiven Ergebnisse der geometrischen Bildauswertung ein Situationsschema gesucht, das mit diesen Ergebnissen vereinbar ist. Auf der Grundlage der UMTHL (und der damit gegebenen Repräsentation von Zeit– und Unschärfeaspekten) ist es dabei auch möglich, bestimmte Mindest– und Höchstdauern einer solchen *Ausprägung* eines Situationsschemas vorzugeben. Desweiteren kann auch ein Mindestzusicherungsgrad für die Ausprägung gefordert werden. Die zu verfolgende Strategie bei der Suche nach einem ausprägbaren Situationsschema kann durch den Benutzer vorgegeben werden. Sowohl Tiefen– als auch Breitensuche im Baum sind möglich. Zu Beginn wird dabei das speziellste ausprägbare Situationsschema gesucht. Im nächsten Zeitschritt erfolgt dann die Suche nach einem ausprägbaren Schema auf gleicher Hierarchiestufe. Wird ein solches gefunden, ist zu prüfen, ob sich ggf. ein dieses Schema noch weiter spezialisierendes Situationsschema ausprägen läßt. Schlägt eine Traversierung auf gleicher Hierarchieebene fehl, wird zu einer höheren, allgemeineren Ebene zurückgekehrt. Der so erzeugte Pfad von über die Zeit ausgeprägten Situationsschemata entspricht dem erkannten Ablauf.
[Schäfer 96] verwendete zur Formulierung eines Situationsgraphenbaumes in UMTHL spezielle Prädikat–Präfixe, die auch in der vorliegenden Arbeit Verwendung finden sollen, und die jeweils einen bestimmten Aspekt eines Situationsschemas und dessen Beziehungen zu anderen Situationsschemata bezeichnen. Dies sind im einzelnen:

`run_<`*situationsname*`>` Prädikate mit diesem Präfix bezeichnen den Start einer Traversierung in dem jeweiligen Situationsknoten des Graphen. Entsprechend sind sie wahr, wenn eine Traversierung in diesem Knoten gestartet werden kann.

`state_<`*sitname*`>` Prädikate dieser Form bezeichnen das *lokale* Zustandsschema eines Situationsschemas.

`fstate_<`*sitname*`>` Diese Prädikate bezeichnen das *globale* Zustandsschema eines Situationsschemas. Im Gegensatz zum lokalen Zustandsschema müssen hier auch alle Zustandsbedingungen von jenen allgemeineren Situationsschemata erfüllt sein, welche durch das aktuelle Schema spezialisiert werden.

`actn_<`*sitname*`>` Diese Prädikate bezeichnen das lokale *nicht–inkrementelle* Handlungsschema eines Situationsschemas. Handlungen, welche die Ableitung eines solchen Prädikats ermöglichen, werden nur dann ausgeführt, wenn genau das betroffene Situationschema ausgeprägt wurde.

`acti_<`*sitname*`>` Hierdurch wird das lokale *inkrementelle* Handlungsschema eines Situationsschemas beschrieben. Inkrementelle Handlungen werden stets ausgeführt, solange das betreffende Situationsschema ausgeprägt werden kann.

`facti_<`*sitname*`>` Auch diese Prädikate bezeichnen inkrementelle Handlungsschemata. Sie beinhalten jedoch auch Handlungen, welche von allgemeineren Situationsschemata (auf einem Pfad zum aktuellen Schema) *geerbt* werden. Prädikate dieser Form bezeichnen somit das *globale* inkrementelle Handlungsschema eines Situationsschemas.

`factn_<`*sitname*`>` Dies ist die globale Entsprechung zum lokalen `actn`.

`eval_<`*sitname*`>` Diese Prädikate bezeichnen die *Evaluierung* des Situationsschemas. Sie leiten sich stets aus der weiteren Spezialisierung des Situationsschemas, oder – falls eine solche Spezialisierung fehlschlägt – aus einer Prädiktion eines Nachfolgeschemas auf gleicher Ebene ab.

`pred_<`*sitname*`>` Mit diesen Prädikaten wird stets eine mögliche Prädiktion eines Nachfolgeschemas des aktuellen Situationsschemas ausgedrückt. Schlägt eine Prädiktion auf gleicher Ebene fehl, kann ein Prädikat dieser Form die Prädiktion des aktuellen Situationsschemas jedoch auch an ein allgemeinere Situationschema abgeben.

`note(`p`)` An sich kein spezielles Prädikat zur Situationsanalyse, findet dieses Prädikat jedoch in vielen UMTL–Programm Verwendung: es dient der Ausgabe des übergebenen Prädikats p und kann somit zur Verfolgung sowie zur Analyse der Graphtraversierung genutzt werden.

2.3 Diskurs–Repräsentationstheorie

[Kamp & Reyle 93] beschäftigen sich mit der Frage, wie die in einem natürlichsprachlichen Text gegebenen Zusammenhänge erkannt und in eine semantisch äquivalente Darstellung überführt werden können. Das von ihnen entwickelte Verfahren analysiert zunächst die syntaktische Struktur eines Textes und repräsentiert diese in einem Syntaxbaum. Auf diesem werden dann mittels sogenannter Konstruktionsregeln (*construction rules*, *CRs*) bestimmte Operationen ausgeführt, die nach und nach zu einer Verarbeitung des Baumes und gleichzeitig zum Aufbau einer *Diskursrepräsentationsstruktur* (*DRS*) führen. Dieses Verfahren soll im folgenden genauer beschrieben werden.

2.3.1 Syntaxanalyse

[Kamp & Reyle 93] gehen davon aus, daß das Fragment der natürlichen Sprache, das sie mit ihrer Diskursrepräsentationstheorie erfassen wollen, die Sprache einer bestimmten *Phrasenstrukturgrammatik* ist. Eine solche Grammatik G ist zunächst gegeben durch ein Tupel (V, Σ, P, S) mit:

V , einer endlichen Menge von Nichtterminalen,

Σ , einer endlichen Menge von Terminalen,

P , einer endlichen Menge von Produktionen mit $P \subset ((V) \times (V \cup \Sigma)^*)$ und

S , einem Startsymbol mit $S \in V$.

G ist somit durch die Form der Produktionen eine kontextfreien Grammatik. Aufbauend auf diesem Formalismus werden zusätzlich Attribute definiert, die jedes Terminal und Nichtterminal besitzen kann. Belegungen dieser Attribute innerhalb einer Regel können dann zusätzlich zur Struktur gefordert werden. Eine Regel der Form

$$\texttt{S} \rightarrow \texttt{NP} \quad \texttt{VP}$$

die angibt, daß das Nichtterminal `S` (etwa für Satz) zu einer Folge der beiden Nichtterminale `NP` (für Nominalphrase) und `VP` (für Verbphrase) ableitbar ist, kann durch Hinzunahme von Attributen und Forderungen an deren Belegung zu

$$\begin{matrix} \texttt{S} \\ \left[\ num = a\ \right] \end{matrix} \rightarrow \begin{matrix} \texttt{NP} \\ \left[\ num = a\ \right] \end{matrix} \quad \begin{matrix} \texttt{VP} \\ \left[\ num = a\ \right] \end{matrix}$$

erweitert werden. "*num*" bezeichnet hierbei einen Attributnamen, "*a*" ist die in der Regel geforderte Belegung dieses Attributs. Mit solchen *attributierten* Regeln ist man in der Lage, sowohl die Struktur eines Satzes zu beschreiben als auch Nebenbedingungen, die darüberhinaus gelten müssen, zu repräsentieren. Warum dies nötig ist, wird klar, wenn man die folgenden englischsprachigen Sätze betrachtet:

1. *the cars drive*
2. *a truck brakes*
3. *the cars crosses the intersection*
4. *John drive*

Satz *1* beinhaltet das im Plural stehende Subjekt "*the cars*" sowie das ebenfalls im Plural stehende Prädikat "*drive*" und ist in diesem Sinne ein korrekter englischer Satz. So auch Satz *2*, bei dem Subjekt und Prädikat im Singular stehen. Bei Satz *3* und Satz *4* jedoch stimmen Subjekt und Prädikat im Numerus nicht überein. Diese Sätze sind daher nicht korrekt und sollten auch nicht Teil der von der Grammatik erzeugten Sprache sein. Dies kann durch die oben beschriebene erweiterte Produktion sichergestellt werden. Diese fordert ja gerade, daß die Nichtterminale `VP` und `NP` im Attribut *num* übereinstimmend den Wert *a* besitzen. Die Forderung $num = a$ im Nichtterminal `S` führt zu einer Weitergabe des in `NP` und `VP` auftretenden Attributwertes an das Nichtterminal `S`. Nur Sätze, deren Struktur sich durch Anwendung solcher Produktionen ableiten lassen *und* deren Wörter bzw. Teilphrasen bestimmte Nebenbedingungen

erfüllen, sind somit Teil der von der Grammatik erzeugten Sprache.
Ist ein Satz Teil der Sprache dieser Grammatik, so kann eine Folge von Produktionsanwendungen angegeben werden, die gerade diesen Satz erzeugt. Die Struktur dieser Ableitung wird *Syntaxbaum* genannt. Dieser beinhalten durch seinen Aufbau Informationen über die Struktur des Satzes, sowie in seinen Knoten Informationen über die Attribute und deren Belegungen. Dieser Syntaxbaum ist der Ausgangspunkt des Verfahrens von [Kamp & Reyle 93].

2.3.2 Diskurs–Repräsentationsstruktur

Zur Repräsentation der Semantik eines Textes führen [Kamp & Reyle 93] die sogenannte *Diskurs–Repräsentationsstruktur* (*DRS*) ein. Eine DRS K bezüglich eines *Vokabulars* V und einer *Bezugsträgermenge* R ist definiert als ein Tupel $\langle \mathcal{U}_K, Con_K \rangle$ eines Universums $\mathcal{U}_K \subseteq R$ und einer Menge von Bedingungen Con_K. Das Vokabular V ist hierbei stets die Menge von Zeichenfolgenkonstanten, die in einer DRS Verwendung finden können. Sie besteht neben einigen Spezial–Zeichenfolgenkonstanten vor allem aus den Wörtern der natürlichen Sprache, in der ein in eine DRS zu überführender Text formuliert ist. Die Bezugsträgermenge R stellt eine Menge von zulässigen Bezugsträgerbezeichnern dar und sei wie folgt definiert:

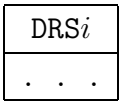
Definition 2.1 *Die Menge R der zulässigen Bezugsträgerbezeichner einer DRS ist die kleinste Menge mit der Eigenschaft, daß für alle $i \in \mathbb{N}$ gilt:*

(i) `<DR`i`>` $\in R$,

(ii) `<ER`i`>` $\in R$ *und*

(iii) `<PDR`i`>` $\in R$.

Zulässige Bezugsträger sind also solche, die sich aus der Konkatenation einer der drei Zeichenfolgen `DR`, `ER` oder `PDR` und einer natürlichen Zahl ergeben.
DRSen nach der obigen Definition seien im folgenden stets durch



mit $i \in \mathbb{N}$ dargestellt. `DRS`i ist hierbei der eindeutige Name der DRS, stellt jedoch keinen *Bezugsträger* der DRS im Sinne der Definition 2.1 dar. Das Universum $\mathcal{U}_K$ einer DRS K ist als Teilmenge von R die Menge von Diskursbezugsträgern aus R, welche die Variablen einer DRS darstellen. Sie werden je nach ihrer Verwendung in folgende Klassen eingeteilt:

Individuen–Bezugsträger Als Individuen–Bezugsträger werden die Bezugsträger bezeichnet, die als Platzhalter für ein Individuum des zu repräsentierenden Diskursbereichs in die DRS eingeführt werden. Individuen–Bezugsträger seien im folgenden mit DRi bezeichnet.

Plural–Bezugsträger Plural–Bezugsträger, die im folgenden stets mit PDRi bezeichnet werden, sind Platzhalter für Mengen von Individuen des Diskursbereichs. Die Zusammensetzung einer solchen Menge von Individuen wird stets durch die Bedingungen einer DRS definiert.

Ereignis–Bezugsträger Mit ERi schließlich werden Ereignisbezugsträger bezeichnet. Sie stellen Platzhalter für den Zeitpunkt oder das Zeitintervall, an oder in dem ein bestimmtes Ereignis des Diskurses stattfand, dar. Wie bei Plural–Bezugsträgern wird auch bei Ereignis–Bezugsträgern die Definition dieser Bezugsträger durch Bedingungen der DRS vorgenommen.

Die Bedingungsmenge Con_K einer DRS K beinhaltet zunächst solche Bedingungen, die eine genauere Definition von Bezugsträgern vornehmen. Dieses sind im Einzelnen:

Explizite Mengen–Definitionen Eine Bedingung der Form

$$\text{PDR}i \text{ = } \bigoplus \text{ (DR}j\text{, ... DR}k\text{, PDR}l\text{, ..., PDR}m\text{)}$$

wird als *explizite* Mengen–Definition eines Plural–Bezugsträgers PDRi bezeichnet. Die durch diesen Bezugsträger repräsentierte Menge stellt dann die Zusammenfassung der auf der rechten Seite der Bedingung aufgeführten Bezugsträger dar.

Implizite Mengen–Definitionen Eine Bedingung der Form

PDRi= $\sum$ DRj

DRSk
DRj
. . .

wird als *implizite* Definition eines Plural–Bezugsträgers PDRi bezeichnet. Als Elemente der zu definierenden Menge werden dann alle diejenigen Individuen betrachtet, für welche der durch DRSk geforderte Zusammenhang gilt. Als Kurzschreibweise für Bedingungen dieser Art soll im folgenden auch PDRi=$\sum$DRj DRSk verwendet werden.

Quantifizierte Mengen–Definitionen Eine Bedingung der Form

$$\text{PDR}i \text{ = } quantifier(\text{PDR}j)$$

wird als *quantifizierte* Mengen–Definition bezeichnet. Die Menge PDRi wird hierdurch als Teilmenge von PDRj definiert, und zwar in dem durch den Quantor $quantifier \in V$ angegebenen Teilmengenverhältnis. Beispiele für Quantoren sind einerseits Zahlwörter (wie `two, three,...`), andererseits aber auch allgemeinere Quantoren (wie `some, most, all,...`).

Explizite Ereignis–Definitionen Eine Bedingung der Form

ERi : | DRSj |
|---|
| . . . |

wird als explizite Definition eines Ereignis–Bezugsträgers bezeichnet. ERi wird hierdurch als das Zeitintervall definiert, in dem der Teildiskurs DRSj stattfindet.

Zusammenfassende Ereignis–Definitionen Eine Bedingung der Form

ERi = [ERj, ..., ERk]

definiert den Ereignis–Bezugsträger ERi als das Zeitintervall, das sich durch Zusammenfassung der Zeitintervalle auf der rechten Seite der Bedingung ergibt.

Zeitraum–Definitionen Eine Bedingung der Form

ERi = DURATION(PDRj)

wird zur Darstellung von explizit genannten Zeiträumen (wie etwa `two minutes`) benötigt.

Neben diesen der reinen Definition von Bezugsträgern dienenden Bedingungen enthält die Bedingungsmenge einer DRS sowohl Bedingungen, die Eigenschaften von Bezugsträgern ausdrücken, als auch solche, die Beziehungen unter diesen Bezugsträgern repräsentieren. Hierbei lassen sich zunächst vier Arten von Bedingungen unterscheiden:

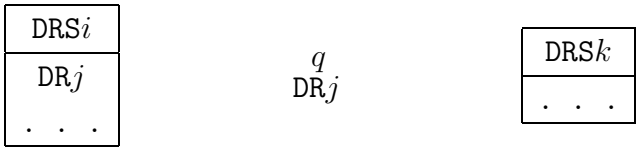
Attribute Attribute sind gegeben durch *attribute*(DRi) bzw. *attribute*(PDRj) mit $attribute \in V$. Sie beschreiben Eigenschaften eines Individuen– bzw. Plural–Bezugsträgers.

Prädikate Prädikate beschreiben Beziehungen, die zwischen Bezugsträgern gelten sollen. Sie werden gegeben durch *predicate*(DRi,...,DRj,PDRk,...,PDRl) mit $predicate \in V$.

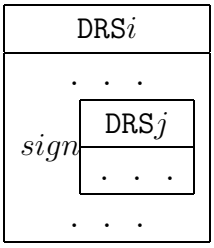
Namen Eine Bedingung der Form *name*(DRi) mit $name \in V$ weist einem Individuen–Bezugsträger einen eindeutigen Eigennamen zu.

Relationen Relationen der Form *relation*(ERi,ERj) mit $relation \in V$ stellen eine Sonderform von Bedingungen dar, da sie stets eine *zeitliche* Beziehung zwischen zwei Ereignis–Bezugsträgern herstellen.

Neben diesen einfachen Bedingungsarten stellt die sogenannte *Duplex–Bedingung* einen komplexeren Zusammenhang zwischen Bezugsträgern her. Sie besteht aus zwei DRSen, die durch einen Quantor verbunden sind. Dies sei durch



dargestellt. Die linke DRS stellt dabei einen Teildiskurs dar, in dem ein bestimmter Individuen–Bezugsträger (hier DRj) vorkommt. Alle diejenigen Belegungen dieses Bezugsträgers, für welche der Teildiskurs erfüllt wird, bilden eine Menge. Für einen durch den Quantor (hier allgemein $q \in V$) bestimmten Teil dieser Menge wird dann der durch die rechte DRS gegebene Teildiskurs gefolgert.
Neben den oben beschriebenen Bedingungsarten kann auch eine DRS DRSj als Unter–DRS eine Bedingung innerhalb einer DRS DRSi darstellen. Eine solche Unter–DRS kann negiert und nicht–negiert auftreten, und stellt stets eine Zusammenfassung eines Teil–Zusammenhangs der gesamten DRS dar. Dies wird im folgenden durch



dargestellt, wobei $sign \in V$ angibt, ob die Unter–DRS negiert oder nicht–negiert zu lesen ist.

2.3.3 Konstruktionsregeln und deren Anwendung

Die Umwandlung eines Syntaxbaumes in eine DRS geschieht durch die Anwendung sogenannter Konstruktionsregeln (*Construction Rules*, *CRs*). Dazu wird zunächst eine leere DRS erzeugt, in die der zu verarbeitende Syntaxbaum eingefügt wird. Eine Konstruktionsregel besteht stets aus einer *auslösenden Struktur* (*Triggering Structure*, *TS*) in Form einer Baumstruktur und einem Aktionsteil. Wird in dem betrachteten Syntaxbaum die auslösende Struktur einer Konstruktionsregel als Teilbaum gefunden, wird

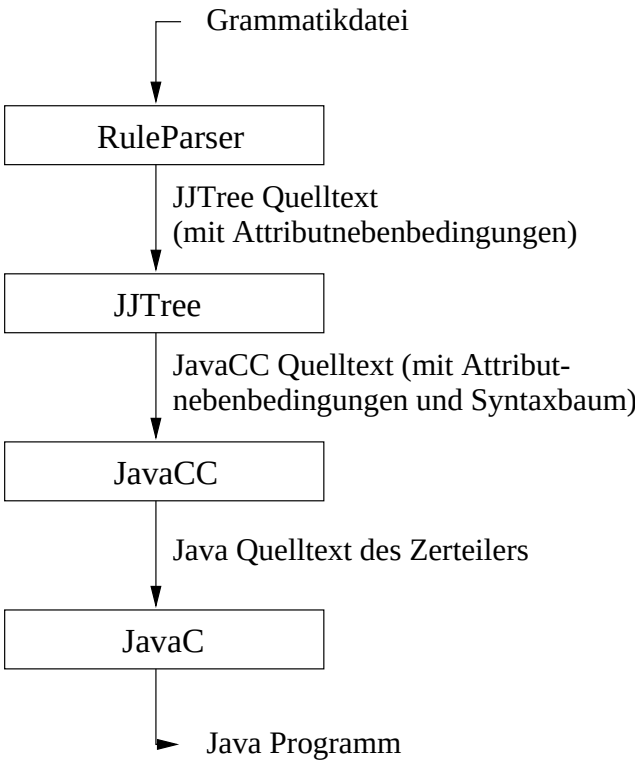


Abbildung 2.2: Übersetzungsabfolge von einer Grammatikdatei bis zu dem zugehörigen Java Zerteilerprogramm. Das von [Arens & Ottlik 2000] erstelle Programm "RuleParser" liest die übergebene Grammatikdatei ein und erzeugt den Quelltext eines "JJTree"–Programms sowie Klassen zur Repräsentation von definierten Konstruktionsregeln. "JJTree" ist ein Vorübersetzer für "JavaCC", einem Übersetzer–Erzeuger für Java. Dieser erzeugt schließlich Java–Quelltext, der dann mit einem Java–Übersetzer in ein Java–Programm übersetzt werden kann.

daraufhin der Aktionsteil dieser Regel ausgeführt. Als Aktionen sind dabei sowohl die Erzeugung neuer DRS–Bedingungen und Bezugsträger als auch Operationen auf dem Syntaxbaum (wie etwa das Löschen bestimmter Teilbäume) möglich. Auch auf schon bestehende DRS–Bedingungen und Bezugsträger kann bei der Aktionsausführung Bezug genommen werden, wodurch z.B. der Rückbezug eines Pronomens auf zuvor genannte Bezugsträger möglich ist. Wird durch eine Aktion eine Bedingung erzeugt, die ihrerseits wieder DRSen enthält, werden im weiteren Verlauf zunächst diese neuen Strukturen auf die Anwendbarkeit von Konstruktionsregeln hin untersucht. Eine DRS, die keine weiteren Unter–DRSen und keinen Syntaxbaum mehr enthält, gilt als *reduziert*. Eine DRS mit Unter–DRSen gilt als reduziert, wenn alle ihre Unter–DRSen reduziert sind und sie selbst keinen Syntaxbaum mehr enthält. Die Umwandlung eines Syntaxbaumes in eine DRS gilt dann als abgeschlossen.

2.3.4 Ein Werkzeug zur DRS–Erzeugung

[Arens & Ottlik 2000] entwickelten auf der Basis des von [Kamp & Reyle 93] vorgestellten Verfahrens zur Diskursrepräsentation ein Werkzeug, das DRSen zu einer gegebenen Grammatik und zugehörigen Konstruktionsregeln erzeugt. Die Grammatik wird dem System dabei in Form einer Textdatei übergeben. Diese Textdatei beinhaltet sowohl die Definition der Nichtterminale der Grammatik, der verwendeten Attribute und deren möglichen Belegungen als auch die Produktionen der Grammatik. In einem weiteren Abschnitt der Textdatei werden die zu verwendenden Konstruktionsregeln mit ihren auslösenden Strukturen und Aktionsteilen definiert. Der Aktionsteil kann dabei Methoden enthalten, die von [Arens & Ottlik 2000] in einer Bibliothek bereitgestellt wurden. Aus dieser Textdatei werden dann automatisch alle zur Syntaxbaum– und anschliessenden DRS–Erzeugung nötigen Klassen und Methoden erzeugt (vgl. Abb. 2.2). Das erstellte Werkzeug zur Verarbeitung der Grammatikdatei sowie die automatisch erstellten Klassen sind in der Programmiersprache Java implementiert.

Kapitel 3

DRS–Erzeugung

3.1 Zielsetzung

Im folgenden Kapitel soll beschrieben werden, wie mit Hilfe des von [Arens & Ottlik 2000] erstellten Werkzeuges DRSen aus Verhaltensbeschreibungen erzeugt werden können. Ausgehend von einem Beispieltext [Nagel 99] wird hierzu zunächst eine Grammatik entwickelt, welche diesen Text erzeugen kann. Daran anschließend werden Konstruktionsregeln erstellt, die einen der Grammatik entsprechend erzeugten Syntaxbaum abarbeiten und daraus schrittweise eine DRS aufbauen. Da das Hauptziel der vorliegenden Arbeit in der Umwandlung von aus Verhaltensbeschreibungen erstellten DRSen in entsprechende Logik–Darstellungen besteht, wurde bei den oben genannten Arbeitsschritten einerseits das Ziel verfolgt, möglichst schnell eine einfache entsprechende Grammatik und zugehörige Konstruktionsregeln zu entwickeln. Andererseits sollten nötige Änderungen an dem bestehenden Werkzeug von [Arens & Ottlik 2000] auf ein Minimum beschränkt werden.
Die Grammatik– bzw. Konstruktionsregel–Dokumentation sowie der Beispieltext sind in Anhang A bzw. Anhang B zu finden. Die allgemeine Vorgehensweise bei der Erstellung dieser Grammatik und Konstruktionsregeln, Besonderheiten des Beispieltextes und dadurch nötige Änderungen an dem Werkzeug von [Arens & Ottlik 2000] werden im folgenden beschrieben.

3.2 Eine Beispielverhaltensbeschreibung

Ausgangspunkt der Untersuchungen dieser Arbeit ist ein Beispieltext, der von [Nagel 99] bereitgestellt wurde. Dieser englischsprachige Text besteht aus vier Abschnitten, die das Verhalten eines Fahrzeuges beschreiben, das eine Kreuzung überquert. Ausgehend von dem abstrakten Begriff *'crossing an intersection'* (*eine Kreuzung überqueren*) beinhaltet jeder dieser vier Abschnitte eine weitergehende *Verfeinerung* der Begriffe

des vorangehenden Abschnitts. Abschnitt I etwa beschreibt, wie sich der Begriff *'cross'* zeitlich durch die Begriffe *'drive to an intersection'*, *'drive across an intersection'* und *'drive away from an intersection'* (etwa: Hinfahren zu , Überfahren und Verlassen einer Kreuzung) genauer auflösen läßt. Neben dieser *zeitlichen* Verfeinerung enthält der Beispieltext auch *begriffliche* Verfeinerungen. So beschreibt etwa Abschnitt IV, wie der Begriff des Fahrens (*'drive'*) genauer beschrieben werden kann durch das Fahren auf einer bestimmten Fahrbahn (*'left-turning lane'*, *'straight ahead lane'* oder *'right-turning lane'*).
Es sei an dieser Stelle darauf hingewiesen, daß der Beispieltext nicht die tatsächlich ausgeführten Aktionen eines bestimmten Fahrzeuges beim Überqueren einer bestimmten Kreuzung beschreibt. Vielmehr werden durch die vier Abschnitte der Verhaltensbeschreibung *allgemein* Begriffe beschrieben, die bei der Beobachtung (und anschließenden Beschreibung des Beobachteten) irgendeines Fahrzeuges beim Überqueren irgendeiner Kreuzung Verwendung finden können.
Grammatikalisch hat dies zur Folge, daß nicht nur Akteure des betrachteten Diskursbereichs (also Fahrzeuge in Kreuzungssituationen) Subjekte der auftretenden Prädikate und Verbformen sind, sondern vor allem auch Nominalphrasen, die verwendete Begriffe beschreiben, als Subjekte von Handlungen auftreten. Dies geschieht in dem betrachteten Beispieltext sowohl in Aktiv– als auch in Passivsätzen.
Eine weitere Besonderheit des Beispieltextes ist die Verwendung bzw. Definition ganzer Teilphrasen des Satzes als *feststehende Begriffe*. Dies wird durch das Setzen der entsprechenden Teilphrase in Hochkommata erreicht, was im folgenden stets als Zitat bezeichnet werden soll. Gerade diese Zitate sind es, die zu beschreibende Begriffe benennen und so eine Verfeinerung dieser Begriffe durch Rückbezug erst ermöglichen.

3.3 Vorgehensweise bei der Grammatikentwicklung

[Arens & Ottlik 2000] verfolgten in ihrer Arbeit das Ziel, natürlichsprachliche Beschreibungen des Geschehens in einer Kreuzungsszene zu verarbeiten. Im Verlauf dieser Arbeit entstand das schon mehrfach erwähnte Werkzeug zur DRS–Erstellung. Zur Demonstration dieses Werkzeuges und zur Verfolgung ihres Gesamtzieles erstellten die Autoren auch eine Grammatik und zugehörige Konstruktionsregeln, um die von ihnen verwendeten englischsprachigen Texte in DRSen zu überführen. Diese Grammatik eignet sich für die Zwecke der vorliegenden Arbeit jedoch in der bestehenden Form nicht, da dort einige der hier auftretenden Grammatikstrukturen (wie etwa Passivsätze) nicht unterstützt wurden. Da mit Hilfe des Werkzeuges zur DRS–Erstellung jedoch die Neuentwicklung einer Grammatik relativ einfach ist, wurde die bestehende Grammatik nicht abgeändert, sondern eine komplett neue Grammatik entwickelt. Es hat sich dabei als hilfreich erwiesen, zunächst die Produktionen für bestimmte Teilphrasen wie etwa Nominalphrasen zu erstellen, diese an den im Beispieltext auftretenden Phrasen zu testen und sie danach in die Gesamtgrammatik zu integrieren. Probleme bei dieser

Vorgehensweise traten lediglich dadurch auf, daß im Beispieltext auch Verbphrasen (als Gerundien oder feststehende Begriffe) die Rolle einer Nominalphrase übernehmen können und so eine schrittweise Integration einzelner Phrasentypen nur bedingt möglich ist. In dieser Weise wurden zunächst Produktionen definiert, die einzelne Sätze erzeugen. Auf dieser Basis wurden dann zuletzt Produktionen hinzugefügt, die die Erzeugung ganzer Textabschnitte ermöglichen.
Auf eine Definition von Morphologie–Regeln für einzelne Wortarten wurde zunächst verzichtet, da so beim Zerteilungsprozeß auftretende Fehler eindeutig auf fehlerhafte Grammatik–Produktionen bzw. Fehler in den Lexikon–Einträgen zurückgeführt werden konnten. Die nachträgliche Definition dieser Morphologie–Regeln hatte jedoch den Nachteil, daß schon bestehende Lexikon–Einträge ggf. geändert bzw. Einträge für abgeleitete Wortformen gelöscht werden mußten.

3.4 Konstruktionsregelerstellung

Die Erstellung von Konstruktionsregeln, die schrittweise einen Syntaxbaum der definierten Grammatik abarbeiten und so eine DRS aufbauen, läßt sich prinzipiell von der Definition der Grammatik trennen. Es sei jedoch darauf hingewiesen, daß eine *konstruktionsregel–freundliche* Gestaltung der Grammatikproduktionen und damit der auftretenden Syntaxbäume die Erstellung von Konstruktionsregeln sehr vereinfachen kann. Ein Beispiel hierfür bieten die in dieser Arbeit verwendeten Produktionen zu Behandlung von mehrfachen Adverbialphrasen (siehe Anhang A.4). Durch den rekursiven Aufbau dieser Produktionen und die damit auftretende Selbstähnlichkeit in den erzeugten Syntaxbäumen wurde hier versucht, eine Abarbeitung dieser Teilphrasen mit relativ wenigen Konstruktionsregeln zu ermöglichen. Dieses Prinzip findet sich auch bei der Behandlung von Aufzählungen wieder.
Grundsätzlich ist bei der Erstellung von Konstruktionsregeln darauf zu achten, daß die spätere Anwendung dieser Regeln dem in Abschnitt 2.3.3 skizzierten Algorithmus folgt. Von Seiten der Textdatei, in der die Konstruktionsregeln definiert werden, kann auf diesen Algorithmus nur durch die Angabe der Reihenfolge der Konstruktionsregeln Einfluß genommen werden (siehe dazu auch Anhang A.5). Dies bedeutet vor allem, daß die gegenseitige Beeinflussung von Konstruktionsregeln – etwa dadurch, daß diese Regel den Syntaxbaum derart umbaut, daß jene Regel erst anwendbar wird – einerseits von Anfang an beachtet werden muß, andererseits aber gerade ein wichtiges Mittel zur Beeinflussung des oben genannten Algorithmus darstellt. Als Beispiel für diese implizite Beeinflussung des Algorithmus seien hier die Konstruktionsregeln zur Behandlung von Nominalphrasen in Form von Zitaten genannt. Diese Regeln ermöglichen erst durch die Struktur, die sie nach ihrer Anwendung im Syntaxbaum hinterlassen, eine weitere Verarbeitung durch eine spezielle Nominalphrasenregel.
Als hilfreich in diesem Zusammenhang hat sich auch die Definition sogenannter *Hilfsattribute* erwiesen. Diese Attribute erhalten ihre Belegungen nicht bereits bei der Zer-

teilung eines Textes und damit bei der Erstellung eines Syntaxbaumes, sondern erst durch die Anwendung einer Konstruktionsregel. Durch solche Hilfsattribute und deren Belegungen lassen sich weitere Bedingungen für die Anwendbarkeit einer Konstruktionsregel definieren. Ein Beispiel hierfür bieten jene Regeln, welche das Einfügen von DRS–Bedingungen aufgrund von Satz–Prädikaten bewirken. Diese werden erst ausgeführt, wenn im Wurzelknoten der entsprechenden Auslösenden Struktur das Attribut `processed` mit dem Wert `ISPROCESSED` belegt ist. Diese Belegung tritt jedoch nur auf, wenn vorher schon eine Konstruktionsregel angewandt wurde, die bereits einen neuen Ereignisreferenten erzeugt hat, in deren definierender DRS sich nun die oben genannte Auslösende Struktur befindet.

3.5 Änderungen an dem Werkzeug zur DRS–Erstellung

Trotz der oben erwähnten Zielsetzung, nötige Änderungen am bestehenden Werkzeug zur DRS–Erzeugung auf ein Minimum zu beschränken, wurden doch einige Änderungen vorgenommen. Dabei wurde jedoch darauf geachtet, daß diese Änderungen die bisherige Allgemeinheit des Werkzeuges nicht einschränken. Sie sind in diesem Sinne als echte Erweiterungen zu sehen und werden im folgenden einzeln beschrieben.

3.5.1 Definition neuer Zeichenfolgen–Konstanten

Zur Verwendung bestimmter feststehender Ausdrücke, die immer gleiche Eigenschaften von oder Beziehungen zwischen Diskursreferenten innerhalb einer DRS ausdrücken, bietet das Werkzeug zur DRS–Erstellung die Möglichkeit, mit Hilfe der Aktionsteilmethode *getString* aus einer Konstruktionsregel heraus diesen feststehenden Ausdruck anzufordern und geeignet zu verwenden. Beispiele für die Verwendung dieser Zeichenfolgenkonstanten sind die vordefinierten zeitlichen Relationen `WHILE`, `BEFORE` und `AFTER`. Zusätzlich zu diesen Konstanten wurden die folgenden Zeichenfolgen definiert:

`SOMEONE` Diese Konstante mit dem Wert `agens` wird stets als Attribut eines Diskursreferenten eingefügt, wenn dieser als *unbekanntes* Subjekt eines Passivsatzes oder einer Verbphrase eingefügt wurde. Das Attribut weist damit lediglich auf die Unkenntnis weiterer Eigenschaften des Diskursreferenten hin.

`CONJUNCTIVE` Diese Konstante wird ebenfalls als Attribut–Name verwendet, diesmal jedoch als Eigenschaftsbezeichnung eines Pluraldiskursreferenten (PDR). Das Attribut sagt damit aus, das jegliche andere Verwendung dieses PDRen stets *konjunktiv*, also für jedes im PDR enthaltene Element, zu lesen ist.

`DISJUNCTIVE` Diese Konstante ist das Gegenstück zu `CONJUNCTIVE`. Das Attribut sagt damit aus, das jegliche andere Verwendung dieses PDRen stets *disjunktiv*, also

für mindestens eines der im PDR enthaltenen Elemente, zu lesen ist.

Die Verwendung aller anderen Zeichenfolgenkonstanten bleibt durch diese Neudefinitionen unberührt.

3.5.2 Die neue Aktionsteilmethode *getSubTreeString*

[Arens & Ottlik 2000] sahen für den Aktionsteil einer Konstruktionsregel die Verwendung vordefinierte Aktionsteilmethoden aus einer umfangreichen Bibliothek vor. Keine dieser Methoden bot jedoch die Möglichkeit, die in dem Beispieltext auftretenden Zitate geeignet zu behandeln. Hierbei soll es einerseits möglich sein, Zitate auch als Teilphrasen der Sätze, in denen sie auftreten, zu behandeln. Dies bedeutet, daß sie im Syntaxbaum nicht einfach als ein Knoten auftreten dürfen, der eben das Zitat enthält. Vielmehr müssen sie sich entsprechend ihrer Struktur verhalten, als seien sie keine Zitate (zitierte Verbphrasen müssen etwa die Rolle des Prädikats eines Satzes übernehmen können, wenn ihre Verwendung dies vorsieht). Andererseits wird durch die Verwendung eines Zitats stets ein feststehender Begriff definiert. Entsprechend soll dieser Begriff innerhalb der zu erzeugenden DRS auch die gesamte Zeichenfolge enthalten, die innerhalb des Zitats vorkam. Da ein Knoten im Syntaxbaum bisher jedoch nur entweder genau ein Wort der Sprache enthielt (Blatt des Baumes) oder kein Wort, mit den Aktionsteilmethoden aber auch nur genau auf diesen Inhalt zugegriffen werden konnte, war das Zitat–Problem mit den bisherigen Methoden nicht zu lösen. Die einfachste Lösung schien deshalb in der Definition einer neuen Aktionsteilmethode *String getSubTreeString(SimpleNode sn)* zu bestehen. Diese liefert bei Übergabe eines beliebigen Knotens des Syntaxbaumes eine Zeichenfolge zurück, deren Inhalt sich aus allen unterhalb dieses Knotens befindlichen Wörten der Sprache ergibt. Die einzelnen Wörter der Zeichenfolge sind dabei in der Reihenfolge ihrer Nennung im ursprünglichen Text – durch Unterstriche getrennt – konkateniert.

3.5.3 Die neue Aktionsteilmethode *getDefAntecedent*

Die oben erwähnten vordefinierten Aktionsteilmethoden wurden bisher auch verwendet, um Rückbezüge, die etwa für die korrekte Behandlung von Personalpronomina nötig sind, zu ermöglichen. Zu diesem Zweck war die Methode *String getAntecedent(SimpleNode sn)* vorgesehen. Diese suchte unter den erreichbaren Diskursreferenten (DRen) einer DRS (erreichbar sind die DRen der DRS selbst und die DRen aller übergeordneter DRSen) diejenigen, deren Attributbelegungen mit denen des übergebenen Knotens eines Syntaxbaumes übereinstimmte. Zurückgegeben wurde der Bezeichner des DRen, der als letzter gefunden wurde (was der letzten Nennung im ursprünglichen Text entspricht). Der in der vorliegenden Arbeit behandelte Beispieltext enthält keine Personalpronomina. Ein Rückbezug von DRen bzw. die Identifikation neu eingeführter Referenten mit bereits vorhandenen wird hier jedoch durch die häufige Verwendung *de-*

finiter Nominalphrasen nötig. Hierbei soll ein durch eine Nominalphrase eingeführter DR mit einem bereits bestehenden identifiziert werden, was durch die Verwendung eines *bestimmten* Artikels eingeleitet wird. Entscheidend für die Wahl des *Vorgängers* ist dabei jedoch nicht dessen Attributbelegung. Vielmehr muß dieser die gleichen Eigenschaften – ausgedrückt durch Namens–, Attribut– und Prädikatsbedingungen der DRS – besitzen. Gesucht war also eine Aktionsteilmethode, die einem Vorgänger aufgrund der ihn betreffenden DRS–Bedingungen auswählt. Eine solches Verhalten wurde mit der neu definierten Methode *String getDefAntecedent(SimpleNode sn)* erreicht. Diese liefert den zuletzt eingeführten DRen, der alle Eigenschaften des übergebenen DRen (der Name des übergebenen DRen ist gegeben durch den übergebenen Knoten) besitzt. Schlägt ein Rückbezug fehl (es wurde kein DR mit gleichen Eigenschaften gefunden), wird der übergebene DR selbst zurückgegeben. In diesem Fall findet also keinerlei Rückbezug statt.

3.5.4 Änderungen am Sprachzerteiler

Der in der vorliegenden Arbeit verwendete Beispieltext weicht neben den schon beschriebenen Aspekten auch durch einige nun verwendete Zeichen von den bisher von [Arens & Ottlik 2000] betrachteten Texten ab. So muß der Sprachzerteiler auch Kommata als Satzzeichen und Hochkommata als ein Zitat umgebende Zeichen erkennen können. Die Verarbeitung von Hochkommata war in dem bisherigen Sprachzerteiler jedoch nicht vorgesehen. Die Erkennung von Kommata funktionierte, führte aber durch die gleichzeitige Verwendung dieser Zeichen als Trennzeichen im Dateiformat des Lexikons zu Speicher– bzw. Ladeproblemen. Eine entsprechende Änderung wurde vorgenommen, so daß nun auch die erwähnten Zeichen erkannt werden können. Die Verarbeitung bisher verarbeitbarer Texte wird hierdurch nicht beeinflußt. Ältere Lexika sind durch die Änderungen im Dateiformat jedoch nicht mehr lesbar.

Kapitel 4

Repräsentation von Situationsgraphenbäumen

Im vorangehenden Kapitel wurde beschrieben, wie mit Hilfe eines schon bestehenden Werkzeuges zur DRS–Erzeugung aus Verhaltensbeschreibungen in Form natürlichsprachlicher Texte DRSen erzeugt werden können. Nötige Änderungen an dem verwendeten Werkzeug wurden beschrieben. Das nächste Ziel der vorliegenden Arbeit ist es, das in diesen DRSen enthaltene Wissen in Situationsgraphenbäume (*situation graph tree*, *SGT*) bzw. in entsprechende Logikprogramme zu überführen. Hierzu muß geklärt werden, wie diese Verhaltensschemata rechnerintern repräsentiert werden sollen. Der folgende Abschnitt wird sich deshalb zunächst mit der rechnerinternen Repräsentation von SGTs auseinandersetzen. Ein weiterer Abschnitt behandelt die Frage, wie der Benutzer bei der Erstellung von SGTs durch geeignete Anzeige–Mittel unterstützt werden kann.

4.1 Java–Klassen zur Repräsentation von Situationsgraphenbäumen

Zur Repräsentation von SGTs sowie von UMTL–Programmen wurden zwei Java–Pakete implementiert. Das Paket `SIT` beinhaltet alle Klassen, die zur Verarbeitung und Repräsentation von SGTs und deren Elementen nötig sind. Im einzelnen sind dies die Klassen `SGTElement`, welche die Oberklasse aller Elemente eines SGT darstellt. Hiervon werden zunächst die Klassen `Edge`, welche die Oberklasse aller in SGTs vorkommenden Kanten darstellt, die Klasse `Scheme`, von der alle verwendeten Schemata einer Situation abgeleitet sind, und die Klassen `Situation`, `SituationGraph` und `SituationGraphTree` abgeleitet. Unterklassen von `Edge` sind die Klassen `PredEdge` zur Repräsentation von Prädiktionskanten und `SpecEdge`, welche Spezialisierungskanten darstellt. `StateScheme` und `ActionScheme` schließlich sind Unterklassen von `Scheme`

und repräsentieren entsprechend Zustands– bzw. Aktionsschemata.
Der Aufbau eines SGT wird von einem Objekt der Klasse `SituationGraphTree` mit Hilfe von *Hashtables* verwaltet, von denen für jede Elementart des SGT eine eigene existiert. Dies ist der einzige Ort, an dem die einzelnen Elemente tatsächlich gespeichert werden. Alle anderen Verweise auf Elemente – etwa Quell– und Zielsituation einer Prädiktionskante – geschehen lediglich mittels eindeutiger Element–Namen. Über diese Element–Namen können dann bei dem SGT, zu dem die Prädiktionskante gehört, die entsprechenden Elemente erfragt werden.
Ein Vorteil dieser Vorgehensweise liegt in der zentralen Verwaltung des SGT. Insbesondere bei einer schrittweisen Erstellung von Verhaltensschemata können so leicht neue Elemente dem SGT hinzugefügt werden. Da Prädiktionskanten und Spezialisierungskanten auch Elemente des SGT sind, werden auch diese einfach als solche hinzugefügt. Einzelne Situationsgraphen können schließlich einfach als Zusammenfassung von bestehenden Elementen definiert werden. Aber auch das Hinzufügen eines zunächst leeren Situationsgraphen und anschließendes Füllen diese Graphen mit neuen Elementen ist möglich.
Das eigentliche Ziel dieser Arbeit besteht im Erstellen von Logik–Programmen, die einem SGT entsprechen. Zu diesem Zweck wurde neben dem oben beschriebenen Paket `SIT` ein Paket `UMTL` implementiert, welches UMTL–Programme repräsentieren kann. Bestandteil dieses Pakets ist neben einigen Klassen, die Repräsentations–Klassen für Teile von UMTL–Programmen wie Prädikate, Regeln und Fakten darstellen, vor allem ein Zerteiler, der UMTL–Programme in Form von Textdateien einlesen und daraus Objekte der entsprechenden Klassen des Pakets erstellt.
Mit Hilfe eines Konvertierungsprogramms auf Basis der beiden oben beschriebenen Pakete ist es nun möglich, bestehende UMTL–Darstellungen von SGTs einzulesen und grafisch darzustellen.

4.2 Graphische Darstellung von Situationsgraphenbäumen

Die textuelle Inspektion von SGTs – sowohl als `SIT++`-Programm wie als entsprechendes Logikprogramm – ist sehr aufwendig und fehlerträchtig. Gleiches gilt auch für eine textuelle Ausgabe der oben beschriebenen Datenstrukturen. Aus diesem Grund ist es ein Ziel der vorliegenden Arbeit, dem Benutzer bzw. Entwickler von Verhaltensschemata ein geeignetes Werkzeug bereit zu stellen, mit dem eine grafische Inspektion dieser Schemata möglich ist. Die Struktur von SGTs ist verglichen mit einfachen Graphen oder Bäumen relativ komplex. Wie in Kapitel 2.2 beschrieben, bestehen SGTs zunächst aus Situationsknoten, die durch gerichtete Prädiktionskanten miteinander verbunden sind. Zusammenfassungen von Teilmengen dieser Situationsknoten und der sie verbindenden Prädiktionskanten werden als Situationsgraphen bezeichnet. Jeder dieser Situationsgraphen stellt eine zeitlich oder begrifflich genauere Beschreibung derjenigen Situation dar,

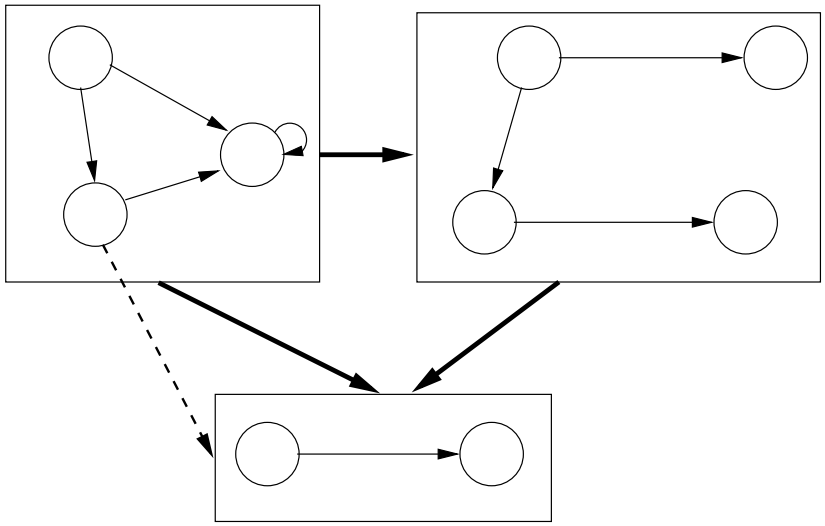


Abbildung 4.1: Abbild eines hierarchischen Graphen mit zwei Hierarchie–Ebenen. Die Kreise beschreiben die Knoten der unteren Ebene. Gerichtete Kanten auf dieser Ebene sind durch dünne Pfeile dargestellt. Die obere Hierarchie–Ebene wird durch Rechtecke (Knoten) bzw. dicke Pfeile (Kanten) gebildet. Die gestrichelte Kante verbindet einen Knoten der unteren Ebene mit einem Knoten der oberen Ebene und ist deshalb hier nicht zulässig.

mit welcher der Situationsgraph durch eine gerichtete Spezialisierungskante verbunden ist. Die Gesamtheit von Situationsknoten und mit diesen Knoten verbundenen Situationgraphen, die wiederum Situationsknoten beinhalten, wird Situationsgraphenbaum genannt. Der auf diese Weise gebildete Graph besitzt als Knoten daher Situationen und Situationsgraphen, als Kanten einerseits Prädiktionskanten zwischen Situationsknoten, andererseits Spezialisierungskanten, welche Situationsknoten mit Situationsgraphen verbinden. Diese letzteren Kanten verbinden daher Knoten unterschiedlichen Typs. Eine strikte Trennung des Graphen in zwei Hierarchie–Ebenen – die untere der Situationsknoten und Prädiktionskanten und die obere der Situationsgraphen – wie in Abbildung 4.1 ist deshalb nicht möglich.

Ein Konstrukt der Graphentheorie, das mächtig genug ist, um SGTs zu repräsentieren, ist mit dem Konstrukt des *gerichteten Hypergraphen* gegeben [Gon. & Min. 84]. Ein gerichteter Hypergraph $\mathcal{H} = (S, U)$ wird definiert durch eine Menge S, deren Elemente Knoten genannt werden, und einer Menge U, deren Elemente $U_j \in U$ Paare von Teilmengen von S sind und *gerichtete Hyperkanten* genannt werden. Jede solche gerichtete Hyperkante $U_j = ({U_j}^-, {U_j}^+)$ verbindet die Knoten in ${U_j}^-$ mit denen in ${U_j}^+$. Um SGTs als Hypergraphen zu modellieren, wird als die Menge S der Knoten des Hypergraphen die Menge aller im Situationsgraphenbaum auftretenden Situationsknoten gewählt. Die Prädiktionskanten zwischen Situationsknoten werden als Hyperkanten

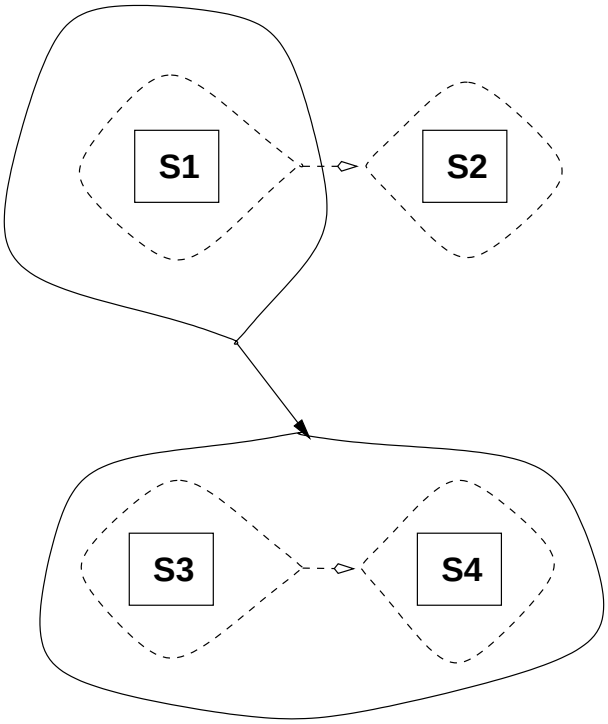


Abbildung 4.2: Darstellung eines SGT als Hypergraph. **S1** bis **S4** bezeichnen Situationsknoten. Die gestrichelten Hyperkanten stellen Prädiktionskanten dar und verbinden jeweils einelementige Situationsknoten–Mengen (z.B. $\{\mathbf{S1}\} \rightarrow \{\mathbf{S2}\}$). Die durchgezogene Hyperkante verbindet die einelementige Situationsknotenmenge $\{\mathbf{S1}\}$ mit der Menge $\{\mathbf{S3},\mathbf{S4}\}$. Sie stellt eine Spezialisierungskante des SGT dar.

$U_{j,P} = (U_{j,P}{}^{-}, U_{j,P}{}^{+})$ mit $|U_{j,P}{}^{-}| = |U_{j,P}{}^{+}| = 1$ modelliert[1]. Sie verbinden somit einelementige Knotenmengen. Spezialisierungskanten dagegen werden durch Hyperkanten der Form $U_{j,S} = (U_{j,S}{}^{-}, U_{j,S}{}^{+})$ mit $|U_{j,S}{}^{-}| = 1$ dargestellt und verbinden somit eine einelementige Knotenmenge mit einer Knotenmenge beliebiger Grösse. Diese zweite Knotenmenge beschreibt dann implizit einen Situationsgraphen, der die in dieser Menge enthaltenen Situationsknoten umfaßt. Eine explizite Modellierung von Situationsgraphen in Form von Knoten entfällt. Die Darstellung eines SGT als Hypergraph ist in Abbildung 4.2 zu sehen.

Die bereitzustellende grafische Anzeige für SGTs soll im Rahmen der vorliegenden Arbeit vor allem zum schnellen Auffinden von Fehlern bei der Generierung von Verhaltensschemata dienen. Eine spätere Anwendung dieser grafischen Schnittstelle zu SGTs könnte daneben aber auch die *interaktive* Erstellung von Schemata sein. Ein initia-

[1] Der *normale* gerichtete Graph stellt gerade einen Spezialfall des gerichteten Hypergraphen dar, in dem für jedes $U_j = (U_j{}^{-}, U_j{}^{+})$ gilt $|U_j{}^{-}| = |U_j{}^{+}| = 1$

les Verhaltensschema könnte hierbei aus natürlichsprachlichen Texten gewonnen und dem Benutzer des Systems anschließend zur grafischen Bearbeitung präsentiert werden. Endziel einer solchen Vorgehensweise könnte dann eine Entwicklungsoberfläche sein, die dem Benutzer das Erstellen und Bearbeiten von Verhaltensschemata sowohl mit sprachlichen als auch mit grafischen Mittel ermöglicht.

4.2.1 Vergleich unterschiedlicher Möglichkeiten der Darstellung

Zur Entwicklung einer grafischen Anzeige von Situationsgraphenbäumen bieten sich verschiedene Möglichkeiten. Eine zeitsparende Herangehensweise besteht darin, verfügbare bestehende Werkzeuge zur Anzeige von Graphen bzw. Diagrammen zu verwenden. Diese sind ggf. an die Struktur von SGTs anzupassen, falls dies möglich ist. Eine weitere Möglichkeit besteht in der Eigen-Implementierung der grafischen Schnittstelle. Diese beiden Methoden haben den Vorteil, daß sie jeweils als Pakete oder Klassen in ein bestehendes System integriert werden können. Eine dritte Möglichkeit schließlich besteht in der Ausgabe der SGTs in ein Dokumentdatei-Format, das mächtig genug ist, um dessen Struktur wiederzugeben. Die Anzeige einer solche Dokumentdatei könnte dann von hierfür vorgesehenen Anwendungen übernommen werden. Diese drei unterschiedlichen Vorgehensweisen sollen im folgenden verglichen werden.

4.2.1.1 Verwendung von Graph-Visualisierungswerkzeugen

Eine mögliche Vorgehensweise zur Erstellung einer grafischen Schnittstelle besteht in der Verwendung eines fertigen Werkzeuges zur Graph-Darstellung. Diese Werkzeuge sind mehr oder weniger leistungsfähig, manche sind frei erhältlich, manche kommerziell. Von den gefundenen Werkzeugen (vgl. etwa [GD 1] oder [GD 2]) zur Graph-Darstellung wurden zunächst diejenigen aussortiert, die nicht frei erhältlich sind. Von den verbleibenden wurden desweiteren nur diejenigen betrachtet, die in Java implementiert wurden und deren Quelltext verfügbar ist, um nötige Änderungen an der Anzeige bzw. die Möglichkeit der Interaktion des Benutzers mit dem angezeigten Graph dem bestehenden Werkzeug hinzufügen zu können. Gemäß dieser Auslese wurden die folgenden Werkzeuge näher untersucht:

- *VGJ* (*Visualizing Graphs with Java*, s. [VGJ]) ist ein in Java implementierter Graphen-Editor. Das Werkzeug bieten neben dem Erstellen von neuen Graphen auch die Möglichkeit, anzuzeigende Graphen aus Textdateien einzulesen, die dem *GML*–Standard (*Graph Modelling Language*, s. [GML]) entsprechen. *VGJ* enthält verschiedene Anordnungs–Algorithmen für gerichtete und ungerichtete, zyklische wie azyklische Graphen. Hypergraphen können, unter anderem wegen der Bindung an den *GML*–Standard, nicht angezeigt bzw. repräsentiert werden.

- *GraphPanel 2.1* (s. [GraphPanel]) stellt wie *VGJ* einen Graph–Editor dar. Der Funktionsumfang dieses Werkzeuges ist jedoch noch weiter eingeschränkt als der oben beschriebene. Auch hier fehlt die Möglichkeit zur Darstellung von hierarchischer Graphen bzw. von Hypergraphen.

- *DiaGen* (*Diagram Editor Generator*, s. [DiaGen]) stellt einen sehr interessanten Ansatz zur grafischen Anzeige fast beliebiger Diagramme dar. Die Art der anzuzeigenden Diagramme wird dem Werkzeug mittels einer Textdatei übergeben. Diese enthält neben einer Gammatik, der die später anzuzeigenden bzw. zu erstellenden Diagramme genügen müssen, auch Nebenbedingungen, welche die Anordnung einzelner Diagramm–Komponenten untereinander in der späteren Anzeige bestimmen. Da die Grammatik der anzuzeigenden Diagramme innerhalb des von *DiaGen* vorgegebenen Rahmens frei definiert werden kann, wird davon ausgegangen, daß dieses Werkzeug prinzipiell mächtig genug wäre, SGTs editierfähig anzuzeigen.

Zusammenfassend ist zu sagen, daß von den gefundenen Werkzeugen lediglich *DiaGen* mächtig genug erscheint, um SGTs darzustellen. Der Umstand, daß mit diesem Werkzeug auch sofort ein Editor für SGTs zur Verfügung stünde, stellt einen weiteren Vorteil dar. Versuche, eine von *DiaGen* verarbeitbare Grammatik für SGTs zu erstellen schlugen jedoch bisher fehl. Auch die nötige Angabe von Nebenbedingungen, die letztendlich das Erscheinungsbild der Diagrammanzeige bestimmen, deutet darauf hin, daß die durch die Verwendung fertiger Werkzeuge angestrebte *zeitsparende* Erstellung einer grafischen Schnittstelle zu SGTs mit diesem Werkzeug nicht zu erreichen ist.

4.2.1.2 Eigen–Implementierung eines Visualisierungswerkzeuges

Die zweite oben erwähnte Möglichkeit der Erstellung einer grafischen Schnittstelle zu SGTs besteht in der Eigen–Implementierung der dazu notwendigen Klassen und Pakete. Diese Vorgehensweise hätte den Vorteil, daß das damit erstellte Werkzeug leicht in das Gesamtsystem zur Erstellung von Verhaltensschemata integriert werden könnte. Desweiteren ist bei einer Eigen–Implementierung die genaue Anpassung an die durch die Struktur von SGTs gestellten Bedürfnisse prinzipiell mit entsprechendem Zeitaufwand möglich. Bei genauerer Betrachtung der in Kapitel 2.2 beschriebenen Struktur von SGTs wurde jedoch klar, daß diese Aufgabe nicht mit vertretbaren Zeitaufwand zu bewerkstelligen ist. Allein die übersichtliche Darstellung – und nur eine solche würde eine grafischen Schnittstelle rechtfertigen – einzelner Situationsgraphen gestaltet sich dadurch schwierig, daß diese Graphen in den meisten Fällen zyklisch sind. Einige Probleme beim Zeichnen von Graphen scheinen darüberhinaus noch Gegenstand der Forschung zu sein (vlg. etwa [GD 3]), weshalb eine Eigen–Implementierung der grafischen Schnittstelle im Rahmen der vorliegenden Arbeit – und der damit verbundenen Zeitbeschränkung – nicht angestrebt wird.

4.2.1.3 Darstellung von Situationsgraphenbäumen als HTML–Bäume

Die dritte der oben erwähnten Möglichkeiten zur grafischen Anzeige besteht in der Überführung des SGT in ein geeignetes Dokumentformat. Geeignet heißt hierbei, daß die Struktur von SGTs in dem Dokumentformat repräsentierbar sein muß. Ein solches Dokumentformat ist mit der Beschreibungssprache *HTML* (*Hypertext Markup Language*) gegeben. Dokumente bestehen hierbei einerseits aus einem Inhalt (in Form von Text, Bildern oder ähnlichem), andererseits aber aus Verweisen auf andere Stellen im Dokument oder gar auf andere Dokumente (siehe auch [Musc. & Kenn. 98]). Überträgt man nun die Inhalte von Situationsschemata in textuelle Inhalte von HTML–Dokumenten, die Prädiktions– und Spezialisierungskanten dagegen in Verweise zwischen diesen HTML–Dokumenten, kann auf diese Weise eine Menge von HTML–Dokumenten generiert werden, welche Struktur und Inhalt eines SGT wiedergeben. Diese HTML–Dokumente können dann mit einem herkömmlichen Netz–Stöberer (*Browser*) angezeigt werden. Es sei an dieser Stelle darauf hingewiesen, daß diese Form der Darstellung zwei klare Nachteile gegenüber den beiden zuvor genannten Arten besitzt. Zum einen stellen auf die oben beschriebene Weise generierte HTML–Dokumente eine Sackgasse dar, d.h. sie dienen der reinen Anzeige von SGTs und lassen keine Bearbeitung dieser Verhaltensschemata zu. Zum anderen kann mit Hilfe von HTML–Dokumenten in der oben beschriebenen Form keine Übersicht über den gesamten SGT generiert werden. Ein HTML–Dokument enthält stets nur die Informationen eines Ausschnitts des Schemas. Die Zusammenhänge werden erst durch die Verbindungen zwischen den Dokumenten ersichtlich.

4.2.2 Zusammenfassung

Die vorangehenden Abschnitte stellen einige Möglichkeiten der grafischen Visualisierung von Situationgraphenbäumen vor. Die zunächst angestrebte zeitsparende Verwendung eines fertigen Visualisierungswerkzeuges scheiterte an der Komplexität des einzig für mächtig genug befundenen Werkzeuges. Eine Eigen–Implementierung wurde als prinzipiell möglich, jedoch wegen des sehr hohen Zeitaufwandes einer solchen Implementierung als im Rahmen der vorliegenden Arbeit nicht machbar eingeschätzt. Als Kompromiß zwischen Zeitaufwand und erreichtem Nutzen der Visualisierung wurde die automatische Generierung von HTML–Dokumenten aus SGTs zur Visualisierung gewählt. Abbildung 4.3 zeigt das Abbild einer Stöberer–Anzeige des HTML–Dokuments eines SGT. Das angezeigte Dokument gliedert sich dabei stets in drei Rahmen (*frames*). Links oben befindet sich ein Rahmen, der alle im Situationsgraphenbaum auftretenden Situationsgraphen enthält. Diese sind je nach ihrer Spezialisierungsstufe (von dem allgemeinsten bis zum speziellsten Situationsgraphen) sortiert. Situationsgraphen unterschiedlicher Spezialisierungsstufen sind durch horizontale Linien voneinander getrennt. Zusätzlich zu dem Namen eines Graphen wird hier auch diejenige Situation angegeben, welche durch diesen Graphen spezialisiert wird. Der linke untere Rahmen enthält eine Liste aller im oben links ausgewählten Situations-

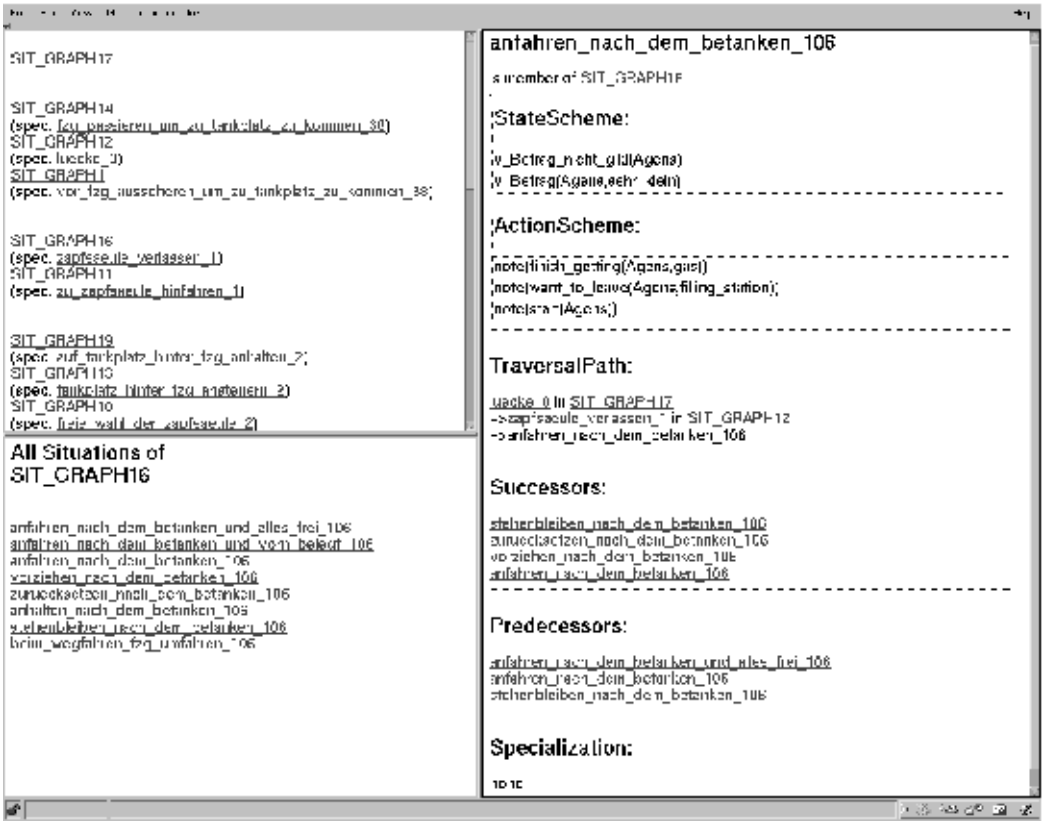


Abbildung 4.3: Abbild der Stöberer–Oberfläche bei der Anzeige eines aus einem Situationsgraphenbaum automatisch generierten HTML–Dokuments. Das Dokument gliedert sich in drei Rahmen: links oben eine Übersicht aller auftretenden Situationsgraphen, links unten eine Liste aller in einem ausgewählten Situationsgraphen auftretenden Situationsschemata. Der rechte Rahmen enthält die textuelle Darstellung eines ausgewählten Situationschemas. Nähere Erläuterung im Text.

graphen auftretenden Situationsschemata. Bei Auswahl eines dieser Schemata wird im rechten Rahmen der Inhalt dieses Schemas angezeigt. Beginnend mit dem Namen des Schemas und der Angabe der Graphenzugehörigkeit folgt dann die tabellarische Angabe des Zustands– und Aktionsschemas (*StateScheme* bzw. *ActionScheme*). Daran anschließend wird angegeben, auf welchem Spezialisierungspfad von einem Situationsschema des allgemeinsten Graphen aus das gerade angezeigte Schema erreichbar ist (*TraversalPath*). Es folgt eine Liste der Prädiktionskanten von diesem Schema aus zu Nachfolgeschemata (*Successors*) bzw. Angaben über Prädiktionskanten, die von anderen Schemata auf dieses Schema verweisen (*Predecessors*). Abschließend wird eine möglicherweise vorhandene Spezialisierung dieses Schemas angegeben (*Specialization*). Situationsgraphennamen und Namen von Situationsschemata sind hierbei stets als Verknüpfung ausgelegt. Das Anklicken dieser Verknüpfungen bewirkt eine Anzeige der

ausgewählten Information im dafür vorgesehenen Rahmen des Dokuments. Detailinformationen können so direkt abgelesen werden, während die Struktur des gesamten SGT sich durch das Verfolgen der Verknüpfungen erschliessen läßt.

Kapitel 5

Umwandlung von DRSen in Verhaltensschemata

In Kapitel 3 wurde dargestellt, wie aus Verhaltensbeschreibungen DRSen generiert werden können. Kapitel 4 befaßte sich mit der Repräsentation und geeigneten Darstellung von Verhaltensschemata in Form von Situationsgraphenbäumen. Das folgende Kapitel behandelt die Frage, wie das in (aus Verhaltensbeschreibungen erstellten) DRSen enthaltene Wissen in Verhaltensschemata überführt werden kann.

5.1 Motivation und Vorgehensweise

5.1.1 Zur Bedeutung einer DRS

[Kamp & Reyle 93] geben die Bedeutung einer DRS $K = \langle \mathcal{U}, Con \rangle$ durch eine Übersetzung der DRS in eine äquivalente Formel der Prädikatenlogik Erster Stufe an. Hierzu fordern sie zunächst die Endlichkeit der beiden Mengen $\mathcal{U}$ und Con. Diese Forderung stellt jedoch nur eine formale Einschränkung der bisherigen Definition einer DRS dar, da notwendigerweise jede aus einem endlichen Text erzeugte DRS selbst endlich ist. Eine weitere Forderung erwächst aus der Linearität prädikatenlogischer Formeln: sowohl $\mathcal{U}$ als auch Con sind definiert als Mengen und daher nicht geordnet. Eine algorithmische Übersetzung der *ungeordneten* DRS in eine *lineare* Formel setzt daher die Wahl einer (beliebigen) Ordnung auf $\mathcal{U}$ und Con voraus. Die Autoren weisen daraufhin, daß alle möglichen Ordnungen zu möglicherweise syntaktisch verschiedenen, logisch aber äquivalenten Formeln führen.
Der Wahrheitswert einer DRS ist damit gegeben durch den Wahrheitswert einer äquivalenten logischen Formel. Dieser wird durch die Abbildung der Formel auf ein zugrunde liegendes *Modell* $\mathcal{M} = \langle U_{\mathcal{M}}, Name_{\mathcal{M}}, Pred_{\mathcal{M}} \rangle$[1] gefunden. Dieses Modell beinhaltet ei-

[1] Zur Behandlung von Mengen in Form von Plural–Bezugsträgern (s. [Kamp & Reyle 93], S. 425) sowie zur Behandlung von Ereignissen (s. [Kamp & Reyle 93], S. 677) erweitern die Autoren das hier

ne Menge von Individuen $U_{\mathcal{M}}$, eine Funktion $Name_{\mathcal{M}}$, welche jedem Eigennamen ein ihn tragendes Individuum aus $U_{\mathcal{M}}$ zuweist, und eine Abbildung $Pred_{\mathcal{M}}$, welche jedem n–stelligen Prädikat der Formel eine Menge von Tupeln aus ${U_{\mathcal{M}}}^n$ zuweist. Ein Namensprädikat $n(x)$ der Formel wird als wahr interpretiert, wenn eine Funktion f existiert, die der Variablen x ein Element a aus $U_{\mathcal{M}}$ zuweist, so daß $Name_{\mathcal{M}}(n) = a$. Analog wird ein Prädikat $p(x_0, \ldots, x_n)$ als wahr interpretiert, wenn eine Funktion f existiert, die jedem x_i ein a_i zuweist, so daß $(a_0, \ldots, a_n) \in Pred_{\mathcal{M}}(p)$. Das Modell $\mathcal{M}$ stellt insofern die *Grundwahrheit* des betrachteten Diskursbereichs dar. Jede DRS (bzw. die ihr entsprechende logische Formel) wird auf diese Grundwahrheit abgebildet, indem alle ihre Bedingungen auf das Modell abgebildet werden.
Dies macht jedoch nur Sinn, wenn tatsächlich jede Bedingung einer DRS eine Aussage über die Grundwahrheit des Diskursbereichs darstellt. In der vorliegenden Arbeit werden DRSen aus Verhaltensbeschreibungen erstellt. Diese Texte beinhalten sowohl Informationen über den Diskursbereich *Straßenverkehr* als auch Informationen darüber, wie diese ersteren Informationen in ein Verhaltensschema zu integrieren sind. Als Beispiel sei hier der Satz *(III,1)* der in Anhang B.1.2 aufgeführten Beispiel–Verhaltensbeschreibung genannt:

> *'Driving' can be specialized ... by the requirement that the agent drives behind a patient vehicle.*

"*'Driving'*" und "*the agent drives behind a patient vehicle*" sind schema–artige Informationen über Vorgänge im Diskursbereich *Straßenverkehr*. Die zweite der beiden Aussagen etwa kann als ein zwei–stelliges Prädikat `drive_behind(`a`,`b`)` aufgefaßt werden, wobei a und b Platzhalter für Elemente aus der betrachteten Individuen–Menge des Diskursbereichs darstellen. Informationen dieser Art sollen im folgenden als *diskursspezifische* Informationen bezeichnet werden. "*can be specialized*" dagegen besagt, daß der Begriff "*'Driving'*" durch einen anderen Begriff ("*the requirement that ...*") spezialisiert werden kann. Analog zum obigen Beispiel würde man diese Aussage wiederum als zweistelliges Prädikat `specialize(`c`,`d`)` auffassen. Hierbei sind jedoch c und d keine Platzhalter für Individuen des obigen Diskursbereichs, sondern sie bezeichnen *selbst* schema–artige Informationen über den Diskursbereich. Dieses Prädikat kann deshalb nicht als diskursspezifisch verstanden werden. Es stellt vielmehr eine Anweisung dar, wie andere diskursspezifische Informationen zu kombinieren sind. Informationen dieser Art sollen im folgenden stets als *schemaspezifisch* bezeichnet werden. Informationen der beiden genannten Arten müssen bei einer Überführung eines Textes in ein Verhaltensschema unterschiedlich behandelt werden. An welchem Punkt der Überführung dies am sinnvollsten zu geschehen hat, soll im folgenden untersucht werden.

---

genannte Modell um entsprechende Strukturen. Diese Erweiterungen betreffen das hier betrachtete Problem jedoch nicht.

5.1.2 Trennung der Informationsarten bei der DRS–Erstellung

Der erste Schritt der Überführung einer textuellen Verhaltensbeschreibung in ein Verhaltensschema besteht in der Erzeugung einer DRS aus dem Text. Eine von mehreren Möglichkeiten der unterschiedlichen Verarbeitung von diskurs– und schemaspezifischen Informationen besteht daher in der unterschiedlichen Behandlung der entsprechenden Teilphrasen des Textes bei der DRS–Erstellung. Der Unterschied zwischen diskursspezifischen und schemaspezifischen Informationen soll an einem weiteren Beispiel verdeutlicht werden. Man betrachte die beiden Sätze:

(1) *A is the left lane of B.*

(2) *C is the last specialization of D.*

A, *B*, *C* und *D* sind in diesem Fall Platzhalter für beliebige Nominal–Phrasen. Satz (1) weist *A* als die linke Fahrbahn von *B* aus. Hierbei handelt es sich also um eine diskursspezifische Information über den Diskursbereich *Straßenverkehr*. Satz (2) besagt, daß *C* die letzte Spezialisierung von *D* sei. Dies stellt eine schemaspezifische Information dar. Interessant ist an diesem Beispiel vor allem die syntaktisch gleiche Struktur der beiden Sätze, was sich in strukturell identischen Syntaxbäumen widerspiegelt[2]. Konstruktionsregeln, deren auslösende Struktur in dem einen Syntaxbaum gefunden wird, wären also auch auf den anderen Syntaxbaum anwendbar und würden somit zur Verarbeitung beider Phrasen den gleichen Beitrag leisten. Eine unterschiedliche Behandlung der Informationsarten aufgrund von Unterschieden in der syntaktischen Struktur der entsprechenden Phrasen ist daher nicht möglich.
Die auslösende Struktur einer Konstruktionsregel kann jedoch neben der reinen Struktur auch die Forderung nach einer bestimmten Belegung einzelner Attribute innerhalb dieser Struktur beinhalten. Es wäre daher denkbar, bestimmte Wörter der Sprache mit entsprechenden Attributen auszustatten (etwa einem Attribut *information_kind*, mit den Belegungen *DISCOURSE* und *SCHEME*). Auf unterschiedliche Belegungen dieser Attribute könnten dann entsprechend unterschiedliche Konstruktionsregeln anwendbar sein. Einer getrennten Behandlung von diskurs– und schemaspezifischen Informationen bei der DRS–Erstellung stünde dann nichts mehr im Wege.
Das Problem der Trennung der beiden Informationsarten besteht neben dem Erkennen im Syntaxbaum des Textes jedoch auch in der DRS–seitigen, getrennten Repräsentation dieser Informationsarten. Die DRS in der hier vorgestellten Form eignet sich für eine solche Trennung von diskurs– und schemaspezifischen Informationen jedoch nicht – insbesondere unter Beachtung der im vorangehenden Abschnitt beschriebenen Standard–Übersetzung in eine logische Formel.

[2]Dies gilt zumindest unter der Voraussetzung, daß *"lane"* und *"specialization"* beide als Nomen und damit als grammatikalisch gleichwertig behandelt werden. Die Möglichkeit, für diese und ähnliche Wörter unterschiedliche Nichtterminale einzuführen, um so die Erzeugung unterschiedlicher Syntax–Bäume zu *erzwingen*, wird hier nicht betrachtet, da hierdurch die Grenze zwischen Syntax und Semantik zu verwischen droht.

Als Fazit dieser Überlegungen bleibt festzuhalten, daß eine getrennte Behandlung von diskurs– und schemaspezifischen Informationen bereits bei der Erstellung von DRSen aus Syntaxbäumen möglich wäre. Die Notwendigkeit einer getrennten Repräsentation dieser Informationen in einer DRS ist bisher jedoch nicht gegeben. Eine Anpassung der DRS an diese Trennung der Informationsarten, was auch eine Anpassung der bestehenden Werkzeuge zur DRS–Erstellung zur Folge hätte, wird in der vorliegenden Arbeit daher nicht angestrebt.

5.1.3 Trennung der Informationsarten bei der DRS–Transformation

Eine zweite Möglichkeit der Trennung von diskurs– und schemaspezifischen Informationen besteht in der unterschiedlichen *Transformation* der entsprechenden DRS–Bedingungen. Aus dem Syntaxbaum einer Verhaltenbeschreibung wird zunächst eine DRS erstellt, wobei die Informationsarten noch nicht getrennt werden. Diese Trennung erfolgt erst bei der Überführung der DRS in ein Verhaltensschema. *Diskursspezifische* Informationen innerhalb einer DRS – etwa in Form von Prädikat–Bedingungen – sollten bei dieser Überführung entsprechend der oben beschriebenen Standard–Übersetzung einer DRS in eine logische Formel behandelt werden: dies führt zu Prädikaten der logischen Formel. *Schemaspezifische* Informationen dagegen beschreiben die Art und Weise, wie diskursspezifische Informationen zu verknüpfen sind. Hierdurch stellen sich zwei Fragen, die jedoch eng miteinander zusammenhängen: zum einen, wie diskurs– und schemaspezifische Informationen in einer DRS jeweils erkannt werden können, und zum anderen, falls eine solche Erkennung gelingt, wie dann die Bedeutung der schemaspezifischen Informationen bestimmt werden kann.
Zur Beanwortung beider Fragen dient ein *Bedeutungs–Wörterbuch*. In diesem Wörterbuch ist zu jedem schemaspezifischen Prädikat dessen Bedeutung in Form von Aktionen auf dem Verhaltensschema gespeichert. Ist ein Prädikat einer DRS zu transformieren, wird zunächst in diesem Wörterbuch nachgesehen, ob zu dem Prädikat eine Bedeutung gespeichert ist. Ist dies der Fall, kann das Prädikat als schemaspezifisch betrachtet werden und die gefundenen Aktionen werden ausgeführt. Ist keine Bedeutung des Prädikats bekannt, wird es als diskursspezifisch angesehen. Hierdurch kann also auch die Trennung von schema– und diskursspezifischen Informationen geschehen.
Eine weitere Frage betrifft die Reihenfolge der Transformation von diskurs– und schemaspezifischen Informationen: kann die Transformation der beiden Informationsarten strikt getrennt werden? Wenn ja, welche der beiden Informationsarten sollte zuerst transformiert werden? Zur genaueren Untersuchung dieser Frage soll der folgenden Beispielsatz betrachtet werden:

(3) *'Driving' specializes 'moving'.*

Dieser Satz besagt, das der Begriff *"driving"* den Begriff *"moving"* spezialisiert. In der Terminologie von Situationsgraphenbäumen bedeutet dies, daß eine Situationschema,

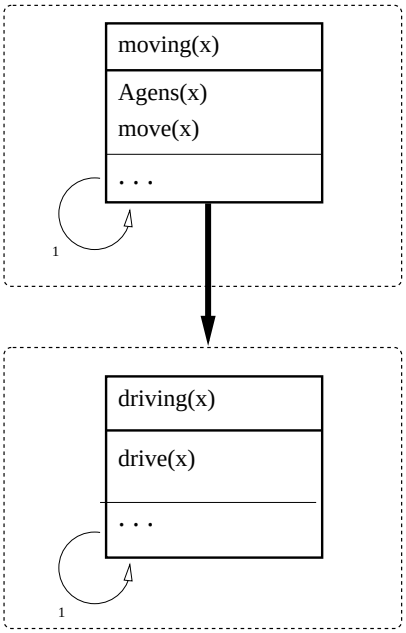
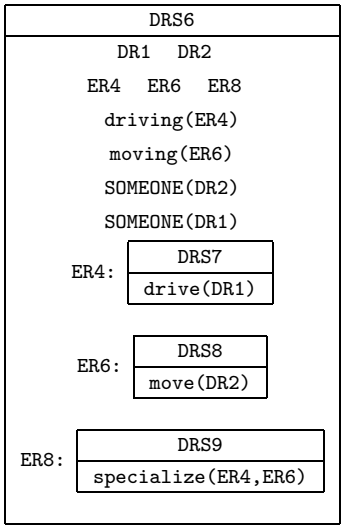


Abbildung 5.1: Ein möglicher Situationsgraphenbaum zu Beispielsatz (3). Nähere Erläuterung im Text.

in dem ein Akteur als *sich bewegend* (*moving*) beschrieben wird, einen spezialisierenden Situationsgraphen besitzt, der zumindest ein Situationschema enthält, welches den Akteur als *fahrend* (*driving*) beschreibt. Ein entsprechender SGT ist in Abbildung 5.1 dargestellt. Die mit dem Werkzeug von [Arens & Ottlik 2000] aus dem Beispielsatz erzeugte DRS hat die Form:



Sie besteht aus den zwei Individuen–Bezugsträgern `DR1` und `DR2` sowie den drei Ereignis–Bezugsträgern `ER4`, `ER6` und `ER8`. Desweiteren beinhaltet die DRS zwei Namens–Bedingungen (`moving(ER6)` und `driving(ER4)`), die aus der Nennung der entsprechenden Teilphrasen des Satzes in Zitat–Form resultieren. Die beiden Attribut–Bedingungen `SOMEONE(DR1)` und `SOMEONE(DR2)` wurden in die DRS eingefügt, da durch den Satz keine Subjekte der beiden Verben *drive* und *move* gegeben und diese somit unbekannt sind. Einen wesentlichen Teil der obigen DRS nehmen die Definitionen der drei Ereignis–Bezugsträger ein. Zwei dieser Definitionen (`ER4` und `ER6`) beinhalten diskursspezifische Prädikate (`drive(DR1)` und `move(DR2)`). Die dritte Definition schließlich beinhaltet das schemaspezifische Prädikat `specialize(ER4,ER6)`.
Die Spezialisierungsbeziehung ist im Sinne der Terminologie von SGTs eine Beziehung zwischen einem Situationsschema und einem dieses Schema spezialisierenden Situationsgraphen. Beide Argumente des oben genannten Prädikats `specialize(ER4, ER6)` sind jedoch Ereignis–Bezugsträger. Bei der Transformation einer DRS mit der Transformation der schemaspezifischen Informationen zu beginnen, erscheint deshalb in diesem Fall nicht sinnvoll, da an diesem Punkt noch nicht bekannt ist, ob sich die beiden Ereignis–Bezugsträger auch als die von der Spezialisierungsbeziehung gewünschten SGT–Elemente interpretieren lassen. Auch ist noch nicht bekannt, welche weiteren Informationen der DRS zu einer solchen Interpretation benötigt würden.
Dies spricht zunächst dafür, bei der Transformation der DRS mit der Transformation der diskursspezifischen Informationen zu beginnen. Eine solche diskursspezifische Information stellt z.B. das Prädikat `drive(DR1)` dar. Wie in Abbildung 5.1 zu sehen ist, soll dieses Prädikat in das Zustandsschema einer Situation übernommen werden. Eine solche Situation existiert jedoch zu diesem Zeitpunkt noch nicht. Dies setzt die Transformation einer schemaspezifischen Information voraus, die zum Erzeugen einer solchen Situation führt. Also führt auch die Reihenfolge, erst die diskursspezifischen, und danach die schemaspezifischen Informationen zu transformieren, nicht zum Ziel.
Dieser Widerspruch führt zu der Schlußfolgerung, daß eine strikte Trennung der Transformation der beiden Informationsarten nicht möglich ist. Vielmehr hängt die Transformation einer einzelnen DRS–Bedingung häufig von der Transformation anderer Bedingungen ab. Im folgenden soll deshalb versucht werden, ein Verfahren zu entwickeln, das die Transformation von aus Verhaltensbeschreibungen erstellten DRSen schrittweise vornimmt und dabei nicht von einzelnen Bedingungen, sondern von ganzen Teilmengen der DRS ausgeht. Jeder der dabei ausgeführten Schritte kann somit sowohl die Transformation diskurs– als auch schemaspezifischer Informationen beinhalten.

5.2 DRS–Transformation durch Transformations–Regeln

Ähnlich dem von [Kamp & Reyle 93] vorgestellten Verfahren zur Umwandlung von Syntax–Bäumen in DRSen soll ein Verfahren entwickelt werden, daß eine DRS *schritt-*

weise in ein Verhaltensschema überführt. An die Stelle der Konstruktionsregeln im oben erwähnten Verfahren treten nun *Transformationsregeln* (TRn). War die Anwendbarkeit einer Konstruktionsregel auf einen Syntaxbaum abhängig vom Auffinden einer auslösenden Struktur *TS* darin, soll die Anwendbarkeit einer TR abhängig sein vom Auffinden eines bestimmten Musters P (*pattern*) in einer zu transformierenden DRS. Da dieses Muster selbst wieder als eine DRS angegeben werden kann, wird im folgenden zu klären sein, wann eine Muster–DRS P in einer zu transformierenden DRS K enthalten ist.
Nach der Erkennung der Anwendbarkeit einer Konstruktionsregel wurde in dem Verfahren von [Kamp & Reyle 93] stets der Aktionsteil der Regel ausgeführt. Analog hierzu soll auch eine TR einen Aktionsteil besitzen. Dieser kann sowohl aus Aktionen, die das Verhaltensschema aufbauen, als auch aus Aktionen, die Veränderungen an der zu transformierenden DRS vornehmen, bestehen.
Die unterschiedliche Behandlung von *diskursspezifischen* und *schemaspezifischen* Informationen innerhalb einer DRS soll durch den Rückgriff auf ein *Bedeutungs–Wörterbuch* bewirkt werden. Hierin wird die Bedeutung der schemaspezifischen Informationen, welche als Anweisungen zum Aufbau des Verhaltensschemas und zur Integration der diskursspezifischen Informationen in dieses Schema zu verstehen sind, wiederum durch Aktionen auf dem Verhaltensschema als auch auf der DRS definiert. Der Zugriff auf dieses Bedeutungs–Wörterbuch erfolgt dann mittels einer speziellen TR.

5.2.1 Mustererkennung auf DRSen

Zur schrittweisen Transformation einer DRS in ein Verhaltensschema mit Hilfe von TRn ist es nötig zu entscheiden, ob eine TR auf die DRS im aktuellen Zustand anwendbar ist, d.h. ob die Muster–DRS $P = \langle \mathcal{U}_P, Con_P \rangle$ der TR in der zu transformierenden DRS $K = \langle \mathcal{U}_K, Con_K \rangle$ *enthalten* ist. Die Muster–DRS P kann hierbei eine DRS bezüglich des Vokabulars V und der Bezugsträgermenge R im bisherigen Sinne (vgl. Kapitel 2.3.2) darstellen. Unter dem *Enthaltensein* von P in K soll dann verstanden werden, daß jeder Bezugsträger aus $\mathcal{U}_P$ auch in $\mathcal{U}_K$ vorkommt (also $\mathcal{U}_P \subseteq \mathcal{U}_K$), sowie daß jede Bedingung aus Con_P auch in Con_K vorkommmt ($Con_P \subseteq Con_K$).
Häufiger jedoch wird es nötig sein, in einer Muster–DRS P auch Platzhalter für Bezugsträger aus R und Bezeichner aus V zu verwenden, da durch die Muster–DRS lediglich bestimmte Konstellationen von Bezugsträgern und Bedingungen gefordert werden sollen, unabhängig von deren aktueller Belegung in einer zu transformierenden DRS. Dies erfordert zunächst die Erweiterung des Vokabulars und der Menge von zulässigen Bezugsträgern, bezüglich derer eine Muster–DRS definiert werden soll. Das Vokabular einer Muster–DRS muß um Zeichenfolgen erweitert werden, die als Platzhalter für einer Zeichenfolge aus V fungieren können. Entsprechend wird die Bezugsträgermenge R um Platzhalter für Bezugsträger zu erweitern sein.

Definition 5.1 *Die Menge Var_{DR} der zulässigen Platzhalter für Individuen–Bezugsträger aus R sei definiert durch $Var_{DR} := \{\texttt{<vDR}i\texttt{>} | i \in I\!N\}$.*

Definition 5.2 *Die Menge Var_{ER} der zulässigen Platzhalter für Ereignis–Bezugsträger aus R sei definiert durch $Var_{ER} := \{\texttt{<vER}i\texttt{>} | i \in \mathbb{N}\}$.*

Definition 5.3 *Die Menge Var_{PDR} der zulässigen Platzhalter für Plural–Bezugsträger aus R sei definiert durch $Var_{PDR} := \{\texttt{<vPDR}i\texttt{>} | i \in \mathbb{N}\}$.*

Definition 5.4 *Die Menge Var_{Name} der zulässigen Platzhalter für Namens–Bezeichner aus V sei definiert durch $Var_{Name} := \{\texttt{<vNAME}i\texttt{>} | i \in \mathbb{N}\}$.*

Definition 5.5 *Die Menge $Var_{Attribute}$ der zulässigen Platzhalter für Attributs–Bezeichner aus V sei definiert durch $Var_{Attribute} := \{\texttt{<vATTRIBUTE}i\texttt{>} | i \in \mathbb{N}\}$.*

Definition 5.6 *Die Menge $Var_{Predicate}$ der zulässigen Platzhalter für Prädikats–Bezeichner aus V sei definiert durch $Var_{Predicate} := \{\texttt{<vPREDICATE}i\texttt{>} | i \in \mathbb{N}\}$.*

Definition 5.7 *Die Menge $Var_{Relation}$ der zulässigen Platzhalter für Relations–Bezeichner aus V sei definiert durch $Var_{Relation} := \{\texttt{<vRELATION}i\texttt{>} | i \in \mathbb{N}\}$.*

Definition 5.8 *Die Menge $Var_{Quantifier}$ der zulässigen Platzhalter für Quantoren–Bezeichner aus V sei definiert durch $Var_{Quantifier} := \{\texttt{<vQUANTIFIER}i\texttt{>} | i \in \mathbb{N}\}$.*

Die Elemente der obigen Platzhalter–Mengen ergeben sich durch die Konkatenation einer bestimmten Zeichenfolge, welche die Verwendung eines Platzhalters angibt (etwa `vDR` für Individuen–Bezugsträger aus R und `vPREDICATE` für Prädikatsbezeichner aus V) und einem eindeutigen Zähler $i \in \mathbb{N}$. Mit Hilfe der obigen Definitionen kann nun die Menge der zulässigen Bezugsträger und das Vokabular einer Muster–DRS bei gegebenem Vokabular V und gegebener Bezugsträger–Menge R definiert werden.

Definition 5.9 *Die Menge R_P der zulässigen Bezugsträger einer Muster–DRS zu einer gegebenen Bezugsträger–Menge R ist definiert durch*
$R_P := R \cup Var_{DR} \cup Var_{ER} \cup Var_{PDR}$.

Definition 5.10 *Das Vokabular V_P einer Muster–DRS zu einem gegebenen Vokabular V ist definiert durch*
$V_P := V \cup Var_{Name} \cup Var_{Attribute} \cup Var_{Predicate} \cup Var_{Relation} \cup Var_{Quantifier}$.

Bei gegebenen Vokabular V und gegebener Bezugsträger–Menge R ist die Menge der Muster–DRSen demnach gegeben als die Menge aller DRSen bezüglich des Vokabulars V_P und der Bezugsträger–Menge R_P. Insbesondere ist also auch jede DRS bezüglich V und R eine Muster–DRS, da sowohl $V \subset V_P$ als auch $R \subset R_P$ gilt. Im folgenden soll untersucht werden, wann eine solche Muster–DRS bezüglich V_P und R_P in einer DRS bezüglich V und R enthalten ist.

Der einfachste zu untersuchende Fall tritt ein, wenn eine Muster–DRS $P = \langle \mathcal{U}_P, Con_P \rangle$ nur Elemente aus V und R *benutzt*. Es gilt dann einerseits $\mathcal{U}_P \subseteq R (\subset R_P)$. Andererseits kommen in jeder Bedingung aus Con_P nur Zeichenfolgen aus V vor. Das *Enthaltensein* einer solchen Muster–DRS in einer zu transformierenden DRS $K = \langle \mathcal{U}_K, Con_K \rangle$ kann dann durch einfache Teilmengenbeziehungen ausgedrückt werden. Jeder Bezugsträger aus $\mathcal{U}_P$ muß eine Entsprechung in $\mathcal{U}_K$ findet (also $\mathcal{U}_P \subseteq \mathcal{U}_K$) und jede Bedingung aus Con_P muß auch in Con_K vorkommen (d.h. $Con_P \subseteq Con_K$).
Komplizierter wird es, wenn in P auch Platzhalter für Bezugsträger aus R oder für Zeichenfolgen aus V verwendet werden. In einem solchen Fall soll die Muster–DRS P in K *enthalten* sein, wenn für alle verwendeten Platzhalter in P eine *Variablenbelegung* in R und V gefunden werden kann, so daß die nach der Anwendung dieser Variablenbelegung resultierende DRS bezüglich V und R in K enthalten ist.

Definition 5.11 *Eine Abbildung* $f : (R_P \cup V_P) \rightarrow (R \cup V)$ *heißt Variablenbelegung* $:\Longleftrightarrow$

(i) $f_{|R} \equiv id_{|R}$,

(ii) $f_{|V} \equiv id_{|V}$,

(iii) $f(R_P) \subseteq R$ *und*

(iv) $f(V_P) \subseteq V$.

Eine Variablenbelegung f bildet also stets R_P auf R und V_P auf V ab, wobei Elemente aus V und R jeweils auf sich selbst abgebildet werden müssen. $f(R_P)$ ist hierbei definiert durch $f(R_P) = \{f(x) | x \in R_P\}$ ($f(V_P)$ analog).
Unter dem Ergebnis der Anwendung einer solchen Variablenbelegung f auf eine Bedingungsmenge Con_P soll diejenige Bedingungsmenge verstanden werden, die sich durch Anwendung von f auf jede Zeichenfolge aus V_P und jeden vorkommenden Bezugsträger aus R_P in Con_P ergibt. Die resultierende Bedingungsmenge wird mit $Con_{P,f}$ bezeichnet. Entsprechend wird unter dem Ergebnis der Anwendung einer Variablenbelegung f auf eine DRS $P = \langle \mathcal{U}_P, Con_P \rangle$ diejenige DRS verstanden, welche sich durch Anwendung von f auf das Universum $\mathcal{U}_P$ und die Bedingungsmenge Con_P ergibt. Die resultierende DRS soll dann mit $P_f = \langle f(\mathcal{U}_P), Con_{P,f} \rangle$ bezeichnet werden.

Definition 5.12 *Das Ergebnis der Anwendung einer Variablenbelegung* f *auf eine Bedingungsmenge* Con_P *wird mit* $Con_{P,f}$ *bezeichnet und sei definiert als die kleinste Menge, für die gilt:*

(i) für jede explizite Mengen–Definition $p{=}\bigoplus(arg_0, \ldots, arg_n) \in Con_P$ *mit* $p \in R_P$ *und* $arg_i \in R_P \; \forall i \in \{0, \ldots, n\}$ *ist* $f(p){=}\bigoplus(f(arg_0), \ldots, f(arg_n)) \in Con_{P,f}$,

(ii) für jede implizite Mengen–Definition $p{=}\sum dK_1 \in Con_P$ *(vgl. Kapitel 2.3.2) mit* $p, d \in R_P$ *und einer DRS* K_1 *bezüglich* V_P *und* R_P *ist* $f(p){=}\sum f(d)K_{1,f} \in Con_{P,f}$,

(iii) *für jede quantifizierte Mengen–Definition* $p\texttt{=}q\texttt{(}r\texttt{)} \in Con_P$ *mit* $p, r \in R_P$ *und* $q \in V_P$ *ist* $f(p)\texttt{=}f(q)\texttt{(}f(r)\texttt{)} \in Con_{P,f}$,

(iv) *für jede explizite Ereignis–Definition* $e\texttt{:}K_1 \in Con_P$ *mit* $e \in R_P$ *und einer DRS* K_1 *bezüglich* V_P *und* R_P *ist* $f(e)\texttt{:}K_{1,f} \in Con_{P,f}$,

(v) *für jede zusammenfassende Ereignis–Definition* $e\texttt{=[}arg_0\texttt{,}\ldots\texttt{,}arg_n\texttt{]} \in Con_P$ *mit* $e \in R_P$ *und* $arg_i \in R_P\ \forall i \in \{0,\ldots,n\}$ *ist* $f(e)\texttt{=[}f(arg_0)\texttt{,}\ldots\texttt{,}f(arg_n)\texttt{]} \in Con_{P,f}$,

(vi) *für jede Zeitraum–Definition* $e\texttt{=DURATION(}p\texttt{)} \in Con_P$ *mit* $e, p \in R_P$ *ist* $f(e)\texttt{=DURATION(}f(p)\texttt{)} \in Con_{P,f}$,

(vii) *für jede Namens–Bedingung* $name\texttt{(}arg\texttt{)} \in Con_P$ *mit* $name \in V_P$ *und* $arg \in R_P$ *ist* $f(name)\texttt{(}f(arg)\texttt{)} \in Con_{P,f}$,

(viii) *für jede Attributs–Bedingung* $attribute\texttt{(}arg\texttt{)} \in Con_P$ *mit* $attribute \in V_P$ *und* $arg \in R_P$ *ist* $f(attribute)\texttt{(}f(arg)\texttt{)} \in Con_{P,f}$,

(ix) *für jede Prädikats–Bedingung* $predicate\texttt{(}arg_0\texttt{,}\ldots\texttt{,}arg_n\texttt{)} \in Con_P$ *mit* $predicate \in V_P$ *und* $arg_i \in R_P\ \forall i \in \{0,\ldots,n\}$ *ist* $f(predicate)\texttt{(}f(arg_0)\texttt{,}\ldots\texttt{,}f(arg_n)\texttt{)} \in Con_{P,f}$,

(x) *für jede Relations–Bedingung* $relation\texttt{(}arg_0\texttt{,}arg_1\texttt{)} \in Con_P$ *mit* $relation \in V_P$ *und* $arg_i \in R_P\ \forall i \in \{0,1\}$ *ist* $f(relation)\texttt{(}f(arg_0)\texttt{,}f(arg_1)\texttt{)} \in Con_{P,f}$,

(xi) *für jede Duplex–Bedingung*

$$K_1 \quad \begin{matrix} q \\ d \end{matrix} \quad K_2 \quad \in Con_P$$

mit $d \in R_P$, $q \in V_P$ *und DRSen* K_1, K_2 *bezüglich* V_P *und* R_P *ist*

$$K_{1,f} \quad \begin{matrix} f(q) \\ f(d) \end{matrix} \quad K_{2,f} \quad \in Con_{P,f} \text{ und}$$

(xii) *für jede Unter–DRS sign* $K_1 \in Con_P$ *mit* $sign \in V_P$ *und einer DRS* K_1 *bezüglich* V_P *und* R_P *ist sign* $K_{1,f} \in Con_{P,f}$.

Mit Hilfe dieser Definition kann nun das *Enthaltensein* einer Muster–DRS in einer zu transformierenden DRS K definiert werden.

Definition 5.13 *Eine Muster–DRS* $P = \langle \mathcal{U}_P, Con_P \rangle$ *ist in einer DRS* $K = \langle \mathcal{U}_K, Con_K \rangle$ *enthalten* $:\Longleftrightarrow$ *es existiert eine Variablenbelegung* f, *so daß gilt:*

(i) $f(\mathcal{U}_P) \subseteq \mathcal{U}_K$ *und*

(ii) $Con_{P,f} \subseteq Con_K$.

Das Enthaltensein von P *in* K *wird mit* $P \trianglelefteq_f K$ *bezeichnet.*

5.2.2 Aktionsteilmethoden einer Transformationsregel

Ist die Anwendbarkeit einer Transformationsregel entsprechend der obigen Definitionen festgestellt (d.h. die Muster–DRS der TR ist in der zu transformierenden DRS enthalten), sollen bestimmte Aktionen ausgeführt werden, die zum Aufbau des Verhaltensschemas führen. Immer wieder auftretende Aktionen werden also zum Beispiel das Erstellen und Einfügen neuer Situationsschemata sowie das Einfügen von Prädiktions– und Spezialisierungsbeziehungen zwischen diesen Schemata sein.
Neben diesen Aktionen auf dem Verhaltensschema muß eine TR aber auch Aktionen auf der zu transformierenden DRS selbst ausführen können. Naheliegend ist beispielsweise das Löschen (oder zumindest das Markieren) derjenigen Bezugsträger und Bedingungen der DRS, welche der Muster–DRS entsprechen und somit zur Anwendbarkeit der TR geführt haben. Wäre dies nicht möglich, könnte nicht entschieden werden, ob eine Transformation dieses Teils der DRS schon stattgefunden hat. Die Transformation der DRS wäre niemals abgeschlossen. Aber auch das Einfügen neuer DRS–Bedingungen sollte von Seiten der TRn möglich sein, da auf diese Weise – ähnlich zu der Anwendung von Konstruktionsregeln auf Syntaxbäume – eine TR nach ihrer Anwendung Muster in einer DRS hinterlassen kann, welche die Anwendbarkeit einer anderen TR erst möglich machen.
Der Aktionsteil einer TR besteht somit aus Aktionsteilmethoden, die in einer Bibliothek bereitgestellt werden. Diese beinhaltet sowohl diejenigen Methoden, welche schon im Aktionsteil einer Konstruktionsregel zum Aufbau von DRSen Verwendung fanden, als auch neue Methoden, die der Veränderung einer bestehenden DRS und dem Aufbau eines Verhaltensschemas dienen.

5.2.3 Das Bedeutungs–Wörterbuch

Wie in Abschnitt 5.1.3 beschrieben, hängt die unterschiedliche Bedeutung einzelner DRS–Bedingungen nicht nur von der Struktur des DRS–Musters ab, in das sie eingebunden sind, sondern auch von den darin verwendeten Begriffen. Die Transformation des schemaspezifischen DRS–Prädikats `specialize()` äußert sich beispielsweise nicht darin, dieses Prädikat einfach in das zu erstellende Verhaltensschema zu übernehmen, sondern in Aktionen, die zwischen bestehenden Elementen des Schemas neue Verknüpfungen einfügen. Welche Aktionen dies bei der Transformation welches schemaspezifischen Prädikats sind, wird in einem Bedeutungs–Wörterbuch festgelegt. Der prinzipielle Aufbau dieses Wörterbuches ist in Abbildung 5.2 zu sehen.
Der Zugriff auf das Bedeutungs–Wörterbuch geschieht mittels spezieller TRn. Diese besitzen eine Muster–DRS, die lediglich aus genau einer Prädikats–Bedingung (bzw. Attributs–Bedingung usw.) besteht, deren Bezeichner nicht genauer spezifiziert wird (`vPREDICATE`*i*`(<`*parameter*`>)`). Dies führt dazu, daß TRn dieser Form auf die entsprechenden Bedingungen einer DRS stets anwendbar sind. Wird die Anwendbarkeit einer solchen TR festgestellt, folgt jedoch nicht die Ausführung von deren Aktionsteil. Es wird vielmehr ein der gefundenen Bedingung entsprechender Eintrag mit der gleichen

```
. . .
specialize(ERi,ERj) {
  [<Aktionsteilmethoden>]
}

concatenation(DRi) {
  of(DRi,PDRj) {
    [<Aktionsteilmethoden>]
  }
}
. . .
```

Abbildung 5.2: Schematische Darstellung eines Ausschnitts des Bedeutungs–Wörterbuches. Jeder Eintrag des Wörterbuches beginnt mit dem Prädikat, dessen Bedeutung definiert werden soll. Die Argumentliste des Prädikats kann hierbei auch Platzhalter beinhalten. Hierauf folgt dann entweder sofort eine Liste von Aktionsteilmethoden: die Bedeutung eines Prädikats ergibt sich dann durch die Ausführung dieser Methoden. In anderen Fällen folgt dem ersten Prädikat eine Liste von weiteren Prädikaten, welche die Grund–Bedeutung des Prädikats genauer eingrenzen.

Anzahl und Typisierung der Parameter im Bedeutungs–Wörterbuch gesucht. Ist ein solcher vorhanden, werden die dort verzeichneten Aktionsteilmethoden als Aktionsteil der TR aufgefaßt und ausgeführt. Fehlt ein solcher Eintrag, gilt der Aktionsteil der TR als leer.

5.3 Ein Algorithmus zur schrittweisen Transformation von DRSen

Mit Hilfe der obigen Definitionen und Vereinbarungen ist es nun möglich, einen Algorithmus zur schrittweisen Interpretation einer DRS zu formulieren (siehe Abbildung 5.3). Als Eingabe dient dem Algorithmus neben dem Bedeutungs–Wörterbuch und den TRn die zu transformierende DRS K und ein initiales Verhaltensschema V. Dieses Schema muß nicht notwendigerweise leer sein. Auch der iterative Aufbau eines Verhaltensschemas aus mehreren DRSen ist so möglich. Die Transformation einer DRS geschieht durch wiederholtes Testen der Anwendbarkeit jeder TR, wobei stets die durch $\mathcal{R}$ vorgegebene Ordnung der TRn eingehalten wird. Solange die Muster–DRS einer TR in K enthalten ist, wird deren Aktionsteil ausgeführt. Hierdurch werden sowohl K als

0 Eingabe:

- eine zu transformierende DRS K
- ein Tupel $\mathcal{R} = \langle TR_0, \ldots, TR_n \rangle$ von TRn mit Muster–DRSen $\langle P_0, \ldots, P_n \rangle$
- ein Bedeutungs–Wörterbuch W
- ein initiales Verhaltensschema V

1 Wiederhole:

- Für alle $i = 1, \ldots, n$:
  - Wiederhole:
    - Wenn $P_i \trianglelefteq_f K$, wende TR_i auf K an.

    Bis: $P_i \trianglelefteq_f K$ gilt nicht mehr.

Bis: Keine der TRn war anwendbar.

2 Ausgabe:

- das aktualisierte Verhaltensschema V.

Abbildung 5.3: Der Algorithmus zur schrittweisen Transformation einer DRS in ein Verhaltensschema. Nähere Erläuterung im Text.

auch V ggf. verändert. Ist die Muster–DRS einer TR nicht mehr in K enthalten, wird die nächste TR in $\mathcal{R}$ betrachtet. In dieser Weise wird $\mathcal{R}$ solange durchlaufen, bis bei einem Durchlauf keine der Regeln anwendbar war. Die Transformation der DRS gilt dann als beendet. Das Schema V zu diesem Zeitpunkt beinhaltet dann alle Informationen, die mit den bekannten TRn aus K erlangt werden konnten.

Kapitel 6

Implementierung eines DRS–Transformators

Im vorangehenden Kapitel wurde ein Verfahren zur Transformation einer DRS in ein Verhaltensschema in Form eines Situationsgraphenbaumes entwickelt. Das folgende Kapitel behandelt die Implementierung dieses Verfahrens. Ein erster Abschnitt beschäftigt sich eingehender mit den bei der Implementierung verfolgten Zielen. Der zweite Abschnitt stellt die daraus resultierende Klassenstruktur und die Funktion der einzelnen Klassen vor. Ein dritter Abschnitt schließt das Kapitel mit der beispielhaften Transformation einer einfachen Verhaltensbeschreibung ab.

6.1 Zielsetzung der Implementierung

Mit der Implementierung des im vorangehenden Kapitels beschriebenen Verfahrens zur DRS–Transformation wurde vor allem das Ziel verfolgt, die Anwendbarkeit des Verfahrens zu belegen. Hierbei stand im Vordergrund, den Weg von einer DRS durch die wiederholte Anwendung von Transformationsregeln bis zur Darstellung eines SGT in UMTL komplett zu beschreiten. Daneben spielten auch andere Aspekte, wie die Wiederverwendbarkeit einiger Klassen und Methoden für andere Aufgaben eine Rolle. Schließlich sollten die das Verfahren implementierenden Klassen und Methoden sich möglichst nahtlos in das bisherige Werkzeug zur DRS–Erstellung einfügen, um auf diese Weise die gesamte Verarbeitungskette von dem natürlichsprachlichen Text über die daraus erstellte DRS bis zur Darstellung eines SGT in UMTL in einem Programm-System zu vereinen. Die bei der Verfolgung der einzelnen oben genannten Ziele wichtigen Punkte sollen im folgenden vorgestellt werden.

6.1.1 Mustererkennung auf DRSen und TR–Anwendung

Die Anwendung einer Transformationsregel wird durch das Auffinden der Muster–DRS dieser TR in der zu transformierenden DRS ausgelöst. Entscheidend für diese *Mustererkennung* ist auf Seiten der Muster–DRS das Erkennen aller verwendeten *Platzhalter* für Bezugsträger und Bezeichner. Auf der Seite der zu transformierenden DRS müssen daraufhin alle möglichen Belegungen für jeden dieser Platzhalter gefunden werden. Dabei ist darauf zu achten, daß jeder Platzhalter durch seinen Namen implizit typisiert ist: Platzhalter für Individuen–Bezugsträger beispielsweise dürfen nur durch Individuen–Bezugsträger belegt werden. Nach der Bestimmung der Platzhalter und ihrer möglichen *jeweiligen* Belegungen muß eine *gemeinsame* Belegung gefunden werden, die der Definition des Enthaltenseins (siehe Definition 5.13) entspricht. Die Belegung zweier Platzhalter durch den gleichen Bezeichner wird dabei ausgeschlossen. Enthält beispielsweise das zu untersuchende Muster nur eine Prädikats–Bedingung der Form `vPREDICATE1(vDR1,vDR2)` ist die Menge der verwendeten Platzhalter gegeben durch {`vPREDCIATE1, vDR1, vDR2`}. Besitzt die zu transformierende DRS zum Zeitpunkt der Untersuchung drei Prädikatsbedingungen mit den Prädikatsbezeichnern {$\mathtt{p}_1$, $\mathtt{p}_2$, $\mathtt{p}_3$} und ebenfalls drei Individuenbezugsträger {`DR1, DR2, DR3`}, so sind in diesem Fall prinzipiell $3 * 3 * 2 = 18$ Belegungen der Platzhalter möglich (drei Belegungen des Prädikats, drei für den ersten und die zwei verbleibenden Belegungen für den zweiten Bezugsträger). Man erkennt, daß die Anzahl der möglichen Belegungen schnell sowohl mit der Anzahl der verwendeten Platzhalter als auch mit der Anzahl der möglichen jeweiligen Belegungen wächst. Die Suche nach derjenigen Belegung, welche die Muster–DRS in der zu transformierenden DRS enthalten sein läßt, wurde bisher durch einfaches Ausprobieren jeder möglichen Belegung gelöst. Nebenbedingungen zur Beschränkung der zu testenden Belegungen (etwa daß `vDR1` als erstes Argument einer Prädikatsbedingung auftreten muß oder daß für `vPREDICATE1` nur Bezeichner von zweistelligen Prädikatsbedingungen in Frage kommen), wurden hierbei nicht genutzt.
Wird eine Belegung im Sinne der Definition 5.13 gefunden, folgt die Ausführung des Aktionsteils der gerade betrachteten TR. Die gefundene Belegung aller Platzhalter steht bei dieser Ausführung weiterhin zur Verfügung, wodurch der Zugriff auf die *Werte* dieser Platzhalter auch in den einzelnen Aktionsteilmethoden gewährleistet ist [1].
Als Aktionsteilmethoden wurden wie in Kapitel 5.2.2 beschrieben zunächst all jene Methoden zugelassen, welche auch zur DRS–Erstellung im Aktionsteil von Konstruktionsregeln dienten. Dies ermöglicht es einer TR, neue Bezugsträger und Bedingungen in die zu transformierende DRS einzufügen. Zusätzlich hierzu wurden neue Methoden definiert, die auch das Entfernen von Bezugsträgern und Bedingungen einer DRS erlau-

[1] Bei der Erstellung von Aktionsteilmethoden hat es sich als sinnvoll erwiesen, diesen Methoden auch Verweise auf Unter–DRSen übergeben zu können. Um diese bereits in der Muster–DRS einer TR geeignet referenzieren zu können, wurde die Menge der Platzhalter um solche für Unter–DRSen erweitert, die mit `vSDRS`i (für SubDRS) mit $i \in \mathbb{N}$ bezeichnet werden. Diese Vorgehensweise dient jedoch lediglich der oben genannten Referenzierung und hat keinen Einfluß auf die Definition des Enthaltenseins einer Muster–DRS in einer zu transformierenden DRS.

ben. Hierdurch besitzen TRn die Möglichkeit, schon transformierte, d.h. abgearbeitete Teile einer DRS zu löschen. Weitere neue Aktionsteilmethoden dienen dem schrittweisen Aufbau eines Verhaltensschemas in Form eines UMTL–Programms. Ein Verzeichnis der neu erstellten Aktionsteilmethoden ist in Anhang C.4 zu finden.

6.1.2 Das Bedeutungs–Wörterbuch

Die Klassen zur Repräsentation des Bedeutungs–Wörterbuches wurden neu erstellt. Das Bedeutungs–Wörterbuch besteht aus einer Textdatei, die beim Start des Programms in eine interne Datenstruktur eingelesen wird. Jeder Eintrag des Wörterbuches besteht aus einem *Schlüsseleintrag*, unter dem er auch vom Programm aus gefunden werden soll. Ein solcher Schlüsseleintrag kann durch eine einstellige Namens– oder Attributsbeziehung bzw. eine zweistellige Relations– oder n-stellige ($n = 1, 2, \ldots$) Prädikatsbedingung gegeben sein. Der Bezeichner der Bedingung wird hierbei als konstant verstanden, Platzhalter können (und sollen) jedoch in den Parametern der Bedingungen vorkommen. Die weitere Struktur eines Eintrags ist die eines Baumes und wird durch eine öffnende geschweifte Klammer eingeleitet. Unterhalb des Schlüsseleintrages können beliebig viele *Untereinträge* folgen, die ihrerseits wieder Untereinträge besitzen können und so fort. Die Blätter dieses *Wörterbuch–Baumes* stellen schließlich *Aktionsblöcke* dar. Diese werden durch eine öffnende eckige Klammer eingeleitet. Hierauf folgt eine Liste von Aktionsteilmethoden–Aufrufen. Diesen Aufrufen können alle auf dem Pfad durch den Baum bis zu diesem Aktionsblock benuzten Platzhalter als Parameter dienen. Der Aktionsblock wird mit einer schließenden eckigen Klammer beendet. Das Ende eines Unter– bzw. Schlüsseleintrags wird mit einer schließenden geschweiften Klammer angezeigt.
Der Zugriff auf das Bedeutungs–Wörterbuch geschieht mittels der Aktionsteilmethode `callActions(<`*bedingung*`>)`. Der einzige Parameter dieser Methode ist durch eine Namens–, Attributs–, Relations– oder Prädikatsbedingung einer DRS gegeben. Beim Aufruf dieser Methode wird vom Programm zunächst ein Schlüsseleintrag zum Bezeichner der übergebenen Bedingung gesucht. Falls ein solcher vorhanden ist, wird zu jedem Blatt des unter diesem Schlüsseleintrag befindlichen Wörterbuch–Baumes eine Muster–DRS aus den auf dem jeweiligen Pfad definierten Bedingungen neu erstellt. Das Programm versucht daraufhin selbstständig, diese erzeugte Muster–DRS in der zu transformierenden DRS entsprechend der oben beschriebenen Mustererkennung zu finden. Die Aktionsteilmethoden desjenigen Blattes des Wörterbuch–Baumes, bei dem eine solche Mustererkennung erfolgreich verläuft, gelten als die *Bedeutung* der übergebenen Bedingung und werden von `callActions` zurückgegeben. Falls die Mustererkennung bei allen für Blätter erzeugten Muster–DRSen fehlschlägt, gilt die Bedingung als nicht bekannt: der Rückgabewert der Methode `callActions` ist in diesem Fall eine leere Liste von auszuführenden Aktionen. Gleiches gilt auch, wenn schon die Suche nach einem Schlüsseleintrag zu der übergebenen Bedingung im Wörterbuch fehlschlug.

6.1.3 Zielsprache und Wiederverwendbarkeit von Aktionsteilmethoden

Die Transformation einer DRS dient in der vorliegenden Arbeit der Erstellung von Verhaltensschemata in Form von Situationsgraphenbäumen. Zur Repräsentation dieser Schemata – und damit zur Festlegung der Zielsprache jeder das Verhaltensschema betreffenden Aktionsteilmethode – bieten sich zwei Möglichkeiten:

1. Datenstrukturen für SGTs: Im Verlauf der Arbeit wurden Klassen implementiert, welche die Struktur und den Inhalt von SGTs repräsentieren können (siehe Kapitel 4.1). Diese Klassen bieten die Möglichkeit, Situationsschemata nach und nach zu erstellen, durch Prädiktionskanten zu verbinden und zu Situationsgraphen zusammenzufassen.

2. Datenstrukturen für UMTL–Programme: Ebenfalls im Verlauf der Arbeit wurden Klassen erstellt, welche Struktur und Inhalt von UMTL–Programmen repräsentieren können. Da diese Klassen zunächst nur dem Einlesen von bestehenden UMTL–Programmen in Form von Textdateien dienten, ist ihr Funktionsumfang beschränkt auf das Hinzufügen kompletter UMTL–Regeln oder –Fakten.

Die erste der beiden Möglichkeiten bieten sich als Zielsprache für den schrittweisen Aufbau von Verhaltensschemata durch TRn an. Die Datenstruktur zur SGT–Repräsentation kann im Verlauf der Transformation sehr lokal geändert bzw. erweitert werden. Wie in Kapitel 2.2 beschrieben, geschieht die Anbindung von Verhaltensschemata an ein System zur Bildfolgenauswertung jedoch durch die Überführung eines Verhaltensschemas in ein äquivalentes UMTL–Programm. Dies spricht für die zweite Möglichkeit, als Zielsprache der Aktionsteilmethoden direkt UMTL–Programme zu wählen, da hierdurch eine anschliessende weitere Übersetzung entfällt. Ein weiteres Argument für diese Wahl der Zielsprache von Aktionsteilmethoden besteht darin, daß UMTL *allgemeiner* ist als die in UMTL ausdrückbaren Situationsgraphenbäume. Aktionsteilmethoden, welche UMTL–Regeln oder –Fakten erzeugen, wären in diesem Sinne eher für andere Einsatzfelder der DRS–Transformation wiederverwendbar als solche, deren Zielsprache von einer spezielleren Datenstruktur abhängt[2].
Mit der Entscheidung für UMTL als Zielsprache ändert sich auch die Vorgehensweise bei der DRS–Transformation: da die entsprechenden Klassen zur Repräsentation von UMTL–Programmen bisher keine Möglichkeit vorsehen, schon bestehende UMTL–Regeln zu verändern, können Informationen aus der DRS erst dann in UMTL–Regeln resultieren, wenn sowohl ihr Rumpf als auch ihr Kopf in der endgültigen Form bekannt ist. Eine Zwischenspeicherung von Informationen kann daher nur in der DRS selbst, nicht aber in dem zu erstellenden UMTL–Programm geschehen.

[2] An dieser Stelle sei auf [Arens & Ottlik 2000] verwiesen, deren Ziel es war, UMTL–Faktenlisten aus DRSen zu erzeugen, und somit als Zielsprache einer möglichen DRS–Transformation auch UMTL anstrebten.

6.1.4 Anbindung an das bisherige Werkzeug zur DRS–Erstellung

[Arens & Ottlik 2000] entwickelten ein Werkzeug, mit dem sich zu einer gegebenen Grammatik und zugehörigen Konstruktionsregel–Definitionen ein Werkzeug zur DRS–Erstellung generieren läßt. Die Anbindung der neu erstellten Klassen zur DRS–Transformation an ein auf diese Weise generiertes Werkzeug zur DRS–Erstellung gestaltet sich relativ einfach: die Benutzungs–Oberfläche des Werkzeuges wurde um Schaltflächen erweitert, durch deren Anwahl die Transformation einer erstellten DRS gestartet wird. Um etwaige Parameter des DRS–Transformators zu ändern (wie etwa den Dateinamen der einzulesenden Bedeutungs–Wörterbuchdatei), wurden weitere Eingabemasken definiert und in das System integriert. Diese Vorgehensweise ist jedoch lediglich als Notlösung anzusehen und soll im folgenden Abschnitt noch aus einer anderen Sicht betrachtet werden.

6.1.5 Generierung von DRS–Transformatoren

Wie oben beschrieben, hängt die Art der Erstellung von DRSen aus natürlichsprachlichen Texten von der in einer Textdatei definierten Grammatik und der zugehörigen Konstruktionsregeln ab. Durch diese *freie Wahl* der DRS–Erstellung kann von einem einzigen, für alle diese prinzipiell möglichen *DRS–Sorten* gültigen DRS–Transformator nicht gesprochen werden. Vielmehr sollte zu einem aus einer Grammatik–Definition generierten System zur DRS–Erstellung einer oder mehrere DRS–Transformatoren ebenfalls generiert werden können. Vorstellbar wäre in Anlehnung an den ersten Generierungsschritt wiederum die Generierung eines DRS–Transformators aus einer Textdatei heraus, welche die Definition aller verwendeten Transformationsregeln und deren Reihenfolge in einem geeigneten Format enthält. Dies hätte neben der Möglichkeit des schnellen Änderns des DRS–Transformators auch den Vorteil, daß die oben beschriebene (von Hand durchgeführte) Anbindung an das Werkzeug zur DRS–Erstellung automatisiert werden könnte. Die Möglichkeit einer solchen automatischen Anbindung, welche auch das nachträgliche Hinzufügen eines weiteren DRS–Transformators einschließt, wurde bei der Implementierung vorgesehen. Die automatische Generierung von DRS–Transformatoren aus Textdateien heraus wurde jedoch in der vorliegenden Arbeit aus Zeitgründen nicht angestrebt.

6.2 Beschreibung der Klassenstruktur

Die oben beschriebenen Ziele und Vorgehensweisen der Implementierung eines DRS–Transformators resultierten in einigen Java–Klassen, deren Aufgaben und Beziehungen zueinander hier kurz vorgestellt werden sollen. Die zentralen Klassen der Implementierung bilden die Klasse `DRSTransformer` bzw. `TransformationRule`. Diese stel-

len die Oberklasse aller Transformatoren bzw. Transformationsregeln dar. Während `DRSTransformer` die Steuerung des Transformationsprozeßes und die Anbindung an das Werkzeug zur DRS–Erstellung realisiert [3], sind in `TransformationRule` alle nötigen Methoden zur Muster–Erkennung und damit verbundenen Suche nach einer Belegung für die Platzhalter einer Muster–DRS implementiert. Die (bisher) einzige Unterklasse von `DRSTransformer` stellt die Klasse `DRS2SITConverter` dar. In ihr werden Attribute und Methoden definiert, die speziell für die Transformation einer DRS in ein Verhaltensschema in Form eines UMTL–Programms benötigt werden. Allgemeine Aktionsteilmethoden, welche nicht dem speziellen Zweck der Erstellung eines Verhaltensschemas dienen, sind dagegen in der Klasse `TransformationRule` definiert.
Wie jede Unterklasse von `DRSTransformer` besitzt auch `DRS2SITConverter` eine Liste von in diesem Tranformator benutzten Transformationsregeln. Jede Transformationsregel besitzt als Unterklasse von `TranformationRule` Methoden zum Test auf Anwendbarkeit sowie zur Anwendung selbst. Eine spezielle Unterklasse von `TransformationRule` stellt die Klasse `TR_Dictionary` dar, welche Methoden zum Zugriff auf das Bedeutungs–Wörterbuch realisiert. Das Bedeutungs–Wörterbuch selbst sowie die Methoden zum Einlesen eines solchen Wörterbuches repräsentieren die Klassen `TransformationDictionary` (das eigentliche Wörterbuch in Form einer Streuspeichertabelle (*Hashtable*)), `DictEntryNode` (die Repräsentation eines Wörterbuch–Eintrages mit Methoden zur Erstellung von Muster–DRSen aus Wörterbucheinträgen) sowie die Klasse `DictReader`, welche einen mit `javacc` erstellten Zerteiler für Wörterbuchdateien darstellt.
Zu Test– und Dokumentationszwecken wurden die Klassen `TransformationRule` sowie `DRSTransformer` um Methoden erweitert, welche die übersichtliche Darstellung von Transformations–Regeln bzw. die Abfolge eines Transformationsprozeßes in LaTeX ermöglichen. Eine Aufstellung der bisher erstellten und verwendeten Transformationsregeln findet sich in Anhang C.

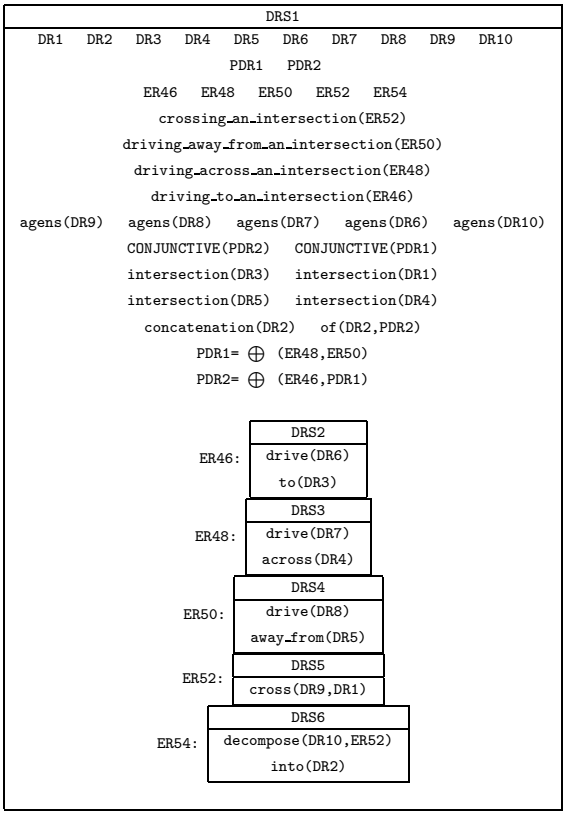
6.3 Beispielhafte DRS–Transformation

Die Transformation einer aus einer natürlichsprachlichen Verhaltensbeschreibung erstellten DRS in ein Verhaltenschema in Form eines UMTL–Programms mit Hilfe des implementierten DRS–Transformators soll nun abschließend an einem etwas komplexeren Beispiel beschrieben werden. Die Verhaltensbeschreibung, deren Transformation gezeigt werden soll, wurde in Anlehnung an den ersten Satz des Beispieltextes formuliert. Sie besagt, daß der Begriff des *Überfahrens einer Kreuzung* zeitlich genauer aufgelöst werden kann durch eine Verknüpfung von drei Begriffen: dem des *an eine Kreuzung Heranfahrens*, des *Kreuzung Überfahrens* und schließlich des *von einer Kreuzung Wegfahrens*. Der englische Text hierzu lautet:

[3] Jeder definierte DRS–Transformator stellt eine Unterklasse von `DRSTransformer` dar, und ist in dieser Oberklasse anzumelden. Beim Start des Gesamtsystems wird die Liste aller angemeldeten Transformatoren von Hauptprogramm ausgelesen und dem Benutzer zugänglich gemacht.

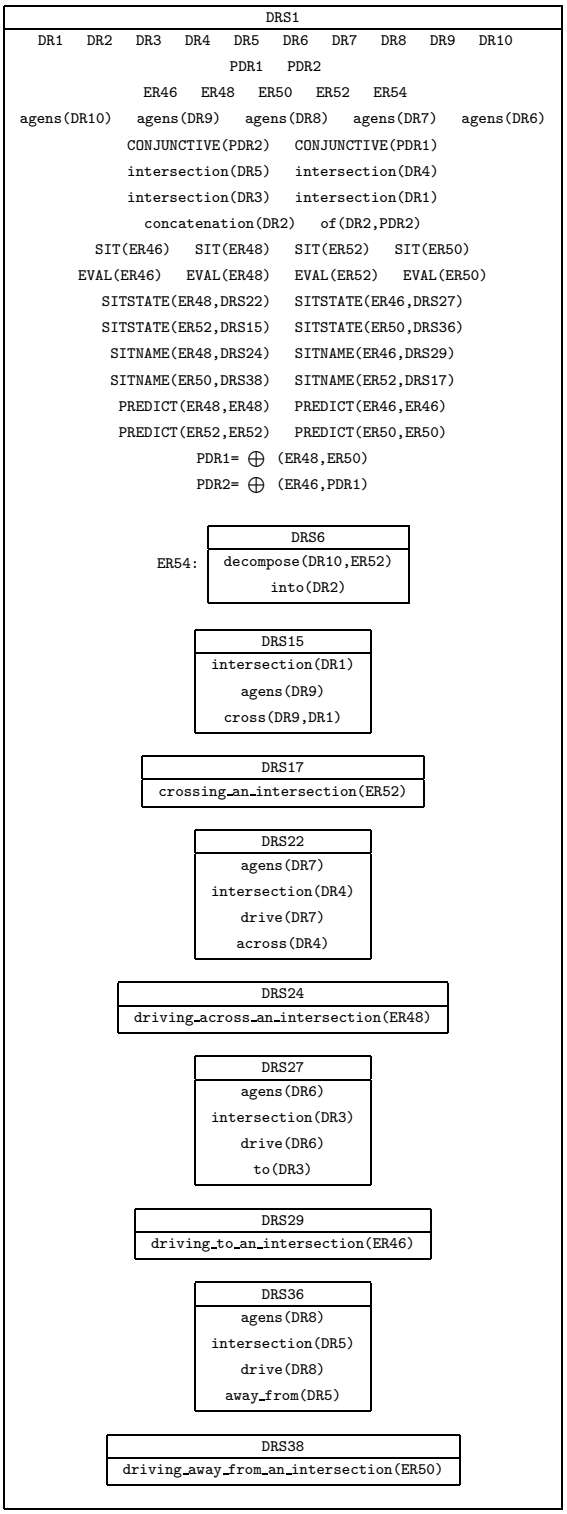
> *'crossing an intersection' is decomposed into a concatenation of 'driving to an intersection', 'driving across an intersection' and 'driving away from an intersection'.*

Dieser Text wurde gewählt, da er einerseits einfach genug ist, um die daraus erstellte DRS (mit einigen Nachbearbeitungen aus Platzgründen) noch darzustellen. Andererseits besitzt der Text genügend Komplexität, um einige wichtige Vorgehensweisen bei der DRS–Transformation vorstellen zu können. Ein Satz oder Abschnitt der in Anhang B.1 vorgestellten Beispielverhaltensbeschreibung wurde deshalb hier nicht verwendet, weil einige Probleme bei dessen Verarbeitung im Rahmen dieser Arbeit nicht geklärt werden konnten. Auf diese Probleme wird im letzten Kapitel noch näher eingegangen werden.
Die aus dem Beispieltext erstellte DRS hat die Form:



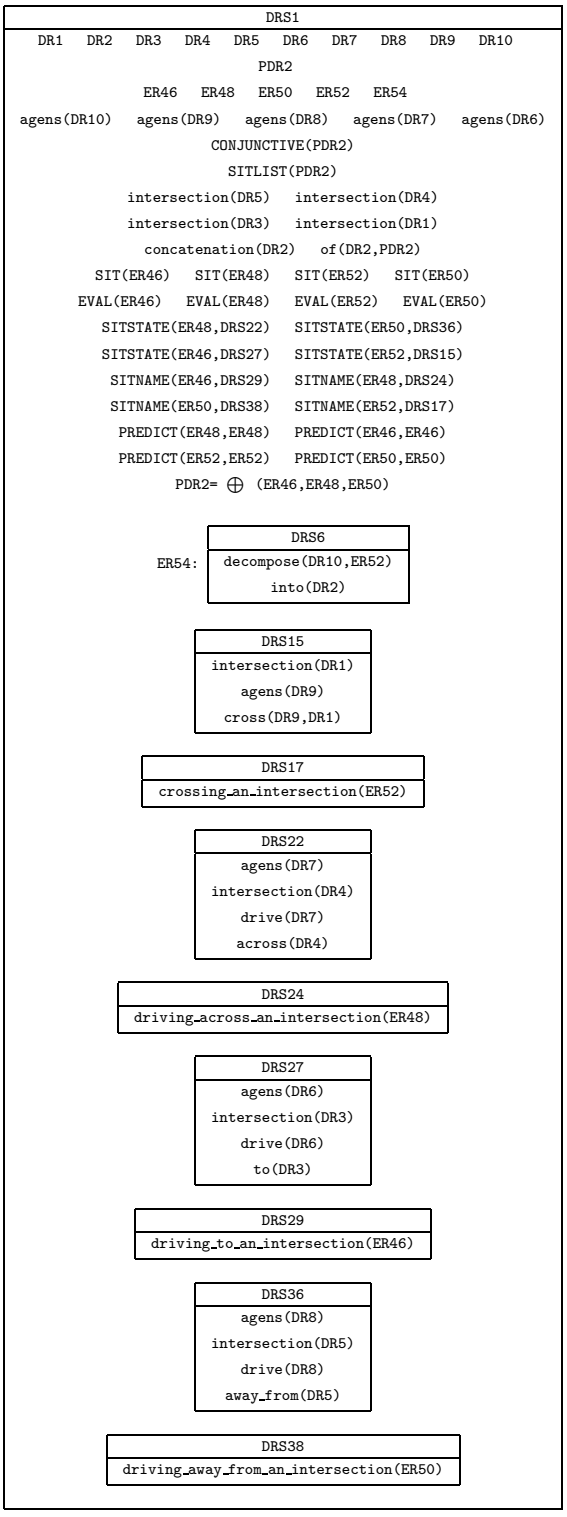
Die erste Aufgabe des Transformationsprozeßes besteht im Erkennen von Strukturen, welche zu Situationsschemata führen sollen. Dies sind in der obigen DRS alle jenen Ereignis–Bezugsträger, welche auch als Argument einer Namensbedingung auftreten (`ER46`, `ER48`, `ER50` und `ER52`). Um auf dieses Muster zu reagieren, wurde die TR `TR_SITMATCH1` (siehe Anhang C.2) definiert. Ist diese TR anwendbar, erzeugt sie zwei neue Unter–DRSen in der betrachteten DRS: in eine erste werden der Inhalt des Ereignis–Bezugsträgers sowie alle Attribute aus der übergeordneten DRS kopiert, welche Bezugsträger aus der Ereignis–Definition betreffen. In der zweiten Unter–DRS wird

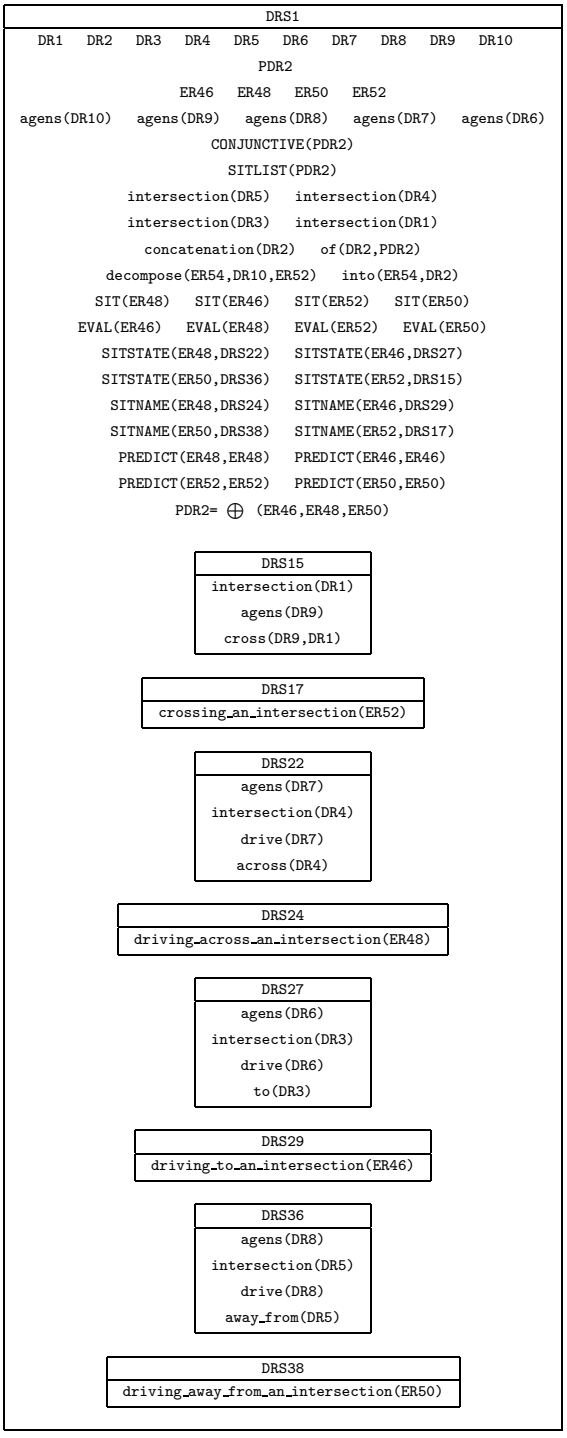
die Namensbedingung zur späteren Verarbeitung zwischengespeichert. Nach viermaliger Anwendung dieser TR hat die verbleibende DRS die Form:



Zusätzlich zu den neu eingeführten Unter–DRSen wurden in der obigen DRS auch einige neue Bedingungen eingeführt: Zum einem jeweils ein Attribut der Form `SIT(ER`i`)`, welches den jeweiligen Ereignis–Bezugsträger für die weitere Transformation als zu einem Situationsschema verarbeitbar markiert, zum anderen die Prädikate `SITSTATE(ER`i`, DRS`j`)` sowie `SITNAME(ER`i`,DRS`k`)`, welche den Bezug der neu eingeführten DRSen zu den Ereignis–Bezugsträgern, für welche sie eingeführt wurden, erhält. Die ebenfalls neuen Prädikate der Form `PREDICT(ER`i`,ER`i`)` betreffen die spätere Prädiktion im SGT: sie führen im weiteren Verlauf der Transformation dazu, daß jedes Situationsschema mit sich selbst durch eine Prädiktionskante verbunden wird. Eine ähnliche Aufgabe besitzen die ebenfalls neu eingeführten Attribute der Form `EVAL(ER`i`)`: sie werden im weiteren Verlauf der Transformation zur Erzeugung spezieller Evaluierungs–Regeln des UMTL–Programms führen.
Die nächsten Transformationsschritte betreffen die in der DRS definierten Plural–Bezugsträger sowie die noch verbliebene Ereignis–Definition. Im Beispieltext wurde beschrieben, daß ein Begriff durch die Verknüpfung dreier anderer Begriffe genauer aufgelöst wird. Durch die definierte Grammatik und die zugehörigen Konstruktionsregeln führt nun jede Menge mit mehr als zwei Elementen zu einer Menge von Pluralbezugsträgern, welche sich hierarchisch enthalten: bei drei Elementen bildet etwa ein `PDR` mit zwei Elementen die Basis, ein zweiter `PDR` enthält schließlich das dritte Element und den ersten `PDR`. Diese Verschachtelung muß von der DRS–Transformation zuerst aufgelöst werden, bevor andere Verweise auf diese Plural–Bezugsträger weiter verarbeitet werden dürfen. Genau dies ist die Aufgabe der TRn `TR_PDRCOLLECT1` bzw. `TR_PDRCOLLECT2`. Sie ersetzen sukzessive die verschachtelten `PDRen` durch einen einzigen, welcher dann alle ursprünglichen Elemente der im Text vorkommenden Menge beinhaltet. Eine weitere Aufgabe speziell dieser TRn besteht zusätzlich darin, zu prüfen, ob alle Elemente des `PDR` Ereignis–Bezugsträger sind, welche als Situationen erkannt wurden. Ist dies der Fall, wird ein neues Attribut der Form `SITLIST(PDR`i`)` in die DRS eingefügt, welches den `PDR` als Liste von Situationsschemata ausweist.
Die bisher noch verbliebene Ereignis–Bezugsträger–Definition enthält schema–spezifische Prädikate (`decompose` und `into`), welche Anweisungen zur späteren Spezialisierung eines Situationsschemas darstellen. Um diese Prädikate der weiteren Transformation leichter zugänglich zu machen, werden sie von fünf allein für diesen Zweck vorgesehenen TRn (`TR_PREDPOP1` bis `TR_PREDPOP5`) in die oberste Ebene der zu transformierenden DRS verschoben. Im vorliegenden Fall genügt hierzu jeweils eine Anwendung von `TR_PREDPOP3` (für `decompose`) sowie `TR_PREDPOP4` (für `into`). Als neues erstes Argument der Prädikats–Bedingungen wird bei diesem Verschieben der Ereignis–Bezugsträger eingefügt, aus welchem das Prädikat stammt. Hierdurch kann bei der weiteren Transformation festgestellt werden, welche Prädikate aus der gleichen Ereignis–Bezugsträger–Definition stammten. Dies ist wichtig, um beispielsweise aus gleichen Adverbien und Präpositionen erstellte Prädikate eindeutig einem aus einem Verb erzeugten Prädikat zuordnen zu können.
Die obige DRS nach einmaliger Anwendung von `TR_PDRCOLLECT1` ist auf der folgenden Seite zu sehen. Auf der darauf folgenden Seite befindet sich die Abbildung der

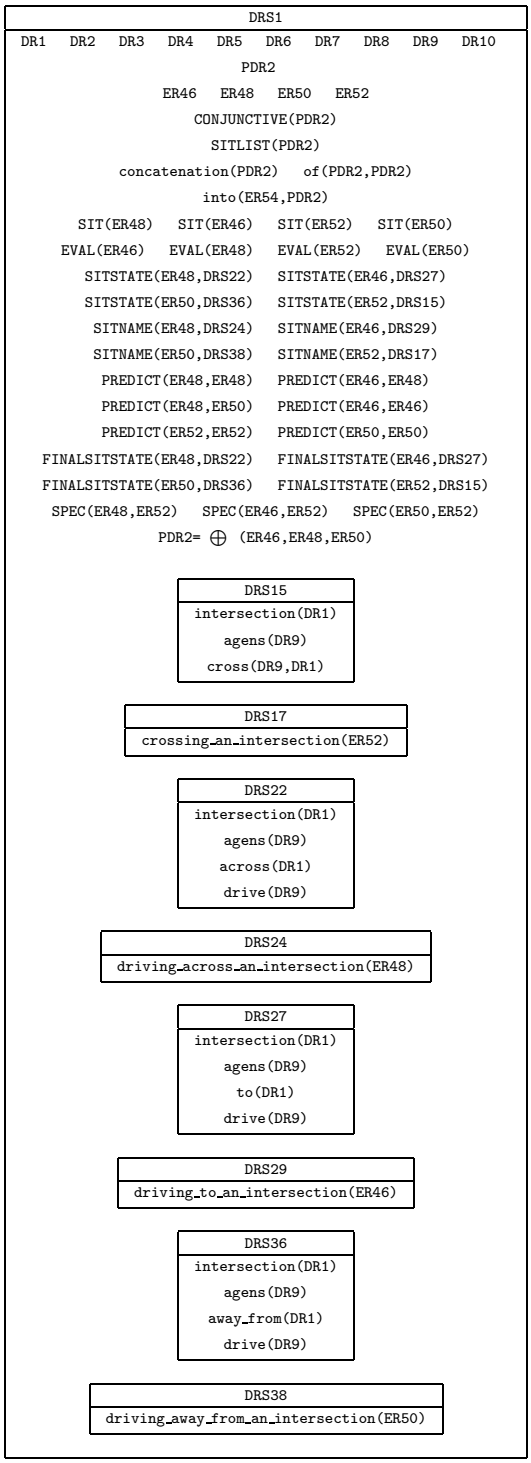
DRS, welche nach der Anwendung der beiden oben genannten TRn `TR_PREDPOP3` und `TR_PREDPOP4` verbleibt.





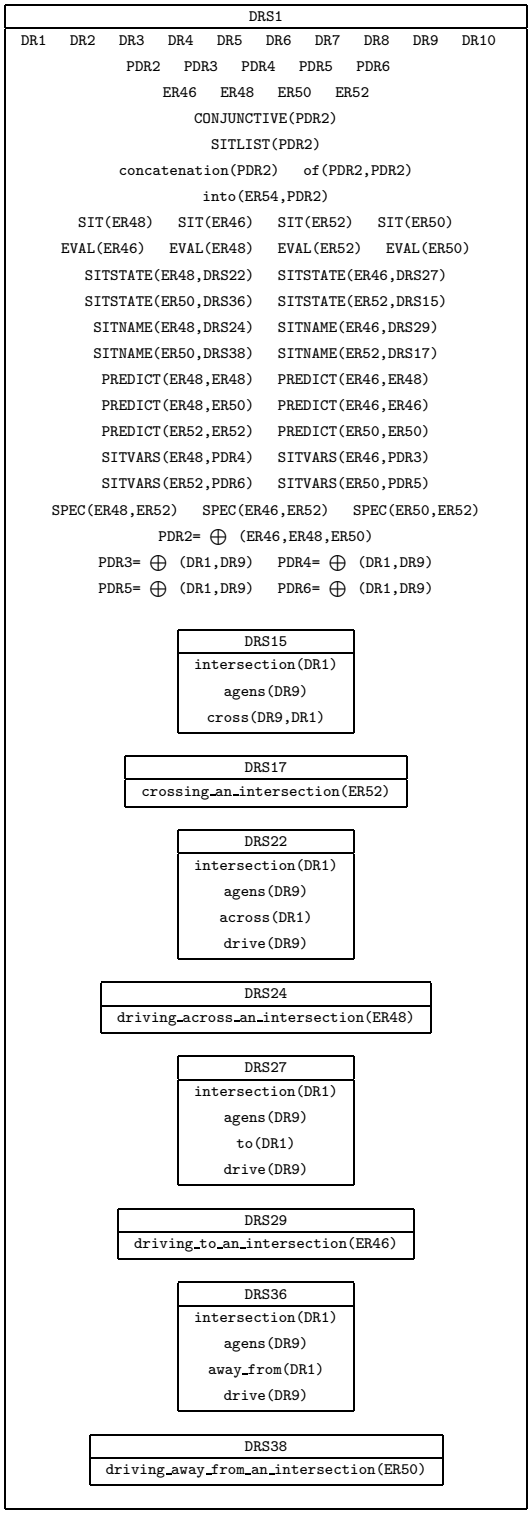
Die nächsten Schritte der Transformation betreffen die Bedeutung von schema–spezifischen Bedingungen. Dies sind in der obigen DRS einerseits die Bedingungen `concatenation(DR2)` und `of(DR2,PDR2)`: sie weisen `DR2` als die Verknüpfung aller Elemente von `PDR2` aus. Die andere Gruppe von schema–spezifischen Bedingungen besteht aus den bereits verschobenen Bedingungen `decompose(ER54,DR10,ER52)` und `into(ER54,DR2)`:

sie sollen zu einer Spezialisierung eines Situationsschemas führen. Zum Zugriff auf das Bedeutungs–Wörterbuch, welcher zur Transformation dieser Bedingungen nötig ist, wurden die TRn `TR_DICTPRED1` bis `TR_DICTPRED4` definiert. Diese besitzen in ihrer Muster–DRS lediglich genau eine Attributs– bzw. Prädikats–Bedingung und beziehen ihren Aktionsteil aus dem Bedeutungs–Wörterbuch.

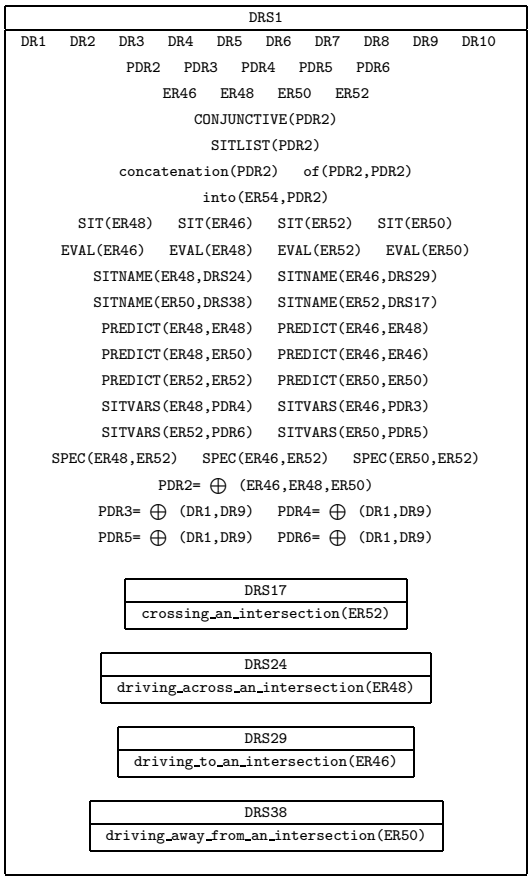


Das in der vorliegenden Arbeit verwendete Bedeutungs–Wörterbuch ist in Anhang C.3 zu finden. Die resultierende DRS nach mehrmaliger Anwendung von `TR_DICTPRED3` und einmaliger Anwendung von `TR_DICTPRED1` ist auf der vorangehenden Seite abgebildet. Die Transformation von `concatenation` (in Verbindung mit `of`) hat hierbei zunächst zur Ersetzung aller Vorkommen von `DR2` durch `PDR2` geführt. Die Bedingungen `decompose` (zusammen mit `into`) brachte einerseits neue Bedingungen der Form `PREDICT(ER`i`,ER`j`)` in die DRS ein: diese bereiten eine spätere Prädiktion zwischen Situationen entsprechend der Reihenfolge ihres Auftretens in dem beteiligten `PDR` vor. Entscheidend an dieser Stelle ist aber die Erzeugung von Bedingungen der Form `FINALSITSTATE(ER`i`,DRS`j`)` und `SPEC(ER`i`,ER`j`)` sowie die damit verbundenen Aktionen.
Bedingungen der Form `SPEC(ER`i`,ER`j`)` führen im weiteren Verlauf der Transformation zur Erzeugung derjenigen UMTL–Regeln, welche eine Spezialisierungs–Beziehung zwischen Situationsschemata ausdrücken. Mit der Einführung einer solchen Spezialisierungs–Beziehung ändern sich jedoch auch die Beziehungen zwischen Bezugsträgern der DRS: Während bei für sich stehenden Begriffen wie *eine Kreuzung überfahren* und *an eine Kreuzung heran fahren* nichts über die Beziehung zwischen den handelnden Akteuren der beiden Aktionen ausgesagt wird, impliziert eine Spezialisierung des einen Begriffes durch den anderen, daß es sich in beiden Fällen um *denselben* Akteur handelt. Gleiches gilt auch für die in den beiden Phrasen erwähnten *Kreuzungen*. Die Transformation von schema–spezifischen Bedingungen einer DRS, welche zu Spezialisierungsbeziehungen zwischen Situationsschemata führen, müssen daher einen Abgleich zwischen Bezugsträgern der beteiligten Situationsschemata durchführen. In der obigen DRS bedeutet dies die Übernahme des Bezugsträgers für den Handelnden (`agens`) und die Kreuzung (`intersection`) aus den die spezialisierte Situation repräsentierenden Bedingungen in alle die spezialisierenden Situationen repräsentierenden Bedingungen. Um dem weiteren Transformationsprozeß anzuzeigen, daß dieser Schritt erfolgreich abgeschlossen ist, wurden daraufhin die Bedingungen der Form `FINALSITSTATE(ER`i`,DRS`j`)` eingeführt.
Erwähnenswert ist an dieser Stelle noch, daß durch die wiederholte Anwendung von `TR_DICTPRED3` auch alle Bedingungen der Form `intersection(DR`i`)` sowie `agens(DR`j`)` aus der obersten Ebene der DRS gelöscht wurden. Dies liegt daran, daß diese Bedingungen die gleichen Form wie etwa `concatenation(DR`i`)` besitzen und somit der Muster–DRS dieser TR entsprechen. Das Bedeutungs–Wörterbuch besitzt zu diesen Bedingungen jedoch keinen Eintrag, weshalb sie – ohne weitere Aktionen auszulösen – gelöscht werden. Daß im Gegensatz hierzu die Bedingung `concatenation(PDR2)` nach Anwendung der TR in der DRS verbleibt, liegt an der durchgeführten Ersetzung aller Vorkommen von `DR2` durch `PDR2`: nach Anwendung der TR hat die eigentlich zu löschende Bedinung `concatenation(DR2)` eine andere Form und kann deshalb nicht mehr gefunden werden.
Der Transformationsprozeß ist nun fast soweit fortgeschritten, daß mit der Erzeugung aller nötigen UMTL–Regeln begonnen werden kann. Einzig die Bestimmung der Variablen, die in einem Situationsschema vorkommen sollen, ist noch durchzuführen. Hierzu

dient die TR `TR_SITVARS1`: Ihre Muster–DRS beinhaltet neben einer Unter–DRS vor allem die oben eingeführte Bedingung `FINALSITSTATE(ER`i`)`. Sie ist somit nun anwendbar. Die nach mehrfacher Anwendung dieser TR resultierende DRS lautet:



Der Aktionsteil von TR_SITVARS1 verursacht für jeden als Situation erkannten ER die Erzeugung eines PDR, welcher alle in dem Zustandsschema der Situation verwendeten Bezugsträger enthält. Daneben wird eine neue Bedingung der Form SITVARS(ERi,PDRj) erzeugt, welche den Bezug des ER zu dem neu engeführten PDR herstellt und für weitere Schritte zugänglich macht. Das Vorhandensein einer Bedingung SITVARS(ERi,PDRj) in Verbindung mit der zuvor eingeführten Bedingung der Form SITSTATE(ERi,DRSj) ermöglicht nun die Anwendung von TR_WRITESTATE1. Diese TR erzeugt sämtliche das Zustandsschema eines Situationsschemas betreffenden UMTL–Regeln. Anschließend wird die SITSTATE–Bedingung aus der DRS entfernt. Die wiederum mehrfache Anwendung dieser TR hinterläßt die DRS:

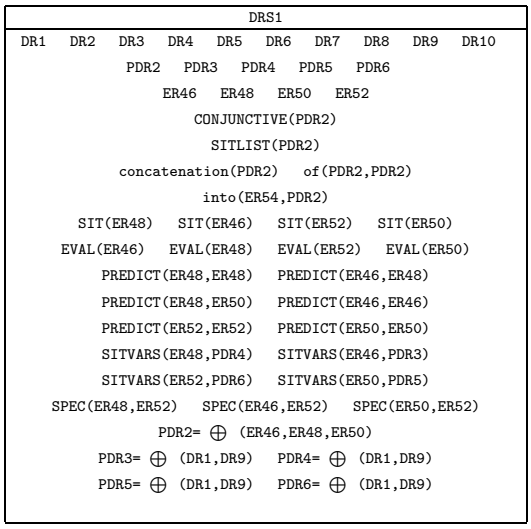


Diejenigen Unter–DRSen, welche die für das Zustandsschema einer Situation benötigten Bedingungen enthielten, wurden im obigen Abbild der DRS aus Platzgründen von Hand gelöscht. Die TR selbst beläßt diese Unter–DRSen in der zu transformierenden DRS, um sie einer möglichen späteren Verwendung nicht zu entziehen. Die entsprechenden DRSen werden jedoch im vorliegenden Fall nicht mehr benötigt. Das zu erstellende UMTL–Programm umfaßt nach der mehrfachen Anwendung von TR_WRITESTATE1 die Regeln:

```
-i : +i !  (state_ER44(DR1,DR9) :- (to(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER44(DR1,DR9) :- (true)).
```

```
-i : +i !  (state_ER46(DR1,DR9) :- (across(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER46(DR1,DR9) :- (true)).
-i : +i !  (state_ER48(DR1,DR9) :- (away_from(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER48(DR1,DR9) :- (true)).
-i : +i !  (state_ER50(DR1,DR9) :- (cross(DR9,DR1),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER50(DR1,DR9) :- (true)).
```

Die nächste TR (`TR_WRITEACTION1`) schreibt diejenigen UMTL–Regeln, welche das Handlungsschema von Situationsschemata definieren. Als Handlungsschema wurde in der vorliegenden Arbeit lediglich die Anweisung zur Ausgabe des Namens des Situationsschemas betrachtet. Entsprechend beinhaltet die Muster–DRS von `TR_WRITEACTION1` neben der Bedingung `SITVARS(ER`i`,PDR`j`)` auch die Bedingung `SITNAME(ER`i`,DRS`j`)`. Die resultierende DRS nach mehrfacher Anwendung dieser TR lautet (wiederum wurden die ab jetzt überflüssigen Unter–DRSen von Hand gelöscht):

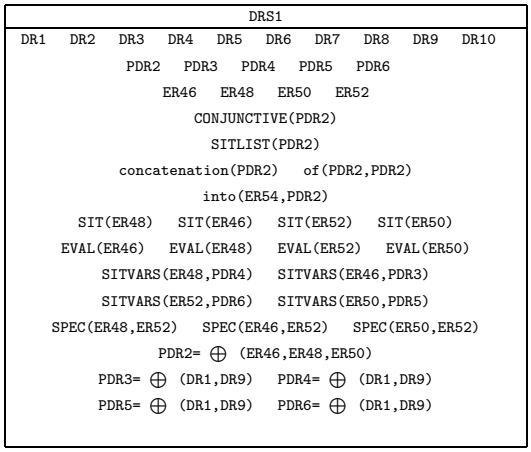


Das zu erstellende UMTL–Programm wurde durch die Anwendung der TR erweitert und lautet nun:

```
-i : +i !  (state_ER44(DR1,DR9) :- (to(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER44(DR1,DR9) :- (true)).
-i : +i !  (state_ER46(DR1,DR9) :- (across(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER46(DR1,DR9) :- (true)).
-i : +i !  (state_ER48(DR1,DR9) :- (away_from(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER48(DR1,DR9) :- (true)).
-i : +i !  (state_ER50(DR1,DR9) :- (cross(DR9,DR1),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER50(DR1,DR9) :- (true)).
-i : +i !  (actn_ER44(DR1,DR9) :- note(driving_to_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER46(DR1,DR9) :- note(driving_across_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER48(DR1,DR9) :- note(driving_away_from_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER50(DR1,DR9) :- note(crossing_an_intersection(DR1,DR9))).
```

Der nächste Schritt zur Transformation der vorliegenden DRS besteht in der wiederholten Anwendung der TRn `TR_WRITEPREDICT1` bzw. `TR_WRITEPREDICT2`. Diese TRn schreiben jene Regeln des zu erstellenden UMTL–Programms, welche die Prädiktionsbeziehungen zwischen Situationsschemata ausdrücken. Hierfür sind durch die Implementierung der Muster–Erkennung auf DRSen zwei TRn notwendig: eine, welche in

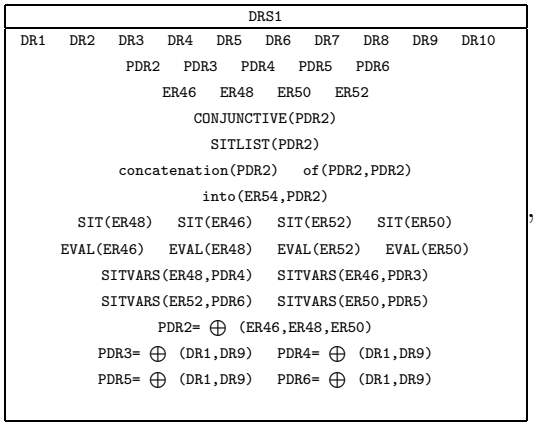
ihrer Muster–DRS eine Bedingung der Form `PREDICT(ERi,ERi)` besitzt und entsprechend Prädiktionen von Situationsschemata auf sich selbst realisiert sowie eine weitere TR mit einer Bedingung der Form `PREDICT(ERi,ERj)`, um Prädiktionsbeziehungen zwischen verschiedenen Situationsschemata zu erreichen. Durch wiederholte Anwendung der TRn werden alle vorhandenen Bedingungen dieser Art abgearbeitet und die entsprechenden UMTL–Regeln erzeugt. Die resultierende DRS lautet dann:



Das erstellte UMTL–Programm nach dieser TR–Anwendung hat die Form:

```
-i : +i !  (state_ER44(DR1,DR9) :- (to(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER44(DR1,DR9) :- (true)).
-i : +i !  (state_ER46(DR1,DR9) :- (across(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER46(DR1,DR9) :- (true)).
-i : +i !  (state_ER48(DR1,DR9) :- (away_from(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER48(DR1,DR9) :- (true)).
-i : +i !  (state_ER50(DR1,DR9) :- (cross(DR9,DR1),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER50(DR1,DR9) :- (true)).
-i : +i !  (actn_ER44(DR1,DR9) :- note(driving_to_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER46(DR1,DR9) :- note(driving_across_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER48(DR1,DR9) :- note(driving_away_from_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER50(DR1,DR9) :- note(crossing_an_intersection(DR1,DR9))).
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER44(DR1,DR9),!,eval_ER44(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER48(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER50(DR1,DR9) :- (+1 : +1 !  (fstate_ER50(DR1,DR9),!,eval_ER50(DR1,DR9)))).
```

Der vorletzte Schritt der Transformation besteht in der Erstellung der UMTL–Regeln, welche die Spezialisierungsbeziehungen zwischen Situationsschemata realisieren. Die entsprechenden Bedingungen der DRS der Form `SPEC(ERi,ERj)`, welche diesen Zusammenhang ausdrücken, wurden in vorangegangenen Transformationsschritten erzeugt. Diese Bedingungen führen nun zur Anwendung der TR `TR_WRITESPEC1`, welche diese als Teil ihrer Muster–DRS besitzt. Nach mehrfacher Anwendung dieser TR gelangt man zu der DRS:



sowie zu dem erweiterten UMTL–Programm:

```
-i : +i !  (state_ER44(DR1,DR9) :- (to(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER44(DR1,DR9) :- (true)).
-i : +i !  (state_ER46(DR1,DR9) :- (across(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER46(DR1,DR9) :- (true)).
-i : +i !  (state_ER48(DR1,DR9) :- (away_from(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER48(DR1,DR9) :- (true)).
-i : +i !  (state_ER50(DR1,DR9) :- (cross(DR9,DR1),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER50(DR1,DR9) :- (true)).
-i : +i !  (actn_ER44(DR1,DR9) :- note(driving_to_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER46(DR1,DR9) :- note(driving_across_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER48(DR1,DR9) :- note(driving_away_from_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER50(DR1,DR9) :- note(crossing_an_intersection(DR1,DR9))).
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER44(DR1,DR9),!,eval_ER44(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER48(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER50(DR1,DR9) :- (+1 : +1 !  (fstate_ER50(DR1,DR9),!,eval_ER50(DR1,DR9)))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER44(DR1,DR9),!,eval_ER44(DR1,DR9))).
-i : +i !  (fstate_ER44(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER44(DR1,DR9))).
-i : +i !  (facti_ER44(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER44(DR1,DR9))).
-i : +i !  (fact_ER44(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER44(DR1,DR9),acti_ER44(DR1,DR9))).
-i : +i !  (pred_ER44(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER46(DR1,DR9),!,eval_ER46(DR1,DR9))).
-i : +i !  (fstate_ER46(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER46(DR1,DR9))).
-i : +i !  (facti_ER46(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER46(DR1,DR9))).
-i : +i !  (fact_ER46(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER46(DR1,DR9),acti_ER46(DR1,DR9))).
-i : +i !  (pred_ER46(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER48(DR1,DR9),!,eval_ER48(DR1,DR9))).
-i : +i !  (fstate_ER48(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER48(DR1,DR9))).
-i : +i !  (facti_ER48(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER48(DR1,DR9))).
-i : +i !  (fact_ER48(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER48(DR1,DR9),acti_ER48(DR1,DR9))).
-i : +i !  (pred_ER48(DR1,DR9) :- (pred_ER50(DR1,DR9))).
```

Der letzte Transformationsschritt besteht nun in der Abarbeitung der bereits im ersten Transformationsschritt eingeführten Bedingungen der Form `EVAL(ER`i`)`. Diese lassen nun die TR `TR_WRITEEVAL1` anwendbar werden, welche spezielle Evaluierungs–Regeln des UMTL–Programms schreibt. Diese Regeln führen bei einer späteren Traversierung des durch das UMTL–Programm formulierten SGT dazu, daß eine ggf. fehlgeschlagene Ausprägung einer spezielleren Situation durch eine Prädiktion der schon ausgeprägten Situation führt. Durch die Funktion dieser Regeln – die gerade bei einem Fehlschlag anderer Regeln zum Tragen kommt – konnten sie nicht schon früher in das UMTL–Programm geschrieben werden.
Die Transformation der DRS ist mit der wiederholten Anwendung der TR `TR_WRITE-EVAL1` abgeschlossen. Das resultierende UMTL–Programm an diesem Punkt lautet:

```
-i : +i !  (state_ER44(DR1,DR9) :- (to(DR1),drive(DR9),intersection(DR1),agens(DR9))).
```

```
-i : +i !  (acti_ER44(DR1,DR9) :- (true)).
-i : +i !  (state_ER46(DR1,DR9) :- (across(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER46(DR1,DR9) :- (true)).
-i : +i !  (state_ER48(DR1,DR9) :- (away_from(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER48(DR1,DR9) :- (true)).
-i : +i !  (state_ER50(DR1,DR9) :- (cross(DR9,DR1),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER50(DR1,DR9) :- (true)).
-i : +i !  (actn_ER44(DR1,DR9) :- note(driving_to_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER46(DR1,DR9) :- note(driving_across_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER48(DR1,DR9) :- note(driving_away_from_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER50(DR1,DR9) :- note(crossing_an_intersection(DR1,DR9))).
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER44(DR1,DR9),!,eval_ER44(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER48(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER50(DR1,DR9) :- (+1 : +1 !  (fstate_ER50(DR1,DR9),!,eval_ER50(DR1,DR9)))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER44(DR1,DR9),!,eval_ER44(DR1,DR9))).
-i : +i !  (fstate_ER44(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER44(DR1,DR9))).
-i : +i !  (facti_ER44(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER44(DR1,DR9))).
-i : +i !  (fact_ER44(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER44(DR1,DR9),acti_ER44(DR1,DR9))).
-i : +i !  (pred_ER44(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER46(DR1,DR9),!,eval_ER46(DR1,DR9))).
-i : +i !  (fstate_ER46(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER46(DR1,DR9))).
-i : +i !  (facti_ER46(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER46(DR1,DR9))).
-i : +i !  (fact_ER46(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER46(DR1,DR9),acti_ER46(DR1,DR9))).
-i : +i !  (pred_ER46(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER48(DR1,DR9),!,eval_ER48(DR1,DR9))).
-i : +i !  (fstate_ER48(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER48(DR1,DR9))).
-i : +i !  (facti_ER48(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER48(DR1,DR9))).
-i : +i !  (fact_ER48(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER48(DR1,DR9),acti_ER48(DR1,DR9))).
-i : +i !  (pred_ER48(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER44(DR1,DR9) :- (fact_ER44(DR1,DR9),!,pred_ER44(DR1,DR9))).
-i : +i !  (eval_ER46(DR1,DR9) :- (fact_ER46(DR1,DR9),!,pred_ER46(DR1,DR9))).
-i : +i !  (eval_ER48(DR1,DR9) :- (fact_ER48(DR1,DR9),!,pred_ER48(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (fact_ER50(DR1,DR9),!,pred_ER50(DR1,DR9))).
```

6.4 Ergebnis der Transformation

Das obige UMTL–Programm beinhaltet nun fast alle Regeln, welche die Beziehungen zwischen Situationsschemata in dem zu erstellen Verhaltensschema repräsentieren. Um das erstellte Programm jedoch einer Situationsanalyse durch das Logik–System *F–Limette* zugänglich zu machen, sind noch einige zusätzliche Regeln nötig, welche jedoch auch von dem Transformations–Algorithmus in der Klasse `DRS2SITConverter` automatisch vorgenommen werden. Zum einen müssen alle globalen Situationsschemata (dies sind solchen Schemata, die nicht schon eine Spezialisierung eines anderen Schemas darstellen), mit einer zentralen, die Traversierung des SGT startenden Regel verknüpft werden. Dies geschieht durch jeweils eine Regel, welche die Traversierung (`traversal`) aus dem Prädikate `run_traversal_ER`*i* folgert. Das Prädikt `run_traversal_ER`*i* für jedes dieser globalen Situationsschemata wird aus der Ausprägbarkeit diese Schemas gefolgert. Alle weiteren noch anzufügenden Regeln in diesem Schritt betreffen die Tatsache, daß globale Situationsschemata gerade kein anderes Situationsschema spezialisieren, und deshalb ihr Zustands– und Aktionsschema auch nicht von den Schemata eines solchen abhängen kann. Das endgültige Resultat der Transformation lautet somit:

```
-i : +i !  (state_ER44(DR1,DR9) :- (to(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER44(DR1,DR9) :- (true)).
-i : +i !  (state_ER46(DR1,DR9) :- (across(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER46(DR1,DR9) :- (true)).
-i : +i !  (state_ER48(DR1,DR9) :- (away_from(DR1),drive(DR9),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER48(DR1,DR9) :- (true)).
-i : +i !  (state_ER50(DR1,DR9) :- (cross(DR9,DR1),intersection(DR1),agens(DR9))).
-i : +i !  (acti_ER50(DR1,DR9) :- (true)).
-i : +i !  (actn_ER44(DR1,DR9) :- note(driving_to_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER46(DR1,DR9) :- note(driving_across_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER48(DR1,DR9) :- note(driving_away_from_an_intersection(DR1,DR9))).
-i : +i !  (actn_ER50(DR1,DR9) :- note(crossing_an_intersection(DR1,DR9))).
```

```
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER44(DR1,DR9) :- (+1 : +1 !  (fstate_ER44(DR1,DR9),!,eval_ER44(DR1,DR9)))).
-i : +i !  (pred_ER46(DR1,DR9) :- (+1 : +1 !  (fstate_ER46(DR1,DR9),!,eval_ER46(DR1,DR9)))).
-i : +i !  (pred_ER48(DR1,DR9) :- (+1 : +1 !  (fstate_ER48(DR1,DR9),!,eval_ER48(DR1,DR9)))).
-i : +i !  (pred_ER50(DR1,DR9) :- (+1 : +1 !  (fstate_ER50(DR1,DR9),!,eval_ER50(DR1,DR9)))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER44(DR1,DR9),!,eval_ER44(DR1,DR9))).
-i : +i !  (fstate_ER44(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER44(DR1,DR9))).
-i : +i !  (facti_ER44(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER44(DR1,DR9))).
-i : +i !  (fact_ER44(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER44(DR1,DR9),acti_ER44(DR1,DR9))).
-i : +i !  (pred_ER44(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER46(DR1,DR9),!,eval_ER46(DR1,DR9))).
-i : +i !  (fstate_ER46(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER46(DR1,DR9))).
-i : +i !  (facti_ER46(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER46(DR1,DR9))).
-i : +i !  (fact_ER46(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER46(DR1,DR9),acti_ER46(DR1,DR9))).
-i : +i !  (pred_ER46(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (state_ER48(DR1,DR9),!,eval_ER48(DR1,DR9))).
-i : +i !  (fstate_ER48(DR1,DR9) :- (fstate_ER50(DR1,DR9),!,state_ER48(DR1,DR9))).
-i : +i !  (facti_ER48(DR1,DR9) :- (facti_ER50(DR1,DR9),acti_ER48(DR1,DR9))).
-i : +i !  (fact_ER48(DR1,DR9) :- (facti_ER50(DR1,DR9),actn_ER48(DR1,DR9),acti_ER48(DR1,DR9))).
-i : +i !  (pred_ER48(DR1,DR9) :- (pred_ER50(DR1,DR9))).
-i : +i !  (eval_ER44(DR1,DR9) :- (fact_ER44(DR1,DR9),!,pred_ER44(DR1,DR9))).
-i : +i !  (eval_ER46(DR1,DR9) :- (fact_ER46(DR1,DR9),!,pred_ER46(DR1,DR9))).
-i : +i !  (eval_ER48(DR1,DR9) :- (fact_ER48(DR1,DR9),!,pred_ER48(DR1,DR9))).
-i : +i !  (eval_ER50(DR1,DR9) :- (fact_ER50(DR1,DR9),!,pred_ER50(DR1,DR9))).
-i : +i !  (pred_ER50(DR1,DR9) :- (fail)).
-i : +i !  (fstate_ER50(DR1,DR9) :- (state_ER50(DR1,DR9))).
-i : +i !  (facti_ER50(DR1,DR9) :- (acti_ER50(DR1,DR9))).
-i : +i !  (fact_ER50(DR1,DR9) :- (acti_ER50(DR1,DR9),actn_ER50(DR1,DR9))).
-i : +i !  (run_traversal_ER50(DR1,DR9) :- (state_ER50(DR1,DR9),!,eval_ER50(DR1,DR9))).
-i : +i !  (traversal :- (run_traversal_ER50(DR1,DR9))).
```

Die Transformation ist damit abgeschlossen.

6.5 Ablauferkennung mit dem erstellten Verhaltensschema

Abschließend soll mit dem oben erstellten UMTL–Programm eine beispielhafte Ablauferkennung mittels *F–Limette* durchgeführt werden. Als Grundlage der Ablauferkennung dient eine von Hand erstellte UMTL–Faktenliste, welche die Auswertungsergebnisse einer vorangehenden Bildfolgenauswertung *simuliert* [4]. Diese Faktenliste hat die Form:

```
-i : +i ! agens(a).
-i : +i ! intersection(b).

 2 :  5 ! to(b).
 6 : 10 ! across(b).
10 : 13 ! away_from(b).

 0 : 20 ! cross(a,b).
 0 : 20 ! drive(a).
```

Die Faktenliste besagt, daß – bezogen auf eine beliebige Zeitskala – die Prädikate `agens(a)` und `intersection(b)` *immer* (`-i : +i`) erfüllt sind. Diese Prädikate beschreiben feststehende Eigenschaften der beiden Individuen `a` und `b`. Dagegen gelten alle weiteren Prädikate nur für einen bestimmten Zeitraum (`to(b)` etwa von Zeitpunkt

[4] Die in der verwendeten Faktenliste benutzten Begriffe stellen bereits Abstraktionen der von dem Bildfolgenausertungssystem gelieferten Begriffe dar. Auf den Zusammenhang zwischen Bildfolgenauswertungsergebnissen und den im Verhaltensschema benutzten Begriffen wird im folgenden Kapitel noch eingegangen werden.

2 bis Zeitpunkt 5). Diese Prädikate stellen daher zeitlich begrenzte Eigenschaften von bzw. Beziehungen zwischen Individuen dar.
Die Ablauferkennung auf dieser Faktenliste mit dem oben erstellten Verhaltensschema wird durch den Aufruf von *F–Limette* der Form:

```
limette3 sit data scheme starttra
```

gestartet. `limette3` bezeichnet den Aufruf des Logik–Programms, `sit`, `data`, `scheme` und `starttra` bezeichnen die dem Logik–Programm übergebenen UMTL–Programme. `sit` enthält dabei einige Definitionen, welche beispielsweise zur Ausgabe von Prädikaten verwendet werden. `data` ist der Name der obigen UMTL–Faktenliste, `scheme` bezeichnet das im vorangehenden Abschnitt erstellte Verhaltensschema in Form eines UMTL–Programms. `starttra` schließlich enthält die UMTL–Anfrage, welche die Ablauferkennung startet gefolgt von der Anweisung zur Beendigung von *F–Limette* und hat die Form:

```
?- traversal.
?- quit.
```

Das Ergebnis der durchgeführten Ablauferkennung hat die Form:

```
0 : 0 ! crossing_an_intersection(b,a).
1 : 1 ! crossing_an_intersection(b,a).
2 : 2 ! driving_to_an_intersection(b,a).
3 : 3 ! driving_to_an_intersection(b,a).
4 : 4 ! driving_to_an_intersection(b,a).
5 : 5 ! driving_to_an_intersection(b,a).
6 : 6 ! driving_across_an_intersection(b,a).
7 : 7 ! driving_across_an_intersection(b,a).
8 : 8 ! driving_across_an_intersection(b,a).
9 : 9 ! driving_across_an_intersection(b,a).
10 : 10 ! driving_away_from_an_intersection(b,a).
11 : 11 ! driving_away_from_an_intersection(b,a).
12 : 12 ! driving_away_from_an_intersection(b,a).
13 : 13 ! driving_away_from_an_intersection(b,a).
14 : 14 ! crossing_an_intersection(b,a).
15 : 15 ! crossing_an_intersection(b,a).
16 : 16 ! crossing_an_intersection(b,a).
17 : 17 ! crossing_an_intersection(b,a).
18 : 18 ! crossing_an_intersection(b,a).
19 : 19 ! crossing_an_intersection(b,a).
20 : 20 ! crossing_an_intersection(b,a).
```

Man erkennt, daß für jeden Zeitpunkt im Intervall von 0 bis 20 ein Situationsschema des Verhaltenschemas ausgeprägt und dessen Name ausgegeben wurde. Während zunächst (von 0 bis 1) nur das allgemeinste Schema `crossing_an_intersection` ausprägbar war, führte daraufhin die Gültigkeit des Prädikats `to(b)` in der obigen Faktenliste im Zeitraum von 2 bis 5 zur Ausprägung des spezielleren Schemas `driving_to_an_intersection`. In dieser Weise fuhr die Ablauferkennung fort bis schließlich wiederum nur das allgemeinste Schema ausprägbar war. Die Ablauferkennung endet im Zeitpunkt 20, ab dem selbst die Ausprägung des allgemeinsten Schemas nicht mehr möglich ist.

Kapitel 7

Zusammenfassung

7.1 Erreichte Ergebnisse

Ziel dieser Arbeit war es zu untersuchen, inwieweit sich natürlichsprachliche Texte zur Bereitstellung von Wissen über das Verhalten von beobachteten Akteuren in der Bildfolgenauswertung eignen und wie dieses Wissen automatisch in eine durch das Bildfolgenauswertungssystem verarbeitbare Form gebracht werden kann.
Aufbauend auf bisherigen Arbeiten (vgl. [Arens & Ottlik 2000]) am Institut füt Algorithmen und Kognitive Systeme der Universität Karlsruhe (TH) wurde zunächst ein Werkzeug erstellt, mit dem natürlichsprachliche Verhaltensbeschreibungen entsprechend der Diskursrepräsentationstheorie von [Kamp & Reyle 93] in DRSen überführt werden können. Hierzu mußte eine Grammatik und die zugehörigen Konstruktionsregeln erstellt werden.
In einem weiteren Schritt wurden Java–Klassen entwickelt und implementiert, welche die bisher am Institut verwendeten Verhaltensschemata in Form von Situationsgraphenbäumen sowie in Form von UMTL–Programmen repräsentieren können. Im Zusammenhang mit dieser Implementierung wurde untersucht, auf welche Weise dem Benutzer bzw. Entwickler dieser Verhaltensschemata geeignete Werkzeuge zur grafischen Inspektion der Schemata bereitgestellt werden können. Diese Untersuchungen konnten aus Zeitgründen nicht weiter vertieft werden. Es war jedoch möglich, eine automatische Generierung von HTML–Dokumenten aus SGTs zu realisieren, welche schon im Verlauf der vorliegenden Arbeit die Inspektion der erstellten Schemata sehr vereinfachten.
Ein Schwerpunkt der Arbeit beschäftigte sich mit der Frage, wie aus Verhaltensbeschreibungen erstellte DRSen in Verhaltensschemata in Form von UMTL–Programmen überführt werden können. Ein zentrales Ergebnis dieser Untersuchungen ist die Tatsache, daß aus Verhaltensbeschreibungen erstellte DRSen unterschiedliche *Informationsarten* – namentlich schema– bzw. diskurs–spezifische Informationen – enthalten und eine Übersetzung der DRSen nach UMTL in Anlehnung an [Kamp & Reyle 93] deshalb nicht möglich ist. Diesem Umstand Rechnung tragend konnte ein Verfahren

entwickelt werden, welches DRSen schrittweise in ein Verhaltensschema *transformiert*. Die prinzipielle Anwendbarkeit des Verfahrens konnte anhand einer prototypischen Implementierung gezeigt werden.
Ergebnis der Implementierung auf dem aktuellen Stand ist ein Programm–System, welches den gesamten Weg von einfachen natürlichsprachlichen Verhaltensbeschreibungen zu einer für die Bildauswertung nutzbaren rechner–internen Darstellung des darin enthaltenen Wissens automatisch beschreiten kann.

7.2 Ausblick

Eine zentrale, bisher ungeklärte Frage betrifft den *Rückbezug* auf schon bestehende Situationsschemata bei der DRS–Transformation. Bisher können von dem implementierten Programm–System lediglich Verhaltensbeschreibungen korrekt verarbeitet werden, welche nur einfache Verhaltensschemata beschreiben. Komplexere Texte, welche auch wiederholt Bezug auf einen bestimmten Begriff (bzw. ein bestimmtes Situationsschema) nehmen, setzen voraus, das solche Bezüge im Text auf schon bestehende (d.h. ggf. schon erstellte) Situationsschemata vom Programm erkannt und entsprechend verarbeitet werden können. Zukünftige Untersuchungen sollten vor allem diese Frage des Rückbezuges klären.
Ein weiterer Punkt, der noch genauerer Untersuchung bedarf, betrifft die Erstellung von UMTL–Regeln bei der DRS–Transformation. Die bisher von Transformationsregeln verwendeten Aktionsteilmethoden sind als *Makro–Befehle* in dem Sinne zu sehen, daß sie ganze Bündel von UMTL–Regeln aus den ihnen übergebenen Informationen erstellen. Für eine weitere Benutzung des Transformationsverfahrens – unter Umständen auch für die Verarbeitung andersartiger natürlichsprachlicher Texte – wäre es wünschenswert, diese Makro–Befehle durch kleinere, flexibler einsetzbare Methoden zu ersetzen. Eine Schritt in diese Richtung könnte die Erstellung von Implikationsbeziehungen innerhalb der zu transformierenden DRS sein, welche dann in weiteren Transformationsschritten in UMTL–Regeln übersetzt werden. Insgesamt sollte die Transformation einer DRS in einen *Umformungs–* sowie einen *Übersetzungsteil* getrennt werden: Im Umformungsteil sollten lediglich Aktionen auf der DRS durchgeführt werden. Alle Informationen, welche bei diesen Schritten benötigt werden, sollten ebenfalls nur in der DRS zwischengespeichert werden. Im Übersetzungteil sollten keine Transformationen an der DRS mehr vorgenommen werden. Jede DRS–Bedingung sollte zu diesem Zeitpunkt bereits eine Form haben, die sie einer Standard–Übersetzung nach UMTL zugänglich macht. Lediglich schon übersetzte Bedingungen der DRS werden in diesem Teil der Transformation gelöscht und die DRS so abgearbeitet. Eine solch strikte Trennung wurde in der vorliegenden Arbeit und der durchgeführten Implementierung ansatzweise realisert, muß jedoch noch klarer vollzogen werden.
Die Anbindung der durch DRS–Transformation erstellten Verhaltensschemata an die Bildfolgenauswertung muß ebenfalls genauer untersucht werden. Bisher wurde davon

ausgegangen, daß die in einer natürlichsprachlichen Verhaltensbeschreibungen auftretenden Begriffe (wie etwa das Verb *drive*) direkt in das zu erstellende Verhaltensschema übernommen werden können. Diese Begriffe müssen bisher durch die *konkreteren* Begriffe der Bildfolgenauswertung (wie etwa Position und Geschwindigkeit eines beobachteten Akteurs) in einer *Terminologie* erklärt werden (vgl. [Haag & Nagel 2000]). Es wäre denkbar, eine solche Erklärung abstrakterer Begriffe in Form einer Terminologie bereits in die DRS–Transformation zu integrieren.
Ein letzter Punkt, der hier erwähnt werden soll, betrifft die grafische Inspektion von Verhaltensschemata in Form von SGTs. Die in der vorliegenden Arbeit angefangenen Untersuchungen zur Erstellung einer ggf. dann auch interaktiv nutzbaren grafischen Schnittstelle zu SGTs sollten fortgesetzt werden. Vielversprechend in diesem Zusammenhang erscheint vor allem das in Kapitel 4 vorgestellte Werkzeug *DiaGen* zur Erstellung von Diagramm–Editoren.

Anhang A

Grammatik–Dokumentation

Der folgende Abschnitt dokumentiert die in der vorliegenden Arbeit verwendete Grammatik sowie die zugehörigen Konstruktionsregeln. Anhang A.1 beinhaltet zunächst die Textdatei, aus der heraus mit dem von [Arens & Ottlik 2000] erstellten Programm automatisch ein DRS–Erzeuger generiert wird. Anhang A.2 bzw. A.3 beinhalten die in dieser Textdatei definierten Attribute und deren Verteilung auf die einzelnen Nichtterminale. Die drei darauffolgenden Abschnitte stellen die Grammatik, die Reihenfolge der Konstruktionsregeln sowie die Konstruktionsregeln selbst in übersichtlicher Form dar. Diese Abschnitte sind der automatisch erstellten Dokumentation entnommen. Abgeschlossen wird die Dokumentation im Anhang A.7 mit einer Aufstellung der verwendeten Morphologie–Regeln.

A.1 Grammatikdatei

```
/*=========================*/
/*
 *  NONTERMINALS
 */
/*=========================*/
NONTERMINALS

STARTSYMBOL #

TEXT #
SEN #
RPRO #
NP #
ADJP #
PVP #
VP #
CONJOFNP #
MPREP #
D #
N #
V #
ADJ #
TO #
SEP #
AND #
X #
PREP #
ADVP #
DO #
NOT #
BE #
MADVP #
EMPTYWORD #
BY #
THAT #
DOT #
SUCH #

/*=========================*/
/*
 *  ATTRIBUTES
 */
/*=========================*/
ATTRIBUTES

modality [ISMODAL,ISNOTMODAL] #
finitivity [GERUND,FINITE,INFINITE,PP] #
transitivity [TRANSITIVE,INTRANSITIVE] #
genusverbi [ACTIVE, PASSIVE] #
number [SINGULAR,PLURAL] #
citation [ISCITATION, ISNOCITATION] #
determined [HASDETERMINER, HASNODETERMINER] #
definity [DEFINITE, INDEFINITE] #
reading [CONJUNCTIVE,DISJUNCTIVE] #

/* attributes only for CR-appliance */
status [ISREDUCED] #
processed [ISPROCESSED, ISNOTPROCESSED] #
action [CONTAINSACTION, CONTAINSNOACTION] #


/*=========================*/
```

```
/*
 *  WORDPROPERTIES
 */
/*========================*/
WORDPROPERTIES

D [number,definity] #
N [number] #
V [modality,finitivity, transitivity, number] #
ADJ [_] #
TO [_] #
SEP [_] #
AND [reading] #
X [_] #
PREP [_] #
DO [finitivity, number] #
NOT [_] #
BE [finitivity, number] #
BY [_] #
THAT [_] #
DOT [_] #
SUCH [_] #

/*========================*/
/*
 *  RULES
 */
/*========================*/
RULES

STARTSYMBOL [_] -> TEXT [_] #

TEXT [_] -> SEN [_] DOT [_] TEXT [_] #
TEXT [_] -> SEN [_] DOT [_] #

SEN [processed = ISNOTPROCESSED, number = a]
   -> CONJOFNP [number = a]
      PVP [finitivity = FINITE, number = a] #

NP [number=a,citation=ISCITATION,determined=d, action = e]
  -> X [_]
     NP [number=a,citation=ISNOCITATION,determined=d, action = e]
     X [_] #
NP [number=a,determined=HASDETERMINER,
    citation=ISNOCITATION,action=e]
   -> D [number=a]
      NP [number=a,determined=HASNODETERMINER, action = e] #
NP [number=a,determined=HASNODETERMINER,
    citation=ISNOCITATION,action=e]
   -> ADJ [_]
      NP [number=a,determined=HASNODETERMINER, action = e] #
NP [number=a,determined=HASNODETERMINER,
    citation=ISNOCITATION,action=CONTAINSNOACTION]
   -> N [number=a]
      ADJP [_] #
NP [number=SINGULAR,determined=HASNODETERMINER,
    citation=ISNOCITATION, action = CONTAINSNOACTION]
   -> N [number=SINGULAR] #
NP [number=SINGULAR,determined=HASNODETERMINER,
    citation=ISNOCITATION, action = e]
   -> PVP [finitivity=GERUND, action = e] #
NP [number=SINGULAR,determined=HASNODETERMINER,
    citation=ISNOCITATION, action = e]
   -> PVP [finitivity=INFINITE, action = e] #

CONJOFNP [number = PLURAL, reading = r]
         -> NP [_]
            SEP [_]
            CONJOFNP [number = PLURAL, reading = r] #
CONJOFNP [number = PLURAL, reading = r]
         -> NP [_]
            AND [reading = r]
            NP [_] #
CONJOFNP [number = a]
         -> NP [number = a] #

ADJP [_] -> THAT [_] SEN [_] #
ADJP [_] -> PREP [_] CONJOFNP [_] #
ADJP [_] -> X [_] PVP [finitivity = GERUND] X [_] #
ADJP [_] -> PVP [finitivity = GERUND] #

PVP [finitivity = a, number = b, action = e, genusverbi = g]
    -> VP [finitivity = a, number = b, action = e, genusverbi = g]
       MADVP [_] #
PVP [finitivity = a, number = b, action = e, genusverbi = g]
    -> VP [finitivity = a, number = b, action = e, genusverbi = g] #

VP [finitivity = INFINITE, action = e, genusverbi = g]
   -> TO [_]
      VP [finitivity = INFINITE, action = e, genusverbi = g] #
VP [finitivity = FINITE, number = b,
    action = e, genusverbi = ACTIVE]
   -> DO [finitivity = FINITE, number = b]
      NOT [_]
      VP [finitivity = INFINITE, action = e] #
VP [finitivity = a, number = b,
    action = CONTAINSNOACTION, genusverbi = PASSIVE]
   -> BE [finitivity = a, number = b]
      V [finitivity = PP] #
VP [finitivity = FINITE, number = b,
    action = e, genusverbi = g]
   -> V [modality=ISMODAL, finitivity = FINITE, number = b]
      VP [finitivity = INFINITE, action = e, genusverbi = g] #
VP [finitivity = a, transitivity = TRANSITIVE, number = b,
    action = e, genusverbi = ACTIVE]
   -> V [finitivity = a, transitivity = TRANSITIVE, number = b]
      CONJOFNP [action = e] #
VP [finitivity = a, transitivity = INTRANSITIVE, number = b,
    action = CONTAINSNOACTION, genusverbi = ACTIVE]
   -> V [modality=ISNOTMODAL,finitivity=a,transitivity=INTRANSITIVE,number=b] #

ADVP [_] -> SUCH [_] THAT [_] SEN [_] #
ADVP [_] -> BY [_] CONJOFNP [_] #
ADVP [_] -> MPREP [_] CONJOFNP [_] #

MADVP [_] -> ADVP [_] SEP [_] MADVP [_] #
MADVP [_] -> ADVP [_] AND [_] ADVP [_] #
MADVP [_] -> ADVP [_] ADVP [_] #
MADVP [_] -> ADVP [_] #

MPREP [_] -> PREP [_] PREP [_] #
MPREP [_] -> PREP [_] #


/*========================*/
/*
 *  LEX_RULES
 */
/*========================*/
D [_] -> LEX { } #
ADJ [_] -> LEX { } #
TO [_] -> LEX { } #
SEP [_] -> LEX { } #
AND [_] -> LEX { } #
DO [_] -> LEX { } #
BE [_] -> LEX { } #
NOT [_] -> LEX { } #
X [_] -> LEX { } #
PREP [_] -> LEX { } #
BY [_] -> LEX { } #
THAT [_] -> LEX { } #
DOT [_] -> LEX { } #
SUCH [_] -> LEX { } #

V [_] -> LEX
  { [finitivity = PP]
    <vowel>"y"            -> <vowel>"yed";
    <cons>"y"             -> <cons>"ied";
    <cons1><vowel><cons2> -> <cons1><vowel><cons2><cons2>"ed";
    "e"                   -> "ed";
     _                    -> "ed";
  }
  { [finitivity = FINITE]
    <vowel>"y"            -> <vowel>"ys";
    <cons>"y"             -> <cons>"ies";
    <cons1><vowel><cons2> -> <cons1><vowel><cons2><cons2>"es";
    "e"                   -> "es";
     _                    -> "s";
  }
  { [finitivity = GERUND]
    "e"                   -> "ing";
    "ie"                  -> "ying";
    <cons1><vowel><cons2> -> <cons1><vowel><cons2><cons2>"ing";
    _                     -> "ing";
  }
#

N [_] -> LEX
  { [number = PLURAL]
    "o"                   -> "oes";
    "ss"                  -> "sses";
    "sh"                  -> "shes";
    "ch"                  -> "ches";
    "x"                   -> "xes";
    <vowel>"y"            -> <vowel>"ys";
    <cons>"y"             -> <cons>"ies";
```

```
    <vowel>"s"            -> <vowel>"sses";
    "f"                   -> "ves";
    "fe"                  -> "ves";
    _                     -> "s";
  }
#

/*========================*/
/*
 *  CONSTRUCTIONRULES
 */
/*========================*/
CONSTRUCTION_RULES

CR_CITATION1
    TS
      1:NP [_] {
        2:X [_]
        3:NP [_]
        4:X [_]
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:getWord(s)
          3:TS.3.copy()
        }
      #
    #
    ACTION
      s = getSubTreeString(TS.3);
      substitute(HS_1, TS.1);
    #
  # // end of CR_CITATION1


CR_CITATION2
    TS
      1:ADJP [_] {
        2:X [_]
        3:PVP [_]
        4:X [_]
      }
    #
    HELP_STRUCTURES
      HS_1
        1: TS.1 {
          2:getWord(s)
          3:TS.3.copy()
        }
      #
    #
    ACTION
      s = getSubTreeString(TS.3);
      substitute(HS_1, TS.1);
    #
  # // end of CR_CITATION2


CR_NOUN1
    TS
      1:N [_] {
        2:getWord(n)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:getWord(d)
      #
    #
    ACTION
      d = addNewLocalDR(TS.2);
      addNewLocalAttribute(n,d);
      substitute(HS_1, TS.1);
    #
  # // end of CR_NOUN


CR_NOUNPHRASE1
    TS
      1:NP [_] {
        2:getWord(d)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.2.copy()
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_NOUNPHRASE1


CR_NOUNPHRASE2
    TS
      1:NP [_] {
        2:getWord(c)
        3:getWord(d)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.3.copy()
      #
    #
    ACTION
      addNewName(c,d);
      substitute(HS_1, TS.1);
    #
  # // end of CR_NOUNPHRASE2


CR_ADJECTIVE1
    TS
      1:NP [_] {
        2:ADJ [_] {
          3:getWord(a)
        }
        4:getWord(d)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.4.copy()
      #
    #
    ACTION
      addNewLocalAttribute(a,d);
      substitute(HS_1, TS.1);
    #
  # // end of CR_ADJECTIVE1

CR_DETERMINER1
    TS
      1:NP [_] {
        2:D [definity = INDEFINITE] {
          3:getWord(d)
        }
        4:getWord(x)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.4.copy()
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_DETERMINER1


CR_DETERMINER2
    TS
      1:NP [_] {
        2:D [definity = DEFINITE] {
          3:getWord(d)
        }
        4:getWord(x)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:getWord(y)
      #
    #
    ACTION
      y = getDefAntecedent(TS.4);
      substitute(HS_1, TS.1);
    #
  # // end of CR_DETERMINER2
```

```
CR_CONJOFNP4
    TS
      1:CONJOFNP [reading = CONJUNCTIVE] {
        2:getWord(x)
        3:SEP [_]
        4:getWord(y)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:getWord(z)
      #
    #
    ACTION
      s = getString(CONJUNCTIVE);
      z = addNewLocalReferentSum(x,y);
      addNewLocalAttribute(s,z);
      substitute(HS_1, TS.1);
    #
  # // end of CR_CONJOFNP4


CR_CONJOFNP5
    TS
      1:CONJOFNP [reading = DISJUNCTIVE] {
        2:getWord(x)
        3:SEP [_]
        4:getWord(y)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:getWord(z)
      #
    #
    ACTION
      s = getString(DISJUNCTIVE);
      z = addNewLocalReferentSum(x,y);
      addNewLocalAttribute(s,z);
      substitute(HS_1, TS.1);
    #
  # // end of CR_CONJOFNP5


CR_CONJOFNP3
    TS
      1:CONJOFNP [reading = CONJUNCTIVE] {
        2:getWord(x)
        3:AND [_]
        4:getWord(y)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:getWord(z)
      #
    #
    ACTION
      s = getString(CONJUNCTIVE);
      z = addNewLocalReferentSum(x,y);
      addNewLocalAttribute(s,z);
      substitute(HS_1, TS.1);
    #
  # // end of CR_CONJOFNP3


CR_CONJOFNP2
    TS
      1:CONJOFNP [reading = DISJUNCTIVE] {
        2:getWord(x)
        3:AND [_]
        4:getWord(y)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:getWord(z)
      #
    #
    ACTION
      s = getString(DISJUNCTIVE);
      z = addNewLocalReferentSum(x,y);
      addNewLocalAttribute(s,z);
      substitute(HS_1, TS.1);
    #
  # // end of CR_CONJOFNP2
```

```
CR_CONJOFNP1
    TS
      1:CONJOFNP [_] {
        2:getWord(x)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.2.copy()
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_CONJOFNP1


CR_ADJECTIVEPHRASE1
    TS
      1:NP [_] {
        2:getWord(x)
        3:ADJP [_] {
          4:PREP [_] {
            5:getWord(p)
          }
          6:getWord(y)
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.2.copy()
      #
    #
    ACTION
      addNewLocalPredicate(p,x,y);
      substitute(HS_1, TS.1);
    #
  # // end of CR_ADJECTIVEPHRASE1


CR_VERBALADJPHRASE1
    TS
      1:NP [_] {
        2:getWord(x)
        3:ADJP [_] {
          4:PVP [_]
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:SEN [processed = ISPROCESSED] {
          2:getWord(x)
          3:TS.4.copy()
        }
      #
      HS_2
        1:TS.2
      #
    #
    ACTION
      e = getNextFreeEventReferent();
      d = buildDRS(HS_1, POSITIVE);
      addLocalEventReferent(e, d);
      substitute(HS_2, TS.1);
    #
  # // end of CR_VERBALADJPHRASE1


CR_VERBALADJPHRASE2
    TS
      1:NP [_] {
        2:getWord(x)
        3:ADJP [_] {
          4:getWord(c)
          5:PVP [_]
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:SEN [processed = ISPROCESSED] {
          2:getWord(x)
          3:TS.5.copy()
        }
      #
      HS_2
```

```
        1:TS.2
      #
    #
    ACTION
      e = getNextFreeEventReferent();
      d = buildDRS(HS_1, POSITIVE);
      addLocalEventReferent(e, d);
      addNewName(c,e);
      substitute(HS_2, TS.1);
    #
  # // end of CR_VERBALADJPHRASE2


CR_THATADJ1
    TS
      1:ADJP [_] {
        2:THAT [_] {
          3:getWord(t)
        }
        4:SEN [_] {
          5:getWord(n)
          6:PVP [_]
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:PREP [_] {
            3:TS.3
          }
          4:getWord(e)
        }
      #
      HS_2
        1:SEN [processed = ISPROCESSED] {
          2:TS.5.copy()
          3:TS.6.copy()
        }
      #
    #
    ACTION
      e = getNextFreeEventReferent();
      d = buildDRS(HS_2, POSITIVE);
      addLocalEventReferent(e, d);
      substitute(HS_1, TS.1);
    #
  # // end of CR_THATADJ1


CR_SUCHTHAT1
    TS
      1:ADVP [_] {
        2:SUCH [_]
        3:THAT [_]
        4:SEN [_] {
          5:getWord(n)
          6:PVP [_]
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:ADVP [_] {
          2:TS.2.copy()
          3:TS.3.copy()
        }
      #
      HS_2
        1:SEN [processed = ISPROCESSED] {
          2:TS.5.copy()
          3:TS.6.copy()
        }
      #
      HS_3
        1:TS.1 {
          2:MPREP [_] {
            3:getWord(s)
          }
          4:getWord(e)
        }
      #
    #
    ACTION
      s = getSubTreeString(HS_1);
      e = getNextFreeEventReferent();
      d = buildDRS(HS_2, POSITIVE);
      addLocalEventReferent(e, d);
      substitute(HS_3, TS.1);
    #
  # // end of CR_SUCHTHAT1


CR_VERBALNP1
    TS
      1:NP [_] {
        4:PVP [action = CONTAINSNOACTION]
      }
    #
    HELP_STRUCTURES
      HS_1
        1:SEN [processed = ISPROCESSED] {
          2:getWord(x)
          3:TS.4.copy()
        }
      #
      HS_2
        1:getWord(e)
      #
    #
    ACTION
      e = getNextFreeEventReferent();
      s = getString(SOMEONE);
      x = addNewGlobalDR(TS.1);
      addNewGlobalAttribute(s,x);
      d = buildDRS(HS_1, POSITIVE);
      addLocalEventReferent(e, d);
      substitute(HS_2, TS.1);
    #
  # // end of CR_VERBALNP1


CR_VERBALNP2
    TS
      1:NP [_] {
        4:PVP [_]
      }
    #
    HELP_STRUCTURES
      HS_1
        1:SEN [processed = ISPROCESSED] {
          2:getWord(x)
          3:TS.4.copy()
        }
      #
      HS_2
        1:getWord(e)
      #
    #
    ACTION
      e = getNextFreeEventReferent();
      s = getString(SOMEONE);
      x = addNewGlobalDR(TS.1);
      addNewGlobalAttribute(s,x);
      d = buildDRS(HS_1, POSITIVE);
      addLocalEventReferent(e, d);
      substitute(HS_2, TS.1);
    #
  # // end of CR_VERBALNP2


CR_EVENT1
    TS
      1:SEN [processed = ISNOTPROCESSED] {
        2:getWord(x)
        3:PVP [_]
      }
    #
    HELP_STRUCTURES
      HS_1
        1:SEN [processed = ISPROCESSED] {
          2:TS.2.copy()
          3:TS.3.copy()
        }
      #
      HS_2
        1:getWord(e)
      #
    #
    ACTION
      e = getNextFreeEventReferent();
      d = buildDRS(HS_1, POSITIVE);
      addLocalEventReferent(e, d);
      substitute(HS_2, TS.1);
    #
  # // end of CR_EVENT1
```

```
CR_MULTIPREP1
    TS
      1:MPREP [_] {
        2:PREP [_] {
          4:getWord(p)
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:TS.4.copy()
        }
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_MULTIPREP1


CR_MULTIPREP2
    TS
      1:MPREP [_] {
        2:PREP [_] {
          3:getWord(p)
        }
        4:PREP [_] {
          5:getWord(p)
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:getWord(s)
        }
      #
    #
    ACTION
      s = getSubTreeString(TS.1);
      substitute(HS_1, TS.1);
    #
  # // end of CR_MULTIPREP2


CR_ADVERBIALPHRASE1
    TS
      1:ADVP [_] {
        2:MPREP [_] {
          3:getWord(p)
        }
        4:getWord(x)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:ADVP [status = ISREDUCED]
      #
    #
    ACTION
      addNewLocalPredicate(p,x);
      substitute(HS_1, TS.1);
    #
  # // end of CR_ADVERBIALPHRASE1


CR_BYSUBJECT1
    TS
      1:ADVP [_] {
        2:BY [_] {
          3:getWord(b)
        }
        4:getWord(s)
      }
    #
    HELP_STRUCTURES
      HS_1
        1:BY [_] {
          2:getWord(s)
        }
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_BYSUBJECT1


CR_ADVERBIALBY1
    TS
      1:SEN [processed = ISPROCESSED] {
        2:getWord(s)
        3:PVP [genusverbi = ACTIVE] {
          4:VP [_]
          5:MADVP [_] {
            6:ADVP [_] {
              7:BY [_] {
                8:getWord(b)
              }
              9:getWord(e)
            }
          }
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:TS.2
          3:TS.3 {
            4:TS.4.copy()
          }
        }
      #
    #
    ACTION
      addNewLocalPredicate(b,e);
      substitute(HS_1, TS.1);
    #
  # // end of CR_ADVERBIALBY1


CR_MULTIADVP1
    TS
      1:MADVP [_] {
        2:ADVP [status = ISREDUCED]
        3:SEP [_]
        4:MADVP [_]
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.4.copy()
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_MULTIADVP1


CR_MULTIADVP2
    TS
      1:MADVP [_] {
        2:ADVP [status = ISREDUCED]
        3:AND [_]
        4:ADVP [status = ISREDUCED]
      }
    #
    HELP_STRUCTURES
    #
    ACTION
      removeTS();
    #
  # // end of CR_MULTIADVP2


CR_MULTIADVP3
    TS
      1:MADVP [_] {
        2:ADVP [status = ISREDUCED]
      }
    #
    HELP_STRUCTURES
    #
    ACTION
      removeTS();
    #
  # // end of CR_MULTIADVP3


CR_MULTIADVP4
    TS
```

```
      1:MADVP [_] {
        2:ADVP [status = ISREDUCED]
        3:ADVP [status = ISREDUCED]
      }
    #
    HELP_STRUCTURES
    #
    ACTION
      removeTS();
    #
  # // end of CR_MULTIADVP4


CR_MULTIADVP5
    TS
      1:MADVP [_] {
        2:ADVP [status = ISREDUCED]
        3:BY [_]
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.3.copy()
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_MULTIADVP5


CR_MULTIADVP6
    TS
      1:MADVP [_] {
        2:BY [_]
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.2.copy()
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_MULTIADVP6


CR_SENTENCE1
    TS
      1:SEN [processed = ISPROCESSED] {
        2:getWord(d)
        3:PVP [_] {
          4:VP [_] {
            5:V [_] {
              6:getWord(v)
            }
          }
        }
      }
    #
    HELP_STRUCTURES
    #
    ACTION
      addNewLocalPredicate(v,d);
      removeTS();
    #
  # // end of CR_SENTENCE1


CR_SENTENCE2
    TS
      1:SEN [processed = ISPROCESSED] {
        2:getWord(d)
        3:PVP [_] {
          4:VP [_] {
            5:V [_] {
              6:getWord(v)
            }
            7:getWord(e)
          }
        }
      }
    #
    HELP_STRUCTURES
    #
    ACTION
      addNewLocalPredicate(v,d,e);
      removeTS();
    #
  # // end of CR_SENTENCE2


CR_SENTENCE3
    TS
      1:SEN [processed = ISPROCESSED] {
        2:getWord(d)
        3:PVP [_] {
          4:VP [_] {
            5:TO [_]
            6:VP [_]
          }
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:TS.2.copy()
          3:TS.3 {
            4:TS.6.copy()
          }
        }
      #
    #
    ACTION
      substitute(HS_1, TS.1);
    #
  # // end of CR_SENTENCE3


CR_PSENTENCE1
    TS
      1:SEN [processed = ISPROCESSED] {
        2:getWord(o)
        3:PVP [_] {
          4:VP [_] {
            5:BE [_]
            6:V [_]
          }
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:getWord(s)
          3:TS.3 {
            4:TS.4 {
              5:TS.6.copy()
              6:TS.2
            }
          }
        }
      #
    #
    ACTION
      s = addNewGlobalDR(TS.6);
      p = getString(SOMEONE);
      addNewGlobalAttribute(p,s);
      substitute(HS_1, TS.1);
    #
  # // end of CR_PSENTENCE1


CR_PSENTENCE2
    TS
      1:SEN [processed = ISPROCESSED] {
        2:getWord(o)
        3:PVP [genusverbi = PASSIVE] {
          4:VP [_] {
            5:BE [_]
            6:V [_]
          }
          7:BY [_] {
            8:getWord(s)
          }
        }
      }
    #
    HELP_STRUCTURES
      HS_1
        1:TS.1 {
          2:TS.8
          3:TS.3 {
            4:TS.4 {
```

```
            5:TS.6.copy()
            6:TS.2
          }
        }
      }
    #
  #
  ACTION
    substitute(HS_1, TS.1);
  #
# // end of CR_PSENTENCE2


CR_SENTENCE5
  TS
    1:SEN [processed = ISPROCESSED] {
      2:getWord(d)
      3:PVP [_] {
        4:VP [_] {
          5:V [_]
          6:VP [_]
        }
      }
    }
  #
  HELP_STRUCTURES
    HS_1
      1:TS.1 {
        2:TS.2.copy()
        3:TS.3 {
          4:TS.6.copy()
        }
      }
    #
  #
  ACTION
    substitute(HS_1, TS.1);
  #
# // end of CR_SENTENCE5


CR_PSENTENCE5
  TS
    1:SEN [processed = ISPROCESSED] {
      2:getWord(d)
      3:PVP [genusverbi = PASSIVE] {
        4:VP [_] {
          5:V [_]
          6:VP [_]
        }
        7:BY [_]
      }
    }
  #
  HELP_STRUCTURES
    HS_1
      1:TS.1 {
        2:TS.2.copy()
        3:TS.3 {
          4:TS.6.copy()
          5:TS.7.copy()
        }
      }
    #
  #
  ACTION
    substitute(HS_1, TS.1);
  #
# // end of CR_PSENTENCE5


CR_SENTENCE6
  TS
    1:SEN [processed = ISPROCESSED] {
      2:getWord(d)
      3:PVP [_] {
        4:VP [_] {
          5:DO [_]
          6:NOT [_]
          7:VP [_]
        }
      }
    }
  #
  HELP_STRUCTURES
    HS_1
      1:TS.1 {
        2:TS.2.copy()
        3:TS.3 {
          4:TS.7.copy()
        }
      }
    #
  #
  ACTION
    f = buildDRS(HS_1, NEGATIVE);
    addNewLocalSubDRS(f);
    removeTS();
  #
# // end of CR_SENTENCE6


CR_CLEAN1
  TS
    1:STARTSYMBOL [_] {
      2:TEXT [_] {
        3:getWord(x)
        4:DOT [_]
      }
    }
  #
  HELP_STRUCTURES
  #
  ACTION
    removeTS();
  #
# // end of CR_CLEAN1


CR_CLEAN2
  TS
    1:TEXT [_] {
      2:getWord(x)
      3:DOT [_]
      4:TEXT [_]
    }
  #
  HELP_STRUCTURES
    HS_1
      1:TS.4.copy()
    #
  #
  ACTION
    substitute(HS_1, TS.1);
  #
# // end of CR_CLEAN2


CONSTRUCTIONRULE_ORDER

  CR_CITATION1, CR_CITATION2,

  CR_NOUN1,
  CR_NOUNPHRASE1, CR_NOUNPHRASE2,
  CR_CONJOFNP1, CR_CONJOFNP2,
  CR_CONJOFNP3, CR_CONJOFNP4, CR_CONJOFNP5,
  CR_DETERMINER1,CR_DETERMINER2,
  CR_ADJECTIVE1,
  CR_ADJECTIVEPHRASE1,

  CR_VERBALNP1,

  CR_NOUN1,
  CR_NOUNPHRASE1, CR_NOUNPHRASE2,
  CR_CONJOFNP1, CR_CONJOFNP2,
  CR_CONJOFNP3, CR_CONJOFNP4, CR_CONJOFNP5,
  CR_DETERMINER1,CR_DETERMINER2,
  CR_ADJECTIVE1,
  CR_ADJECTIVEPHRASE1,

  CR_VERBALADJPHRASE1, CR_VERBALADJPHRASE2,
  CR_THATADJ1,
  CR_SUCHTHAT1,

  CR_NOUN1,
  CR_NOUNPHRASE1, CR_NOUNPHRASE2,
  CR_CONJOFNP1, CR_CONJOFNP2,
  CR_CONJOFNP3, CR_CONJOFNP4, CR_CONJOFNP5,
  CR_DETERMINER1,CR_DETERMINER2,
  CR_ADJECTIVE1,
  CR_ADJECTIVEPHRASE1,

  CR_THATADJ1,

  CR_NOUN1,
  CR_NOUNPHRASE1, CR_NOUNPHRASE2,
```

```
    CR_CONJOFNP1, CR_CONJOFNP2,
    CR_CONJOFNP3, CR_CONJOFNP4, CR_CONJOFNP5,
    CR_DETERMINER1,CR_DETERMINER2,
    CR_ADJECTIVE1,
    CR_ADJECTIVEPHRASE1,

    CR_VERBALNP2,

    CR_NOUN1,
    CR_NOUNPHRASE1, CR_NOUNPHRASE2,
    CR_CONJOFNP1, CR_CONJOFNP2,
    CR_CONJOFNP3, CR_CONJOFNP4, CR_CONJOFNP5,
    CR_DETERMINER1,CR_DETERMINER2,
    CR_ADJECTIVE1,
    CR_ADJECTIVEPHRASE1,

    CR_NOUN1,
    CR_NOUNPHRASE1, CR_NOUNPHRASE2,
    CR_CONJOFNP1, CR_CONJOFNP2,
    CR_CONJOFNP3, CR_CONJOFNP4, CR_CONJOFNP5,
    CR_DETERMINER1,CR_DETERMINER2,
    CR_ADJECTIVE1,
    CR_ADJECTIVEPHRASE1,

    CR_SUCHTHAT1,

    CR_EVENT1,

    CR_ADVERBIALPHRASE1, CR_ADVERBIALBY1, CR_BYSUBJECT1,
    CR_MULTIPREP1, CR_MULTIPREP2,
    CR_ADVERBIALPHRASE1,
    CR_MULTIADVP1, CR_MULTIADVP2, CR_MULTIADVP3,
    CR_MULTIADVP4, CR_MULTIADVP5, CR_MULTIADVP6,
    CR_ADVERBIALPHRASE1,

    CR_SENTENCE3, CR_PSENTENCE1, CR_PSENTENCE2,
    CR_PSENTENCE5, CR_SENTENCE5, CR_SENTENCE6,
    CR_SENTENCE1, CR_SENTENCE2,

    CR_CLEAN2, CR_CLEAN1
  #
*
```

A.2 Attribut–Verzeichnis

Attribut	mögliche Ausprägung	Bedeutung
modality	[**ISMODAL, ISNOTMODAL**]	gibt an, ob ein Verb ein Modalverb ist oder nicht.
finitivity	[**FINITE, INFINITE, GERUND, PP**]	gibt an, für welche Form eines Verbes ein Nichtterminal (NT) steht.
transitivity	[**TRANSITIVE, INTRANSITIVE**]	gibt die Transitivität eines NT an.
genusverbi	[**ACTIVE, PASSIVE**]	gibt an ob eine Verbphrase im Aktiv oder Passive steht.
number	[**SINGULAR, PLURAL**]	gibt den Numerus eines NT an.
citation	[**ISCITATION, ISNOCITATION**]	gibt an, ob ein NT ein Zitat enthält.
determined	[**HASDETERMINER, HASNODETERMINER**]	gibt an, ob eine Nominalphrase bereits einen Artikel enthält.
definity	[**DEFINITE, INDEFINITE**]	gibt an, ob eine Nominalphrase definit oder indefinit ist.
reading	[**CONJUNCTIVE, DISJUNCTIVE**]	gibt an, ob eine Aufzählung als Konjunktion oder Disjunktion zu lesen ist.

A.3 Nichtterminale und ihre Attribute

Nichtterminal	Attribute	syntaktische Rolle
D	[**number, definity**]	Artikel (*determiner*)
N	[**number**]	Nomen (*noun*)
V	[**modality, finitivity, transitivity, number**]	Verb (*verb*)
ADJ	[ ]	Adjektiv (*adjective*)
TO	[ ]	Spezialterminal für das englische *to*
SEP	[ ]	Spezialterminal für Trennzeichen
AND	[**reading**]	Spezialterminal für das englische *or* bzw. *and*
X	[ ]	Spezialterminal für Zitat–Zeichen
PREP	[ ]	Präposition (*preposition*)
DO	[**finitivity, number**]	Spezialterminal für Formen von *do*
NOT	[ ]	Spezialterminal für das englische *not*
BE	[**finitivity, number**]	Spezialterminal für Formen von *be*
BY	[ ]	Spezialterminal für das englische *by*
THAT	[ ]	Spezialterminal für das englische *that*
DOT	[ ]	Spezialterminal für Satzendzeichen wie Punkte
SUCH	[ ]	Spezialterminal für das englische *such*

A.4 Grammatikregeln

$$\underset{[\ ]}{\texttt{STARTSYMBOL}} \rightarrow \underset{[\ ]}{\texttt{TEXT}}$$

$$\underset{[\ ]}{\texttt{TEXT}} \rightarrow \underset{[\ ]}{\texttt{SEN}}\ \underset{[\ ]}{\texttt{DOT}}\ \underset{[\ ]}{\texttt{TEXT}}$$

$$\underset{[\ ]}{\texttt{TEXT}} \rightarrow \underset{[\ ]}{\texttt{SEN}}\ \underset{[\ ]}{\texttt{DOT}}$$

$$\underset{\left[\begin{array}{l} processed = ISNOTPROCESSED \\ number = a \end{array}\right]}{\texttt{SEN}} \rightarrow \underset{[\ number = a\ ]}{\texttt{CONJOFNP}}\ \underset{\left[\begin{array}{l} finitivity = FINITE \\ number = a \end{array}\right]}{\texttt{PVP}}$$

$$\underset{\left[\begin{array}{l} number = a \\ citation = ISCITATION \\ determined = d \\ action = e \end{array}\right]}{\texttt{NP}} \rightarrow \underset{[\ ]}{\texttt{X}}\ \underset{\left[\begin{array}{l} number = a \\ citation = ISNOCITATION \\ determined = d \\ action = e \end{array}\right]}{\texttt{NP}}\ \underset{[\ ]}{\texttt{X}}$$

$$\underset{\left[\begin{array}{l} number = a \\ determined = HASDETERMINER \\ citation = ISNOCITATION \\ action = e \end{array}\right]}{\texttt{NP}} \rightarrow \underset{[\ number = a\ ]}{\texttt{D}}\ \underset{\left[\begin{array}{l} number = a \\ determined = HASNODETERMINER \\ action = e \end{array}\right]}{\texttt{NP}}$$

$$\underset{\left[\begin{array}{l} number = a \\ determined = HASNODETERMINER \\ citation = ISNOCITATION \\ action = e \end{array}\right]}{\texttt{NP}} \rightarrow \underset{[\ ]}{\texttt{ADJ}}\ \underset{\left[\begin{array}{l} number = a \\ determined = HASNODETERMINER \\ action = e \end{array}\right]}{\texttt{NP}}$$

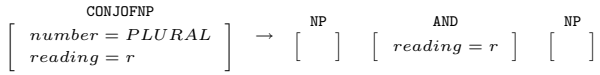
$$\underset{\left[\begin{array}{l} number = a \\ determined = HASNODETERMINER \\ citation = ISNOCITATION \\ action = CONTAINSNOACTION \end{array}\right]}{\texttt{NP}} \rightarrow \underset{[\ number = a\ ]}{\texttt{N}}\ \underset{[\ ]}{\texttt{ADJP}}$$

$$\underset{\left[\begin{array}{l} number = SINGULAR \\ determined = HASNODETERMINER \\ citation = ISNOCITATION \\ action = CONTAINSNOACTION \end{array}\right]}{\texttt{NP}} \rightarrow \underset{[\ number = SINGULAR\ ]}{\texttt{N}}$$

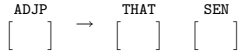
$$\underset{\left[\begin{array}{l} number = SINGULAR \\ determined = HASNODETERMINER \\ citation = ISNOCITATION \\ action = e \end{array}\right]}{\texttt{NP}} \rightarrow \underset{\left[\begin{array}{l} finitivity = GERUND \\ action = e \end{array}\right]}{\texttt{PVP}}$$

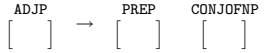
$$\underset{\left[\begin{array}{l} number = SINGULAR \\ determined = HASNODETERMINER \\ citation = ISNOCITATION \\ action = e \end{array}\right]}{\texttt{NP}} \rightarrow \underset{\left[\begin{array}{l} finitivity = INFINITE \\ action = e \end{array}\right]}{\texttt{PVP}}$$

$$\underset{\left[\begin{array}{l} number = PLURAL \\ reading = r \end{array}\right]}{\texttt{CONJOFNP}} \rightarrow \underset{[\ ]}{\texttt{NP}}\ \underset{[\ ]}{\texttt{SEP}}\ \underset{\left[\begin{array}{l} number = PLURAL \\ reading = r \end{array}\right]}{\texttt{CONJOFNP}}$$


CONJOFNP
number = PLURAL
reading = r
NP
AND
reading = r
NP

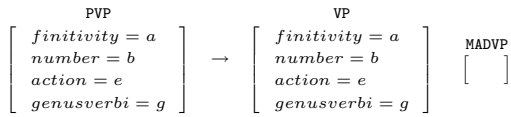

CONJOFNP
number = a
NP
number = a

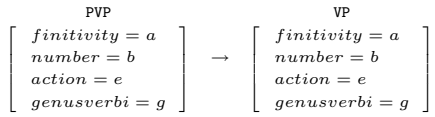

ADJP
THAT
SEN

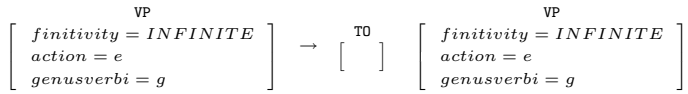

ADJP
PREP
CONJOFNP

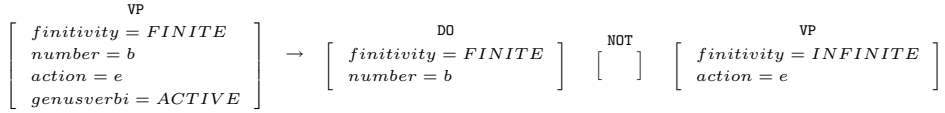

ADJP
X
PVP
finitivity = GERUND
X

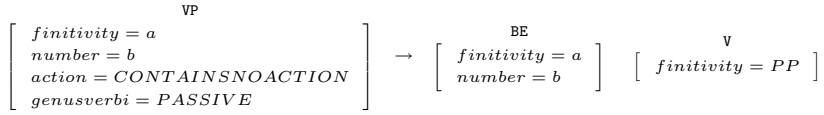

ADJP
PVP
finitivity = GERUND

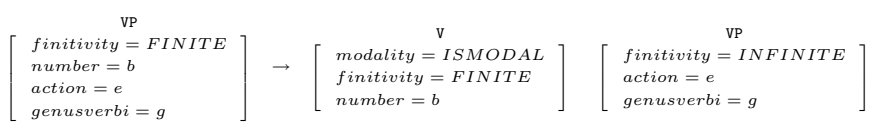

PVP
finitivity = a
number = b
action = e
genusverbi = g
VP
finitivity = a
number = b
action = e
genusverbi = g
MADVP

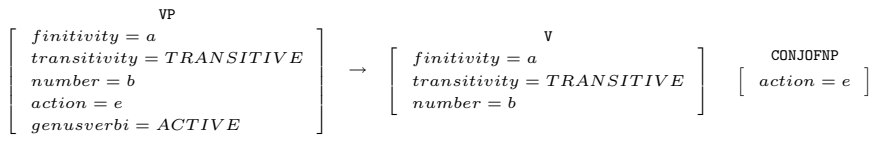

PVP
finitivity = a
number = b
action = e
genusverbi = g
VP
finitivity = a
number = b
action = e
genusverbi = g


VP
finitivity = INFINITE
action = e
genusverbi = g
TO
VP
finitivity = INFINITE
action = e
genusverbi = g


VP
finitivity = FINITE
number = b
action = e
genusverbi = ACTIVE
DO
finitivity = FINITE
number = b
NOT
VP
finitivity = INFINITE
action = e


VP
finitivity = a
number = b
action = CONTAINSNOACTION
genusverbi = PASSIVE
BE
finitivity = a
number = b
V
finitivity = PP


VP
finitivity = FINITE
number = b
action = e
genusverbi = g
V
modality = ISMODAL
finitivity = FINITE
number = b
VP
finitivity = INFINITE
action = e
genusverbi = g


VP
finitivity = a
transitivity = TRANSITIVE
number = b
action = e
genusverbi = ACTIVE
V
finitivity = a
transitivity = TRANSITIVE
number = b
CONJOFNP
action = e

$$\begin{bmatrix} \text{VP} \\ finitivity = a \\ transitivity = INTRANSITIVE \\ number = b \\ action = CONTAINSNOACTION \\ genusverbi = ACTIVE \end{bmatrix} \rightarrow \begin{bmatrix} \text{V} \\ modality = ISNOTMODAL \\ finitivity = a \\ transitivity = INTRANSITIVE \\ number = b \end{bmatrix}$$

$$\begin{bmatrix} \text{ADVP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{SUCH} \end{bmatrix} \begin{bmatrix} \text{THAT} \end{bmatrix} \begin{bmatrix} \text{SEN} \end{bmatrix}$$

$$\begin{bmatrix} \text{ADVP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{BY} \end{bmatrix} \begin{bmatrix} \text{CONJOFNP} \end{bmatrix}$$

$$\begin{bmatrix} \text{ADVP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{MPREP} \end{bmatrix} \begin{bmatrix} \text{CONJOFNP} \end{bmatrix}$$

$$\begin{bmatrix} \text{MADVP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{ADVP} \end{bmatrix} \begin{bmatrix} \text{SEP} \end{bmatrix} \begin{bmatrix} \text{MADVP} \end{bmatrix}$$

$$\begin{bmatrix} \text{MADVP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{ADVP} \end{bmatrix} \begin{bmatrix} \text{AND} \end{bmatrix} \begin{bmatrix} \text{ADVP} \end{bmatrix}$$

$$\begin{bmatrix} \text{MADVP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{ADVP} \end{bmatrix} \begin{bmatrix} \text{ADVP} \end{bmatrix}$$

$$\begin{bmatrix} \text{MADVP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{ADVP} \end{bmatrix}$$

$$\begin{bmatrix} \text{MPREP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{PREP} \end{bmatrix} \begin{bmatrix} \text{PREP} \end{bmatrix}$$

$$\begin{bmatrix} \text{MPREP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{PREP} \end{bmatrix}$$

$$\begin{bmatrix} \text{D} \end{bmatrix} \rightarrow \begin{bmatrix} \text{LEX} \end{bmatrix}$$

$$\begin{bmatrix} \text{ADJ} \end{bmatrix} \rightarrow \begin{bmatrix} \text{LEX} \end{bmatrix}$$

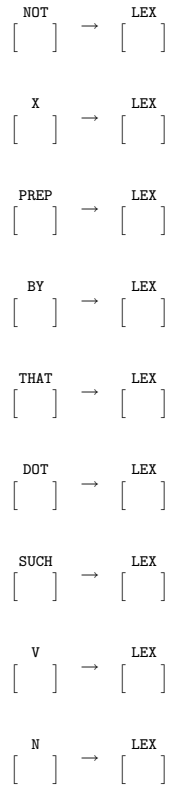
$$\begin{bmatrix} \text{TO} \end{bmatrix} \rightarrow \begin{bmatrix} \text{LEX} \end{bmatrix}$$

$$\begin{bmatrix} \text{SEP} \end{bmatrix} \rightarrow \begin{bmatrix} \text{LEX} \end{bmatrix}$$

$$\begin{bmatrix} \text{AND} \end{bmatrix} \rightarrow \begin{bmatrix} \text{LEX} \end{bmatrix}$$

$$\begin{bmatrix} \text{DO} \end{bmatrix} \rightarrow \begin{bmatrix} \text{LEX} \end{bmatrix}$$

$$\begin{bmatrix} \text{BE} \end{bmatrix} \rightarrow \begin{bmatrix} \text{LEX} \end{bmatrix}$$



A.5 Reihenfolge der Konstruktionsregeln

Der folgende Abschnitt enthält die in der Grammatikdatei definierte Reihenfolge der Konstruktionsregeln. Mehrfachnennungen von Konstruktionsregeln sind hierbei erlaubt. In dieser Reihenfolge versucht das erstellte Programm, die Konstruktionsregeln auf einen Syntaxbaum anzuwenden. Eine Konstruktionsregel wird dabei angewandt, solange ihre auslösende Struktur im Syntaxbaum gefunden wird. Ist das Programm einmal am Ende der Konstruktionsregelliste angelangt, beginnt die Suche nach anwendbaren Konstruktionsregeln wieder am Anfang. Die Anwendung terminiert, wenn keine Konstruktionsregel mehr anwendbar ist oder der Syntaxbaum einer DRS komplett reduziert wurde.

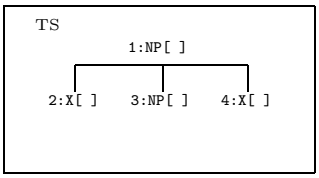
1. CR_CITATION1
2. CR_CITATION2
3. CR_NOUN1
4. CR_NOUNPHRASE1
5. CR_NOUNPHRASE2
6. CR_CONJOFNP1
7. CR_CONJOFNP2
8. CR_CONJOFNP3
9. CR_CONJOFNP4
10. CR_CONJOFNP5
11. CR_DETERMINER1

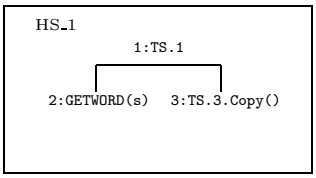
12. CR_DETERMINER2
13. CR_ADJECTIVE1
14. CR_ADJECTIVEPHRASE1
15. CR_VERBALNP1
16. CR_NOUN1
17. CR_NOUNPHRASE1
18. CR_NOUNPHRASE2
19. CR_CONJOFNP1
20. CR_CONJOFNP2
21. CR_CONJOFNP3
22. CR_CONJOFNP4
23. CR_CONJOFNP5
24. CR_DETERMINER1
25. CR_DETERMINER2
26. CR_ADJECTIVE1
27. CR_ADJECTIVEPHRASE1
28. CR_VERBALADJPHRASE1
29. CR_VERBALADJPHRASE2
30. CR_THATADJ1
31. CR_SUCHTHAT1
32. CR_NOUN1
33. CR_NOUNPHRASE1
34. CR_NOUNPHRASE2
35. CR_CONJOFNP1
36. CR_CONJOFNP2
37. CR_CONJOFNP3
38. CR_CONJOFNP4
39. CR_CONJOFNP5
40. CR_DETERMINER1
41. CR_DETERMINER2
42. CR_ADJECTIVE1
43. CR_ADJECTIVEPHRASE1
44. CR_THATADJ1
45. CR_NOUN1
46. CR_NOUNPHRASE1
47. CR_NOUNPHRASE2
48. CR_CONJOFNP1
49. CR_CONJOFNP2
50. CR_CONJOFNP3
51. CR_CONJOFNP4
52. CR_CONJOFNP5
53. CR_DETERMINER1
54. CR_DETERMINER2
55. CR_ADJECTIVE1
56. CR_ADJECTIVEPHRASE1
57. CR_VERBALNP2
58. CR_NOUN1
59. CR_NOUNPHRASE1
60. CR_NOUNPHRASE2
61. CR_CONJOFNP1
62. CR_CONJOFNP2
63. CR_CONJOFNP3
64. CR_CONJOFNP4
65. CR_CONJOFNP5
66. CR_DETERMINER1
67. CR_DETERMINER2

68. CR_ADJECTIVE1
69. CR_ADJECTIVEPHRASE1
70. CR_NOUN1
71. CR_NOUNPHRASE1
72. CR_NOUNPHRASE2
73. CR_CONJOFNP1
74. CR_CONJOFNP2
75. CR_CONJOFNP3
76. CR_CONJOFNP4
77. CR_CONJOFNP5
78. CR_DETERMINER1
79. CR_DETERMINER2
80. CR_ADJECTIVE1
81. CR_ADJECTIVEPHRASE1
82. CR_SUCHTHAT1
83. CR_EVENT1
84. CR_ADVERBIALPHRASE1
85. CR_ADVERBIALBY1
86. CR_BYSUBJECT1
87. CR_MULTIPREP1
88. CR_MULTIPREP2
89. CR_ADVERBIALPHRASE1
90. CR_MULTIADVP1
91. CR_MULTIADVP2
92. CR_MULTIADVP3
93. CR_MULTIADVP4
94. CR_MULTIADVP5
95. CR_MULTIADVP6
96. CR_ADVERBIALPHRASE1
97. CR_SENTENCE3
98. CR_PSENTENCE1
99. CR_PSENTENCE2
100. CR_PSENTENCE5
101. CR_SENTENCE5
102. CR_SENTENCE6
103. CR_SENTENCE1
104. CR_SENTENCE2
105. CR_CLEAN2
106. CR_CLEAN1

A.6 Konstruktionsregeln

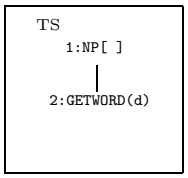
CR_CITATION1

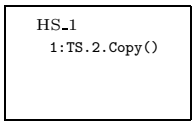




ACTION
s = getSubTreeString(TS.3)
substitute(HS_1, TS.1)

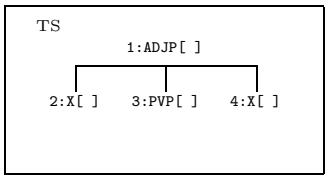
CR_NOUNPHRASE1

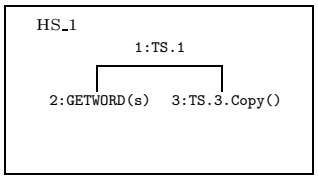




ACTION
substitute(HS_1, TS.1)

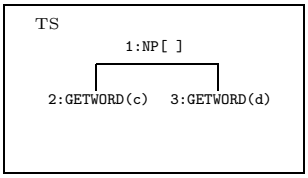
CR_CITATION2

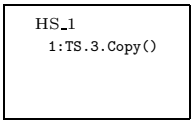




ACTION
s = getSubTreeString(TS.3)
substitute(HS_1, TS.1)

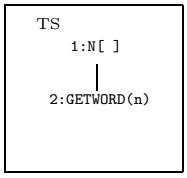
CR_NOUNPHRASE2

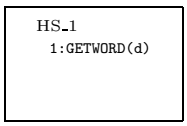




ACTION
addNewName(c, d)
substitute(HS_1, TS.1)

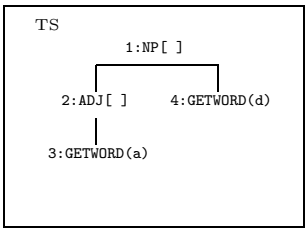
CR_NOUN1

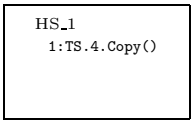




ACTION
d = addNewLocalDR(TS.2)
addNewLocalAttribute(n, d)
substitute(HS_1, TS.1)

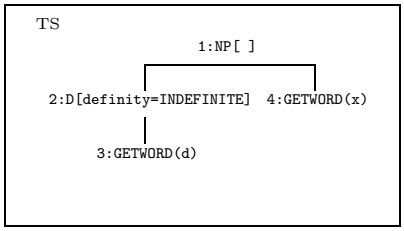
CR_ADJECTIVE1

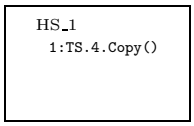




ACTION
addNewLocalAttribute(a, d)
substitute(HS_1, TS.1)

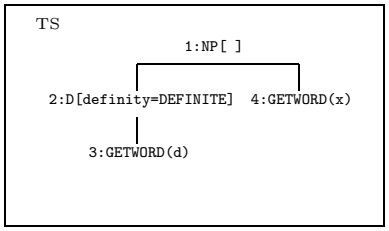
CR_DETERMINER1

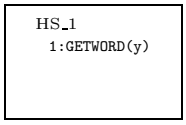




ACTION
substitute(HS_1, TS.1)

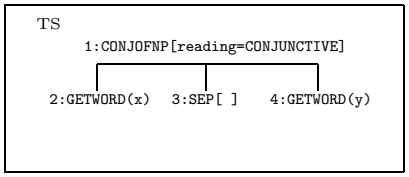
CR_DETERMINER2

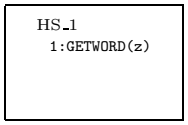




ACTION
y = getDefAntecedent(TS.4)
substitute(HS_1, TS.1)

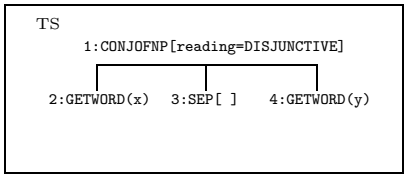
CR_CONJOFNP4

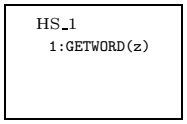




ACTION
s = getString(CONJUNCTIVE)
z = addNewLocalReferentSum(x)
addNewLocalAttribute(s, z)
substitute(HS_1, TS.1)

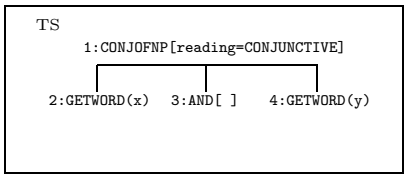
CR_CONJOFNP5

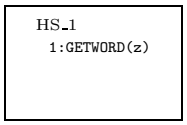




ACTION
s = getString(DISJUNCTIVE)
z = addNewLocalReferentSum(x)
addNewLocalAttribute(s, z)
substitute(HS_1, TS.1)

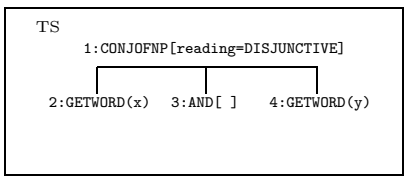
CR_CONJOFNP3

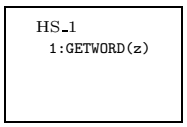




ACTION
s = getString(CONJUNCTIVE)
z = addNewLocalReferentSum(x)
addNewLocalAttribute(s, z)
substitute(HS_1, TS.1)

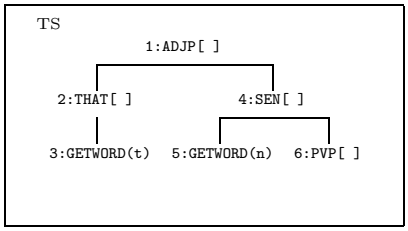
CR_CONJOFNP2

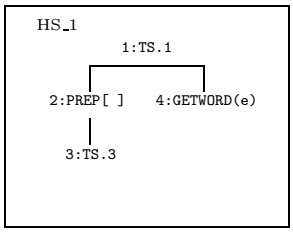


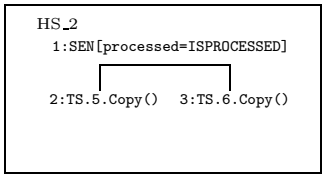


ACTION
s = getString(DISJUNCTIVE)
z = addNewLocalReferentSum(x)
addNewLocalAttribute(s, z)
substitute(HS_1, TS.1)

CR_THATADJ1

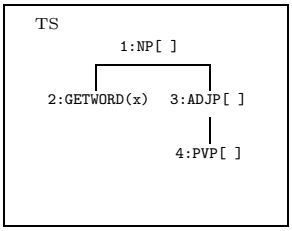


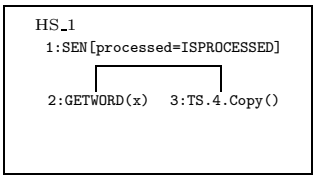


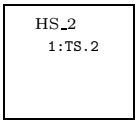


ACTION
e = getNextFreeEventReferent()
d = buildDRS(HS_2, POSITIVE)
addLocalEventReferent(e, d)
substitute(HS_1, TS.1)

CR_VERBALADJPHRASE1

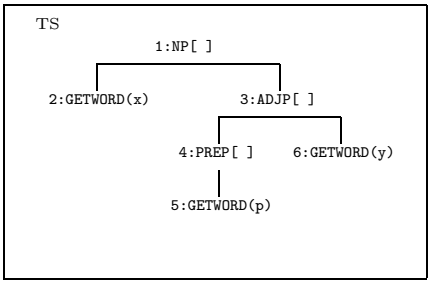


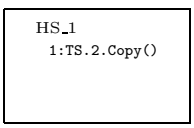




ACTION
e = getNextFreeEventReferent()
d = buildDRS(HS_1, POSITIVE)
addLocalEventReferent(e, d)
substitute(HS_2, TS.1)

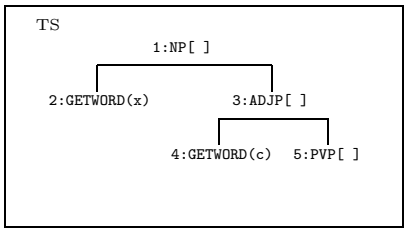
CR_ADJECTIVEPHRASE1

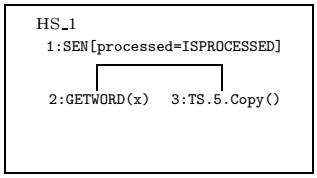


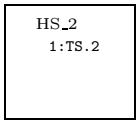


ACTION
addNewLocalPredicate(p, x)
substitute(HS_1, TS.1)

CR_VERBALADJPHRASE2

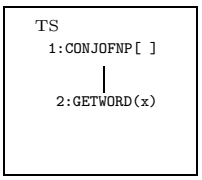


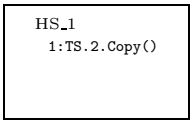




ACTION
e = getNextFreeEventReferent()
d = buildDRS(HS_1, POSITIVE)
addLocalEventReferent(e, d)
addNewName(c, e)
substitute(HS_2, TS.1)

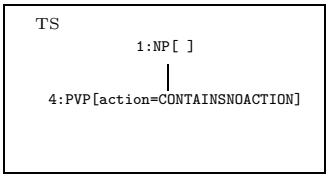
CR_CONJOFNP1

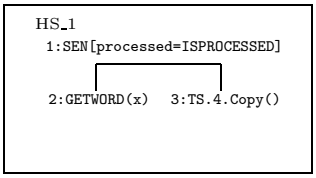


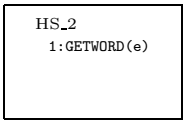


ACTION
substitute(HS_1, TS.1)

CR_VERBALNP1

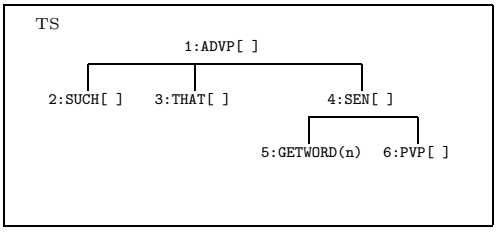


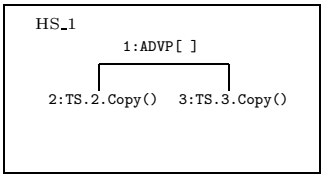


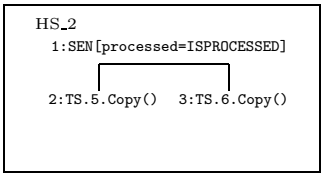


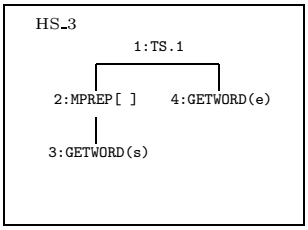
ACTION
e = getNextFreeEventReferent()
s = getString(SOMEONE)
x = addNewGlobalDR(TS.1)
addNewGlobalAttribute(s, x)
d = buildDRS(HS_1, POSITIVE)
addLocalEventReferent(e, d)
substitute(HS_2, TS.1)

CR_SUCHTHAT1



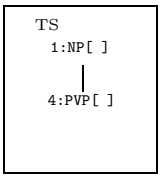


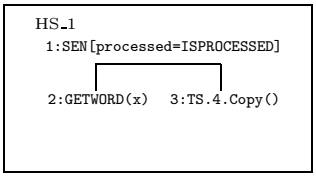


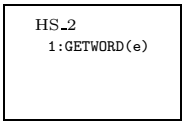


ACTION
s = getSubTreeString(HS_1)
e = getNextFreeEventReferent()
d = buildDRS(HS_2, POSITIVE)
addLocalEventReferent(e, d)
substitute(HS_3, TS.1)

CR_VERBALNP2

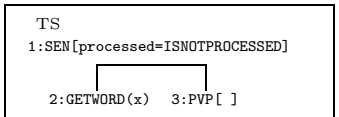


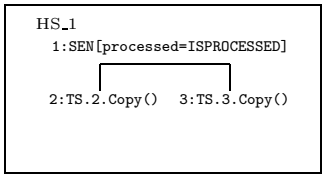


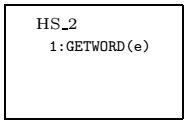


ACTION
e = getNextFreeEventReferent()
s = getString(SOMEONE)
x = addNewGlobalDR(TS.1)
addNewGlobalAttribute(s, x)
d = buildDRS(HS_1, POSITIVE)
addLocalEventReferent(e, d)
substitute(HS_2, TS.1)

CR_EVENT1

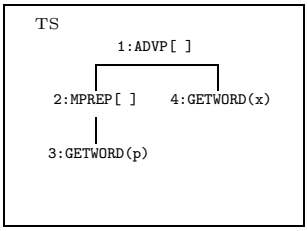


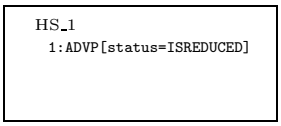




ACTION
e = getNextFreeEventReferent()
d = buildDRS(HS_1, POSITIVE)
addLocalEventReferent(e, d)
substitute(HS_2, TS.1)

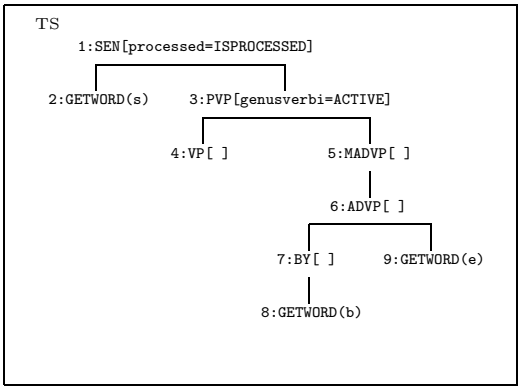
CR_ADVERBIALPHRASE1

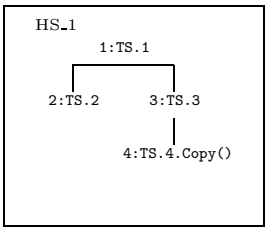




ACTION
addNewLocalPredicate(p, x)
substitute(HS_1, TS.1)

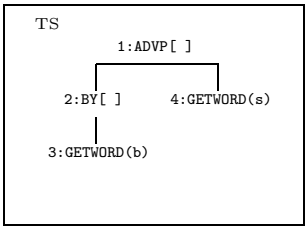
CR_ADVERBIALBY1

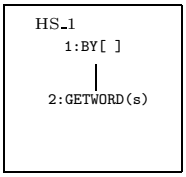




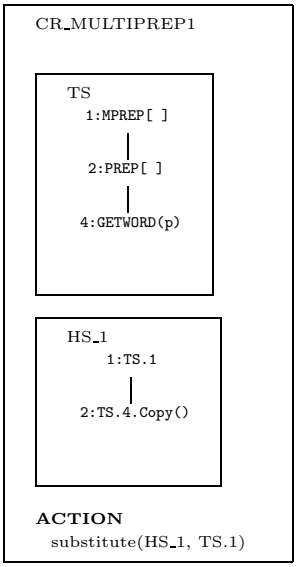
ACTION
addNewLocalPredicate(b, e)
substitute(HS_1, TS.1)

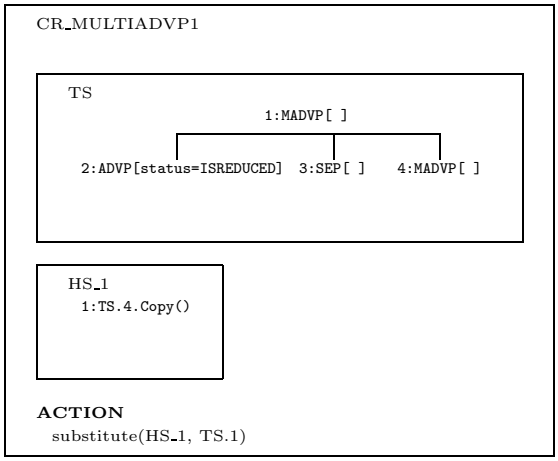
CR_BYSUBJECT1

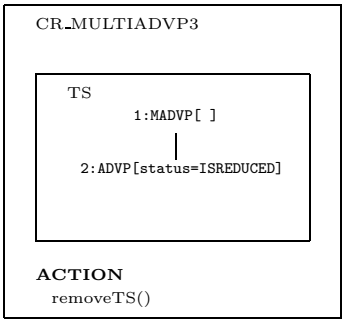


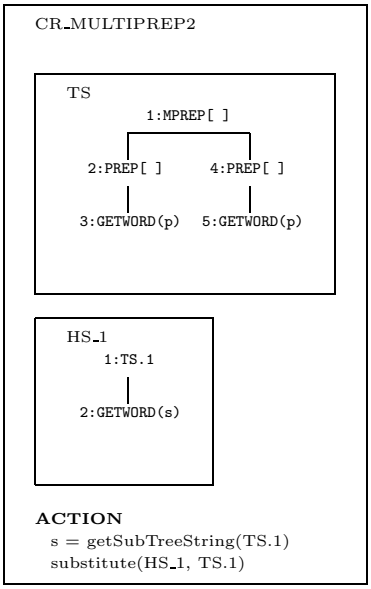


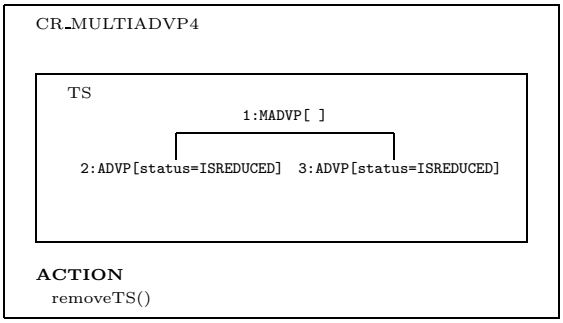
ACTION
substitute(HS_1, TS.1)

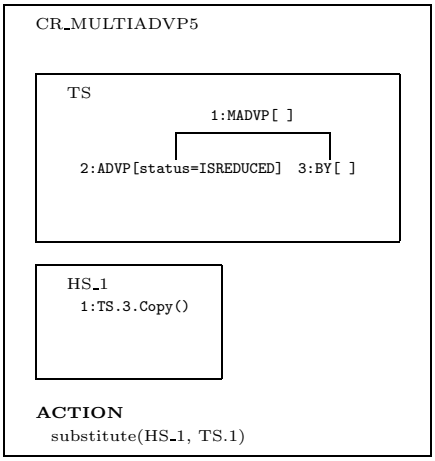

CR_MULTIPREP1
TS
1:MPREP[ ]
2:PREP[ ]
4:GETWORD(p)
HS_1
1:TS.1
2:TS.4.Copy()
ACTION
substitute(HS_1, TS.1)

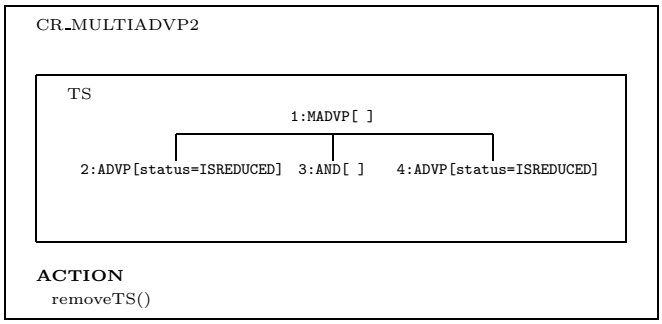

CR_MULTIADVP1
TS
1:MADVP[ ]
2:ADVP[status=ISREDUCED]
3:SEP[ ]
4:MADVP[ ]
HS_1
1:TS.4.Copy()
ACTION
substitute(HS_1, TS.1)


CR_MULTIADVP3
TS
1:MADVP[ ]
2:ADVP[status=ISREDUCED]
ACTION
removeTS()

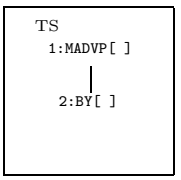

CR_MULTIPREP2
TS
1:MPREP[ ]
2:PREP[ ]
4:PREP[ ]
3:GETWORD(p)
5:GETWORD(p)
HS_1
1:TS.1
2:GETWORD(s)
ACTION
s = getSubTreeString(TS.1)
substitute(HS_1, TS.1)

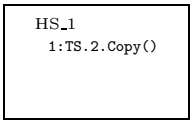

CR_MULTIADVP4
TS
1:MADVP[ ]
2:ADVP[status=ISREDUCED]
3:ADVP[status=ISREDUCED]
ACTION
removeTS()


CR_MULTIADVP5
TS
1:MADVP[ ]
2:ADVP[status=ISREDUCED]
3:BY[ ]
HS_1
1:TS.3.Copy()
ACTION
substitute(HS_1, TS.1)


CR_MULTIADVP2
TS
1:MADVP[ ]
2:ADVP[status=ISREDUCED]
3:AND[ ]
4:ADVP[status=ISREDUCED]
ACTION
removeTS()

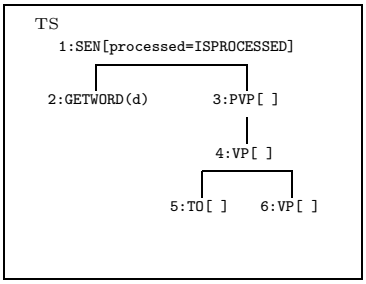
CR_MULTIADVP6

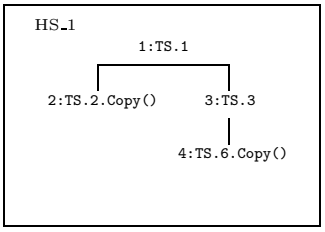




ACTION
substitute(HS_1, TS.1)

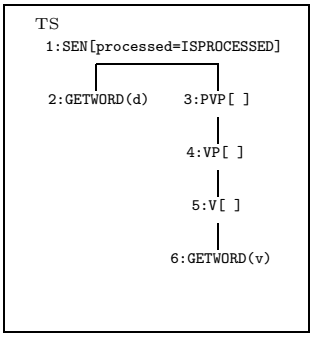
CR_SENTENCE3





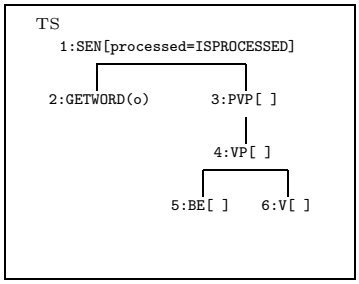
ACTION
substitute(HS_1, TS.1)

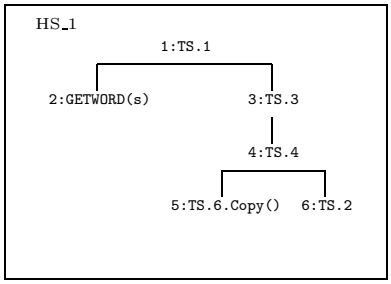
CR_SENTENCE1



ACTION
addNewLocalPredicate(v, d)
removeTS()

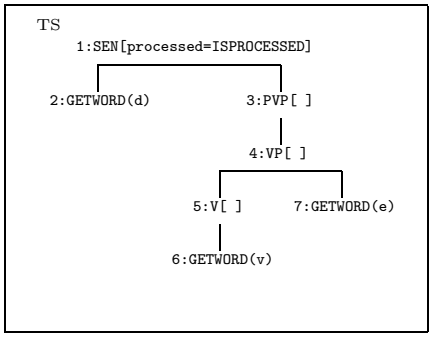
CR_PSENTENCE1





ACTION
s = addNewGlobalDR(TS.6)
p = getString(SOMEONE)
addNewGlobalAttribute(p, s)
substitute(HS_1, TS.1)

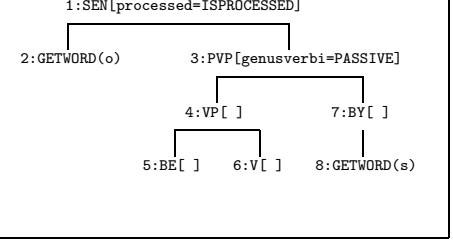
CR_SENTENCE2



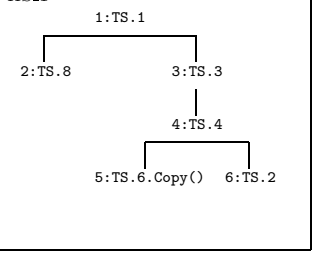
ACTION
addNewLocalPredicate(v, d)
removeTS()

CR_PSENTENCE2

TS



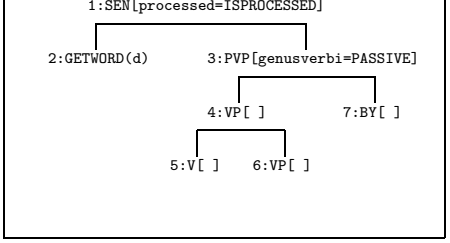
HS_1



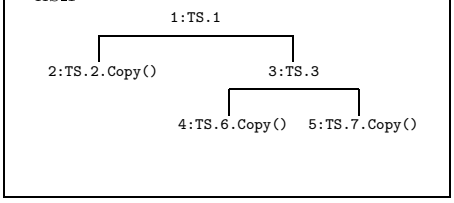
ACTION
substitute(HS_1, TS.1)

CR_PSENTENCE5

TS



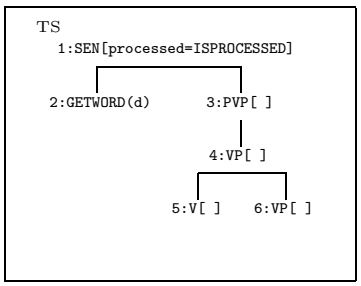
HS_1



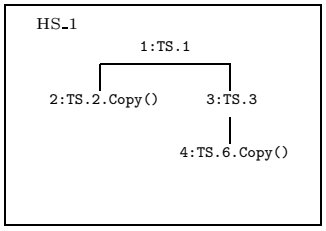
ACTION
substitute(HS_1, TS.1)

CR_SENTENCE5

TS



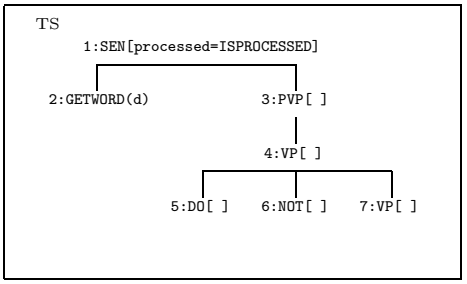
HS_1



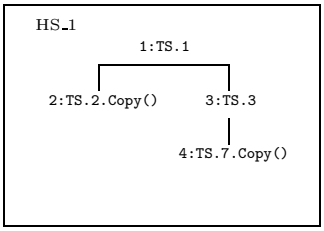
ACTION
substitute(HS_1, TS.1)

CR_SENTENCE6

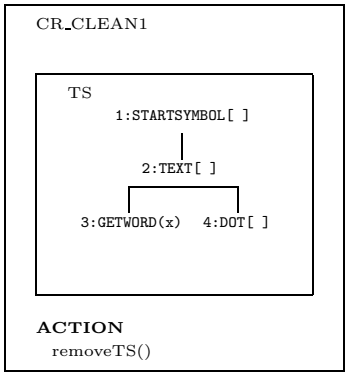
TS

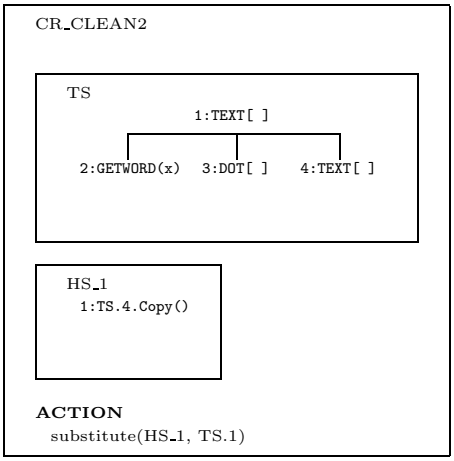


HS_1



ACTION
f = buildDRS(HS_1, NEGATIVE)
addNewLocalSubDRS(f)
removeTS()


CR_CLEAN1
TS
1:STARTSYMBOL[ ]
2:TEXT[ ]
3:GETWORD(x)
4:DOT[ ]
ACTION
removeTS()


CR_CLEAN2
TS
1:TEXT[ ]
2:GETWORD(x)
3:DOT[ ]
4:TEXT[ ]
HS_1
1:TS.4.Copy()
ACTION
substitute(HS_1, TS.1)

A.7 Morphologie

Im folgenden sind die Morphologieregeln der Grammatik aufgeführt. Dabei steht `cons` für einen Konsonaten und `vowel` für einen Vokal.

Morphologie für Wörter aus der Wortart N

Endungen auf “s” für die PLURAL Form

“o”	⟶	“oes”
“ss”	⟶	“sses”
“sh”	⟶	“shes”
“ch”	⟶	“ches”
“x”	⟶	“xes”
`vowel` “y”	⟶	`vowel` “ys”
`cons` “y”	⟶	`cons` “ies”
`vowel` “s”	⟶	`vowel` “sses”
“f”	⟶	“ves”
“fe”	⟶	“ves”

Morphologie für Wörter aus der Wortart V

Endungen auf “ed” für die PP Form

`vowel` “y”	⟶	`vowel` “yed”
`cons` “y”	⟶	`cons` “ied”
`cons1 vowel cons2`	⟶	`cons1 vowel cons2 cons2` “ed”
“e”	⟶	“ed”

Endungen auf “s” für die FINITE Form

`vowel` “y”	⟶	`vowel` “ys”
`cons` “y”	⟶	`cons` “ies”
`cons1 vowel cons2`	⟶	`cons1 vowel cons2 cons2` “es”
“e”	⟶	“es”

Endungen auf “ing” für die GERUND Form

“e”	⟶	“ing”
“ie”	⟶	“ying”
`cons1 vowel cons2`	⟶	`cons1 vowel cons2 cons2` “ing”

Anhang B

Beispieltext und DRS–Verzeichnis

B.1 Beispieltext

Der folgende Abschnitt enthält eine Verhaltensbeschreibung, an deren Beispiel in der vorliegenden Arbeit die Umsetzung in eine entsprechende Logikdarstellung untersucht wird. Der Originaltext stammt aus [Nagel 99] und ist zunächst in vier Abschnitte gegliedert (bezeichnet mit I bis IV). Desweiteren sind die in den einzelnen Abschnitten auftretenden Sätze numeriert (mit i,ii, ...). Dieser Text wurde zunächst umformuliert und vereinfacht, um bestimmte Schwierigkeiten bei der Entwicklung einer entsprechenden Grammatik und der zugehörigen Konstruktionsregeln zu vermeiden. Diese Umformulierungen und Vereinfachungen sind in der folgenden Liste aufgeführt.

1 Alle Prädikate der auftretenden Sätze wurden in die Präsensform (*simple present tense*) gestellt. Die dadurch entstehende Bedeutungsverschiebung wird als vernachlässigbar angesehen.

2 Adverbialphrasen wurden entfernt, sofern sie die zu erstellende Grammatik im Verhältnis zu dem dadurch gewonnenen Sprachumfang zu sehr verkompliziert hätten.

3 Zitate, die nur aus einem Verb bestehen, wurden in die im englischen gebräuchliche Infinitivform (*to ...*) gestellt.

4 Aufzählungen wurden dahingehend umformuliert, daß alle aufgezählten Phrasen von der gleichen Art sind. Dies dient der Vereinfachung der zu erstellenden Grammatik.

5 Relativsätze wurden eliminiert.

6 Einleitende Adverbialphrasen wurden hinter dem Prädikat des Satzes eingefügt.

7 In Klammern stehende genauere Beschreibungen von Nominalphrasen wurden in die Nominalphrase integriert.

8 Der Gebrauch von Zitaten wurde auf komplette Phrasen eingeschränkt. Dies bedeutet, daß sich keine Phrase des Satzes über eine Zitatgrenze hinweg erstrecken darf.

9 Der besitzanzeigende Genitiv wurde durch eine Präpositionalphrase ersetzt.

10 Da Aufzählungen von Nominalphrasen häufiger auftraten, die Aufzählung mehrerer Adverbialphrasen jedoch nur einmal vorkam, wurde auf diese Aufzählungsart verzichtet und die entsprechende Textstelle umformuliert.

11 Adverbien und Adverbialphrasen, welche die zeitliche Abfolge von Aktionen betonen, die sich auch aus dem Satz heraus erschliessen läßt, wurden entfernt (*then*, *further*, *here*).

12 Einige zusammengesetzte Begriffe, die zwar aus Teilphrasen bzw. mehreren Wörtern bestehen, jedoch hier nur in dieser Form auftreten, wurden zu feststehenden Begriffen und durch die Verbindung mit Unterstrichen zu Wörtern der Sprache erklärt (*stop_and_go*).

Die geänderten Stellen im Originaltext sind mit entsprechenden Indizes versehen. Der resultierende Text, von dem in allen folgenden Arbeitsschritten ausgegangen wurde, ist in Abschnitt B.1.2 zu finden. Da durch die Umformulierungen teilweise auch neue Sätze entstanden, wurden diese mit arabischen Ziffern neu numeriert.

B.1.1 Originaltext

(I,i) The abstract statement about a vehicle 'crossing an intersection' will be concretized[1] *in a first step by decomposing 'to cross' into a concatenation of 'to drive to an intersection', 'to drive across an intersection', and 'to drive away from an intersection'. (I,ii) The notion of 'leaving an intersection' will be used*[1] *as a synonym for the last specialization of 'to drive'.*
(II,i) 'Driving' may be decomposed in a second step[2] *into a repeated concatenation of notions from a set which*[5] *comprises 'decelerate*[3]*', 'stop and go*[12]*', 'accelerate*[3]*', and 'proceed*[3]*'. (II,ii) Here*[11]*, 'stop and go*[12]*' stands for a composition of 'stop*[3]*', followed by*[4] *an optional 'wait*[3]*', and then*[11] *by a 'start to move*[12]*'.*
(III,i) In a third step[6]*, 'driving' can be specialized by the requirement that the agent 'drives behind'*[8] *another vehicle (called patient)*[7]*. (III,ii) The patient vehicle should 'drive in front of' and 'in the same direction' as the agent, 'without any vehicle between agent and patient', and the patient being 'near' the agent*[4,8]*. (III,iii) The agent's velocity*[9] *will have to be controlled*[1] *in this case such that a minimum distance between agent and patient will be preserved*[1]*.*

(IV,i) The last concretization step[12] specializes 'to drive' further[11] by adding the requirement that 'to drive' is restricted to either a 'left-turning lane[12]', to a 'straigh ahead lane[12]', or to a 'right-turning lane[12]'[10]. (IV,ii) This restriction of driving on a directional lane does not apply to 'leaving an intersection'.

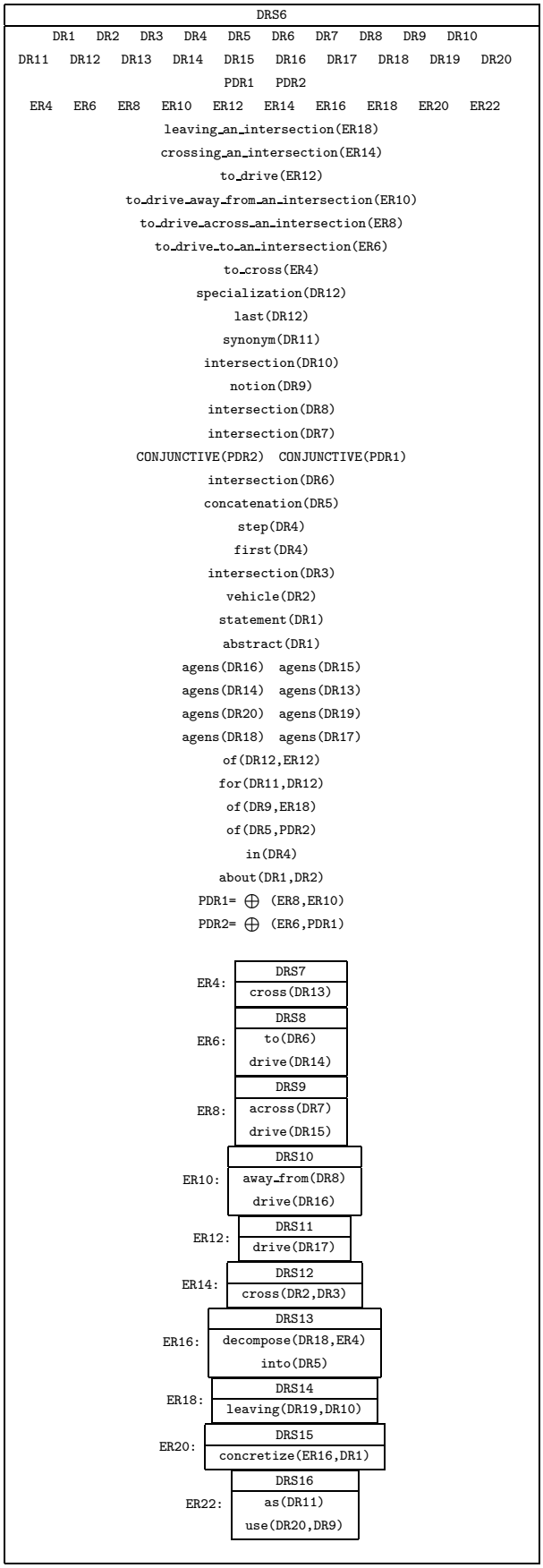
B.1.2 Abgeänderter Text

(I,1) The abstract statement about a vehicle 'crossing an intersection' is concretized in a first step by decomposing 'to cross' into a concatenation of 'to drive to an intersection', 'to drive across an intersection' and 'to drive away from an intersection'. (I,2) The notion of 'leaving an intersection' is used as a synonym for the last specialization of 'to drive'.
(II,1) 'Driving' may be decomposed into a repeated concatenation of notions from the set consisting of 'to decelerate', 'stop_and_go', 'to accelerate' and 'to proceed'. (II,2) 'stop_and_go' stands for a composition of 'to stop', an optional 'to wait' and a 'start_to_move'.
(III,1) 'Driving' can be specialized in a third step by the requirement that the agent drives behind a patient vehicle. (III,2) The patient vehicle should drive in front of the agent, in the same direction as the agent and without any vehicle between the agent and the patient vehicle. (III,3) The patient vehicle should be near the agent. (III,4) The velocity of the agent must be controlled in this case such that a minimum distance between agent and the patient vehicle is preserved.
(IV,1) The last concretization_step specializes 'driving' by adding the requirement that 'driving' is restricted to a 'left_turning_lane', a 'straigh_ahead_lane' or a 'right_turning_lane'. (IV,2) This restriction of 'driving on a directional lane' does not apply to 'leaving an intersection'.

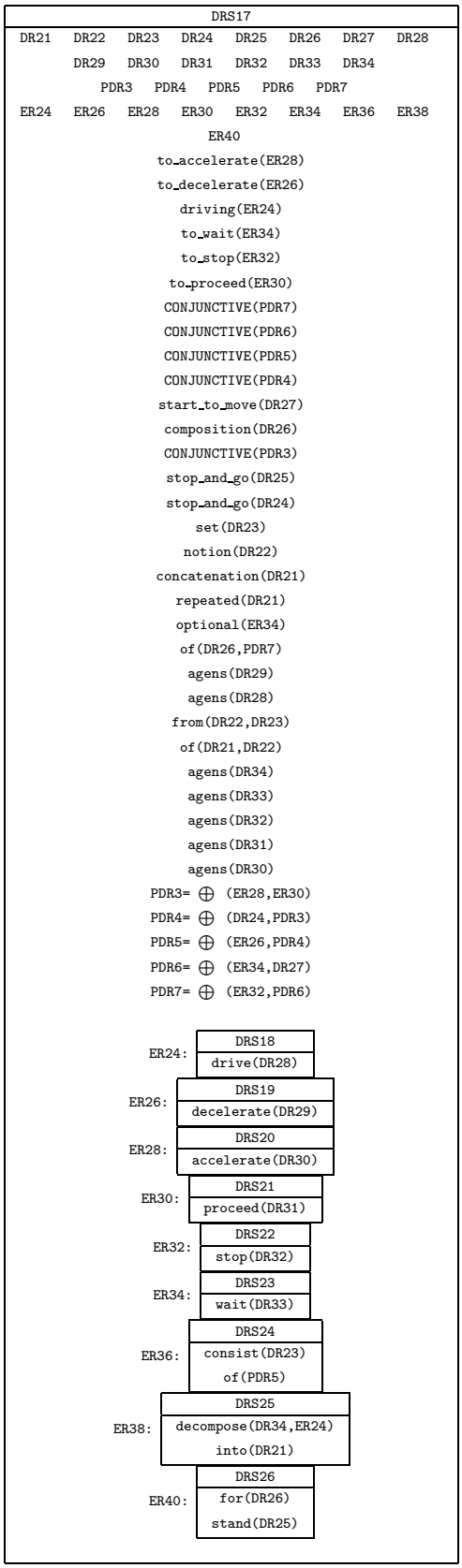
B.2 DRS–Verzeichnis

In den folgenden Abschnitten sind die mit dem Programm von [Arens & Ottlik 2000] aus den Abschnitten I bis IV des Beispieltextes erstellten DRSen aufgeführt. Die hier präsentierte Form der DRSen wurde mit dem Programm automatisch erstellt. Lediglich die DRS des Abschnittes I wurde aus Platzgründen nachbearbeitet.

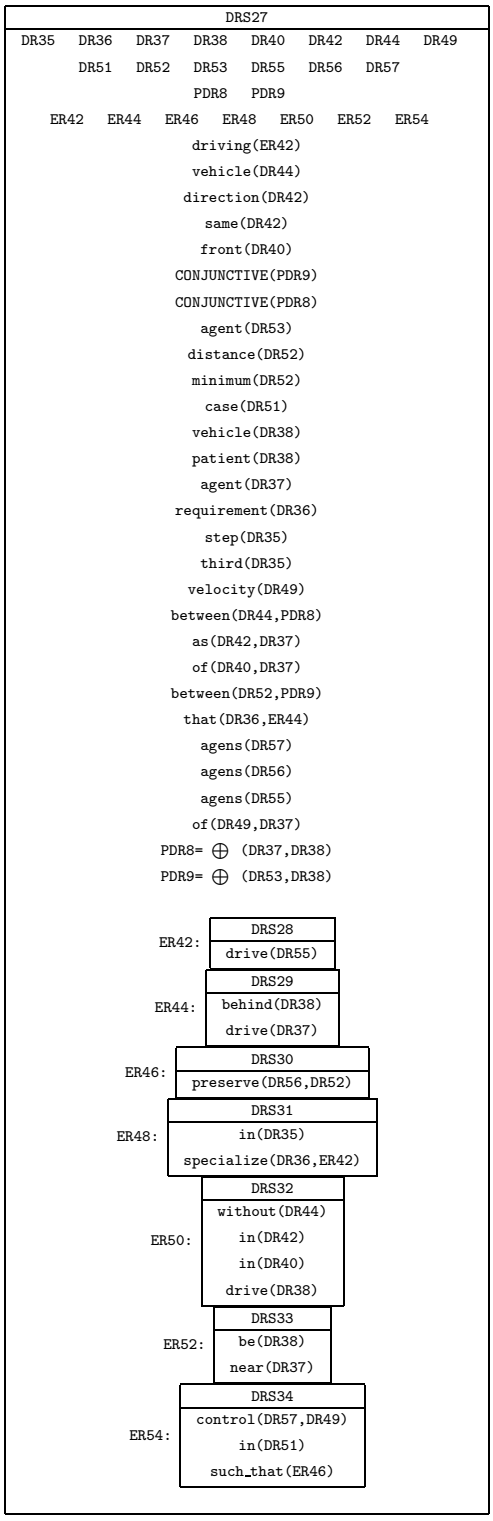
B.2.1 DRS zu Absatz I



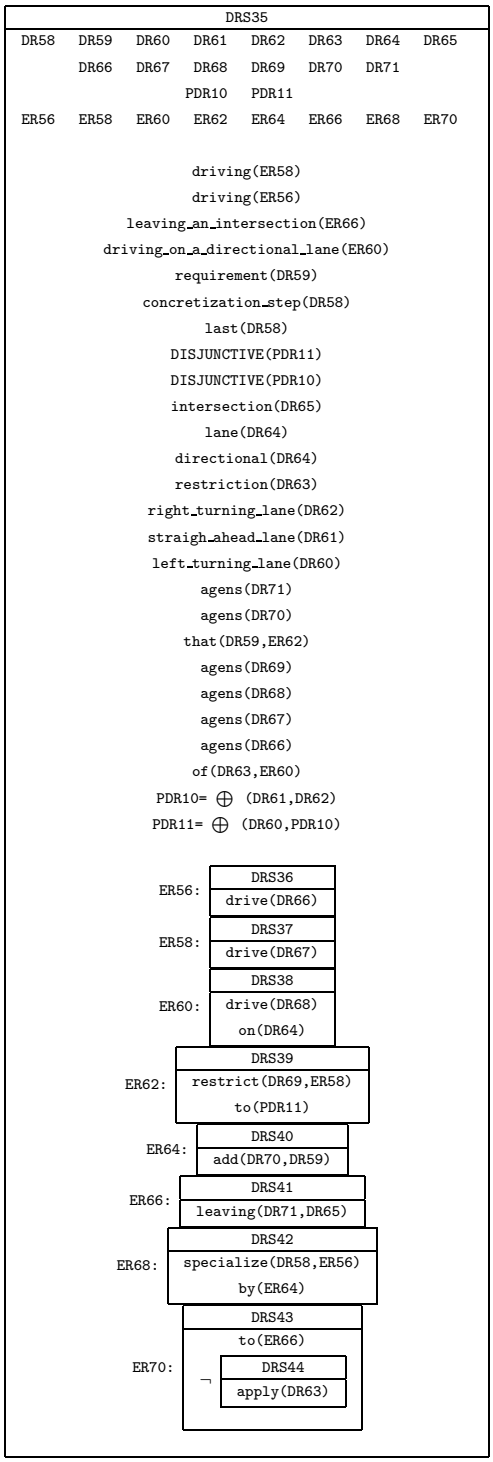
B.2.2 DRS zu Absatz II



B.2.3 DRS zu Absatz III



B.2.4 DRS zu Absatz IV



Anhang C

DRS–Transformator–Dokumentation

Die folgenden Abschnitte enthalten die mit dem Werkzeug zur DRS–Transformation teilweise automatisch erstellte Dokumentation. Abschnitt C.1 dokumentiert die im DRS–Transformator definierte Reihenfolge der TRn. Mehrfachnennungen einzelner TRn sind hierbei erlaubt. Abschnitt C.2 stellt die verwendeten TRn mit ihren Muster–DRSen und Aktionsteilen in übersichtlicher Form dar. Abschnitt C.3 enthält das in der vorliegenden Arbeit verwendete Bedeutungs–Wörterbuch. Die Dokumentation des Transformators wird mit einer Aufstellung der neu definierten Aktionsteilmethoden abgeschlossen.

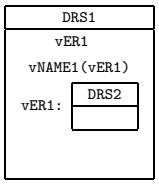
C.1 Reihenfolge der Transformations–Regeln

1. TR_SITMATCH1
2. TR_PDRCOLLECT1
3. TR_PDRCOLLECT2
4. TR_PREDPOP1
5. TR_PREDPOP2
6. TR_PREDPOP3
7. TR_PREDPOP4
8. TR_PREDPOP5
9. TR_DICTPRED2
10. TR_DICTPRED3
11. TR_DICTPRED4
12. TR_DICTPRED1
13. TR_DICTPRED2
14. TR_DICTPRED3
15. TR_DICTPRED4
16. TR_DICTPRED1
17. TR_PDRCOLLECT1
18. TR_PDRCOLLECT2
19. TR_SITVARS1
20. TR_WRITESTATE1
21. TR_WRITEACTION1
22. TR_WRITEPREDICT1
23. TR_WRITEPREDICT2
24. TR_WRITESPEC1
25. TR_WRITEEVAL1

C.2 Tranformationsregeln

TR_SITMATCH1

PatternDRS:

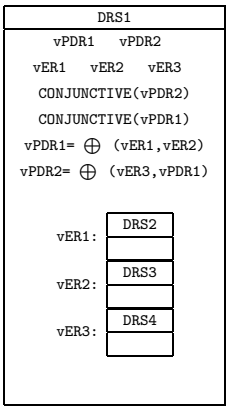


Actions:

```
collectDRKnowledge(DRS2);
addNewAttribute(SIT(vER1));
addNewAttribute(EVAL(vER1));
d = DRS2.copy();
e = new DRS();
e.addNewName(vNAME1(vER1));
addNewSubDRS(e);
addNewSubDRS(d);
addNewPredicate(SITSTATE(vER1,d));
addNewPredciate(SITNAME(vER1,e));
addNewPredciate(PREDICT(vER1,vER1));
removePredicate(vNAME(vER1));
```

TR_PDRCOLLECT1

PatternDRS:

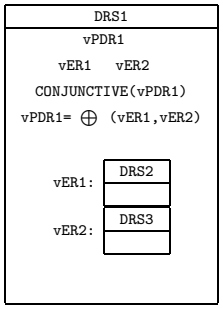


Actions:

```
removePredicate(CONJUNCTIVE(vPDR1));
removePredicate(CONJUNCTIVE(vPDR2));
removeReferentSum(vPDR1);
removeReferentSum(vPDR2);
addReferentSum(vPDR2,vER3,vER1,vER2));
addNewAttribute(SITLIST(vPDR2));
```

TR_PDRCOLLECT2

PatternDRS:

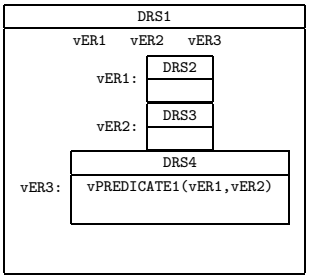


Actions:

```
removePredicate(CONJUNCTIVE(vPDR1));
removeReferentSum(vPDR1);
addReferentSum(vPDR1,vER1,vER2));
addNewAttribute(SITLIST(vPDR1));
```

TR_PREDPOP1

PatternDRS:

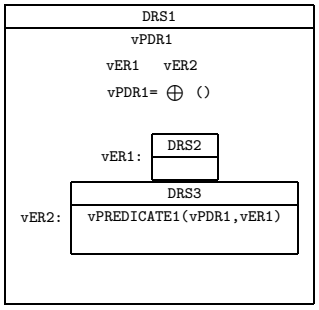


Actions:

```
DRS4.removePredicate(vPREDICATE1(vER1,vER2));
addNewPredicate(vPREDICATE1(vER3,vER1,vER2));
```

TR_PREDPOP2

PatternDRS:

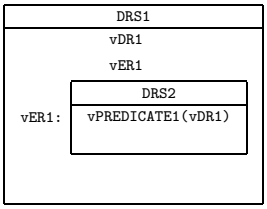


Actions:

```
DRS3.removePredicate(vPREDICATE1(vPDR1,vER1));
addNewPredicate(vPREDICATE1(vER2,vPDR1,vER1));
```

TR_PREDPOP4

PatternDRS:

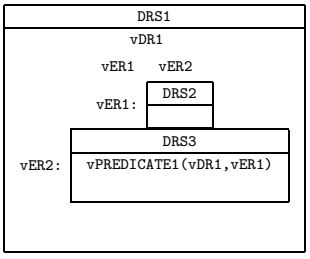


Actions:

```
DRS2.removePredicate(vPREDICATE1(vDR1));
addNewPredicate(vPREDICATE1(vER1,vDR1));
```

TR_PREDPOP3

PatternDRS:

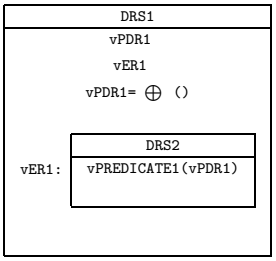


Actions:

```
DRS3.removePredicate(vPREDICATE1(vDR1,vER1));
addNewPredicate(vPREDICATE1(vER2,vDR1,vER1));
```

TR_PREDPOP5

PatternDRS:

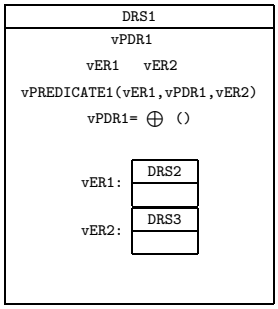


Actions:

```
DRS2.removePredicate(vPREDICATE1(vPDR1));
addNewPredicate(vPREDICATE1(vER1,vPDR1));
```

TR_DICTPRED2

PatternDRS:

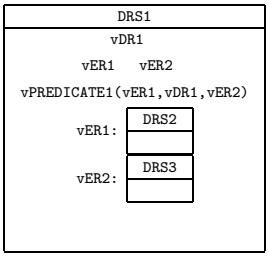


Actions:

```
callActions(vPREDICATE1(vER1,vPDR1,vER2));
removePredicate(vPREDICATE1(vER1,vPDR1,vER2));
```

TR_DICTPRED4

PatternDRS:

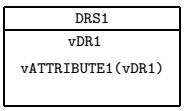


Actions:

```
callActions(vPREDICATE1(vER1,vDR1,vER2));
removePredicate(vPREDICATE1(vER1,vDR1,vER2));
```

TR_DICTPRED3

PatternDRS:

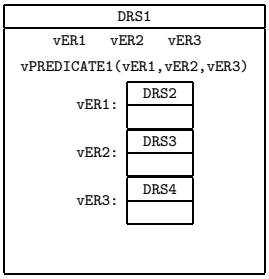


Actions:

```
callActions(vATTRIBUTE1(vDR1));
removePredicate(vATTRIBUTE1(vDR1));
```

TR_DICTPRED1

PatternDRS:

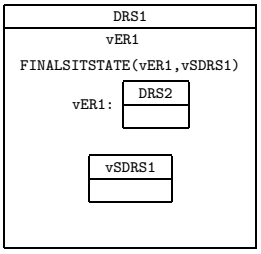


Actions:

```
callActions(vPREDICATE1(vER1,vER2,vER3));
removePredicate(vPREDICATE1(vER1,vER2,vER3));
```

TR_SITVARS1

PatternDRS:

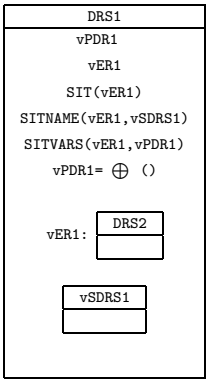


Actions:

```
p = addNewReferentSum(vSDRS1.getAllUsedDRs());
removePredicate(FINALSTATE(vER1,vSDRS1));
addNewPredicate(SITVARS(vER1,p));
removeSubDRS(vSDRS1);
```

TR_WRITEACTION1

PatternDRS:

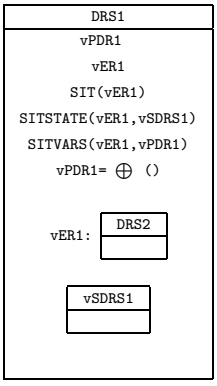


Actions:

```
addSituationAction(vER1,vPDR1.getAllReferents(),vDRS1);
removePredicate(SITNAME(vER1,vDRS1));
```

TR_WRITESTATE1

PatternDRS:

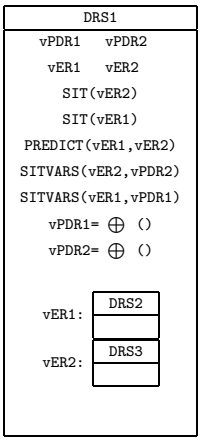


Actions:

```
addSituationState(vER1,vPDR1.getAllReferents(),vDRS1);
removePredicate(SITSTATE(vER1,vDRS1));
```

TR_WRITEPREDICT1

PatternDRS:

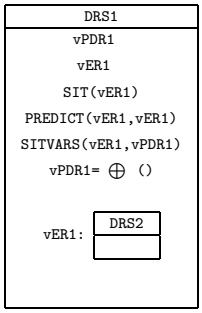


Actions:

```
addSituationPrediction(vER1,vPDR1.getAllReferents()
,vER2,vPDR2.getAllReferents());
removePredicate(PREDICT(vER1,vER2)));
```

TR_WRITEPREDICT2

PatternDRS:

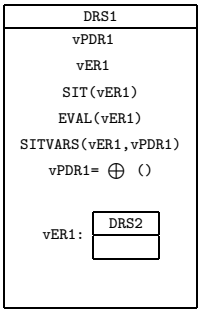


Actions:

```
addSituationPrediction(vER1,vPDR1.getAllReferents()
,vER1,vPDR1.getAllReferents());
removePredicate(PREDICT(vER1,vER1)));
```

TR_WRITEEVAL1

PatternDRS:

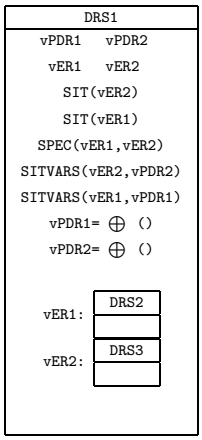


Actions:

```
addSituationEvaluation(vER1,vPDR1.getAllReferents());
removePredicate(EVAL(vER1);
```

TR_WRITESPEC1

PatternDRS:



Actions:

```
addSituationSpecialization(vER1,vPDR1.getAllReferents()
,vER2,vPDR2.getAllReferents());
removePredicate(SPEC(vER1,vER2));
```

C.3 Das verwendete Bedeutungs–Wörterbuch

In diesem Abschnitt ist die in der vorliegenden Arbeit verwendete Bedeutungs–Wörterbuchdatei abgebildet. Die Datei wird mit dem reservierten Wort `DICTIONARY` eingeleitet, das Ende der Datei markiert das Symbol `*`. Jede verwendete Bedingung im Wörterbuch muß typisiert werden, was durch die reservierten Wörter `PRED` (für Prädikate) sowie `ATTR`, `NAME` und `RELA` (entsprechend für Attribute, Namen und Relationen) geschieht. Der Typ einer Bedingung und die Bedingung selbst sind durch einen Doppelpunkt (`:`) zu trennen.

```
DICTIONARY

PRED:specialize(vER 1, vER 2, vER 3) {
  PRED:SITSTATE(vER 2, vSDRS 1) {
    PRED:SITSTATE(vER 3, vSDRS 2) {
      [ addSpecialization(vER 2, vER 3, vSDRS 1, vSDRS 2) ]
    }
  }
}
PRED:specialize(vER 1, vPDR 1, vER 2) {
  ATTR:SITLIST(vPDR 1) {
    ATTR:SIT(vER 2) {
      [ concatenateSituations(vPDR 1)
        specializeSitWithGraph(vER 1, vER 2, vPDR 1)
      ]
    }
  }
}
PRED:decompose(vER 1, vDR 1, vER 2) {
   PRED:into(vER 1, vPDR 1) {
     ATTR:SITLIST(vPDR 1) {
       [ concatenateSituations(vPDR 1)
         specializeSitWithGraph(vER 1, vER 2, vPDR 1)
       ]
    }
  }
}
ATTR:statement(vDR 1) {
  PRED:about(vDR 1, vER 1) {
    [ replaceOccurencies(vDR 1, vER 1) ]
  }
}
ATTR:concatenation(vDR 1) {
   PRED:of(vDR 1, vPDR 1) {
     [ replaceOccurencies(vDR 1, vPDR 1) ]
   }
}
*
```

C.4 Aktionsteilmethoden für Transformationsregeln

Der folgende Abschnitt enthält alle neu definierten Aktionsteilmethoden, welche im Aktionsteil einer TR sowie in den Aktionsblöcken des Bedeutungswörterbuches Verwendung finden. Ein Verzeichnis derjenigen Aktionsteilmethoden, welche schon zur Konstruktion einer DRS aus einem Syntaxbaum heraus definiert wurden, findet sich in [Arens & Ottlik 2000].

void collectDRKnowledge(String drs_id)
Diese Methode sucht zunächst alle Individuen–Bezugsträger, die in der übergebenen DRS Verwendung finden, also in Bedingungen dieser DRS vorkommen. Daraufhin werden alle Attributs– und Namensbedingungen, welche in der globalen DRS zu diesen Bezusgträgern vorhanden sind, durch die Methode gesammelt und in die übergebene DRS kopiert.

void removePredicate(Predicate pred)
Diese Methode löscht das übergebene Prädikat aus der aufrufenden DRS. Die Methode kann auch durch Voranstellen von **DRS*i*.** in einer Unter–DRS aufgerufen werden. Da Namens–, Attributs– sowie Relationsbedingungen Unterklassen der Klasse `Predicate` darstellen, können mit dieser Methode auch jene Bedingungen gelöscht werden.

void removeReferentSum(String refsum_id)
Diese Methode löscht den `PDR` mit dem übergebenen Bezeichner, welcher eine explizite Mengen–Definition darstellt. Wie **removePredicate** kann auch diese Methode in Unter–DRSen aufgerufen werden.

void removeSubDRS(String drs_id)
Diese Methode löscht die übergebene Unter–DRS aus der globalen DRS.

String copy(String drs_id)
Mit dieser Methode läßt sich eine Kopie der übergebenen DRS erstellen, welche dann durch den zurückgegebenen Bezeichner referenziert wird.

void callActions(Predicate)
Diese Methode löst den Zugriff auf das Bedeutungs–Wörterbuch für die übergebene Bedingung aus. Durch die Vererbungsbeziehungen zwischen den Klassen zur Repräsentation von Bedingungen einer DRS können der Methode wie bei **removePredicate** auch Namens–, Attributs– und Relationsbedingungen übergeben werden.

void addSituationAction(String er_id, Vector sit_vars, String drs_id)
Diese Methode schreibt jene UMTL–Regeln, welche das Handlungsschema des zu

erstellenden Situationsschemas repräsentieren. Das erste übergebene Argument bezeichnet den Namen des zu erstellenden Schemas, der an zweiter Stelle übergebenen `Vector` enthält die Variablen, welche in dem Situationsschema vorkommen. An dritter Stelle schließlich wird die DRS übergeben, welche das eigentliche Handlungsschema enthält.

void addSituationState(String er_id, Vector sit_vars, String drs_id)
Diese Methode schreibt alle zur Repräsentation eines Zustandsschemas nötigen UMTL–Regeln. Die übergebenen Parameter entsprechen denen der Methode **addSituationAction**, wobei jedoch die übergebene DRS hier das Zustandsschema enthält.

void addSituationPrediction(String er_id1, Vector vars1, String er_id2, Vector vars2)
Diese Methode schreibt die eine Prädiktion realisierenden UMTL–Regeln. Die Parameter der Methode sind: Der Name der Start–Situation, deren Variablen, der Name der Zielsituation und deren Variablen.

addSituationSpecialization(String er_id1, Vector vars1, String er_id2, Vector vars2)
Diese Methode führt eine Spezialisierung zwischen den übergebenen Situationen ein. Die Parameter der Methode entsprechen denen der Methode **addSituationPrediction**.

addSituationEvaluation(String er_id, Vector sit_vars)
Diese Methode schreibt alle Evaluierungs–Regeln, welche bei einer fehlgeschlagenen Spezialisierung eines Situationsschemas zu einem Prädiktionsversuch auf der gleichen Ebene des entsprechenden SGT führen. Übergeben werden der Methode der Name des Situationsschemas sowie die im Schema vorkommenden Variablen.

void concatenateSituations(String pdr_id)
Diese Methode bereitet die Verbindung aller im durch das Argument angegebenen `PDR` enthaltenen Situationschemata durch Prädiktionskanten vor. Dies geschieht durch das Einfügen entsprechender Prädikate (`PREDICT(ER`i`,ER`j`)`) in die zu transformierende DRS. Die Reihenfolge der Elemente des `PDR` wird hierbei eingehalten.

void specializeSitWithGraph(String er_id1, String sit_id, String pdr_id)
Diese Methode bereitet alle Spezialisierungen des Situjationsschemas **sit_id** durch die im `PDR` **pdr_id** enthaltenen Situationsschemata vor. Dies geschieht durch das

Einfügen der entsprechenden Prädikate (`SPEC(ER`i`,ER`j`)`) in die zu transformierende DRS. **er_id1** ist der Bezeichner der Ereignis–Definition, aus der die entsprechende Spezialisierungs–Anweisung stammt, und wird bisher von der Methode nicht genutzt.

void replaceOccurencies(String old_id, String new_id)
Die Aufgabe dieser Methode besteht in der Ersetzung aller Vorkommen des Bezeichners **old_id** durch den Bezeichner **new_id** in der gesamten zu transformierenden DRS.

Anhang D

Überlegungen zur Java–IDE JBuilder 4

D.1 Vorbemerkungen

Der folgende Text enthält einige Überlegungen zu Java und Java–Entwicklungsumgebungen im Zusammenhang mit der Frage, ob die Anschaffung der Entwicklungsumgebung *JBuilder 4* von Inprise lohnenswert erscheint. Dabei soll vor allem die Laufzeit von Java–Programmen eine Rolle spielen. Da diese im wesentlichen von der verwendeten virtuellen Java–Maschine (*Java Virtual Machine, JVM*) abhängt, die Teil jeder Java–Version ist, folgt zunächst ein Überblick über bisherige Java–Versionen und darin verwendete Begriffe. Anschließend wird auf verschiedene bekannte Entwicklungsumgebungen eingegangen.

D.2 Begriffsklärung

Java ist eine objektorientierte Programmiersprache, die eine mit C++ vergleichbare Syntax verwendet. Im Gegensatz zu C++ wird Java–Quelltext jedoch nicht in maschinenausführbaren Code, sondern in Binärcode der *JVM* übersetzt. Dadurch wird erreicht, daß Java–Programme auf jedem Rechnersystem ausführbar sind, für das eine solche virtuelle Maschine vorhanden ist.
Neben dieser Art der Ausführung von Java–Programmen existieren auch systemspezifische Übersetzer (*just-in-time-compiler, JIT*), die den Java–Quellcode in Maschinencode des verwendeten Rechners übersetzen und somit bessere Laufzeiten erreichen. Von einem Verlust der Systemunabhängigkeit von Java kann hierbei nicht gesprochen werden, da diese Übersetzung in Maschinencode – genau wie die oben angesprochene Übersetzung in Binärcode – bei jeder Ausführung des Java–Programms neu (*just in time*) geschieht.

Neben der *JVM* (respektive JIT) hängt die Leistungsfähigkeit einer Java–Version auch von der *Java–Plattform* ab. Mit diesem Begriff werden die vordefinierten Klassen und Pakete (*packages*) einer Java–Version bezeichnet. Gerade in diesem Bereich hat sich seit der ersten Java–Version am meisten getan.
Mit *JRE* (*Java Runtime Environment*) wird die Zusammenfassung all jene Komponenten bezeichnet, die zur Ausführung, nicht aber zur Entwicklung von Java–Programmen benötigt werden. *JDK* (*Java Development Kit*) schließlich bezeichnet die Zusammenfassung der *JRE*, des Java–Übersetzer *javac* und einiger anderer Entwicklungswerkzeuge. Das *JDK* wurde mit Veröffentlichung der Java–Version 1.2 in *Software Development Kit* (*SDK*) umbenannt.

D.3 Überblick über bisherige Java–Versionen

Java 1.0 Dies war die erste Version der Java–Programmiersprache. Die Java–Plattform beinhaltete 212 Klassen in 8 Paketen. Wegen der geringen Leistungsfähigkeit sowohl der Plattform als auch der *JVM* finden reine Java 1.0–Programme meist nur noch in Form von Applets Verwendung (dies unter anderem auch, weil viele Netz–Stöberer (*Browser*) bis vor Kurzem nur diese Java–Version unterstützten).

Java 1.1 Mit Veröffentlichung dieser Java–Version wurde der Umfang der Plattform auf 504 Klassen in 23 Paketen verdoppelt. Neben signifikanter Leistungssteigerung der *JVM* ist vor allem die Einführung von Inneren Klassen und eines neuen Ereignismodells für grafische Benutzungsoberflächen zu erwähnen (s. [Java 1.1]).

Java 1.2 Die Einführung dieser Version stellt den größten Entwicklungsschritt der Programmiersprache Java dar. Der Umfang der Plattform wurde auf 1520 Klassen in 59 Paketen verdreifacht. Wegen der Integration vieler neuer Komponenten (wie etwa *Java Swing*, eines Pakets zur Entwicklung grafischer Benutzungsoberflächen) wurde diese Version in *Java Plattform 2* umbenannt (s. [Java 1.2]).

Java 1.3 Mit Java 1.3 wurde gegenüber der Version 1.2 vor allem Wert auf die Korrektur von bekannten Fehlern gelegt. Neuerungen in der Plattform betreffen vor allem das Erstellen von verteilten Anwendungen und die Integration eines Pakets zur Audio–Unterstützung (*java.sound*). Bemerkenswert ist daneben die neue *JVM* dieser Version, jetzt *HotSpot* [HotSpot] genannt. Diese virtuelle Maschine arbeitet weiterhin als Interpreter, bei dem jedoch der erzeugte Code während der Ausführung auf laufzeitkritische Stellen hin untersucht wird. Diese Code–Teile werden daraufhin gesondert übersetzt und dabei optimiert (s. [Java 1.3]).

D.3.1 Leistungsvergleich

[JVM98] bietet mit dem *Spec JVM98–Benchmark* ein Testpaket für *JVMs* und *JITs* für alle gängigen Systemplattformen an. Testergebnisse speziell für *HotSpot* auf Sun–

Systemen können unter [JVM98 Query] nachgeschlagen werden.

D.4 Java–Entwicklungsumgebungen

Im folgenden wird auf einige sog. Integrierte Entwicklungsumgebungen (*integrated development environment, IDE*) für Java eingegangen. Diese *IDEs* dienen vor allem der schnellen Entwicklung grafischer Benutzungsoberflächen und der komfortablen Verwaltung grösserer Projekte. Der Funktionsumfang der betrachteten *IDEs* unterscheidet sich dabei kaum. Alle diese Umgebungen beinhalten die Unterstützung eines Versionskontrollsystems (CVS) und eignen sich dadurch zur Projektentwicklung in Gruppenarbeit. Die Unterschiede liegen hauptsächlich in der Unterstützung bestimmter Neuerungen in der Java–Plattform (erstellen verteilter Anwendungen u. ä.) bzw. bestimmter spezieller Java–Versionen für Unternehmen. Ein weiterer Unterschied liegt in den jeweils erhältlichen Versionen (frei vs. kommerziell) und dadurch nicht zuletzt im Preis der verschiedenen *IDEs*. Von Unterschieden in der Leistungsfähigkeit der mit verschiedenen *IDEs* erstellten Anwendungen kann bei Verwendung der gleichen Java–Plattform bzw. *JVM* nicht die Rede sein.

D.4.1 *Forte* von Sun

Forte ist eine *IDE* für Java von *Sun* selbst. Sie ist wohl als Nachfolger der ehemals *Java WorkShop* genannten Umgebung zu sehen. *Forte* ist in der sog. *Community Edition* frei erhältlich, komplett in Java geschrieben, auch der Quellcode ist frei zugänglich. Neben dieser Version existiert auch eine auf verteilte Anwendungen hin erweiterte *Internet Edition*, sowie eine Enterprise Edition, die noch weitere Module enthält. Alle Versionen der *IDE* enthalten das *SDK 2 Version 1.3* und sind für die Betriebssysteme Solaris, Linux und Windows erhältlich (s. [Forte]).

D.4.2 *JBuilder 4* von Inprise

JBuilder von Borland (respektive Inprise) ist wie *Forte* auch in drei verschiedenen Versionen erhältlich. Eine freie, *JBuilder Foundation* genannte, Version sowie zwei kommerzielle Versionen (*Professional* bzw. *Enterprise*), die sich zum einen durch die direkte Unterstützung bestimmter Sonderfunktionen von Java von der freien Version unterscheiden. Zum anderen ist jedoch auch die volle Nutzung einer Versionskontrolle erst in der *Enterprise*–Version möglich (s. [JBuilder 1; JBuilder 2]).

D.4.3 Andere *IDEs*

Neben den o.g. *IDEs* existieren noch weitere Entwicklungsumgebungen für Java, teils integriert, teils aber auch aus Einzelkomponenten zusammensetzbar. Darunter sind unter anderem *VisualAge for Java* von IBM [VisualAge] sowie *CodeWarrior* von Metrowerks [CodeWarrior].

D.5 Zusammenfassung

Der vorliegende Text versucht zunächst, einige Begriffe im Zusammenhang mit Java und Java–Entwicklungsumgebungen zu klären, soweit sie zur Beurteilung der Frage nach der Anschaffung von *JBuilder 4* von Inprise nötig erscheinen. Dabei wird vor allem auf die Unabhängigkeit der *IDE* von der Java–Plattform bzw. der *JVM* und damit von der Leistungsfähigkeit der entwickelten Anwendungen hingewiesen. Ein nennenswerter Unterschied im Funktionsumfang von aktuellen *IDEs* kann nicht festgestellt werden, obwohl dieser bei den meisten Herstellern erst in einer kommerziellen Version voll nutzbar ist. Da eine Installation des *Forte*–Systems von Sun am Institut schon vorliegt, Unterschiede zwischen den Umgebungen kaum gegeben sind und der Nutzen solcher Umgebungen prinzipiell eher bei der schnellen Entwicklung grafischer Oberflächen liegt, erscheint eine Anschaffung von *JBuilder* nicht nötig.

Anhang E

Zeitplan

E.1 Geplanter Verlauf

Zeitraum	Arbeitsschritt
20. 11. — 01. 12. (2 Wochen)	Einarbeitung in die Problemstellung, Literaturstudium.
02. 12. — 15. 12. (2 Wochen)	Einarbeitung in die vorhandenen Werkzeuge zur Situationsgraphenmodellierung und Umsetzung in UMTL. Erstellung von Beispieltexten zur Verhaltensbeschreibung.
16. 12. — 12. 01. (4 Wochen)	Erarbeitung einer Grammatik und der zugehörigen Konstruktionsregeln für die Umsetzung von Verhaltensbeschreibungen in DRSen im Hinblick auf die spätere Übersetzung nach UMTL.
13. 01. — 26. 01. (2 Wochen)	Anpassung der bisherigen Klassen zur Diskursrepräsentation an Verhaltensbeschreibungen.
27. 01. — 16. 02. (3 Wochen)	Entwurf einer rechnerinternen Repräsentation von Situationsgraphenbäumen in Java.
17. 02. — 16. 03. (4 Wochen)	Implementierung des Entwurfs und zugehöriger graphischer Schnittstellen.
17. 03. — 30. 03. (2 Wochen)	Entwurf der nötigen Methoden und Klassen zur Übersetzung von DRSen nach UMTL.
31. 03. — 20. 04. (3 Wochen)	Implementierung des Entwurfs.
21. 04. — 11. 05. (3 Wochen)	Test des Systems mit den erstellten Beispieltexten. Erweiterungen. Ausblick.
12. 05. — 18. 05. (1 Woche)	Abschluß der Arbeit und Vortragsvorbereitung.

E.2 Tatsächlicher Verlauf

Zeitraum	Arbeitsschritt	Dokumentation
20. 11. — 01. 12. (2 Wochen)	Einarbeitung in die Problemstellung, Literaturstudium.	Kapitel 1: Einleitung
02. 12. — 15. 12. (2 Wochen)	Recherche zu Java-IDEs. Einarbeitung in die vorhandenen Werkzeuge zur Situationsgraphenmodellierung und Umsetzung in UMTL.	Erweiterung von Kapitel 1, Kapitel 2: Grundlagen; Anhang D: Java
16. 12. — 12. 01. (4 Wochen)	Erarbeitung einer Grammatik und der zugehörigen Konstruktionsregeln für die Umsetzung von Verhaltensbeschreibungen in DRSen.	Anhang A: Grammatik-Dokumentation, Anhang B: Beispieltext u. DRSen
13. 01. — 26. 01. (2 Wochen)	Anpassung der bisherigen Klassen zur Diskursrepräsentation an Verhaltensbeschreibungen.	Erweiterung von Kapitel 1, Kapitel 3, Anpassungen an Anhang A und B
27. 01. — 16. 02. (3 Wochen)	Entwurf und Teilimplementierung einer rechnerinternen Repräsentation von Situationsgraphenbäumen in Java. Entwurf und Implementierung einer Repräsentation von UMTL-Programmen	Kapitel 4: Repräsentation von Situationsgraphenbäumen
17. 02. — 16. 03. (4 Wochen)	Entwurf eines Verfahrens zur Erstellung von Verhaltensschemata aus DRSen	Kapitel 5: Umwandlung von DRSen in Verhaltensschemata
17. 03. — 30. 03. (2 Wochen)	Überarbeitung und Beginn der Implementierung des Verfahrens.	Änderungen an Kapitel 2 und Kapitel 5
31. 03. — 20. 04. (3 Wochen)	Implementierung der Muster–Erkennung auf DRSen sowie einiger TRen. Beginn der Implementierung des Bedeutungswörterbuches.	Überarbeitung Kapitel 4, Kapitel 5
21. 04. — 11. 05. (3 Wochen)	Abschluß der Implementierung des Transformations–Verfahrens. Test des Verfahrens an einigen einfachen Beispieltexten.	Kapitel 6: Implementierung eines DRS–Transformators. Kapitel 7: Zusammenfassung. Änderungen an Kapitel 1 und 2.
12. 05. — 18. 05. (1 Wochen)	Abschluß der Arbeit.	Letzte Änderungen.

Literaturverzeichnis

[Arens & Ottlik 2000] M. Arens, A. Ottlik: *Automatische Analyse natürlichsprachlicher Texte zur Generierung synthetischer Bildfolgen.* Studienarbeit, Universität Karlsruhe, Institut für Algorithmen und Kognitive Systeme, Mai 2000.

[Brzoska 94] C. Brzoska: *Temporallogisches Programmieren.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Karlsruhe, Januar 1994.

[CodeWarrior] *The CodeWarrior Family of Products.* http://www.metrowerks.com/ .

[DiaGen] *DiaGen: The Diagram Editor Generator* . http://www2.informatik.uni-erlangen.de:80/IMMD-II/Research/Activities/DiaGen/index.html .

[Dinsmore 91] J. Dinsmore: *Logic–Based Processing of Semantically Complex Natural Language Discourse.* New Generation Computing **9**:1 (1991) 39–68.

[Dowding et al. 93] J. Dowding, J. M. Gawron, D. Appelt, J. Bear, L. Cherny, R. Moore, D. Moran: *Gemini: A Natural Language System for Spoken–Language Understanding.* Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics, Columbus, Ohio, USA, June 1993, pp. 54–61.

[Forte] *Forte^{TM} for Java^{TM}. Integrated Development Environment.* http://www.sun.com/forte/ffj/ .

[GD 1] *Graph Drawing.* http://rocana.aist-nara.ac.jp/ hayashi/links/gd.html .

[GD 2] *Geometry in Action: Graph Drawing.* http://www.ics.uci.edu/ eppstein/gina/gdraw.html .

[GD 3] *Graph Drawing.* http://www.cs.brown.edu/people/rt/gd.html .

[GML] *The GML File Format.* http://www.infosun.fmi.uni-passau.de/Graphlet/GML/ .

[Gerber 2000] R. Gerber: *Natürlichsprachliche Beschreibung von Straßenverkehrsszenen durch Bildfolgenauswertung.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Karlsruhe, Januar 2000; Erschienen im elektronischen Volltextarchiv der Universität Karlsruhe, http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=2000/informatik/8 .

[Gil & Melz 96] Y. Kim, E. Melz: *Explicit representation of problem–solving strategies to support knowledge acquisition.* Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96), Portland, Oregon, USA, August 1996, pp. 469–476.

[Gon. & Min. 84] M. Gondran and M. Minoux: *Graphs and Algorithms.* Wiley Series in Discrete Mathematics, John Wiley & Sons Ltd., Chichester a. o. 1984.

[GraphPanel] *GraphPanel 2.1: A Graph Editor.* http://binger.centre.edu/GraphPanel/ .

[Haag 98] M. Haag: *Bildfolgenauswertung zur Erkennung der Absichten von Straßenverkehrsteilnehmern.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Karlsruhe, Juli 1998; erschienen in der Reihe Dissertationen zur Künstlichen Intelligenz (DISKI) **193**; infix–Verlag Sankt Augustin 1998.

[Haag & Nagel 2000] M. Haag, H.–H. Nagel: *Incremental Recognition of Traffic Situations from Video Image Sequences.* Image and Vision Computing **18**:2 (2000) 137–153.

[HotSpot] *The Java HotSpot^{TM} Client and Server Virtual Machines.* http://java.sun.com/j2se/1.3/docs/guide/performance/hotspot.html .

[Java 1.1] *JDK 1.1 New Feature Summary.* http://java.sun.com/j2se/1.1/docs/relnotes/features.html .

[Java 1.2] *Java Development Kit (JDK^{TM}) Version 1.2 Summary of New Features.* http://java.sun.com/j2se/1.2/docs/relnotes/features.html .

[Java 1.3] *Java^{TM} 2 SDK, Standard Edition, version 1.3. Summary of New Features and Enhancements.* http://java.sun.com/j2se/1.3/docs/relnotes/features.html .

[JBuilder 1] *Borland JBuilder.* http://www.inprise.com/jbuilder/ .

[JBuilder 2] *JBuilder 4: Features.* http://www.inprise.com/jbuilder/jb4/feamatrix/ .

[JVM98] *SPEC JVM98.* http://www.spec.org/osg/jvm98/ .

[JVM98 Query] *SPECjvm98 Results – Query.* http://www.spec.org/cgi-bin/osgresults?conf=jvm98 .

[Kamp & Reyle 93] H. Kamp, U. Reyle: *From Discourse to Logic.* Kluwer Academic Publishers, Dordrecht, The Netherlands 1993.

[Kim & Gil 2000] J. Kim, Y. Gil: *Acquiring Problem–Solving Knowledge from End Users: Putting Interdependency Models to the Test.* Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000), Austin, Texas, USA, July/August 2000, pp. 223–229.

[Koller 92] D. Koller: *Detektion, Verfolgung und Klassifikation bewegter Objekte in monokularen Bildfolgen am Beispiel von Straßenverkehrsszenen.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Karlsruhe, Juni 1992; erschienen in der Reihe Dissertationen zur Künstlichen Intelligenz (DISKI) **13**; infix–Verlag Sankt Augustin 1992.

[Kollnig 95] H. Kollnig: *Ermittlung von Verkehrsgeschehen durch Bildfolgenauswertung.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Karlsruhe, Februar 1995; erschienen in der Reihe Dissertationen zur Künstlichen Intelligenz (DISKI) **88**; infix–Verlag Sankt Augustin 1995.

[Krüger 91] W. Krüger: *Begriffsgraphen zur Situationsmodellierung in der Bildfolgenauswertung.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Januar 1991; erschienen in Informatik–Fachberichte **311**, Springer–Verlag Berlin u.a. 1992.

[Leuck 2000] H. Leuck: *Untersuchungen zu einer systematischen Leistungssteigerung in der modellbasierten Bildfolgenauswertung.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Juli 2000; erschienen in der Reihe Berichte aus der Informatik; Shaker–Verlag Aachen 2001.

[Menzel 2000] W. Menzel: *Unscharfe Mengen.* Skriptum zur Vorlesung, Fakultät für Informatik, Universität Karlsruhe, Sommer 2000.

[Musc. & Kenn. 98] C. Musciano, B. Kennedy: *HTML: The Definitive Guide, Third Edition.* O'Reilly & Associates, Inc., Sebastopol, California, USA, 1998.

[Nagel 88] H.–H. Nagel: *From Image Sequences towards Conceptual Descriptions.* Image and Vision Computing **6**:2 (1988) 59–74.

[Nagel 99] H.–H. Nagel: *Natural Language Description of Image Sequences as a Form of Knowledge Representation.* W. Burgard, T. Christaller, A. Cremers (Eds.): KI–99: Advances in Artificial Intelligence, Proc. of the 23rd Annual German Conference on Artificial Intelligence, Lecture Notes in Artificial Intelligence **1701**, Bonn, September 1999, pp. 45–60.

[Schäfer 96] K. H. Schäfer: *Unscharfe zeitlogische Modellierung von Situationen und Handlungen in der Bildfolgenauswertung und Robotik.* Dissertation, Fakultät für Informatik der Universität Karlsruhe (TH), Karlsruhe, Juli 1996; erschienen in der Reihe Dissertationen zur Künstlichen Intelligenz (DISKI) **135**; infix–Verlag Sankt Augustin 1996.

[Schirra 94] J. Schirra: *Bildbeschreibung als Verbindung von visuellem und sprachlichem Raum.* Dissertation, Fakultät für Informatik der Universität des Saarlandes, Saarbrücken, April 1994; erschienen in der Reihe Dissertationen zur Künstlichen Intelligenz (DISKI) **71**; infix–Verlag Sankt Augustin 1994.

[VGJ] *Drawing Graphs with VGJ.*
http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html .

[VisualAge] *IBM Software: Application Development: VisualAge for Java: Overview.*
http://www-4.ibm.com/software/ad/vajava/ .