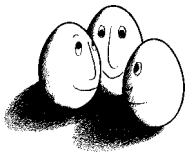
Diplomarbeit

Entwurf eines interaktiven Werkzeugs zur Datenverwaltung in einem situierten Lernsystem

Eckart Zitzler



Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

19. April 1996

Betreuer:

Prof. Dr. Katharina Morik
Dipl.-Inform. Volker Klingspor

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Die vorliegende Arbeit entstand im Rahmen des Forschungsprojekts BLearn-II (ESPRIT P7274), in dem maschinelles Lernen zur Roboternavigation eingesetzt wird. In einem mehrstufigen Lernprozeß werden aus Robotersensordaten operationale Begriffe gelernt, über die der Benutzer den Roboter steuern kann. Hierbei kommen verschiedene Lernverfahren zur Anwendung. Die besondere Art der Lernaufgabe macht es notwendig, mit verschiedenen Versionen von Daten zu arbeiten. Unter Berücksichtigung der enormen Datenmenge ergibt sich daraus das Problem, wie die Daten effizient verwaltet werden können. So entstand der Bedarf nach einer zentralen Datenverwaltung, die einerseits dem Benutzer den Überblick und einfache Datenzugriffe ermöglicht, andererseits von den Daten abstrahiert und das Prinzip des Information Hiding verwirklicht.

In diesem Kapitel werden diese Punkte näher erläutert. Nach einem kurzen Überblick über maschinelles Lernen allgemein wird die Anwendung in der Robotik skizziert und daraus die Motivation der Arbeit abgeleitet. Der anschließende Abschnitt beschäftigt sich mit der Aufgabenstellung und insbesondere meinem Vorgehen, den einzelnen Arbeitsschritten. Es folgt eine Übersicht über die Arbeit.

1.1 Maschinelles Lernen in der Robotik

An Computersysteme, die in der realen Welt agieren und reagieren müssen, werden hohe Anforderungen gestellt. Anpassungsfähigkeit und Flexibilität sind häufig notwendige Leistungsmerkmale, gerade wenn unbekannte Umgebungen oder Situationen gemeistert werden sollen. Die Nachteile vieler Programme zur Roboternavigation liegen aber zumeist in den starren Annahmen über die Umwelt sowie in einer sehr auf „Maschinenbedürfnisse“ zugeschnittenen Benutzerschnittstelle. Maschinelles Lernen kann helfen, den Einsatz von Robotern in verschiedenen Umgebungen zu erleichtern und die Bedienung von Robotern menschengerechter zu gestalten. Ein weiteres Motiv liegt in der Verwendung wissensbasierter Systeme überhaupt. Expertensysteme werden häufig dort eingesetzt, wo es keine befriedigende, algorithmische Lösung eines komplexen Pro-

blems gibt. In der Robotik kann ein Ansatz, der maschinelles Lernen verwendet, zu neuen Erkenntnissen führen und vielleicht auch in besseren, d.h. flexibleren, handhabbareren und sichereren Systemen resultieren.

Im folgenden Abschnitt werden die für die Arbeit erforderlichen Grundlagen des maschinellen Lernens dargestellt. Abschnitt 1.1.2 beschäftigt sich mit den Forschungsarbeiten des Lehrstuhls für Künstliche Intelligenz an der Universität Dortmund, die im Rahmen des Projekts BLearn-II stattfinden.

1.1.1 Induktives Lernen

Es gibt diverse maschinelle Lernverfahren, die anhand verschiedener Kriterien unterschieden werden können. Sie sind z.B. über die Art und Weise, wie geschlußfolgert wird, charakterisiert: Induktion, Deduktion oder auch Abduktion. Woraus gelernt wird, d.h. Beobachtungen oder Beispiele, ist auch ein wichtiges Unterscheidungsmerkmal. Während Beobachtungen unklassifiziert sind, stellen Beispiele einer Kategorie zugeordnete Aussagen dar. In diesem Kontext interessiert uns vor allem das logik-orientierte, induktive Lernen.

Der Bereich des induktiven logischen Programmierens (ILP) beschäftigt sich mit der Charakterisierung von Begriffen bzw. dem Entdecken von Regelhaftigkeiten. Das geschieht durch Induktion von prädikatenlogischen Formeln aus Beispielen, wobei Hintergrundwissen miteinbezogen wird. Dadurch zeichnen sich ILP-Verfahren auch gegenüber anderen Lernverfahren aus, nämlich daß sie in der Lage sind, als Repräsentationsformalismus eine eingeschränkte Prädikatenlogik zu verwenden und Hintergrundwissen beim Lernen mitzuberücksichtigen.

In diesem Zusammenhang sind Begriffslernen und Regellernen zu unterscheiden. Die Begriffslernaufgabe läßt sich folgendermaßen beschreiben [Morik 1995]:

Gegeben:
- positive und negative Beispiele E in einer Sprache LE
- Hintergrundwissen B in einer Sprache LB, wobei $B \cup H \not\models \square$

Ziel: eine Hypothese H in einer Sprache LC, so daß
- $B, H, E \not\models \square$ (Konsistenz)
- $B, H \models E^+$ (Vollständigkeit)
- $\forall e \in E^- : B, H \not\models e$ (Korrektheit)

Die Regellernaufgabe hingegegen lautet [Muggleton und Raedt 1993]:

Gegeben:
- Hintergrundwissen B in einer definiten Theorie LB

Ziel: eine Hypothese H in einer Sprache LC, so daß gilt:
- $\forall h \in H : h$ ist wahr im minimalen Modell von B,
- ist eine Formel $g \subset$ LC wahr im minimalen Modell von B, dann gilt $H \models g$, und
- es gibt keine gültige und vollständige, echte Teilmenge G von H.

Bei der Hypothesensprache handelt es sich um eine eingeschränkte Prädikatenlogik. Das Hintergrundwissen B und die Beispiele E werden meist in Form instanziierter Fakten angegeben.

Wie Relationen effektiv dargestellt werden, spielt in diesem Zusammenhang eine wichtige Rolle. Da eine Attribut-Werte-Repräsentation diese Möglichkeit nur eingeschränkt bietet, scheidet sie als Repräsentationsformalismus aus.

Ich möchte das Szenario exemplarisch an einer Domäne verdeutlichen, dem Lernen von Verwandtschaftsverhältnissen. Gelernt werden soll eine Beschreibung des Begriffs `ist_opa_von`, so daß der Rechner nachher in der Lage ist zu entscheiden, ob zwei Menschen in der Großvater-Enkel-Beziehung stehen. Dazu werden zwei positive und zwei negative Beispiele angegeben (Abbildung 1.1). Dies genügt allerdings noch nicht, um den Begriff charakterisieren zu können. Benötigt werden noch die bestehenden Verwandtschaftsverhältnisse, speziell die Eltern-Kind-Relation. Nun soll eine Hypothese gefunden werden, die mit den gegebenen Verwandtschaften in Einklang steht (Konsistenz), die alle positiven Beispiele abdeckt (Vollständigkeit) und die kein negatives Beispiel herleitet (Korrektheit).

Beispiele:

```
ist_opa_von(max, ecki)
ist_opa_von(schorsch, ecki)
not(ist_opa_von(karin, ecki))
not(ist_opa_von(ingo, ecki))
```

Hintergrundwissen:

```
ist_papa_von(max, karin)
ist_papa_von(schorsch, ingo)
ist_mama_von(karin, ecki)
ist_papa_von(ingo, ecki)
```

Hypothese:

```
ist_opa_von(A, B) :-
  ist_papa_von(A, C),
  ist_papa_von(C, B).
ist_opa_von(A, B) :-
  ist_papa_von(A, C),
  ist_mama_von(C, B).
```

Abbildung 1.1: Szenario beim Lernen von Verwandtschaftsverhältnissen

RDT [Kietz und Wrobel 1992] ist eine logik-orientiertes, induktives Regellernverfahren, das Eingaben in Form von Grundfakten erwartet. Es benutzt Regelschemata, um den Hypothesenraum der Sprache LC zu beschränken und effizient lernen zu können. Lernziel ist das Auffinden möglichst vieler generellster Regeln, die ein vom Benutzer vorgegebenes Akzeptanzkriterium erfüllen. GRDT [Klingspor 1994] ist ein weiterer Lernalgorithmus, der Ideen von RDT und einem anderen Verfahren (GRENDEL [Cohen 1992]) miteinander vereint. Anstatt ein festes Regelschemata zu verwenden, erzeugt GRDT während des Lernvorgangs diese Schemata, die über Grammatiken definiert sind.

Für die vorliegende Arbeit ist es unerheblich, wie die Lernverfahren im einzel-

nen vorgehen. Wichtig sind Eingabe, Parameter und Ausgabe. Gewöhnlich ist das Lernen jedoch kein einmaliger Vorgang. Die Eingabedaten werden immer wieder verändert, die Lernparameter variiert, bis schließlich das Ergebnis zufriedenstellend ist. Letzteres heißt vor allem, daß die Begriffsbeschreibung auch anderen Daten genügt, als denen, die zum Lernen verwendet wurden. Der generelle Ablauf des Lernprozesses, so wie er sich für den Benutzer darstellt, ist in Abbildung 1.2 wiedergegeben. Im ersten Schritt werden die relevanten Daten (Beispiele und Hintergrundwissen bei ILP) zusammengestellt, insbesondere werden ein Lernset und ein Testset erzeugt. Das Lernset dient als Eingabe zum Lernen, das Testset zum Überprüfen des Lernresultats. Die Auswahl der Daten hat auf das Lernergebnis entscheidenden Einfluß, denn eine zu große Datenmenge kann das Lernverfahren überfordern, eine zu kleine die Abgrenzung des Begriffs verunmöglichen. Anschließend, im zweiten Schritt, startet der Benutzer das Lernverfahren. Er muß hierzu die notwendigen Lernparameter setzen, beispielsweise Regelschemata spezifizieren und das Akzeptanzkriterium festlegen. Der dritte Schritt hat die Evaluation des Lernergebnisses zum Ziel. Anhand der Testdaten soll entschieden werden, ob die gelernten Regeln (im Falle von RDT oder GRDT) den gewünschten Begriff ausreichend beschreiben. Die Testergebnisse können genutzt werden, um einen neuen Lernlauf zu konfigurieren. Der Benutzer beginnt wieder bei Schritt eins, es handelt sich also um ein iteratives Vorgehen.

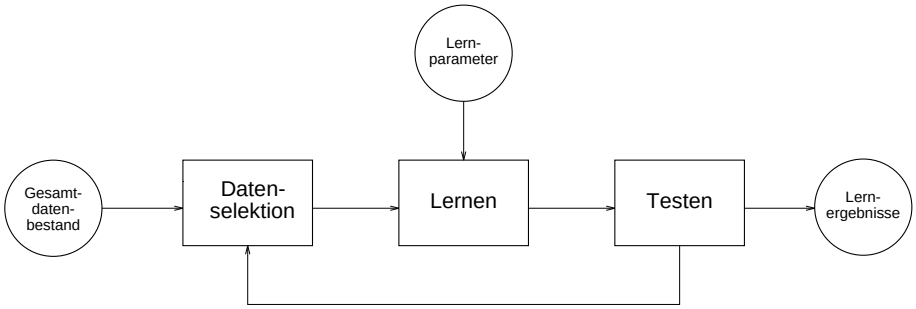


Abbildung 1.2: Das Lernszenario aus Sicht des Benutzers

1.1.2 Lernen operationaler Begriffe aus Robotersensordaten

Wie bereits eingangs erwähnt, soll mit einem neuen Ansatz die Flexibilität des Roboters und die Mensch-Maschine-Kommunikation verbessert werden. Ziel dieses Ansatzes ist es, für den Benutzer leicht verständliche, operationale Begriffe zu lernen, über die der Roboter anschließend gesteuert werden kann. Operationale Konzepte, z.B. das Fahren durch eine Tür, integrieren Wahrnehmungen und Handlungen des Roboters [Klingspor und Morik 1995]. Es gilt, die Lücke zwischen den eher natürlichsprachlichen Begriffen und den numerischen Daten, die der Roboter über seine Umwelt liefert, zu schließen.

Der Roboter PRIAMOS [Dillmann *et al.* 1993] bildet die Basis, auf der die Ideen über operationale Begriffe angewendet werden. Er wurde an der Universität Karlsruhe entwickelt und besitzt 24 Ultraschallsensoren zur Distanzmessung. Die Sensoren befinden sich auf gleicher Höhe und sind symmetrisch an den Seiten und Ecken des Roboters angebracht. PRIAMOS kann in jede Richtung fahren und sich gleichzeitig drehen (drei Freiheitsgrade). Die Daten, die er übermittelt, stellen die Grundlage zum Lernen der operationalen Begriffe dar. Sie beinhalten die Sensordistanzmessungen sowie Informationen über die aktuelle Roboterposition.

Die Lernaufgabe ist zu komplex, als daß sie in einem Schritt bewältigt werden könnte. Ein mehrstufiger Lernprozeß ist notwendig. Dem trägt die Repräsentationshierarchie (Abbildung 1.3) Rechnung, indem sie schrittweise von den vom Roboter gelieferten Daten abstrahiert [Klingspor *et al.* 1996].

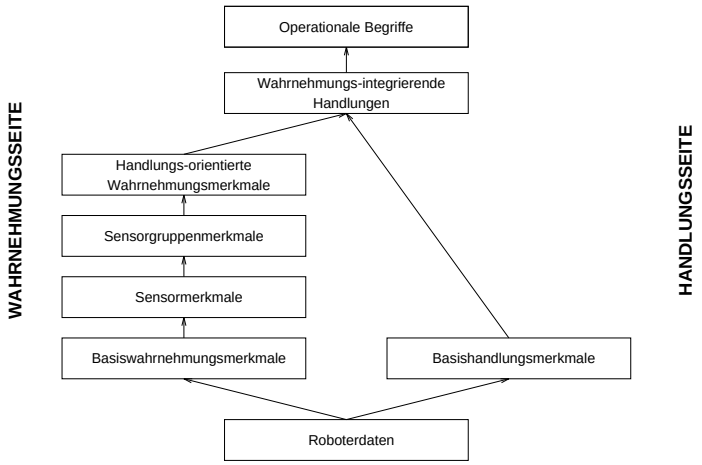


Abbildung 1.3: Die Repräsentationshierarchie

Auf der Handlungsseite werden aus den Roboterdaten zunächst grundlegende Handlungsmerkmale berechnet [Sklorz 1995], die anschließend mit den Wahrnehmungen zu wahrnehmungs-integrierenden Handlungen kombiniert werden. Dem steht auf der Wahrnehmungsseite eine feinere Unterteilung gegenüber. Basiswahrnehmungsmerkmale fassen aufeinanderfolgende Sensordaten zu abstrakteren Begriffen zusammen [Wessel 1995]. Darauf bauen die Sensormerkmale und die Sensorgruppenmerkmale auf. Sie stellen Konzepte dar, die die Wahrnehmungen eines Sensors bzw. einer Gruppe von Sensoren auf einer höheren Ebene beschreiben [Morik und Rieger 1993]. Noch eine Stufe darüber sind die handlungs-orientierten Wahrnehmungsmerkmale angesiedelt.

In diesem Zusammenhang sind Lern- und Anwendungsphase des Roboters zu unterscheiden. Während in der Lernphase auf allen Abstraktionsebenen (eine Ausnahme sind die Basishandlungsmerkmale) die erforderlichen Daten gelernt werden sollen, werden diese in der Anwendungsphase genutzt, um den Roboter zu navigieren. Allerdings liegt das Hauptaugenmerk in meiner Arbeit auf dem

Lernen, wenn auch die Performanzphase nicht unberücksichtigt bleibt.

Grundlagen zum Lernen bilden die Roboterdaten, die aus mehreren, teilweise simulierten Testfahrten entlang festgelegter Pfade (*Traces*) gewonnen wurden; die Daten zu einem Pfad sind klassifiziert. Der erste Lernschritt unterscheidet sich grundsätzlich von den nachfolgenden. Auf dieser Ebene geht es um eine *Signal-To-Symbol*-Konvertierung: numerische Daten müssen in qualitative Merkmale umgewandelt werden. Stephanie Wessel [Wessel 1995] hat im Rahmen ihrer Diplomarbeit einen Algorithmus implementiert, der dieses Problem löst. Lernziel ist nun, geeignete Programmparameter zu finden, so daß aus den berechneten Basiswahrnehmungsmerkmalen sinnvoll die weiteren Wahrnehmungsmerkmale gelernt werden können. Auf den weiteren Ebenen ist die Lernaufgabe prinzipiell gleich: Aus den Lernergebnissen der darunterliegenden Abstraktionsstufe und geeignetem Hintergrundwissen sollen die gewünschten Begriffe gelernt werden. Hier kommen RDT und GRDT zur Anwendung, wenn auch grundsätzlich andere Verfahren einsetzbar sein sollen.

1.2 Motivation

Die Umstände, die ein Werkzeug zur Datenverwaltung nötig machten, lassen sich aus zwei Perspektiven erklären. Wie das bestehende Lern- und Performanzsystem entstand und welche Probleme sich daraus ergaben, wird in Abschnitt 1.2.2 aus der Sicht des Software Engineering geschildert. Demgegenüber sind es aus dem Blickwinkel des maschinellen Lernens die außergewöhnliche Lernaufgabe und die damit verbundenen Schwierigkeiten, die zu dieser Arbeit führten (Abschnitt 1.2.1).

1.2.1 Die Besonderheit der Lernaufgabe

Das Lernen von operationalen Begriffen aus Robotersensordaten erweist sich als ein komplexes Problem. Schrittweise wird von den Daten über die reale Welt abstrahiert bis zur obersten Stufe in der Repräsentationshierarchie—ein mehrstufiger Lernprozeß. In der Regel wird der Lernvorgang auf jeder Ebene separat durchgeführt. Beispiele und Hintergrundwissen werden generiert und zusammengestellt, Lernergebnisse inspiziert und getestet. Sind die Resultate zufriedenstellend, werden sie als Eingabe zum Lernen auf der nächsthöheren Stufe verwendet. Andererseits ist die genaue Bewertung von Lernergebnissen einer Ebene oft erst möglich, wenn abstraktere Konzepte vorliegen. Wurden beispielsweise die wahrnehmungs-integrierenden Handlungen nur unzureichend gelernt, so muß geprüft werden, „an welchem Rädchen zu drehen ist“. Sind die Lernparameter falsch gewählt, verursachen vielleicht ungenügende Sensorgruppenmuster den Mißerfolg, oder liegt der Grund in der ungünstigen Berechnung der Basiswahrnehmungsmerkmale? Der Lernvorgang an sich ist also kein linearer Prozeß, der die Repräsentationshierarchie als Ganzes betrachtet, sondern es finden unzählige, einzelne Lernläufe statt, es wird von Stufe zu Stufe gesprungen.

Da aber in den seltensten Fällen die Ursache mangelhafter Lernresultate direkt bestimmt werden kann, probiert man einfach aus: „Mal schauen, was passiert, wenn ich hier etwas ändere...“ Die Konsequenz ist, daß auf jeder Ebene mit mehreren Varianten im Sinne alternativer Datensätze gearbeitet wird. Man benutzt also Varianten, um diverse Möglichkeiten zum Lernen höherer Konzepte auszutesten. Nehmen wir z.B. die Basiswahrnehmungsmerkmale, deren Erzeugung über Programmparameter beeinflußt wird. So führen unterschiedliche Parameter zu entsprechenden Varianten, die vergleichend zum Lernen der Sensormerkmale benutzt werden können. Man wählt schließlich die Variante, mit der die besten Ergebnisse erzielt wurden. Der Auswahl der Daten zum Lernen kommt somit eine wichtige Bedeutung zu, nicht zuletzt stellt sie auch einen Zeitfaktor im Lernprozeß dar. Das Lernen an sich geht automatisch, doch die Auswahl erfordert die Anwesenheit des Benutzers.

Probleme mit bestehenden Lernumgebungen

Die Fülle von Daten, die in diesem Lernsystem erzeugt und verarbeitet wird, ist immens. Berücksichtigt man dabei noch die Notwendigkeit, mit Datenvarianten zu arbeiten, so wird deutlich, welchen speziellen Anforderungen eine Lernumgebung gerecht werden muß. Existierende wissensbasierte Systeme wie MOBAL [Morik *et al.* 1993] oder GRENDEL [Cohen 1992] sind damit überfordert, sie sind für diese Anwendung nicht konzipiert.

Auf der einen Seite gibt es voll integrierte Wissensakquisitions-Werkzeuge mit graphischer Oberfläche, wie z.B. MOBAL. MOBAL stellt eine komfortable Lernumgebung dar, die den Benutzer in vielerlei Hinsicht unterstützt und ihm mühsame Arbeit abnimmt. Eingaben werden einer Syntaxprüfung unterzogen, die Wissensbasis wird auf Konsistenz und Integrität hin überwacht. Die Benutzerführung ist einfach gehalten und erlaubt den Aufruf verschiedener Lernverfahren in einheitlicher Art und Weise. Es wird daher eine Schnittstelle zur Verfügung gestellt, über die externe Lernverfahren und andere Programme eingebunden werden können (RDT hingegen ist Bestandteil von MOBAL). Außerdem werden Vorwärts- und Rückwärtsinferenzen im On-Line-Betrieb, Beweispfadspeicherung, diverse Statistiken, etc. angeboten. Auch die Verwaltung unterschiedlicher Wissensbasen (Domänen) wird unterstützt.

MOBAL ist jedoch für diese Anwendung ungeeignet, was u.a. an den Massendaten liegt. Den kompletten Datenbestand (bis zu 500.000 Fakten!) einzuladen, führt zu enormen Speicherplatzproblemen und macht die Arbeit mit dem System unmöglich. Das, was ursprünglich als komfortable Unterstützung des Benutzers gedacht war, nämlich Konsistenz- und Integritätsprüfung der Wissensbasis, erweist sich in diesem Fall als hinderlich. Das gilt im allgemeinen für alle wissensbasierten Systeme, die Änderungen der Wissensbasis durch den Benutzer aufwendig überprüfen. Andererseits bietet MOBAL kein Konzept zur Arbeit mit Varianten. Dadurch, daß auf einer Stufe immer wieder mit verschiedenen Daten gelernt und getestet wird, ist es mühsam, eine komplette Lernhierarchie zu erstellen. Man kann sich zwar verschiedene Domänen konstruieren, doch gebraucht würde das Mischen von Domänen. Ich stelle mir beispielsweise einen

Benutzer vor, der denkt: „So, ich stelle nun die ultimative Wissensbasis mit den besten Lernergebnissen zusammen. Was war denn gut?—Ach ja, die Basiswahrnehmungsmerkmale aus Domäne eins waren sehr gut, die Sensormerkmale aus Domäne fünf auch nicht übel. Für die Sensorgruppenmerkmale wähle ich am besten die aus Domäne zwei...“ Zur Zeit muß diese Arbeit noch von Hand erfolgen, d.h. auf UNIX-Ebene oder mit Hilfe eines Editors. Auch der mehrstufige Lernprozeß wirft Probleme auf. MOBAL leitet in einem Lernlauf alles, was möglich ist, ab. Das ist in unserem Fall aber nicht wünschenswert, man möchte den Lernvorgang in seine Einzelschritte zerlegen und die Abstraktionsebenen getrennt betrachten können. Charakteristisch für den Benutzer des Lernsystems ist eben, daß er eine Informatikausbildung besitzt und sich in dem Problembereich auskennt. Es macht keinen Sinn, den Endbenutzer des Roboters mit der Lernphase zu konfrontieren. Folglich soll genau das, was in MOBAL zur Unterstützung des Benutzers automatisiert wurde, wieder explizit sichtbar sein; es handelt sich um eine andere Anwendergruppe. Schließlich läßt sich noch als weiterer Grund der erhöhte Zeitbedarf anführen, der natürlicherweise mit den vielfachen Überprüfungen der Daten verbunden ist. Bei der Masse an Daten ist MOBAL häufig einfach zu langsam (z.B. bei der Evaluierung).

Auf der anderen Seite gibt es Lernumgebungen ohne Datenhaltung und aufwendige Benutzeroberfläche. Bei Ihnen ist die Verwaltung und Überprüfung von Daten vollkommen ausgegrenzt. In der Regel arbeiten sie datei-orientiert, d.h. die Lern- und Testsets sind in Dateien abgelegt und müssen zu geeigneter Zeit eingelesen werden. Manche dieser Systeme bedient der Anwender auf UNIX-Ebene (FOIL [Quinlan 1990]), andere wiederum sind in Prolog eingebettet (GRENDEL). Gemeinsam ist ihnen, daß sie ein einzelnes Lernverfahren unterstützen und ein bis mehrere Evaluierungsverfahren (z.B. Cross Validation) anbieten. Die Verarbeitung von Massendaten ist möglich, soweit es das Lernverfahren zuläßt.

Doch auch diese Umgebungen kommen für ein derartiges Lernsystem nicht in Frage. Zum einen lösen sie die Problematik der Datenzusammenstellung und -verwaltung nicht. Der Benutzer wird allein gelassen in seiner Verzweiflung, den Überblick über die vielen Varianten zu behalten. Ihm bleibt nichts anderes übrig, als alles von Hand machen. Zum anderen ermöglichen sie es nicht, mehrere Lernverfahren einzubinden. Will man also tatsächlich verschiedene Lernalgorithmen einsetzen, wird man mit dem Problem des Datenaustauschs konfrontiert. Jedes Lernverfahren hat sein eigenes Format, die Daten müßten ständig angepaßt werden.

Integration unterschiedlicher Lernverfahren

Da nun deutlich geworden ist, warum existierende Lernunterstützungssysteme nicht oder nur unzureichend eingesetzt werden können, stellt sich die Frage, wie denn dann die Wissensbasis konstruiert ist. Vor allem: Wie tauschen die unterschiedlichen Lernverfahren Daten aus? Letzteres zielt auf die Integration heterogener Repräsentationsformalismen ab. Hierzu gibt es mehrere Ansätze, von denen zwei exemplarisch vorgestellt werden sollen. Im Vergleich dazu be-

schreibe ich anschließend die gewählte Lösung.

Expertensysteme, die von Sachgebietsexperten bedient werden sollen, müssen den Benutzer in vielfacher Weise unterstützen. Das vom Experten eingespeiste Wissen muß auf Fehlerhaftigkeit und Konsistenz mit dem vorliegenden Wissen geprüft werden; zudem sind Ein- und Ausgabe in einer menschenverständlichen Art und Weise zu gestalten. Das Ziel ist, möglichst weit von der computerinternen Wissensrepräsentation zu abstrahieren und auf einer inhaltlichen Ebene zu kommunizieren. Des weiteren erfordern unterschiedliche Arten von Wissen (strukturell, heuristisch, ...), wie sie in vielen Anwendungen vorliegen, auch spezielle Schlußfolgerungsmechanismen [van Heijst 1995, S. 118]. Verschiedene Problem Solver (van Heijst versteht darunter eine Wissensbasis zusammen mit einem Inferenzmechanismus [van Heijst 1995, S. 117]) arbeiten also zusammen an einem Problem und müssen folglich Daten austauschen.

In einem Ansatz [van Heijst 1995] wird eine wissensbasierte Integration vorgeschlagen, die das Wissen inhaltlich betrachtet (Wissensebene), anstatt es in einem Repräsentationsformalismus syntaktisch zu beschreiben (Symbolebene). Möchte ein Problem Solver mit einem anderen kommunizieren, so wird zunächst das in einer bestimmten Repräsentation vorliegende Wissen umgewandelt in eine inhaltliche Beschreibung, um anschließend wieder in die andere Darstellung konvertiert zu werden (Abbildung 1.4).

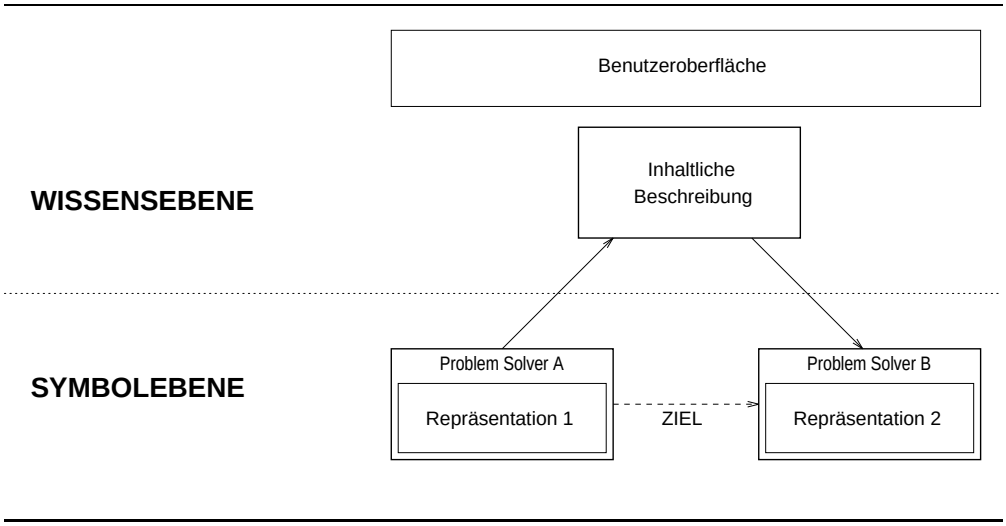


Abbildung 1.4: Wissensbasierte Integration von Repräsentationsformalismen

Die Problem Solver sind eingebettet in eine Umgebung namens CUE. Diese ermöglicht eine Darstellung des Wissens auf inhaltlicher Ebene und ist gleichzeitig in der Lage, zwischen Meta-Modellen von Repräsentationen umzuschalten. Meta-Modelle dienen quasi als Schnittstelle zwischen Problem Solvern und der Umgebung. Für jeden Problem Solver gibt es ein Meta-Modell seines Repräsentationsformalismus, er ist selbst dafür verantwortlich, Daten auf der Meta-Ebene in die interne Darstellung umzuwandeln und umgekehrt. Der Benutzer hingegegen gibt das Wissen immer inhaltlich an, CUE gewährleistet,

daß alle Problem Solver damit arbeiten können. Auch wenn in diesem Kontext von Problem Solvern die Rede ist, läßt sich die Herangehensweise nahtlos auf Lernverfahren übertragen.

Eine ähnliche Herangehensweise wurde auch bei der Entwicklung von CKRL [Causse *et al.* 1990][Kietz 1991], eine Abkürzung für Common Knowledge Representation Language, erwogen, allerdings entschied man sich u.a. aufgrund folgender Nachteile dagegen [Causse *et al.* 1990, S. 4]:

- Die Lernverfahren bzw. Problem Solver sind zu eng mit der Umgebung verzahnt. Wird ein Lernverfahren modifiziert, kann das Änderungen in der Umgebung nach sich ziehen (in CUE beispielsweise eine Spracherweiterung auf Wissensebene). Dies wirkt sich letztlich auf alle eingebundenen Lernverfahren aus.

- Die ständige Transformation von Daten kann u.U. viel Zeit in Anspruch nehmen, so daß das Gesamtsystem ineffizient und unflexibel wird.

Um dies zu vermeiden, folgt CKRL dem Prinzip: Datenkonvertierung nur dann, wenn nötig. CKRL ist eine Beschreibungssprache zur allgemeinen Darstellung von Wissen. Allgemein in dem Sinne, daß ein Format zur Verfügung gestellt wird, welches Ein- und Ausgaben verschiedener Lernverfahren repräsentieren kann. Möchte ein Lernverfahren Daten übermitteln, so sind sie in die CKRL-Darstellung umzuwandeln. Der Empfänger muß seinerseits die CKRL-Datei interpretieren und die Daten in die eigene Repräsentation konvertieren. Dieses Szenario ist in Abbildung 1.5 dargestellt. Im Unterschied zu CUE ist bei CKRL bewußt darauf verzichtet worden, dem Benutzer eine abstraktere Sicht der Wissenbasis zu bieten, das Hauptaugenmerk liegt in der Kommunikation zwischen Lernverfahren.

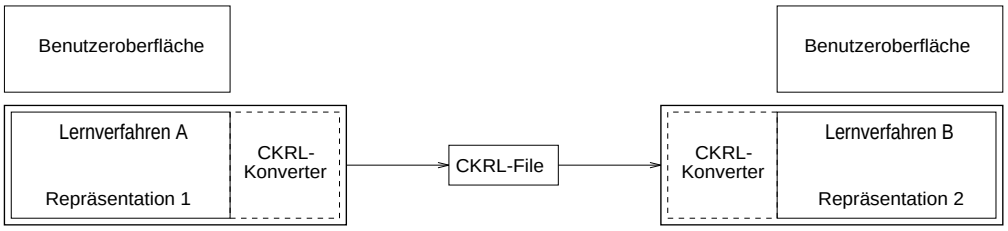


Abbildung 1.5: Kommunikation mittels CKRL

Die Lösung, die für das Lernsystem im Projekt BLearn-II gewählt wurde, geht noch einen Schritt weiter: Die Daten werden nicht mehr in eine höhere Beschreibung übersetzt, sondern in einer vordefinierten Repräsentation ausgetauscht. Während in CUE der Weg über eine inhaltliche Beschreibung beschritten und bei CKRL ein allgemeines Repräsentationsformat benutzt wird, entfällt die Datentransformation zur Kommunikation hier komplett. Man könnte sagen, von

CUE über CKRL bis hin zu diesem Ansatz nimmt das Wissen über den Inhalt der ausgetauschten Datensätze ab. Allerdings ist es auch ein Frage der Einbettung in eine vorgegebene Umgebung. CUE gliedert die beteiligten Problem Solver vollständig unter eine einheitliche Benutzeroberfläche ein. Demgegenüber erlaubt CKRL eine losere Kopplung und sieht die Möglichkeit vor, daß jedes Lernverfahren seine eigene Benutzeroberfläche besitzt. Hier ging man noch weiter, indem das allgemeine Format zum Datenaustausch weggelassen wurde. Dafür sprechen mehrere Gründe. Zum einen bedeutet jede Datenkonvertierung einen Zeitverlust, der den Lernvorgang verzögert. Bei der gegebenen Masse an Daten ist das ein sehr wichtiger Aspekt. Zudem müssen sehr oft Daten ausgetauscht werden, denn der Benutzer probiert ja häufig mehrere Varianten alternativ aus. Der zweite Grund wird in Abschnitt 1.2.2 noch näher erläutert. Da, wie bereits erwähnt, keinen geeignete Lernumgebung zur Verfügung stand, hätte für die Wissensbasis ein eigenes Tool entwickelt werden müssen. Dies war aus Zeitgründen nicht möglich. Schließlich bleibt noch der Anwender hervorzuheben, der dieses System benutzt. Er muß die Details der Lernaufgabe kennen, folglich kann er die Daten korrekt interpretieren. Warum sollte also eine abstraktere Zwischenrepräsentation gewählt werden?

Unserer Meinung nach ist für eine Lösung die Zweckmäßigkeit entscheidend. In der Informatik gibt es zumeist mehrere Methoden zur Lösung ein und desselben Problems; jede hat ihre Vor- und Nachteile. Eine pauschale Bewertung vorzunehmen, die Aussagen der Form „Methode A ist besser als Methode B“ zuläßt, ist jedoch in der Regel nicht möglich. Es kommt auf das Anwendungsgebiet an, die charakteristischen Merkmale. Hier liegt eine ähnliche Situation vor. Während bei CUE eher ein Anwender angesprochen wird, der zwar mit dem Computer umgehen kann, aber von Wissensrepräsentation nichts versteht (z.B. ein Arzt im Krankenhaus), zielt CKRL auf einen anderen Benutzer ab. Er sollte mit maschinellem Lernen vertraut sein, muß aber nicht die Einzelheiten der Anwendung kennen. In unserem Fall ist der Benutzer allerdings ein Experte in maschinellem Lernen und weiß um die Anwendungsdetails.

Datenaustausch auf Dateiebene

Die Entscheidung, die Daten zwischen Ebenen der Repräsentationshierarchie über Dateien auszutauschen, zog natürlich auch Probleme nach sich. In der Praxis zeigte es sich, daß viel Abstimmung und Disziplin im Team nötig sind, um eine konsistente Datenhaltung zu gewährleisten. Die Konsequenzen waren Unübersichtlichkeit und Verschwendung des Plattenspeichers durch mehrfach abgespeicherte, identische Daten. Zudem obliegt es jedem Benutzer selbst zu protokollieren, welche Daten er wann, wie erzeugt und wo abgelegt hat. Das Wiederauffinden von Daten stellte somit ein mühseliges Unterfangen dar, die Zusammenstellung von Lern- und Testsets ist erschwert. Auch die Auswahl von Daten aus Dateien ist ein Aspekt, der Hilfe vom System erfordert.

So entstand der Wunsch nach einer zentralen Datenhaltung, die dem Benutzer die Verwaltung der Dateien abnimmt und ihn bei der Datenauswahl unterstützt. Für diese Zwecke ließen sich auch Programme zur Versionenkontrolle einsetzen

(z.B. *RCS*), doch im nächsten Abschnitt wird klar, daß die Anforderungen an die Datenverwaltung darüber hinausgehen.

1.2.2 Software Prototyping in der Forschung

Am Anfang der Entwicklung des Lernsystems stand die Frage, welche Unterstützungswerkzeuge zur Verfügung stehen und eingesetzt werden können. Es stellte sich heraus, daß bestehende Lernumgebungen den der Lernaufgabe inhärenten Problemen nicht gerecht werden. Somit war die Einbettung in ein vorhandenes System unmöglich, ein eigenständiges System mußte erstellt werden.

Das Wasserfallmodell

Das älteste und wohl einfachste Modell zur Softwareentwicklung ist das Wasserfallmodell. Es faßt die Erstellung von Softwareprodukten als einen linearen Prozeß auf, der die weithin bekannten Phasen der Anforderungsanalyse, des Entwerfens, des Implementierens und des Testens durchläuft. Der starre Ablauf und die Voraussetzungen, von denen das Modell ausgeht, sind jedoch vielfach kritisiert worden. Es treffe nicht immer auf die realen Gegebenheiten zu, so zeige die Praxis. Oftmals können die Anforderungen für ein interaktives System nicht a priori vollständig spezifiziert werden [Abowd *et al.* 1995], und Jalote [Jalote 1991] führt an, daß gerade bei der Entwicklung absolut neuer Systeme die Anforderungen unbekannt sind.

Kritik läßt sich auch aus der Perspektive der Wissensakquisition üben. Während in früheren Ansätzen Wissenserwerb als ein Transfer des Wissens vom Experten zum Computer aufgefaßt wurde, stellte es sich später als sinnvoller heraus, die Fertigkeiten eines Experten zu modellieren. Somit war der Prozeß des Aquirierens von Wissen nicht mehr ein einmaliger Vorgang, sondern zyklischer, infiniter Natur. Diese Sichtweise wurde von Morik [Morik 1989] unter der Bezeichnung Sloppy Modeling eingeführt. Wenn man nun diese Ansicht vertritt, so steht das im krassen Widerspruch zum Wasserfallmodell.

Im vorliegenden Fall sollte schließlich ein Ansatz gewählt werden, der Verfahren des maschinellen Wissenserwerbs verwendet. Erfahrungen in der Domäne der Roboternavigation gab es nicht, es galt, ein komplexes Problem in der Forschung mit neuen Methoden anzugehen. Diese Gründe legten schließlich nahe, daß das Wasserfallmodell für diese Anwendung nicht geeignet ist. Stattdessen entschied man sich für ein Vorgehen, das allgemein als Software Prototyping beschrieben wird.

Evolutionäres Prototyping

Prototyping ist ein Ansatz innerhalb des Software Engineering, der hauptsächlich im Fall von unklaren Anforderungen angewendet wird. Charakteristisch

für diese Methode ist die Erstellung eines Prototypen, der einige oder alle Leistungsmerkmale eines zu entwickelnden Produkts enthält [Spitta 1989, S. 4]. Die Erfahrungen, die mit dieser Vorversion gemacht werden, fließen erneut in den Softwareentwicklungsprozeß mit ein, d.h. die Anforderungsspezifikation wird ergänzt, das Design überarbeitet, etc. Das kann beliebig oft wiederholt werden, bis die Anforderungen schließlich vollständig sind. Der Prototyp dient also dazu, einen unscharfen Problembereich zu durchleuchten und Grundlagen für den Designprozeß zu schaffen. In [Abowd *et al.* 1995] wird in diesem Zusammenhang von iterativem Design gesprochen. Falsche Voraussetzungen, die beim Test des Prototypen festgestellt wurden, führen zu Modifikationen des Designs.

Grundsätzlich werden in der Literatur mehrere Arten des Prototypings unterschieden. Spitta [Spitta 1989, S. 5] teilt auf in die Bereiche exploratives, experimentelles und evolutionäres Prototyping, in [Abowd *et al.* 1995] werden drei ähnliche Ansätze genannt (Verwerfen, Inkrementell und Evolutionär) und Smith [Smith 1991] differenziert zwischen Throw-It-Away-Prototypen und evolutionären Prototypen. Allen Unterscheidungen ist gemeinsam, daß es auf der einen Seite Prototypen gibt, die komplett verworfen werden, und auf der anderen Seite Prototypen, die Basis für das gewünschte Endprodukt sind. Letzteres möchte ich wie Smith als evolutionäres Prototyping bezeichnen. Schrittweise wird das System getestet und überarbeitet, bis die Zielvorstellungen erreicht sind (Abbildung 1.6 [Abowd *et al.* 1995, S. 210]).

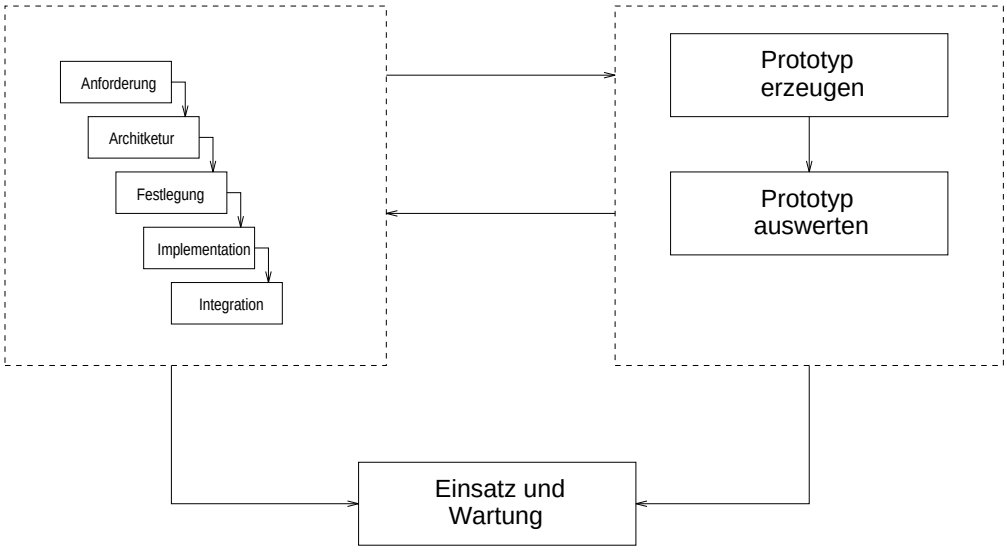


Abbildung 1.6: Evolutionäres Prototyping im Lebenszyklus

Die Gründe, die den Ausschlag gaben, evolutionäres Prototyping zu verfolgen, waren im wesentlichen:

- Es bestanden keine Erfahrungen im Umgang mit dem Roboter, speziell mit dem von ihm gelieferten Daten bzgl. der Anwendung von ILP-

Techniken.

- Ein komplett neuer Ansatz wurde gewählt, um den Roboter zu navigieren. Auch hier konnte auf keinen Erfahrungen aufgebaut werden.
- Die Arbeit mußte in einem kleinen Team unter großem Zeitdruck bewerkstelligt werden. Mit evolutionärem Prototyping konnten die aufwendige Anforderungsanalyse und die Erstellung des Systems verknüpft werden.

In [Smith 1991, S. 57] sind mehrere Umstände aufgelistet, die für eine Herangehensweise nach dem Prototyping-Ansatz sprechen. Die folgenden sind hier ausschlaggebend, sie stimmen im wesentlichen mit den ersten beiden, oben aufgeführten Punkten überein:

- Die Funktion einer Anwendung ist noch nicht vollständig verstanden worden.
- Der Mechanismus, wie eine gewünschte Funktion erbracht werden soll, ist noch nicht vollständig verstanden worden.

So entstand das Lern- und Performanzsystem in einzelnen, in sich abgeschlossenen Komponenten, die immer wieder den neuen Anforderungen angepaßt wurden. Allerdings tauschen die Module, wie bereits in Abschnitt 1.2.1 erwähnt, die Daten auf Dateiebene aus. Dies hat zur Folge, daß Änderungen in einem Modul u.a. viele andere Systemkomponenten betreffen. Zudem ist die Festlegung, Daten in Dateien abzulegen, vielleicht in Weiterentwicklungen nicht mehr aktuell. Schließlich kann es auch sinnvoll sein, die Daten in einer Datenbank zu halten. Gerade in großen Softwaresystemen, an denen mehrere Mitarbeiter beteiligt sind, spielen Datenabstraktion und Geheimnisprinzip eine große Rolle. Die Kommunikation zwischen Systembestandteilen wird so von der Implementierung entkoppelt und ist damit flexibler und wartungsfreundlicher. Um dieses Ziel zu erreichen, muß nun in einem nachträglichen Designschritt das bestehende System überarbeitet werden. Die Datenhaltung soll vom Rest des Systems getrennt werden, gleichzeitig ist eine konsequente Datenkapselung durchzusetzen.

1.3 Aufgabenstellung

Das entscheidende Motiv für diese Arbeit lag in den Problemen, die sich aus der Notwendigkeit, mit alternativen Datensätzen zu experimentieren, ergaben. Für den Benutzer ist es äußerst schwierig, den Überblick zu behalten und sich im Datenwust zurechtzufinden. Die Zusammenstellung der Lerndaten ist zeitintensiv und der Zugriff auf bestehende Daten gestaltet sich unnötig kompliziert. Was man vielmehr möchte, ist eine Entlastung des Benutzers von datenverwaltungstechnischen Aufgaben. Ein Werkzeug ist erforderlich, welches die im Lernprozeß anfallenden Daten verwaltet, den großen Datenpool strukturiert und vor allem einfache Zugriffe darauf erlaubt. Von der Systemseite her liegen die

Schwachstellen in der Intermodulkommunikation. Nötig ist eine Trennung der Datenhaltung von den Funktionsmodulen im Sinne von abstrakten Datentypen. Die implementationsabhängigen Details der Datenspeicherung sollen verborgen bleiben, die Systemkomponenten operieren über festdefinierte Zugriffsoperationen auf den Daten.

Im vorausgegangenen Abschnitt sollte klar geworden sein, warum existierende Unterstützungswerkzeuge nicht oder nur bedingt für diese Zwecke eingesetzt werden können. Es ist der spezielle Typus der Lernaufgabe, für den die meisten Lernumgebungen nicht konzipiert sind, der eben ganz neue Probleme aufwirft. Das Ziel meiner Arbeit ist also, eine allgemeine Umgebung für eine besondere Art von Lernszenario zu erstellen. Grundlagen dafür sind das im Projekt BLearn II entstandene Lernsystem und vor allem die damit gemachten Erfahrungen. Es waren ja gerade diese Erfahrungen, die die Problematik offenlegten und den Bedarf nach einer zentralen Datenverwaltung deutlich werden ließen.

An dieser Stelle muß noch einmal auf die Unterscheidung zwischen Lern- und Anwendungsphase des Roboters eingegangen werden. Ich beschäftige mich ausdrücklich nur mit dem Lernen, wenngleich auch die Anwendung nicht vollständig davon entkoppelt werden kann. Doch grundsätzlich haben die beiden Phasen bzgl. der Datenhandhabung ungleiche Zielsetzungen. Soll der Roboter sich in der realen Welt bewegen, so muß die Steuerung möglichst effizient arbeiten, d.h. Datenkapselung, Datenkonvertierung, etc. sind in diesem Fall hinderlich und verzögern die Programmausführung. Zudem müssen die Informationen, die aus den Sensormessungen gewonnen werden können, inkrementell verarbeitet werden. Direkt, wenn neue Daten über die Umwelt vorliegen, ist darauf zu reagieren. Der Roboter kann ja nicht vorausschauen, er hat kein vorgegebenes Bild des Raums, in dem er sich befindet. Und noch etwas kommt hinzu: Die Daten können früh wieder verworfen werden. Anders stellt sich dies beim Lernen dar. Die klassifizierten Roboterfahrten liegen stets komplett vor, alle relevanten Daten sind a priori gegeben. Der Benutzer kontrolliert die Lernläufe und sucht aus den Daten die gerade interessanten heraus. Immer wieder wird mit anderen Daten gelernt, die gewonnenen Ergebnisse werden ständig überprüft. Daraus ergeben sich die bereits geschilderten Schwierigkeiten. In der Performanzphase liegen die Lernresultate jedoch vor, die Massendaten sind also hauptsächlich ein Phänomen des Lernvorgangs. In diesem Sinne soll das zu entwerfende Werkzeug eine Lernumgebung sein. In Fällen, daß Module in beiden Phasen zur Anwendung kommen (z.B. die Berechnung der Basiswahrnehmungsmerkmale oder der Basishandlungsmerkmale), ist das mitzuberücksichtigen.

Welche Funktionen das Werkzeug zur Datenverwaltung im Gesamtsystem übernehmen soll, ist in Abbildung 1.7 schematisiert. Der Benutzer kommuniziert sowohl mit der Datenverwaltung als auch mit den Systembestandteilen, die anwendungsspezifisch sind. Über das Data Management Tool kann er sich Daten ansehen, Informationen über Daten erhalten, Datensätze konstruieren, usw., auf der anderen Seite stößt er Lernverfahren an, generiert neue Daten (z.B. Hintergrundwissen) oder arbeitet, wie beim Testen der Lernergebnisse, anderweitig auf den Daten. In der Abbildung ist in gestrichelten Linien ein Kasten dargestellt, der für eine graphische Benutzeroberfläche steht. Dies ist als Er-

weiterung zu meiner Arbeit zu sehen, und ich werde im letzten Kapitel eine mögliche Realisierung skizzieren. Das Werkzeug selbst ist interaktiv und stellt die Basis für eine solche Weiterentwicklung dar.

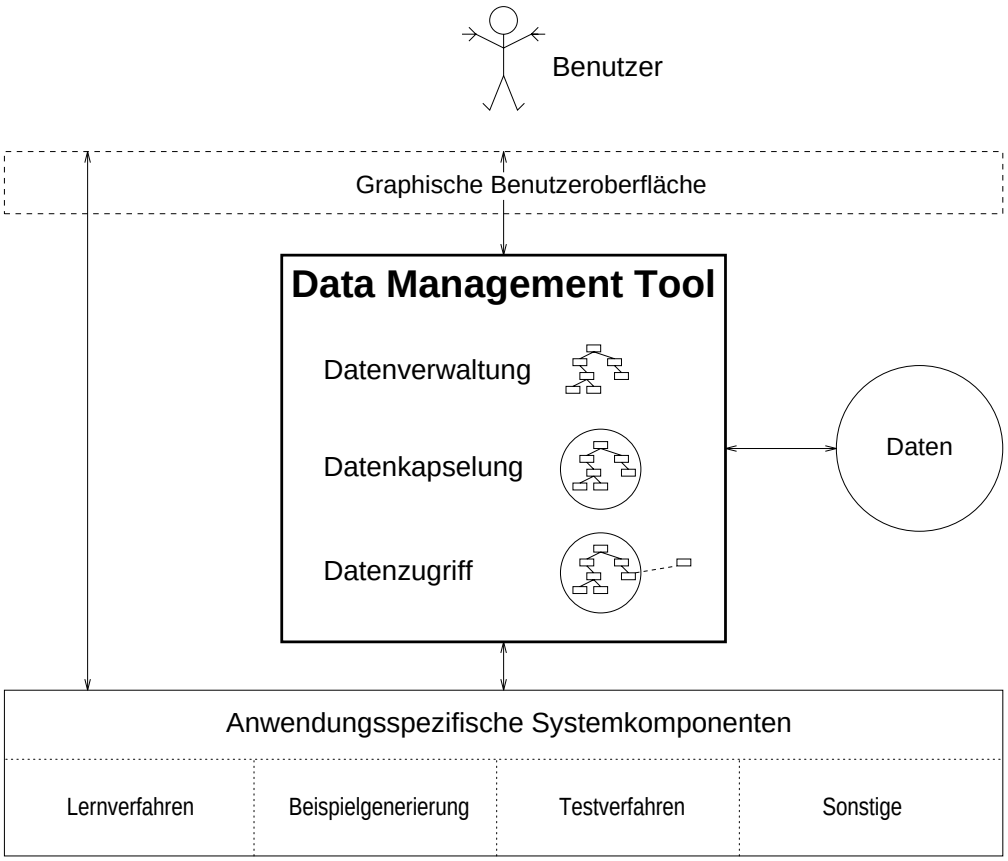


Abbildung 1.7: Einordnung des Werkzeugs in das Gesamtsystem

Im Gegensatz zur vorangegangenen Entwicklung des Lernsystems sind nun die Voraussetzungen klar abgegrenzt. Die traditionelle Vorgehensweise des Software Engineering kann schrittweise angewendet werden: Anforderungsanalyse, Design, Codierung.

Im ersten Schritt sind die Anforderungen des Benutzers und des Systems an die Datenverwaltung herauszuarbeiten und auf einen Nenner zu bringen. U.a. ist zu klären, welche Daten für den Benutzer verwaltet und vom anwendungsspezifischen Systemteil abgekapselt werden sollen. Wie möchte der Benutzer auf die Daten zugreifen, welche Operationen benötigt das System? Insbesondere bedarf es einer genauen Erfassung des bestehenden Systems, denn der BLearn-abhängige Teil muß vom Rest getrennt werden. Die Frage, die geklärt werden muß, ist: Was ist charakteristisch für ein Lernsystem dieser Art, wie sieht seine Struktur aus? Die Rahmenbedingungen, d.h. die zugrundeliegende Programmiersprache, die verwendeten Datenformate, etc., sind abzuklären und das Szenario, das sich

für den Benutzer stellt, genau zu beschreiben. Danach kann eine Anforderungsspezifikation erstellt werden. Sie muß auch die Design-Ziele beinhalten, also Kriterien, nach denen das fertige Werkzeug später bewertet werden kann.

Während des Entwurfs wird sich das Wechselspiel zwischen Benutzer- und Systemseite wiederholen. Die Anforderungen beider Seiten sind zu berücksichtigen, ggf. müssen Kompromisse eingegangen werden. Wichtige Entscheidungen diesbezüglich sind, wie die Daten organisiert und strukturiert werden, wie von ihnen abstrahiert wird, was zu kapseln und zu verwalten ist.

In der Implementierungsphase ist nach programmtechnischen Möglichkeiten zu suchen, mit denen die gesteckten Ziele erreicht werden können. In diesem Zusammenhang ist z.B. zu klären, wie der Mehrbenutzerbetrieb technisch realisiert werden soll. Konflikte, die auftreten, wenn zwei Benutzer gleichzeitig auf die gleichen Daten schreibend zugreifen wollen, sind aufzulösen.

1.4 Übersicht

In Kapitel 2 beschreibe ich, was das Werkzeug zur Datenverwaltung leisten soll. Ausgehend von den Anforderungen, die von der Benutzerseite und der Seite des Lernsystems gestellt werden, formuliere ich die Design-Ziele als Rahmen für die Konzeptgestaltung.

Gegenstand von Kapitel 3 ist die Ausführung meiner Lösungsidee, wie die drei Aspekte Datenverwaltung, Datenkapselung und Datenzugriff miteinander vereint werden können. Ich stelle ein übergreifendes Konzept vor, das einem Grobentwurf entspricht und das Fundament für die Programmentwicklung bildet.

Kapitel 4 behandelt das von mir implementierte Programm, das Data Management Tool. Dabei liegt der Schwerpunkt in der Beschreibung der äußeren Schnittstelle, sowohl zum Benutzer als auch zum Lernsystem hin.

Einzelheiten der technischen Realisierung werden in Kapitel 5 erläutert. Ich gehe auf die Programmarchitektur des Data Management Tools ein und beschäftige mich eingehender mit zwei grundlegenden Problemen bei der Implementierung.

Abschließend werfe ich in Kapitel 6 einen Blick zurück. Ich setze mich kritisch damit auseinander, inwiefern das Data Management Tool die gesteckten Ziele erreicht, wo es Verbesserungsmöglichkeiten gibt und wo der Nutzen im Vergleich zu anderen Systemen besteht. Des weiteren gebe ich einen Ausblick auf mögliche Arbeiten in der Zukunft.

Kapitel 2

Anforderungen an die Datenverwaltung

Nachdem im vorangegangenen Kapitel erläutert wurde, *warum* eine zentrale Datenverwaltung erforderlich ist, soll hier nun das Was geklärt werden (was soll die Datenverwaltung leisten?). Ich stelle in Abschnitt 2.1 das vorhandene Lernsystem dar und beschreibe somit den Ausgangspunkt meiner Arbeit. Dies beinhaltet die Erfassung der verschiedenen Systembestandteile sowie ihrer Kommunikation untereinander. Dabei sind insbesondere die Datenflüsse und die Datenformate von Interesse. Anschließend gehe ich auf die Konsequenzen für meine Arbeit ein, d.h. welchen Anforderungen die Datenverwaltung von der Systemseite her gerecht werden muß. In Abschnitt 2.2 ist das Ganze aus Sicht des Benutzers geschildert: Wie werden die Daten bislang gehalten, welche Probleme ergeben sich daraus und was erwartet der Anwender von einem Werkzeug zur Datenverwaltung? Der nachfolgende Abschnitt steckt dann die Rahmenbedingungen dafür ab und stellt u.a. ein Programm vor, das die Zusammenstellung der Lerndaten erheblich vereinfacht. Mit Hilfe dieses Programms kann ein wichtiger Implementierungsaspekt der Datenverwaltung erleichtert werden. Auf der anderen Seite ist es mit Blick auf den Benutzer wünschenswert, dieses Werkzeug mit der Datenhaltung abzustimmen. Das Kapitel endet schließlich mit der Beschreibung der Design-Ziele, also den Kriterien, an denen sich der Entwurf orientieren muß und nach denen später das fertige Programm bewertet werden kann (inwiefern wurden die gesteckten Ziele erreicht?).

2.1 Systemseite

Am Anfang stand für mich die Frage, wie das zugrundeliegende System eigentlich aussieht und welche charakteristischen Merkmale es aufweist. Die Schwierigkeit hierbei war, daß es sich um ein loses System handelt, dessen Module von verschiedenen Entwicklern konzipiert und implementiert wurden. Teilweise handelte es sich um Diplomarbeiten, so daß die Autoren auf absehbare Zeit für Fragen nicht mehr zur Verfügung standen. Ich habe daher ein Formular

entworfen, das als Orientierungshilfe bei der Systemanalyse diente und auf dessen Grundlage ich die erforderlichen Informationen über das System gewinnen konnte (Abbildung 2.1).

```
%+-----------------------------------------------------------------------------
%+ <Filename>: <Modulname>
%+
%+ AUTOR: <Name>
%+ STAND: <Datum>
%+-----------------------------------------------------------------------------
%+ MODULBESCHREIBUNG:
%+   - Was leistet das Modul?
%+   - Welche Aufgabenbereiche werden abgedeckt?
%+   - Wie ist das Modul in das Gesamtsystem einzuordnen?
%+
%+ EIN-/AUSGABEDATEN:
%+   - Welche Daten importiert das Modul, welche exportiert es?
%+   - Wie sehen die Daten aus, welche Entitaeten gibt es?
%+
%+ SCHNITTSTELLE:
%+   - Wie werden die Daten ausgetauscht?
%+   - Welche Prozeduren sind nach aussen sichtbar?
%+   - Wird auf andere Module zugegriffen, wie werden diese benutzt?
%+
%+ LAUFZEITBEDINGUNGEN:
%+   - Gibt es Voraussetzungen innerhalb des Systems, die waehrend der
%+     Laufzeit fuer dieses Modul erforderlich sind?
%+-----------------------------------------------------------------------------
```

Abbildung 2.1: Formular zur Modulspezifikation

2.1.1 Systemstruktur

Die für die Lernphase relevanten Module können im wesentlichen den Schritten im allgemeinen Lernszenario (vgl. Abbildung 1.2) zugeordnet werden:

1. Generierung der Daten zum Lernen, d.h. Beispiele und Hintergrundwissen,
2. maschinelles Lernen, womit die diversen Lernverfahren gemeint sind, und
3. Testen der Lernergebnisse.

Einige Module, z.B. zur Datenkonvertierung, fallen in keine der Kategorien und werden von mir unter „Sonstige“ aufgeführt.

Programme zur Datengenerierung erzeugen aus vorgegebenen Daten Beispiele und Hintergrundwissen. In der Regel werden die Eingabedaten aus einer Datei gelesen und die Ausgabedaten in eine andere Datei geschrieben. Da es sich hauptsächlich um Prolog-Programme handelt, müssen die Eingabedaten teilweise auch erst in die Prolog-Wissensbasis geladen werden. Das ändert jedoch

prinzipiell nichts an der datei-orientierten Ein- und Ausgabe. Es gibt Module zur Erzeugung von Basiswahrnehmungsmerkmalen, Basishandlungsmerkmalen, Sensormerkmalen und Sensorgruppenmerkmalen; auf den höheren Ebenen werden die Beispiele von Hand eingegeben. Eine dritte Möglichkeit der Datengenerierung besteht in der Anwendung von Regeln. Durch Rückwärtsverkettung lassen sich die benötigten Daten herleiten, wobei als Eingabe eine Menge von Regeln (Regeldatei), Beispiele und Hintergrundwissen sowie eine Liste von Prädikatsnamen benötigt werden.

Bei den Lernverfahren stellt sich die Situation ähnlich dar, wenn auch Unterschiede bestehen. Die ILP-Verfahren erwarten die Lerndaten in einer Datei, die in Prolog, MOBAL oder sonstiges eingelesen wird. Die Beispiele und das Hintergrundwissen müssen in dem geforderten Format vorliegen, die Formate sind teilweise unterschiedlich. Um einen Lernlauf zu starten, sind zudem noch das Metawissen, das Akzeptanzkriterium und das Lernziel zu spezifizieren. Das Metawissen ist in Dateien abgelegt und vom Lernverfahren abhängig. RDT benötigt Regelschemata, GRDT Grammatiken. Weiterhin besteht das Akzeptanzkriterium aus einzelnen Parametern, während das Lernziel durch eine mindestens einelementige Menge von Prädikatsnamen (plus Stelligkeit) beschrieben wird. Die Ausgabe des Lernalgorithmus setzt sich zusammen aus den gelernten Regeln und einem Protokoll. Die Regeln werden auch in Dateien abgelegt, ihr Format ist abermals programmspezifisch.

Die Lernschleife zum Lernen der Parameter, die die Berechnung der Basiswahrnehmungsmerkmale steuern, gestaltet sich anders (siehe [Wessel 1995]). Bei gegebenen Roboterdaten werden iterativ Basiswahrnehmungsmerkmale und daraus höhere Konzepte berechnet. Durch Variation der Programmparameter und Bewertung der Lernergebnisse auf höheren Ebenen wird schließlich der optimale Parametersatz ermittelt. Eingabe sind also die Roboterdaten, Ausgabe ist der Parametersatz.

Unter den dritten Punkt, dem Testen, fallen Programme zur Überprüfung von gelernten Regeln. Notwendig hierfür sind natürlich die Regeln selbst wie auch Beispiele und Hintergrundwissen, analog zum Lernen. Des weiteren spezifiziert eine Liste von Prädikatsnamen die zu bewertenden Regeln. Ausgabe ist schließlich ein Protokoll mit dem Ergebnis der Evaluation.

Unter die sonstigen Module fallen u.a. Programme zur Datenkonvertierung. Beispielsweise müssen die Rohdaten, die der Roboter liefert, erst in ein einheitliches Format gebracht werden, bevor sie verarbeitet werden können. Andere Programme, wie z.B. Visualisierungstools, sind in diesem Kontext nicht von Interesse.

In Abbildung 2.2 ist das Zusammenspiel der Systemkomponenten untereinander noch einmal vereinfacht dargestellt.

2.1.2 Datenbeschreibung

Aus Abbildung 2.2 geht auch hervor, daß es im wesentlichen fünf Arten von Daten gibt. Für die Datenverwaltung sind allerdings nur die konvertierten Daten

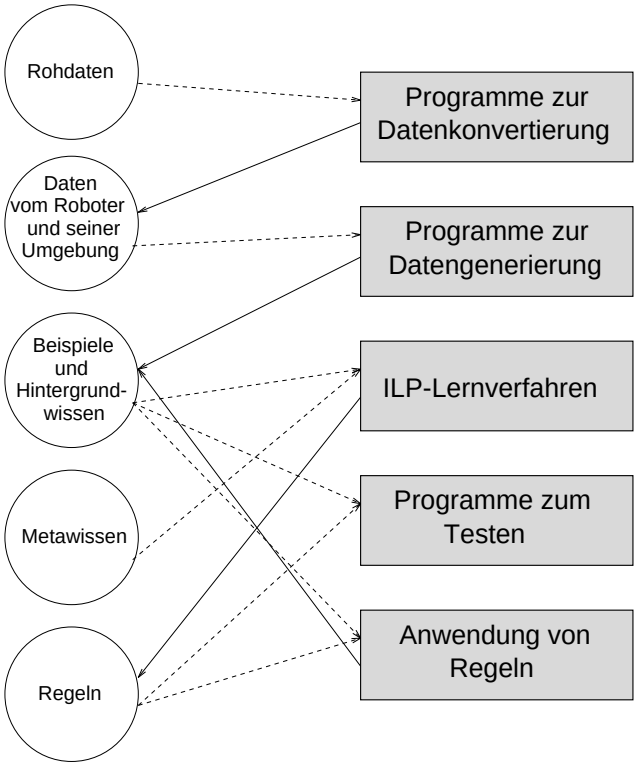


Abbildung 2.2: Daten und Programme

vom Roboter und seiner Umgebung, die Beispiele und das Hintergrundwissen sowie die Regeln von Belang. Die Roboterrohdaten spielen insofern keine Rolle, als daß das System sie in dieser Form nicht benutzt und nur mit den konvertierten Daten arbeitet. Das Metawissen ist stark von den verwendeten Lernverfahren abhängig. Es macht keinen Sinn, Daten zu kapseln, die an ein einzelnes Modul gebunden sind und in einem speziellen Format vorliegen müssen. Der Benutzer erzeugt beispielsweise Regelschemata mit einem Editor und speichert sie in Dateien. Das Lernverfahren liest diese Dateien dann ein. Da also nur der Anwender und das Lernmodul beteiligt sind, ist Datenkapselung in diesem Fall nicht erforderlich.

Basis für alle weiteren Berechnungen sind die konvertierten Daten vom Roboter und seiner Umgebung. Sie sind durch Prolog-Fakten repräsentiert und lassen sich in drei Gruppen aufteilen: Sensordistanzmessungen, Roboterpositionen und Szenedaten. Die Sensordistanzmessungen werden dargestellt durch `messung/13`-Fakten und beziehen sich jeweils auf einen Pfad, einen Zeitpunkt und einen Sensor. Da es zu weit gehen würde, die Repräsentationsdetails haarklein zu erläutern, gebe ich stattdessen in Anhang A eine knappe Übersicht über die Datenformate. (Für die Datenverwaltung ist es ohnehin unerheblich, was die Daten im einzelnen aussagen oder bedeuten; sie sollen ja nur in syn-

taktischer Hinsicht betrachtet werden.) Die Roboterpositionen geben Auskunft über einen Pfad, denn sie beinhalten Informationen zu den Koordinaten und der Orientierung des Roboters. Demgegenüber beschreiben die Szenedaten die Räume, in denen die Robotertestfahrten durchgeführt wurden.

Beispiele und Hintergrundwissen werden ebenfalls durch Prolog-Fakten dargestellt. Wie sie auf den einzelnen Ebenen der Repräsentationshierarchie aussehen, kann wieder dem Anhang entnommen werden.

Um die Zusammenhänge zwischen den Daten deutlicher zu machen, habe ich in einem gerichteten Graphen Ein- und Ausgabedaten in Beziehung gesetzt (Abbildung 2.3). Die Kästchen, d.h. die Knoten des Graphen, stehen für die verschiedenen Arten von Fakten, quasi Klassen von Daten. Neu hinzugekommen sind dabei die Nachfolgerrelationen, die Sensorklassen und die Richtungsrelationen. Sie werden zum Lernen der Sensorgruppenmerkmale bzw. der operationalen Begriffe als Hintergrundwissen benötigt.

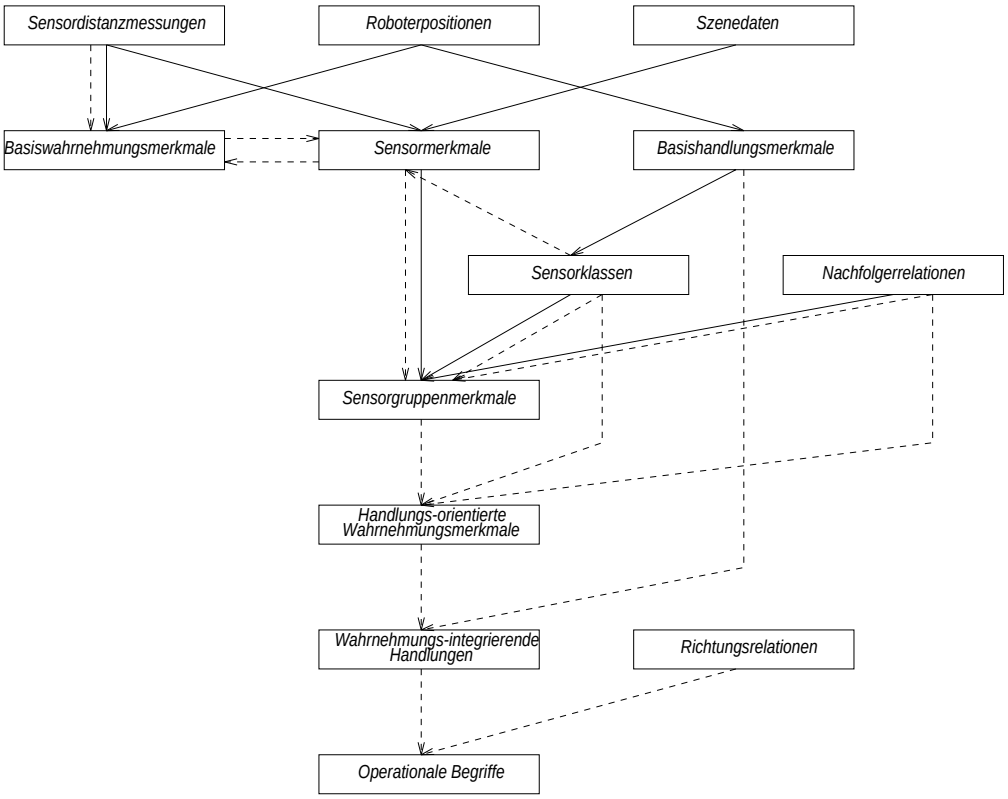


Abbildung 2.3: Zusammenhänge zwischen den Daten

In dem Graphen gibt es zwei Pfeiltypen. Die durchgezogenen Kanten bedeuten „wird benötigt zur Berechnung von“ und spiegeln somit den Prozeß der Datengenerierung wider. Daraus ist ersichtlich, daß Beispiele für Sensormerkmale aus den Szenedaten und den Sensormessungen erzeugt werden, Basiswahrneh-

mungsmerkmale hingegen aus den Roboterpositionen. Besitzt ein Knoten keinen eingehenden durchgezogenen Pfeil, so sind die Daten auf dieser Ebene entweder vorgegeben oder von Hand erzeugt.

Die gestrichelten Pfeile veranschaulichen die Abhängigkeiten beim Lernen. Sie stehen für die Relation „ist Hintergrundwissen zum Lernen von“ und geben an, was zum Lernen auf einer Ebene gebraucht wird. Beispielsweise benötigt RDT Beispiele von Sensorgruppenmerkmalen und Hintergrundwissen durch Vorgabe von Sensormerkmalen, der Nachfolgerrelation und den Sensorklassen, um Sensorgruppenmerkmale zu lernen.

Als letztes bleiben noch die Regeln übrig. Wie gesagt, die Lernverfahren haben zumeist auch hier ihr eigenes Format, die Darstellung von Regeln ist also nicht vereinheitlicht. Deshalb entschieden wir uns bei der Datenverwaltung dafür, Regeln stets in Prolog-Notation aufzuführen. So wird die Unabhängigkeit vom verwendeten Lernalgorithmus gewährleistet.

2.1.3 Anforderungen

Wie aus dem vorangegangenen Abschnitt ersichtlich wird, geht es um eine Verwaltung von Fakten und Regelmengen. Aus der Sicht des Systems ist dabei vor allem die Datenkapselung von entscheidender Bedeutung. Die Art der Datenspeicherung soll verborgen bleiben, es darf möglichst keine Rolle spielen, ob die Daten in Dateien oder einer Datenbank abgelegt werden[1]. Es sind also Routinen erforderlich, mit denen Fakten und Regeln zum Datenpool hinzugefügt und aus ihm entfernt werden können. Fakten müssen zudem als positiv oder negativ gekennzeichnet werden können, schließlich gibt es positive und negative Beispiele.

Darüber hinaus sollte vom konkreten Aussehen der Fakten und Regeln abstrahiert werden, um die Daten noch weiter von den Funktionsmodulen zu entkop-

[1] Die Art der Modulkommunikation legte nahe, bei der Datenverwaltung ebenfalls Dateien einzusetzen—die Daten liegen ja schon in dieser Form vor. Das spricht jedoch nicht unbedingt gegen den Einsatz einer Datenbank. Tatsächlich wurde es bereits früher einmal erwogen, die Daten in einer Datenbank zu halten. Diese Idee mußte allerdings verworfen werden, weil damit die Handhabung alternativer Datensätze, also verschiedener Versionen von Daten, erschwert oder gar verunmöglicht wird.

Bei der Entwicklung meiner Arbeit wurde dieser Aspekt neu aufgegriffen, doch es stellte sich heraus, daß eine Realisierung den zeitlichen Rahmen gesprengt hätte. Auf der einen Seite war ich schon zu weit fortgeschritten, um das Design wieder zu ändern, auf der anderen Seite wären die sich daraus ergebenden Aktivitäten sehr zeitintensiv gewesen (Anbindung an die Datenbank, Überführung der Daten in die Datenbank, etc.) Die Datenhaltung mittels einer Datenbank ist in diesem Kontext eben programmtechnisch komplizierter umzusetzen.

Auch wenn Datenbanken gerade für Massendaten ausgelegt sind und effiziente Datenzugriffe (z.B. SQL-Anfragen) erlauben, hat die Verwendung von Dateien trotzdem Vorteile. Dateien unterstützen die Aufteilung des Datenpools in Datensätze optimal und erlauben eine flexiblere Anpassung an neue Datenformate. Das Data Management Tool läßt sich deshalb sofort in neuen Umgebungen einsetzen, ohne daß wie bei einer Datenbank entsprechende Tabellen angelegt oder modifiziert werden müssen.

Es wird sich wohl erst in der Praxis zeigen müssen, ob sich der Aufwand für die Benutzung einer Datenbank lohnt. Zumindest gewährleistet die Datenkapselung, daß dieser Schritt später möglich ist.

peln. Zugriffsoperationen, die Fakten und Regeln gemäß ihrer Struktur zerlegen und zusammensetzen können, müssen zur Verfügung gestellt werden.

Nehmen wir ganz konkret ein Fakt, das ein Basiswahrnehmungsmerkmal repräsentiert:

```
stable(t701, 75, s0, 5, 23, 0)
```

Das erste Argument identifiziert den Trace, das zweite gibt die Orientierung an, das dritte bezeichnet eine Sensorkennung, es folgen Anfangs- und Endzeitpunkt sowie zum Schluß ein Gradient. Möchte ein Modul auf das erste Argument zugreifen, also die Tracekennung, so könnte das in Prolog so aussehen:

```
stable(Trace, _, _, _, _, _)
```

Die Variable `Trace` wird dann über Unifikation mit dem Term `t701` belegt.

Diese Methode hat den Nachteil, daß alle Module, die Basiswahrnehmungsmerkmale verarbeiten, wissen müssen, an welcher Position die Tracekennung steht. Was passiert aber, wenn sich Änderungen in der Repräsentation ergeben und die Tracekennung auf einmal an die zweite Argumentstelle „rutscht"? Dann muß der Quelltext aller beteiligten Module überarbeitet werden—eine unnötige und lästige Arbeit.

Eine Alternative ist, das Aussehen der Fakten zu deklarieren und nicht mehr in Abhängigkeit von Argumentpositionen, sondern von Argumenttypen zuzugreifen. Man könnte eine Routine `get_trace` schreiben, die in den Deklarationen nachsieht und ermittelt, wo in einem bestimmten Fakt die Tracekennung steht. Dann würde in den Modulen, um bei unserem Beispiel zu bleiben, folgende Zeile stehen:

```
get_trace(stable(t701, 75, s0, 5, 23, 0), Trace)
```

Sie erfüllt den gleichen Zweck wie die oben stehende Zeile, abstrahiert aber von den Argumentpositionen. Jedoch ist auch diese Methode nicht optimal, weil Wissen über die Daten in den Zugriffsroutinen fest codiert ist. In `get_trace` steckt z.B. implizit die Annahme, daß die Fakten sich auf Traces beziehen—das sollte vom Programm entkoppelt werden. In Abschnitt 3.6 werde ich eine flexiblere Lösung vorstellen, hier geht es zunächst einmal nur um die Unterscheidung zwischen Zugriffen über Argumentpositionen und Argumenttypen.

Eine weitere wichtige Forderung an die Datenverwaltung ergibt sich aus der Zeitorientierung der Daten. Wenn sich der Roboter bewegt (oder auch steht), werden in gewissen Zeitabständen die Sensormessungen erfaßt. Es ensteht also eine Kette von Wahrnehmungen, die stets in dieser Reihenfolge betrachtet werden muß. Werden nun Daten zum Lernen zusammengestellt, ist zu garantieren, daß die zeitliche Reihenfolge erhalten bleibt. Eine mögliche Lösung wäre beispielsweise, eine Sortierung nach Zeitpunkten anzubieten. Des weiteren dürfen innerhalb einer solchen Kette keine doppelten Fakten auftreten, weil einige Module davon ausgehen, daß zu jedem Zeitpunkt höchstens ein Fakt existiert.

Als letztes bleibt die Notwendigkeit einer gezielten Datenauswahl zu nennen. Werden die Daten in Dateien abgelegt und in dieser Form verwaltet, so muß es möglich sein, bestimmte Ausschnitte des Datenpools in den Hauptspeicher zu laden; Ziel ist also die Eingrenzung auf die relevanten Daten. Nur so kann verhindert werden, daß ähnliche Speicherprobleme wie z.B. bei MOBAL auftreten.

2.2 Benutzerseite

Wie gestaltet sich nun für den Benutzer die Arbeit mit dem System? In der Regel erzeugt er sich zunächst die erforderlichen Lerndaten, indem er entsprechende Programme zur Beispielgenerierung startet oder die Daten über einen Editor eingibt. Er erhält so eine Menge von Dateien, aus denen er schließlich von Hand oder über Filterprogramme die relevanten Fakten zu Lern- und Testsets verknüpft. Anschließend müssen die Daten ggf. noch in ein anderes Format umgewandelt werden, bevor die Lernverfahren angestoßen werden können. Die gelernten Regeln werden getestet, und evtl. folgt ein weiterer Lernlauf.

Die Probleme, die im Umgang mit den Dateien entstehen, sind vielschichtig. Es fängt bereits bei der Erzeugung von Beispielen und Hintergrundwissen an. Jeder Benutzer erstellt seine Dateien für sich und ist selbst dafür verantwortlich zu protokollieren, was in einer Datei enthalten ist. Doch gesetzt der Fall, er weiß es, so kann trotzdem Konfusion entstehen, wenn er nämlich vor dem Bildschirm sitzt und sich fragt: „Verdammt, aus welchen Ursprungsdaten, mit welchem Programm und welchen Programmparametern habe ich die Datei damals generiert?“. Im Zweifelsfall wird er die gewünschten Daten ein weiteres Mal berechnen lassen.

Je mehr Dateien sich aber nun im Verzeichnis befinden, desto unübersichtlicher wird die Angelegenheit, und desto wahrscheinlicher ist es, daß identische Daten mehrfach abgespeichert werden. Die Konsequenz ist Verschwendung des Plattenspeichers, was angesichts der Unmenge an Daten auch bei heutigen Speicherkapazitäten nicht tragbar ist. Sinnvoller wäre, sich immer eine größere Anzahl von Beispielen kreieren zu lassen und aus diesen temporär die jeweils interessanten herauszusuchen. Benötigt würden Zugriffsroutinen, mit deren Hilfe gezielt Fakten und Regeln aus einer Datei herausgefiltert werden können (vgl. hierzu auch Abschnitt 2.3.2).

Vor allem sollte der arme Benutzer mit den zerrauften Haaren von dem Dateien-Chaos befreit werden. Zu jedem Datensatz müßte vermerkt sein, wer ihn erstellt hat (Benutzername), wann er erstellt wurde (Datum), woraus er berechnet wurde (Ursprungsdaten), und wie er erstellt wurde (von Hand oder mittels eines Programms). Zudem wäre ein Benutzerkommentar sinnvoll, der zusätzliche Informationen beinhalten kann.

Wenn die Datenverwaltung dies unterstützt, kann der Benutzer den Überblick behalten und die Speichervergeudung läßt sich verringern. Des weiteren hat es den Vorteil, daß mehrere Benutzer mit den Dateien arbeiten können. Vorher wurden die Dateien zwar in einem zentralen Verzeichnis gehalten, doch der

Dateiname allein ist häufig nicht aussagekräftig genug, um Schlußfolgerungen über den Inhalt zuzulassen. Über die Datenverwaltung ist es auch anderen Leute möglich zu erfahren, um welche Daten es sich konkret handelt. So können alle mit den gleichen Daten arbeiten, denn es ist mühsam, wenn jeder für sich die Daten generieren muß.

Was u.a. auf der Systemseite gefordert wird, gilt auch hier: Die Implementierungsdetails der Datenspeicherung sollten verborgen bleiben, und vom Aussehen der Daten ist zu abstrahieren. Mit anderen Worten: Auch wenn die Fakten und Regeln intern über eine Dateistruktur verwaltet werden, sollte der Anwender nichts von den Dateien merken. Zum anderen kann kein Mensch die Einzelheiten der Repräsentation ständig präsent haben. Man kann zwar voraussetzen, daß er weiß, wie beispielsweise ein Sensormerkmal charakterisiert ist, doch was an welcher Argumentstelle steht, wird er wohl kaum behalten können. Folglich sollten auch aus bedienerfreundlichen Gründen Prädikate deklariert werden, um auf Fakten über Argumenttypen zugreifen zu könnnen. Wünschenswert wäre u.a. eine Funktion, die ein Fakt in seine Bestandteile zerlegt. Wenn der Benutzer dann die Prozedur mit der Eingabe `stable(t701, 75, s0, 5, 23, 0)` aufruft, könnte folgende Ausgabe erscheinen:

```
<predname>       = stable
<trace>          = t701
<orientierung>   = 75
<sensor>         = s0
<startzeitpunkt> = 5
<endzeitpunkt>   = 23
<gradient>       = 0
```

In Bezug auf die Probleme, die mit den unterschiedlichen Formaten der Lernverfahren einhergehen, ist zu sagen, daß die Datenverwaltung nur eine einheitliche Form unterstützen sollte. Die Daten liegen dann in normierter Weise vor, so daß sie ggf. in andere Repräsentationen überführt werden können. Das ist wesentlich effektiver, als alle Daten immer in das aktuelle Format zu transformieren und schließlich entnervt im „Konvertierungswust“ zu versinken. Ein einheitliches Format stellt einen Bezugspunkt dar, der eine konsistente Datenhaltung ermöglicht. Wie ich bereits mehrfach erläutert habe, soll für Fakten und Regeln die Prolog-Notation gelten.

Um die Anforderungen aus der Perspektive des Benutzers noch einmal zusammenzufassen: Ziel ist eine zentrale Datenverwaltung, die den Überblick über die Daten sicherstellt und somit der Verschwendung von externem Speicherplatz vorbeugt. Zentral, weil mehrere Anwender mit den Daten gleichzeitig arbeiten und die enorme Datenmenge eine Duplizierung des Datenbestands verbietet. Die Art der Speicherung ist zu verbergen und entsprechend muß auch ein Konzept bestehen, daß einerseits den Datenpool strukturiert und andererseits die Möglichkeit eröffnet, Daten in Einheiten (den Varianten) zusammenzufassen. Zu jeder Variante muß festgehalten werden, wer sie wann, wie und woraus erzeugt hat, und es ist zu gewährleisten, daß Varianten nicht mehrfach abgespeichert werden. Der Benutzer soll sich die Beschreibung der Datensätze

aufgelistet ansehen und nach bestimmten Varianten suchen können. Weiterhin muß vom konkreten Aussehen der Fakten und Regeln abstrahiert werden, um eine größtmögliche Entkopplung von den Daten zu erreichen. Unter Zuhilfenahme von Deklarationen sollen die Daten zerlegt und zusammengesetzt werden. Schließlich will man auch noch aus den extern abgelegten Daten Auswahlen treffen und Selektionen in den Hauptspeicher laden. Dabei sind auch mehrstufige Selektionen denkbar. Umgekehrt macht es u.U. auch Sinn, ausgewählte Daten extern als eine Variante abzuspeichern. Die Eingrenzung auf kleine Datenbereiche vermeidet eine Überlastung des Hauptspeichers und erleichtert dem Benutzer die Zusammenstellung der Lerndaten.

2.3 Rahmenbedingungen

In diesem Abschnitt möchte ich kurz auf die äußeren Gegebenheiten eingehen, die den Entwurf des Werkzeugs zur Datenverwaltung beeinflußten. Dies ist zum einen die Plattform, auf der das Programm entwickelt wurde, und zum anderen ein Tool, das einen Teilbereich des Aufgabenfeldes, nämlich die Auswahl von Fakten, abdeckt.

2.3.1 Entwicklungsumgebung

Die meisten Module des Systems sind in Prolog codiert, einige auch in C. Grundlage dafür ist das UNIX-Betriebssystem. Da auch die zu verwaltenden Daten in Prolog-Notation vorliegen, ist es naheliegend, die Datenverwaltung ebenfalls in Prolog einzubetten. Zumindest muß eine Prolog-Schnittstelle existieren, damit die Module die Datenzugriffsroutinen aufrufen können.

Als Entwicklungsumgebung stand Quintus-Prolog[2] in der Version 3.2 zur Verfügung. Es erlaubt die Einbindung von C-Code, was wichtig ist, um UNIX-Systemcalls realisieren zu können.

2.3.2 Das Data Preparation Tool

Am Lehrstuhl für Künstliche Intelligenz der Universität Dortmund ist ein Werkzeug entwickelt worden, das den Benutzer bei der Zusammenstellung von Lerndaten unterstützt [Rieger 1995][Rieger 1996]. Es benutzt Hornklauseln als Repräsentationsformalismus und ist in Prolog codiert.

Zunächst einmal erlaubt das Tool, die Daten zu strukturieren. Dafür führt Rieger die *Abstraction Levels* ein, die Gruppen von Prädikaten unter einem Begriff zusammenfassen. Diese Abstraktionsebenen können mit den Stufen in der Repräsentationshierarchie (siehe Abbildung 1.3) verglichen werden. Merkmal eines Abstraction Levels ist, daß alle Prädikate die gleiche Stelligkeit sowie eine einheitliche Argumentstruktur aufweisen. Beispielsweise können Basiswahrnehmungsmerkmale als sechsstellige Prädikate beschrieben werden, mit der Argu-

[2]Quintus Corporation, an Intergraph Company, California, USA

mentfolge Tracekennung, Orientierung, Sensorkennung, Startzeitpunkt, Endzeitpunkt, Gradient; `increasing`, `stable`, `decreasing`, etc. sind als Prädikatsnamen zugelassen. Diese Definitionen haben den Sinn, daß der Benutzer nicht immer angeben muß, welche Daten er im einzelnen betrachten will. Es genügt die Angabe des Abstraction Levels, um die Daten einzugrenzen.

Die Auswahl bzw. die Zusammenstellung von Daten kann auf dreierlei Art und Weise vorgenommen werden: durch syntaktische Auswahl, statistische Auswahl oder Fallauswahl (in den englischsprachigen Originaltexten steht *Case Selection*).

Angenommen wir haben bereits alle erforderlichen Daten erzeugt und sie in Dateien abgelegt. Wie können wir nun alle Basiswahrnehmungsmerkmale zum Trace `t701` heraussuchen? Dies ermöglicht die `select`-Anweisung, die folgendermaßen aussieht:

```
select(<source>, <reference>, <constraints>, <sample>)
```

Das erste Argument bezieht sich auf die Datenquelle, aus der die Daten stammen. Das kann ein Programm sein, welches die Fakten generiert, es kann eine Datei sein, die die Fakten beinhaltet, oder auch ein bereits erstelltes Sample, unter dem die Fakten zusammengefaßt sind. Als ein *Sample* bezeichnet Rieger eine Zusammenstellung von positiven und negativen Fakten im Hauptspeicher; Samples stellen das Ergebnis einer syntaktischen Auswahl dar. Als zweites Argument ist die Abstraktionsebene anzugeben, die festlegt, welche Fakten überhaupt von Interesse sind. Wie die gesuchten Daten aussehen, beschreiben die *Constraints*—so werden in [Rieger 1995] Angaben über die erlaubten Argumente genannt. Für jede Argumentposition kann eine Menge von Werten spezifiziert werden, die an dieser Stelle vorkommen dürfen.

In unserem Beispiel müßten wir sagen, daß an erster Stelle nur der Term `t701` vorkommen darf. Die entsprechende Anweisung sieht dann so aus:

```
select(file(bpf1),
       alevel(basic_perception_features/6),
       [arg(1, [t701])], SampleId)
```

Das Tool wird die Datei `bpf1` durchsuchen und alle Fakten, die Basiswahrnehmungsmerkmale zum Trace `t701` sind, im Hauptspeicher als ein Sample ablegen. Die Kennung des Samples wird zurückgegeben, mit der der Benutzer dann auf die Daten zugreifen kann. (Intern werden die Fakten in zwei Listen für positive und negative Beispiele gehalten.)

Aus dem erstellten Sample kann erneut ausgewählt werden. Möchten wir nur Fakten bzgl. der Sensoren `s0` und `s1` betrachten, dann würde

```
select(sample(SampleId),
       alevel(basic_perception_features/6),
       [arg(3, [s0, s1])], SecondSampleId)
```

das Gewünschte liefern.

Bei der syntaktischen Auswahl werden die Daten also aufgrund bestimmter Merkmale (Prädikatsname, Stelligkeit, Argumente) extrahiert. Demgegenüber können Daten auch nach statistischen Kriterien zusammengestellt werden. Z.B. kann aus einem gegebenen Sample zufällig ein gewisser Prozentsatz an negativen und an positiven Fakten herausgefiltert werden. Die statistische Auswahl war allerdings zu dem Zeitpunkt, als ich mit dem Design der Datenverwaltung begann, noch nicht implementiert. Sie mußte deshalb in meiner Arbeit unberücksichtigt bleiben.

Als letztes bleibt die Case Selection zu erwähnen. Sie dient dazu, Beispiele und Hintergrundwissen geeignet in Beziehung zu setzen. Für jedes Beispiel gibt es relevante Fakten aus dem Hintergrundwissen. Um die Lernverfahren nicht mit unnötigen Daten zu überlasten, ist es wichtig, nur das wirklich erforderliche Hintergrundwissen einzugeben. Daher bietet die Case Selection die Möglichkeit, zu jedem Beispiel das „interessante“ Hintergrundwissen herauszusuchen.

Als Eingabe ist dazu ein Sample mit den Beispielen, ein Sample mit dem Hintergrundwissen und eine Relationenliste nötig. Die Liste mit den Relationen drückt aus, wie die Argumentwerte eines Hintergrundwissen-Fakts beschaffen sein müssen, um für ein Beispiel-Fakt relevant zu sein. Da aber verschiedene Beispiele vorliegen, kann das nicht über die Angabe konkreter Argumentwerte bewerkstelligt werden. Es muß vielmehr der Bezug zwischen den Argumenten des Beispiel-Fakts und denen der Hintergrundwissen-Fakten hergestellt werden.

Nehmen wir Basiswahrnehmungsmerkmale und Sensormerkmale (dieses Beispiel stammt aus [Rieger 1995, S. 12]). Ein Basiswahrnehmungsmerkmal ist relevant für ein Sensormerkmal, wenn es sich auf den gleichen Trace, den gleichen Sensor und das gleiche Zeitintervall bezieht. Dies läßt sich als Folge von Relationen zwischen Argumenten ausdrücken (Abbildung 2.4).

Das Ergebnis der Case Selection ist eine Liste von Fällen (*Cases*), jeder Fall besteht aus einem Beispiel und dem dazugehörigen Hintergrundwissen.

Da das Data Preparation Tool effektive Methoden zur Datenselektion anbietet, sollte es mit der Datenverwaltung abgestimmt werden. Insbesondere ist zu prüfen, ob beide Dinge integriert werden können. Dabei ist bzgl. der Datenauswahl allerdings folgendes zu beachten:

- Von den Argumentstellen ist zu abstrahieren, Argumenttypen sollen verwendet werden.
- Eine Datenauswahl sollte auch Prädikate unterschiedlicher Stelligkeit und Argumentstruktur beinhalten können.
- Auswahlen sollen nicht nur auf Faktenmengen, sondern auch auf Regelmengen getroffen werden können.

Das entspricht den Anforderungen, die von der System- und der Benutzerseite an die Datenverwaltung gestellt werden.

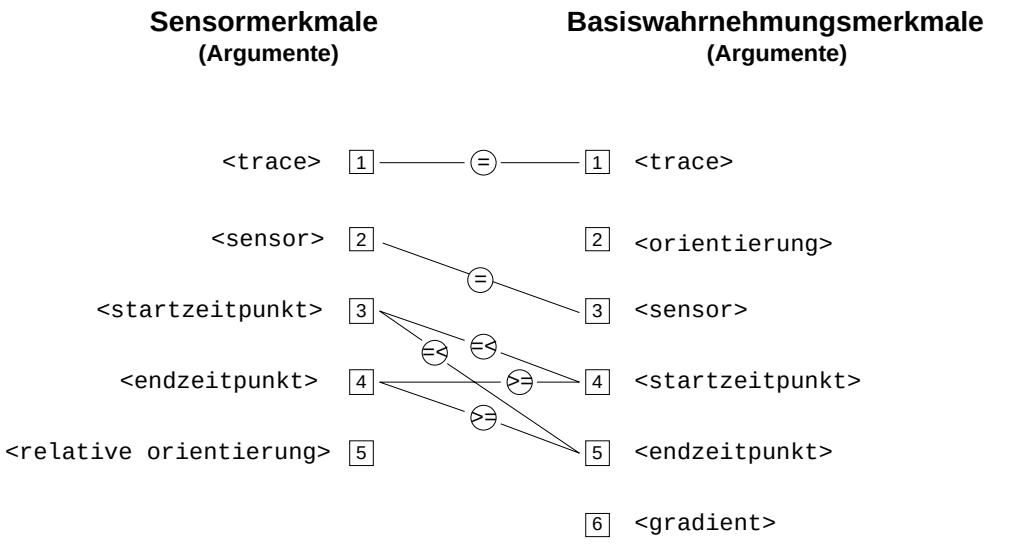


Abbildung 2.4: Case Selection

2.4 Design-Ziele

Nachdem die Ausgangssituation aus System- und Benutzersicht geschildert wurde und auch die Rahmenbedingungen abgesteckt sind, müssen nun die Design-Ziele formuliert werden. Damit sind im wesentlichen die Aufgaben gemeint, die das Data Management Tool (im weiteren kurz DMT genannt) erfüllen soll. Letztendlich ist es auch daran zu messen.

Als Ziele haben sich herauskristallisiert:

Strukturierung des Datenpools: Der Datenbestand muß durch das DMT strukturierbar sein, d.h. es sollte möglich sein, bestimmte Klassen von Daten zu bilden. Das erlaubt dem Benutzer einerseits den Datenpool übersichtlich aufzuteilen, andererseits wird somit die Vermischung ungleicher Daten vermieden. Diese Forderung entspringt hauptsächlich der Vorstellung, daß sich die Repräsentationshierarchie in der Datenhaltung widerspiegelt. Jeder einzelne Abstraktionsschritt steht ja für eine Art von Daten, und der Benutzer sollte auf jede dieser Arten gesondert zugreifen können. Somit sind auch für das Lernen Beispiele und Hintergrundwissen getrennt.

Alternative Datensätze: Es muß dem Benutzer möglich sein, Dateneinheiten zu bilden, also Daten zu Datensätzen zusammenzufassen. Verschiedene Datensätze müssen nicht disjunkt sein, sie stellen eher Alternativen im Sinne von möglichen Varianten dar. Dies ist motiviert durch die Besonderheit der Lernaufgabe, weil man eben immer wieder alternative Lernsets austestet, um das Lernergebnis zu optimieren. Zu jedem Datensatz

müssen die folgenden Zusatzinformationen verwaltet werden:

- der Zeitpunkt der Erstellung,
- der Name des Benutzers, der den Datensatz angelegt hat, und
- ein Benutzerkommentar, der zur Erläuterung dient.

Damit soll gewährleistet sein, daß der Benutzer die Datensätze später wieder identifizieren kann.

Effiziente Speicherung: Dies ist an sich ein Kriterium, das einer Bewertung schlecht zugänglich ist, denn was heißt schon effizient. Natürlich sollte eine Datenhaltung mit möglichst wenig Speicherplatz auskommen und dabei schnelle Datenzugriffe erlauben—ein typischer *Trade Off* zwischen Speicher- und Zeitanforderungen. Doch hier geht es vielmehr darum, das mehrfache Vorkommen identischer Daten zu vermeiden. Ein Problem bei der Arbeit mit den Dateien war ja, daß Benutzer wegen der Unübersichtlichkeit und aus Unkenntnis noch einmal Daten generierten, die bereits vorlagen. Das DMT sollte in der Lage sein, doppelte Datensätze zu erkennen, das dem Benutzer zu melden und somit die mehrfache Speicherung zu verhindern.

Übersichtlichkeit: Damit der Benutzer sich jederzeit einen Überblick über die vorhandenen Datensätze verschaffen kann, muß das DMT eine Funktion bereitstellen, mit der die vorhandenen Datensätze aufgelistet werden können. Die zu einem Datensatz verfügbaren Informationen müssen angezeigt werden, so daß sich der Benutzer orientieren kann, ohne den Inhalt der Datensätze ansehen zu müssen.

Datensuche: Es muß dem Benutzer möglich sein, gezielt nach bestimmten Datensätzen zu suchen, ohne die ganze Liste von Datensätzen durchzugehen oder die Daten direkt anzusehen. Dies sollte anhand von festdefinierten Suchkriterien geschehen, also

- wann,
- von wem,
- wie,
- woraus der Datensatz erzeugt wurde und
- welcher Kommentar ihm beigestellt ist.

Bei einer Suche müssen die Kriterien nicht vollständig spezifiziert werden.

Revidierbarkeit: Wenn Datensätze erzeugt werden können, muß umgekehrt auch das Löschen von Datensätzen erlaubt sein. Somit kann der Benutzer verschiedene Varianten zum Lernen testen und schließlich die nutzlosen Varianten wieder verwerfen. Das erhöht die Übersichtlichkeit und spart Speicherplatz.

Datenkapselung: Das System bzw. der Benuzter dürfen nicht direkt auf den Daten arbeiten, die Implementierungsdetails der Datenspeicherung sollen verborgen bleiben. Das DMT muß daher Operationen auf dem Datenpool zur Verfügung stellen, mit denen Fakten und Regeln hinzugefügt und gelöscht werden können.

Konsistenzprüfung: Das DMT hat die Operationen auf den Daten hinsichtlich der Konsistenz zu überwachen. Die Datensätze stellen Mengen von Fakten und Regeln dar, und somit sind die Datenoperationen Mengenoperationen. Das ist wichtig, weil einige Daten zeitorientiert sind und sichergestellt werden muß , daß zu jedem Zeitpunkt nur ein Datum existiert. Aus diesem Grund muß das DMT gewährleisten, daß neue Fakten und Regeln nur zu einem Datensatz hinzugefügt werden, wenn sie noch nicht darin vorkommen.

Datenabstraktion: Von den Daten selbst soll weitgehend abstrahiert werden, allerdings nur auf einer syntaktischen Ebene. Fakten und Regeln müssen in ihre Bestandteile zerlegt, umgekehrt auch aus den Einzelteilen zusammengesetzt werden können. Auf die Argumente von Fakten soll nicht mehr über die Argumentpositionen zugegriffen werden können, sondern über die Argumenttypen.

Datenauswahl: Die Daten werden auf einem externen Speichermedium abgelegt. Um immer nur die gerade interessierenden Daten in den Hauptspeicher laden zu können, muß das DMT eine flexible Datenauswahl unterstützen. Wie beim Data Preparation Tool vorgesehen, muß es möglich sein, ganz gezielt bestimmte Daten zu selektieren und diese zusammengefaßt in den Hauptspeicher zu kopieren. Dadurch läßt sich einer Speicherproblematik, wie sie bei MOBAL angesprochen wurde, aus dem Wege gehen.

Flexibilität: Das DMT muß daten- und somit anwendungsunabhängig sein. Fest steht nur, daß Fakten und Regeln verwaltet werden, es dürfen aber keine Annahmen über das konkrete Aussehen der Daten gemacht werden.

Mehrbenutzerbetrieb: Der Wunsch nach einer zentralen Datenhaltung zieht die Forderung nach sich, daß mehrere Benutzer parallel auf den Daten arbeiten können. Alle müssen zu einem Zeitpunkt die gleiche Sicht auf den Datenbestand haben, insbesondere müssen Änderungen, die ein Benutzer vornimmt, auch den anderen Benutzer übermittelt werden. Konflikte, die durch den Mehrbenutzerbetrieb enstehen, müssen aufgelöst bzw. verhindert werden. Z.B. darf es nicht möglich sein, daß ein Benutzer einen Datensatz liest, während ein anderer ihn modifiziert.

Kapitel 3

Integratives Konzept zur Datenorganisation

In der Einführung habe ich bereits erwähnt, daß das von mir zu entwickelnde Werkzeug, das Data Management Tool, drei Aufgabenbereiche abzudecken hat: Datenverwaltung, Datenkapselung und Datenzugriff. Der erste Begriff ist hier etwas doppeldeutig, weil ich im vorherigen Kapitel von Anforderungen an eine zentrale Datenverwaltung sprach und damit das Ganze meinte. In diesem Kontext verstehe ich darunter jedoch die rein administrativen Aufgaben des Tools, wie Anlegen, Löschen, Auflisten von Datensätzen, Suche nach Datensätzen, etc. Demgegenüber bezieht sich Datenkapselung auf das Information Hiding, also dem Verbergen der Implementationsdetails. Das umfaßt die Manipulation von Datensätzen sowie die Zerlegung und Zusammensetzung von Fakten und Regeln. Der letzte Aspekt, der Datenzugriff, spricht die Notwendigkeit an, aus dem Datenpool gezielt Daten herauszusuchen und in den Hauptspeicher zu laden. Alle drei Gesichtspunkte müssen gleichermaßen berücksichtigt werden.

Gegenstand dieses Kapitels ist die Beschreibung meiner Lösungsidee, wie die genannten Aspekte miteinander vereint werden können. Ich stelle ein übergreifendes Konzept vor, das einem Grobentwurf entspricht und das Fundament für die Programmentwicklung bildet. Die Einzelheiten des Designs und die technische Umsetzung werden in den nachfolgenden zwei Kapiteln behandelt.

Allerdings sind noch ein paar einführende Worte notwendig, um den Aufbau dieses Kapitels verstehen zu können. Das Konzept hat sich quasi stufenförmig entwickelt, denn es ergaben sich immer wieder Verbesserungen, die letztendlich aufeinander aufbauten. Dementsprechend eng sind die einzelnen Aspekte dieses Konzepts miteinander verwoben—das erschwert eine klar gegliederte Darstellung. Zudem erscheint es mir bei der Beschreibung der Lösungsidee wichtiger, meine Gedankengänge sichtbar und somit nachvollziehbar zu machen, als ein fertiges Resultat zu präsentieren (das tue ich in Kapitel 4). Deshalb folgt die Gliederung hier nicht nach inhaltlichen Gesichtspunkten, sondern richtet sich nach meiner Herangehensweise an die Problemstellung: Vom Allgemeinen zum Speziellen. Entsprechend behandle ich zuerst die Frage, wie der Datenpool auf einer abstrakten Ebene strukturiert werden kann (Abschnitt 3.1). Da aber eine

abstrakte Struktur nur dann sinnvoll ist, wenn sie sich mit vertretbarem Aufwand realisieren läßt, beschäftige ich mich in Abschnitt 3.2 mit der Umsetzung auf Dateiebene. Damit ist das Aussehen des Datenpools festgelegt, und auf dieser Basis kann dann entschieden werden, wie Daten aus dem Datenpool auszuwählen sind (Abschnitt 3.3). Anschließend verfeinere ich das Konzept, indem ich kläre, was als anwendungsabhängig vom Programm zu trennen ist (Deklarationen) und wie Domänen definiert werden können (Abschnitt 3.4). Am Ende des Kapitels gehe ich auf die Fragen der Datenkapselung ein: welche Operationen auf Datenmengen werden benötigt, wie kann vom Aussehen von Fakten und Regeln abstrahiert werden?

3.1 Datenklassen und Varianten

Im Vordergrund steht zunächst einmal das Problem der Datenstrukturierung. Der Datenpool sollte in sinnvolle und unabhängige Bereiche aufgegliedert werden, so daß der Benutzer den Datenbestand besser überblicken kann und Daten, die nicht unmittelbar zusammengehören, getrennt sind. Meiner Meinung nach bietet sich am ehesten eine Einteilung nach den verschiedenen Kategorien von Daten (Sensordistanzmessungen, Basiswahrnehmungsmerkmale, ...) an. Das entspricht im wesentlichen auch der Idee der Abstraction Levels beim Data Preparation Tool. Doch im Gegensatz dazu kommen hier zu den Fakten die Regeln als weitere Datenstruktur hinzu. Während Beispiele und Hintergrundwissen als Fakten vorliegen, werden die Lernergebnisse durch Regeln repräsentiert. Allerdings sollten Eingabedaten und Ausgabedaten der Lernverfahren nicht miteinander vermischt werden, es handelt sich um zwei unterschiedliche Arten von Daten. Aus diesem Grund erachte ich es als sinnvoll, Fakten und Regeln ebenfalls zu trennen.

Nach diesen Vorüberlegungen kann nun der Begriff der Datenklasse eingeführt werden:

Definition 1 *Eine Datenklasse bezeichnet eine Kategorie von Fakten oder Regeln, die inhaltlich zusammengehören.*

Eine inhaltliche Zusammengehörigkeit läßt sich natürlich nur schwer fassen. Es ist aber auch nicht die Aufgabe des Tools, herauszufinden, welche Daten eine Kategorie bilden. Vielmehr soll dem Benutzer die Möglichkeit zur Strukturierung geboten werden. Die Festlegung der Datenklassen hat er zu treffen, er muß entscheiden, welche Einteilung sinnvoll ist. Mit anderen Worten: Eine Datenklasse dient dazu, bestimmte Daten unter einer Bezeichnung zusammenzufassen. Somit kann der Anwender auf einfache Art und Weise einen Datenbereich fokussieren. Aus der Definition folgt auch, daß der Datenpool in eine Menge von Datenklassen aufgeteilt ist. Die Abstraktionsebenen der Repräsentationshierarchie können auf die Verwaltungsstruktur abgebildet werden, getrennt nach Beispielen und Hintergrundwissen sowie Lernergebnissen.

Zu jeder Datenklasse gibt es eine Menge alternativer Datensätze. Das ergibt sich im Prinzip schon aus der Beschreibung des Lernszenarios. Beispielswei-

se induzieren verschiedene Programmparameter bei der Berechnung der Basiswahrnehmungsmerkmale auch verschiedene Datensätze. In meiner Terminologie stellen die Basiswahrnehmungsmerkmale eine Datenklasse dar, die Datensätze bezeichne ich als Varianten. Eine Variante ist demnach eine Zusammenfassung von Fakten oder Regeln zu einer Einheit, je nach Definition der zugehörigen Datenklasse. Hierbei ist hervorzuheben, daß es sich um Mengen handelt; dies wurde bereits in Kapitel 2 als Design-Ziel formuliert. Allerdings wurde dort auch gefordert, daß doppelte Datensätze erkannt und vermieden werden. Die Frage ist, woran man erkennen kann, ob zwei Varianten identisch sind. Oder anders herum: Was unterscheidet Varianten voneinander?

Intuitiv würde man sagen, zwei Varianten sind gleich, wenn sie dieselben Daten umfassen. Ein inhaltlicher Vergleich von Datensätzen ist jedoch rechenintensiv und bei einem großen Datenbestand nicht durchführbar. Es muß ein Weg gefunden werden, den Inhalt einer Variante knapp zu beschreiben. Dafür eignet sich die Art der Erzeugung. Sie gibt Auskunft darüber, aus welchen Daten, mit welchem Programm und welchen Parametern die Daten einer Variante berechnet wurden. Diese Informationen reichen aus, um eine Variante zu charakterisieren. Eine Variante ist also dann etwas, was ein Programm in einem Durchlauf als Ausgabe liefert. Diese Definition ist zwar noch nicht ganz vollständig und auch ungenau, doch sie verdeutlicht die Idee. Das Ausgangsproblem war ja, daß gleiche, d.h. auf dieselbe Art und Weise berechnete Beispiele mehrfach abgespeichert wurden.

Programme können über den Namen spezifiziert werden, doch was ist mit den Eingabedaten für das Programm, wie können sie eindeutig beschrieben werden? Dazu verweise ich noch einmal auf die Abbildung 2.3 auf Seite 26. Jeder Knoten des Graphen stellt eine Datenklasse dar, die mehrere Varianten umfaßt. Die Varianten sind auf unterschiedliche Art und Weise erzeugt worden und repräsentieren daher auch alternative, eigenständige Datensätze. Um nun auf einer Abstraktionsebene Daten zu generieren, werden Daten aus anderen Datenklassen benötigt. Diese Zusammenhänge drücken die Kanten des Graphen aus. Folglich lassen sich die Eingabedaten für ein Programm zur Beispielgenerierung als eine Liste von Varianten spezifizieren. Zu jeder Variante ist die Datenklasse anzugeben und evtl. eine Einschränkung im Sinne einer syntaktischen Auswahl (mehr dazu in Abschnitt 3.3). Beispielsweise könnte eine Eingabe zur Berechnung von Basiswahrnehmungsmerkmalen so aussehen:

1. Alle Fakten aus Variante 1 der Sensordistanzmessungen zum Trace `t715`.
2. Alle Fakten aus Variante 4 der Roboterpositionen zum Trace `t715`.

Analog können Beispiele und Hintergrundwissen als Eingabe zum Lernen spezifiziert werden. Zusammen mit dem Namen des Lernverfahrens und einer geeigneten Kodierung der Lernparameter läßt sich so der Inhalt von Regel-Varianten beschreiben.

Doch nicht alle Varianten werden automatisch generiert. Oft möchte man als Benutzer auch eine Variante selbst erstellen, indem man die Daten von Hand

eingibt. Oder es werden Varianten einfach fest vorgegeben, weil sie die Ausgangsdaten für alle weiteren Berechnungen darstellen (Sensordistanzmessungen, Roboterpositionen und Szenedaten). Für diese Fälle führe ich eine zweite Art von Varianten ein, nämlich Varianten, über deren Inhalt keine Aussage gemacht werden kann. Sie sind einfach vom Anwender erstellt und daher per definitionem einmalig. Bei ihnen entfällt die Überprüfung, ob ein identischer Datensatz schon existiert.

Schließlich besteht in der syntaktischen Auswahl noch eine dritte Möglichkeit, Varianten zu erzeugen. Wie ich später noch erläutern werde, kann aus bestehenden Varianten eine Datenauswahl getroffen werden. Solche Selektionen lassen sich dann wiederum als eigenständige Varianten abspeichern.

Jetzt kann die Definition einer Variante präzisiert werden:

Definition 2 *Eine Variante ist eine Fakten- bzw. Regelmenge innerhalb einer Datenklasse, die auf dreierlei Art und Weise erzeugt werden kann:*

1. *Durch ein Programm,*
2. *durch den Benutzer oder*
3. *durch eine syntaktische Auswahl auf einer bestehenden Variante.*

Datenklassen und Varianten verleihen dem Datenpool eine abstrakte Struktur, die unabhängig ist von der Art der Datenspeicherung. Für den Benutzer stellt sich der Datenbestand folglich als eine Datensatzmenge dar, die disjunkt in mehrere Kategorien aufgeteilt werden kann; jeder Datensatz gehört genau einer Kategorie an. Das sagt aber noch nichts darüber aus, wie der Datenpool organisiert ist[1]. Implizit habe ich bis hierhin stillschweigend vorausgesetzt, daß er sich aus einzelnen Datensätzen zusammensetzt, doch man könnte ihn auch als eine große Datenmenge betrachten. Da zu jeder Variante eine Inhaltsbeschreibung existiert, könnten die Daten zu den Datensätzen im Bedarfsfall ausgewählt bzw. erzeugt werden. Datensätze würden demnach nicht als eigenständige Einheiten auf dem externen Speichermedium abgelegt, sondern sie wären teilweise nur temporär im Hauptspeicher. Das hätte natürlich den Vorteil minimaler Speicherplatzanforderungen. Aus zwei Gründen ist diese Lösung aber unbefriedigend und nicht realisierbar:

1. Bei der Masse an Daten, wie sie in der BLearn-Anwendung vorliegt, ist die ständige Neuberechnung von Daten viel zu zeitintensiv und dem Anwender nicht zuzumuten. Die Daten sollen einmal generiert werden und von da an zur Verfügung stehen.
2. Datensätze, die der Benutzer von Hand erstellt (das kommt bei BLearn sehr häufig vor), können nicht über eine Inhaltsbeschreibung spezifiziert werden. Der Inhalt kann allein durch die Aufzählung jedes einzelnen Datums repräsentiert werden.

[1] Ich verstehe unter dem Begriff Datenpool immer die Gesamtheit der extern abgespeicherten Daten.

Das spricht dafür, Datensätze als externe Dateneinheiten abzuspeichern. Die Speicherplatzausnutzung ist in diesem Fall allerdings nicht optimal, denn es kann ja nur verhindert werden, daß automatisch generierte Datensätze mehrfach vorkommen; unterschiedliche Varianten können aber trotzdem eine nichtleere Schnittmenge haben! In diesem Kontext möchte ich betonen, daß ich unter Varianten nur die extern abgelegten Datensätze verstehe. Weiter unten werde ich eine zweite Art von Datensätzen einführen: die Samples. Diesen Begriff übernehme ich von Anke Rieger [Rieger 1996] und meine damit Datensätze im Hauptspeicher. Samples werden im Gegensatz zu Varianten nur durch syntaktische Auswahlen erzeugt und sind ausschließlich temporär (doch dazu mehr in Abschnitt 3.3).

3.2 Verwaltungsstruktur auf Dateiebene

Der Datenpool ist nun auf einer abstrakten Ebene in Datenklassen und Varianten untergliedert. Das nächste Problem ist die Abbildung dieser Struktur auf eine Dateistruktur. Zwar sollte die konkrete Speicherung der Daten verborgen bleiben, doch eine vollständige Entkopplung läßt sich nicht erreichen. Z.B. muß spezifiziert werden, wo die Daten abgelegt werden sollen. Notwendig ist also eine Deklaration eines Verzeichnisses (bei Datenbanken könnten das Tabellennamen sein). Andererseits hat die Größe von Dateien entscheidenden Einfluß auf die Zugriffszeiten; bei den vorliegenden Datenmassen ein nicht zu vernachlässigender Aspekt. Deshalb wird zusätzliches Wissen über die Daten benötigt, um eine effiziente Speicherung zu gewährleisten.

In einer ersten Lösung sah ich vor, jeder Variante genau eine Datei zuzuordnen, die die entsprechenden Fakten oder Regeln enthält. Das zieht allerdings lange Zugriffszeiten nach sich, denn bei der syntaktischen Auswahl mit dem Data Preparation Tool müssen alle Fakten eingelesen und betrachtet werden. In der Praxis trat man diesem Problem entgegen, indem die Dateien möglichst klein gehalten wurden. Beispielsweise waren die Sensordistanzmessungen jeweils nach dem Trace in Dateien aufgesplittet. Mein Konzept sieht allerdings vor, daß sich eine Datenauswahl immer auf eine Variante bezieht, nie auf mehrere. Folglich müssen die Sensordistanzmessungen in eine Variante gepackt werden, wenn man auf verschiedene Traces gleichzeitig zugreifen möchte. Gäbe es nur eine Datei pro Variante, müßte bei jeder Selektion eine immense Datenmasse durchsucht werden, was zeitlich nicht zu vertreten ist. Stattdessen entschied ich mich dafür, daß jede Variante in Partitionen aufgeteilt wird. Für die Sensordistanzmessungen würde es sich anbieten, die Daten zu einem Trace jeweils in einer Partition, also Datei, zu halten. Bei einer Auswahl müßten nur noch die in Frage kommenden Partitionen untersucht werden—ein erheblicher Zeitgewinn.

Die Frage ist nur, wo definiert wird, wie Varianten aufzusplitten sind. Diese Entscheidung kann eigentlich nur der Benutzer in Abhängigkeit von der jeweiligen Anwendung treffen. Die Art der Partitionierung sollte deshalb extern deklariert werden, für jede Datenklasse einzeln. Und welche Kriterien gibt es, um die Variantendateien bilden zu können? Dafür bieten sich meiner Meinung

nach die charakteristischen Merkmale der Daten (Fakten und Regeln) an, also hauptsächlich der Prädikatsname und die Argumente. Ich definiere daher einen Aufteilungsschlüssel als eine Folge von Argumenttypen einschließlich des Prädikatsnamens. Diese Folge kann natürlich auch leer sein, dann ist den Varianten der Datenklasse genau eine Datei zugeordnet. Oder sie besteht aus mehreren Kriterien, z.B. könnte es ja auch sinnvoll sein, nach Trace und Sensor zu partitionieren. Das hieße dann, daß alle Fakten, die sich auf den gleichen Trace und den gleichen Sensor beziehen, in einer Datei stehen. Dabei setze ich stillschweigend das Wissen über die Argumentpositionen voraus. Irgendwo muß folglich deklariert sein, wo bei welchem Prädikat die Tracekennung und die Sensorkennung aufgeführt sind. Darauf gehe ich im nächsten Abschnitt ein. Zunächst einmal fasse ich das eben Gesagte formal zusammen:

Definition 3 *Ein Partitionierungsschlüssel ist eine Folge von Argumenttypen, die auch den Prädikatsnamen als Unterscheidungskriterium enthalten kann; die Nullfolge ist zulässig. Jeder Datenklasse ist genau ein Schlüssel zugeordnet, der bestimmt, wie die zugehörigen Varianten in Dateien (Partitionen) aufgesplittet werden.*

Allerdings nehme ich hier eine Einschränkung vor: Regelmengen lassen sich nur nach dem Prädikatsnamen aufteilen. Es sind zwar andere Unterscheidungskriterien denkbar, doch für unsere Anwendung reicht das aus. Das hängt auch mit der Deklaration der Regel-Datenklassen zusammen, worauf ich weiter unten noch näher eingehen werden. Des weiteren muß bei einem Schlüssel gewährleistet sein, daß er auf alle Elemente anwendbar ist. D.h. er darf sich nur auf Argumenttypen beziehen, die allen Fakten einer Datenklasse gemeinsam sind. Ansonsten können die Elemente nicht eindeutig einer Datei zugewiesen werden.

Die Partitionen stellen die kleinsten Dateneinheiten im Datenpool dar. Wie sie mit Datenklassen und Varianten im Zusammenhang stehen, veranschaulicht Abbildung 3.1.

3.3 Selektion von Fakten und Regeln

Bislang war nur die Rede davon, wie der Datenpool aussieht, der sich auf einem externen Speichermedium befindet. Will man aber mit den Daten arbeiten, so muß man auf sie zugreifen können, d.h. Ausschnitte des Datenpools in den Hauptspeicher laden können. Es geht folglich um eine gezielte Datenauswahl, so wie sie beispielsweise das Data Preparation Tool mit der `select`-Anweisung ermöglicht.

Die erste Frage, die sich stellt, ist, nach welchen Kriterien die Auswahl erfolgen soll. Verschiedenste Kriterien sind möglich, doch was hier benötigt wird, ist die Selektion nach syntaktischen Merkmalen. Man möchte z.B. alle Regeln mit einem bestimmten Prädikatsnamen und alle Fakten mit gewissen Argumenten finden. Worauf man also Bezug nimmt, sind die einzelnen Bestandteile von Fakten und Regeln. Indem für Bestandteile zulässige Werte angegeben werden, ist

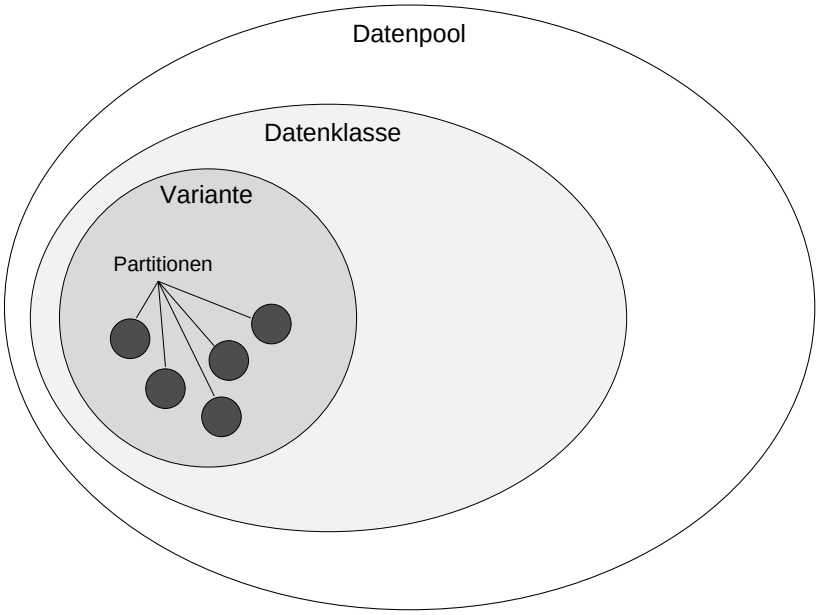


Abbildung 3.1: Gesamtstruktur des Datenpools

es möglich, die gewünschten Daten herauszusuchen. Doch welche Bestandteile gibt es eigentlich?

Fakten setzen sich zusammen aus dem Prädikatsnamen und den Argumenten. Letztere möchte man nicht über die Positionen, sondern über den Typ referenzieren können. Das setzt aber voraus, daß die Argumenttypen der Prädikate irgendwo deklariert sind. D.h. das Wissen, welcher Argumenttyp an welcher Position bei welchem Prädikat steht, muß der Datenverwaltung formal vorliegen. Z.B. könnte die Deklaration des Basiswahrnehmungsmerkmals `stable` so aussehen:

```
stable(<trace>, <orientierung>, <sensor>, <startzeitpunkt>,
       <endzeitpunkt>, <gradient>).
```

Um alle `stable`-Fakten zum Trace `t701` und den Sensoren `s0`, `s1` zu erfassen, würde man schreiben

```
<predname> = stable
<trace>    = t701
<sensor>   = s0,s1
```

Da die Einführung von Datenklassen den Sinn hat, unterschiedliche Daten zu trennen, müssen sich Prädikatsdeklarationen konsequenterweiser immer auf eine Klasse beziehen.

Regeln hingegen haben eine komplexere Struktur, wenn man alle Details berücksichtigt. Ähnlich den Fakten wäre es möglich, mit Deklarationen zu arbeiten,

doch für unsere Anwendung wird hauptsächlich die Auswahl über den Prädikatsnamen benötigt. Deshalb kommen bei den Regeln nur die allgemeinen Bestandteile Prädikatsname, Regelkopf und Regelkörper in Betracht. Das ist für diese Zwecke ausreichend.

Nachdem festgelegt ist, nach welchen Kriterien die Datenauswahl erfolgt, bleibt zu klären, woraus ausgewählt wird, was also die grundlegenden Zugriffseinheiten des Datenpools sind. Dafür kommen eigentlich nur die Varianten in Frage. Datenklassen scheiden aus, weil es keinen Sinn macht, Daten unterschiedlicher Datensätze zu vermischen. Das würde dem Variantenkonzept widersprechen. Andererseits sind Partitionen zu speziell und zu nah mit der Implementierung verknüpft, als daß sie als Zugriffseinheiten in Betracht kommen würden. Partitionen sollten dem Benutzer verborgen bleiben, die Entscheidung, welche Partitionen bei einer Auswahl von Interesse sind, muß automatisch getroffen werden. Ich habe die Partitionierungsschlüssel ja eingeführt, damit der Zugriff auf Dateiebene beschleunigt werden kann, nicht weil Varianten konzeptuell noch weiter untergliedert werden sollen.

Bestimmte Daten aus dem Datenpool können also über syntaktische Merkmale selektiert werden. Solche Selektionen werden in den Hauptspeicher geladen und dort als eine Einheit abgelegt. In Anlehnung an die Terminologie von Anke Rieger [Rieger 1996] bezeichne ich eine solche Zusammenstellung als ein Sample[2]:

Definition 4 *Ein Sample ist eine Fakten- bzw. Regelmenge im Hauptspeicher, die durch eine syntaktische Auswahl auf einer Variante oder einem Sample entstanden ist.*

Samples unterscheiden sich von Varianten darin, daß sie sich im Hauptspeicher befinden und nur auf eine Art und Weise erzeugt werden können. Abbildung 3.2 verdeutlicht diesen Sachverhalt.

Um die Selektion von Faktenmengen zu realisieren, kann das Data Preparation Tool eingesetzt werden. Allerdings ist dabei über die Argumenttypen von den Argumentpositionen zu abstrahieren. Im Hinblick darauf wäre es auch wünschenswert, die Case Selection bei dem Datenverwaltungssystem in einer erweiterten Form zur Verfügung zu stellen. Und zwar so, daß die Relationenlisten sich ebenfalls nur auf die Typen und nicht auf die Positionen beziehen. Die Case Selection auf Regeln anzuwenden, macht in diesem Kontext jedoch keinen Sinn.

3.4 Lernumgebung und Domänen

Das Werkzeug zur Datenverwaltung soll anwendungsunabhängig sein. Dieser Anforderung muß bei dem Design und der Programmentwicklung entsprochen werden. Ganz konkret für mein Konzept heißt das, die Festlegung der Datenklassen und der Partitionierungsschlüssel sowie die Deklaration der Prädikate und des Verzeichnisses, wo die Daten abgelegt werden sollen, sind vom Programm zu

[2]Analog dazu nenne ich im weiteren die Auswahlkriterien auch Constraints.

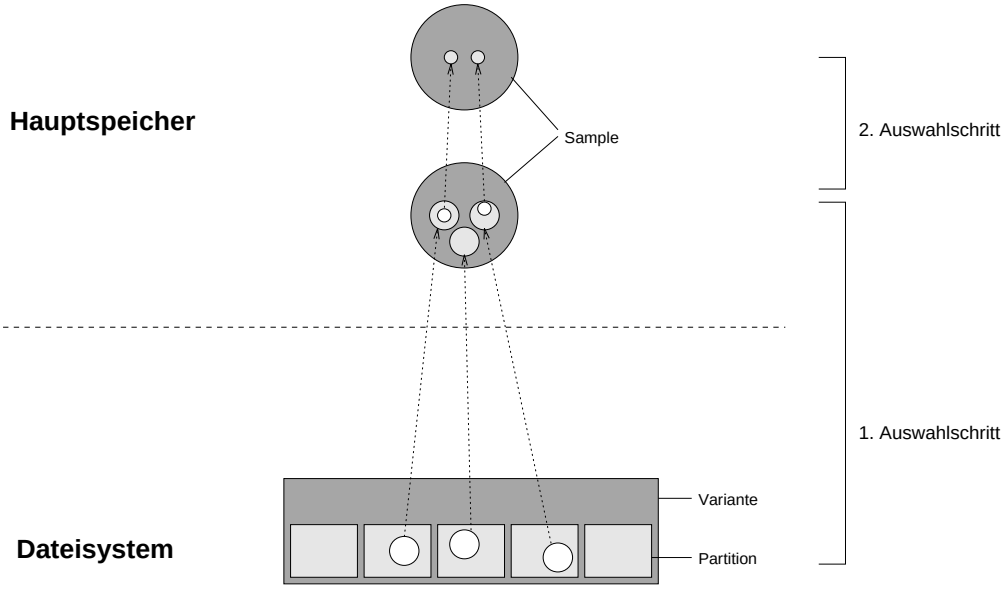


Abbildung 3.2: Samples und Varianten

entkoppeln. Diese Informationen können beispielsweise in einer Deklarationsdatei aufgeführt sein, der das Tool dann das Wissen über die Daten entnehmen kann. Auf einer abstrakteren Ebene möchte ich das als eine Lernumgebung bezeichnen, unabhängig davon, wie die Informationen repräsentiert werden.

Definition 5 *Eine Lernumgebung bildet den Rahmen für die Organisation des Datenpools und legt fest*

- *wo die Daten abgelegt sind,*
- *welche Datenklassen es gibt,*
- *wie die Varianten in Dateien aufgesplittet werden (Partitionierungsschlüssel) und*
- *welches Format die Prädikate in Fakten-Datenklassen haben.*

Für unsere Anwendung würde eine Lernumgebung im Prinzip die Umsetzung der Repräsentationshierarchie auf die Struktur des Datenpools bedeuten. Sie umfaßt also die anwendungsspezifischen Informationen über die Daten.

Lernumgebungen beziehen sich immer auf einen Datenpool, Selektionen hingegen stellen einen sehr speziellen Ausschnitt daraus dar. Der Benutzer hat folglich zum einen die Sicht auf die gesamte Repräsentationshierarchie mit all ihren Varianten und zum anderen auf kleine Datenmengen auf einer Abstraktionsebene. Wünschenswert wäre aber noch ein Zwischenkonzept, das wie in MOBAL die Definition von Domänen erlaubt. Das entspricht einer Zusammenfassung

einzelner Datenauswahlen auf allen Stufen, quasi einer Instanz der Repräsentationshierarchie. Worauf dabei abgezielt wird, habe ich schon in der Einführung angesprochen: Man möchte sich auf jeder Ebene geeignete Daten heraussuchen und diese dann als eine Einheit bezeichnen. Damit wird das möglich, was ich im Zusammenhang mit MOBAL als Mischen von Domänen beschrieben habe. Bis jetzt ist in meinem Entwurf nur vorgesehen, entweder alle Domänen (Lernumgebung) oder Teile von Domänen (Selektionen) zu betrachten. Es fehlt der Begriff der Domäne.

Definition 6 *Eine Domäne ist eine Zusammenfassung von Daten, die pro Datenklasse höchstens eine Auswahl auf einer Variante enthält.*

Um Mißverständnissen vorzubeugen: Domänen sind ein abstraktes Konstrukt. *Zusammenfassung* bedeutet nicht, daß eine Domäne physikalisch als eine Einheit abgespeichert wird. Dann hätten wir wieder die alten Speicherplatzprobleme. Nein, eine Domäne dient vielmehr dazu, eine Menge von Samples zu beschreiben (zu jeder Datenklasse maximal ein Sample). Da Samples ja an und für sich nicht extern gesichert werden, müssen sie bei jedem Systemstart neu generiert werden. Hat man allerdings einen festen Satz an Samples zusammen, ist das mühsam. Domänen haben daher den Sinn, vormals erstellte Samples automatisch erzeugen zu lassen. Abbildung 3.3 verdeutlicht diesen Sachverhalt.

3.5 Operationen auf Datenmengen

Ich unterscheide zwei Arten von Datenmengen: Varianten und Samples. Varianten sind extern abgelegt, Samples befinden sich im Hauptspeicher. Da die Daten gekapselt werden sollen, muß das Data Management Tool Operationen auf diesen Datenmengen zur Verfügung stellen. Benötigt wird quasi der abstrakte Datentyp *Menge*. Die Frage ist nur, welche Operationen erforderlich sind.

Fangen wir mit den Varianten an. Elementar sind das Anlegen und das Entfernen von Datensätzen. Anlegen heißt, eine leere Datenmenge wird auf dem externen Speichermedium erzeugt; Entfernen bedeutet, der Speicherplatz, den eine bestehende Datenmenge auf der Platte belegt, wird freigegeben. Bei der Erzeugung ergibt sich allerdings das Problem, doppelte Datensätze zu erkennen. Da die Datenmengen nicht Element für Element verglichen werden können, müssen die Informationen über den Inhalt einer Datenmenge bereits bei ihrer Erzeugung vorliegen. Das wiederum setzt voraus, daß die Elemente, die später zur Menge hinzugefügt werden, auch wirklich den angegebenen Informationen über die Datengenerierung entsprechen. Als weitere Operationen werden das Hinzufügen und das Entfernen von Mengenelementen (Fakten oder Regeln) benötigt. Für die Programme zur Beispielgenerierung würde es vielleicht genügen, alle Daten direkt beim Anlegen einer Variante zu übergeben. Dann wären die Routinen zur Manipulation von Datensätzen überflüssig. Dagegen spricht jedoch, daß auch der Benutzer Varianten erstellt. In der Regel sind diese Datenmengen nicht starr, sondern werden mehrfach geändert. Aus diesem Grund entschied ich

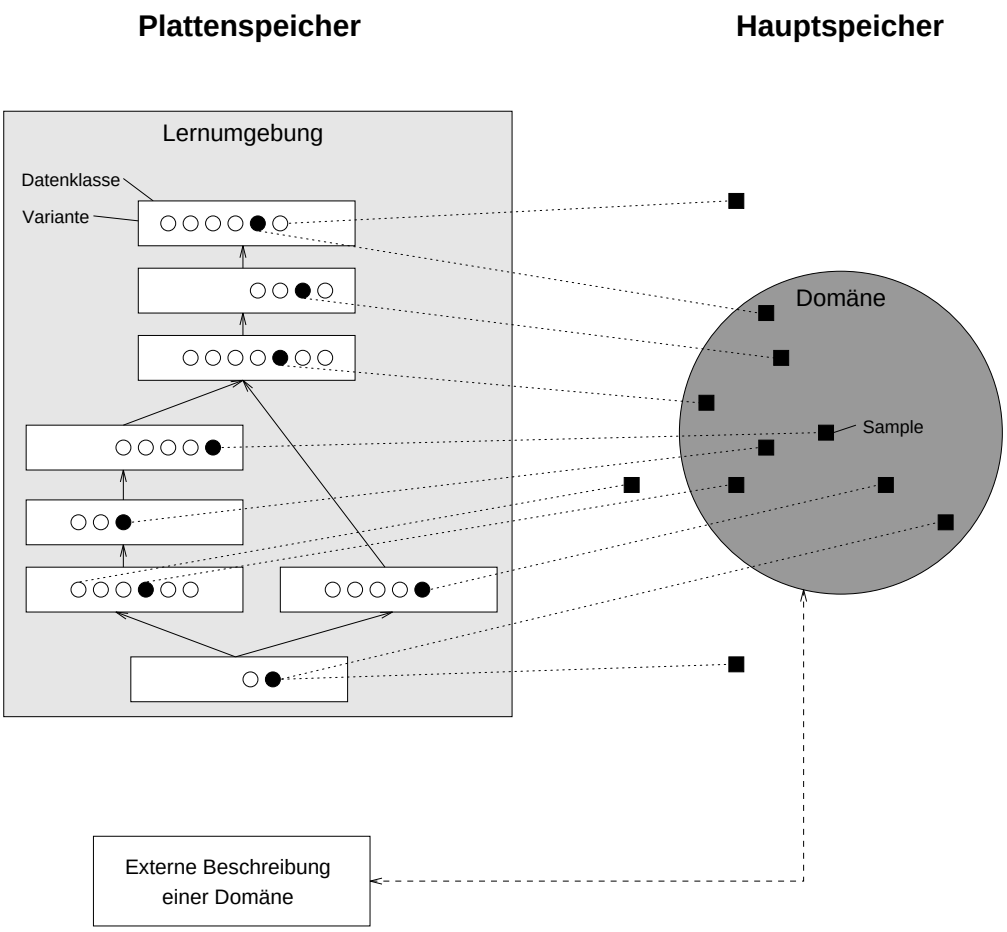


Abbildung 3.3: Domänen als Mengen von Samples

mich für die beiden einfachen Mengenoperationen Hinzufügen und Entfernen, die sehr flexibel eingesetzt werden können.

Bei Samples sieht die Situation ein wenig anders aus. Samples werden automatisch durch syntaktische Auswahl auf Varianten erzeugt. Sinnvolle Operationen sind das Löschen, die Vereinigung und das Sortieren von Samples. Die Entfernung ist wichtig, um ein Überladen des Hauptspeichers mit (ungenutzten) Samples zu vermeiden—gerade wenn viele Selektionen vorgenommen werden. Mit der Vereinigung zweier Samples können Daten zusammengefaßt werden, um sie beispielsweise später in ihrer Gesamtheit zu sortieren. Sortierung setzt aber voraus, daß eine Ordnung auf den Elementen eines Samples besteht. Samples stellen folglich geordnete Datenmengen dar, so daß besondere Reihenfolgeaspekte der Daten berücksichtigt werden können. Z.B. ist es häufig nötig, Fakten nach den Zeitpunkten zu sortieren. Für den Benutzer kann es aber zum Ansehen von Selektionen auch nützlich sein, nach weiteren Merkmalen zu ordnen—möglicherweise erst nach Zeitpunkten und dann nach Sensoren. Gewünscht ist eine

Sortierung nach einer Folge von Kriterien, wahlweise auf- oder absteigend. Dabei kommen als Vergleichskriterien wieder die Bestandteile von Fakten und Regeln in Betracht, die auch zur Datenauswahl benutzt werden.

3.6 Kapselung von Mengenelementen

Im vorherigen Kapitel wurde die Forderung formuliert, Fakten und Regeln gemäß ihrer Komponenten zerlegen und zusammensetzen zu können. Sinn und Zweck ist es, vom konkreten Aussehen der Daten zu abstrahieren. So gesehen wird hier eine zweistufige Datenkapselung verfolgt: auf der ersten Stufe die Datenmengen, auf der zweiten Stufe die Mengenelemente.

Die Struktur der Fakten ist offensichtlich. Sie besitzen einen Prädikatsnamen und eine Folge von Argumenten, deren Typen in der Lernumgebung deklariert sind. Demgegenüber kann eine Regel sehr weit untergliedert werden. Meiner Meinung nach ist hier aber eine Zerlegung in Kopf und Körper ausreichend, weil die verschiedenen Regelformate hierin im wesentlichen übereinstimmen.

Die Zugriffsroutinen auf die Mengenelemente sollten nun so gestaltet sein, daß sie von den konkreten Daten unabhängig sind. Beispielsweise hatte ich in Abschnitt 2.1.3 auf Seite 28 eine Routine `get_trace` vorgeschlagen, die aus einem beliebigen Fakt den Trace extrahiert. Dadurch werden die Daten zwar gekapselt, doch `get_trace` ist spezifisch für die BLearn-Anwendung. Soll das Werkzeug zur Datenverwaltung in einem anderen Bereich eingesetzt werden, müssen neue Zugriffe definiert werden, jeweils in Abhängigkeit der Argumenttypen. Um eine größtmögliche Entkopplung von Programm und Daten zu erreichen, sind die Zugriffsroutinen so zu gestalten, daß die Bestandteile, insbesondere die Argumenttypen, explizit als Parameter übergeben werden. Um bei dem Beispiel zu bleiben: `get_trace` müßte verallgemeinert werden zu `get`, so daß der Aufruf dann wäre

```
get(trace, stable(t701, 75, s0, 5, 23, 0), Trace).
```

An erster Stelle steht dabei der Typ des Arguments, auf das zugegriffen werden soll.

Kapitel 4

Das Data Management Tool

Auf der Grundlage des integrativen Konzepts stelle ich nun das Werkzeug zur Datenverwaltung vor. Dabei liegt der Schwerpunkt in der Beschreibung der äußeren Schnittstelle, sowohl zum Benutzer als auch zu externen Programmen hin, die Daten generieren oder auf sie zugreifen. Es geht um die Fragen, wie sich die Arbeit mit dem System gestaltet, welche Funktionalität es bietet und wie ganz konkret der Grobentwurf umgesetzt wurde. Im Gegensatz zum vorherigen Kapitel verfolge ich hier allerdings eine Gliederung, die sich an den Aufgabenbereichen des Tools orientiert: Variantenverwaltung, Datenmanipulation und Datenselektion. Diese Aspekte sind so eng miteinander verwoben, daß ich zunächst einmal in Abschnitt 4.1 einen kurzen Überblick über die zur Verfügung gestellten Routinen gebe. Das geschieht anhand einer einfachen Beispielanwendung. Abschnitt 4.2 beschäftigt sich mit der Realisierung des Begriffs der Lernumgebung. Folgende Punkte sind hier von Interesse: wie sieht eine Lernumgebung aus, wie wird sie erstellt und wie arbeitet man damit? Der daran anschließende Abschnitt hat die Verwaltung von Datensätzen zum Thema. Es wird erläutert, wie Varianten angelegt und gelöscht werden, wie man Varianten suchen kann und wie sich ermitteln läßt, ob ein Datensatz schon existiert. Datenmanipulation, Gegenstand von Abschnitt 4.4, umfaßt die Kapselung von Mengenelementen sowie die Operationen auf Datenmengen. Im einzelnen geht es darum, wie Fakten und Regeln zerlegt, zusammengesetzt, zu Varianten oder Samples hinzugefügt und aus Varianten bzw. Samples entfernt werden können. Das Kapitel endet mit der Darstellung der Möglichkeiten, Daten auszuwählen und mit Selektionen zu arbeiten.

4.1 Ein kurzer Überblick anhand eines Beispiels

Ich möchte den Anfang mit einem kleinen Durchgang machen, der über die grundlegenden Funktionen des Data Management Tools informiert. Und zwar soll mein altes Kinderspielzeug mit dem Computer archiviert werden. Früher hatte ich Autos, Puppen, Spiele, etc., besonders gut kann ich mich noch an meine Eisenbahn erinnern. Die Arten von Spielzeug stellen die Datenklassen meines Anwendungsgebiets dar. Zu überlegen ist jetzt, wie die einzelnen Spiel-

```
directory('/home/kimo-d/zitzler/Spielzeug').

dataclass(autos, facts, '/Autos', []).
dataclass(puppen, facts, '/Puppen', [geschlecht]).

predicate(autos, mein_auto/2, [hersteller, farbe]).

predicate(puppen, puppe_hell/2, [name, geschlecht]).
predicate(puppen, puppe_dunkel/2, [name, geschlecht]).
```

Abbildung 4.1: Deklarationen für die Spielzeug-Anwendung

zeuge als Fakten repräsentiert werden. Da ich damals eine einfache, wenn auch etwas eigene Art hatte, die Dinge zu beschreiben, möchte ich das heute beibehalten. Autos sind gekennzeichnet durch die Herstellerfirma und die Farbe; die Prädikatsdeklaration lautet:

```
mein_auto(<hersteller>, <farbe>).
```

Puppen hatten natürlich immer einen Namen, allerdings trennte ich strikt nach den Geschlechtern. Zudem besaß ich Puppen unterschiedlicher Hautfarbe. Ich führe daher zwei zweistellige Prädikate ein:

```
puppe_hell(<name>, <geschlecht>),
puppe_dunkel(<name>, <geschlecht>).
```

Um das Beispiel nicht unnötig aufzublähen, will ich es bei diesen beiden Arten von Spielzeugen bewenden lassen.

Das Programm verlangt, daß ich zunächst eine Lernumgebung in Form einer Deklarationsdatei definiere. Die Deklarationen legen die Datenklassen fest, beschreiben die Prädikate und geben an, wo die Daten abzulegen sind. Der Inhalt der Deklarationsdatei für meine Anwendung wird in Abbildung 4.1 wiedergegeben. Der erste Fakt bezeichnet das Verzeichnis, unter dem die Daten abgespeichert werden. Die folgenden zwei Fakten definieren die Datenklassen `autos` und `puppen` als Kategorien von Fakten (das dritte Argument eines `dataclass/4`-Fakts steht für ein Unterverzeichnis und das vierte für den Partitionierungsschlüssel). Die drei `predicate/3`-Fakten deklarieren die Prädikate für die beiden Datenklassen.

Die Deklarationsdatei wird mit dem Kommando `load_declarations` eingeladen, anschließend muß das Tool mittels `varianten_init` an die neue Lernumgebung angepaßt werden:

```
| ?- load_declarations('SpielzeugDeklarationen').
```

```
yes
| ?- varianten_init.

yes
| ?-
```

Der letzte Aufruf bewirkt auch, daß ggf. die Verwaltungsstrukturen auf Dateiebene angelegt und Informationen über bereits bestehende Varianten gelesen werden.

Jetzt habe ich einen leeren Datenpool erzeugt, und es wird Zeit, ihn zu füllen. Dazu lege ich eine erste Variante an, die alle meine schönsten Puppen erfassen soll. Der Befehl dazu lautet:

```
| ?- new_variant(puppen, die_schoensten,
                 [user(zitzler), date([96,2,28,12,45,0]),
                 comment('Meine schoensten Puppen'),
                 attribute(none), source([]), generator(user)]).

yes
| ?-
```

Die Prozedur generiert eine leere Variante in der Datenklasse `puppen`, ihr Name ist `die_schoensten`. Die zusätzlichen Angaben in der Liste, die als drittes Argument aufgeführt ist, spezifizieren die Eigenschaften der Variante. Diese Eigenschaften geben u.a. Auskunft darüber, daß die Variante von mir am 28. Februar 1996 um viertel vor eins erzeugt wurde, daß sie „meine schönsten Puppen“ erfaßt und daß ich ihren Inhalt per Hand erstelle (`generator(user)`).

Ich möchte nun meine drei (ehemaligen) Lieblingspuppen in die Variante aufnehmen. Dazu setze ich zuerst einmal die entsprechenden Fakten zusammen, um sie nachfolgend zu dem Datensatz hinzuzufügen. Die Funktion `compose` generiert ein Fakt zu einer Datenklasse gemäß der Prädikatsdeklarationen. Sie braucht dafür die Angabe des Prädikatsnamens und der Argumente. Mit `add` kann ein Fakt zu einer Variante hinzugefügt werden. Als erstes soll meine helle Puppe Babsi drankommen[1]:

```
| ?- compose(puppen,
            [predname,   name,  geschlecht],
            [puppe_hell, babsi, maedchen],
            Fact),
     add(variant(puppen, die_schoensten), [Fact]).

Fact = puppe_hell(babsi,maedchen)

| ?-
```

[1] Natürlich ist das nicht der gängige Weg, um Daten zu einer Variante hinzuzufügen; das wäre für den Benutzer viel zu umständlich. In der Regel wird er die Prozedur `add_file_to_variant` benutzen, die Daten aus einer Datei in eine Variante schreibt (siehe Abschnitt 4.4.2). Im vorliegenden Fall habe ich es so gemacht, weil es sich nur um drei Fakten handelt.

Das Gleiche mache ich mit meiner dunklen Puppe John und meiner hellen Puppe Jan (allerdings ohne `compose` zu benutzen):

```
| ?- add(variant(puppen,die_schoensten),
         [puppe_dunkel(john,junge),puppe_hell(jan,junge)]).

yes
| ?-
```

Die Variante beinhaltet nun drei Fakten, die nach dem Merkmal Geschlecht getrennt in zwei Partitionen abgelegt sind. Welche Partitionen es gibt, kann ich erfahren, wenn ich mir mittels `variant_info` die Informationen zu meinem Datensatz anzeigen lasse:

```
| ?- variant_info(puppen, die_schoensten).

VARIANTE: die_schoensten

Datenklasse:     puppen
Erstellt von:    zitzler
Erstellt am:     28.2.96 um 12:45 Uhr
Attribut:        none
Kommentar:       Meine schoensten Puppen
Generiert durch: Benutzer
Partitionen:     [junge], [maedchen]


yes
| ?-
```

Wie wäre es, sich den Inhalt der Variante anzugucken? Mit `synt_select` kann ich eine syntaktische Auswahl treffen und die gefundenen Fakten als ein Sample im Hauptspeicher ablegen. Anschließend ist es möglich, auf das Sample zuzugreifen und die Fakten zu inspizieren. Zunächst sollen alle Fakten ausgewählt werden:

```
| ?- synt_select(variant(puppen, die_schoensten), all, Sample).

Sample = sid1

| ?-
```

Zurückgegeben wird die Kennung des Samples, die wir brauchen, um über `access_sample` an die Fakten heranzukommen:

```
| ?- access_sample(sid1, FactList, _).


FactList = [puppe_hell(babsi,maedchen),puppe_dunkel(john,junge),
puppe_hell(jan,junge)]

| ?-
```

Wenn ich früher wütend war, habe ich manchmal meine Puppen in die Ecke geschmissen. Aber nur die männlichen Puppen, weil die nicht so schön waren. Also, bitte alle männlichen Puppen aussortieren:

```
| ?- synt_select(sample(sid1), [item(geschlecht, [junge])], Sample).

Sample = sid2

| ?- access_sample(sid2, FactList, _).

FactList = [puppe_dunkel(john,junge),puppe_hell(jan,junge)]

| ?-
```

Das zweite Sample möchte ich nun in aufsteigender Reihenfolge nach den Namen sortiert haben. Dazu bediene ich mich der Routine `sort_sample`. Anschließend soll das erste Element der Faktenliste in seine Bestandteile zerlegt werden.

```
| ?- sort_sample(sid2, [name], <),
     access_sample(sid2, FactList, _),
     FactList = [FirstElement|_],
     compose(puppen, Items, Terms, FirstElement).

FactList = [puppe_hell(jan,junge),puppe_dunkel(john,junge)],
FirstElement = puppe_hell(jan,junge),
Items = [predname,name,geschlecht],
Terms = [puppe_hell,jan,junge]

| ?-
```

Selbstverständlich hatte ich auch sehr häßliche Puppen—sie müssen in eine getrennte Variante:

```
| ?- new_variant(puppen, die_bloedsten,
                 [user(zitzler), date([96,2,28,16,52,0]),
                 comment('Meine allerbloedsten Puppen'),
                 attribute(none), source([]), generator(user)]).

yes
| ?-
```

Jetzt gibt es bereits zwei Datensätze. Die Prozedur `variant_properties` ermöglicht nun, gezielt nach Varianten zu suchen. Ich möchte beispielsweise alle Varianten ermitteln, die ich 1996 erstellt habe:

```
| ?- variant_properties(puppen, Variant,
                        [user(zitzler), date([96,_,_,_,_,_])]).

Variant = die_schoensten ;

Variant = die_bloedsten ;

no
| ?-
```

Natürlich ist die Suche in der vorliegenden Situation witzlos, es geht mir hier nur um die Demonstration des Aufrufs.

An dieser Stelle möchte ich die Übersicht beenden. Ich habe die elementaren Funktionen kurz umrissen, um eine Idee von der Arbeit mit dem System zu vermitteln. Die nachfolgenden Abschnitte beschäftigen sich eingehender mit den einzelnen Aspekten, also wie eine Lernumgebung aussieht, wie Varianten verwaltet werden, wie Daten modifiziert und selektiert werden—allerdings nur soweit, wie es der Rahmen dieser Arbeit zuläßt. Die genaue Spezifikation der Programmschnittstelle ist in Anhang B aufgeführt.

4.2 Lernumgebungen

Eine Lernumgebung legt als erstes den Ort fest, an dem die Daten abgelegt werden. Wie wir im vorherigen Abschnitt gesehen haben, geschieht das durch Angabe eines Verzeichnisses. Dieses Verzeichnis enthält wiederum eine Menge von Unterverzeichnissen, und zwar für jede Datenklasse eins (auf die Vorteile dieser Aufteilung gehe ich weiter unten ein). Des weiteren umfaßt eine Lernumgebung die Definition der Datenklassen, der Partitionierungsschlüssel und der Prädikate in Fakten-Datenklassen. All diese Festlegungen werden in einer Deklarationsdatei abgelegt, die dann in das Tool eingeladen werden muß—somit bleibt das Programm anwendungsunabhängig. Die Deklarationsdatei bestimmt also, welcher Datenpool benutzt werden soll und wie er strukturiert ist.

4.2.1 Format der Deklarationsdatei

Die Deklarationen werden in einer Textdatei durch Prolog-Fakten repräsentiert, wobei drei Arten zu unterscheiden sind: `directory/1`-Fakten, `dataclass/4`-Fakten und `predicate/3`-Fakten.

Das `directory/1`-Fakt spezifiziert das Hauptverzeichnis, unter dem sich die Unterverzeichnisse für die Datenklassen befinden; auch interne Verwaltungsinformationen werden dort abgelegt. Das Format der Deklaration lautet

```
directory(<pfad>).
```

Der komplette Pfad ist vom Wurzelverzeichnis aus anzugeben.

Die Definition der Datenklassen erfolgt durch Fakten der Form

```
dataclass(<name>, <typ>, <verzeichnis>, <schluessel>).
```

Das erste Argument gibt den Namen der Datenklasse an; selbstverständlich muß jede Datenklasse einen eindeutigen Namen haben. Das zweite Argument legt die Art der Daten fest, die die Datenklasse umfaßt: Fakten oder Regeln. Das wird durch die beiden Terme `facts` und `rules` ausgedrückt. An dritter Stelle steht das Unterverzeichnis, das die Datensätze zu der Datenklasse enthält, und an vierter Stelle ist der Partitionierungsschlüssel aufgeführt. Der Schlüssel ist in

Form einer Liste spezifiziert, die den Term `predname` für den Prädikatsnamen und zusätzlich bei Fakten Argumenttypen beinhalten kann. Beispielsweise wird die Datenklasse der Sensordistanzmessungen in der BLearn-Deklarationsdatei beschrieben durch

```
dataclass(measurements, facts, '\Measurements', [trace]).
```

Prädikatsdeklarationen werden durch `predicate/3`-Fakten dargestellt. Sie haben folgende Struktur:

```
predicate(<datenklasse>, <name>/<aritaet>, <argumente>).
```

Zuerst ist der Name der Datenklasse anzugeben, zu der die Deklaration gehört (folglich sind Prädikatsdeklarationen immer an eine Datenklasse gebunden). An zweiter Stelle steht ein Term, der sich aus Prädikatsnamen und Stelligkeit zusammensetzt. Anhand dieser beiden Informationen kann zu jedem Fakt einer Datenklasse die entsprechende Prädikatsdeklaration gefunden werden. Oder mit anderen Worten: Prädikatsname und Stelligkeit legen fest, auf welche Fakten sich die Deklaration bezieht. Deshalb ist es auch möglich, innerhalb einer Datenklasse gleichnamige Prädikate unterschiedlicher Stelligkeit und Prädikate derselben Stelligkeit, aber mit verschiedenen Namen zu deklarieren. Als letztes kommt die Definition der Argumenttypen, die als Liste von Termen vorliegen muß. Die Reihenfolge der Argumente gibt dabei die Anordnung der Typnamen in der Liste vor. Bleiben wir beim Beispiel der Sensordistanzmessungen. Die `messung/13`-Fakten werden deklariert durch

```
predicate(measurements, messung/13,
          [trace, time, sensor, distance, sx, sy, sz,
          sorientation, px, py, pz, object, edge]).
```

Wie ich die Argumenttypen im einzelnen benenne, ist unerheblich. Der Vorteil dieser Deklaration liegt in der Abstraktion von den Argumentpositionen, die durch die Reihenfolge der Typen implizit gegeben sind. Ändert sich möglicherweise die Argumentstruktur dahingehend, daß die Tracekennung an die zweite Stelle „rutscht“, so sind in der Liste `trace` und `time` gegeneinander auszutauschen. Für den Benutzer ist jedoch alles beim alten geblieben. Möchte er auf die Tracekennung zugreifen, so gibt er nur den Typ `trace` an und das Tool ermittelt automatisch die entsprechende Argumentposition.

4.2.2 Laden der Deklarationen

Der erste Schritt bei der Benutzung des Data Management Tools besteht darin, eine Deklarationsdatei einzulesen. Das erledigt der Befehl `load_declarations`, der als Argument einen Dateinamen erwartet. Ich werde im weiteren die BLearn-Anwendung heranziehen, um die Kommandos zu verdeutlichen. Deshalb lautet die Anweisung in diesem Fall

```
load_declarations('BLearnDeklarationen').
```

Bereits vorher bestehende Deklarationen werden aus der Prolog-Wissensbasis entfernt, so daß die Lernumgebung jederzeit gewechselt werden kann. Trat beim Laden ein Fehler auf, so läßt sich die Fehlerursache über die Routine `dekl_error/1` ermitteln. Ansonsten sind die eingelesenen Fakten aus der Deklarationsdatei allgemein zugreifbar.

Anschließend muß das Tool mittels des Befehls `varianten_init` an die aktuelle Lernumgebung angepaßt werden. Dabei werden alle erforderlichen und noch nicht existierenden Verzeichnisse und Verwaltungsdateien angelegt und bestehende Variantentabellen eingelesen. Zu jeder Datenklasse gibt es eine solche Variantentabelle, in der die vorhandenen Varianten aufgeführt sind. Dann läßt sich mit dem Datenpool arbeiten. Z.B. kann man sich Informationen über eine Datenklasse anzeigen lassen:

```
| ?- dataclass_info(measurements).

DATENKLASSE: measurements

Art der Daten:    Fakten
Unterverzeichnis: /Measurements
Schluessel:       [trace]
Praedikate:       messung(trace,time,sensor,distance,sx,sy,sz,
                          sorientation,px,py,pz,object,edge)
Anzahl Varianten: 2


yes
| ?-
```

Die Routine `dataclass_info/1` ist gerade dann nützlich, wenn die Prädikatsdeklarationen noch einmal eingesehen werden müssen. Mir ist es häufig passiert, daß ich die Bezeichnungen der Argumenttypen vergaß.

4.2.3 Arbeiten mit verschiedenen Deklarationsdateien

Die Verwendung verschiedener Deklarationsdateien erlaubt das Arbeiten auf mehreren Datenpools. Doch es ist auch möglich, auf ein und demselben Datenbestand mit unterschiedlichen Deklarationen zu operieren—sogar parallel im Mehrbenutzerbetrieb. Inwiefern das sinnvoll ist, muß von Fall zu Fall abgeklärt werden; ich möchte hier nur die diversen Möglichkeiten darstellen.

Zunächst einmal ist die Benennung der Datenklassen frei; jeder Benutzer kann also für die gleiche Klasse einen anderen Namen wählen. Entscheidend ist allein das Unterverzeichnis, das bei der Definition der Datenklasse angegeben wird. Deshalb lassen sich die Namen von Datenklassen nachträglich auch noch ändern.

Des weiteren können durch Datenklassen-Deklarationen Ausschnitte über dem Datenpool festgelegt werden. Denn man braucht ja immer nur die Datenklassen aufführen, die benutzt werden sollen—auch wenn weitere existieren. Z.B. gibt es bei BLearn auch andere Ansätze, die Sensormerkmale darzustellen. Für jeden dieser Ansätze definiert man eine eigene Datenklasse Sensormerkmale,

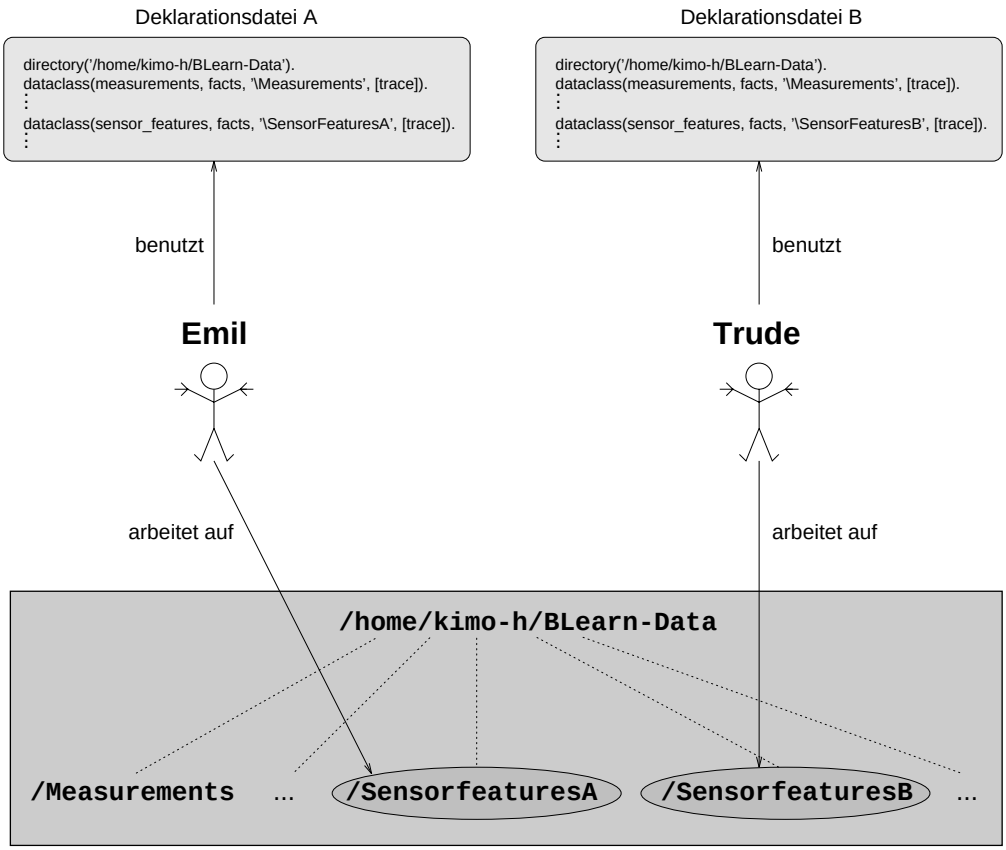


Abbildung 4.2: Benutzung alternativer Deklarationsdateien

der jeweils ein spezielles Unterverzeichnis zugeordnet wird. Entsprechend sind verschiedene Deklarationsdateien zu erstellen, die sich nur darin unterscheiden, daß für die Datenklasse `sensor_features` ein anderes Verzeichnis und andere Prädikatsdeklarationen vorkommen. In Abhängigkeit davon, welche Deklarationsdatei benutzt wird, kann man so mit verschiedenen Repräsentationen arbeiten—natürlich auch mehrere Benutzer gleichzeitig (Abbildung 4.2).

Ausschnitte lassen sich aber auch über die Wahl der Prädikatsdeklarationen bestimmen. Aus einem Datensatz werden nämlich immer nur diejenigen Fakten betrachtet, die auch tatsächlich deklariert sind.

Abschließend bleibt noch zu erwähnen, daß der Partitionierungsschlüssel zu einer Datenklasse jederzeit geändert werden kann. Zwar behält eine Variante den ihr einmal zugewiesenen Schlüssel, doch innerhalb einer Datenklasse kann jede Variante einen anderen Schlüssel besitzen. Das heißt auch, daß verschiedene Anwender für eine Datenklasse unterschiedliche Schlüssel definieren können. Angenommen, bei Emil und Trude sei das der Fall. Dann werden alle Varianten, die Emil anlegt, nach Emils Schlüssel aufgesplittet und Trudes Varianten bekommen ihren Schlüssel zugewiesen. Auch wenn beide auf demselben Unter-

verzeichnis (derselben Datenklasse) operieren.

4.3 Variantenverwaltung

Unter dem Begriff Variantenverwaltung verstehe ich alle Aufgaben des Tools, die mit der Registrierung von Datensätzen zusammenhängen. Dabei geht es ausdrücklich nicht um den Inhalt von Varianten (abgesehen von der Inhaltsbeschreibung), sondern um Datensätze als Ganzes. Intern handelt es sich um die Verwaltung der Dateien bzw. Partitionen.

4.3.1 Anlegen von Varianten

Erzeugt wird ein neuer Datensatz mit der Routine `new_variant`, wie wir es in Abschnitt 4.2 bereits gesehen haben:

```
new_variant(<datenklasse>, <name>, <eigenschaften>).
```

Das erste Argument bezeichnet die Datenklasse, der der Datensatz angehören soll, das zweite Argument gibt den Namen der Variante an. Dazu ist zu bemerken, daß jede Datenklasse ihren eigenen Namensraum hat. D.h. innerhalb einer Datenklasse müssen alle Varianten eindeutige Namen besitzen, Varianten unterschiedlicher Datenklassen können jedoch durchaus gleich benannt sein. Das dritte Argument spezifiziert die Eigenschaften der Variante, insbesondere die Art der Erzeugung. Sie werden in Form einer Liste übergeben. Erinnern wir uns: In Kapitel 2 war unter dem Design-Ziel „Alternative Datensätze“ gefordert worden, daß zu jedem Datensatz der Zeitpunkt der Erstellung, der Benutzername und ein Kommentar registriert wird. Dafür stehen die Terme `date([<jahr>, <monat>, <tag>, <stunde>, <minute>, <sekunde>])` sowie `user(<name>)` und `comment(<term>)`. Zusätzlich habe ich eine weitere Eigenschaft eingeführt, die über den Term `attribute(<term>)` bestimmt wird. Jeder Variante ist ein spezielles Attribut zugeordnet, das zum Wiederauffinden eines Datensatzes genutzt werden kann. In BLearn ist es beispielsweise so, daß die Daten aufgeteilt werden können bzgl. der Räume, in denen die zugrundeliegenden Roboterdaten gewonnen wurden. Diese Zusatzinformation kann den Datensätzen als Attribut zugewiesen werden, so daß es nachher ein leichtes ist, die Varianten herauszusuchen, die sich auf einen gewissen Raum beziehen.

Wichtig für die Erkennung doppelter Datensätze ist die Beschreibung des Varianteninhalts, also die Art der Erzeugung. Sie wird über die folgenden zwei Terme spezifiziert: `generator(<erzeuger>)` und `source(<datenquellen>)`. Der erste Ausdruck steht für die Methode, wie die Daten generiert werden, also durch den Benutzer (`generator(user)`), durch eine Auswahl (`generator(selection)`) oder durch ein Programm (`generator(program(<name>))`). Im letzten Fall muß der Term `<name>` den Namen des Programms sowie ggf. die Programmparameter repräsentieren; die Struktur des Terms legt der Benutzer für jedes

Programm eindeutig fest[2]. Die Ausgangsdaten werden als Liste von Samplekennungen angegeben, z.B. `source([sid1, sid7])` unter der Annahme, daß die Samples `sid1` und `sid7` existieren. Das Tool übersetzt diese Liste in eine Liste von Datenquellen, wobei jede Datenquelle spezifiziert ist durch die Datenklasse, den Variantennamen und die Auswahlmethode (das Format dafür ist `[<datenklasse>, <variante>, <methode>]`). Natürlich kann der Anwender auch diese explizite Beschreibung verwenden.

Die exakte Definition des Aufrufs von `new_variant` kann in Anhang B eingesehen werden, mir erscheint es an dieser Stelle sinnvoller, den Sachverhalt an drei Beispielen zu verdeutlichen. Die Beispiele beziehen sich auf die drei möglichen Arten der Datenerzeugung:

Der Benutzer generiert die Daten „per Hand“: Wir möchten einen Datensatz erstellen, über den wir keine exakte Aussage bzgl. des Inhalts machen können. Das wird durch die Eigenschaften `generator(user)` und `source([])` dargestellt. Dann könnte das Anlegen einer solchen Variante in der Datenklasse der Basishandlungsmerkmale so aussehen:

```
| ?- new_variant(action_features, beispiel1,
                 [user(zitzler), date([96,3,4,17,44,0]),
                 comment('Keine Aussage ueber den Inhalt'),
                 attribute(raum1), source([]), generator(user)]).

yes
| ?-
```

Ein Programm generiert die Daten: Angenommen wir wollen die Basishandlungsmerkmale mit einem Programm names *basic-action* berechnen. Das Programm benötigt dazu Daten aus der Datenklasse der Roboterpositionen, die wir bereits durch eine syntaktische Auswahl in dem Sample `sid2` zusammengestellt haben. Um nun den Datensatz der Basishandlungsmerkmale anzulegen und registrieren zu lassen, könnten wir schreiben:

```
| ?- new_variant(action_features, beispiel2,
                 [user(zitzler), date([96,3,4,17,50,0]),
                 comment('Vom Programm basic-action generiert'),
                 attribute(raum1), source([sid2]),
                 generator(program(basic-action))]).

yes
| ?-
```

Das Programm *basic-action* kann nicht über Programmparameter gesteuert werden, deshalb genügt die Angabe des Namens. Würden jedoch zwei

[2]Da das Tool unabhängig sein soll von einer speziellen Anwendung, muß der Benutzer dafür sorgen, daß er für jedes Programm immer die gleiche Termstruktur verwendet. Ansonsten können doppelte Datensätze nicht erkannt werden.

Parameter A und B die Datengenerierung beeinflussen, so müßten diese mitangegeben werden. Ist A = 11 und B = 5, so ließe sich das durch `generator(program(basic-action(11,5)))` ausdrücken.

Die Daten stammen aus einer Auswahl: Wir haben ein Sample `sid3` aus einer Variante der Datenklasse `action_features` erstellt. Aus irgendeinem Grund gefällt uns diese Zusammenstellung so gut, daß wir sie als eigenständige Variante abspeichern wollen. Der leere Datensatz würde zunächst einmal erzeugt über

```
| ?- new_variant(action_features, beispiel3,
                 [user(zitzler), date([96,3,4,17,57,0]),
                 comment('Ueber eine Auswahl erstellt'),
                 attribute(raum1), source([sid3]),
                 generator(selection)]).

yes
| ?-
```

Um es noch einmal hervorzuheben: Der Befehl `new_variant` dient einzig und allein dazu, einen leeren Datensatz anzulegen und ihn registrieren zu lassen. Die Daten werden erst danach hinzugefügt (siehe Abschnitt 4.4.2). Das hat u.a. den Vorteil, daß ein Mehrfachvorkommen eines Datensatzes bereits erkannt wird, bevor die eigentlichen Daten vorliegen. Denn `new_variant` kann ja schon anhand der Erzeugungsmethode (`generator(...)`) und der Datenquellen (`source(...)`) überprüfen, ob der Datensatz existiert. D.h. gibt es eine Variante, die aus denselben Daten generiert wurde, so müssen sich die Datensätze in der Methode der Datenerzeugung unterscheiden. Ausnahme sind Varianten mit der Kennzeichnung `generator(user)`—sie sind per definitionem einmalig.

Die Prozedur `new_variant` schlägt fehl, wenn der Name der Variante schon vergeben ist, die Variante bereits existiert oder ein Bedienungsfehler aufgetreten ist. Die Fehlerursache kann über die Routine `varianten_error/1` ermittelt werden, z.B.:

```
| ?- new_variant(action_features, beispiel4,
                 [user(zitzler), date([96,3,4,18,12,0]),
                 comment('Die Variante existiert schon'),
                 attribute(raum1), source([sid2]),
                 generator(program(basic-action))]).

no
| ?- varianten_error(Fehler).

Fehler = variant_exists

| ?-
```

Ein Wort noch zum Partitionierungsschlüssel: `new_variant` weist der Variante automatisch den Schlüssel zu, der in der Lernumgebung für die Datenklasse

deklariert ist. Es gibt jedoch auch die Möglichkeit, explizit einen Schlüssel festzulegen, indem er als viertes Argument übergeben wird (siehe Beschreibung von `new_variant/4` in Anhang B). In der Regel merkt der Benutzer aber nichts von den Partitionen, es sei denn er läßt sich über `variant_info/2` die Informationen zu einer Variante anzeigen. Da wird dann aufgeführt, welche Partitionen es gibt. Um näheres über die Variante `traces701-740` der Datenklasse der Sensordistanzmessungen zu erfahren, würde man eintippen:

```
| ?- variant_info(measurements, 'traces701-740').

VARIANTE: traces701-740

Datenklasse:     measurements
Erstellt von:    volker
Erstellt am:     13.6.95 um 0:00 Uhr
Attribut:        Traces7
Kommentar:       Daten aus /home/kimo-a/ls8/BLearn/Traces7/Messungen
Generiert durch: Benutzer
Partitionen:     [t701], [t702], [t703], [t704], [t705], [t706],
                 [t707], [t708], [t709], [t710], [t711], [t712],
                 [t713], [t714], [t715], [t716], [t717], [t718],
                 [t719], [t720], [t721], [t722], [t723], [t724],
                 [t725], [t726], [t727], [t728], [t729], [t730],
                 [t731], [t732], [t733], [t734], [t735], [t736],
                 [t737], [t738], [t739], [t740]


yes
| ?-
```

Zum Schluß möchte ich darauf hinweisen, daß die Eigenschaften einer Variante nachträglich geändert werden können, beispielsweise um den Kommentar zu ergänzen. Das bewerkstelligt die Prozedur `change_variant_properties/3`. Sie wird analog zu `new_variant/3` aufgerufen, mit dem Unterschied, daß die Variante schon existieren muß:

```
change_variant_properties(<datenklasse>, <name>,
                          <eigenschaften>).
```

4.3.2 Löschen von Varianten

Eine Variante kann über den Befehl `delete_variant` gelöscht werden. Die Aufrufsyntax lautet:

```
delete_variant(<datenklasse>, <variante>).
```

Das bedeutet, daß sowohl die Registrierung gelöscht wird, als auch alle Partitionen entfernt werden.

Allerdings sollte man damit vorsichtig sein, denn möglicherweise sind andere Varianten aus der zu löschenden erzeugt worden. Nehmen wir an, wir haben das

Sample `sid10` aus der Variante A gezogen; Variante B wurde u.a. erzeugt mit den Eigenschaften `user(selection)` und `source([sid10])`. Wird nun Variante A entfernt, ist die Inhaltsbeschreibung von Variante B nicht mehr eindeutig. Der Benutzer könnte erneut eine Variante mit dem Namen A anlegen, die sich aber grundlegend von ihrer Vorgängerin unterscheidet. In diesem Fall wäre die Inhaltsbeschreibung von Variante B sogar falsch.

Eigentlich sollte das Tool solche Inkonsistenzen erkennen und anzeigen, welche Varianten aus der zu löschenden generiert worden sind. Das ist aber wegen eines Umstands nicht möglich: Das Tool kann immer nur den Ausschnitt des Datenpools betrachten, der in der Lernumgebung festgelegt wurde. Eine notwendige Überprüfung beim Löschen einer Variante wäre aber, ob irgendeine andere Variante diese Variante als Datenquelle (`source(...)`) aufgeführt hat. Bei dem Beispiel von Emil und Trude in Abschnitt 4.2.3 weiß das Programm aber einmal nichts von dem Unterverzeichnis `\SensorFeaturesA` und das andere Mal nichts von den Varianten in `\SensorFeaturesB`. D.h. es könnten nicht alle erforderlichen Varianten geprüft werden.

Eine Lösung wäre, das Löschen von Varianten zu verbieten, oder man müßte fordern, daß immer der ganze Datenpool deklariert wird. Im zweiten Fall trägt der Benutzer immer noch Verantwortung, denn er muß dafür sorgen, daß auch wirklich alles deklariert ist. Außerdem wäre das Definieren von Ausschnitten, wie bei Emil und Trude, nicht mehr möglich. Deshalb sehe ich einen Kompromiß darin, das Löschen einer Variante als eine Ausnahme zu betrachten. Dann ist das Entfernen einer Variante zwar erlaubt, doch der Anwender trägt die Verantwortung dafür, daß keine Inkonsistenzen auftreten.

4.3.3 Suche nach Varianten

Eine wichtige Funktion des Data Management Tool ist das Suchen von Varianten. Der Benutzer kann sich zwar mit `variant_info/2` alle Varianten des Datenpools oder einer Datenklasse anzeigen lassen, indem er das Backtracking von Prolog nutzt; doch eine gezielte Suche ist auf diese Art und Weise nicht möglich. Dafür gibt es die Routine `variant_properties/3`, die mehrfach eingesetzt werden kann. Zuerst einmal lassen sich mit ihr die Eigenschaften einer Variante ermitteln, wenn das letzte Argument nicht instanziiert ist (Datenklasse und Variante können, müssen aber nicht angegeben werden):

```
| ?- variant_properties(action_features, beispiel1, Eigenschaften).

Eigenschaften = [user(zitzler),date([96,3,4,17,44,0]),
comment('Keine Aussage ueber den Inhalt'),attribute(raum1),
source([]),generator(user)]

| ?-
```

Die Eigenschaften werden in dem gleichen Format zurückgegeben, wie es auch `new_variant` erwartet.

Die zweite Möglichkeit besteht darin, die Eigenschaften zu spezifizieren und die ersten beiden Argumente (oder nur eins von beiden) nicht zu instanziieren.

Dann kann nach Varianten mit bestimmten Eigenschaften gesucht werden, wie ich es in Abschnitt 4.1 schon vorgestellt habe. Die Liste der Eigenschaften hat weder vollständig zu sein, noch müssen alle Terme der Liste variablenfrei sein. Das Programm versucht vielmehr, die angegebenen Eigenschaften mit den intern für die Varianten vermerkten zu unifizieren. Daher ist z.B. auch folgendes machbar:

```
| ?- variant_properties(action_features, Name, [attribute(raum1),
                        user(zitzler), comment(Kommentar)]).

Name = beispiel1,
Kommentar = 'Keine Aussage ueber den Inhalt' ;

Name = beispiel2,
Kommentar = 'Vom Programm basic-action generiert' ;

Name = beispiel3,
Kommentar = 'Ueber eine Auswahl erstellt' ;

no
| ?-
```

Man kann auch alle Argumente beim Aufruf von `variant_properties` angeben. In diesem Fall wird überprüft, ob eine Variante bestimmte Eigenschaften erfüllt:

```
| ?- variant_properties(action_features, beispiel2,
                        [generator(program(basic-action))]).

yes
| ?-
```

4.3.4 Überprüfung der Existenz einer Variante

Betrachten wir die Situation, daß ein Benutzer eine Variante anlegen will, `new_variant` jedoch fehlschlägt, weil es die Variante schon gibt. Dann möchte er selbstverständlich wissen, wie diese Variante heißt, um auf die Daten zugreifen zu können. Es gäbe die Möglichkeit, die Variante über `variant_properties` herauszusuchen. Dabei gibt es allerdings ein Problem, das mit der Repräsentation der Datenquellen (`source(...)`) zusammenhängt. Jede Datenquelle wird beschrieben durch eine Liste

`[<datenklasse>, <variante>, <methode>]`,

die die Datenklasse, die Variante und die Auswahlmethode (dargestellt durch den Term `synt_select(<constraints>)`) spezifiziert. Auch wenn der Benutzer immer Samplekennungen in `source(...)` angeben kann, wandelt das Tool sie in dieses Format um[3]. Das läßt sich an der Variante `beispiel2` sehen, die aus dem Sample `sid2` erstellt wurde (vgl. Abschnitt 4.3.1, Seite 59):

[3] Samplekennungen sind ja temporär und prozeßabhängig—sie sind daher nicht geeignet, den Inhalt einer Variante allgemeingültig zu beschreiben.

```
| ?- variant_properties(action_features, beispiel2,
                         [source(Datenquellen)]).

Datenquellen = [[positions,'rp_traces701-740',
synt_select([item(trace,[t715])])]]

| ?-
```

Versucht man nun mittels Unifikation zu prüfen, ob zwei Varianten dieselben Datenquellen besitzen, so werden Übereinstimmungen nicht immer richtig erkannt. Das liegt daran, daß Reihenfolgeaspekte in der Liste der Datenquellen und bei den Constraints nicht ausgeglichen werden. Z.B. sind die folgenden beiden Terme semantisch äquivalent, jedoch nicht unifizierbar:

```
| ?- source([[action_features, a, synt_select(all)],
             [measurements, b, synt_select(item(trace, [t1,t7]))]]) =
     source([[measurements, b, synt_select(item(trace, [t7,t1]))],
             [action_features, a, synt_select(all)]]).

no
| ?-
```

Aus diesem Grund stellt das Tool eine weitere Funktion zur Verfügung, die genau das mitberücksichtigt: `check_variant_existence/3`. Sie benötigt die Angabe der Datenklasse sowie der Eigenschaften `source` und `generator` und gibt einen Variantennamen zurück. Existiert keine Variante, die auf die spezifizierte Art erzeugt wurde (genauer: deren Daten auf diese Weise generiert wurden), so wird der Term `no_variant` zurückgeliefert. Unter Annahme, daß das Sample `sid2` von vorhin noch besteht, kann `check_variant_existence` folgendermaßen eingesetzt werden:

```
| ?- check_variant_existence(action_features,[source([sid2]),
                              generator(program(basic-action))], Name).

Name = beispiel2

| ?-
```

4.4 Datenmanipulation

Nachdem dargestellt wurde, wie Varianten erzeugt und gelöscht werden können, geht es nun um die Möglichkeiten, Varianten zu modifizieren. Ganz konkret ist damit das Hinzufügen und Entfernen von Elementen gemeint. Weiterhin können Elemente in ihre Bestandteile zerlegt und zusammengesetzt werden, was hauptsächlich bei Fakten interessant ist (Abstraktion von den Argumentpositionen). Darauf möchte ich zuerst eingehen.

4.4.1 Zusammensetzen und Zerlegen von Fakten und Regeln

Die Funktion zur Erzeugung eines Mengenelements ist `compose/4`. Sie setzt das Element aus den Bestandteilen zusammen, die als Argumente übergeben

werden. Allerdings ist die Art und Anzahl der Bestandteile abhängig vom Typ des Elements. Bei Fakten richtet sich das nach Prädikatsnamen und Stelligkeit, bei Regeln hingegen sind die Komponenten stets dieselben: Regelkopf und Regelkörper. Deshalb benötigt `compose` zwei Angaben, um ein Element generieren zu können: zum einen die Art der Bestandteile und zum anderen die Bestandteile selbst. Des weiteren ist natürlich die Datenklasse wichtig, um zu wissen, um welche Daten es sich handelt. Der Aufruf von `compose` gestaltet sich dementsprechend:

```
compose(<datenklasse>, <bestandteilarten>, <bestandteile>,
        <element>).
```

Die Bestandteilarten werden in einer Liste übergeben, die dem Format des Partitionierungsschlüssels ähnelt. Bei Fakten sind die Argumenttypen (wie in der Lernumgebung deklariert) und der Prädikatsname (dargestellt durch den Term `predname`) anzugeben, bei Regeln sind folgende zwei Listen zulässig: `[head, body]`, die für die Bestandteile Kopf und Körper steht, und `[clause]`, die eine Regel als Ganzes meint[4]. Analog dazu beinhaltet eine weitere Liste die Bestandteile, und zwar in zu den Bestandteilsarten korrespondierender Reihenfolge. Das wird an zwei Beispielen deutlicher:

```
| ?- compose(measurements,
             [predname,trace,time,sensor,distance,sx,sy,sz,
              sorientation,px,py,pz,object,edge],
             [messung,t715,2,s11,0.54,0.54,0.546,0,165,0,0,0,0,3],
             Fact).

Fact = messung(t715,2,s11,5.4E-01,5.4E-01,5.46E-01,0,165,0,0,0,0,3)

| ?- compose(pf_rules, [head,body], [p((X)),(a((X)), b((X)))], Rule).

X = _674,
Rule = p(X):-a(X),b(X)

| ?-
```

Wichtig bei `compose` ist, daß immer alle Bestandteile angegeben werden. Umgekehrt kann die Routine aber auch zur vollständigen Zerlegung eines Elements genutzt werden, indem nur das Element und die Datenklasse spezifiziert werden (vgl. Abschnitt 4.1, Seite 53).

Das Gegenstück zu `compose` ist `decompose`. Mit Hilfe dieser Routine kann auf einzelne Komponenten eines Elements zugegriffen werden. Der Aufruf ist prinzipiell derselbe wie bei `compose`, allerdings ist die Reihenfolge der Argumente anders und es müssen nicht alle Bestandteilsarten angegeben werden:

```
decompose(<datenklasse>, <element>, <bestandteilarten>,
          <bestandteile>).
```

[4] Die Bestandteilsart `[clause]` habe ich ursprünglich für das Zerlegen von Regeln eingeführt, damit die Variablenbindung in Prolog nicht verloren geht. Das ist wichtig, wenn man mit den Regeln später in Prolog weiterarbeiten möchte. Der Vollständigkeit halber habe ich `[clause]` bei der Zusammensetzung mitaufgenommen.

Bei Regeln ist als Bestandteilsart jedoch auch der Term `predname` erlaubt, womit gesondert auf den Prädikatsnamen zugegriffen werden kann.

In diesem Sinne entspricht `decompose` der Funktion `get`, die ich in Abschnitt 3.6 vorgeschlagen habe:

```
| ?- decompose(basic_perceptual_features,
               stable(t701,75,s0,5,23,0), [trace], [Trace]).

Trace = t701

| ?-
```

Allerdings kann auf mehrere Bestandteile gleichzeitig zugegriffen werden.

Zum Schluß möchte ich noch darauf hinweisen, daß `compose` und `decompose` nicht unbedingt benutzt werden müssen, um mit Mengenelementen arbeiten zu können. Sie sind vielmehr als eine zusätzliche Möglichkeit anzusehen, mit der gerade bei Fakten vom konkreten Aussehen abstrahiert werden kann. Das ist vor allem für Programme nützlich, die mit dem Data Management Tool gekoppelt sind. Sie können durch die Verwendung der beiden Routinen davon unabhängig sein, wie bei Fakten die Argumente angeordnet sind. Doch will beispielsweise der Benutzer ein Fakt zu einer Variante hinzufügen, so muß er nicht zwingend dieses Fakt mit `compose` generieren—er kann es natürlich auch direkt von Hand eingeben.

4.4.2 Hinzufügen und Entfernen von Fakten und Regeln

Wurde eine leere Variante erzeugt, so müssen anschließend die eigentlichen Daten eingespeist werden. Das ist mit dem Befehl `add` möglich. In der einfachsten Form werden der Routine die Datenklasse, der Name der Variante und eine Liste von einzufügenden Elementen übergeben:

```
add(variant(<datenklasse>, <variante>), <elemente>).
```

Alle Elemente, die noch nicht in der Variante vorkommen, werden dann in den Datensatz mitaufgenommen. Dabei weist das Tool die Elemente intern den jeweiligen Partitionen zu, notfalls werden noch nicht existierende Partitionen angelegt. Jedoch wird unterschieden nach positiven und negativen Elementen. Diese Forderung hatte ich in Kapitel 2 formuliert, damit Beispiele dementsprechend gekennzeichnet werden können. Das eben vorgestellte `add/2` betrachtet die angegebenen Elemente automatisch als positiv, mit `add/3` kann jedoch explizit das Vorzeichen gesetzt werden:

```
add(variant(<datenklasse>, <variante>), <element>,
            <vorzeichen>)
```

Als Vorzeichen sind die Terme `pos` und `neg` zulässig. Da bei unserer Anwendung nur Hornklauseln verwendet werden , haben Regeln ausschließlich positive

Köpfe. Folglich werden Regeln unabhängig vom angegebenen Vorzeichen immer als positiv gekennzeichnet.

Beim Aufruf von `add` wird vielleicht verwundern, daß das erste Argument ein zusammengesetzter Term ist, wo doch bei den bisherigen Routinen Datenklasse und Variantenname immer als eigenständige Argumente auftraten. Die Erklärung ist einfach: Auch Samples können über `add` modifiziert werden. In diesem Fall ist als erstes Argument der Term `sample(<kennung>)` anzugeben.

Das Entfernen von Elementen aus einer Variante oder einem Sample gestaltet sich ähnlich. Es gibt zwei Prozeduren `erase/2` und `erase/3`, die auf genau dieselbe Art wie `add/2` und `add/3` aufgerufen werden.

Ich möchte noch auf eine Routine hinweisen, die in einer Datei abgelegte Elemente zu einer Variante hinzufügt:

```
add_file_to_variant(<datenklasse>, <variante>, <datei>).
```

Sie wurde ursprünglich entworfen, um bereits vorliegende Daten in den Datenpool einzuspeisen, also konkret die in Dateien abgelegten BLearn-Daten in die Verwaltungsstruktur des Tools zu übertragen. Selbstverständlich läßt sie sich auch für andere Zwecke einsetzen, z.B. zum Speichern eines Samples als eine Variante (mehr dazu in Abschnitt 4.5.3).

4.5 Datenselektion

Bis jetzt wurde hauptsächlich besprochen, wie mit Varianten, also extern abgelegten Datensätzen gearbeitet werden kann. Gegenstand dieses Abschnitts ist hingegen die Arbeit mit Datenmengen im Hauptspeicher (Samples). Die nun zu beantwortenden Fragen sind: wie werden aus Varianten Daten ausgewählt bzw. Samples erstellt und welche Operationen auf Samples sind möglich? Ich gehe daher im einzelnen auf die syntaktische Auswahl und die Case Selection ein, stelle anschließend die Operationen auf Samples vor und schildere am Ende, wie mit Domänen umgegangen wird.

4.5.1 Syntaktische Auswahl

Mit dem Kommando `synt_select` können Daten aus einer Variante selektiert und in den Hauptspeicher geladen werden. Die Daten werden ähnlich wie bei der `select`-Anweisung des Data Preparation Tools über Constraints beschrieben, also Kriterien, die die Daten eindeutig charakterisieren. Dabei werden für einzelne Bestandteile die erlaubten Werte angegeben. Bei Fakten lassen sich folglich Auswahlen über den Prädikatsnamen und die Argumente treffen, Regeln kann man einschränken über den Prädikatsnamen, den Regelkopf und den Regelkörper. Die Constraints werden als eine Liste von Termen der Form `item(<bestandteilart>, <werte>)` spezifiziert, wobei die Werte wiederum in einer Liste aufgeführt sind. Alternativ dazu sind auch die Terme `all` und `empty` als Constraints zulässig, wenn man alle bzw. keine Elemente selektieren will.

Um aus einer Variante Daten herauszuziehen, wird die Funktion `synt_select` folgendermaßen aufgerufen:

```
synt_select(variant(<datenklasse>, <name>), <constraints>,
            <sample>).
```

Das letzte Argument stellt das Ergebnis der Funktion dar und steht für die Kennung des erstellten Samples. Die Generierung des Samples erfolgt in mehreren Schritten: Zuerst bestimmt das Tool anhand der Constraints alle in Betracht kommenden Partitionen, im ungünstigsten Fall sind das sämtliche Partitionen der Variante. Daraufhin werden die Dateien nacheinander eingelesen und die Elemente mit den Constraints abgeglichen. Die Elemente, auf die die Beschreibung zutrifft, werden dann im Hauptspeicher als ein Sample abgelegt.

Bei der Selektion von Fakten gibt es jedoch noch zwei Besonderheiten:

1. Wird bei den Constraints auf einen Argumenttyp Bezug genommen, der nur für einige Fakten der Datenklasse deklariert ist, so hat das Kriterium auch nur bei diesen Fakten Einfluß auf die Selektion. Nehme ich beispielsweise eine Auswahl mit den Constraints `[item(trace, [t715])]` vor, so umfaßt das erstellte Sample alle Fakten, die als Tracekennung `t715` besitzen oder denen überhaupt keine Tracekennung zugeordnet ist.

2. Eine spezielle Bedeutung kommt der leeren Liste als Werteangabe zu. Somit wird die Menge der in Frage kommenden Fakten auf diejenigen eingeschränkt, die den entsprechenden Argumenttyp besitzen. Demnach liefert eine Auswahl mit den Constraints `[item(trace, [])]` alle Fakten, die eine Tracekennung enthalten.

Betrachten wir ein Beispiel:

```
| ?- synt_select(variant(measurements,'traces701-740'),
                 [item(trace, [t715])], Sample).

Sample = sid4

| ?-
```

Im Sample `sid4` sind nun alle Sensordistanzmessungen zum Trace `t715` zusammengefaßt. Aus diesem Sample kann erneut ausgewählt werden, indem analog zu `add` und `erase` das erste Argument durch den Term `sample(sid4)` ersetzt wird:

```
| ?- synt_select(sample(sid4), [item(sensor, [s0,s1])], Sample).

Sample = sid5

| ?- synt_select(sample(sid5), [item(time, [2])], Sample).

Sample = sid6

| ?-
```

Das Sample `sid5` enthält alle Distanzmessungen der Sensoren `s0` und `s1` zum Trace `t715`. Eine weitere Auswahl schränkt die Fakten auf den Zeitpunkt 2 ein (Sample `sid6`).

Möchten wir auf die Elemente eines Samples zugreifen, so muß die Routine `access_sample` benutzt werden. Sie liefert zwei Listen zurück, eine mit den positiven und eine mit den negativen Elementen[5]:

```
| ?- access_sample(sid6, PositiveElemente, NegativeElemente).

PositiveElemente = [
  messung(t715,2,s0,4.54E+00,1.46E+00,5.3E+00,0.0E+00,0.0E+00,
          0.0E+00,0.0E+00,0.0E+00,0,1),
  messung(t715,2,s1,4.54E+00,1.46E+00,5.42E+00,0.0E+00,1.5E+01,
          0.0E+00,0.0E+00,0.0E+00,0,1)],
NegativeElemente = []


| ?-
```

Es gibt noch eine erweiterte Fassung dieser Prozedur, die zusätzliche Informationen zu einem Sample liefert. Ich verweise dabei auf die Beschreibung von `access_sample/4` in Anhang B, denn der Benutzer wird wohl in der Regel die formatierte Ausgabe über `sample_info/1` vorziehen:

```
| ?- sample_info(sid6).

SAMPLE: sid6

Art der Daten:              Fakten
Datenklasse:                measurements
Ausgangsdaten:              Sample sid5
Methode:                    syntaktische Auswahl (synt_select)
Constraints:                time = [2]
Ursprungsvariante:          Variante traces701-740
Direkte Methode:            syntaktische Auswahl (synt_select)
Constraints:                time = [2]
                            sensor = [s0,s1]
                            trace = [t715]
Anzahl positiver Elemente: 2
Anzahl negativer Elemente: 0
```

[5]Die Elemente eines Samples werden intern ebenfalls in einer Liste gehalten. Jedoch sind auch andere Datenstrukturen denkbar, z.B. Bäume. Zudem unterstützt Quintus-Prolog das Modulkonzept, so daß die Fakten und Regeln direkt in ein Modul geladen werden könnten. Module haben den Vorteil, daß sehr schnell auf einzelne Elemente zugegriffen werden kann—das ist interessant bei syntaktischen Auswahlen auf Samples. Fakten werden allerdings nur dann schnell gefunden, wenn über die ersten Argumentstellen zugegriffen wird. Soll ein `messung`-Fakt gefunden werden, bei dem nur das dreizehnte Argument gegeben ist, schmilzt dieser Zeitvorteil gegenüber Listen jedoch rapide. In der Regel müssen bei einer Auswahl sowieso alle Elemente betrachtet werden. Listen bieten zudem die Möglichkeit, eine Reihenfolge auf den Elementen zu definieren—das wird benötigt, um Elemente z.B. nach Zeitpunkten zu sortieren. Nicht zuletzt benutzt das Data Preparation Tool ebenfalls Listen. Die Verwendung von Listen in meinem Programm vereinfacht die Einbindung dieses Tools sehr.

```
Kennung des DPT-Samples:   sample6


yes
| ?-
```

Unter anderem wird angezeigt, wie das Sample erzeugt wurde (nämlich durch eine Auswahl auf dem Sample `sid5`) und wie es direkt aus der ursprünglichen Variante gezogen werden kann. Diesen Zusammenhang veranschaulicht Abbildung 4.3. Die zuletzt aufgeführte Information über die Kennung des DPT-Samples hängt mit der Einbindung des Data Preparation Tools zusammen und ist hier nicht weiter von Interesse.

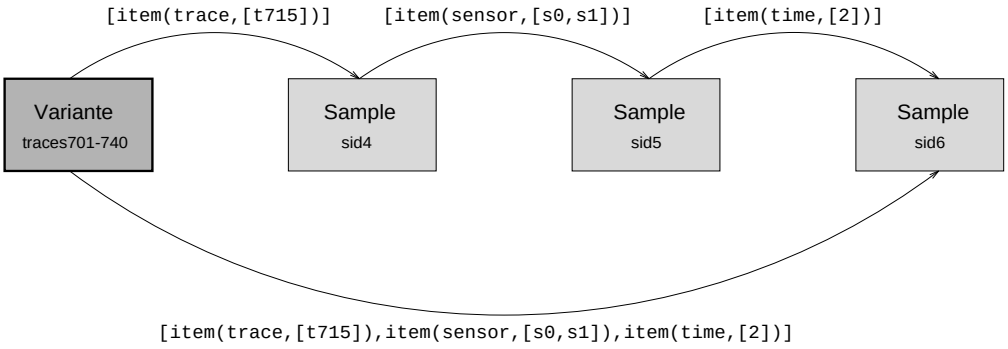


Abbildung 4.3: Samplefolgen und Direktauswahl

4.5.2 Case Selection

Die Fallauswahl erfolgt über den Befehl `case_select`, der als Eingabe zwei Samplekennungen und die Relationenliste erwartet:

```
case_select(<sample_beispiele>, <relationenliste>,
            <sample_hintergrundwissen>, <caseliste>).
```

Das erste Sample enthält die Beispiele, zu denen das relevante Hintergrundwissen (gegeben durch das zweite Sample) gesucht werden soll[6]. Die Relationenliste gibt an, wie Fakten des Hintergrundwissens zu einem Beispiel-Fakt in Beziehung stehen müssen, um für das Beispiel relevant zu sein. Das wird ausgedrückt über Relationen zwischen Argumenttypen, dargestellt im Format

```
[<argumenttyp>, <relation>, <argumenttyp>].
```

[6] Die Samples müssen Faktenmengen sein, da die Case Selection nur auf Fakten anwendbar ist.

Der erste Argumenttyp bezieht sich auf die Beispiel-Fakten, der dritte auf die Hintergrundwissen-Fakten. Als Relationen sind `=`, `<`, `>`, etc., erlaubt (siehe Anhang B). Das Ergebnis der Case Selection ist eine Menge von Listen, die an erster Stelle ein Beispiel-Fakt und nachfolgend die gefundenen Hintergrundwissen-Fakten enthalten.

Um die Benutzung von `case_select/4` klarer zu machen, nehme ich exemplarisch zwei Samples, die ich von Hand erstellt habe (dadurch wird die Darstellung übersichtlicher). Das erste Sample (`sid7`) beinhaltet die Basiswahrnehmungsmerkmale

```
no_movement(t715,75,s11,85,88,-999),
straight_away(t715,75,s11,88,112,45),
decr_peak(t715,75,s11,112,113,-88),
straight_away(t715,75,s11,113,131,44),
no_movement(t715,75,s11,131,133,-999),
stable(t701,285,s1,69,84,1),
incr_peak(t701,285,s1,84,85,88),
decr_peak(t720,77,s11,112,113,-88),
```

das zweite (`sid8`) die Sensormerkmale

```
s_jump(t715,s11,88,131,straight_away),
s_line(t701,s1,69,84,parallel).
```

Zu den Sensormerkmalen sollen nun die geeigneten Basiswahrnehmungsmerkmale gefunden werden, so wie es bereits in Kapitel 2, Seite 34, vorgestellt wurde:

```
| ?- case_select(sid8,
                 [[trace, '=', trace], [sensor, '=', sensor],
                  [time1, '=<', time1], [time1, '=<', time2],
                  [time2, '>=', time1], [time2, '>=', time2]],
                 sid7, Cases).

Cases = [
[s_jump(t715,s11,88,131,straight_away),
  straight_away(t715,75,s11,88,112,45),
  decr_peak(t715,75,s11,112,113,-88),
  straight_away(t715,75,s11,113,131,44)],
[s_line(t701,s1,69,84,parallel),
  stable(t701,285,s1,69,84,1)]]

| ?-
```

4.5.3 Operationen auf Samples

Die wohl wichtigste Operation auf einem Sample ist das Sortieren. Die Sortierung kann wahlweise auf- oder absteigend vorgenommen werden und erfolgt nach bestimmten Kriterien, die der Benutzer festlegt. Als Kriterien sind wieder die Bestandteilsarten wie bei `decompose` zulässig. Die Routine zum Sortieren heißt `sort_sample/3`:

```
sort_sample(<sample>, <bestandteilsarten>, <relation>).
```

Das erste Argument steht für die Samplekennung, das zweite für eine Liste von Bestandteilsarten und als letztes wird die Vergleichsrelation zwischen den Elementen angegeben (`<` oder `>`). Um das Sample `sid7` aufsteigend nach Startzeitpunkt und Endzeitpunkt zu sortieren, müßte man schreiben:

```
| ?- sort_sample(sid7, [time1,time2], <).

yes
| ?-
```

Da bei der Sortierung die vorherige Reihenfolge der Elemente soweit wie möglich erhalten bleibt, kann auch in erster Ordnung aufsteigend nach dem Startzeitpunkt und in zweiter Ordnung absteigend nach dem Endzeitpunkt sortiert werden:

```
| ?- sort_sample(sid7, [time2], >), sort_sample(sid7, [time1], <).

yes
| ?-
```

Als weitere Operationen auf Samples werden die Routinen `delete_sample/1` und `merge_samples/3` zur Verfügung gestellt. Mit `delete_sample` kann der Speicherplatz, den ein Sample belegt, freigegeben werden, `merge_samples` konstruiert aus zwei Samples durch Mengenvereinigung ein drittes.

Schließlich gibt es auch noch die Möglichkeit, die Elemente eines Samples aufzulisten (`list_sample(<sample>)`). Interessant ist aber vor allem die Prozedur `list_sample/2`, z.B. um ein Sample als eine Variante abzulegen. Angenommen wir wollen das Sample `sid6` mit den Sensordistanzmessungen als einen Datensatz abspeichern. Dafür legen wir als erstes eine leere Variante an:

```
| ?- new_variant(measurements, test,
                 [user(zitzler), date([96,8,3,15,9,0]),
                 comment('Nur fuer Testzwecke!'),
                 attribute(none), source([sid6]),
                 generator(selection)]).

yes
| ?-
```

Um nun die Daten des Samples in die Variante zu übertragen, werden sie zunächst in eine Datei geschrieben. Das erledigt `list_sample/2`, indem die Ausgabe in die Datei umgelenkt wird:

```
| ?- open(tempfile, write, Stream), list_sample(sid6, Stream),
     close(Stream).

Stream = '$stream'(5505027)

| ?-
```

Anschließend läßt sich `add_file_to_variant/4` einsetzen:

```
| ?- add_file_to_variant(measurements, test, tempfile),
     delete_file(tempfile).

yes
| ?-
```

Jetzt ist das Sample `sid6` als Variante `test` extern abgespeichert.

An dieser Stelle habe ich noch etwas zu `new_variant` nachzutragen: Wird ein Datensatz mit der Eigenschaft `generator(selection)` erzeugt, so müssen die in `source(...)` angegebenen Samples stets direkt aus der Ursprungsvariante generierbar sein. Denn sonst können die Datenquellen nicht mehr eindeutig beschrieben werden. Mit anderen Worten: Wurde ein Sample mit `add` oder `erase` manipuliert, dann kann es nicht mehr direkt aus der Variante gezogen werden. Das gilt auch für Samples, die durch `merge_samples` erzeugt wurden. Solche Samples müssen in mit `generator(user)` gekennzeichneten Datensätzen abgelegt werden.

4.5.4 Speichern und Laden von Domänen

Wurden während einer Sitzung mehrere Samples erstellt, so sind vielleicht einige darunter, die man immer wieder benutzen will. Da es aber mühsam ist, diese Samples bei jeder Sitzung von Hand zu generieren, gibt es die Routine `save_domain/2`. Sie speichert die Informationen, wie bestimmte Samples erzeugt wurden, in einer Datei ab. Das nutzt später `load_domain/3`, um die Samples automatisch zu generieren. Auf diese Art und Weise lassen sich Domänen definieren, die dann bei Bedarf geladen werden können.

Das Speichern einer Domäne ist sehr einfach:

```
save_domain(<samples>, <datei>).
```

Eine Liste von Samplekennungen charakterisiert die Samples, die unter dem Namen der Datei zu einer Domäne zusammengefaßt werden sollen. Wohlgemerkt, in der Datei werden nur die Beschreibungen der Samples, nicht die Daten an sich abgespeichert!

Entsprechend gestaltet sich das Laden einer Domäne:

```
load_domain(<datei>, <datenklassen>, <samples>).
```

Allerdings besitzen die erstellten Samples natürlich andere Kennungen; sie werden ja temporär vergeben. Deshalb wird zusätzlich eine Liste zurückgegeben, die zu jedem Sample die Datenklasse aufführt.

Schließlich existiert noch eine dritte Funktion, die speziell für das Arbeiten mit Domänen entwickelt wurde: `domain_select/3`. Mit ihr ist es möglich, auf mehreren Samples eine identische, syntaktische Auswahl vorzunehmen. Haben wir

z.B. eine Domäne eingeladen, die Daten zu den Traces `t701` und `t702` umfaßt, so interessiert uns in einem speziellen Fall vielleicht nur Trace `t701`. Normalerweise müßte dafür nacheinander auf allen Samples der Domäne eine Auswahl mit den Constraints `[item(trace, [t701])]` durchgeführt werden—das ist sehr aufwendig. Statt dessen kann man `domain_select` benutzen, womit genau dieser Vorgang automatisiert wird. Der Aufruf ähnelt dem von `synt_select`:

```
domain_select(<samples>, <constraints>, <neue_samples>).
```

Das erste Argument stellt eine Liste der Samples dar, aus denen selektiert werden soll. Zurückgegeben wird ebenfalls eine Liste, und zwar der Samples, die durch die Selektionen entstanden sind. Die Reihenfolge beider Listen korrespondiert miteinander, d.h. das erste Sample der Rückgabeliste wurde mit einem `synt_select` auf dem ersten Sample der Eingabeliste generiert.

Das folgende Beispiel zeigt konkret, wie die drei Routinen angewendet werden können:

```
| ?- synt_select(variant(measurements, 'traces701-740'),
                 [item(trace, [t701,t702])], Sample1),
     synt_select(variant(action_features, 'ba_traces701-740'),
                 [item(trace, [t701,t702])], Sample2).

Sample1 = sid10,
Sample2 = sid11

| ?- save_domain([sid10,sid11], 'TestDomaene').

yes
| ?- load_domain('TestDomaene', Datenklassen, Samples),
     domain_select(Samples, [item(trace, [t701])], SelectSamples).

Datenklassen = [measurements,action_features],
Samples = [sid12,sid13],
SelectSamples = [sid14,sid15]

| ?-
```

Kapitel 5

Technische Realisierung

Im vorherigen Kapitel lag der Schwerpunkt auf der Beschreibung der äußeren Schnittstelle, also wie sich das Data Management Tool nach außen hin präsentiert. Demgegenüber stehen hier die Interna im Mittelpunkt, es geht um die technische Umsetzung des Entwurfs. In Abschnitt 5.1 stelle ich daher zuerst die Programmarchitekur vor, erläutere, welche Module es gibt und wie diese miteinander in Zusammenhang stehen. Der darauf folgende Abschnitt beschäftigt sich dann mit einigen Details der Implementation. Dabei gehe ich näher auf den Mehrbenutzerbetrieb, die Variantenverwaltung und den gekapselten Datenzugriff ein. Abschließend skizziere ich die Möglichkeiten, externe Programme mit dem Data Management Tool zu koppeln.

5.1 Programmarchitektur

Das System zur Datenverwaltung besteht im wesentlichen aus fünf Komponenten, die jede für sich einen bestimmten Aufgabenbereich abdecken. Das Modul dekl.pl umfaßt alle Funktionen, die mit der Bearbeitung von Deklarationsdateien zusammenhängen, varianten.pl hat die Registrierung von Datensätzen und die Verwaltung der dazugehörigen Dateien zur Aufgabe. Davon abgetrennt sind die Routinen, die auf den Inhalt von Dateien zugreifen und Dateien modifizieren. Sie sind in dem Modul zugriff.pl zusammengefaßt, welches für die Bereiche Datenmanipulation und Datenselektion zuständig ist. Auf einer Ebene darunter befindet sich die Funktionseinheit lock.pl, die zur Realisierung des Mehrbenutzerbetriebs notwendig ist und einen einfachen File-Locking-Mechanismus zur Verfügung stellt. Außerdem gibt es noch das Modul benutzer.pl. Es enthält die Routinen, die dem Benutzer die Arbeit mit dem Tool erleichtern, insbesondere geht es um die formatierte Ausgabe von Informationen und die Behandlung von Domänen. Wie die Systemkomponenten miteinander in Beziehung stehen, zeigt Abbildung 5.1. Die Pfeile spiegeln wider, welche Komponenten auf welche Module zugreifen.

Die elementaren Routinen, die die Programmschnittstelle bilden, werden von dekl.pl, varianten.pl und zugriff.pl exportiert. Diese drei Module sind die zen-

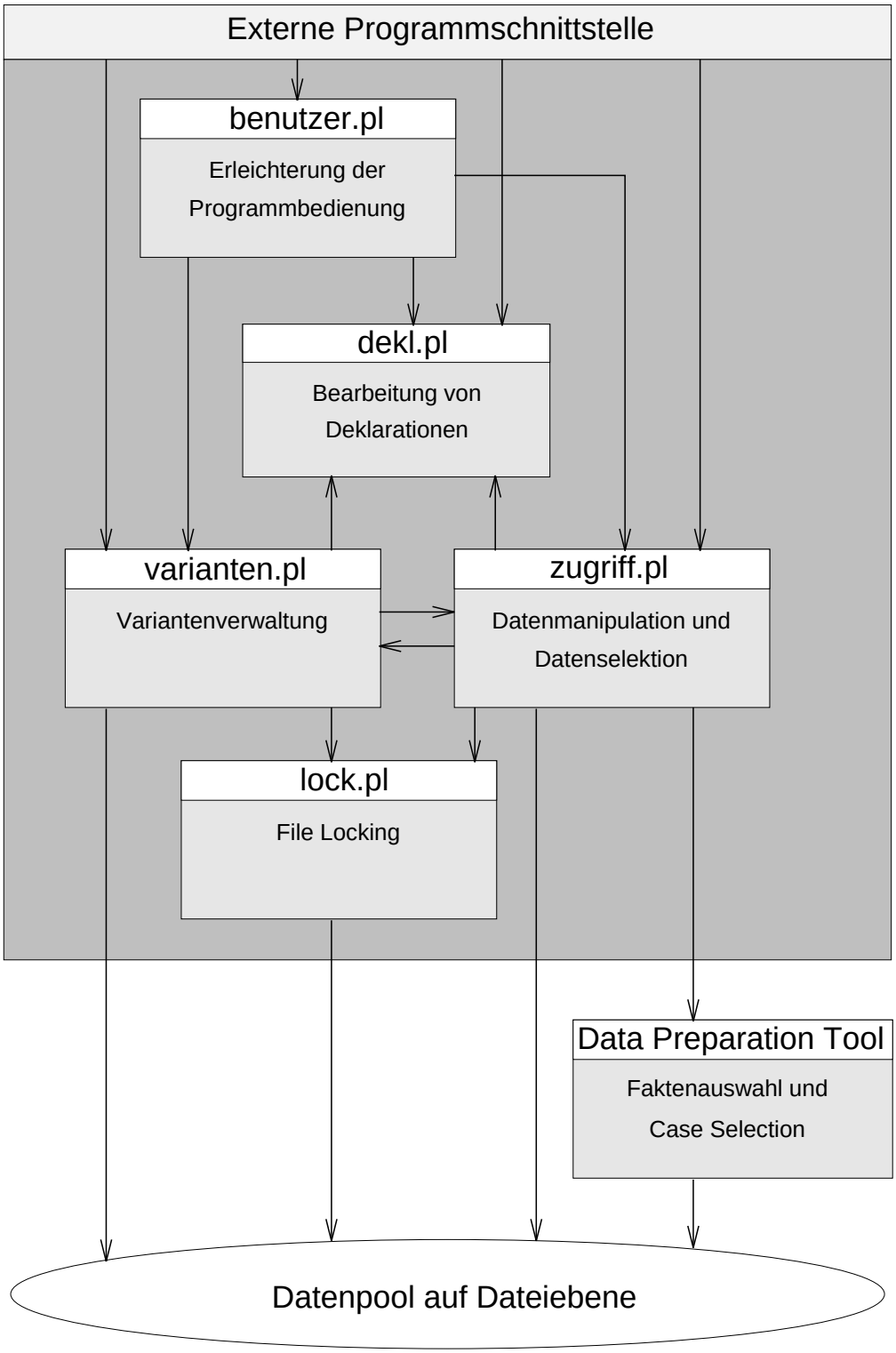


Abbildung 5.1: Architektur des Data Management Tools

tralen Bestandteile des Tools und dementsprechend eng miteinander verzahnt. Die Variantenverwaltung benötigt die Deklarationen, weil sie die interne Verzeichnisstruktur vorgeben, und greift auf zugriff.pl zu, damit Samplekennungen in Beschreibungen von Datenquellen transformiert werden können. Umgekehrt benutzt zugriff.pl auch varianten.pl, z.B. wenn die Dateien einer Variante ermittelt werden müssen. Natürlich sind auch für zugriff.pl die Deklarationen wichtig. Auf diesen drei Komponenten baut benutzer.pl auf, um eine zusätzliche, vereinfachende Funktionalität zu bieten. Das Modul lock.pl hingegen ist auf einer unteren Ebene angesiedelt, denn es wird ausschließlich von varianten.pl und zugriff.pl gebraucht. Schließlich muß noch das Data Preparation Tool genannt werden, auch wenn es nicht Bestandteil des Programms ist. Es wird von zugriff.pl genutzt, um die Auswahl auf Fakten und die Case Selection zu realisieren.

Damit das Ganze nicht zu abstrakt bleibt, möchte ich kurz auf die Schnittstellen der einzelnen Module eingehen:

dekl.pl: Zentrale Funktion dieses Moduls ist `load_declarations`. Daran ist die Routine `dekl_error` gekoppelt, die Auskunft über einen Fehler beim Laden einer Deklarationsdatei gibt. Die eingelesenen Fakten werden von dekl.pl exportiert, so daß sie anderen Modulen zur Verfügung stehen. Im Prinzip handelt es sich also um eine Übertragung der Fakten aus der Deklarationsdatei in die Prolog-Wissensbasis. Wichtigste Aufgabe ist dabei die Überprüfung der Deklarationen auf Korrektheit und Konsistenz (soweit das möglich ist).

varianten.pl: Hierunter sind alle Prozeduren eingeordnet, die in Abschnitt 4.3 unter der Überschrift Variantenverwaltung vorgestellt wurden (Ausnahme: `variant_info`). Genauer gesagt, geht es um die Verwaltung der Dateien, die die Variantendaten beinhalten (Partitionen), und der Informationen über Datensätze. Da das aber abgekapselt ist von der Arbeit *auf* Partitionen, werden intern noch drei weitere Funktionen exportiert. Sie sind sehr eng mit der Implementierung der Datenstrukturen verknüpft und somit speziell auf die Datenhaltung mit Dateien ausgerichtet. Es gibt eine Routine, um eine Partition anzulegen (`create_partition`), eine, mit der der Schlüssel einer Variante ermittelt werden kann (`variant_key`), und schließlich noch eine weitere, die zu einer Variante alle zugehörigen Dateien liefert (`variant_filenames`). Schlägt ein von varianten.pl exportiertes Prädikat fehl, so kann die Fehlerursache über `varianten_error` abgefragt werden.

zugriff.pl: Der Aufgabenbereich dieses Moduls läßt sich am besten als gekapselter Datenzugriff umschreiben. Das Wort Kapselung steht für die Routinen `compose`, `decompose`, `add` und `erase`. Mit Datenzugriff ist vor allem die Funktion `synt_select` gemeint. Damit verbunden sind die Prozeduren, die auf Samples arbeiten: `case_select`, `access_sample`, `sort_sample`, `delete_sample` und `merge_samples`. Des weiteren gibt es noch eine Routine, die mit der Implementierung von `add` und `erase` zusammenhängt: `clean_variant`. Da es sehr zeitaufwendig ist, wenn z.B. das Entfernen

eines Fakts aus einer Variante sofort durchgeführt wird (komplettes Neuschreiben der Datei!), werden die Operationen am Ende der Datei vermerkt. Erst bei einem Zugriff auf die Datei werden die Änderungen tatsächlich vorgenommen. Mit `clean_variant` wird diese Bereinigung explizit erzwungen. Ich habe diese Routine im Hinblick auf zukünftige Erweiterungen des Tools in die Modulschnittstelle aufgenommen. Zum Schluß bleibt zu erwähnen, daß zugriff.pl ähnlich wie varianten.pl eine Prozedur zur Fehlerermittlung zur Verfügung stellt (`zugriff_error`).

lock.pl: Es werden zwei einfache Routinen exportiert, nämlich zum Sperren (`lock_file`) und zum Freigeben (`unlock_file`) einer Datei. Auf diese Art und Weise läßt sich ein exklusiver Dateizugriff realisieren, allerdings nur unter der Voraussetzung, daß alle beteiligten Prozesse diese Routinen auch nutzen. Darauf komme ich noch in Abschnitt 5.2.1 zu sprechen.

benutzer.pl: Unter diesem Modul habe ich alle Funktionen zusammengefaßt, die letztendlich zur Entlastung des Benutzers gedacht und nicht elementar für die Datenverwaltung sind. Darunter fallen die Routinen zur übersichtlichen Ausgabe (`dataclass_info`, `variant_info`, `sample_info`) und ebenfalls die zur Arbeit mit Domänen (`save_domain`, `load_domain` sowie `domain_select`). Schließlich sind auch die Prozeduren `list_sample` und `add_file_to_variant` in diesen Kontext einzuordnen.

5.2 Implementationsdetails

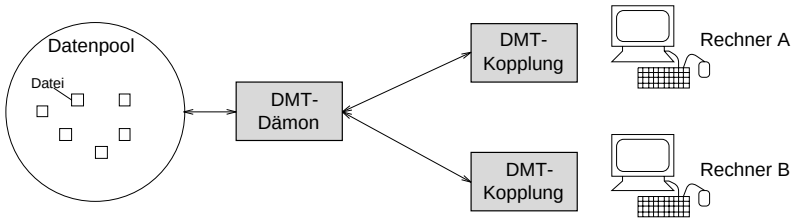
Nachdem die Struktur des Programms dargelegt wurde, möchte ich nun auf einige Probleme bei der Implementierung eingehen und insbesondere die gewählten Lösungen vorstellen. Grundsätzlich bei der Programmentwicklung war die Frage, wie der Mehrbenutzerbetrieb verwirklicht werden kann, vor allem wie Lese- und Schreibkonflikte vermieden werden. Meine Überlegungen dazu stelle ich in Abschnitt 5.2.1 vor. Die konkreten Verwaltungsstrukturen auf Dateiebene sind Gegenstand von Abschnitt 5.2.2, wohingegen Einzelheiten des Datenzugriffs (u.a. die Einbindung des Data Preparation Tools) in Abschnitt 5.2.3 behandelt werden.

5.2.1 Mehrbenutzerbetrieb

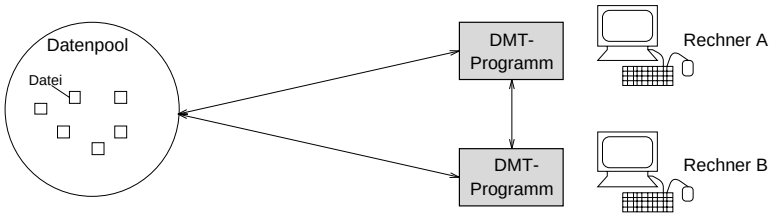
Arbeitet nur ein Benutzer mit dem Data Management Tool, so können bzgl. der Datenzugriffe keine Konflikte auftreten; zu jedem Zeitpunkt wird nur eine Aktion durchgeführt. Anders stellt sich die Situation dar, wenn mehrere Benutzer auf den Daten operieren. Da die Datenhaltung zentral erfolgt, kann es zu zeitkritischen Abläufen (*race conditions*) [Tanenbaum 1994] kommen. Betrachten wir beispielsweise den Fall, daß zwei Benutzer zum gleichen Zeitpunkt eine Variante anlegen wollen—in derselben Datenklasse mit identischen Namen. Das Problem dabei ist, daß beide Benutzer unterschiedliche Varianteneigenschaften spezifizieren. Folglich darf nur ein Aufruf von `new_variant` erfolgreich sein,

der andere muß mit der Fehlermeldung `variant_exists` fehlschlagen. Doch was passiert im unglücklichsten Fall, wenn keine besonderen Vorkehrungen getroffen werden? Der Prozeß des einen Benutzers schaut nach, ob die Variante existiert, und legt sie dann an. Verläuft dieser Vorgang bei dem Prozeß des anderen Benutzers genau parallel, so führt das zu Inkonsistenzen in der Datenhaltung. Schwierigkeiten gibt es auch, wenn mehrere Anwender auf derselben Variante Schreiboperationen durchführen. Solche Konflikte sind zu vermeiden, bestimmte Aktionen müßen unter wechselseitigem Ausschluß stattfinden.

Lösung 1: Das Data Management Tool als Dämon-Prozeß



Lösung 2: Nachrichtenaustausch per TCP/IP



Lösung 3: File Locking

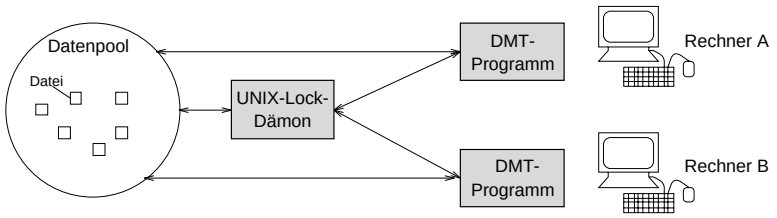


Abbildung 5.2: Realisierungsmöglichkeiten des wechselseitigen Ausschlusses

Zu diesem Thema sind zahlreiche Verfahren entwickelt worden. Hier interessiert jedoch vor allem, wie ein wechselseitiger Ausschluß unter UNIX realisiert werden kann. Dabei ist als Randbedingung zu beachten, daß es sich um ein verteiltes System handelt; die Rechner haben keinen gemeinsamen Hauptspeicher. Die wohl eleganteste Lösung ist ein sogenannter Dämon-Prozeß [Gulbins 1988], der zyklisch im Hintergrund abläuft und von allen Teilnehmern genutzt wird. Allein dieser Dämon-Prozeß arbeitet auf den Daten, so daß zu einem Zeitpunkt immer

nur eine Anweisung abgearbeitet wird. Die Anwender kommunizieren auf ihren Rechnern mit „abgespeckten“ Programmen, die zwar die Funktionalität des Data Management Tools anbieten, doch die eigentlichen Aktionen nicht selbst ausführen, sondern an den Dämon weiterleiten. Eine andere Möglichkeit besteht darin, über TCP/IP Nachrichten zwischen den beteiligten Prozessen auszutauschen. Das Data Management Tool, das auf einem Rechner läuft, würde sich dann von allen anderen anderen gestarteten DMT-Programmen die Erlaubnis holen, ein Kommando durchzuführen. In diesem Fall aber am einfachsten zu implementieren, ist eine dritte Lösung: File Locking. UNIX stellt u.a. mehrere atomare Routinen zur Verfügung, mit denen Dateien gesperrt und wieder freigegeben werden können. Da der gesamte Datenpool sich aus Dateien zusammensetzt, kann so mit wenig Aufwand der wechselseitige Ausschluß verwirklicht werden. Die anderen beiden Ansätze sind im Vergleich dazu zu aufwendig, als daß sie im Rahmen meiner Arbeit eingesetzt werden könnten. In Abbildung 5.2 habe ich diese drei Lösungen noch einmal gegenübergestellt.

Das Modul lock.pl exportiert zwei Prozeduren, die den File-Locking-Mechanismus von UNIX in Prolog nutzbar machen: `lock_file/1` und `unlock_file/1`. Beim Sperren einer Datei mittels `lock_file` wird zunächst geprüft, ob die Datei bereits von einem anderen Prozeß belegt ist. In diesem Fall wird aktiv gewartet, bis die Sperre über `unlock_file` aufgehoben wird[1]. Sperren stellen demnach eine einfache, aber effektive Methode dar, wie die beteiligten Prozeße aufeinander abgestimmt werden können—soweit das Prinzip. Wie das File Locking konkret eingesetzt wird, bespreche ich weitergehend in den folgenden zwei Abschnitten.

5.2.2 Variantenverwaltung

Unter dem Stichwort Variantenverwaltung möchte ich die Arbeitsweise des Moduls varianten.pl erläutern, wobei die externen Speicherstrukturen im Mittelpunkt stehen. Es wurde bereits angesprochen, daß sich der Datenpool unter einem Verzeichnis befindet, das in der Deklarationsdatei spezifiziert wurde (`directory(...)`). Dieses Verzeichnis enthält mehrere Unterverzeichnisse, die jeweils die Daten einer Datenklasse aufnehmen; das sind im einzelnen die Partitionen und die Variantentabelle. In der Variantentabelle, die in der Datei mit dem Namen `varianten` gehalten wird, sind alle Informationen zu den Varianten einer Datenklasse aufgeführt. Sie liefert folglich Auskunft darüber, welche Varianten es gibt, welche Eigenschaften sie haben und aus welchen Partitionen sie bestehen. Die Partitionen umfassen (einen Teil der) Fakten bzw. Regeln einer Variante und stellen jeweils eine Datei dar. Schon aus dem Dateinamen ist ersichtlich, um welche Partition es sich handelt: er setzt sich zusammen aus dem Variantennamen und dem Schlüsselwert. Z.B. heißt die Partition der Variante `traces701-740`, die die Fakten zum Trace `t701` beinhaltet, `traces701-740[t701]`.

Da Dateizugriffe sehr zeitintensiv sind, lädt das Data Management Tool die

[1] Ich setze aktives Warten ein, weil das Data Management Tool interaktiv ist. So kann der Benutzer die Ausführung einer auf die Aufhebung einer Sperre wartenden Routine abbrechen, ohne daß der gesamte Prolog-Prozeß entfernt werden muß.

Variantentabellen der deklarierten Datenklassen in den Hauptspeicher. Bei der Durchführung einer Aktion (`delete_variant`, `variant_properties`, ...) wird dann auf den internen Variantentabellen gearbeitet. Ggf. müssen allerdings die extern abgelegten Tabellen aktualisiert werden, z.B. wenn eine Variante gelöscht wurde. Somit stimmen interne und externe Variantentabellen bzgl. des Inhalts stets überein. Ein Problem dabei ist aber, daß alle in Ausführung befindlichen DMT-Programme zu jedem Zeitpunkt den gleichen Informationsstand haben müssen. Wenn also ein Benutzer eine Variante anlegt, müssen alle anderen Benutzer davon Kenntnis erlangen. D.h. nicht nur externe Variantentabellen sind an die internen anzupassen, sondern auch umgekehrt. Aus diesem Grunde gibt es unter dem Hauptverzeichnis eine Datei `datenklassen`. Sie beinhaltet die Datenklassentabelle, die zu jeder externen Variantentabelle den letzten Änderungszeitpunkt aufführt. Mit Hilfe dieser Informationen kann dann entschieden werden, ob die internen Variantentabellen sich auf dem neuesten Stand befinden. Dazu muß sich das DMT-Programm nur merken, wann die Variantentabellen zum letzten Mal aufeinander abgestimmt wurden. Ändert auf der anderen Seite eine Instanz des Data Management Tools eine Variantentabelle, so muß natürlich auch die Datenklassentabelle aktualisiert werden. Nur so kann gewährleistet bleiben, daß eine einheitliche Sicht auf den Datenpool vorliegt.

Die Konflikte, die der Mehrbenutzerbetrieb nach sich zieht, sind dadurch aber noch nicht vollständig gelöst. Greifen wir nochmal den in Abschnitt 5.2.1 vorgestellten Fall auf, daß zwei Benutzer zum gleichen Zeitpunkt dieselbe Variante (mit unterschiedlichen Eigenschaften) erzeugen wollen. Der eine Benutzer arbeite auf Rechner A, der andere auf Rechner B. Die DMT-Programme werden zunächst über ihre internen Variantentabellen prüfen, ob die Variante schon existiert. Verläuft der Test negativ, werden die internen Tabellen geändert und auch die externe Tabelle aktualisiert. Dabei kann es aber zu Schreibkonflikten kommen, die die externe Tabelle in einen inkonsistenten Zustand überführen. Zudem muß ja garantiert werden, daß nur die interne Variantentabelle eines Rechners geschrieben wird, denn nur ein `new_variant`-Aufruf darf erfolgreich sein. Wie ich bereits erwähnt habe, müssen solche Aktionen unter wechselseitigem Ausschluß stattfinden. Damit meine ich die Aktionen, die Änderungen in einer Variantentabelle nach sich ziehen: `new_variant`, `delete_variant`, `change_variant_properties` und `create_partition`.

Die Lösung des Problems ist recht einfach: Bevor ein DMT-Programm eine Aktion ausführt, sperrt es die Datei `datenklassen.lock`[2]. Ist sie bereits von einem anderen Prozeß gesperrt, wird auf die Freigabe gewartet. Dann erfolgt zunächst eine Aktualisierung der internen Variantentabellen, je nachdem, ob sich in der Zwischenzeit etwas geändert hat. Bei Aktionen, die keine Modifikationen vornehmen, wird `datenklassen.lock` anschließend wieder freigegeben. Andernfalls bleibt sie so lange gesperrt, bis die Aktion abgeschlossen ist. Diese Vorgehensweise läßt sich exemplarisch an dem vorgestellten Szenario verdeutlichen (Abbildung 5.3). Auf Rechner A und Rechner B wird zum selben Zeitpunkt

[2]Eigentlich sollte die Datei `datenklassen` gesperrt werden, doch aufgrund der Beschaffenheit der UNIX-Lock-Routine kann die Datei dann unter Prolog nicht verändert werden. Daher wird statt dessen die leere Datei `datenklassen.lock` zum Sperren benutzt.

der Befehl

```
new_variant(measurements, test2, ...)
```

gegeben. Rechner A ist ein bißchen schneller als B und kann erfolgreich die Sperre auf `datenklassen.lock` setzen (Punkt 1). Rechner B hingegen muß warten, bis die Aktion von A durchgeführt ist (Punkt 2). Erst dann hat B Zugang zu der Datenklassentabelle (Punkt 6), die auch sofort überprüft wird (Punkt 7). Da sich die Variantentabelle zu der Datenklasse `measurements` inzwischen aber geändert hat, müssen die internen Informationen von B aktualisiert werden (Punkt 8). Jetzt kann die Aktion ausgeführt werden, doch weil die Variante `test2` nun schon existiert, schlägt `new_variant` mit der Meldung `variant_exists` fehl.

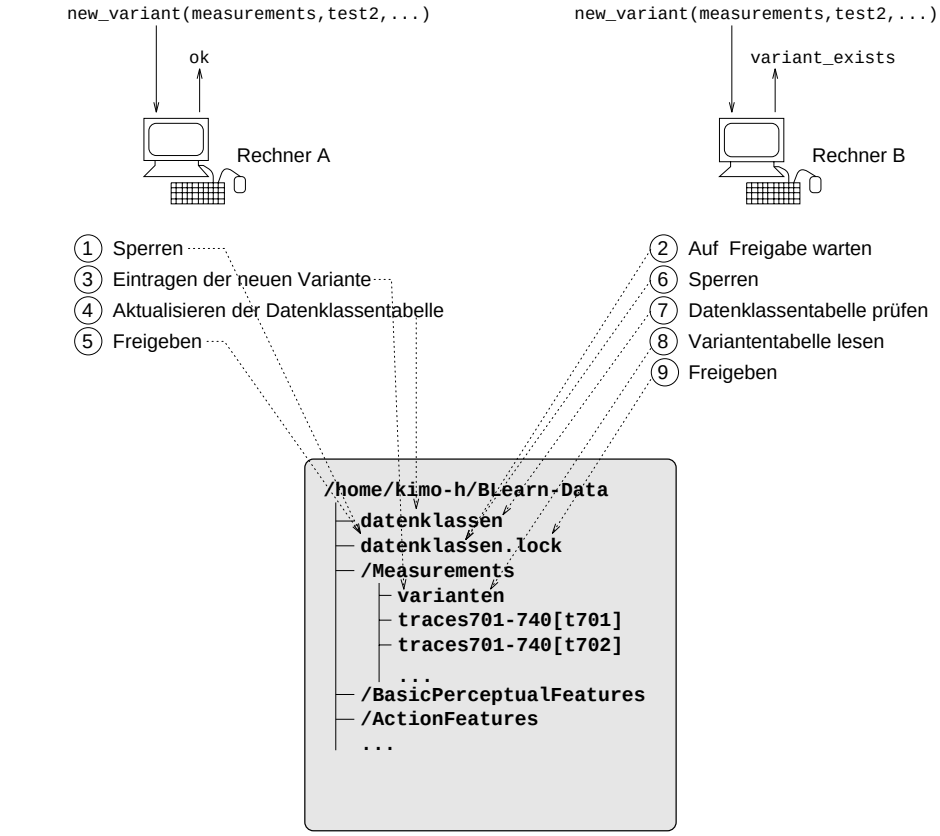


Abbildung 5.3: Behandlung zweier konkurrierender Anweisungen

5.2.3 Gekapselter Datenzugriff

Der gekapselte Datenzugriff wird durch das Modul zugriff.pl unter Zuhilfenahme des Data Preparation Tools realisiert. Während varianten.pl die Daten, insbe-

sondere die Partitionen, verwaltet, arbeitet zugriff.pl auf ihnen. Dabei muß man Datenmanipulation und Datenselektion unterscheiden.

Modifikation von Varianten

Mit `add` und `erase` können Varianten und Samples verändert werden. Da Samples intern als Listen gespeichert sind, gestaltet sich bei ihnen das Hinzufügen und Entfernen von Elementen sehr einfach—dafür stellt Quintus-Prolog Listenoperationen zur Verfügung. Demgegenüber ist die Modifikation von Varianten komplizierter. Als erstes müssen die Elemente, die beim Aufruf von `add` oder `erase` übergeben wurden, den jeweiligen Partitionen zugeordnet werden. Im zweiten Schritt kann dann die Operation auf den einzelnen Partitionen durchgeführt werden. Hierbei gibt es jedoch ein Zeitproblem, das mit der Verwendung von Dateien zusammenhängt. Soll beispielsweise ein Fakt aus einer Datei (Partition) entfernt werden, so muß sie schlimmstenfalls komplett neu geschrieben werden. Und selbst im günstigsten Fall ist die Datei einmal zu durchsuchen; schließlich muß geprüft werden, ob es das Fakt überhaupt gibt. Ähnlich sieht die Situation bei einem Aufruf von `add` aus. Daß eine zu verändernde Partition überarbeitet und evtl. neu geschrieben werden muß, läßt sich natürlich nicht vermeiden. Die Frage ist nur, wann das geschieht. Kritisch wird es nämlich, wenn mehrere Modifikationen direkt hintereinander getätigt werden. Dann werden u.U. Dateien mehrfach überarbeitet, was sehr viel Zeit kostet. Geschickter ist es, die Veränderungen zu jeder Partition zu sammeln und erst bei Bedarf die tatsächliche Überarbeitung vorzunehmen.

So macht es auch zugriff.pl: Fakten und Regeln werden zunächst ans Ende der entsprechenden Partition geschrieben (das geht verhältnismäßig schnell). Sie sind jeweils von zwei Kommentarzeilen eingeschlossen, die Auskunft darüber geben, ob das Element hinzugefügt oder entfernt werden soll. Bis dahin hat keinerlei Überprüfung stattgefunden. Die vermerkten Operationen werden erst durch einen Aufruf von `clean_variant` wirklich ausgeführt. Dabei testet die Routine alle Partitionen der Variante daraufhin, ob sie von Modifikationen betroffen sind. Anschließend werden die ensprechenden Dateien bereinigt und auf den aktuellen Stand gebracht. Da dieser Vorgang jedoch vor dem Benutzer verborgen werden sollte, wird `clean_variant` bei einer syntaktischen Auswahl auf einer Variante automatisch aufgerufen.

In diesem Kontext muß noch erwähnt werden, daß Operationen auf Dateien immer unter Benutzung des File Lockings erfolgen. Nur so kann ein reibungsloses Arbeiten auf dem Datenpool ohne die bekannten Schreib-/Lesekonflikte garantiert werden.

Einbindung des Data Preparation Tools

Im folgenden möchte ich noch auf die Einbindung des Data Preparation Tools eingehen, was hauptsächlich die syntaktische Auswahl und die Case Selection betrifft. Betrachten wir zuerst die Routine `synt_select` genauer. Soll aus einem Fakten-Sample ein weiteres Sample gezogen werden, so besteht die Aufgabe

von zugriff.pl darin, die Constraints vom DMT-Format in das DPT-Format zu konvertieren. D.h. die Argumenttypen sind durch die Argumentpositionen zu ersetzen, was auf der Grundlage der Prädikatsdeklarationen geschieht. Da das Data Management Tool aber keine besonderen Bedingungen an diese Deklarationen stellt (wie es z.B. bei den Abstraction Levels von Anke Rieger der Fall ist, daß alle Prädikate die gleiche Argumentstruktur und die gleiche Stelligkeit aufzuweisen haben), müssen für jedes Prädikat die Constraints einzeln transformiert werden. Mit anderen Worten: Für jedes in Frage kommende Prädikat sind die DPT-Constraints mit den Argumentpositionen gesondert anzugeben. Dementsprechend wird eine spezielle Version der `select`-Anweisung des Data Preparation Tools genutzt:

```
select(<source>, <constraints>, <sample>).
```

Sie unterscheidet sich von der in Kapitel 2 vorgestellten Version durch den Wegfall des zweiten Arguments (Abstraction Level) und durch ein verändertes Format der Constraints. Die Constraints werden nämlich hier durch eine Menge von Listen repräsentiert, die zu jedem Prädikat die erlaubten Werte spezifiziert (siehe auch Abbildung 5.4).

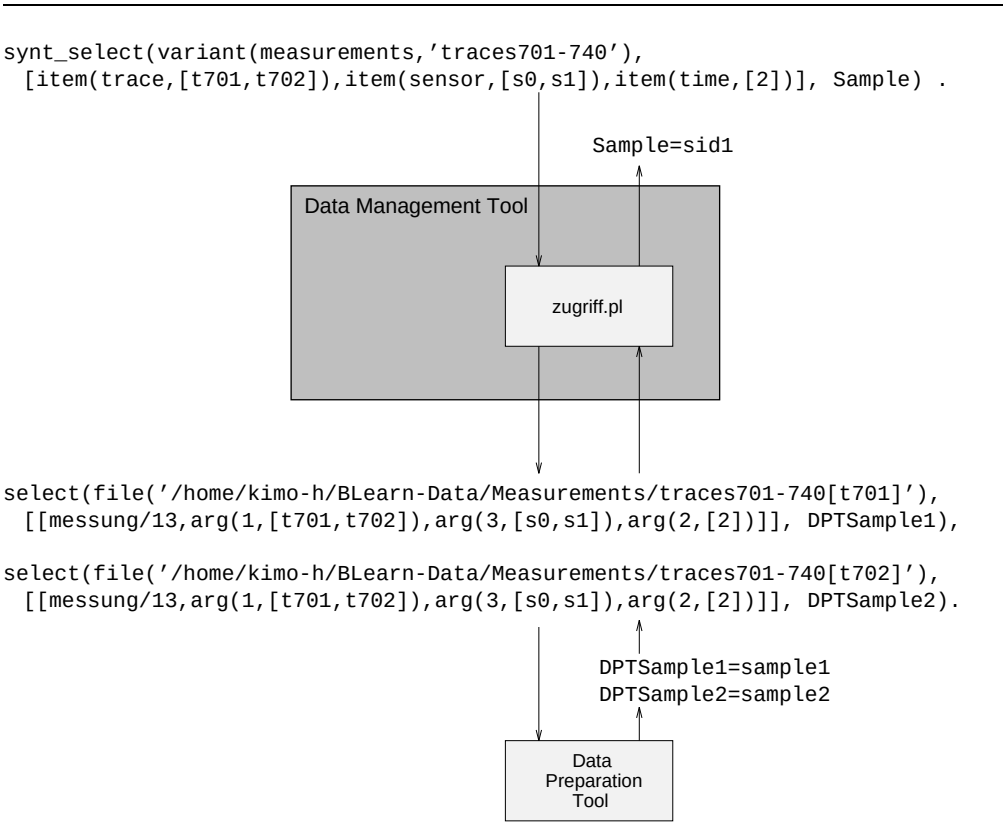


Abbildung 5.4: Umsetzung von `synt_select`- in `select`-Anweisungen

Bei der Fakten-Auswahl auf einer Fakten-Variante kommt zu der Constraints-Konvertierung noch hinzu, daß die zu betrachtenden Partitionen ermittelt werden müssen. Auf jeder Partition ist dann ein `select` durchzuführen. Das wird beispielhaft in Abbildung 5.4 illustriert. Der Benutzer möchte aus der Variante `traces701-740` der Sensordistanzmessungen bestimmte Fakten zu den Traces `t701` und `t702` auswählen. Das Data Management Tool generiert aus den Argumenten des `synt_select`-Aufrufs zwei `select`-Anweisungen—weil eben zwei Partition durchsucht werden müssen. Das Data Preparation Tool erstellt daraufhin zwei Samples, die intern von zugriff.pl zu einem zusammengefaßt werden (die Daten werden aber nicht doppelt im Hauptspeicher gehalten!)

Bei der Case Selection ist es änhlich wie bei der syntaktischen Auswahl: Die Argumenttypen müssen in Argumentpositionen überführt werden. Hier sind aber wiederum keine allgemeinen Aussagen über mehrere Prädikate möglich, wie das bei den Abstraction Levels des Data Preparation Tools der Fall ist. Argumentpositionen sind an Prädikate gebunden, so daß die DMT-Relationenliste in eine Menge von DPT-Relationenlisten transformiert werden muß. D.h. für jede Kombination aus einem Prädikat des Beispiel-Samples und einem des Hintergrundwissen-Samples sind die Relationen explizit anzugeben. Dafür wird wieder eine flexiblere Aufrufmöglichkeit der Case Selection genutzt, die das Data Preparation Tool zusätzlich anbietet.

5.3 Anbindung externer Programme

Zum Schluß möchte ich mich kurz der Frage widmen, wie Programme zur Datengenerierung und Lernverfahren mit dem Data Management Tool gekoppelt werden können. Der Benutzer muß für diese Programme die Eingabedaten zusammenstellen sowie später die Ausgabedaten in den Datenpool einspeisen. Ich sehe prinzipiell drei Möglichkeiten, wie das bewerkstelligt werden kann.

Die erste Möglichkeit besteht darin, daß die Programme vollständig von der Datenverwaltung getrennt sind. In diesem Fall übernimmt der Benutzer die gesamte Arbeit. Er wählt die erforderlichen Daten aus dem Datenpool aus und bereitet sie für das Programm auf. In der Regel heißt das, daß er sie in einer Datei ablegt (über `list_sample/2`). Anschließend startet er das Programm und läßt sich so aus den Eingabedaten neue Daten berechnen. Die Ausgabedaten müssen wiederum in einer Variante abgelegt werden. Sind sie in einer Datei gespeichert, so kann das mit `add_file_to_variant/3` geschehen. Diese Methode kann einfach durchgeführt werden, wenn die Programme datei-orientiert arbeiten, also die Eingabe aus einer Datei lesen und die Ausgabe in eine Datei schreiben. Das ist in der BLearn-Anwendung am häufigsten der Fall. Der Vorteil dabei ist natürlich, daß keinerlei Anpassungen vorgenommen werden müssen. Allerdings bleibt die Arbeit bei dem Benutzer hängen.

Optimal aus der Sicht des Anwenders wäre ein Programmaufruf, bei dem die Eingabe über Sample-Kennungen spezifiziert wird und die Ausgabeeinheit entsprechend über den Namen einer Variante. Das Programm würde sich dann aus den Samples die Daten holen, die neuen Daten daraus berechnen und sie

direkt in die angegebene Variante schreiben. Die zweite Möglichkeit sähe also vor, daß das Programm komplett überarbeitet wird und direkt die Routinen des Data Management Tools nutzt. So kann das Programm entsprechend von allen Funktionen des Tools Gebrauch machen.

Diese Lösung ist aber in der BLearn-Anwendung problematisch, weil einige Programme sowohl in der Lernphase und als auch in der Performanzphase eingesetzt werden. Die Datenverwaltung ist ja ausdrücklich für das Lernen konzipiert, in der Anwendung geht es hingegen um effiziente Verarbeitung der Daten. Deswegen sollten die Programme nicht direkt mit dem Data Management Tool kommunizieren, sondern eigenständig bleiben. Trotzdem könnte durch ein Rahmenprogramm die benutzerfreundliche Schnittstelle wie oben realisiert werden. Dieses Rahmenprogramm übernimmt dann die Funktion des Benutzers in der ersten Lösung. Die Daten werden aufbereitet, das eigentliche Programm ausgeführt und anschließend die Ausgabe in eine Variante geschrieben. Das verbindet Flexibilität mit Benutzerfreundlichkeit. Natürlich ist mit der Programmierung dieser Rahmenprogramme ein zusätzlicher Aufwand verbunden. Im Hinblick auf eine Erweiterung des Data Management Tools wäre ein allgemein einsetzbares Rahmenprogramm denkbar, womit externe Programme mit dem Datenverwaltungssystem gekoppelt werden können.

Kapitel 6

Diskussion

In diesem letzten Kapitel geht es um eine nüchterne Betrachtung der Arbeitsergebnisse. Ich möchte mich kritisch damit auseinandersetzen, inwiefern das Data Management Tool den gesteckten Zielen gerecht wird, wo es verbessert werden könnte und wie es im Vergleich zu anderen Systemen abschneidet. Diese Fragen werden in Abschnitt 6.1 behandelt. Welche zukünftigen Arbeiten denkbar sind, stelle ich in Abschnitt 6.2 dar.

6.1 Kritische Bewertung der Arbeit

Um es noch einmal zusammenzufassen: Ziel dieser Arbeit war die Entwicklung eines Datenverwaltungssystems, das exemplarisch in der BLearn-Domäne einzusetzen ist. Das grundlegende Motiv dahinter lag in der Besonderheit der Lernaufgabe, die es notwendig macht, mit verschiedenen Versionen von Daten zu arbeiten. Bestehende Lernumgebungen unterstützen das nur unzureichend oder gar nicht.

Nachdem ich die Anforderungen an die zentrale Datenverwaltung spezifiziert hatte, leitete ich daraus ein umfassendes Konzept ab. Auf der Grundlage dieses Konzepts entstand schließlich das Data Management Tool. Nun gilt es zu bewerten, ob das Tool auch wirklich das leistet, was es soll.

6.1.1 Erfüllung der Anforderungen

Ein Softwaresystem dieser Art muß sich an den Zielen messen, die in der Anforderungsspezifikation formuliert wurden. Im Gegensatz zu Lernverfahren, Optimierungsmethoden, o.ä. gibt es jedoch keine klare Bewertungsfunktion, nach der die Güte der gewählten Lösung bestimmt werden kann. Vielmehr gilt es zu prüfen, ob die Problematik, die Ausgangspunkt für die Entwicklung war, durch das Programm gelöst wird. Das möchte ich im einzelnen an den in Kapitel 2 aufgestellten Design-Zielen belegen:

Strukturierung des Datenpools

Mit dem DMT ist es möglich, den Datenpool in eine Menge von Datenklassen aufzuteilen, so daß die einzelnen Ebenen der Abstraktionshierarchie getrennt werden können. Das erlaubt dem Benutzer eine übersichtliche Gliederung und eine gezielte Einschränkung auf bestimmte Datenbereiche. Das wurde ja auch schon vor der Benutzung des DMT verfolgt, indem die Dateien in einer nach Kategorien geordneten Verzeichnisstruktur abgelegt wurden. Die Einführung der Datenklassen resultiert im Prinzip direkt aus diesen Erfahrungen. Darüber hinaus gewährleistet das DMT aber, daß der Datenpool strukturiert *bleibt* und daß die Struktur immer gleich ist. Dies ist eine wichtige Voraussetzung dafür, daß mehrere Benutzer auf einem zentralen Datenbestand können.

Alternative Datensätze

Das Variantenkonzept erlaubt dem Benutzer, Daten zu Datensätzen zusammenzufassen und zu jedem Datensatz weitere Informationen (Benutzername, Kommentar, etc.) zu speichern. Die Notwendigkeit, alternative Dateneinheiten zu bilden, liegt in der Besonderheit der Lernaufgabe begründet. Dementsprechend sind die Varianten der zentrale Aspekt des DMT. Doch auch Dateien können als Datensätze betrachtet werden. Was das DMT außerdem bietet und womit es den Benutzer hauptsächlich entlastet, ist die Verwaltung der Zusatzinformationen zu jeder Variante. So wird das Dateien-Chaos vermieden, weil jeder Benutzer die Datensätze eindeutig identifizieren kann. Das war ja gerade ein Ausgangsproblem, daß die Benutzer über den Inhalt von Dateien vielfach nicht oder nicht mehr Bescheid wußten. Dieses Problem ist damit behoben.

Effiziente Speicherung

Das DMT ist in der Lage, doppelte Datensätze anhand ihrer Inhaltsbeschreibung zu erkennen, es kann aber nicht verhindern, daß mehrere Datensätze existieren, die die gleichen Daten beinhalten. Das habe ich bereits in Kapitel 3 diskutiert. Ich meine aber, daß diese Lösung der Aufgabenstellung gerecht wird, wenn man vom ursprünglichen Problem ausgeht, daß identische Daten häufig mehrfach erzeugt und gespeichert wurden. Das wird nämlich durch das DMT verhindert. Des weiteren unterstützt es den Benutzer, wenn er feststellen will, ob bestimmte Daten schon existieren (`variant_exists`).

In Kapitel 3 habe ich auch begründet, warum Datensätze extern als Einheiten gespeichert werden müssen. Vor diesem Hintergrund ist die Speicherplatzausnutzung des DMT sehr gut. Denn sind zwei Datensätze auf verschiedene Art und Weise generiert worden, zufällig aber inhaltlich identisch, so handelt es sich trotzdem um zwei eigenständige Datensätze und nicht um die gleichen. Natürlich gäbe es auf unterster Ebene die Möglichkeit, inhaltlich gleiche Datensätze nur einmal abzulegen, doch das wäre programmtechnisch aufwendig und rechenintensiv.

Effiziente Speicherung heißt aber auch schneller Datenzugriff. Mit der Möglichkeit, Variantendaten über Partitionierungsschlüssel zu indexieren, bietet das DMT einen wirkungsvollen Mechanismus zur Beschleunigung des Datenzugriffs an. Das ist vor allem ein großer Vorteil, wenn man berücksichtigt, daß die Daten sequentiell und unstrukturiert in Dateien abgelegt sind. Allerdings muß diese Aussage vor dem Hintergrund gesehen werden, daß die verwendete Speicherungsform im Prinzip die im zeitlichen Rahmen einzig realisierbare war. Denn an sich ist hier durchaus Kritik angebracht. Bei syntaktischen Auswahlen müssen nämlich stets ganze Dateien gelesen werden. Mittels der Aufteilung in Partitionen wird die Menge der zu untersuchenden Daten zwar reduziert, doch wie effektiv diese Einschränkung ist, hängt von den Auswahlkriterien und dem Partitionierungsschlüssel ab. Kann aus den Constraints einer Auswahl nicht auf die Schlüsselwerte der entsprechenden Partitionen geschlossen werden, müssen alle Dateien einer Variante gelesen werden. Geschickter wäre es, die Daten in speziellen Datenstrukturen zu speichern, die für das gezielte Wiederauffinden von Elementen konzipiert sind. Z.B. könnte man bei Fakten ein mehrdimensionales Hashing einsetzen, wobei sich das Gridfile [Ottmann und Widmayer 1990] anbieten würde. Natürlich müßte vorher eine Aufwandsabschätzung vorgenommen werden. Daß ich keine geeignetere Datenstruktur verwendet habe, hängt u.a. auch mit der Benutzung des Data Preparation Tools zusammen. Hätte ich die vom DPT zur Verfügung gestellte Funktionalität (Faktenauswahl, Case Selection) selbst implementieren müssen, so wäre das weit über den Rahmen einer Diplomarbeit hinausgegangen. Nicht zuletzt hätte eine erfolgsversprechende Alternative in der Verwendung einer Datenbank gelegen. In Kapitel 2 habe ich dargelegt, warum auch diese Möglichkeit nicht in Frage kam. So gesehen ist die eingesetzte Speicherungsart bzgl. der Zugriffszeiten kaum optimal, was jedoch das in diesem Kontext praktisch Durchführbare anbetrifft, die sinnvollste Methode.

Übersichtlichkeit

Der Benutzer hat mit dem DMT die Möglichkeit, sich Datensätze mit ihren Zusatzinformationen auflisten zu lassen. Somit kann er sich jederzeit einen Überblick über die vorhandenen Daten verschaffen. Dabei ist u.a. auch das angesprochen, was bereits unter dem Punkt „Alternative Datensätze“ hervorgehoben wurde: Es sind die Zusatzinformationen zu den Varianten, die dem Benutzer den Überblick garantieren. Mit Übersichtlichkeit ist aber auch gemeint, daß sich der Benutzer ganz bestimmte Datensätze anzeigen lassen kann (die er z.B. vorher über eine Suche ermittelt hat). Das DMT unterstützt den Anwender also dabei, sich über den Inhalt des Datenpools zu orientieren. Vorher konnte sich der Benutzer nur an Verzeichnissen und Dateinamen orientieren.

Datensuche

Das Suchen von Datensätzen ist eine elementare Funktion des DMT zur Unterstützung des Anwenders. Er kann die Suche nach verschiedenen Kriterien vor-

nehmen, und gerade die freie Nutzung der Datensatzattribute bietet vielfältige Möglichkeiten dabei. Je nachdem wie detailliert die Attribute der Datensätze spezifiziert sind, kann sehr gezielt gesucht werden. Das ist etwas, was vorher überhaupt nicht möglich war. Einzig und allein die Dateinamen konnten als Suchkriterium genutzt werden.

Revidierbarkeit

Mit dem DMT können Datensätze erzeugt und auch gelöscht werden; somit ist das Ziel der Revidierbarkeit erreicht. Allerdings ist in diesem Zusammenhang eine Erweiterung des DMT naheliegend: Das System könnte den Benutzer warnen, wenn er eine Variante löschen will, auf die Referenzen bestehen. Die Problematik dabei habe ich in Kapitel 4 angesprochen. Verzichtet man jedoch auf die Forderung, daß der Benutzer erfährt, welche Varianten auf eine zu entfernende verweisen, so gibt es eine Lösung dafür. Sie könnte so aussehen, daß zu jeder Variante ein Zähler gehalten wird, der die Anzahl der Bezüge auf diese Variante speichert; diese Zähler müssen natürlich laufend aktualisiert werden. Eine Variante wird folglich nur dann ohne Nachfrage gelöscht, wenn der entsprechende Zähler den Wert Null hat.

Datenkapselung

Die Datenkapselung war ein sehr wichtiges Design-Ziel, das das DMT im wesentlichen erfüllt. Der Benutzer oder die Programme manipulieren Varianten über bestimmte Routinen (`add`, `erase`, `add_file_to_variant`), ohne die Implementationsdetails der Datenspeicherung zu kennen. Folglich gewährleistet das DMT die so wichtige Unabhängigkeit von den verwendeten Speicherungsmethoden. Theoretisch könnten die Daten auch in einer Datenbank gehalten werden. Für den Anwender ändert sich jedoch dadurch nichts, er benutzt weiterhin dieselben DMT-Funktionen.

Eine vollständige Abstraktion von der Speicherungsmethode konnte aber aus Effizienzgründen nicht erreicht werden. In Kapitel 3 habe ich ausführlich belegt, warum die Einführung von Partitionen notwendig ist. Dadurch, daß in der Deklarationsdatei Partitionierungsschlüssel anzugeben sind, wird im Prinzip etwas nach außen getragen, was mit der Speicherung in Dateien zusammenhängt. Andererseits ist aber nur der Benutzer in der Lage, die Partitionierungsschlüssel festzulegen; das DMT kann sie nicht intern bestimmen. Aus diesem Grund und weil die Zugriffszeiten ohne die Partitionierung von Varianten vollkommen inakzeptabel sind, halte ich ein Aufweichen der vollständigen Trennung zwischen Programmschnittstelle und Implementierung in diesem Fall für zulässig und notwendig. Schließlich bleiben dem Benutzer trotzdem die Einzelheiten der konkreten Speicherung verborgen, und eine Umstellung der Datenhaltung ist weiterhin problemlos möglich. Die Partitionierungsschlüssel sind ja zusätzliche Informationen, die genutzt werden können, aber nicht müssen.

Konsistenzprüfung

Das DMT stellt sicher, daß Varianten und Samples im Sinne von Mengen behandelt werden. Das betrifft das Hinzufügen und Entfernen von Elementen. Somit wird der Forderung entsprochen, daß in Datensätzen keine doppelten Elemente vorkommen dürfen, was vor allem bei zeitorientierten Daten wichtig ist.

Datenabstraktion

Über die Funktionen `compose` und `decompose` können Fakten und Regeln gemäß ihrer Bestandteile zusammengesetzt und zerlegt werden. Bei Fakten wird darüber hinaus von den Argumentpositionen abstrahiert, indem der Zugriff über Argumenttypen erfolgt, die in einer externen Deklarationsdatei definiert sind. Damit entspricht das DMT einer wesentlichen Forderung, die von System- und Benutzerseite gestellt wurde. Der Datenaustausch zwischen Modulen des Lernsystems wird von den Repräsentationsdetails entkoppelt und bleibt folglich flexibel. Für den Benutzer stellt die Verwendung von Argumenttypen eine erhebliche Erleichterung dar.

Datenauswahl

Die Selektion von Daten ist ein zentraler Aspekt bei der Vermeidung von Speicherplatzproblemen im Hauptspeicher. Das DMT erlaubt es, Ausschnitte des Datenpools in den Hauptspeicher zu laden und erneut aus Ausschnitten zu selektieren. Dafür stellt es die Routine `synt_select` zur Verfügung, die nach syntaktischen Kriterien auswählt. Diese Funktion genügt den in Kapitel 2 formulierten Forderungen. Zwar sind auch andere Auswahlverfahren (z.B. statistische) denkbar, doch wie ich bereits sagte, war ich auf die Benutzung des Data Preparation Tools angewiesen. Des weiteren sind Constraints mit Relationen (größer, kleiner, etc.) eine mögliche Erweiterung. Das Grundproblem ist aber gelöst, nämlich nur diejenigen Daten in den Hauptspeicher zu laden, die relevant sind.

Flexibilität

Das DMT ist vollkommen unabhängig von den jeweiligen Daten und Anwendungen. Durch die Verwendung von Deklarationsdateien kann es in beliebigen Bereichen eingesetzt werden.

Mehrbenutzerbetrieb

Mit dem DMT können mehrere Benutzer gleichzeitig auf einem zentralen Datenbestand arbeiten. Das war zwar auch vorher möglich, doch das DMT gewährleistet, daß keine Schreib- oder Schreib-/Lesekonflikte auftreten. Zudem wird sichergestellt, daß zu einem Zeitpunkt alle Benutzer die gleiche Sicht auf den Datenpool haben.

6.1.2 Verbesserungsmöglichkeiten

An dieser Stelle möchte ich auf Verbesserungsmöglichkeiten des Data Management Tools eingehen, die ich aufgrund der gemachten Erfahrungen im nachhinein als sinnvoll erachte. Unter der Prämisse, daß ich mit meinem jetzigen Wissensstand noch einmal mit der Programmentwicklung von vorne beginnen könnte und mir dabei mehr Zeit zur Verfügung stände, würde ich folgendes anders machen:

- Ich würde den Teil des DMT, der mit Datenzugriff und Datenspeicherung zu tun hat, in einer dafür geeigneteren Programmiersprache codieren. Z.B. würden sich C oder Perl anbieten, Prolog ist für solche Aufgaben nicht ausgelegt und auch einfach zu langsam. So könnte die Datenauswahl um einiges effizienter durchgeführt werden. Auf Prolog kann aber aus zwei Gründen nicht gänzlich verzichtet werden. Zum einen ist das Lernsystem in BLearn größtenteils in Prolog geschrieben, und deshalb muß das DMT eine Prolog-Schnittstelle aufweisen. Zum anderen sind die Backtracking-Möglichkeiten von Prolog bei verschiedenen Routinen (z.B. `variant_properties`) sehr wichtig.

- Das File Locking könnte verfeinert werden, indem zwischen Schreib- und Lesesperren unterschieden würde. Momentan kann zu einem Zeitpunkt immer nur ein Benutzer auf eine Partition zugreifen, auch wenn nur Leseoperationen erfolgen. Die eleganteste und wohl sauberste Lösung, um den wechselseitigen Ausschluß zu realiseren, ist aber die Implementierung als ein Dämonprozeß, wie ich es in Kapitel 5 vorgestellt habe.

- Was mir am Schluß bei der Arbeit mit dem DMT eingefallen ist, sind zusätzliche Funktionen zur Behandlung von Domänen. Es wäre sehr nützlich, wenn Domänen als Ganzes gelöscht werden könnten und wenn es eine Routine `domain_info` gäbe, die zu einer Domäne anzeigt, welche Samples sie umfaßt.

Dazu kommen noch die Punkte, die ich in Abschnitt 6.1.1 angesprochen habe; das betrifft die Art der Datenspeicherung sowie das Löschen von Varianten.

Viele Verbesserungs- und Erweiterungsmöglichkeiten werden sich auch im Laufe der Zeit ergeben, wenn das DMT im Alltag benutzt wird. Schließlich ist die Entwicklung eines Softwaresystems (fast) nie vollständig abgeschlossen.

6.1.3 Vergleich zu anderen Systemen

Wie wir gesehen haben, erfüllt das DMT die Anforderungen, die ich als Liste von Design-Zielen formuliert habe. Ich möchte nun einen Schritt weitergehen und fragen: Welchen Nutzen bringt das DMT?

Das von mir entwickelte Programm wird als Unterstützungswerkzeug beim maschinellen Lernen eingesetzt und deckt hauptsächlich drei Aufgabenbereiche ab:

1. Es verwaltet die Lerndaten.

2. Es erlaubt die Selektion von Daten, unterstützt den Benutzer also bei der Zusammenstellung von Lernsets und Domänen.

3. Es kapselt die Daten ab, womit der Benutzer entlastet und der Datenaustausch zwischen Lernverfahren oder anderen Programmen geregelt wird.

Gegenüber anderen Lernumgebungen oder Unterstützungswerkzeugen zeichnet sich das DMT dadurch aus, daß es erstens alle drei genannten Aspekte miteinander vereint und daß es zweitens vor allem die Arbeit mit verschiedenen Versionen von Daten ermöglicht. Das Letztere grenzt es z.B. von MOBAL ab. Mit dem DMT kann das durchgeführt werden, was ich in der Einführung als das Mischen von MOBAL-Domänen bezeichnete. Somit kann auch eine Speicherproblematik wie bei MOBAL verhindert werden. Was die Datenkapselung anbetrifft, so weist das DMT Ähnlichkeiten zu MOBAL auf. Das Zusammensetzen und Zerlegen von Datenelementen wird in einer umfassenderen Form auch von MOBAL über die Programmschnittstelle [Kietz 1990] angeboten. Im Vergleich zu CKRL hebt sich das DMT dadurch ab, daß es nicht nur den Datenaustausch zwischen Lernverfahren regelt, sondern darüber hinaus diese Daten verwaltet. Natürlich muß fairerweise gesagt werden, daß CKRL einen Datenaustausch auf einer höheren Ebene erlaubt und damit flexibler ist.

Zusammenfassend läßt sich festhalten, daß das DMT mit dem Variantenkonzept eine wichtige Lücke schließt. Es bietet für Lernssysteme, die ähnlich wie das BLearn-System geartet sind, eine komfortable Umgebung an, mit der die Lerndaten verwaltet, Daten zum Lernen zusammengestellt und der Datenaustausch zwischen Lernverfahren geregelt wird. Das waren im wesentlichen die Ziele, die mit der Entwicklung eines Datenverwaltungssystems verfolgt wurden. Das DMT ist somit ein auf eine besondere Art von Lernaufgabe zugeschnittenes Unterstützungswerkzeug.

6.2 Ausblick

Für die Zukunft sehe ich zwei lohnenswerte Erweiterungen, die die Arbeit mit dem DMT weiter vereinfachen. Das erste betrifft eine erleichterte Anbindung von externen Programmen, das zweite zielt auf eine graphische Benutzeroberfläche ab, die dem Anwender eine schnellere und komfortablere Bedienung ermöglicht.

6.2.1 Toolanbindung

Welche Möglichkeiten es gibt, externe Programme mit dem DMT zu koppeln, habe ich bereits im letzten Abschnitt von Kapitel 5 erläutert. Um allerdings die Anbindung an das DMT so einfach wie möglich zu gestalten, sollte es ein spezielles Programm zur Toolanbindung geben, das den Datenaustausch zwischen

DMT und externen Tools abwickelt. Möchte der Benutzer z.B. ein Lernverfahren starten, so ruft er dieses Programm mit einer Menge von Samplekennungen auf, die die Lerndaten spezifizieren. Das Programm bereitet die Daten auf und startet dann das Lernverfahren. Die Toolanbindung stellt demnach das Bindeglied zwischen Benutzer und DMT einerseits und dem externen Programm andererseits dar.

6.2.2 Graphische Benutzeroberfläche

Mit dem DMT ist die Grundlage für eine graphische Benutzeroberfläche des Lernsystems gelegt; vorher haben die Strukturen dafür noch gar nicht existiert, und es fehlte eine einheitliche Datenhaltung. Da das DMT die Verbindung zwischen den einzelnen Systemkomponenten herstellt und andererseits festdefinierte Zugriffe auf einem zentralen Datenbestand ermöglicht, ist nun von außen eine klare Sicht auf das Gesamtsystem gegeben. Die graphische Oberfläche kann darauf aufbauen, um dem Benutzer ein komfortableres Arbeiten mit dem System zu ermöglichen. Dabei greift sie natürlich auf die vom DMT zur Verfügung gestellte Funktionalität zurück und bietet diese auch an. Darüber hinaus erlaubt eine graphische Oberfläche aber viel mehr Möglichkeiten, um Informationen darzustellen. In einem Fenster könnte sich der Benutzer z.B. alle Varianten der Datenklasse A anzeigen lassen, in einem anderen Fenster alle der Datenklasse B. Ein drittes Fenster könnte den Inhalt einer bestimmten Variante anzeigen. In einem vierten Fenster wären möglicherweise alle Samples aufgeführt, die der Benutzer bereits erstellt hat.

6.3 Danksagung

Abschließend möchte ich mich bei all denjenigen bedanken, die in irgendeiner Form zu dieser Arbeit beigetragen haben. An erster Stelle sind Katharina Morik und Volker Klingspor zu nennen. Sie hatten stets ein offenes Ohr für mich und gaben mir viele wertvolle Anregungen. Ich danke Anke Rieger, daß sie mir das Data Preparation Tool zur Verfügung stellte, und Stephanie Wessel, die nach Schnittstellenvorgabe einen ersten Prototypen des Moduls varianten.pl erstellte. Der Prototyp wurde nachher verworfen, doch die Erfahrungen, die mit ihm gewonnen wurden, waren sehr nützlich bei der Implementierung der Endversion.

Anhang A

Datenformate

Sensordistanzmessungen:

```
messung(<Trace>, <Zeitpunkt>, <Sensor>, <Distanz>, <Orientierung>,
  <SensorpositionX>, <SensorpositionY>, <SensorpositionZ>, <PunktX>,
  <PunktY>, <PunktZ>, <Objekt>, <Kante>)
```

Roboterpositionen:

```
rp(<Trace>, <Zeitpunkt>, <PositionX>, <PositionY>, <Orientierung>)
```

Szenedaten:

```
scene_item(<EckeLinksUntenX>, <EckeLinksUntenY>, <EckeLinksUntenZ>,
  <EckeRechtsUntenX>, <EckeRechtsUntenY>, <EckeRechtsUntenZ>,
  <EckeRechtsObenX>, <EckeRechtsObenY>, <EckeRechtsObenZ>,
  <EckeLinksObenX>, <EckeLinksObenY>, <EckeLinksObenZ>, <Object>,
  <Kante>)
```

Basiswahrnehmungsmerkmale:

```
stable(<Trace>, <Orientierung>, <Sensor>, <Startzeitpunkt>,
  <Endzeitpunkt>, <Gradient>)
```

Analog dazu:

```
increasing/6, decreasing/6, incr_peak/6, decr_peak/6, no_movement/6,
no_measurement/6, something_happened/6, straight_away/6, straight_to/6
```

Sensormerkmale:

```
s_line(<Trace>, <Sensor>, <Startzeitpunkt>, <Endzeitpunkt>,
  <RelativeOrientierung>)
```

Analog dazu:

```
s_jump/5, s_convex/5, s_concave/5
```

Basishandlungsmerkmale:

```
stand(<Trace>, <Startzeitpunkt>, <Endzeitpunkt>)
move(<Trace>, <Startzeitpunkt>, <Endzeitpunkt>, <Geschwindigkeit>,
  <RelativeBewegung>)
amove(<Trace>, <Startzeitpunkt>, <Endzeitpunkt>, <Geschwindigkeit>,
  <Bewegung>)
rotate(<Trace>, <Startzeitpunkt>, <Endzeitpunkt>,
  <RelativeGeschwindigkeit>, <Richtung>)
```

Sensorklassen:

```
sclass(<Trace>, <Sensor>, <Startzeitpunkt>, <Endzeitpunkt>,
  <Sensorklasse>)
```

Nachfolgerrelationen:

```
d1succ(<Startzeitpunkt>, <Endzeitpunkt>)
```

Analog dazu:

```
d3succ/2, d7succ/2
```

Sensorgruppenmerkmale:

```
sg_line(<Trace>, <Sensorklasse>, <Startzeitpunkt>, <Endzeitpunkt>,
  <Orientierung>)
```

Analog dazu:

```
sg_jump/5, sg_convex/5, sg_concave/5
```

Handlungs-orientierte Wahrnehmungsmerkmale:

```
through_door(<Trace>, <Startzeitpunkt>, <Endzeitpunkt>, <Richtung>)
```

Wahrnehmungs-integrierende Handlungen:

```
standing(<Trace>, <Startzeitpunkt>, <Endzeitpunkt>, <Geschwindigkeit>,
  <Bewegung>, <Wahrnehmung>, <Richtung>)
```

Analog dazu:

```
moving/7, parallel_moving/7, rotating/7
```

Richtungsrelationen:

```
bg_r_and_new_pdirect(<Wahrnehmungsrichtung>, <Drehrichtung>,
  <WahrnehmungsrichtungDanach>)
bg_m_and_new_pdirect(<Wahrnehmungsrichtung>, <Bewegungsrichtung>,
  <WahrnehmungsrichtungDanach>)
bg_pdirect_of_corner1(<Wahrnehmungsrichtung>, <Bewegungsrichtung>,
  <WahrnehmungsrichtungDanach>)
bg_pdirect_of_corner2(<Wahrnehmungsrichtung>, <Drehrichtung>,
  <WahrnehmungsrichtungDanach>)
bg_opposite_direct(<Richtung>, <RichtungGegenueber>)
bg_new_pside(<Seite>, <SeiteDiagonal>)
```

Operationale Begriffe:

```
move_through_door(<Trace>, <Zeitpunkt1>, <Zeitpunkt2>, <Zeitpunkt3>)
```

Analog dazu:

```
rotate_in_front_of_door/4, move_in_front_of_door/4, move_along_door/4,
move_parallel_in_corner/4, rotate_in_front_of_wall/4,
move_closer_to_wall/4
```

Anhang B

Modulschnittstellen

B.1 dekl.pl

`load_declarations/1`

Synopsis:
`load_declarations(+FileSpec)`

Argumente:
`FileSpec`: Ein Atom, das die Deklarationsdatei spezifiziert.

Beschreibung:
Liest die angegebene Deklarationsdatei ein, überprüft deren Format und fügt die darin enthaltenen Fakten zur Wissensbasis des Moduls hinzu. Eine Deklarationsdatei ist eine Prolog-Textdatei, die folgende Prädikate enthält: Ein `directory/1`-Fakt, das das Verzeichnis, unter dem die gesamten Variantendaten abgespeichert sind, festlegt, ein oder mehrere `dataclass/4`-Fakten zur Definition von Datenklassen und `predicate/3`-Fakten, die Prädikatsdeklarationen darstellen. Die eingelesenen Fakten sind anschließend verfügbar und werden vom Modul dekl.pl exportiert. Die Daten einer evtl. vorher geladenen Deklarationsdatei werden aus der Wissensbasis entfernt. Konnte das Einlesen nicht ordnungsgemäß durchgeführt werden, schlägt `load_declarations` fehl, die Art des aufgetretenen Fehlers kann über `dekl_error` ermittelt werden.

`dekl_error/1`

Synopsis:
`dekl_error(-ErrorCode)`

Argumente:
`ErrorCode`: Ein Atom, das den aufgetretenen Fehler beschreibt. Da-

bei sind folgende Terme festgelegt: `no_error` (kein Fehler aufgetreten), `instantiation_error` (die Deklarationsdatei wurde nicht spezifiziert), `open_error` (die Deklarationsdatei konnte nicht ordnungsgemäß geöffnet werden), `invalid_term(<Term>)` (der Term `<Term>` ist nicht korrekt), `no_directory_defined` (in der Deklarationsdatei kommt kein `directory/1`-Fakt vor), `directory_multiple_defined` (es gibt mehrere `directory/1`-Fakten), `invalid_directory_format(<Term>)` (das Verzeichnis beginnt nicht mit einem Slash oder endet mit einem Slash), `no_dataclass_defined` (es kommen keine `dataclass/4`-Fakten vor), `dataclass_multiple_defined(<Term>)` (eine Datenklasse wird mehrfach definiert), `no_predicates_defined(<Term>)` (zu einer Datenklasse, die Faktenmengen beinhaltet, werden keine Prädikate deklariert), `no_predicates_to_define(<Term>)` (zu einer Datenklasse, die Regelmengen beinhaltet, gibt es Prädikatsdeklarationen), `key_not_corresponding_to_args(<Term>)` (der für eine Datenklasse definierte Schlüssel läßt sich mit einer Prädikatsdeklaration nicht vereinbaren) und `predicate_multiple_defined(<Term>)` (ein Prädikat wird mehrfach deklariert).

Beschreibung:

Über dieses Prädikat läßt sich feststellen, warum eine Deklarationsdatei nicht geladen werden konnte bzw. warum `load_declarations` fehlschlug.

`directory/1`

Synopsis:

```
directory(?Path)
```

Argumente:

`Path`: Atom, das den Pfad beschreibt. Der Pfad muß mit ‚`/`' beginnen, darf jedoch nicht mit diesem Zeichen enden.

Beschreibung:

Hauptverzeichnis, unter dem sich die Unterverzeichnisse der Datenklassen mit den dazugehörigen Variantendaten befinden.

`dataclass/4`

Synopsis:

```
dataclass(?Name, ?Type, ?SubDir, ?Key)
```

Argumente:

`Name`: Ein Atom, das den Namen der Datenklasse festlegt.

`Type`: `facts` oder `rules`. Beschreibt die Art der Daten, die die Datenklasse umfaßt.

`SubDir`: Unterverzeichnis, unter dem die Variantentabelle sowie die Varianten selbst abgespeichert werden. Der angegebene Pfad ist als Erweiterung zum Hauptverzeichnis zu sehen und muß mit einem Slash beginnen; das letzte Zeichen darf kein Slash sein.

`Key`: Eine Liste, die den Term `predname` für den Prädikatsnamen und zusätzlich bei Faktenmengen Argumenttypen beinhalten kann. Sind letztere angegeben, so muß gewährleistet sein, daß jede Prädikatsdeklaration für die Datenklasse diese Argumenttypen enthält. Der Schlüssel legt fest, wie eine Variante in Dateien aufgeteilt wird.

Beschreibung:

Definition einer Datenklasse, so wie sie in der Deklarationsdatei aufgeführt ist.

`predicate/3`

Synopsis:

```
predicate(?DataClass, ?Predicate, ?ArgList)
```

Argumente:

`DataClass`: Name der Datenklasse, für die das Prädikat deklariert wird.

`Predicate`: Ein Term der Form `<PredName>/<Arity>`, mit dem Prädikatsname und Stelligkeit festgelegt werden.

`ArgList`: Eine Liste von Atomen, die die Typen der Prädikatsargumente in der entsprechenden Reihenfolge bezeichnen.

Beschreibung:

Eine Prädikatsdeklaration für eine Datenklasse, die Faktenmengen umfaßt.

B.2 varianten.pl

`varianten_init/0`

Synopsis:

```
varianten_init
```

Beschreibung:

Initialisiert die Variantenverwaltung und paßt sie den Deklarationen aus dekl.pl an. Die erforderlichen Verzeichnisse und Verwaltungsdateien werden, wenn sie noch nicht existieren, angelegt und in Abhängigkeit von den Datenklassendefinitionen aktualisiert. Anschließend erfolgt das Einlesen der Variantentabellen, in denen zu jeder Datenklasse die vorhandenen Varianten aufgeführt sind. Dieses Prädikat muß vor der Benutzung von

varianten.pl und nach jeder Änderung in den Deklarationen aufgerufen werden.

`new_variant/4`

Synopsis:

`new_variant(+DataClass, +Name, +Properties, +Key)`

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`Properties`: Eine Liste, die die Varianteneigenschaften spezifiziert. Sie muß folgende Elemente (in beliebiger Reihenfolge) enthalten:

1. `user(<Atom>)`: Das Argument bezeichnet den Benutzer, der die Variante erzeugt.
2. `date([<Year>,<Month>,<Day>,<Hour>,<Minutes>,<Seconds>])`: Zeitpunkt der Erzeugung; die Listenelemente sind natürliche Zahlen.
3. `comment(<Term>)`: Ein beliebiger Kommentar, der hilfreiche Informationen zu der Variante beinhalten kann.
4. `attribute(<Term>)`: Jede Variante besitzt ein Attribut, das als zusätzliches Kennzeichen zum Wiederauffinden dient.
5. `generator(<Term>)`: Beschreibt die Methode, wie die Daten, die die Variante umfaßt, erzeugt wurden. Es gibt drei mögliche Terme: `user` (vom Benutzer „per Hand“ erzeugt), `selection` (aus einer vorhandenen Variante wurde eine Auswahl getroffen, d.h. die Daten stammen aus einem Sample) und `program(<Term>)` (ein Programm hat die Daten generiert; der Term spezifiert das Programm sowie ggf. die eingestellten Programmparameter).
6. `source(<List>)`: Legt die Quelldaten fest (das sind die Eingabedaten für das Programm im Falle von `generator(program(...))`), aus denen die Variantendaten erzeugt wurden. Die Liste ist entweder leer oder beinhaltet mehrere Elemente der Form `[<DataClass>, <VariantName>, <Method>]`, die jedes für sich eine Datenquelle beschreiben; die Reihenfolge der Elemente ist unerheblich. Über die Datenklasse, den Variantennamen und die Methode (zur Zeit gibt es nur die syntaktische Auswahl, die durch den Term `synt_select(<Constraints>)` repräsentiert wird) ist genau spezifiziert, wie die Daten zu einer Datenquelle gewonnen wurden. Alternativ dazu kann eine Datenquelle auch über eine Sample-Kennung spezifiziert werden.

`Key`: Eine Liste, die den Schlüssel beschreibt, nach dem die Variante in Dateien aufgeteilt wird (bzgl. des Formats der Liste siehe `dataclass\4`).

Der Schlüssel `[predname]` bewirkt beispielsweise, daß alle Elemente der Variante, die den gleichen Prädikatsnamen haben, in einer Partition gespeichert werden.

Beschreibung:

Zu der gewünschten Datenklasse wird eine neue Variante unter dem angegebenen Namen und mit den spezifizierten Eigenschaften angelegt—der Name darf noch nicht vergeben sein. Existiert bereits eine Variante, die aus den gleichen Daten erzeugt wurde (`source(...)`), so muß die neue Variante entweder mit `generator(user)` gekennzeichnet sein, oder die Art der Erzeugung muß sich von der vorhandenen Variante unterscheiden.

`new_variant/3`

Synopsis:

```
new_variant(+DataClass, +Name, +Properties)
```

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`Properties`: Eine Liste, die die Varianteneigenschaften spezifiziert (bzgl. des Formats siehe `new_variant/4`).

Beschreibung:

Entspricht in der Funktionalität dem Prädikat `new_variant/4`, jedoch wird der Variante automatisch der für die Datenklasse vordefinierte Schlüssel zugewiesen.

`create_partition/4`

Synopsis:

```
create_partition(+DataClass, +Name, +KeyValue, -FileSpec)
```

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`KeyValue`: Ein Liste von Atomen, die der Datei als Schlüsselwert zugeordnet wird. Die Anzahl der Listenelemente muß dem Variantenschlüssel (ebenfalls eine Liste) entsprechen.

`FileSpec`: Spezifikation der neu erzeugten Datei.

Beschreibung:

Erzeugt eine neue Datei zu der spezifizierten Variante. Ihr wird der angegebene Schlüsselwert zugewiesen.

`delete_variant/2`

Synopsis:

`delete_variant(+DataClass, +Name)`

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

Beschreibung:

Die Registrierung der durch Datenklasse und Namen bestimmten Variante wird entfernt, und die zugehörigen Dateien werden gelöscht (falls sie existieren).

`change_variant_properties/3`

Synopsis:

`change_variant_properties(+DataClass, +Name, +Properties)`

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`Properties`: Eine Liste, die die Varianteneigenschaften spezifiziert (bzgl. des Formats siehe `new_variant/4`).

Beschreibung:

Die alten Eigenschaften der Variante werden durch die angegebenen Eigenschaften ersetzt. Auch hier gilt wie bei dem Anlegen einer Variante, daß noch keine Variante existieren darf, die auf die gleiche Art und Weise erzeugt wurde (Ausnahme: `generator(user)`).

`check_variant_existence/3`

Synopsis:

`check_variant_existence(+DataClass, +Properties, ?Name)`

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Properties`: Eine Liste der Varianteneigenschaften (bzgl. des Formats siehe `new_variant/4`). Es müssen jedoch nicht alle Eigenschaften spezifiziert werden, entscheidend sind `source(...)` und `generator(...)`.

`Name`: Ein Atom für den Namen der Variante. Wurde keine Variante gefunden, so wird der Term `no_variant` zurückgeliefert.

Beschreibung:

Überprüft, ob eine Variante existiert, die auf die Art und Weise, wie sie in den angegebenen Eigenschaften spezifiziert ist, erzeugt wurde. Varianten, die vom Benutzer „per Hand“ erstellt wurden (`generator(user)`), werden hierbei nicht berücksichtigt. Dieses Prädikat hat insofern eine andere Bedeutung als `variant_properties`, als daß keine reine Unifikation erfolgt, sondern bei `source(...)` Reihenfolgeaspekte, z.B. bei den Constraints (`[item(trace, [t1, t2])]` ist gleich `[item(trace, [t2, t1])]`) ausgeglichen werden.

`variant_properties/3`

Synopsis:

`variant_properties(?DataClass, ?Name, ?Properties)`

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`Properties`: Eine Liste, die die Varianteneigenschaften spezifiziert (bzgl. des Formats siehe `new_variant/4`). Ist dieses Argument instanziiert, so müssen nicht alle Eigenschaften angegeben werden, sondern es ist beispielsweise auch der teilweise instanziierte Term `[date([95, A, B, C, D, E]), user(zitzler)]` zulässig, der alle Varianten beschreibt, die von mir 1995 erstellt wurden.

Beschreibung:

Dieses Prädikat ist erfüllt, wenn die durch die Datenklasse und den Namen bestimmte Variante die angegebenen Eigenschaften hat. Es wird versucht, die Liste, die die Eigenschaften beschreibt, mit den intern registrierten zu unifizieren—deshalb müssen nicht alle erforderlichen Kennzeichen angegeben werden.

`variant_key/3`

Synopsis:

`variant_key(?DataClass, ?Name, ?Key)`

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`Key`: Eine Liste von Atomen, die den Schlüssel beschreibt, nach dem die Variante in Dateien aufgeteilt wird.

Beschreibung:

Dieses Prädikat ist erfüllt, wenn die durch die Datenklasse und den Namen bestimmte Variante den angegebenen Schlüssel hat.

`variant_filenames/3`

Synopsis:
`variant_filenames(?DataClass, ?Name, ?KeyFilePairList)`

Argumente:
`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`KeyFilePairList`: Eine Liste, die sich aus Termen der Gestalt `file(<KeyTerm>, <FileSpec>)` zusammensetzt.

Beschreibung:
Liefert zu der gewünschten Variante alle dazugehörigen Dateien (Partitionen) mit den jeweils registrierten Schlüsselwerten.

`varianten_error/1`

Synopsis:
`varianten_error(?ErrorCode)`

Argumente:
`ErrorCode`: Ein Atom, das den aufgetretenen Fehler beschreibt. Die folgenden Fehlerfälle werden unterschieden: `no_error` (kein Fehler aufgetreten), `invalid_arguments` (die übergebenen Argumente sind nicht korrekt), `file_error` (Fehler bei einer Dateioperation), `name_clash` (eine Variante konnte nicht neu erzeugt werden, weil der ihr zugewiesene Name bereits vergeben ist), `variant_exists` (es gibt schon eine Variante, die auf diese Art und mit diesen Eingabedaten erzeugt wurde), `partition_exists` (für den gewünschten Schlüsselwert existiert bereits eine Datei) und `sample_error(<Term>)` (auf ein Sample kann entweder nicht zugegriffen werden, oder es läßt sich nicht ermitteln, wie das Sample direkt aus der Ursprungsvariante gezogen werden kann).

Beschreibung:
Liefert den Code des zuletzt aufgetretenen Fehlers. Alle Prädikate, die von varianten.pl exportiert werden, schlagen im Fehlerfall fehl (Ausnahme: Backtracking).

B.3 zugriff.pl

`synt_select/3`

Synopsis:
`synt_select(+Source, +Constraints, -SampleId)`

Argumente:

`Source`: Die Einheit, aus der die Daten ausgewählt werden sollen. Entweder ein Term der Gestalt `variant(<DataClass>, <Name>)`, mit der eine Variante bezeichnet wird, oder ein Term `sample(<SampleId>)`, der ein bestehendes Sample spezifiziert.

`Constraints`: Beschreibt die Bedingungen, nach denen die Daten selektiert werden sollen. Eine Liste aus Termen der Form `item(<Item>, <Values>)`, mit denen die Wertebereiche der verschiedenen Kriterien angegeben werden können; es steht `<Item>` für ein Atom und `<Values>` für eine Liste von Termen. Statt dessen kann auch ein Atom angegeben werden, wobei `all` bedeutet, daß alle Elemente der Datenmenge ausgewählt werden sollen, und `empty` für den Fall steht, daß ein leeres Sample erstellt werden soll.

`SampleId`: Eine Kennung für das erstellte Sample.

Beschreibung:

Mit Hilfe dieses Prädikats kann aus einer Variante eine bestimmte Teilmenge von Daten herausgefiltert werden. Dies geschieht anhand syntaktischer Kriterien, die die gesuchten Daten beschreiben. Stellt die Variante eine Faktenmenge dar, so kann eine Einschränkung über den Prädikatsnamen (dieses Kriterium ist vordefiniert als `predname`) als auch über die verschiedenen Argumenttypen erfolgen, die in `dekl.pl` festgelegt sind und aus der Deklarationsdatei stammen. Im Falle einer Regelmenge gibt es die Kriterien Prädikatsname (`predname`), Kopf (`head`), Körper (`body`) und Klausel (`clause`), was die Regel als Ganzes betrachtet. Die Bedingungen für die Datenselektion sind spezifiziert, indem für die gewünschten Kriterien zulässige Werte aufgeführt werden (z.B. `item(predname, [s_line, s_jump])`). Sind keine Werte angegeben, so werden damit alle Elemente beschrieben, bei denen das Kriterium überhaupt anwendbar ist. Mit anderen Worten: Will man beispielsweise alle Fakten auswählen, die als Argument einen Anfangszeitpunkt haben (`tstart`), dann liefert eine syntaktische Auswahl mit den Constraints `[item(tstart, [])]` die gewünschten Elemente. Alternativ dazu gibt es jedoch auch die Möglichkeit alle Daten auszuwählen (`all`) oder keine (`empty`). Die selektierten Daten werden in den Speicher eingelesen und als ein Sample abgelegt. Die Kennung des erzeugten Samples wird zurückgegeben. Zusätzlich ist es auch möglich, aus bereits bestehenden Samples zu selektieren.

Eine Anmerkung, die den Zugriff auf Varianten betrifft: Da Varianten prinzipiell Dateien darstellen, werden Änderungen (Hinzufügen oder Entfernen von Elementen) aus Zeitgründen nicht sofort durchgeführt, sondern zunächst nur vermerkt. Für eine Aktualisierung ist ein Aufruf des Prädikats `clean_variant` nötig, der ggf. automatisch bei einem `synt_select` getätigt wird.

`case_select/4`

Synopsis:

`case_select(+TarSampleId, +RelList, +DefSampleId, -CaseList)`

Argumente:

`TarSampleId`: Die Kennung des Samples, das die Zielprädikate enthält.

`RelList`: Eine Relationenliste der Form `[<TargetItem>, <Relation>, <DefItem>]`, wobei `<TargetItem>` einen Argumenttyp der Zielprädikate und `<DefItem>` einen Argumenttyp der definierenden Prädikate bezeichnet. `Relation` steht für einen der Terme `'='`, `'\='`, `'=:='`, `'=\='`, `'<'`, `'>'`, `'=<'`, `'>='`.

`DefSampleId`: Die Kennung des Samples, das die definierenden Prädikate enthält.

`CaseList`: Eine Liste, die wiederum Listen enthält, deren erstes Element das Zielprädikat ist und deren folgenden Elemente die definierenden Prädikate sind.

Beschreibung:

Zu jedem Zielprädikat werden die definierenden Prädikate bestimmt, die durch die Relationenliste beschrieben werden. Jede Relation bezieht sich einerseits auf ein Argument eines Zielprädikats und andererseits auf ein Argument eines definierenden Prädikats; die beiden Argumente müssen in dem durch die Relation gegebenen Verhältnis stehen. Diese Routine ist nur auf Faktenmengen anwendbar.

`access_sample/3`

Synopsis:

`access_sample(+SampleId, -PosList, -NegList)`

Argumente:

`SampleId`: Die Kennung des Samples, auf das zugegriffen werden soll.

`PosList`: Liste der positiven Elemente.

`NegList`: Liste der negativen Elemente.

Beschreibung:

Über die Kennung kann auf die unter einem Sample zusammengefaßten Daten zugegriffen werden.

`access_sample/4`

Synopsis:

`access_sample(+SampleId, -PosList, -NegList, -InfoList)`

Argumente:

`SampleId`: Die Kennung des Samples, auf das zugegriffen werden soll.

`PosList`: Liste der positiven Elemente.

`NegList`: Liste der negativen Elemente.

`InfoList`: Eine Liste von Einträgen, die die verschiedenen Informationen zu einem Sample beinhaltet. Ihr Format ist fest vorgegeben und sieht folgendermaßen aus: `[<Type>,<DataClass>, <Source>, <Method>, <DirectSource>, <DirectMethod>, <DPTSampleId>]`. `<Type>` steht für einen der Terme `facts` oder `rules` und gibt den Typ der Daten an. `<DataClass>` bezeichnet die Datenklasse des Samples, `Source` die Quelle, aus der die Daten gewonnen wurden (das kann `sample(<SampleId>)`, `variant(<DataClass>, <Name>)` oder eine Liste von Termen `sample(<SampleId>)` sein) und `Method` die Art und Weise, wie die Daten gewonnen wurden (`synt_select(<Constraints>)`, `merge_samples` oder `user_defined` bei durch `add` bzw. `erase` manipulierten Samples). An den nächsten beiden Listenpositionen stehen `<DirectSource>` und `<DirectMethod>`, deren Format dem von `<Source>` und `<Method>` gleicht. Sie geben an, wie das Sample direkt aus der Ursprungsvariante erzeugt werden kann. Diese „Rückverfolgung“ ist jedoch nur dann möglich, wenn die Samplekette von der Variante bis zum Zielsample eine Folge von syntaktischen Auswahlen ist und auch noch vollständig erhalten ist. Gibt es keine direkte Methode oder kann sie nicht mehr ermittelt werden, so werden die Terme `no_source` und `no_method` übergeben. An letzer Stelle ist die Kennung des entsprechenden Samples aufgeführt (`<DPTSampleId>`), das vom *Data Preparation Tool* erzeugt wurde (nur bei Faktenmengen von Bedeutung).

Beschreibung:

Über die Kennung kann auf ein Sample zugegriffen werden. Als Erweiterung zu `access_sample/3` sind zusätzliche Informationen über das Sample abrufbar.

`sort_sample/3`

Synopsis:

`sort_sample(+SampleId, +SortItems, +Relation)`

Argumente:

`SampleId`: Kennung des Samples, das sortiert werden soll.

`SortItems`: Eine Liste der Kriterien, nach denen sortiert werden soll. Bei Faktenmengen können hierbei der Prädikatsname (`predname`) als auch entsprechende Argumenttypen angegeben werden, für Regelmengen sind die Terme `predname`, `head`, `body` und `clause` zulässig.

`Relation`: Bestimmt die Reihenfolge, nach der sortiert wird. Es steht `<` für aufsteigende und `>` für absteigende Sortierung.

Beschreibung:
Sortiert die Daten eines Samples in auf- oder absteigender Reihenfolge nach den angegebenen Sortierkriterien.

`delete_sample/1`

Synopsis:
`delete_sample(+SampleId)`

Argumente:
`SampleId`: Kennung des Samples, das aus dem Speicher entfernt werden soll.

Beschreibung:
Entfernt ein Sample aus dem Speicher.

`merge_samples/3`

Synopsis:
`merge_samples(+SrcSampleId1, +SrcSampleId2, -SampleId)`

Argumente:
`SrcSampleId1/2`: Die Kennungen der Samples, deren Daten zusammengefaßt werden sollen.

`SampleId`: Kennung des erstellten Samples.

Beschreibung:
Faßt die Daten zweier Samples gemäß einer Mengenvereinigung zusammen und legt sie als ein neues Sample ab.

`add/3`

Synopsis:
`add(+Source, +MemberList, +Sign)`

Argumente:
`Source`: Die Dateneinheit, in die ein neues Element eingefügt werden soll. Entweder ein Term der Gestalt `variant(<DataClass>, <Name>)`, mit der eine Variante bezeichnet wird, oder ein Term `sample(<SampleId>)`, der ein bestehendes Sample spezifiziert.

`MemberList`: Liste der einzufügenden Elemente.

`Sign`: Gibt an, ob die Elemente positiv (hierfür muß der Term `pos` angegeben werden) oder negativ (entspricht dem Term `neg`) gekennzeichnet sind.

Beschreibung:

Zu einer Variante oder einem Sample werden neue Elemente hinzugefügt, falls diese nicht bereits existieren. Das Argument `Sign` gibt an, ob die Elemente positiv oder negativ klassifiziert sind. Regeln werden unabhängig von der angegebenen Klassifikation als positiv gekennzeichnet.

`add/2`

Synopsis:

```
add(+Source, +MemberList)
```

Argumente:

`Source`: Die Dateneinheit, in die ein neues Element eingefügt werden soll. Entweder ein Term der Gestalt `variant(<DataClass>, <Name>)`, mit der eine Variante bezeichnet wird, oder ein Term `sample(<SampleId>)`, der ein bestehendes Sample spezifiziert.

`MemberList`: Die Liste der einzufügenden Elemente.

Beschreibung:

Zu einer Variante oder einem Sample werden neue Elemente hinzugefügt, falls diese nicht bereits existieren. Die Elemente sind automatisch positiv klassifiziert.

`erase/3`

Synopsis:

```
erase(+Source, +MemberList, +Sign)
```

Argumente:

`Source`: Die Dateneinheit, aus der ein Element gelöscht werden soll. Entweder ein Term der Gestalt `variant(<DataClass>, <Name>)`, mit der eine Variante bezeichnet wird, oder ein Term `sample(<SampleId>)`, der ein bestehendes Sample spezifiziert.

`MemberList`: Liste der zu entfernenden Elemente.

`Sign`: Gibt an, ob die Elemente positiv (hierfür muß der Term `pos` angegeben werden) oder negativ (entspricht dem Term `neg`) gekennzeichnet sind.

Beschreibung:

Aus einer Variante oder einem Sample werden Elemente entfernt, falls diese existieren. Das Argument `Sign` gibt an, ob die Elemente positiv oder negativ klassifiziert sind. Bei Regeln spielt die Klassifikation keine Rolle.

`erase/2`

Synopsis:
`erase(+Source, +MemberList)`

Argumente:
`Source`: Die Dateneinheit, aus der ein Element gelöscht werden soll. Entweder ein Term der Gestalt `variant(<DataClass>, <Name>)`, mit der eine Variante bezeichnet wird, oder ein Term `sample(<SampleId>)`, der ein bestehendes Sample spezifiziert.

`MemberList`: Liste der zu entfernenden Elemente.

Beschreibung:
Aus einer Variante oder einem Sample werden Elemente entfernt, falls diese existieren. Die Elemente sind automatisch positiv klassifiziert.

`compose/4`

Synopsis:
`compose(?DataClass, ?ItemList, ?TermList, ?Member)`

Argumente:
`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`ItemList`: Eine Liste der Kriterien, die das zusammengesetzte Element eindeutig spezifiziert.

`TermList`: Eine zu `ItemList` korrespondierende Liste, die für jedes Kriterium einen Term enthält. Die Position des Kriteriums in `ItemList` bestimmt die Position des entsprechenden Terms in `TermList`.

`Member`: Das zusammengesetzte Element.

Beschreibung:
Ein neues Element einer bestimmten Datenklasse wird erzeugt, ohne daß es in eine Variante oder ein Sample eingefügt wird. Soll ein Fakt kreiert werden, so müssen für den Prädikatsnamen sowie für alle dem gewünschten Prädikat zugeordneten Argumenttypen Werte angegeben werden. Bei einer Regel müssen entweder Kopf und Körper spezifiziert werden, oder sie kann als Ganzes mittels des Kriteriums `clause` übergeben werden.

`decompose/4`

Synopsis:
`decompose(?DataClass, ?Member, ?ItemList, ?TermList)`

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Member`: Element, das zerlegt werden soll.

`ItemList`: Eine Liste der Kriterien, anhand derer das Element zerlegt werden soll.

`TermList`: Eine zu `ItemList` korrespondierende Liste, die für jedes Kriterium einen Term enthält. Die Position des Kriteriums in `ItemList` bestimmt die Position des entsprechenden Terms in `TermList`.

Beschreibung:

Das Gegenstück zu `compose`. Ein Element aus einer Datenklasse wird in seine Bestandteile zerlegt bzw. zu den angegebenen Kriterien werden die entsprechenden Werte ermittelt.

`clean_variant/3`

Synopsis:

```
clean_variant(+DataClass, +Name, +Condition)
```

Argumente:

`DataClass`: Ein Atom, das die Datenklasse bezeichnet.

`Name`: Ein Atom für den Namen der Variante.

`Condition`: Ein Atom, welches die Bedingung setzt, wann die Variante bereinigt werden soll. `modified` bedeutet, daß nur im Falle von zurückgestellten Änderungen bereinigt werden soll, `always` erzwingt eine Bereinigung in jedem Fall.

Beschreibung:

Varianten bestehen aus ein oder mehreren Dateien, und es ist somit sehr zeitaufwendig, Änderungen an ihnen vorzunehmen, gerade was das Entfernen von Elementen betrifft. Daher werden Modifikationen über `add` oder `erase` nicht sofort durchgeführt, sondern sie werden gesammelt und jeweils vermerkt. Bei jedem erneuten Zugriff über beispielsweise `synt_select` muß eine Variante allerdings auf den neuesten Stand gebracht werden. Alle Änderungen müssen jetzt vollzogen werden, wobei u.a. zu beachten ist, daß keine Elemente doppelt vorkommen. Wird nur über zugriff.pl auf die Dateien zugegriffen, so braucht sich der Benutzer nicht um die Bereinigung von Varianten kümmern, es geschieht automatisch. Will man jedoch eine Variante mit einem Editor bearbeiten, so muß sichergestellt sein, daß sie bereits aktualisiert ist. Dies ist der eine Grund, warum dieses Prädikat zur Verfügung gestellt wird. In diesem Fall erfolgt der Aufruf mit `modified` als drittes Argument. Es kann jedoch auch vorkommen, daß der Benutzer eine Variante per Hand geändert hat. Dann kann eine Bereinigung sinnvoll sein, um sicherzustellen, daß keine Daten doppelt in der Datei vorkommen. Der Aufruf von `clean_variant`

mit `always` als drittes Argument erzwingt eine Bereinigung der Variante, auch wenn in der Zwischenzeit keine Änderungen über `add` oder `delete` vorgenommen wurden.

Ein Hinweis noch den Mehrbenutzerbetrieb betreffend: Grundsätzlich gilt, daß an einer Variante nur Änderungen vorgenommen werden sollten, wenn keine anderen Benutzer auf sie zugreifen. Jedoch kann es vorkommen, daß nach einer Modifikation zwei Benutzer über die Variante eine syntaktische Auswahl vornehmen und somit die automatischen `clean_variant`-Aufrufe in Konflikt geraten. Aus diesem Grunde ist ein einfacher Locking-Mechanismus implementiert, so daß nur ein Prozeß die Variante tatsächlich bereinigt, während der andere wartet, bis dies geschehen ist.

`zugriff_error/1`

Synopsis:

`zugriff_error(-ErrorCode)`

Argumente:

`ErrorCode`: Eine Kodierung für den aufgetretenen Fehler. Folgende Terme werden als Fehlercode benutzt: `no_error` heißt, daß kein Fehler vorliegt, `invalid_arguments` bedeutet, dem vorher aufgerufenen Prädikat wurden falsche Argumente übergeben, `internal_error` steht für das Auftreten eines unvorhergesehenen Programmfehlers und `file_error` bezeichnet einen Dateifehler.

Beschreibung:

Alle Prädikate, die von zugriff.pl exportiert werden, sind im Falle eines Fehlers nicht erfüllt. Die Ursache für das Fehlschlagen eines Prädikats kann mit Hilfe von `zugriff_error` ermittelt werden. Es wird immer der zuletzt aufgetretene Fehler gemeldet.

B.4 benutzer.pl

`dataclass_info/1`

Synopsis:

`dataclass_info(?DataClass)`

Argumente:

`DataClass`: Name der Datenklasse, zu der Informationen ausgegeben werden soll.

Beschreibung:

Dieses Prädikat stellt die zu einer Datenklasse verfügbaren Informationen in übersichtlicher Form dar, ohne umständlich über dekl.pl gehen zu müssen.

`variant_info/2`

Synopsis:

```
variant_info(?DataClass, ?VariantName)
```

Argumente:

`DataClass`: Name der Datenklasse, zu der die Variante gehört.

`VariantName`: Name der Variante, zu der Informationen gewünscht werden.

Beschreibung:

Gibt die zu einer Variante abrufbaren Informationen strukturiert auf der Standardausgabe aus.

`sample_info/1`

Synopsis:

```
sample_info(+SampleId)
```

Argumente:

`SampleId`: Kennung des Samples.

Beschreibung:

Die Informationen, die über `access_sample/4` für das angegebene Sample abrufbar sind, werden formatiert dargeboten. Dies beinhaltet jedoch nicht die Daten, die unter der Sample-Kennung abgelegt sind; sie können mittels `list_sample` ausgegeben werden.

`list_sample/2`

Synopsis:

```
list_sample(+SampleId, +Stream)
```

Argumente:

`SampleId`: Kennung des Samples.

`Stream`: Stream, in den die Ausgabe erfolgen soll.

Beschreibung:

Gibt alle Elemente, die unter dem spezifizierten Sample zusammengefaßt sind, auf dem angegebenen Stream aus.

`list_sample/1`

Synopsis:
`list_sample(+SampleId)`

Argumente:
`SampleId`: Kennung des Samples.

Beschreibung:
Listet alle Elemente, die unter dem spezifizierten Sample zusammengefaßt sind, über die Standardausgabe auf.

`add_file_to_variant/3`

Synopsis:
`add_file_to_variant(+DataClass, +VariantName, +FileSpec)`

Argumente:
`DataClass`: Name der Datenklasse.

`VariantName`: Name der Variante, die die Daten aufnehmen soll.

`FileSpec`: Spezifikation der Datei, deren Inhalt in eine Variante geschrieben werden soll.

Beschreibung:
Liest eine Prolog-Datei ein, fügt die darin enthaltenen Terme per `add` zu der angegeben Variante hinzu und bereinigt anschließend diese Variante. Dabei wird Negation bzw. mehrfache Negation von Termen berücksichtigt, so daß Fakten auch korrekt klassifiziert werden.

`save_domain/2`

Synopsis:
`save_domain(+SampleIdList, +FileSpec)`

Argumente:
`SampleIdList`: Liste von Sample-Kennungen.

`FileSpec`: Spezifikation der Datei, in der die Domäne abgespeichert werden soll.

Beschreibung:
Mit dieser Routine können Informationen zur Erstellung bestimmter Samples abgespeichert werden. Samples sind ja an und für sich temporär, müssen also bei jedem Systemstart neu angelegt werden. Damit das nicht jedesmal von Hand erfolgen muß, bietet `save_domain` die Möglichkeit, eine Domäne als eine Menge von Samples zu definieren. Diese Domäne kann

dann mittels `load_domain` wieder eingeladen werden, wobei die Samples automatisch generiert werden.

`load_domain/3`

Synopsis:

```
load_domain(+FileSpec, -DataClassList, -SampleIdList)
```

Argumente:

`FileSpec`: Spezifikation der Datei, in der die Domäne abgespeichert ist.

`DataClassList`: Eine Liste der Namen der Datenklassen.

`SampleIdList`: Eine Liste der Sample-Kennungen.

Beschreibung:

Dieses Prädikat generiert automatisch die Samples, die unter der angegebenen Domäne zusammengefaßt sind. Während `save_domain` Informationen zur Erstellung bestimmter Samples abspeichert, nutzt `load_domain` diese Informationen, um die Samples wieder zu erzeugen. Natürlich ändern sich dabei die Kennungen der Samples. Deswegen wird zusätzlich eine Liste der Namen der Datenklassen zurückgeliefert, denen die Samples angehören. Die Reihenfolge der Listenelemente korrespondiert bei beiden Listen miteinander.

`domain_select/3`

Synopsis:

```
domain_select(+SrcSampleIdList, +Constraints, -SampleIdList)
```

Argumente:

`SrcSampleIdList`: Eine Liste von Kennungen der Samples, aus denen ausgewählt werden soll.

`Constraints`: Kriterien, nach denen die Auswahl erfolgen soll. Bzgl. des Formats siehe `synt_select/3`.

`SampleIdList`: Eine Liste von Kennungen der Samples, die neu erzeugt wurden.

Beschreibung:

Auf einer Menge von Samples (einer Domäne) wird nacheinander eine syntaktische Auswahl mit identischen Constraints vorgenommen. Es entsteht eine gleichgrosse Menge von neuen Samples. Die Reihenfolge der Sample-Listen korrespondieren miteinander, d.h. das erste Sample in der `SampleIdList` wurde mit einem `synt_select` auf dem ersten Sample der `SrcSampleIdList` generiert, usw.

B.5 lock.pl

`lock_file/1`

Synopsis:
`lock_file(+FileSpec)`

Argumente:
`FileSpec`: Spezifikation der zu sperrenden Datei.

Beschreibung:
Dieses Prädikat realisiert einen einfachen Locking-Mechanismus auf Dateien, unter der Annahme, daß alle beteiligten Prozesse nur dann auf eine Datei zugreifen, wenn sie auch erfolgreich gesperrt wurde; Dateizugriffe sind also auch ohne einen Lock möglich, führen jedoch zu unvorhersehbaren Seiteneffekten. Ein Aufruf von `lock_file` bewirkt, daß die angegebene Datei gesperrt wird. Ist die Datei bereits von einem anderen Prozeß gelockt, so wird entsprechend lange gewartet, bis die Datei wieder freigegeben ist.

`unlock_file/1`

Synopsis:
`unlock_file(+FileSpec)`

Argumente:
`FileSpec`: Spezifikation der zu entsperrenden Datei.

Beschreibung:
Hebt die Sperre auf der angegebenen Datei auf. Ist die Datei nicht gesperrt, hat das Prädikat keine Wirkung.

Literaturverzeichnis

[Abowd *et al.* 1995] G. Abowd, R. Beale, A. Dix, J. Finlay (1995): *Mensch, Maschine, Methodik*. Prentice Hall, München, etc.

[Causse *et al.* 1990] K. Causse, K. Morik, C. Rouveirol, P. Sims (1990): *Comparative Study of the Representation Languages Used in the MLT*. Arbeitspapiere der GMD, 453, Subreihe Künstliche Intelligenz Nr. 4, St. Augustin.

[Cohen 1992] W.W. Cohen (1992): *The Grendel Learning System*. AT&T Bell Laboratories, September 1992.

[Dillmann *et al.* 1993] R. Dillmann, J. Kreuziger, F. Wallner (1993): PRIAMOS—An Experimental Platform for Reflexive Navigation. In Groen, Hirose, Thorpe (Hrsg.): *IAS-3: Intelligent Autonomous Systems*, IOS Press, S. 174–183.

[Gulbins 1988] J. Gulbins (1988): *UNIX: eine Einführung in Begriffe und Kommandos von UNIX—Version 7, bis System V.3*. Springer-Verlag, Berlin, etc.

[Jalote 1991] P. Jalote (1991): *An Integrated Approach To Software Engineering*. Springer-Verlag, New York, etc.

[Kietz 1990] J.U. Kietz (1990): *Deliverable 4.2/G — MOBAL's Program Interface*. MLT-Projekt (ESPRIT 2154), Referenz: GMD/P2154/6/1, 24. Juli 1991.

[Kietz 1991] J.U. Kietz (1991): *Deliverable 4.2/G — MOBAL's CKRL Interface*. MLT-Projekt (ESPRIT 2154), Referenz: GMD/P2154/24/1, September 1991.

[Kietz und Wrobel 1992] J.U. Kietz, S. Wrobel (1992): Controlling the Complexity of Learning in Logic through Syntactic and Task-Oriented Models. In S. Muggleton (Hrsg.): *Inductive Logic Programming*, Kapitel 16, S. 335–360, Academic Press, London.

[Klingspor 1994] V. Klingspor (1994): GRDT: Enhancing Model-Based Learning for its Application in Robot Navigation. Forschungsbericht LS8/5, Universität Dortmund, Fachbereich Informatik, Lehrstuhl VIII.

[Klingspor und Morik 1995] V. Klingspor, K. Morik, (1995): Towards concept formation grounded on perception and action of a mobile robot. In Rembold, Dillmann, Hertzberger, Kanade (Hrsg.): *IAS-4: Intelligent Autonomous Systems*, IOS Press, S. 271–278.

[Klingspor *et al.* 1996] V. Klingspor, K. Morik, A. Rieger (1996): Learning Concepts from Sensor Data of a Mobile Robot. In *Machine Learning* (to appear).

[Morik 1989] K. Morik (1989): Sloppy modeling. In K. Morik (Hrsg.): *Knowledge Representation and Organization in Machine Learning*, S. 107–134, Springer Verlag, Berlin, etc.

[Morik 1995] Morik, K. (1995): Maschinelles Lernen. In G. Görz (Hrsg.): *Einführung in die Künstliche Intelligenz*, 2. Auflage, Kapitel 3, S. 243–293, Addison-Wesley, Bonn, etc.

[Morik und Rieger 1993] K. Morik, A. Rieger (1993): Learning action-oriented perceptual features for robot navigation. Forschungsbericht LS8/3, Universität Dortmund, Fachbereich Informatik, Lehrstuhl VIII.

[Morik *et al.* 1993] K. Morik, S. Wrobel, J.U. Kietz, W. Emde (1993): *Knowledge Acquisition and Machine Learning: Theory, Methods, and Applications.* Academic Press, London, etc.

[Muggleton und Raedt 1993] S. Muggleton, L. De Raedt (1993): Inductive Logic Programming: Theory and Methods. CW 178, Department of Computing Science, K.U. Leuven.

[Ottmann und Widmayer 1990] T. Ottmann, P. Widmayer (1990): *Algorithmen und Datenstrukturen.* BI-Wissenschaftsverlag, Mannheim, etc.

[Quinlan 1990] J.R. Quinlan (1990): Learning Logical Definitions from Relations. In *Machine Learning*, 5(3), S. 239–266.

[Rieger 1995] A. Rieger (1995): Data Preparation for Inductive Learning in Robotics. Forschungsbericht LS8/19, Universität Dortmund, Fachbereich Informatik, Lehrstuhl VIII.

[Rieger 1996] A. Rieger (1996): A Data Preparation Tool for Relational Inductive Learning in Robotics. In *Proceedings of the 13th European Meeting on Cybernetics and Systems Research*, World Scientific.

[Sklorz 1995] S. Sklorz (1995): *Repräsentation operationaler Begriffe zum Lernen aus Roboter-Sensordaten.* Universität Dortmund, Fachbereich Informatik, Diplomarbeit.

[Smith 1991] M.F. Smith (1991): *Software prototyping: adoption, practice and management.* McGraw-Hill, London, etc.

[Spitta 1989] T. Spitta (1989): *Software Engineering und Prototyping.* Springer-Verlag, Berlin, etc.

[Tanenbaum 1994] A.S. Tanenbaum (1994): *Moderne Betriebssysteme.* Prentice-Hall, London.

[van Heijst 1995] G.A.C.M. van Heijst (1995): *The Role of Ontologies in Knowledge Engineering.* Kapitel 7, S. 117–127, Universität Amsterdam, Fachbereich Psychologie, Dissertation.

[Wessel 1995] S. Wessel (1995): *Lernen qualitativer Merkmale aus numerischen Robotersensordaten.* Universität Dortmund, Fachbereich Informatik, Diplomarbeit.