„Objektorientierte Software-Entwicklung am Beispiel des probabilistischen wissensbasierten Systems SPIRIT“

Diplomarbeit an der Universität Ulm
Fakultät für Informatik

vorgelegt von

Holger Knublauch

1. Gutachter: Prof. Dr. H. Partsch
2. Gutachter: Prof. Dr. W. Rödder

Inhaltsverzeichnis

Einleitung ... **3**

1 Aufgabenstellung, Problemanalyse und Methodik ... **5**

1.1 SPIRIT - eine kompakte Einführung ... 5
1.2 Die Anforderungen (Requirements) ... 6
1.3 Neuentwicklung und Reengineering ... 9

2 Analyse ... **13**

2.1 Domain Object Model ... 13
2.2 System Object Model ... 15
2.3 Operation Model ... 16
2.4 Life-Cycle Model ... 21

3 Design ... **23**

3.1 Erweiterung des Analyse-Modells um Implementierungskonzepte ... 23
3.2 Object Interaction Graphs ... 32
3.3 Visibility Graphs ... 43
3.4 Inheritance Graphs ... 46
3.5 Class Descriptions ... 50

4 Implementierung, Test und Dokumentation ... **59**

4.1 JAVA als Programmiersprache im Software Engineering ... 59
4.2 Implementierung des Systemkerns ... 62
4.3 Test ... 64
4.4 Dokumentation für Anwendungsprogrammierer ... 68

5 Fazit ... **69**

5.1 Erfahrungen mit FUSION ... 69
5.2 Vergleich von SPIRIT mit den Vorgängerversionen ... 73
5.3 Ausblick ... 75

Anhang A: Syntax von Regeln und Fakten in SPIRIT ... **76**

Anhang B: Format von SPIRIT-Dateien ... **77**

Anhang C: Von SPIRIT gemeldete Fehler (*Exceptions*) ... **78**

Anhang D: Einige Algorithmen ... **79**

Anhang E: Kommandos des Testtreibers SPECTER ... **87**

Anhang F: Data-Dictionary ... **88**

Literaturverzeichnis ... **89**

Zusammenfassung ... **90**

Einleitung

SPIRIT [RM96] ist ein probabilistisches entscheidungsunterstützendes System, das sein Wissen aus ausdrucksstarken Regeln und Fakten über diskrete Variablen erhält. Nach der Deklaration der Variablen und Regeln durch einen Experten baut SPIRIT eine gemeinsame Wahrscheinlichkeits-Verteilung auf, in der die durch einen iterativen Lernprozeß ermittelten bedingten Wahrscheinlichkeiten gespeichert werden. Nach dem Lernen der Regeln können daraufhin einfache und komplexe Anfragen an die Wissensbasis gestellt werden.

Das Projekt SPIRIT ist ein Forschungsschwerpunkt des Lehrgebietes für BWL, insbes. Operations Research an der FernUni Hagen. Im Rahmen meiner Tätigkeit als studentische Hilfskraft für diesen Lehrstuhl war ich an der Entwicklung verschiedener experimenteller Versionen von SPIRIT (in C++, bzw. Borland Delphi) beteiligt, die im folgenden als *Vorgängerversionen* bezeichnet werden. Unsere bei diesen Versionen erworbenen Erfahrungen, die Absicherung der Forschungsergebnisse und die in der Praxis festgestellten positiven und negativen Eigenschaften empfahlen einen gründlichen Neuentwurf (ein sog. *Reengineering*) des Projektes.

Aufgabenstellung

Aufgabe der vorliegenden Diplomarbeit war die Entwicklung einer objektorientierten Klassenbibliothek zur Erstellung von wissensbasierten Systemen nach der Theorie zu SPIRIT. Obwohl die Arbeit zahlreiche Konzepte und Algorithmen enthält, die zur Umsetzung der theoretischen Ansätze größtenteils neu entwickelt werden mußten, ist ein wesentlicher Schwerpunkt im Bereich *Software-Engineering* zu sehen. Auf das theoretische Fundament kann und möchte ich vor allem nur dort eingehen, wo es zum Verständnis aus Sicht der Informatik nötig erscheint.

Die Software wurde mit der objektorientierten Entwicklungsmethode *FUSION* [Col94] entworfen und in der plattformunabhängigen Programmiersprache *JAVA* [Hof96] implementiert. Ergebnis sollte neben der lauffähigen Software und einer sauberen Dokumentation auch ein nachvollziehbarer Bericht über den Entwicklungsprozeß an sich sein. Insbesondere sollte auf die Erfahrungen mit FUSION eingegangen und JAVA auf seine Tauglichkeit als Implementierungssprache für ein solches Projekt getestet werden. Mich persönlich interessierte außerdem, ob und wenn ja bis zu welcher Stufe sich ein so großes Projekt überhaupt abstrakt entwickeln läßt, ohne mit der Implementierung zu beginnen und auf Quelltextebene zu arbeiten.

Übersicht

Die vorliegende Arbeit beschreibt Vorgehensweise und Ergebnisse des Entwicklungsprozesses für das Projekt SPIRIT. Nach einer kompakten Einführung in das Konzept von SPIRIT in Abschnitt 1.1, wird in Abschnitt 1.2 die Aufgabenstellung in die Form relativ exakter *Requirements* gebracht, auf denen die darauffolgenden Arbeitsschritte basieren. In Abschnitt 1.3.1 wird dann auf die bisherigen Versionen von SPIRIT eingegangen, die in den zurückliegenden drei Jahren implementiert und erprobt wurden. Abschnitt 1.3.2 stellt die gewählte Entwicklungsmethode FUSION kurz vor und erläutert die Vorgehensweise bei den folgenden Schritten.

Aufbauend auf dem Anforderungsdokument aus Abschnitt 1.2 enthalten Kapitel 2 und 3 die Ergebnisse von Analyse- und Entwurfsphase nach FUSION. Hierbei wurde die von FUSION vorgeschlagene Bearbeitungsreihenfolge mit den jeweiligen Zwischenergebnissen im wesentlichen eingehalten, so daß sich die Gliederung der Unterkapitel direkt aus den erforderlichen Schritten des Entwicklungsprozesses ergab.

Anschließend gibt Kapitel 4 einen Überblick über die Implementierungs- und Testphase. Hierbei wird zunächst kurz die Wahl von JAVA als Programmiersprache begründet, auf dessen Besonderheiten eingegangen und die Eignung von JAVA zur Implementierung von softwaretechnisch entworfener Software geprüft (Abschnitt 4.1). Der darauffolgende Abschnitt 4.2 erläutert die Vorgehensweise bei der Umwandlung der Entwurfsmodelle in ein JAVA-Programm. Dann beschreibt Abschnitt 4.3 die in der Testphase entwickelten Testtreiber SPECTER und SPIRITSHELL, die zur Erprobung und Optimierung der System-Operationen eingesetzt wurden. Außerdem werden einige der durchgeführten Testläufe beschrieben und Ergebnisse der Testphase genannt. Schließlich wird in Abschnitt 4.4 kurz auf die Dokumentation der SPIRIT-Klassenbibliothek für Anwendungsprogrammierer eingegangen.

Das abschließende Kapitel 5 faßt die Ergebnisse des Entwicklungsprozesses zusammen. Insbesondere wird auf die mit FUSION gemachten Erfahrungen eingegangen (Abschnitt 5.1). Ein kritischer Vergleich der neuen SPIRIT-Version mit seinen Vorgängern wird in Abschnitt 5.2 durchgeführt. Zum Abschluß der Arbeit enthält Abschnitt 5.3 einen kurzen Ausblick auf die Zukunft des Projekts, sowie von FUSION und JAVA.

Zur Ergänzung der vorangegangenen Kapitel enthält der Anhang zusätzliche Referenzen. Die Anhänge A, B und D ergänzen und präzisieren die *Requirements*. Anhang A beschreibt die zulässige Syntax von Regeln und Fakten in SPIRIT, während das Format von SPIRIT-Dateien in Anhang B definiert wird. Die Liste der von SPIRIT gelieferten Fehlercodes befindet sich in Anhang C. Anhang D enthält eine Auswahl der verwendeten Algorithmen und Methoden. Anhang E stellt eine Referenz der Kommandos der SPIRIT-Shell SPECTER bereit. Schließlich erläutert Anhang F verwendete Fachtermini und neu eingeführte Datentypen in einem *Data-Dictionary* gemäß FUSION.

Zielgruppe der Arbeit

Ich habe mich bemüht, diese Arbeit kurz aber vollständig zu halten. Dem Leser soll es möglich sein, die in SPIRIT verwendeten Algorithmen und Konzepte zu verstehen und sie ggfs. selbst zu implementieren. Hierzu sind vor allem die Abschnitte 3.1 und Anhang D relevant. Ich gehe nur dort auf die allgemeine Theorie wissensbasierter Systeme ein, wo es zum Verständnis nötig ist. Der Leser sollte allerdings über Grundkenntnisse aus Wahrscheinlichkeits- und Graphentheorie verfügen. Als einführende Literatur zu SPIRIT und probabilistischen Expertensystemen allgemein möchte ich [Röd94], [MR92], [RM96] und [Pea88] nennen. Ein Anwendungsbeispiel zu SPIRIT mit Ausschnitten aus den bisherigen Programmversionen kann [KIMR96] entnommen werden.

Neben den speziellen Details des vorliegenden Projektes, möchte ich auch einen Erfahrungsbericht über die Entwicklungsmethode FUSION geben. Zu diesem Zweck enthält Abschnitt 5.1 eine Bewertung dieser Methode. Der interessierte Leser sollte dazu mit den Grundkonzepten des objektorientierten Softwareentwurfes vertraut sein. Eine kompakte und übersichtliche Einführung in die verwendete Methode FUSION befindet sich in [Mal95].

Schließlich möchte ich einen Eindruck von der Implementierungssprache JAVA geben. Dazu ist Kapitel 4 relevant. Kenntnisse dieser Programmiersprache werden allerdings nicht vorausgesetzt, da ich in dieser Arbeit nicht auf den Quelltext von SPIRIT eingehe. Als einführende Literatur zu JAVA sei hiermit auf [Fla96], [Hof96], [New96] und [Rit95] verwiesen.

Danksagung

An dieser Stelle möchte ich mich insbes. für die freundliche Unterstützung von Herrn C.-H. Meyer bedanken, der als geduldiger Ansprechpartner am Lehrstuhl in Hagen zur Verfügung stand und dessen konstruktive Anregungen, Anforderungen und Erläuterungen Grundlage der vorliegenden Arbeit waren.

Konventionen

- Alle im Entwicklungsprozeß verwendeten Bezeichner sind in englischer Sprache. Ich bin mir dessen bewußt, daß es durch diese Mischung der Sprachen oft zu unsauberen Formulierungen kommen kann. Zur Vermeidung umständlicher Formulierungen verwende ich englische Bezeichnungen dennoch gleichberechtigt mit deutschen Wörtern.
- Um Verwirrungen und neue Wortschöpfungen zu vermeiden, wurden auch die Bezeichnungen für die Modelle und Teilschritte von FUSION in ihrer Originalsprache belassen. Solche Begriffe werden meist *kursiv* dargestellt (z.B. *System Object Model*).
- Eigennamen von Programmen, Systemen und Firmen werden meist in KAPITÄLCHEN gedruckt (z.B. WINDOWS95)
- Namen von Klassen beginnen mit einem Großbuchstaben (z.B. Rule)
- Namen von neu definierten Datentypen beginnen ebenfalls mit einem Großbuchstaben (z.B.: VarType), nicht jedoch die i.a. vordefinierten Typen boolean, int und double.
- Namen von Variablen beginnen mit einem Kleinbuchstaben (z.B. rule)
- Namen von Objekten sind (wo möglich) an den Namen ihrer Klasse angelehnt (z.B. rule: Rule). Hierbei werden lange Klassennamen zumeist abgekürzt (z.B. var: Variable).
- Die Stelligkeit (*arity*) einer Variable v wird mit |v| abgekürzt.
- Bei der Erstellung des Programmcodes habe ich darauf geachtet, daß schnell von der vorliegenden Arbeit zum entsprechenden Code gewechselt werden kann. Der Code ist zumeist umfangreich dokumentiert, Variablen, Klassen und Methoden haben dieselben Bezeichnungen wie in diesem Text und der Quelltext ist ordentlich und einheitlich strukturiert.
- In Auflistungen von Funktionen, Attributen, etc. wurde eine alphabetische Ordnung gewählt, wenn sich aus dem semantischen Kontext keine sinnvollere Reihenfolge anbietet.

1 Aufgabenstellung, Problemanalyse und Methodik

Dieses Kapitel gibt in Abschnitt 1.1 zunächst eine extrem kompakte Einführung in die grundlegenden Konzepte und Ideen von SPIRIT aus Sicht eines Informatikers. Anschließend werden diese Konzepte in Abschnitt 1.2 in einem relativ präzisen Anforderungsdokument (den *Requirements*) genauer vorgestellt. Diese Anforderungen dienen gleichzeitig als Ausgangspunkt für den in der vorliegenden Arbeit beschriebenen objektorientierten Entwicklungsprozeß.

Nach der Formulierung der *Requirements* erfolgt ein Überblick über die bisherigen Versionen von SPIRIT, die daraus wiederverwertbaren Teile und die Motive für ein gründliches *Reengineering* (1.3.1). Abschnitt 1.3.2 geht auf die verwendete Methodik (FUSION) ein und erläutert die weitere Vorgehensweise im Entwicklungsprozeß, die sich mit der Verwendung von FUSION anbot.

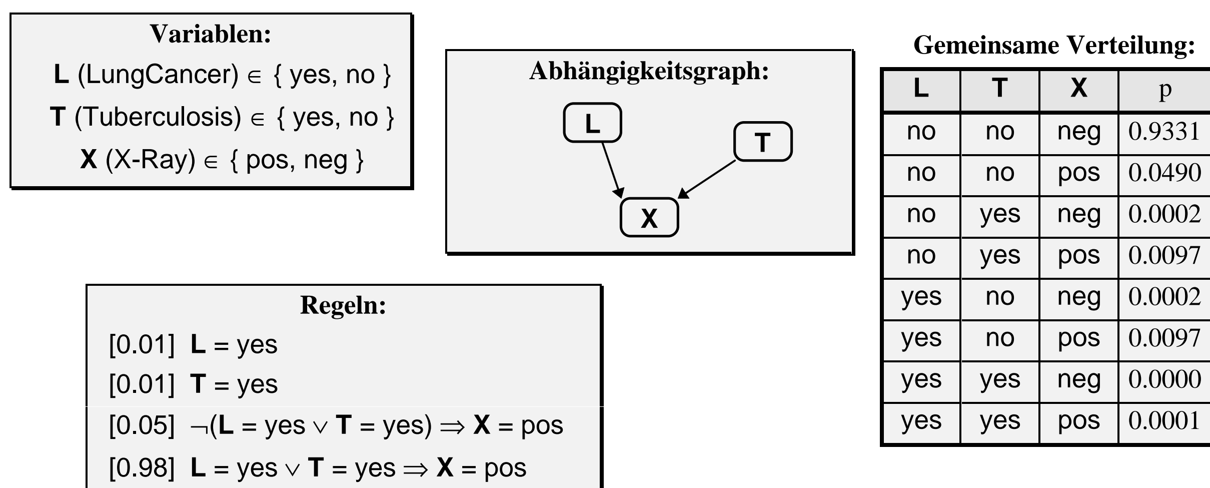
1.1 SPIRIT - eine kompakte Einführung

SPIRIT ist ein probabilistisches, wissensbasiertes, entscheidungsunterstützendes System:

- *probabilistisch*, weil es mit (normalerweise hochdimensionalen) Wahrscheinlichkeitsverteilungen arbeitet,
- *wissensbasiert*, weil es auf Regeln beruht, die Expertenwissen repräsentieren, und
- *entscheidungsunterstützend*, indem es ermöglicht, aus diesen Regeln (mehr oder weniger) sinnvolle Schlußfolgerungen zu ziehen.

Die Definition einer *Wissensbasis* (sh. Abbildung 1) wird von einem Experten (oder *Knowledge-Engineer*) vorgenommen. Dieser deklariert zunächst diskrete *Zufallsvariablen*, deren Wertebereiche die Zustände eines zu modellierenden Problembereiches repräsentieren. Zwischen den Variablen werden dann mithilfe sog. probabilistischer *Regeln* stochastische *Abhängigkeiten* aufgebaut, die in einem Graphen visualisiert werden können. Regeln sind unbedingte oder bedingte Wenn-Dann-Aussagen, deren umgangssprachliche Formulierung z.B. wie folgt aussehen könnte: Wenn ein Patient Krebs oder Tuberkulose hat, dann ist das Ergebnis der Röntgenuntersuchung mit Wahrscheinlichkeit 0.98 positiv. Nach der Eingabe solcher Regeln wird über einen Iterationsprozeß automatisch eine gemeinsame probabilistische *Verteilung* über die Variablen aufgebaut. An diese Verteilung können anschließend Anfragen gestellt werden. Um (im Beispiel unten) herauszufinden, welchen Einfluß ein negatives Röntgenergebnis auf die Wahrscheinlichkeit hat, daß ein Patient an Lungenkrebs erkrankt ist, wird der Variable X *Evidenz* zugewiesen, d.h. es wird angenommen, daß X=neg gilt. Dies beeinflußt die Randsummen der übrigen Variablen.

SPIRIT ist ein System zur Erzeugung und Verwaltung von derartigen Wissensbasen.



L	T	X	p
no	no	neg	0.9331
no	no	pos	0.0490
no	yes	neg	0.0002
no	yes	pos	0.0097
yes	no	neg	0.0002
yes	no	pos	0.0097
yes	yes	neg	0.0000
yes	yes	pos	0.0001

Abbildung 1: Eine einfache Wissensbasis zur Diagnose von Lungenkrebs

1.2 Die Anforderungen (Requirements)

Dieser Abschnitt umfaßt die Anforderungen, die das zu entwickelnde System SPIRIT erfüllen soll. Die Anforderungen entstanden im Dialog mit dem auftraggebenden Lehrstuhl und basieren auf den Erfahrungen mit den Vorgängerversionen. Gemäß den in [Som96, Kapitel 7] gewählten Begriffen handelt es sich hierbei um eine *Requirements Definition*, die im Gegensatz zu einer *Requirements Specification* allgemein verständlich und informal ist. Daß darin einige Formulierungen etwas schwammig sind liegt vor allem daran, daß die hier angegebenen *Requirements* in ihrem Umfang stark komprimiert sind (es wären sonst wesentlich mehr Seiten erforderlich gewesen). Teile der Anforderungen wurden in die Abschnitte A, B und D des Anhangs verlagert. Sie werden außerdem in der Analysephase in Kapitel 2 wesentlich präzisiert.

Leider ist es im gegebenen Rahmen nicht möglich, eine allgemeine Einführung in die Theorie probabilistischer Wissensbasen zu geben. Möglicherweise sind dadurch einige Zusammenhänge ohne Kenntnis der Vorgespräche, die zur Aufstellung der *Requirements* geführt haben, schwer verständlich. Viele zugrundeliegende theoretische Konzepte von SPIRIT werden allerdings in Abschnitt 3.1 noch ausführlich behandelt.

Zu entwickeln ist eine plattformunabhängige, effiziente und erweiterbare Klassenbibliothek (im folgenden: *SPIRIT*) zur Erzeugung, Verwaltung und Befragung von probabilistischen Wissensbasen. SPIRIT soll leicht von graphischen Benutzeroberflächen (im folgenden: *Shells*) eingebunden werden können und eine überschaubare, aber ausreichende Menge von Funktionen als Schnittstelle bereitstellen. Das primäre Optimierungsziel ist die Ablaufgeschwindigkeit.

Die *Wissensbasis* eines probabilistischen Expertensystems wird erzeugt durch

- eine endliche Menge von Variablen,
- eine endliche Menge von probabilistischen Regeln und Fakten, die Beziehungen zwischen den Variablen herstellen,
- einen ungerichteten und einen gemischten Graphen zur Visualisierung dieser Beziehungen, und
- eine azyklische Struktur von Randverteilungen zur Speicherung, Änderung und Befragung der jeweils gültigen bedingten Wahrscheinlichkeiten (sog. LEG-Wald).

Eine Wissensbasis soll dupliziert werden können, damit auf einer Kopie temporäre Berechnungen (z.B. komplexe Anfragen) ausgeführt werden können, ohne die Original-Wissensbasis zu ändern.

Wissensbasen sollen in dem in Anhang B definierten SPIRIT-Dateiformat gespeichert und geladen werden können. Beim Laden sollen die in der Datei befindlichen Variablen und Regeln zu einer bestehenden Wissensbasis hinzugenommen werden, um somit die Kombination verschiedener Wissensbasen zu ermöglichen.

1.2.1 Variablen

Variablen beschreiben einen durch die Wissensbasis zu modellierenden Teil eines Problembereiches und können in SPIRIT einen der in Tabelle 1 aufgeführten Typen haben.

Tabelle 1: Die Variablentypen in SPIRIT

Variablentyp	zulässige Wertemenge	Operatoren
Boolean	vordefiniert: 0 und 1 (für *false* und *true*)	(is *true*)
Nominal	benutzerdefinierte, endliche, nichtleere Menge diskreter Werte	$=, \neq, \in, \notin$
Number	benutzerdefinierte, endliche, nichtleere Menge rationaler Werte	$=, \neq, <, >, \in, \notin$
Interval	benutzerdefinierte Intervallgrenzen (endliche rationale Zahlen)	$<, >$

Bei Variablen der Typen *Nominal* und *Number* müssen alle zulässigen Werte vor der Nennung in Regeln explizit deklariert werden. Bei Variablen vom Typ *Interval* müssen entsprechend die Intervall-Grenzen vordefiniert werden.

Variablen können dynamisch hinzugefügt, gelöscht und geändert werden. Mit einer Variable werden alle Regeln gelöscht, die ein Vorkommen dieser Variable enthalten. Mit einem Variablenwert werden alle Regeln gelöscht, die auf diesen Wert zugreifen müssen. Variablen sollen umbenannt werden können.

Zu jedem Variablenwert soll die aktuell gültige bedingte Wahrscheinlichkeit (gemäß Algorithmus VarValueActProb) ermittelt werden können. Zu allen Variablen soll ein Erwartungswert (gemäß Algorithmus VarValueExpected) errechenbar sein. Hierzu kann Werten von *Boolean*- und *Nominal*-Variablen eine reelle Zahl als Nutzen zugewiesen werden (das sog. *Utility*), die in der Formel verwendet wird.

1.2.2 Regeln und Fakten

Regeln werden als Strings deklariert, die gemäß Anhang A logische Aussagen über die Variablen machen. Jeder Regel wird bei der Deklaration eine vorgegebene bedingte Wahrscheinlichkeit oder ein Gültigkeitsintervall zugewiesen. Diese Werte sollen jederzeit geändert werden können.

Fakten sind Regeln mit einer Tautologie als Prämisse. Regeln mit einer vorgegebenen bedingten Wahrscheinlichkeit von 0 oder 1 heißen *sicher*. SPIRIT soll Funktionen zum Edieren und Verwalten von Regeln bereitstellen.

Zu jeder Regel sind weiterhin folgende Informationen interessant:

- eine Darstellung der Regel in kanonischer Disjunktiver Normalform (DNF),
- die aktuell gültige bedingte Wahrscheinlichkeit (gemäß Algorithmus RuleActProb),
- die aktuell gültige bedingte Prämissenwahrscheinlichkeit (gemäß Algorithmus RuleActPremProb),
- der durch die Lernprozesse ermittelte Lagrange-Parameter α (sh. Algorithmus RuleLearnIteration), und
- ein *Aktivierungszustand* $\in$ {*aktiv*, *aktivierbar, passiv*} (hierzu später mehr).

Jeder Variable, Regel und Wissensbasis sollen zusätzliche (*User*-) Informationen frei zugeordnet werden können, die zusammen mit dem Objekt gehalten und gespeichert werden (z.B. Bildschirmkoordinaten oder Kommentare).

1.2.3 Graphen auf der Variablenmenge

Das Vorkommen von Variablen in Regeln definiert folgende Graphen auf der Variablenmenge:

- *ungerichteter Graph*: Zwischen zwei Variablen befindet sich eine Kante, genau dann wenn beide Variablen in einer gemeinsamen Regel vorkommen.
- *gemischter Graph*: Zwischen zwei Variablen befindet sich eine ungerichtete Kante, wenn beide Variablen gemeinsam in der Conclusio einer Regel vorkommen. Zwischen zwei Variablen A und B befindet sich ein Pfeil von A nach B, wenn A in der Prämisse einer Regel vorkommt, deren Conclusio die Variable B enthält.

SPIRIT soll Funktionen zur Darstellung dieser Graphen bereitstellen. Hierzu zählt eine Funktion zur Ermittlung der Kantenstärken (gemäß Algorithmus VarGraphEdgeThickness).

1.2.4 LEG-Wälder

Zur Speicherung der aktuellen bedingten Wahrscheinlichkeiten der Variablen baut eine Wissensbasis eine gemeinsame Wahrscheinlichkeits-Verteilung auf. Eine effiziente Form der Speicherung dieser Verteilungen ist die Form eines *LEG-Waldes*. Eine *LEG* (*Local Event Group*) besteht aus einer nichtleeren Variablenmenge und einer Tabelle. Die Variablenmenge gibt die Variablen an, deren bedingte Wahrscheinlichkeiten in der LEG gespeichert werden. Die Tabelle enthält zu jeder *Konfiguration* (gemäß dem kartesischen Produkt) über die Variablenmenge eine positive reelle Zahl. Diese Einzelwahrscheinlichkeiten summieren sich in konsistentem Zustand zu 1 auf.

In SPIRIT müssen die LEGs eine Baumstruktur bilden, die die sog. *Junction-Tree-Property* [JJ94, S.360] erfüllt. Dieser LEG-Wald kann auf Wunsch neu aufgebaut werden (Algorithmus LEGForestBuild). Zum Neuaufbau des LEG-Waldes wird eine Eliminationsreihenfolge der Variablen benötigt. Zur Erzeugung dieser Reihenfolgen sollen verschiedene heuristische Algorithmen (VarEnum... gemäß Anhang D) implementiert werden. Es soll der Shell aber auch ermöglicht werden, eigene Eliminationsfolgen vorzugeben (die z.B. über externe Programme ermittelt wurden). Wird keine Eliminationsreihenfolge angegeben, so gilt eine willkürliche Reihenfolge. Nach dem Neuaufbau soll zunächst Gleichverteilung eingerichtet werdern. Sollen die zuvor gültigen Randwahrscheinlichkeiten wiederhergestellt werden, so muß von der Shell ein sog. *Re-Init* aufgerufen werden (Algorithmus LEGForestReinit).

Das Verfahren zum Neuaufbau des LEG-Waldes kann scheitern, wenn die Größe einer LEG eine vorgegebene Obergrenze überschreitet. In diesem Fall muß die zuvor bestehende Struktur wiederhergestellt werden. Eine ähnliche Situation kann entstehen, wenn der Wertebereich einer Variable durch Hinzunahme weiterer Variablenwerte vergrößert wird. Da sich diese Vergrößerung direkt auf die LEGs auswirkt, die die Variable enthalten, kann es sein, daß eine LEG zu groß wird. Dann muß die Hinzunahme des Variablenwertes als unzulässig zurückgewiesen werden.

Jede *LEG* ist anfangs gleichverteilt und soll an Änderungen der Variablenmenge (z.B. Löschen von Variablenwerten) möglichst ohne Informationsverlust angepaßt werden. Jede Variable muß immer in mindestens einer LEG auftauchen.

Zum Lernen muß jeder Regel eine LEG zugeordnet werden. Eine Regel *paßt* in eine LEG, wenn die Menge der in der Regel vorkommenden Variablen eine Untermenge der Variablen der LEG ist. Regeln, die in keine LEG passen, erhalten den Zustand *passiv*, während jeder *aktiven* Regel die kleinste passende LEG zugewiesen wird.

Nur *aktive* Regeln sind für Lernschritte zulässig. Der Aktivierungszustand einer Regel kann (soweit möglich) auch explizit gesetzt werden. Der Zustand *aktivierbar* besagt, daß die Regel zwar in eine LEG paßt, aber nicht gelernt werden soll.

Die zuvor deklarierten Regeln können in passende *LEG-Bäume* gelernt (hineinpropagiert) werden. Hierzu dient ein Iterationsprozeß über alle aktiven Regeln und LEGs gemäß Algorithmus RuleLearnIteration. Dieser Prozeß (im folgenden oft mit *Iteration* bezeichnet) soll schrittweise überwacht werden können, da er recht lange oder sogar unendlich lange laufen kann. Er besteht aus lokalen und globalen Iterationen.

Eine globale Iteration kann abgebrochen werden, wenn die relative Entropie zwischen den Verteilungen vor und nach einem vollständigen Durchlauf nahe 0 ist. Im Falle einer widersprüchlichen Regelmenge kann es allerdings sein, daß die Iteration nicht terminiert. Daher werden Informationen über den Fortschritt der Iteration benötigt. Insbesondere sollen folgende Werte nach jedem Iterationsschritt zugänglich sein:

- die nächste zu lernende Regel (falls in der aktuellen LEG mind. eine aktive Regel ist),
- die relative Entropie der letzten Regelanwendung,
- die laufende Entropie (beginnend mit der absoluten Entropie wird jeweils die relative Entropie substrahiert),
- die Anzahl lokaler und globaler Iterationen, und
- zu jeder Regel: die Summe der bewirkten relativen Entropien, die Anzahl der Verwendungen in Lernschritten.

Als Option zur Iteration kann eine ε-Schwelle für die Entropieänderungen und eine Maximalzahl lokaler Iterationen angegeben werden.

Während des Iterationsvorganges dürfen keine Änderungen an der Struktur der Wissensbasis vorgenommen und keine Anfragen oder Evidenzzuweisungen durchgeführt werden, da sich die Werte in den Tabellen der LEGs möglicherweise nicht zu 1 aufsummieren und somit keinen konsistenten Zustand beschreiben.

Jeder Zufallsvariable kann gemäß Algorithmus AssignEvidence Evidenz zugewiesen werden. Dabei wird einer ihrer Werte vorgegeben und die LEG-Tabellen werden so abgeändert, daß die Wahrscheinlichkeit dieses Wertes 1 wird. Die Werte von Variablen vom Typ *Interval* sind hierbei die Intervalle selbst. Nach der Zuweisung von Evidenz können die Auswirkungen dieser Änderung auf die übrigen bedingten Wahrscheinlichkeiten ermittelt werden. Evidenzzuweisungen müssen möglichst schnell durchgeführt und zurückgesetzt werden können. Sind Evidenzen zugewiesen worden, so sind keine strukturellen Änderungen an der Wissensbasis zulässig.

Eine andere Möglichkeit von Anfragen an die Wissensbasis sind *komplexe Anfragen*. Hierbei wird zu einem gegebenen Wissensstand eine zusätzliche Menge von Regeln gelernt und eine weitere Regelmenge als Anfragemenge benutzt. Diese Anfragen können bereits mit den oben beschriebenen Anforderungen durchgeführt werden und erfordern keine weiteren Funktionen.

Die LEG-Struktur soll auf Gleichverteilung gesetzt werden können. Hierbei werden alle Evidenzen zurückgenommen und die Lagrange-Parameter (α) der Regeln auf 1 zurückgesetzt. Außerdem soll die absolute Entropie der Verteilung errechnet werden können (Algorithmus LEGForestAbsEntropy).

Da die Größe der LEGs großen Einfluß auf die Antwortzeit und die Iterationsgeschwindigkeit des Systems hat, soll die Shell Informationen über die LEG-Struktur erfragen können.

1.2.5 Erweiterungen

Für SPIRIT sind mittelfristig folgende Erweiterungen geplant, die für den Entwurf zu bedenken sind:

- Nachträgliches Hinzufügen von Unabhängigkeitsbedingungen, mit denen der Import von Bayes-Netzen und deren Verallgemeinerung möglich wird.
- Erweiterungen um die beiden Klassen Entscheidungs- und Nutzen-Variablen.
- Hinzunahme stetiger Zufallsvariablen.
- Verfahren zur automatischen Gewinnung von Struktur-Informationen aus Falldaten.
- Schätzung von bedingten Wahrscheinlichkeiten der Regeln aus einer Menge von Fallbeispielen.

1.3 Neuentwicklung und Reengineering

Die in diesem Dokument beschriebene Klassenbibliothek SPIRIT ist ein Zwischenschritt eines mehrjährigen Forschungsprogrammes unter der Leitung von Prof. Dr. W. Rödder am Lehrstuhl für BWL, insbes. Operations Research an der FernUni Hagen. Zwischen September 1993 und der Entstehung der vorliegenden Arbeit im Wintersemester 1996/97 war ich an diesem Lehrstuhl als studentische Hilfskraft beschäftigt und als Software-Entwickler an der Entstehung verschiedener experimenteller Vorgängerversionen beteiligt. Meine Aufgabe bestand darin, die von den Wissenschaftlern erstellten theoretischen Konzepte in Software umzusetzen. Hierzu mußte ich mich in die (mir fachfremde) Thematik einarbeiten und zahlreiche Konzepte, Datenstrukturen und Algorithmen entwickeln, die in dieser Form noch nicht bestanden. An einigen Stellen konnten meine Erfahrungen bei der Programmentwicklung konzeptionelle Unklarheiten offenlegen und Verbesserungsvorschläge einbringen. Somit lag selten die klassische Konstellation zwischen Auftraggeber und Software-Entwickler vor, sondern eher ein Dialog, von dem beide Seiten profitierten.

Der folgende Abschnitt 1.3.1 gibt eine Übersicht über die bisher entwickelten Versionen und motiviert das sich aufdrängende *Reengineering* zu der vorliegenden Version. Anschließend wird die weitere Vorgehensweise erläutert und dabei insbesondere auf die objektorientierte Entwicklungsmethode FUSION eingegangen (Abschnitt 1.3.2). Ich gebe im Laufe dieses Abschnittes außerdem an, welche Teile des Projektes von mir neu entwickelten wurden und welche aus anderen Quellen und den Vorgängerversionen übernommen werden konnten.

1.3.1 Die Vorgängerversionen und Gründe für das Reengineering

Zur Sammlung erster Erfahrungen auf dem Gebiet probabilistischer Wissensbasen begann meine Tätigkeit mit der Entwicklung einer *Demo-Version* von SPIRIT, in der verschiedene Techniken zur Repräsentation und Akquisition von Wissen in einer gemeinsamen Verteilung über diskrete Variablen erprobt wurden. Zu dieser Demo-Version gehörten vier Teilprogramme, an denen Nützlichkeit und Eignung der probabilistischen Methoden für verschiedene Anwendungsbereiche ausgeleuchtet werden sollten (darunter war z.B. ein selbständig lernendes Spielprogramm). Allerdings waren die eingesetzten Methoden vergleichsweise primitiv und ergaben keinen wiederverwertbaren Code.

Nach der Demo-Version folgte die erste allgemein einsetzbare *Version 1*. Mit dem Fortschritt der Forschung am Lehrstuhl konnten hier deutlich verbesserte Technologien eingesetzt werden. Um eine allgemein verwendbare Funktionssammlung aufzubauen, wurden ab dieser Version Oberfläche und Systemkern getrennt. Nach der Fertigstellung von Version 1, die noch von einer Person (komplett in C++) geschrieben wurde, wurden Oberflächen- und Systemprogrammierung auf verschiedene Programmierer verteilt. Dabei wurde die visuelle Programmier-Umgebung BORLAND DELPHI zur Entwicklung einer graphischen Benutzeroberfläche eingesetzt, so daß die Shell für *Version 2* in einer anderen Programmiersprache als der Systemkern geschrieben wurde.

Die Schnittstelle zwischen Systemkern und Shell wurde über ca. 100 Funktionen geregelt. Diese Funktionen durften aus technischen Gründen nur Argumente in atomaren Datentypen erhalten und griffen erst intern auf die Objekte der Wissensbasis zu. Diese Objekthierarchie bestand aus etwa 20 Klassen, die teilweise beträchtliche Größe hatten. (Ein Ausschnitt dieses Klassenmodells wird später im Abschnitt 5.2.1 mit der neuentwickelten Struktur verglichen.)

Der Systemkern wuchs von Version zu Version und erfuhr während seiner Entwicklung zahlreiche, zum Teil gravierende konzeptionelle Änderungen. Durch den Fortschritt der parallel zur Programmierung stattfindenden Forschungsaktivitäten

- stellten sich Methoden und Funktionen als Sackgasse, bzw. als überflüssig heraus,
- konnten viele Anforderungen nicht präzise formuliert werden,
- ergaben sich neue Anforderungen aus den Erfahrungen mit den implementierten Versionen, und
- erwiesen sich einige neue theoretische Methoden als effizienter, als die im Programm verwendeten.

All dies führte dazu, daß die Software nicht sauber konzipiert, implementiert und dokumentiert werden konnte, sondern recht unsystematisch entstand. Dies kann und sollte in einem solchen Forschungsprojekt auch nicht verhindert werden, da Forschung zu großen Teilen aus mehr oder weniger gelungenen Versuchen besteht. Die jeweils neuen Anforderungen ergaben sich kurzfristig und waren oft nur unzureichend ausgereift, bzw. wurden bald durch neue Methoden ersetzt. Änderungen wurden meist direkt in den Code eingebaut, ohne die implizierten konzeptionellen Änderungen für andere Programmteile zu beachten. Mit der Zeit wurde es immer schwieriger, den Systemkern auf neue Probleme einzustellen und gleichzeitig eine fehlerfreie Funktionsweise seiner alten Module zu garantieren. Auch tiefgreifende konzeptionelle Änderungen, bzw. Erweiterungen wurden unmöglich. Statt dessen blähte sich der Code immer weiter auf und wurde unübersichtlich. Schließlich bestand der Systemkern von Version 2 aus 35 Modulen mit insgesamt über 11000 Zeilen Quelltext.

Die Sprache C++ erwies sich als für ein solches Projekt ungeeignet und fehlererzeugend, insbesondere aufgrund der umständlichen und unübersichtlichen Speicherverwaltung mit Zeigern. Ein großer Teil des Programmcodes enthielt technisch erforderliche Implementierungsdetails, die mit der eigentlichen Funktionalität des Systems wenig zu tun hatten und sehr schwer auffindbare Fehler provozierten. Auch fehlte es an plattformunabhängigen Tool-Klassenbibliotheken, um SPIRIT auf anderen Betriebssystemen lauffähig zu machen. Die gleichzeitig entwickelte graphische Oberfläche war ohnehin völlig an das Betriebssystem WINDOWS 95 gebunden. Ein weiteres Problem bei der Verwendung von C++ war die Mischung von objektorientierten und klassischen imperativen Sprachelementen. So gab es in den Vorgängerversionen eine Reihe globaler Funktionen, die aus jedem Objekt aufgerufen werden konnten, gleichzeitig aber den Zustand anderer Objekte ändern konnten. Aus Bequemlichkeit wurden die Zugriffspfade der Objekte häufig über diese globalen Funktionen geregelt, wodurch die Klassenhierarchie umgangen wurde. Weitere negative Eigenschaften der Vorgängerversionen werden später in Abschnitt 5.2 aufgeführt. Ein nicht zu unterschätzender Grund gegen C++ ist ferner die lange Compilier- und Linkzeit des Codes.

Eine positive Motivation für die Neuentwicklung war, daß es den am Projekt beteiligten Wissenschaftlern gelang, das theoretische Fundament soweit abzusichern, daß es möglich wurde, eindeutige *Requirements* mit entsprechenden Algorithmen zu formulieren. Hierdurch konnte das Projekt auf sauberer Grundlage neu entworfen werden.

Folgende Ziele sollten mit der Neuentwicklung von SPIRIT erreicht werden:

- eine sauber konzipierte, fehlerfreie, leicht erweiterbare und plattformunabhängige Klassenbibliothek,
- die Implementierung neuer, effizienter Algorithmen und Datenstrukturen, und
- eine ordentliche Dokumentation für System- und Anwendungsprogrammierer.

1.3.2 Vorgehensweise im Entwicklungsprozeß

Diese Arbeit beschreibt Vorgehensweise und Ergebnisse des Entwicklungsprozesses für das Projekt SPIRIT. Im Gegensatz zu den Vorgängerversionen wurde die aktuelle Version nach softwaretechnologischen Methoden entworfen und implementiert. In der Software-Technik (oder *Software-Engineering*) werden verschiedene Entwicklungsmethoden verwendet, um von einem abstrakten Anforderungsdokument zu einem ausführbaren Programm zu gelangen. Die meisten dieser Methoden schlagen die schrittweise Erstellung und Verfeinerung von Analyse- und Entwurfsmodellen vor, die schließlich zu einer implementierbaren Struktur führen.

In SPIRIT wurde die objektorientierte Methode FUSION verwendet. FUSION wurde in den HEWLETT-PACKARD-LABORATORIES in Bristol/England entwickelt und insbes. durch das Buch [Col94] bekannt. Ich habe mich bemüht, den in diesem Lehrbuch erläuterten Schritten möglichst genau zu folgen, um FUSION abschließend gerecht beurteilen zu können. Wie die meisten anderen Entwicklungsmethoden, bleibt FUSION an einigen Stellen allerdings recht unscharf und läßt Freiräume, die nach eigenem Ermessen ausgefüllt werden können. Im Projekt SPIRIT habe ich diese Freiräume dann genutzt, wenn mir die Syntax der Modelle und Schemata unnötig kompliziert erschien. Anstelle von präzisen und platzraubenden Definitionen finden sich in dem vorliegendem Entwurfsdokument also häufig "lockere" Beschreibungen in Textform und mit Abbildungen. Oft werden auch Details ausgelassen, die für das Verständnis der Probleme nicht unbedingt nötig sind. Dennoch wurde darauf geachtet, daß alle notwendigen Informationen und Entwurfsergebnisse in der Arbeit genannt werden. Damit sollte es dem Leser ohne großen Aufwand möglich sein, ein System wie SPIRIT oder Teile davon selbst zu implementieren.

Mein persönlicher Anteil an diesem Projekt ist schwer abgrenzbar. Meine primäre Aufgabe bestand natürlich in der Umsetzung der theoretischen Konzepte, die vom Lehrstuhl erarbeitet wurden. Allerdings hatte ich dazu praktisch alle Freiheiten und führte die Arbeit hauptsächlich selbständig durch. Auch arbeitete ich als *externer* Mitarbeiter und hatte zumeist nur (wöchentlichen) telephonischen Kontakt zum Lehrstuhl. Die neuen Programmversionen u.ä. schickte ich über das Internet von Ulm nach Hagen. Da meine Aufgabensteller zudem Nicht-Informatiker waren, blieben mir alle Entscheidungen bzgl. der im Projekt verwendeten Entwicklungsmethode selbst überlassen und ich erhielt keine Anweisungen zur Auswahl und Entwicklung bestimmter Algorithmen und Datenstrukturen. Somit wurden alle Entwurfsentscheidungen (hier in den Kapiteln 2 und 3) von mir getroffen. Da auch die Systemkerne der Vorgängerversionen von mir entwickelt wurden, konnte ich wenig auf andere Quellen zurückgreifen. Von den im Anhang D erwähnten fachspezifischen Algorithmen sind die meisten informal vom auftraggebenden Lehrstuhl vorgeschlagen worden. Deren Umsetzung, saubere Formulierung und Verbesserung blieb mir überlassen. Hierzu waren lange Gespräche mit dem projektleitenden Lehrstuhl und die im Literaturverzeichnis angegebene Literatur hilfreich. Bei manchen Algorithmen konnte ich während der vergangenen Zeit auch kleinere Optimierungen anbringen (z.B. eine Beschleunigung des Iterationsalgorithmus um durchschnittlich etwa ein Fünftel, durch Einführung des dritten Kriteriums für die Rest-Liste im Algorithmus RuleLearnIteration (sh. Anhang D)).

Auch die neu erschienene und umstrittene Programmiersprache JAVA wurde von mir zur Implementierung vorgeschlagen. Wie sich erst im späteren Verlauf der Arbeit herausstellte (es gab anfangs kaum Tools und keine effizienten Compiler), erwies sich JAVA als Glücksgriff: JAVA ist dabei sich auf dem Markt zu etablieren und scheint auch unter Fachleuten größere Anerkennung zu erlangen. So findet sich in einer Pressemitteilung der Firma HEWLETT PACKARD (für die FUSION entwickelt wurde) vom 7. Oktober 1996 eine Anmerkung, daß die kommende Generation der Methode FUSION selbst um eine JAVA-Unterstützung angereichert werden soll.

FUSION geht von einem Anforderungsdokument aus, das über einen Dialog zwischen Programmentwickler und Aufgabensteller aufgebaut wird. Die *Requirements* für das vorliegende Projekt ergaben sich aus längeren Gesprächen mit dem Auftraggeber und beruhten zu großen Teilen auf den Erfahrungen mit den Vorgängerversionen. Entsprechend war es naheliegend, den alten Programmcode nach relevanten Funktionen, Konzepten und Algorithmen zu durchforsten. Die in Abschnitt 1.2 beschriebenen Anforderungen konnten daher vergleichend auf Vollständigkeit und Konsistenz überprüft werden. Dennoch wurden die *Requirements* im Laufe der Entwicklung (insbes. bis zum Anfang der *Design-Phase*) an einigen Stellen erweitert, geändert und präzisiert.

Hierbei erwies sich die *Analysephase* als besonders ergiebig. In FUSION ist die Aufgabe der *Analysephase* die Festlegung, welche Dienste das zu entwickelnde System *nach außen* bereitstellen soll. Hierzu werden im wesentlichen ein erstes *Klassenmodell* und eine Spezifikation der *System-Operationen* aufgebaut. In der *Analysephase* wurden hierzu zunächst sowohl Anforderungsdokument, als auch die Klassenhierarchien der Vorgängerversionen nach potentiellen Objektklassen durchsucht. Hierbei stellte sich heraus, daß nur sehr wenige Klassen nach außen sichtbar sein müssen und ein großer Teil der Datenstrukturen statt dessen nur interne Funktionen erfüllt. Daher entstand ein recht einfaches *System Object Model*. Nach dem Entwurf dieses Klassen-Modells wurden die erforderlichen System-Operationen definiert. Hierbei wurden die Anforderungen noch einmal mit dem Code der Vorgängerversionen verglichen, um eine vollständige aber minimale Funktionssammlung aufzubauen. Gleichzeitig wurden die System-Operationen in einen allgemeinen *Life-Cycle* eingebaut, der zulässige Verwendungsmöglichkeiten der Operationen vorführt.

Nach der *Analysephase* folgt in FUSION die *Design-* (oder Entwurfs-) Phase. In dieser Phase sollen die abstrakten Spezifikationen aus der vorangegangenen Phase in eine implementierbare Klassenstruktur gebracht werden. Im Projekt SPIRIT war es nicht möglich, die *Analysephase* definitiv von der Entwurfsphase abzugrenzen. Statt dessen wurden mit der Design-Phase neue Probleme und Unvollständigkeiten aufgedeckt, die eine Überprüfung und Änderung des Analyse-Modells erforderten.

FUSION schlägt in der Entwurfsphase die Erstellung von vier Modellen vor. Um in diesen Modellen eine möglichst hohe Präzision zu erreichen, war es in diesem Projekt zunächst erforderlich, die Klassenhierarchie aus der Analyse um interne Konzepte zu erweitern. Daher wurde FUSION um einen umfangreichen Zwischenschritt zur Erweiterung des Analyse-Modells ergänzt. In diesem Zwischenschritt wurden zahlreiche Datenstrukturen entwickelt, die teilweise auf den Erfahrungen aus den Vorgängerversionen basierten. Nachdem diese system-internen Fragestellungen geklärt wurden, konnten die folgenden Entwurfsmodelle relativ schnell entworfen werden.

Das erste Ergebnis der Entwurfsphase sind die sog. *Object Interaction Graphs* (OIGs). Diese visualisieren den internen Ablauf der System-Operationen und die dazu erforderlichen Nachrichtenströme zwischen den Objekten. Für diese Graphen wird also festgelegt, wie die Funktionen intern arbeiten und welche Informationen sie dazu von anderen Objekten benötigen. Am Aufbau der OIGs stellt sich heraus, wie geschickt die Überlegungen in den vorangegangenen Phasen und wie geeignet die entworfenen Klassenstrukturen sind. Bei SPIRIT zeigte sich, daß die Entwicklung der OIGs ohne den Zwischenschritt der Erweiterung des Analyse-Modelles nicht möglich gewesen wäre. Durch diese Erweiterungen waren allerdings viele wichtige Entscheidungen getroffen, so daß sich die OIGs fast direkt ergaben. Dennoch dauerte ihre Erstellung sehr lange, da viele Überlegungen bis ins Details getrieben werden mussten und auf keinen Fall Lücken und Unschärfen gelassen werden sollten. Schließlich hatte ich mir das (ehrgeizige) Ziel gesteckt, den Entwurf so detailliert wie möglich auf die Spitze zu treiben, um die Implementierung zu vereinfachen. Da diese Detailtreue auf Anhieb sehr schwierig zu erreichen war, wurden die Graphen in den nachfolgenden Schritten des Entwicklungsprozesses immer wieder kleineren Anpassungen unterworfen.

Die *Visibility Graphs* (VG), die FUSION als nächstes Ergebnis vorschlägt, stellen für jede Objektklasse dar, welche Zugriffe auf die anderen Klassen nötig sind. Dadurch soll u.a. geklärt werden, wie die Beziehungen zwischen den Objekten des *Object Models* (die sich in gewisser Weise "zwischen" den Klassen befinden) auf die einzelnen Klassen umgesetzt werden. Im vorliegenden Projekt wurden die Sichtbarkeitsrelationen bereits in den OIGs berücksichtigt, so daß sich die Erstellung der VGs als recht trivial erwies und wenig neue Erkenntnisse brachte.

Die Design-Phase endet mit dem wichtigsten Schritt, der Erstellung der *Class Descriptions*. Diese fassen die gesamten Ergebnisse der übrigen Schritte zusammen und geben einen codierten Rahmen für die spätere Implementierung. Zeitgleich mit den Klassenbeschreibungen werden in FUSION üblicherweise

Vererbungshierarchien ermittelt und in *Inheritance Graphs* abgebildet. Die Klassenhierarchie in SPIRIT wurde bereits in den vorherigen Schritten bedacht und erforderte an dieser Stelle wenig Aufwand. Dennoch wurden hier zur weiteren Strukturierung des Modells einige neue Klassen eingeführt.

Der für den Erfolg eines Projekts natürlich wichtigste Schritt ist die *Implementierung* der vorangegangenen Entwürfe. Eingabe zu dieser Phase sind i.w. die *Class Descriptions* und die *Object Interaction Graphs*. Aus diesen Modellen läßt sich im Idealfall sehr schnell lauffähiger Quelltext entwickeln. Während und nach der Implementierung müssen die entstehenden Programmteile auf Korrektheit (und Effizienz) geprüft werden.

Im Projekt SPIRIT verlief der Schritt der Implementierung überraschend schnell und problemlos. Nach der Fertigstellung der Klassenhierarchie konnte ich den Hauptteil des Codes (ca. 12000 Zeilen Quelltext) innerhalb von nur zwei (sehr "intensiven") Wochen erstellen. Nach der Erstellung dieses Gerüstes (und Systemkernes) begann allerdings eine sehr aufwendige *Testphase*, in der zwei komplette Benutzeroberflächen als Testtreiber entstanden. Zum einen entwickelte ich eine einfache aber flexible und leistungsstarke Kommandozeilen-Shell (SPECTER), mit der die einzelnen Funktionen der SPIRIT-Bibliothek direkt geprüft und Geschwindigkeitstests durchgeführt werden können. Der zweite Testtreiber (SPIRITSHELL) ist eine graphische Oberfläche, die eine einfache und allgemeine Wissensbearbeitung ermöglicht und nicht nur für Fachleute konzipiert ist. Mit diesen beiden Testprogrammen wurden daraufhin zahlreiche Tests durchgeführt (auch Praxistests von Mitarbeitern des Lehrstuhls und externen Interessenten: die SPIRITSHELL konnte über das Internet direkt aus einem *WWW-Browser* gestartet werden und war somit weltweit und plattformunabhängig zugänglich). Die Testphase wurde auch dazu benutzt, um alternative Algorithmen zu testen, deren Eigenschaften schlecht theoretisch gemessen werden konnten. Dabei gelang die Optimierung einiger Algorithmen.

FUSION schlägt als weiteres Ergebnis des Entwicklungsprozesses ein sog. *Data-Dictionary* vor. Hier sollen alle wichtigen Bezeichner der einzelnen Modelle referenziert werden können. Sinn dieses Dictionaries ist vor allem, die Arbeit für den Außenstehenden lesbarer und den Austausch zwischen verschiedenen Entwicklern einfacher zu machen. Entgegen der vorgeschlagenen Bearbeitungsreihenfolge habe ich allerdings das *Data-Dictionary* in diesem Projekt nicht während des Entwicklungsprozesses aufgebaut und gepflegt, da dies für mich (als einzigen Entwickler) wenig Nutzen, sondern lediglich viel Aufwand mit sich gebracht hätte. Statt dessen habe ich erst nach Fertigstellung der entgültigen Version von SPIRIT mit dem Aufbau eines kleineren *Data-Dictionaries* begonnen.

Die vorliegende Arbeit kann als Einführung von Systemprogrammierern (insbes. meinen Nachfolgern im Projekt) angesehen werden. Für die Programmierer von Anwendungen, die SPIRIT verwenden, sind die meisten hier aufgeführten Details uninteressant. Daher entstanden neben der Klassenbibliothek SPIRIT zwei Handbücher für Anwendungsprogrammierer. Das Programmierhandbuch gibt eine Einführung in die grundlegenden Konzepte und die verfügbaren Operationen, während das Referenzhandbuch eine Übersicht über alle Klassen, Methoden und Attribute von SPIRIT bietet.

Übrigens standen mir zur Programmentwicklung keine computergestützten *Software-Engineering- (CASE-) Tools* zur Verfügung, so daß ich die in Analyse und Design erforderlichen Modelle komplett mit einem im PC-Bereich weit verbreiteten Textverarbeitungssystem entwerfen mußte. Dies war insbesondere bei der Erstellung der aufwendigen *Object-Interaction-Graphs* und Klassenmodelle recht umständlich. Der Einsatz geeigneter integrierter Werkzeuge hätte den für die Entwicklung benötigten Zeitraum vermutlich wesentlich verkürzt. Für die Implementierung hingegen setzte ich mit SYMANTEC CAFÉ ein recht gelungenes JAVA-Programmpaket ein, das zwar einige Kinderkrankheiten aufwies, aber dennoch einen großen Anteil an der flotten Implementierung hatte.

Die beschriebenen Schritte des Entwicklungsprozesses (Analyse, Design und Implementierung) werden in den folgenden drei Kapiteln ausführlich behandelt.

2 Analyse

Dieses Kapitel beinhaltet die Ergebnisse der *Analysephase* nach FUSION. Aufgabe dieser Phase war eine präzise Beschreibung dessen, *was* das System SPIRIT leisten soll. Daher wird in diesem Kapitel weniger auf die internen Strukturen, sondern hauptsächlich auf das nach *außen* sichtbare Verhalten des Systems eingegangen. In der anschließenden Entwurfsphase wird dann in Kapitel 3 präzisiert, *wie* das System funktioniert.

Zunächst wird in Abschnitt 2.1 das *Domain Object Model* vorgestellt, das die grundlegenden Konzepte (Klassen) des Systems und die Beziehungen zwischen ihnen darstellt. Daraufhin trennt Abschnitt 2.2 mit dem *System Object Model* die Systemumgebung vom eigentlichen Systemkern. Mit dem *Operation Model* aus Abschnitt 2.3 werden dann die vom System bereitzustellenden Operationen eingeführt. Abschließend definiert Abschnitt 2.4 die zulässigen *Life-Cycles* einer Wissensbasis und führt so einige Verwendungsmöglichkeiten der Systemoperationen vor.

2.1 Domain Object Model

Dieser Abschnitt beschreibt die in der *Analysephase* ermittelten Klassen und Beziehungen des Systems. Im Mittelpunkt der Überlegungen zur Erstellung der Objektklassen stand der Grundsatz, das System im wesentlichen als *Black-Box* zu betrachten. Der Anwender sollte lediglich die *Ergebnisse* der System-Operationen zu sehen bekommen und möglichst nicht mit den internen Abläufen konfrontiert werden. Insbesondere ist im Projekt SPIRIT davon auszugehen, daß in Zukunft neue Algorithmen und Datenstrukturen gefunden werden, die teilweise wesentliche interne Änderungen nach sich ziehen könnten. Im Falle des Iterationsvorganges zum Lernen von Regeln gibt es z.B. verschiedene theoretische Ansätze, deren Vor- und Nachteile ohne eine Implementierung nur schwer ermittelbar sind. Das bedeutet, daß relativ unklar ist, ob das letztlich realisierte Iterationsverfahren überhaupt optimal ist.

Somit stellen die in diesem Abschnitt vorgestellten Klassen nur eine nach außen sichtbare Schale dar, hinter der sich wesentlich komplexere Strukturen verbergen, die jedoch für den Benutzer nicht relevant sind. Dementsprechend ist das im folgenden diskutierte Modell gemessen am Gesamtumfang des Systems recht klein.

Abbildung 2 stellt die beiden wichtigsten Objektklassen eines probabilistischen Expertensystems vor. Ein *System* besteht aus einer Shell, die als Schnittstelle zum Endanwender fungiert, und einer beliebigen Menge von Wissensbasen (*Knowledge-Base* oder einfach KBase), die durch die Shell verwaltet werden können.

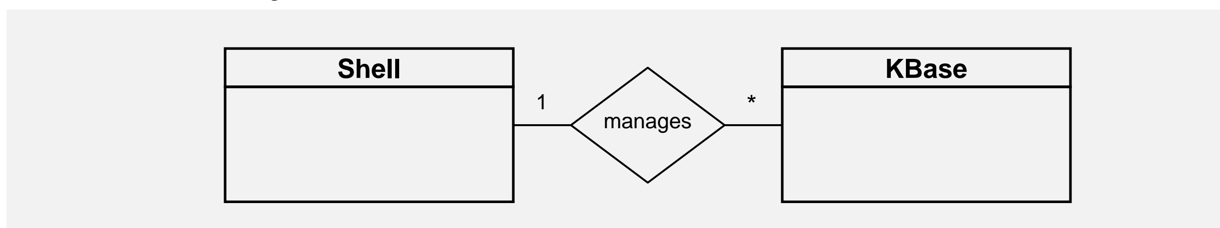


Abbildung 2: *Domain Object Model* mit Shell und Wissensbasis

Bei der Gestaltung einer Shell sind dem Programmierer viele Freiheiten gegeben. Beispielsweise benötigt eine medizinische Spezialanwendung, die Anfragen auf einer statischen Regelmenge ermöglicht, nur einen Bruchteil der zur Verfügung stehenden Funktionen der Wissensbasis. Demgegenüber muß ein allgemeiner *Knowledge-Editor* auf wesentlich mehr Operationen zurückgreifen. Die Klasse KBase muß dementsprechend ein umfangreiches Repertoire an Schnittstellen-Funktionen bereitstellen, um den unterschiedlichsten Anforderungen zu genügen. Welche der Funktionen in welchem Zusammenhang später genutzt werden, bleibt der Shell überlassen.

Abbildung 3 stellt den groben Aufbau einer Wissensbasis der Klasse KBase dar. Diese besteht in erster Linie aus Variablen (Klasse Variable), Regeln (Rule) und LEGs. Hinzu kommt maximal ein IterationManager, der einen Iterationsprozeß zum Lernen gemäß Algorithmus RuleLearnIteration steuert.

Jede Variable verfügt über einen eindeutigen Namen und einen Typ. An diesem Punkt des Entwicklungsprozesses ist es noch nicht sinnvoll, detaillierter auf die Unterschiede zwischen den vier möglichen Variablentypen einzugehen: die Zusammenhänge werden zu Beginn der Entwurfsphase deutlicher, wenn die interne Darstellung von Regeln vorgestellt wird. Bis hierhin können wir abstrahieren, daß jede Variable mindestens zwei diskrete Werte (Values) besitzen muß. Je nach Variablentyp sind dies entweder

Wahrheitswerte, Bezeichner, Zahlen oder Intervalle (dies wird hier nicht weiter präzisiert, ebenso wie die *Utilities* der Werte (vgl. Abschnitt 1.2.1 unten)). Bei *Number*- und *Interval*-Variablen müssen die Werte aufsteigend sortiert gespeichert werden. Schließlich gibt das Attribut enum den Index einer Variable in der Eliminationsreihenfolge (sh. Algorithmus LEGForestBuild) an.

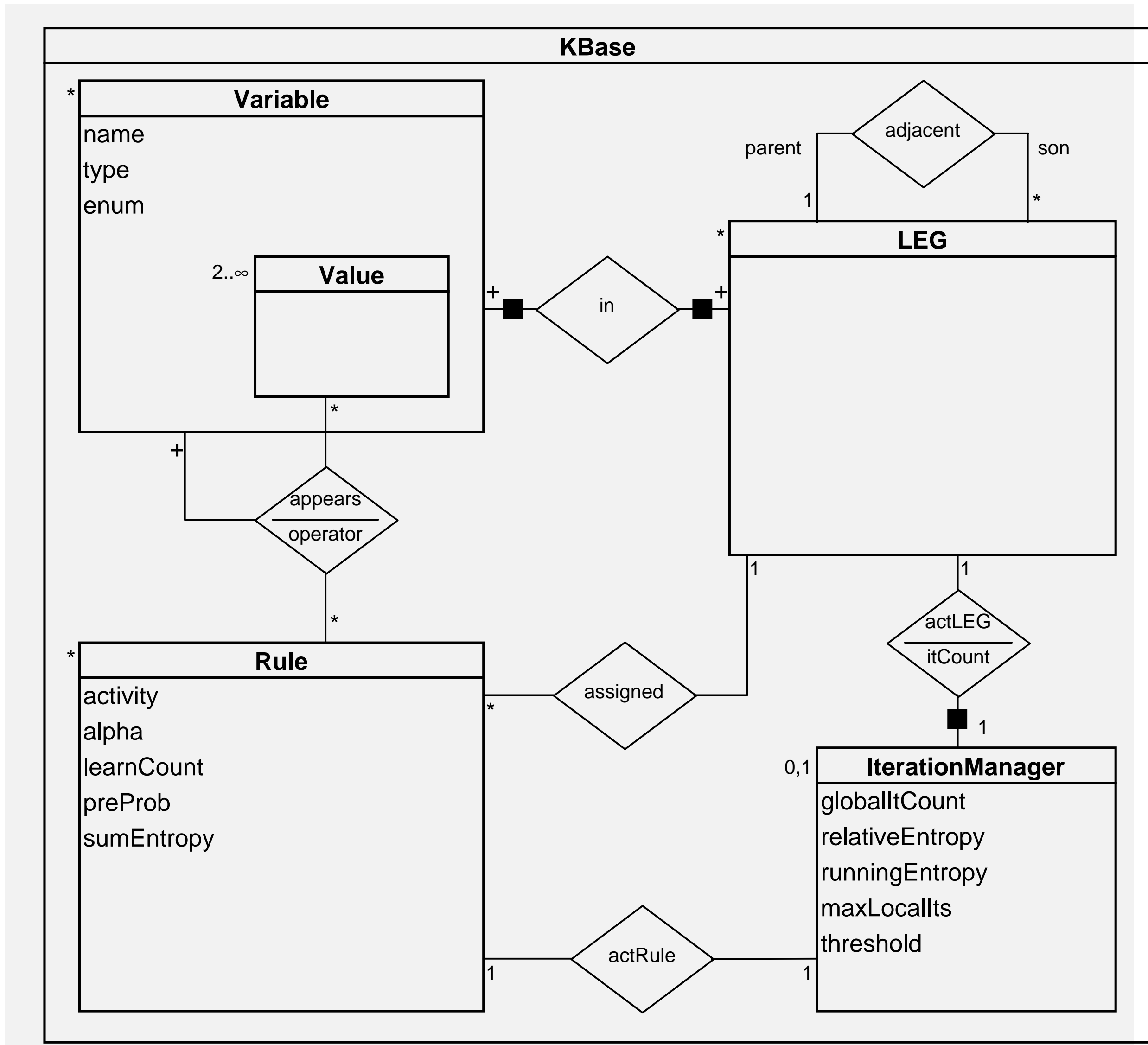


Abbildung 3: *Object Model* der Klasse KBase

Jede Regel hält Informationen über ihren Aktivierungszustand (activity), den aktuellen Lagrange-Parameter (alpha), die Anzahl der bisherigen Verwendungen zum Lernen (learnCount), die Summe der relativen Entropien der Lernschritte (sumEntropy) und die vorgegebene bedingte Wahrscheinlichkeit, die hier mit preProb (*prescribed probability*) bezeichnet wird. Diese kann entweder eine einzelne Wahrscheinlichkeit oder ein Intervall zwischen 0 und 1 sein, wird aber zur Vereinfachung in der *Analysephase* zunächst als einzelnes, atomares Attribut behandelt.

Der Iterationsprozeß ist ein wesentlicher Bestandteil bei der Erzeugung einer widerspruchsfreien Wissensbasis. Da diese Iteration sehr lange dauern kann und der Benutzer über deren Fortschritt informiert werden, bzw. sogar gezielt in deren Ablauf eingreifen möchte, muß ein schrittweiser Ablauf ermöglicht werden. Das hierzu verwendete Modul ist der IterationManager. Er speichert die aktuellen Entropien und die Anzahl der globalen Iterationen. Zudem verfügt er über Optionen (threshold und maxLocalIts), die festlegen, wann eine lokale Iteration beendet werden kann.

Jede Variable der Wissensbasis muß in mindestens einer LEG dargestellt sein (Relationship in), wobei jede LEG mindestens eine Variable enthält. Regeln können Variablen mit ihren Werten in Beziehung setzen, indem sie z.B. Aussagen wie $A>a_2$ machen. Dies wird über die Relation appears ausgedrückt. In jeder Regel muß mindestens eine Variable auftauchen. In dieser Relation gibt operator den Verknüpfungsoperator an ($=$, $\neq$, $<$, $>$, $\in$ oder $\notin$).

Dementsprechend dürfen an einer appears-Relation nur Values der verknüpften Variable teilnehmen (genauer: sei (r, var, val_{var}) eine appears-Relation, dann muß gelten: val_{var} ist aggregiert in var). Auch diese Zusammenhänge werden später präzisiert, nachdem festgelegt wurde, wie Regeln intern verwaltet werden.

Einer Regel kann mit assigned eine LEG zugeordnet werden, in die sie paßt. Eine solche LEG muß eine Obermenge der in der Regel vorkommenden Variablen enthalten. Der Aktivierungszustand (activity) gibt dann an, ob die Regel active oder activable ist. Es kann allerdings sein, daß eine Regel in keine LEG der aktuellen Struktur paßt. In diesem Fall ist die Regel an keiner assigned-Relation beteiligt, hat den Aktivierungszustand passive und steht nicht für Lernschritte zur Verfügung.

Die LEGs sind in einem Wald angeordnet, dessen Struktur i.d.R. durch die Variablen- und Regelmengen bestimmt wird. Die Nachbarschaftsverhältnisse der LEG-Objekte werden über die Beziehung adjacent verwaltet. Jede (parent) LEG kann beliebig viele (son) LEGs haben. Isolierte LEGs sind an keiner adjacent-Beziehung beteiligt.

Der IterationManager betrachtet zu jedem Zeitpunkt genau eine LEG (actLEG). Ist dieser LEG eine nichtleere Menge von aktiven Regeln zugeordnet, so gibt actRule die nächste Regel an, die in die LEG gelernt werden wird. Zur aktuellen LEG wird die Anzahl der lokalen Iterationen gespeichert. Überschreitet dieser Wert die Option maxLocalIts, oder ändert sich die relative Entropie während einer lokalen Iteration um weniger als threshold, so kann die lokale Iteration abgebrochen und mit der nächsten LEG fortgefahren werden.

Bei der Ermittlung des *Domain Object Model* war es für mich hilfreich, noch einmal die Klassenhierarchie der Vorgängerversionen zu untersuchen. Dabei stellte ich fest, daß mein damaliger Ansatz zahlreiche Mängel enthielt. Mir wurden einige Zusammenhänge zwischen den Klassen deutlich, die ich damals völlig anders umgesetzt hatte. Beispielsweise bestanden im alten Modell keine direkten Verknüpfungen zwischen Regeln und LEGs (hier: assigned) und zwischen Regeln und Variablen (hier: appears). Diese Relationen wurden statt dessen bei Bedarf jeweils neu berechnet. Auch wurden die Nachbarschaftsverhältnisse zwischen den LEGs nicht über eine Relation (adjacent) verknüpft, sondern mit einem separaten LEGGraph-Objekt. Letztlich konnte aus dem alten Modell (außer Erfahrungen) nicht viel übernommen werden.

2.2 System Object Model

Mit Einführung des *System Object Model* wird der eigentliche Systemkern von seiner Umgebung getrennt. Im Projekt SPIRIT war es naheliegend, diese Trennungslinie zwischen Shell und Wissensbasis zu ziehen. Eine Shell kann eine oder mehrere Wissensbasen verwalten und interagiert dazu mit Instanzen der Klasse KBase. Sie stößt über den Aufruf von System-Operationen Änderungen in der Wissensbasis an und ist in der Lage, Anfragen an diese zu stellen. Die KBase liefert die entsprechenden Ergebnisse der Anfragen zurück an die Shell, die wiederum für deren Darstellung und Verwertung verantwortlich ist.

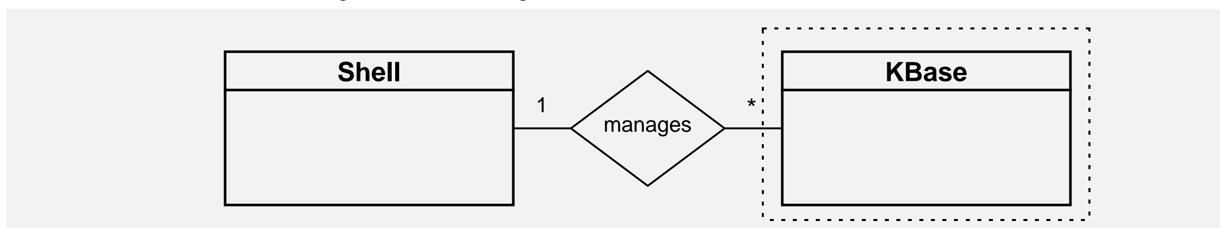


Abbildung 4: Das *System Object Model*

Dieses Modell entspricht übrigens genau dem Ansatz, der in den Vorgängerversionen mit der Aufspaltung des Systems in Oberfläche und Systemkern eingeführt wurde.

2.3 Operation Model

Dieser Abschnitt enthält eine Auflistung der in der *Analysephase* ermittelten System-Operationen von SPIRIT. Dies sind genau die Funktionen, über die eine Shell später mit einer Wissensbasis kommuniziert. Entsprechend war es hier besonders wichtig, keine Operation zu vergessen. Ich ging daher sowohl die *Requirements* als auch das *Interface* der letzten Vorgängerversion noch einmal sorgfältig durch. Dabei stellten sich einige Unklarheiten heraus, die in den Anforderungen nachgebessert werden mußten. Auch in der zuvor definierten Objektstruktur wurden noch kleinere Anpassungen vorgenommen, die sich aus Überlegungen im Zusammenhang mit dem *Operation Model* ergaben.

In diesem Abschnitt wurde die von FUSION vorgeschlagene Form der *Operation-Schemata* zur Definition der System-Operationen etwas aufgelockert. Neben anderen Vereinfachungen gelten folgende Konventionen:

- Leere Felder werden (zur Platzersparnis) weggelassen.
- Die Instanz der Klasse KBase, auf die sich die Operationen beziehen, wird mit kBase bezeichnet.
- Auf das **Result**-Feld wird häufig verzichtet. Die Wirkung der entsprechenden Funktionen kann mit den restlichen Feldern der Schemata (z.B. **Sends** und **Changes**) ausreichend nachvollzogen werden.
- Das Ziel der im **Sends**-Feld angegebenen Nachrichten ist immer eine Instanz der Klasse Shell.
- Alle vorkommenden Datentypen sind im *Data-Dictionary* im Anhang F beschrieben.
- Bei vielen Items wird in Klammern seine Wirkung als Parameter, etc. beschrieben. Bei Items vom Typ boolean wird angegeben, was im Fall *true* passiert.
- Das Schlüsselwort **delete** vor einem Item gibt an, daß das Item durch die Operation gelöscht wird.
- Aussagen können mit den Schlüsselwörtern **if** <Condition> **then** an Bedingungen geknüpft werden.
- Aggregierte Objekte können mit @ auf ihr umgebendes (Parent-) Objekt zugreifen (z.B. Value@Variable)
- Einige Operationen geben im Feld **Sends** ein Item vom Typ Exception an. Dieses wird nur in Ausnahmefällen geliefert. In solchen Fällen wird die Funktion frühzeitig abgebrochen und das **Result** ist teilweise ungültig.
- Im Feld **Assumes** werden folgende Kürzel verwendet, die mit dem Schlüsselwort **not** negiert werden können:
  - **iterating**: kBase befindet sich gerade in einer Iteration, d.h. es existiert ein IterationManager
  - **evidence**: Es ist mindestens einer Variable Evidenz zugewiesen (mit assignEvidence)

Bei der Ermittlung der erforderlichen System-Operationen war es sehr hilfreich, die *Interface*-Funktionen der Vorgängerversionen als Ausgangspunkt zu nehmen. Ich überprüfte, welche dieser (vielen!) Funktionen damals überhaupt verwendet wurden, und welche sich aus "historischen" Gründen als überflüssig und umständlich erwiesen. Die im folgenden vorgestellten Operationen bilden eine kleine Untermenge der alten Funktionen.

Die folgenden Unterabschnitte nennen *alle* System-Operationen. Die Funktionen sind gemäß ihrer Aufgabe in fünf Gruppen unterteilt.

2.3.1 Verwaltung von Wissensbasen und LEG-Wäldern

Operation:	**absEntropy**
Description:	Ermittelt die absolute Entropie der aktuellen Verteilung.
Reads:	LEG (Zur Errechnung der Entropie gemäß Algorithmus LEGForestAbsEntropy)
Sends:	entropy: double
Assumes:	**not iterating**. **not evidence**.

Operation:	**assignEvidence**
Description:	Weist einer gegebenen Menge von Variablen Evidenz gemäß Algorithmus AssignEvidence zu.
Reads:	**supplied** valueList: Vector of Value (zuzuweisende Werte); in, LEG, adjacent
Changes:	LEG (Die angegebenen Werte haben die Wahrscheinlichkeit eins)
Sends:	failure: Exception (eine LEG wurde vollständig null, z.B. wenn einer Variable doppelt Evidenz zugewiesen wird)
Assumes:	**not iterating**. Keiner Variable kann mehr als ein Wert zugewiesen werden.
Result:	**evidence:** Den Variablen der in valueList angegebenen Werte wurden (zusätzlich zu evtl. schon zuvor gültigen) Evidenzen zugewiesen.

Operation:	**assignEvidenceReset**
Description:	Nimmt alle Evidenzzuweisungen (über assignEvidence) zurück, es gilt also **not evidence**.
Reads:	in, LEG, adjacent
Changes:	LEG (Die Wahrscheinlichkeiten ohne Evidenzzuweisungen wurden wiederhergestellt)
Assumes:	**not iterating**. **evidence**.

Operation:	**clone**
Description:	Erzeugt eine Kopie von kBase.
Reads:	**supplied** withRules: boolean (die aktuellen Regeln werden mitkopiert); KBase
Changes:	**new** copy: KBase (Variablen, LEG-Struktur und ggfs. Regeln wurden aus kBase übernommen)
Sends:	copy
Assumes:	**not iterating**. **not evidence**.
Result:	**if** ¬withRules **then** copy enthält keine Regeln.

Operation:	**enumerateVars**
Description:	Erzeugt eine neue Variablennumerierung für den Neuaufbau der LEG-Struktur.
Reads:	**supplied** method: EnumMethod (zu verwendende Heuristik) Rule, Variable (zur Ermittlung der neuen Eliminationsfolge)
Changes:	Variable.enum
Result:	Die Variablen wurden mit der gewählten Heuristik neu numeriert.

Operation:	**exportFile**
Description:	Schreibt die Wissensbasis in eine SPIRIT-Datei gemäß Anhang B.
Reads:	**supplied** file: File (zu erzeugende SPIRIT-Datei); Variable, Rule
Sends:	error: Exception (Schreibfehler)
Assumes:	**not iterating**. **not evidence**.
Result:	Die aktuelle Wissensbasis wurde gemäß SPIRIT-Format in die angeg. Datei geschrieben.

Operation:	**importFile**
Description:	Lädt eine SPIRIT-Datei zu kBase hinzu.
Reads:	**supplied** file: File (existierende SPIRIT-Datei)
Changes:	kBase
Sends:	error: Exception (Lesefehler, bzw. ungültige Deklaration)
Assumes:	**not iterating**. **not evidence**.
Result:	Die in der Datei befindlichen Variablen und Regeln wurden zu kBase hinzugefügt. Beim ersten Auftreten eines Fehlers wird die Exception gemeldet und abgebrochen.

Operation:	**rebuild**
Description:	Versucht den LEG-Wald neu aufzubauen.
Reads:	Rule, Variable (zur Ermittlung der neuen LEG-Struktur gemäß Algorithmus LEGForestBuild)
Changes:	LEG
Sends:	failure: Exception (LEG zu groß)
Assumes:	**not iterating**. **not evidence**.
Result:	Der LEG-Wald wurde mit der aktuellen Variablennumerierung neu aufgebaut. kBase ist gleichverteilt (alte Wahrscheinlichkeiten mittels kBaseReinit wiederherstellbar). **if** Neuaufbau schlug fehl (LEG zu groß) **then** alte Struktur behalten, Exception gemeldet

Operation:	**reinit**
Description:	Führt ein Re-Init gemäß Algorithmus LEGForestReinit aus.
Reads:	**supplied** withReset: boolean (zuvor gleichverteilen); Rule.alpha, adjacent, assigned
Changes:	LEG
Assumes:	**not iterating**. **not evidence**.

Operation:	**reset**
Description:	Setzt alle LEGs auf Gleichverteilung zurück.
Reads:	Variable (für die Dichtefunktion in Algorithmus LEGReset)
Changes:	LEG; Rule.alpha, Rule.learnCount und Rule.sumEntr (werden zurückgesetzt)
Assumes:	**not iterating**. **not evidence**.

2.3.2 Verwaltung von Variablen

Operation:	**addVar**
Description:	Fügt eine neue Variable zur Wissensbasis hinzu.
Reads:	**supplied** name: String (Name der neuen Variable) **supplied** type: VarType (Typ der neuen Variable) **if** type ∈ { nominal, number, interval } **then supplied** initValues: Vector of Value **with** |initValues| > 1 (zu jeder Variable müssen mindestens zwei Werte angegeben werden)
Changes:	**new** var: Variable **with** var.type = type und var.name = name **if** type = boolean **then** für var werden die zwei Werte 0 und 1 angelegt. **else** für var werden die in initValues angegebenen Werte oder Intervallgrenzen angelegt. **new** LEG **with** var einzige enthaltene Variable (Relationship in)
Assumes:	**not iterating**. **not evidence**. { var': Variable ∈ kBase | var'.name = name } = ∅

Operation:	**deleteVar**
Description:	Löscht eine angegebene Variable.
Reads:	**supplied** var: Variable (zu löschende Variable); appears (zum Finden der Regeln)
Changes:	**delete** var; **delete** rule: Rule **with** var.appears.rule; LEG
Assumes:	**not iterating**. **not evidence**.
Result:	var wurde aus allen LEGs gestrichen und dadurch leer gewordene LEGs gelöscht.

Operation:	**addValue**
Description:	Erzeugt einen neuen Wert für eine *Nominal*- oder *Number*-Variable oder eine neue Intervall-Grenze für eine *Interval*-Variable.
Reads:	**supplied** var: Variable (Variable zu der ein Wert hinzugefügt werden soll) **supplied** value: Value (Bezeichnung des neuen Variablenwertes, bzw. double-Darstellung der neuen Intervall-Grenze) var.in.LEG (zur Änderung der LEGs, in denen var vorkommt).
Changes:	var.in.LEG (die Wertetabellen, in denen var vorkommt, werden erweitert) **new** var.value: Value (neuer Wert wurde wie in value angegeben erzeugt)
Sends:	failure: Exception (eine LEG wurde durch den neuen Wert zu groß)
Assumes:	**not iterating**. **not evidence**. var.type ∈ { nominal, number, interval }

Operation:	**deleteValue**
Description:	Löscht einen Variablenwert einer *Nominal*- oder *Number*-Variable oder eine Intervall-Grenze für eine *Interval*-Variable.
Reads:	**supplied** value: Value (zu löschender Wert); Variable
Changes:	**delete** value; **delete** value.appears.Rule; LEG
Assumes:	**not iterating**. **not evidence**. value.Variable.type ∈ { nominal, number, interval }, var hat mindestens 3 Werte
Result:	Alle Regeln, die auf value zugreifen mußten, wurden gelöscht.

Operation:	**valueProb**
Description:	Ermittelt die aktuelle bedingte Wahrscheinlichkeit eines Variablenwertes gemäß Algorithmus VarValueActProb.
Reads:	**supplied** value: Value (Wert dessen bed. Wahrscheinlichkeit ermittelt werden soll) value@Variable.in.LEG (zur Errechnung der bedingten Wahrscheinlichkeit).
Sends:	p: double (ermittelte bedingte Wahrscheinlichkeit)
Assumes:	**not iterating**.

2.3.3 Verwaltung von Regeln

Operation:	**addRule**
Description:	Fügt eine neue Regel in kBase ein.
Reads:	**supplied** ruleString: String (textuelle Darstellung der neuen Regel) **supplied** preProb: PreProb (vorgegebene bed. Wahrscheinlichkeit der neuen Regel) LEG (zur Suche nach einem passenden LEG für die neue Regel), Variable (Regel-Parser)
Changes:	**new** rule: Rule **with** rule.preProb = preProb, rule.alpha = 1, rule._ = 0
Sends:	failure: Exception (der angegebene ruleString war fehlerhaft (sh. auch ruleStringCheck))
Assumes:	**not iterating**. **not evidence**.
Result:	**if** ∃ passende LEG für rule **then** rule.activity = active **else** passive.

Operation:	**deleteRule**
Description:	Löscht eine angegebene Regel.
Reads:	**supplied** rule: Rule (zu löschende Regel)
Changes:	**delete** rule
Assumes:	**not iterating**. **not evidence**.

Operation:	**ruleProb / rulePremProb**
Description:	Ermittelt die aktuelle bedingte Wahrscheinlichkeit einer Regel / Prämisse einer Regel.
Reads:	**supplied** rule: Rule (Regel, deren Wahrscheinlichkeit berechnet werden soll) rule.assigned.LEG (zur Errechnung der Wahrscheinlichkeit)
Sends:	p: double (bedingte Wahrsch'keit gemäß Algorithmus RuleActProb / RuleActPremProb)
Assumes:	**not iterating**.

Operation:	**ruleStringCheck**
Description:	Prüft, ob ein gegebener String eine gemäß Anhang A zulässige Regel darstellt.
Reads:	**supplied** string: String (zu prüfender String) Variable (für die Syntaxprüfung)
Sends:	error: Exception mit Fehlerursache und -Position

2.3.4 Darstellung der Variablengraphen

Operation:	**graphAdjacentVars**
Description:	Ermittelt die benachbarten Knoten einer Variable in einem Variablengraphen.
Reads:	**supplied** graphType: GraphType, var: Variable; Rule (zur Ermittlung der Abhängigkeiten)
Sends:	vars: Variable[] (Die Liste der Nachbarknoten von var)

Operation:	**graphEdgeType**
Description:	Ermittelt die Art der Kante zwischen zwei Variablen in einem Variablengraphen.
Reads:	**supplied** graphType: GraphType, var0, var1: Variable (die beiden Ecken der Kante) Rule (zur Ermittlung der Abhängigkeiten)
Sends:	edgeType: EdgeType

Operation:	**graphEdgeThickness**
Description:	Ermittelt die Stärke der Kante zwischen zwei Variablen in einem Variablengraphen.
Reads:	**supplied** var0, var1: Variable (die beiden Ecken der zu messenden Kante) LEG (zur Ermittlung der Kantendicke gemäß Algorithmus VarGraphEdgeThickness)
Sends:	thickness: double (ermittelte Kantendicke)

2.3.5 Steuerung und Überwachung der Iteration

Operation:	**beginIteration**
Description:	Bereitet eine neue Iteration vor.
Reads:	**supplied** threshold: double (ε-Schwelle für die Terminierung der Iterationen) **supplied** maxLocalIts: int (maximale Anzahl lokaler Iterationen) LEG, Rule (für die Startposition der Iteration)
Changes:	**new** im: IterationManager mit im.threshold = threshold, im.maxLocalIts = maxLocalIts, im.runningEntropy = absEntropy (), im._ = 0
Assumes:	**not iterating**. **not evidence**.
Result:	**iterating:** kBase besitzt den neuen IterationManager im.

Operation:	**endIteration**
Description:	Beendet eine laufende Iteration.
Changes:	**delete** kBase.IterationManager; LEG
Assumes:	**iterating**.
Result:	kBase wurde in einen konsistenten Zustand überführt.

Operation:	**stepIteration**
Description:	Führt den im IterationManager folgenden Schritt aus (Algorithmus RuleLearnIteration)
Reads:	IterationManager, LEG, Rule, Variable
Changes:	LEG, IterationManager, Rule
Sends:	ready: boolean (wenn Iteration terminiert ist, also das Abbruchkriterium erfüllt ist)
Assumes:	**iterating**.
Result:	Die Wirkung hängt davon ab, in welchem Zustand der IterationManager war (Details später).

Zusätzlich zu den hier aufgeführten muß SPIRIT noch eine Reihe trivialer, (fast) unbeschränkt aufrufbarer Funktionen unterstützen, die keiner genaueren Untersuchung bedürfen, da sie nur sehr lokale Wirkung haben:

Name	Beschreibung
(get|set)RulePreProb	Liefert/Ändert die vorgegebene bed. Wahrscheinlichkeit einer Regel (Rule.preProb)
(get|set)VarEnum	Liefert/Ändert den Index einer Variable in der Eliminationsfolge (Variable.enum)
(get|set)Utility	Liefert/Ändert den Nutzen eines Variablenwertes
expectedValue	Liefert den Erwartungswert einer Variable (ähnlich wie valueProb)
get[Rule|Var]UserInfo	Liefert freie Informationen, die der Wissensbasis/Regel/Variable zugeordnet sind
getIterationStatus	Liefert den Status einer laufenden Iteration (die Attribute und Relationen)
getLEGInfo	Liefert Informationen über die LEG-Struktur (Variablen pro LEG und Adjazenz)
getRuleActivity	Liefert den Aktivierungszustand einer Regel (Rule.activity)
getRuleAlpha	Liefert den Lagrange-Parameter Alpha einer Regel (Rule.alpha)
getRuleCount	Liefert die Anzahl der Regeln in der Wissensbasis
getRuleLearnCount	Liefert die Zahl der Verwendungen einer Regel in Lernschritten (Rule.learnCount)
getRuleString(DNF)	Liefert eine Regel in String-Darstellung (auch in DNF)
getRuleSumEntropy	Liefert die von einer Regel bewirkten Entropieänderungen (Rule.sumEntropy)
getValueName	Liefert einen Variablenwert in String-Darstellung
getVarArity	Liefert die Anzahl der Werte einer Variable (= Anzahl der aggregierten Values)
getVarCount	Liefert die Anzahl der Variablen in der Wissensbasis
getVarName	Liefert den Namen einer gegebenen Variable (Variable.name)
getVarRules	Liefert eine Liste der Regeln, die eine gegebene Variable verwenden (appears)
getVarType	Liefert den Typ einer gegebenen Variable (Variable.type)
renameVar	Nennt eine Variable um
ruleIsFact	Prüft, ob es sich bei einer angegebenen Regel um ein Fakt handelt
set[Var|Rule]UserInfo	Ändert die freien Informationen, die der Wissensbasis/Regel/Variable zugeordnet sind
toggleRuleActivity	Wechselt den Aktivierungszustand einer Regel zwischen activable und active

2.4 Life-Cycle Model

In diesem Abschnitt werden zulässige Lebenszyklen von Instanzen der Klasse KBase definiert. Die Entwicklung dieser *Life-Cycles* diente dazu, spätere Verwendungsmöglichkeiten der System-Operationen herauszuarbeiten. Im folgenden werden zwei beispielhafte Szenarien vorgestellt, die den Einsatz von Wissensbasen in Shells beschreiben. Dabei entspricht das Szenario aus Abschnitt 2.4.2 zugleich dem allgemeinsten Fall.

2.4.1 Einsatz in einem einfachen medizinischen Anfragesystem

Gegeben sei eine Shell zur Diagnose von Krankheiten. Darin gibt ein Arzt auf einer Bildschirmmaske an, welche Krankheitssymptome bei einem Patienten beobachtet wurden. Die Wissensbasis wird anhand dieser Vorgaben nach möglichen Ursachen für die Symptome befragt. Dabei wird eine konsistente Regelmenge, die eine probabilistische Beziehung zwischen Symptomen und Krankheiten herstellt, als gegeben vorausgesetzt.

Die Wissensbasis wird nach dem Systemstart mit Variablen (Symptome und mögliche Krankheiten) und Regeln "gefüttert". Anschließend wird eine geeignete LEG-Struktur aufgebaut, um mit einer Iteration aus den Regeln eine gemeinsame Verteilung zu ermitteln. Dieser Verteilung können dann Evidenzen zugewiesen werden. Dazu gibt der Arzt diejenigen Variablenwerte vor, die als sicher beobachtet gelten. Um zu einer Diagnose zu gelangen, können daraufhin die neuen bedingten Wahrscheinlichkeiten der (Krankheits-) Variablen von der Shell ausgewertet und angezeigt werden.

lifecycle *KBase*$_{med}$:	*Initialisierung . Struktur-Aufbau . Iteration . Evidenzen&Anfragen**
Initialisierung =	addVar$^+$. addRule$^+$
Struktur-Aufbau =	enumerateVars . rebuild
Iteration =	beginIteration . stepIteration$^+$. endIteration
Evidenzen&Anfragen =	assignEvidence . (valueProb | expectedValue)* . assignEvidenceReset

2.4.2 Einsatz in einer allgemeinen Knowledge-Engineering-Shell

Gegeben sei ein komplexer Regel-Editor, der einen *Knowledge-Engineer* bei der Erstellung einer konsistenten Regelmenge unterstützt und gleichzeitig alle Formen von Anfragen zuläßt. Wie sich zeigt, müssen hierbei die weitaus meisten Funktionen in beliebiger Reihenfolge aufgerufen werden können, da der Anwender der Shell häufig zwischen Bearbeitung, Iteration und Anfragen wechseln muß und gleichzeitig einen Überblick über den aktuellen Stand der Wissensbearbeitung benötigt. Während der Iteration befindet sich die Wissensbasis in einem Zustand, der Änderungen und komplexe Anfragen verbietet. Daher sind in diesem Stadium nicht alle Operationen zulässig. Ähnliches gilt, nachdem Evidenzzuweisungen gemacht wurden. Die in den folgenden Diagrammen nicht aufgeführten Funktionen sind zu jedem Zeitpunkt aufrufbar.

lifecycle *KBase* :	(*Bearbeitung* | *Lernen* | *Anfrage*)*

Eine Wissensbasis sollte zunächst von einem Experten mit Wissen ausgestattet werden. Hierzu muß SPIRIT folgende Operationen ermöglichen:

- Import und Export von Wissensbasen, Duplizierung von Wissensbasen zum Weiterarbeiten mit der Kopie,
- Erzeugen und Löschen von Variablen und Variablenwerten,
- Erzeugen, Bearbeiten und Löschen von Regeln, und
- Neuaufbau der LEG-Struktur, und Ermittlung der absoluten Entropie.

All diese Operationen können in beliebiger Reihenfolge aufgerufen werden. Allerdings kann nicht verhindert werden, daß eine Shell eine ungültige Folge von Operationen startet. Beispielsweise können nur Variablen gelöscht werden, die zuvor angelegt wurden. Auch können einige Operationen mit unzulässigen Argumenten aufgerufen werden. Diese Fälle sind mit der hier verwendeten Syntax nicht beschreibbar und müssen in den späteren Funktionen mithilfe einer entsprechenden Ausnahmebehandlung abgefangen werden.

Bearbeitung =	*SystemBefehle* | *VariablenBearbeitung* | *RegelBearbeitung* | *LEGBearbeitung*
SystemBefehle =	importFile | exportFile | clone
VariablenBearbeitung =	addVar | deleteVar | addValue | deleteValue
RegelBearbeitung =	addRule | deleteRule | toggleRuleActivity
LEGBearbeitung =	enumerateVars | rebuild | absEntropy

Der zweite wichtige Schritt bei der Erstellung einer Wissensbasis ist das Lernen von probabilistischem Wissen in die interne LEG-Struktur. Hierbei sind folgende Methoden zu unterscheiden:

- Iteration der Regeln. Durch die Iteration werden oft Inkonsistenzen in der Regelmenge aufgedeckt, die durch weitere Bearbeitungsschritte behoben werden können. Regeldefinition und Iteration wechseln sich i.a. so lange ab, bis der Knowledge-Engineer zu einer zufriedenstellenden Wissensbasis gekommen ist.
- Zuweisung von Evidenzen an Variablen und Abfrage der bewirkten Änderungen
- Zurücksetzen und Re-Init der gesamten LEG-Struktur

Lernen =	*Iteration* | *Evidenzen&Anfragen* | reset | reinit
Iteration =	beginIteration . (stepIteration | getIterationStatus)* . endIteration
Evidenzen&Anfragen =	assignEvidence . (valueProb | expectedValue)* . assignEvidenceReset

Nachdem die Regeln der Wissensbasis durch die Iteration in den LEG-Wald gelernt wurden, können Anfragen gestellt werden. Es gibt zwei Arten von Anfragen:

- Anfragen an die aktuellen bedingten Wahrscheinlichkeiten der Variablenwerte, Berechnung des Erwartungswertes von Variablen und die Ermittlung der Kantenstärken der beiden Variablengraphen
- Anfragen an die aktuellen bedingten Wahrscheinlichkeiten der Regeln und deren Prämissen

Anfrage =	*VariablenAnfrage* | *RegelAnfrage*
VariablenAnfrage =	valueProb | expectedValue | graphEdgeThickness
RegelAnfrage =	ruleProb | rulePremProb

Mit der Fertigstellung der *Life-Cycles* wurde auch die *Analysephase* abgeschlossen. Die Analyse hatte den Zweck festzulegen, *was* das zu entwickelnde System leisten soll. Als primäre Ergebnisse entstanden dabei das *System Object Model*, das eine Grobstruktur der späteren Klassenhierarchie bildet, und das *Operation Model*, das die von außen aufrufbaren Funktionen spezifiziert. Diese beiden Modelle können als Eingabe für den nachfolgenden Design-Prozeß verstanden werden. Neben diesen "handfesten" Ergebnissen trug die *Analysephase* als sekundäres Ergebnis auch zum tieferen Verständnis der Aufgabenstellung bei und half bei der Aufdeckung von Inkonsistenzen und der Lösung von Unklarheiten in den *Requirements*.

3 Design

In diesem Kapitel werden die Ergebnisse der Entwurfsphase gemäß FUSION vorgestellt. Aufgabe dieser Phase war die Erstellung von Software-Strukturen zur Umsetzung der abstrakten Definitionen des Analyse-Modells. Dazu wird zunächst das Analyse-Modell um solche Konzepte erweitert, die eine Implementierung mit geeigneten Klassen ermöglichen (Abschnitt 3.1). Anschließend werden die vier Modelle des Design-Prozesses in der durch FUSION vorgeschlagenen Reihenfolge aufgebaut. Die *Object Interaction Graphs* visualisieren den Informationsfluß zwischen den an einer System-Operation beteiligten Objekten (Abschnitt 3.2). Die hierzu erforderlichen Zugriffspfade zwischen den Objektklassen werden in den *Visibility Graphs* dargestellt (Abschnitt 3.3). Anschließend werden die Vererbungsstrukturen durch *Inheritance Graphs* abgebildet (Abschnitt 3.4). Als wichtigstes Ergebnis der Entwurfsphase definieren die *Class Descriptions* die zu realisierenden Klassen mit ihren Feldern und Attributen (Abschnitt 3.5). Die dort dargestellten Klassen bilden den Ausgangspunkt für die im anschließenden Kapitel 4 beschriebene Implementierungsphase.

3.1 Erweiterung des Analyse-Modells um Implementierungskonzepte

Mit dem Beginn der Entwurfsphase gemäß FUSION werden üblicherweise neue Objektklassen eingeführt, die das in der *Analysephase* entworfene *System Object Model* erweitern. Diese neuen Klassen ermöglichen die softwaretechnische Umsetzung der abstrakten Anforderungen. Gerade im vorliegenden Projekt SPIRIT waren die Ergebnisse der *Analysephase sehr* abstrakt, da der meiste Aufwand im Inneren der *"Black-Box"* KBase getrieben werden muß und nach außen nicht sichtbar ist. Um für die folgenden Schritte des Entwicklungsprozesses einen präziseren Entwurf zu ermöglichen, mußte eine Reihe grundlegender Konzepte geklärt werden.

Einige dieser Konzepte basieren auf Ansätzen aus den Vorgängerversionen. Allerdings wurden diese damals weder ordentlich dokumentiert noch optimal implementiert. Durch meine Überlegungen zu den internen Strukturen deckte ich konzeptionelle Mängel auf und fand elegantere Lösungen. Zusammen mit diesem Kapitel entstanden zahlreiche der im Anhang D genannten Algorithmen. Somit trug dieser Abschnitt wesentlich zu einem tieferen Verständnis der Problematik bei.

3.1.1 Verwaltung der Variablen- und Regelmengen

Das *Object Model* aus Abbildung 3 (Seite 14) zeigt, daß eine Wissensbasis eine beliebige Menge von Variablen und Regeln enthalten kann. Zur Verwaltung dieser beiden Mengen wird eine *Container-Struktur* benötigt. Insbesondere muß es der Wissensbasis möglich sein, Variablen und Regeln mit bestimmten Eigenschaften gezielt suchen zu können. Die neu eingeführten Klassen VarList und RuleList dienen als Ansprechpartner für solche Operationen. Eine Wissensbasis enthält jeweils genau eine Instanz dieser Klassen.

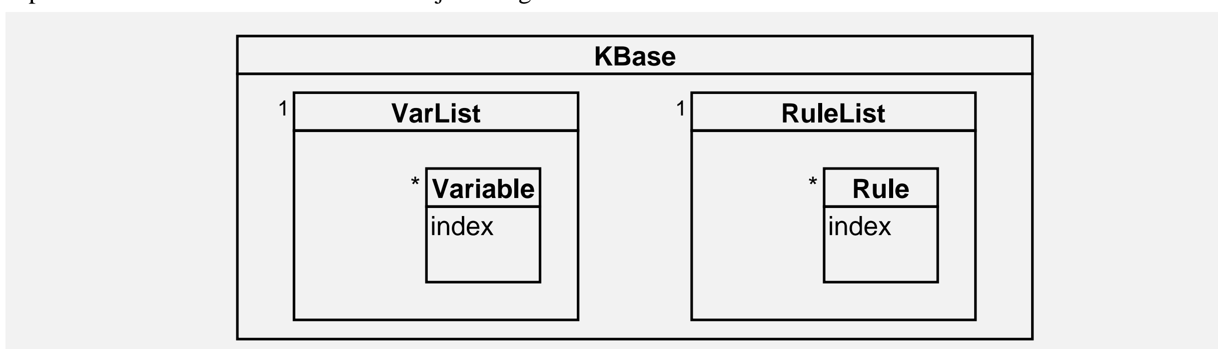


Abbildung 5: Erweiterung des *System Object Model* um die Klassen VarList und RuleList

Die Listen sind anfangs leer und können Objekte der Klassen Variable und Rule aufnehmen. Diese werden intern in der Reihenfolge ihrer Definition sortiert. Dadurch sind sie über einen eindeutigen, bei 0 beginnenden *Index* von außen adressierbar. Zugleich kann eine Variable und eine Regel durch das eigene Attribut index feststellen, wie es von der Shell referenziert wird. Dies vereinfacht zahlreiche Algorithmen und die Kommunikation mit der Shell enorm. Es muß allerdings darauf geachtet werden, daß die Indizes beim Löschen von Variablen und Regeln angepaßt werden. Abbildung 5 zeigt das erweiterte Diagramm der Klasse KBase.

3.1.2 Darstellung der unterschiedlichen Variablentypen

In der *Analysephase* wurden die Unterschiede zwischen den vier Variablentypen vernachlässigt. Statt dessen wurde abstrahiert, daß Variablen über eine mindestens zweielementige Menge von Werten verfügen. Für die nachfolgenden Überlegungen zur Speicherung von Regeln ist es notwendig, dies zu präzisieren. Hierzu werden die in Abbildung 6 definierten Klassen eingeführt. Die Klasse Value wird zur abstrakten Klasse, von der vier Unterklassen abgeleitet werden. Jede Variable darf allerdings nicht Instanzen unterschiedlicher Value-Klassen aggregieren. Da später für die einzelnen Variablentypen unterschiedliche Methoden benötigt werden, erhält auch die Klasse Variable vier (hier nicht dargestellte) Unterklassen. Zur Ermittlung des Erwartungswertes von Variablen muß jeder Wert zudem über ein utility-Feld verfügen, das bei *Number*- und *Interval*-Variablen allerdings nicht veränderbar ist.

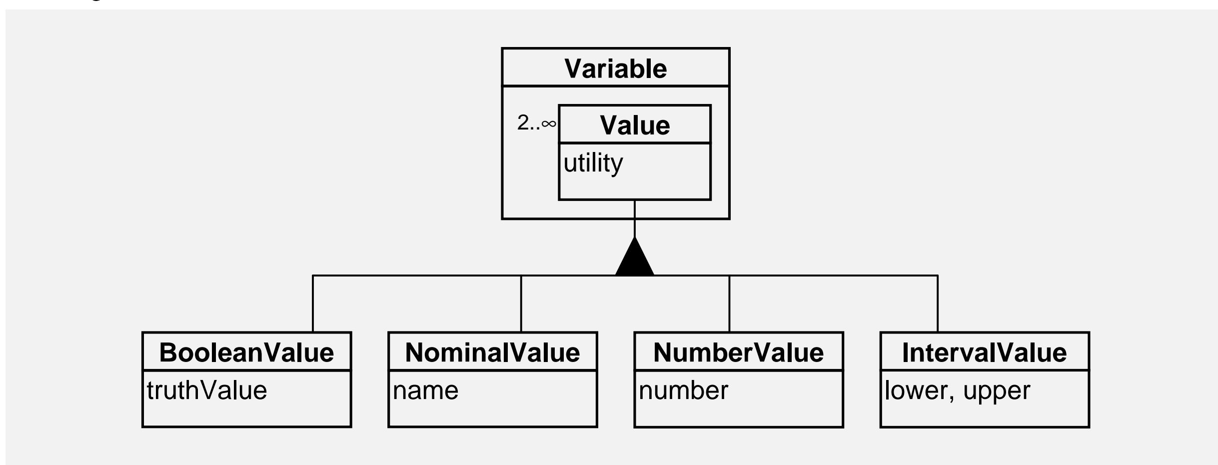


Abbildung 6: Klassenmodell zur Darstellung der Variablentypen

3.1.3 Interne Organisation einer LEG

Eine LEG besteht aus einer nichtleeren Menge von Variablen und einer Tabelle aus Wahrscheinlichkeiten für alle *Konfigurationen* über die Variablen. Wie in Abbildung 7 angedeutet, wird für diese Tabellen lediglich ein *Array* aus positiven double-Werten benötigt, die über Indizes adressiert werden können.

Variablen:
A (*Boolean*)
B (*Nominal*, ternär)
C (*Nominal*, binär)

Tabellengröße:
|**A**| * |**B**| * |**C**| =
2 * 3 * 2 = 12

Index der Konfiguration
A=0 ∧ **B=1** ∧ **C=1 :**
0*1 + 1*|**A**| + 1*|**A**|*|**B**| =
0 + 2 + 6 = 8

Index	A	B	C	P
0	0	0	0	0.125
1	1	0	0	0.025
2	0	1	0	0.301
3	1	1	0	0.173
4	0	2	0	0.031
5	1	2	0	0.000
6	0	0	1	0.088
7	1	0	1	0.091
8	0	1	1	0.002
9	1	1	1	0.019
10	0	2	1	0.103
11	1	2	1	0.042
			Σ	1.000

Abbildung 7: Interne Darstellung einer LEG-Tabelle

Eine LEG besteht also aus einer Liste von Variablen und einem Array aus double-Werten. Die interne Speicherung und Organisation dieses Arrays sollte allerdings gemäß dem Geheimnisprinzip nicht nach außen sichtbar sein. Daher sollte die später zu implementierende Klasse LEG über Methoden verfügen, die den Zugriff auf ausgewählte Konfigurationen ermöglichen. Beispielsweise wäre eine Funktion denkbar, die alle Konfigurationen aufsummiert, bei denen eine bestimmte Variable einen spezifizierten Wert annimmt.

3.1.4 Interne Darstellung von Regeln

Um die Intuition des Benutzers bei der Definition der Regeln und Fakten zu unterstützen, läßt SPIRIT eine sehr ausdrucksstarke Syntax für Regeln zu (sh. Anhang A). Die Shell liefert eine vom Anwender eingegebene Regel

als String an die Wissensbasis, die zunächst Syntax und Semantik der Regel mit einem Parser überprüft und dann für eine effiziente, interne Darstellung derselben verantwortlich ist.

SPIRIT stellt Regeln auf mehrere Arten dar:

- entsprechend der Benutzereingabe als String,
- als Baum von Ausdrücken,
- in kanonischer Disjunktiver Normalform (DNF), und
- falls es eine passende LEG gibt: Mit Listen der enthaltenen Konfigurationen in der zugeordneten LEG.

In allen vier Fällen müssen Prämisse und Conclusio einer Regel getrennt verwaltet werden, da sie unterschiedliche Funktionen ausüben. Bei Fakten ist die Prämisse entsprechend leer.

Die Regeldarstellung mit Ausdrucksbäumen ist die allgemeinste Speicherungsform, da die drei anderen Formen aus ihr rekonstruiert werden können. Der folgende Unterabschnitt geht daher genauer auf Ausdrucksbäume ein.

3.1.4.1 Regeldarstellung mit Ausdrucksbäumen

Jede Regel in SPIRIT kann als rekursive Struktur aus Teilausdrücken verstanden werden. Teilausdrücke sind hierbei geklammerte, über Junktoren verknüpfte andere Teilausdrücke. Zur Einführung in diese Darstellungsform zeigt Abbildung 8 ein Beispiel.

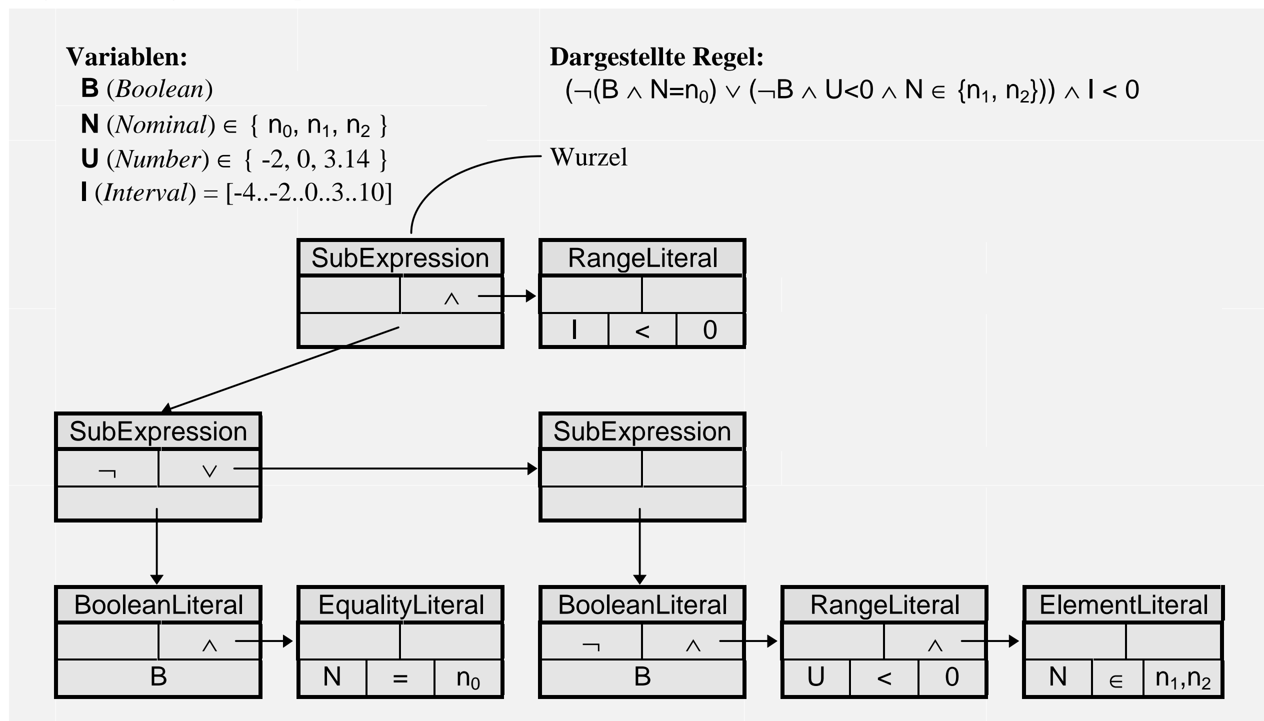


Abbildung 8: Beispiel zur Darstellung von Regeln mit Ausdrucksbäumen

Wie im Beispiel und in Abbildung 10 angedeutet, werden zur internen Darstellung der Ausdrucksbäume neue Objektklassen eingeführt. Jede Regel besitzt für die Regelseiten eine oder zwei Instanzen der Klasse Expression. Eine Expression ist ein Unterausdruck, der über einen Junktor (Disjunktion oder Konjunktion) mit einer anderen Expression verknüpft werden kann. Diese beliebig langen Verknüpfungen sind von links nach rechts zu interpretieren. Außerdem kann jede Expression in negierter Form vorliegen.

Die Klasse Expression ist abstrakt, d.h. jede Instanz muß entweder eine SubExpression oder ein Literal sein. Eine SubExpression stellt einen geklammerten Unterausdruck dar und verfügt dazu über genau einen Verweis auf eine (sub-) Expression: den linken Teil einer Kette von Tochterausdrücken. Hat eine solche Kette die Länge eins, so stellt sie eine redundante Klammerebene dar, die unter Berücksichtigung der Negation gestrichen werden kann.

Literale sind die atomaren Ausdrücke in einer Regel. Jedes Literal bezieht sich auf genau eine Variable, über die es eine Aussage macht. Entsprechend den verschiedenen Operatoren und Variablentypen werden vier neue Unterklassen der abstrakten Klasse Literal eingeführt, die in Abbildung 9 zusammengestellt sind.

Unterklasse	Beispiel	Verknüpft mit	Operator	Operator bei Negation

BooleanLiteral	B	BooleanValue	(is true)	(is false)
EqualityLiteral	$N = n_0$	Nominal-, NumberValue	=	≠
RangeLiteral	U < 0	Number-, IntervalValue	<	≥ / >
ElementLiteral	$N \in \{ n_1, n_2 \}$	Nominal-, NumberValue	∈	∉

Abbildung 9: Unterklassen der Klasse Literal

In der in Abbildung 10 dargestellten Klassenhierarchie sind die Unterklassen von Literal nicht enthalten. Allerdings werden diese und die übrigen Erweiterungen des *Object Models* in den späteren *Inheritance Graphs* (Abschnitt 3.4) ohnehin noch zusammenfassend dargestellt.

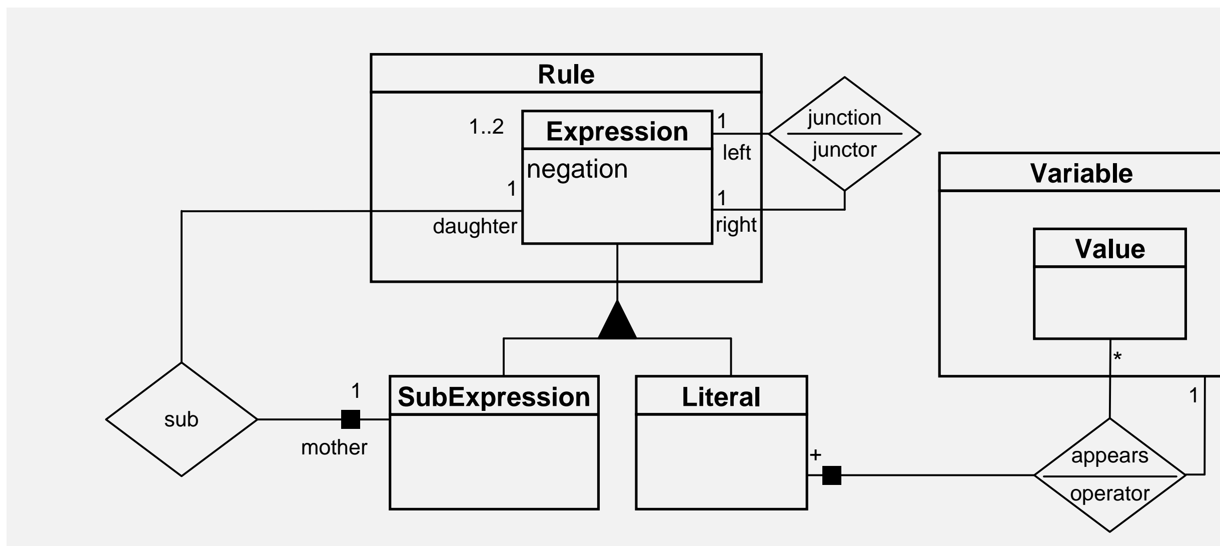
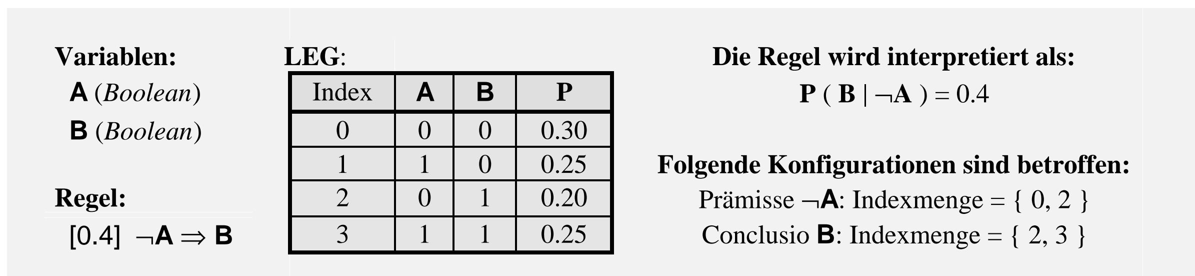


Abbildung 10: Erweiterung des *Object Model* zur internen Regeldarstellung

Naheliegend bei dieser Darstellungsmethode von Regeln ist der Gedanke, die Umwandlung des Regel-Strings mit einem *Recursive-Descent-Parser* zu erledigen. Dieser Ansatz wurde in der späteren Implementierung tatsächlich verwirklicht (sh. hierzu auch Abschnitt 3.1.7). Außerdem können in dieser Form gespeicherte Regeln sehr elegant in eine Disjunktive Normalform umgewandelt werden, die zur effizienten Auswertung benötigt und im folgenden Unterabschnitt vorgestellt wird.

3.1.4.2 Regeldarstellung in Disjunktiver Normalform und mit Index-Listen

Regeln können als Mengen von Konfigurationen über die vorkommenden Variablen angesehen werden. In Abbildung 11 wird der Zusammenhang zwischen Regeln und LEGs verdeutlicht. Zwischen den beiden booleschen Variablen A und B wird hier über die angegebene Regel eine Beziehung hergestellt, die in der gezeigten LEG über A und B repräsentiert wird. Jede Regelseite beschreibt demnach eine Teilmenge der Konfigurationen der LEG. Zum Lernen einer Regel und zur Ermittlung der bedingten Wahrscheinlichkeiten wird auf diese Konfigurationen mit Multiplikation und Summation zugegriffen. Es bietet sich an, die Listen der Konfigurationen für diese Aufgaben bei Bedarf zu ermitteln und dann dauerhaft zu speichern. Hierzu wird die Klasse IndexList eingeführt. Eine IndexList beschreibt eine Teilmenge der Konfigurationen einer LEG. Zur effizienten Erzeugung dieser Index-Listen bietet sich die Regelspeicherung in kanonischer Disjunktiver Normalform an.



Variablen:
A (*Boolean*)
B (*Boolean*)

Regel:
[0.4] ¬A ⇒ B

LEG:

Index	A	B	P
0	0	0	0.30
1	1	0	0.25
2	0	1	0.20
3	1	1	0.25

Die Regel wird interpretiert als:
P (B | ¬A) = 0.4

Folgende Konfigurationen sind betroffen:
Prämisse ¬A: Indexmenge = { 0, 2 }
Conclusio B: Indexmenge = { 2, 3 }

Abbildung 11: Zusammenhang zwischen Regeln und LEGs

Ausgangspunkt für die hier vorgestellte Regeldarstellung sind Ausdrucksbäume, die bereits in (nicht-kanonischer) DNF vorliegen. Ein Verfahren zur Regelumwandlung nach DNF wird als Algorithmus RuleTransformToDNF im Anhang D erläutert. Nach Vervollständigung der Konjunkte um die nicht genannten Variablen entsteht eine *kanonische* DNF. Eine Speicherungsform derselben wird in Abbildung 12 vorgestellt. Zur eindeutigen Speicherung einer Regel wird demnach für jede Regelseite eine Liste der enthaltenen Konjunkte der kanonischen DNF benötigt. Hierzu wird die Objektklasse DNF eingeführt. Sie besteht hauptsächlich aus einer Liste von Integer-Werten, die als Indizes zu einer (imaginären) LEG-Tabelle interpretiert werden können - der LEG aus den in der Regel vorkommenden Variablen. Aus den Indizes lassen sich mit Division und Modulo-Berechnung problemlos die einzelnen Variablenwerte zurückgewinnen.

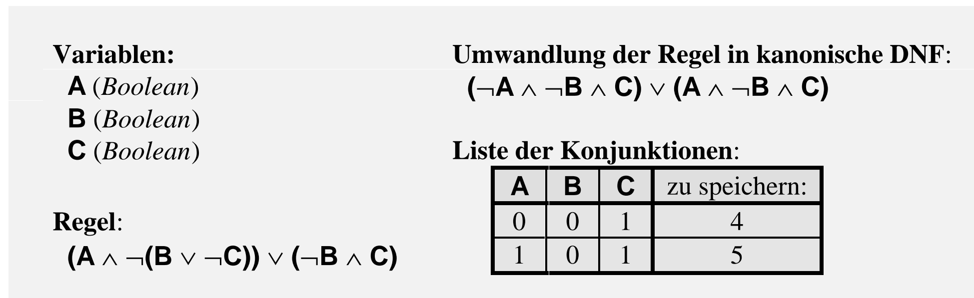


Abbildung 12: Regeldarstellung in kanonischer DNF

Somit ergibt sich das in Abbildung 13 dargestellte Klassenmodell. Jede Regel hält für ihre Regelseiten eine DNF. Wird eine Regel einer LEG zugeordnet (assigned), so werden aus den DNFs IndexLists ermittelt, über die auf die LEG zugegriffen werden kann. Selbstverständlich müssen die Index-Listen, die an einer assigned-Relation beteiligt sind, an die entsprechende Regel aggregiert sein. Außerdem werden Index-Listen dann ungültig, wenn sich die zugewiesene LEG oder die DNF geändert hat (bspw. durch Löschen eines Variablenwertes).

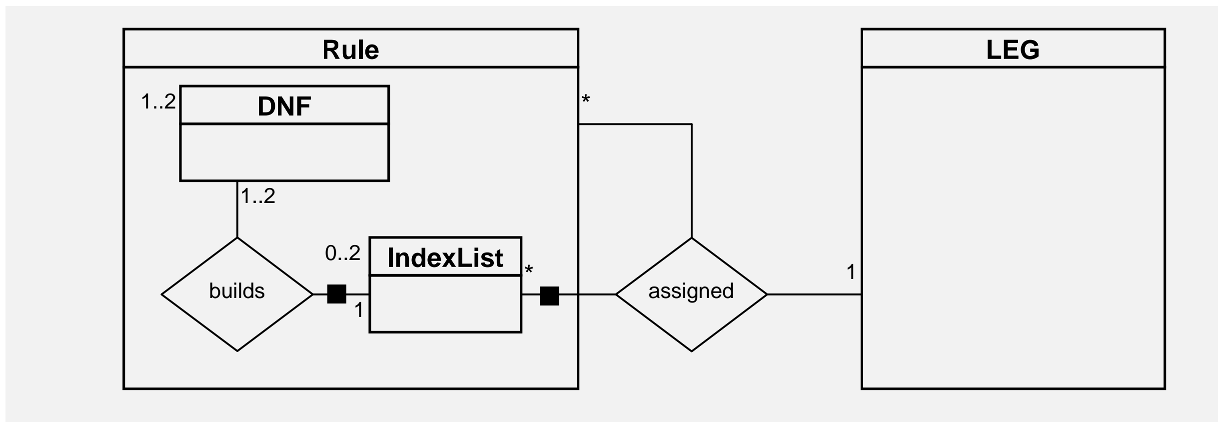


Abbildung 13: Erweiterung des *Object Models* zur Verknüpfung von Regeln und LEGs

Objekte der Klasse IndexList können zur Effizienzsteigerung auch an anderen Stellen eingesetzt werden, nämlich immer dort, wo *mehrfach* auf bestimmte LEG-Konfigurationen zugegriffen werden muß. Außerdem lassen sich Index-Listen optimieren, indem diejenigen Konfigurationen gestrichen werden, deren Wahrscheinlichkeit in der LEG null ist (Nullen haben auf Summation und Multiplikation in vielen Algorithmen keinen Einfluß). Diese Optimierungen sind für spätere Versionen vorgesehen.

3.1.5 LEG-Wälder und Iterationsprozeß

Jede Wissensbasis verfügt über genau eine azyklische Struktur von LEGs (sh. Abbildung 14). Sind zwei LEGs benachbart, so bilden ihre Variablenmengen eine nichtleere Schnittmenge. Diese Schnittmengen werden *Separatoren* genannt. Jede LEG kann beliebig viele Nachbarn haben, über die sie über die Relation adjacent benachbart ist. Da diese Nachbarschaftsrelation eine Baumstruktur beschreibt, kann jede beliebige LEG als Wurzel und Einstiegspunkt von Traversierungsalgorithmen angesehen werden. Die Menge aller LEG-Bäume wird in der Klasse LEGForest verwaltet. Diese wird insbesondere daher eingeführt, weil einige Operationen auf der Gesamtheit der LEG-Bäume agieren und hierzu ein *Controller*-Objekt benötigen.

Der Iterationsprozeß dient zum *Lernen* der aktiven Regeln in die aktuellen LEG-Tabellen. Dazu werden die durch die Regeln angegebenen Konfigurationen mit bestimmten Faktoren multipliziert (vgl. Abbildung 11 und

Algorithmus RuleLearn). Wurde eine LEG durch das Lernen von Regeln verändert, so ist sie i.a. mit seinen Nachbarn nicht mehr konsistent, d.h. die Randwahrscheinlichkeiten der Variablen im Schnitt sind ungleich. Daher müssen nach einem Lernschritt die Nachbarn *kalibriert* werden. Vom informationstheoretischen Standpunkt aus läßt sich das so erklären, daß eine Regel neue Informationen in eine LEG eingebracht hat, und diese Informationen in den Rest des LEG-Baumes verteilt werden müssen.

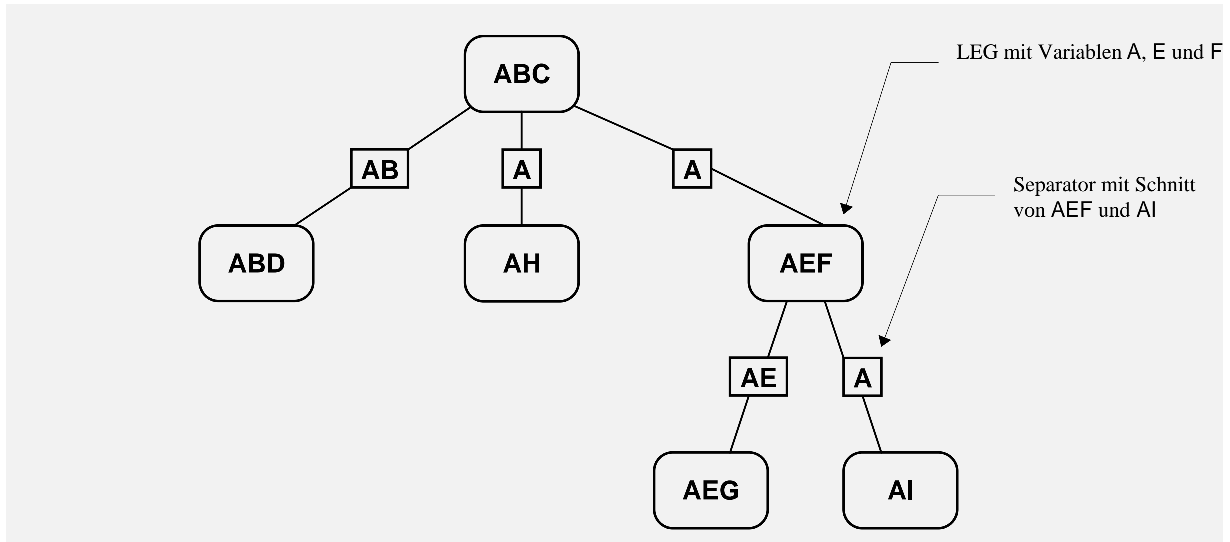


Abbildung 14: Ein einfacher LEG-Baum

Hierzu dient der Iterationsprozeß. Beginnend in der Wurzel (im Beispiel: ABC), werden zunächst die passenden Regeln gelernt. Dies wird in sog. *lokalen Iterationen* bewerkstelligt. Diese enden, wenn die Lernschritte keine weiteren Informationsgewinne (oder Entropieänderungen) bewirkten. Dann kann zur ersten Tochter-LEG gewechselt werden (hier: ABD). Diese kalibriert sich über den Separator (hier: AB) mit der ersten LEG und führt dann ihrerseits lokale Iterationen durch. Ist ein Unterbaum abgearbeitet, so schickt er die neuen Informationen zurück an die Mutter-LEG, die dann mit der nächsten Tochter weitermacht, etc. Eine globale Iteration ist beendet, wenn die Wurzel komplett aus allen Töchtern Informationen aufgesammelt hat. Bewirkte eine globale Iteration keine weiteren Entropieänderungen, so kann der Lernprozeß für diesen LEG-Baum beendet werden.

Die Steuerung des Iterationsprozesses, der in der Praxis durchaus einige Minuten dauern kann, wird von einer Instanz der Klasse IterationManager vorgenommen. Sie muß in der Lage sein, den Vorgang in kürzere Berechnungsschritte aufzuteilen, um einen von der Shell kontrollier- und darstellbaren Verlauf zu ermöglichen. Wünscht die Shell einen Iterationsprozeß, so wird intern ein IterationManager angelegt. Ruft die Shell daraufhin die stepIteration-Funktion auf, so wird der in der Iteration folgende kleinstmögliche Teilschritt ausgeführt und dann zur Shell zurückgekehrt. Dann ist diese in der Lage, den Zustand der Iteration über die Funktion getIterationStatus abzufragen. Dadurch kann die Shell jederzeit entscheiden, ob die Iteration fortgesetzt oder beendet werden soll. Da die LEGs während des Iterationsprozesses nicht immer konsistent sind, muß der Iteration-Manager bei Abbruch der Iteration dafür sorgen, daß die zur Konsistenz erforderlichen Schritte noch ausgeführt werden. Für die Implementierung bedeutet dies, daß der rekursive Algorithmus RuleLearnIteration in einen iterativen Prozeß umgeformt werden muß. Die hierzu nötigen Details werden an dieser Stelle nicht weiter ausgeführt.

Nun noch zwei wichtige Überlegungen zur Performancesteigerung. Das primäre Optimierungsziel im Projekt SPIRIT ist die Ablaufgeschwindigkeit. Geschwindigkeitsoptimierungen haben also z.B. Vorrang vor einem geringen Speicherverbrauch. Dies betrifft insbesondere die Operationen auf LEGs, die mit großen Zahlenmengen arbeiten.

Ein sehr häufig auftretender Vorgang ist die Kalibrierung benachbarter LEGs. Bei der Kalibrierung (sh. auch Algorithmus LEGCalibrate) werden in einer LEG die Randsummen der Variablen im Separator gebildet, durch die entsprechenden Randsummen in der anderen LEG geteilt und dann auf die dort passenden Konfigurationen multipliziert. Dadurch haben die Schnitt-Variablen in beiden LEGs dieselben Randwahrscheinlichkeiten. Es lassen sich hierbei zwei Schritte optimieren. Erstens können die Konfigurationen, die bei der Bildung der Randverteilungen benötigt werden, über (einmal ermittelte) Index-Listen gespeichert werden, und zweitens muß die Randverteilung einer LEG nicht immer neu berechnet werden, sondern kann (aufgrund der Baumtraversierung im Iterationsprozeß) aus den vorangegangenen Schritten übernommen werden. Somit enthält ein Separator immer die Informationen aus *dem* Nachbarn, von dem aus er im Baumdurchlauf zuletzt durchlaufen wurde.

Auch die Zuweisung und Zurücknahme von Evidenzen sollte laut Anforderungsdokument besonders schnell möglich sein. Daher wird jede LEG um eine Tabelle (copyP) erweitert, in die die originale Verteilung (ohne Evidenzzuweisungen) kopiert werden kann. Vor der ersten Evidenzzuweisung wird diese Kopie mit den aktuellen Wahrscheinlichkeiten gefüllt. Werden dann Evidenzen zugewiesen, so können die Tabellen der LEGs beliebig mit neuen Werten geändert werden, da die originalen Verteilungen jederzeit aus der Kopie wiederhergestellt werden können. Dies muß vor neuen Lernprozessen und bei Zurücknahme von Evidenzen geschehen. Die Kopien verlieren ihre Gültigkeit, wenn die Wissensbasis zurückgesetzt (reset) oder die LEG-Struktur geändert wird.

Zusammenfassend wird das *Object Model* um die in Abbildung 15 gezeigten Klassen erweitert. Ein LEGForest verfügt über eine Menge von Wurzel-LEGs, die als Einstiegspunkt für Baumdurchläufe dienen. Es gibt zwei Arten von LEGs: normale LEGs und Separatoren, wobei in Separatoren keine Regeln gelernt werden können. Beide LEG-Arten sind mit einer nichtleeren Variablenmenge verknüpft (in) und verfügen über eine Tabelle mit den aktuellen Wahrscheinlichkeiten (p). LEGs verfügen zudem über eine temporäre Kopie dieser Tabellen (copyP) zur schnellen Rücknahme von Evidenzzuweisungen. Je zwei LEGs sind über einen Separator benachbart (adjacent), wobei eine (parent-) LEG mehrere Söhne haben kann und eine Baumstruktur entstehen muß. Jeder Separator verfügt über Index-Listen zur schnellen Berechnung seiner Wahrscheinlichkeitstabelle aus den beiden verbundenen LEGs.

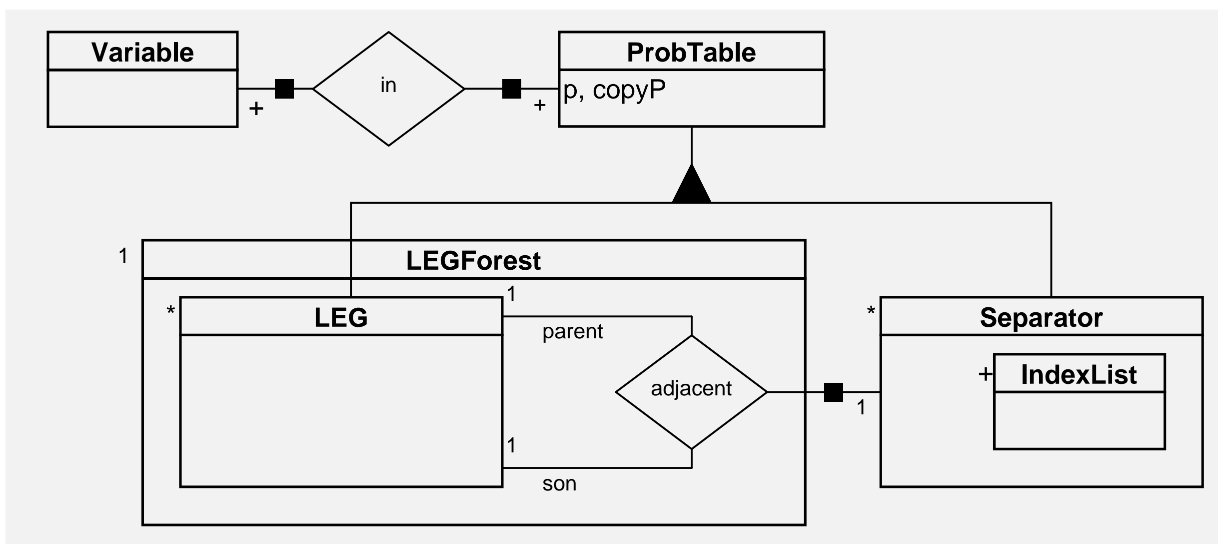


Abbildung 15: Klassen zur Darstellung des LEG-Waldes

3.1.6 Umsetzung einiger Relationships

Im *System Object Model* sind Klassen und Beziehungen dargestellt. Diese Beziehungen befinden sich aus abstrakter Sichtweise "zwischen" den Objekten. Da die Entwurfsphase nach FUSION mit einer implementierbaren Sammlung von Objektklassen enden soll, muß geklärt werden, wie Relationships in dieser Klassenhierarchie tatsächlich umgesetzt werden. FUSION selbst schlägt zur Ermittlung dieser Umsetzungen die *Visibility Graphs* (sh. Abschnitt 3.3) vor, in denen die Sichtbarkeitsbeziehungen zwischen den einzelnen Klassen dargestellt werden. Da diese Graphen allerdings erst nach den *Object Interaction Graphs* (OIGs) aus dem folgenden Entwurfsschritt produziert werden, können deren Ergebnisse erst sehr spät mitberücksichtigt werden. Meiner Ansicht nach sollten *Relationships* (soweit möglich) bereits in den frühen Entwurfsphasen präzisiert werden, damit ein möglichst direkter Bezug zwischen OIGs, den *Class Descriptions* und dem späteren Programmcode besteht.

Die meisten bisher vorgestellten Beziehungen lassen sich recht einfach durch Hinzunahme eines (*Link*-) Attributes realisieren. Allerdings ist es insbesondere bei dynamischen, bilateralen Relationships aufwendig und fehleranfällig, diese Link-Attribute jeweils auf dem neuesten Stand zu halten. Im folgenden möchte ich Lösungsansätze hierzu an den drei wichtigsten Beziehungen des Modells vorstellen. Es sind dies die Beziehungen in, assigned und appears, die den Zusammenhang zwischen Variablen, Regeln und LEGs repräsentieren. Dabei wird in allen drei Fällen zunächst untersucht, in welchem Kontext ein Zugriff auf die Relation aus Sicht der beteiligten Objekte überhaupt nötig oder empfehlenswert ist.

3.1.6.1 Beziehungen zwischen Variablen und Regeln (appears)

Zum Zweck der Performancesteigerung sollte eine Variable die Menge der Regeln kennen, in der sie verwendet wird. Beispielsweise kann somit sehr leicht die Art der Kante zwischen zwei Variablen in einem Variablengraphen ermittelt werden (vgl. Definition der Graphen in Abschnitt 1.2.3). Dies hätte den Vorteil, daß die Graphen überhaupt nicht im Speicher verwaltet werden müssten, sondern bei Bedarf dynamisch erzeugt werden könnten. Ein weiterer Vorteil der Zugriffsmöglichkeit von Variablen auf ihre Regeln wäre, daß bei Änderungen in der Variablenmenge (z.B. Löschen und Hinzufügen von Werten) direkt die betreffenden Regeln mitgeändert werden können. Jede Variable sollte demnach eine Liste der Regeln besitzen, in der sie ein Vorkommen hat. Hierzu wird der Klasse Variable im Klassenmodell ein Attribut namens inRules zugeordnet, das später im wesentlichen durch einen Vektor aus Verweisen auf Regeln umgesetzt werden kann. Diese inRules-Listen müssen bei Änderungen in der Regelmenge allerdings ebenfalls aktualisiert werden.

Zu diesem Zweck sollte jede Regel schnell herausfinden können, welche Variablen in ihr vorkommen. Hierzu muß sie entweder den Ausdrucksbaum durchlaufen oder einfach die DNF-Darstellung zu Rate ziehen, die ohnehin über eine Variablenliste (vars) verfügen muß. Die Beziehung appears wird also durch Hinzunahme einer Regelliste für jede Variable bilateral effizient modelliert.

3.1.6.2 Beziehungen zwischen Regeln und LEGs (assigned)

Jede Regel paßt in eine (eventuell leere) Teilmenge der LEGs. Allerdings ist aus Sicht der Regel nur die kleinste dieser LEGs interessant, da in dieser LEG das Lernen und Ermitteln der aktuellen bedingten Wahrscheinlichkeiten stattfindet. Somit reicht es, jeder Regel ein Attribut assignedLEG zuzuordnen, das (falls die Regel *aktiv* oder *aktivierbar* ist) die minimale LEG angibt, in das die Regel paßt.

Für den Iterationsprozeß und für das Re-Init muß eine LEG leg aus der Menge der passenden Regeln nur diejenigen kennen, die auch tatsächlich in sie gelernt werden, für die also assignedLEG = leg gilt. Hierzu verfügt sie über einen Vector aus Regeln, der mit rules bezeichnet wird. Die assigned-Relation wird demnach ebenfalls beidseitig realisiert.

3.1.6.3 Beziehungen zwischen Variablen und LEGs (in)

Bereits per Definition verfügt jede LEG über die Liste der Variablen, die in ihr repräsentiert werden. Im erweiterten *Object Model* wird diese Verbindung über die Klasse ProbTable (vgl. Abbildung 15) realisiert.

Variablen sollten für Evidenzzuweisungen und zur Ermittlung der aktuellen Wahrscheinlichkeiten der Variablenwerte direkt auf eine (möglichst kleine) LEG zugreifen können, in deren Variablenliste sie erscheint. Dazu erhalten Variablen das Attribut inLEG, das eine ständig aktuelle und eindeutige Zuordnung zu einer LEG bereitstellt. Auch die Beziehung in wird somit in beiden Richtungen dauerhaft gespeichert, allerdings nicht als N-zu-N-Beziehung, sondern als 1-zu-N-Beziehung.

3.1.7 Effiziente Darstellung der Variablengraphen

Die beiden Graphen über der Variablenmenge (der gemischte und ungerichtete Graph, sh. Abschnitt 1.2.3) dienen der Visualisierung der Beziehungen zwischen den einzelnen Variablen. Diese Beziehungen werden durch die Regelmenge eindeutig bestimmt. Die Funktionsschnittstelle von SPIRIT stellt drei graph...-Methoden bereit, die zur vollständigen Darstellung der Graphen ausreichen. Es wäre dabei allerdings äußerst ineffizient, bei jedem Funktionsaufruf eine dynamische Neuberechnung der Ergebnisse durchzuführen. Schon für den Test, ob zwei Variablen miteinander verbunden sind, müßten i.a. alle Regeln durchlaufen werden.

Das Problem wird durch die Einführung eines VarGraph-Objektes gelöst. Jede KBase verfügt über maximal zwei solcher Objekte (für die beiden Graph-Typen). Diese werden erst bei Bedarf erstellt und bleiben solange gültig, wie die Struktur des Graphen (durch Löschen und Hinzufügen von Regeln, Variablen und Werten) unverändert bleibt.

3.1.8 Scanner und Parser

SPIRIT soll in der Lage sein, Regeln in Form von Strings einzulesen. Außerdem soll das System Wissensbasen aus Text-Dateien laden können. Für beide Aufgaben wird ein Übersetzer benötigt, der Zeichenketten in Variablen und Ausdrucksbäume von Regeln umwandeln kann. In SPIRIT wird diese Problemstellung in zwei Teilprobleme zerlegt. Zunächst wandelt ein *Scanner* die Zeichenketten in eine Folge von Symbolen (*Tokens*) um. Es gibt hier drei Arten von Tokens: Strings, Zahlen und Sonderzeichen (wie Operatoren und Klammern). Diese Kette von Tokens ist die Eingabe für den zweiten Teilschritt, den *Parser*. Da die Syntax von Regeln und SPIRIT-Dateien recht einfach ist, kann dieser Parser mit einem *One-Step Look-Ahead-Verfahren* implementiert werden. Ausdrucksbäume in dem im Abschnitt 3.1.4.1 eingeführten Format können dann von einem sog. *Recursive-*

Descent-Verfahren mit einem einzigen Durchlauf über die Kette der Tokens elegant erzeugt werden. Die Vorgehensweise solcher Parser sei hier nur kurz angedeutet (für weitere Details, sh. [WM92, S. 285ff], die *Class Descriptions* aus Abschnitt 3.5 und die Quelltexte der Module RuleParser.java und FileParser.java). Für jedes Nichterminal-Symbol X der Grammatik von SPIRIT-Regeln und Dateien (vgl. Anhang A und B) wird eine Prozedur mit Namen X erzeugt. Die rechten Seiten der Ableitungsregeln der Grammatik werden in die Form von Kontrollstrukturen (wie if, case oder while) gebracht, wobei dort vorkommende Symbole in rekursive Funktionsaufrufe umgewandelt werden.

Somit bietet sich die Einführung der neuen Klasse Scanner an. Der Parser fordert vom Scanner Tokens an, dieser liest sie aus einem Eingabe-Strom (z.B. einer Datei) und ermöglicht dadurch den Aufbau der Expression-Bäume.

3.1.9 Fehlerbehandlung

Als letzter Implementierungsaspekt in diesem Projekt ist hier zu klären, wie SPIRIT auf Ausnahme- und Fehlerfälle reagieren soll. *Innerhalb* der Klassenbibliothek ist es zwar nicht völlig zu verhindern, aber immerhin berechenbar, ob und wann ein Fehler auftritt. Keinen Einfluß hat SPIRIT allerdings darauf, welche Funktionen mit welchen Parametern von einer *Shell* aufgerufen werden. Insbesondere ist es hilfreich, den Benutzer auf potentielle Fehlersituationen hinzuweisen und falsche Eingaben zurückzuweisen. Jedenfalls sollte es der Shell unmöglich sein, SPIRIT in einen ungültigen Zustand zu bringen.

Dies bedeutet, daß alle System-Operationen vor der eigentlichen Ausführung auf die Richtigkeit ihrer Parameter geprüft werden müssen. Hierzu gehören einfache Tests, z.B. um zu prüfen, ob ein Regel-Index zulässig ist. Insbesondere müssen aber die *Preconditions* aus den *Assumes*-Klauseln der Operation-Schemata aus der *Analysephase* geprüft werden. Dazu gehört vor allem der Test, ob gerade eine Iteration läuft oder noch Evidenzen zugewiesen sind. Auch komplexere Abfragen, z.B. ob durch Hinzufügen eines Variablenwertes eine zu große LEG-Tabelle entstehen würde, müssen frühzeitig durchgeführt werden. Fehlerhafte Eingaben sollten mit einer Ausnahmebehandlung abgefangen und die Shell über den Fehler unterrichtet werden.

In der Implementierung bietet sich die Verwendung von sog. *Exceptions* an. Diese Sprachkonstrukte werden von den meisten neueren objektorientierten Sprachen unterstützt. Wird in einer Funktion eine *Exception* gemeldet, so werden alle gerade aktiven Funktionsinstanzen abgebrochen, bis ein Befehlsblock die Ausnahmebehandlung übernimmt. Findet sich keine Funktion zum Abfangen einer *Exception* auf dem Stack (d.h. in der Historie der bisherigen Funktionsaufrufe), so wird sogar das Programm vorzeitig terminiert.

Jede System-Operation in SPIRIT, die eine ungültige Eingabe erhalten oder zu einem unzulässigen Zeitpunkt aufgerufen werden kann, sollte also potentiell in der Lage sein, eine *Exception* zu melden. Da *Exceptions* selbst Objekte sind, können sogar weitere Informationen mitgeschickt werden, z.B. ein Fehlercode und zusätzliche Objekte zur Angabe von Details. Hierzu wird später die Klasse KBE (*KBase-Exception*) eingerichtet. Eine Übersicht über die Fehlercodes von SPIRIT befindet sich in Anhang C.

3.2 Object Interaction Graphs

Dieser Abschnitt enthält die *Object Interaction Graphs*, die in der Regel das erste Ergebnis der Design-Phase nach FUSION sind. Der Zweck dieses Entwurfsschrittes ist die Festlegung der erforderlichen Nachrichtenströme zwischen den Objekten zur Erfüllung der abstrakten Funktionsbeschreibungen aus der *Analysephase*. Dazu wird im folgenden zu jeder (relevanten und nicht-trivialen) System-Operation ein Graph und eine kurze Beschreibung in Text-Form gegeben. Hierzu war es nötig, neue Objektklassen einzuführen, die für die Ausführung der Operationen erforderlich sind:

- File: Ein Verweis auf einen Ein-/Ausgabestrom (z.B. eine Datei auf einem Datenträger: dies wird beim Export von Wissensbasen benötigt).
- Object: Platzhalter für beliebige Objekte (z.B. ein Argument, das je nach Variablentyp anders interpretiert wird).

Zu Beginn dieser Entwurfsphase mußte ich im wesentlichen festlegen, wie die System-Operationen intern funktionieren und welche Objekte daran beteiligt sind. Dies war nicht einfach, weil hier tief in interne Details gedacht werden mußte, die in den bisherigen Entwurfsschritten noch nicht berücksichtigt wurden. Da sich die Klassenhierarchien der neuen und der Vorgängerversionen wesentlich unterscheiden, war eine Untersuchung des alten Programmcodes wenig sinnvoll. Lediglich die Erfahrungen aus den alten Versionen konnte genutzt werden.

Für jede Operation war es erforderlich, ein sog. *Controller*-Objekt zu definieren. Ein Controller ist der Einstiegspunkt einer System-Operation. Um das angestrebte Geheimnisprinzip des Systemkernes zu erhalten, legte ich fest, daß *alle* Operationen von einer Instanz der Klasse KBase gesteuert werden. Hierdurch muß die Shell lediglich mit genau *einem* Objekt kommunizieren und sich nicht um die interne Darstellung von Variablen, Regeln und LEGs kümmern. Außerdem haben viele Operationen keine eindeutig abgrenzbare, lokale Wirkung, sondern Seiteneffekte auf andere Objekte des Systems. Beispielsweise bewirkt das Löschen eines Variablenwertes auch die Verkleinerung der LEGs, die die Variable enthalten. Nur die KBase ist garantiert in der Lage, diese Seiteneffekte auszulösen, da sie Zugriff auf alle relevanten Objekte hat. Zudem können fehlerhafte Eingaben, insbes. die Verletzung der *Preconditions*, nur auf oberster Ebene sicher abgefangen werden. Für meinen Geschmack war es also am elegantesten, nur die Klasse KBase als Schnittstelle zur Shell einzurichten und alle anderen Objekte nach außen zu verkapseln. Ein Nachteil dieses Konzeptes ist, daß einige Funktionen nur noch Werte weiterreichen, ohne selbst Einfluß zu nehmen. Außerdem wird die Klasse KBase in ihrem Umfang recht groß und etwas unübersichtlich.

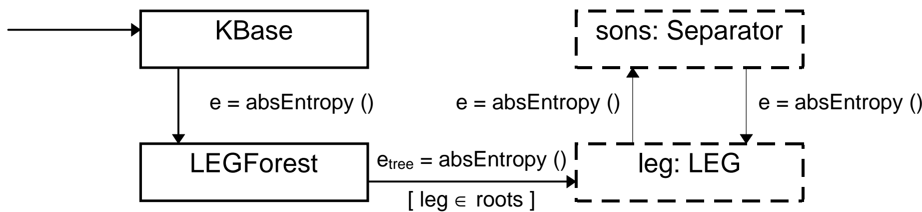
Bei der Entwicklung der OIGs wird davon ausgegangen, daß alle Objekte aufeinander zugreifen können. Diese Annahme vereinfacht den Entwurf, weil nicht auf den Aufbau von Zugriffspfaden geachtet werden muß. Der Nachteil ist allerdings, daß sich die OIGs stark von der späteren Implementierung unterscheiden können und somit keine geeignete Grundlage für die späteren *Class Descriptions* und die Algorithmen zu den Operationen bilden. Um die OIGs mit den folgenden Modellen besser in Einklang zu bringen, habe ich daher bereits bei den OIGs darauf geachtet, daß die Methoden die Zugriffspfade berücksichtigen. Das hat zur Folge, daß organisatorische Parameter, die lediglich den Zugriff auf andere Objekte ermöglichen, in die Funktionsaufrufe mit aufgenommen werden.

Die folgenden Diagramme beschreiben die Funktionsweise der System-Operationen aus der *Analysephase*. Hierbei habe ich die von FUSION vorgeschlagene Form etwas modifiziert. Insbesondere habe ich bei den Parametern von Methodenaufrufen auf die Angabe der Datentypen verzichtet, da die (zahlreichen) in den Graphen auftauchenden Unterfunktionen in den späteren *Class Descriptions* ohnehin noch erläutert werden (Abschnitt 3.5). Andererseits werden die Namen von Objektinstanzen weggelassen, wenn sie dem kleingeschriebenen Klassennamen entsprechen.

Außerdem wird die Tatsache ignoriert, daß fast alle Funktionen Fehler (Exceptions) zurückliefert können. Hierzu sind am Anfang der Funktionen Testschritte nötig, die evtl. weitere Hilfsfunktionen erfordern. Auch muß ich die Darstellung der Diagramme bei einer gewissen Abstraktionstiefe abbrechen, um den gegebenen Rahmen nicht zu sprengen. Dennoch werden auch einige wichtige Unterfunktionen mit eigenen Graphen vorgestellt. Verbergen sich hinter Funktionsaufrufen noch weitere Schritte, die in keinem der Graphen detailliert werden, so wird dies mit (...) angedeutet. Um die Graphen nicht mit irrelevanten Details zu überfrachten wurden Zugriffe auf Containerklassen (z.B. VarList) teilweise weggelassen. Die entsprechenden Stellen sind mit einem Kringel gekennzeichnet (siehe z.B. KBase.exportFile).

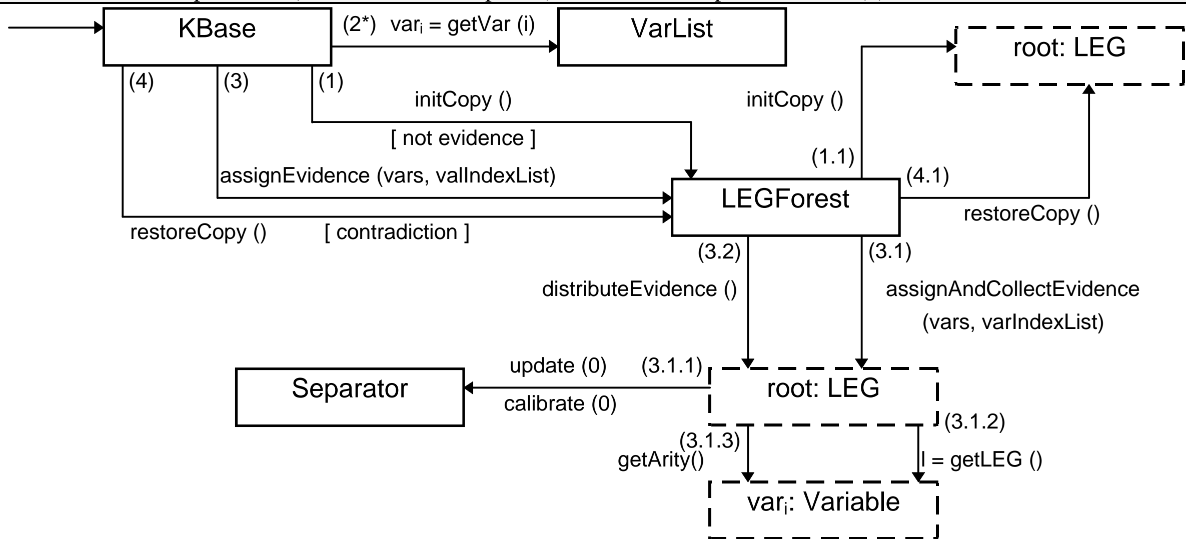
KBase.absEntropy (): double

Errechne die absolute Entropie gemäß Algorithmus LEGForestAbsEntropy:
Baumdurchlauf: addiere die Entropien der LEGs (LEGAbsEntropy) und subtrahiere die Entropien der Separatoren



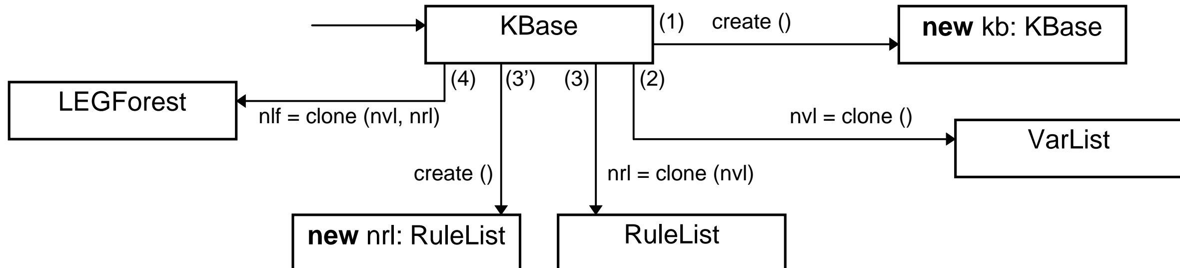
KBase.assignEvidence (varIndexList: int [], valIndexList: int [])

if noch keine Evidenzen zugewiesen **then** Kopiere Verteilung intern (1).
Forme die Liste der Variablen-Indizes in eine Liste von Variablen um (2).
Führe Evidenzzuweisungen auf den LEGs durch gemäß Algorithmus AssignEvidence (3).
Lokale Zuweisungen in den LEGs und "sammeln" der neuen Informationen in Richtung der Wurzeln (3.1).
Rekursion und kalibriere jeweils über den Separator, der ggfs. aktualisiert werden muß (3.1.1).
Für jede Variable aus der angegebenen Liste, die in die jeweilige LEG paßt (3.1.2):
Lösche die Konfigurationen, an denen die Variable *nicht* den angegebenen Wert annimmt (dazu: (3.1.3)).
Verteile die neuen Informationen über Angleichen der LEGs in einem *Top-Down*-Durchlauf in den Baum (3.2).
Tritt dabei ein Widerspruch auf (z.B. eine LEG komplett 0), so stelle die Kopie wieder her (4).



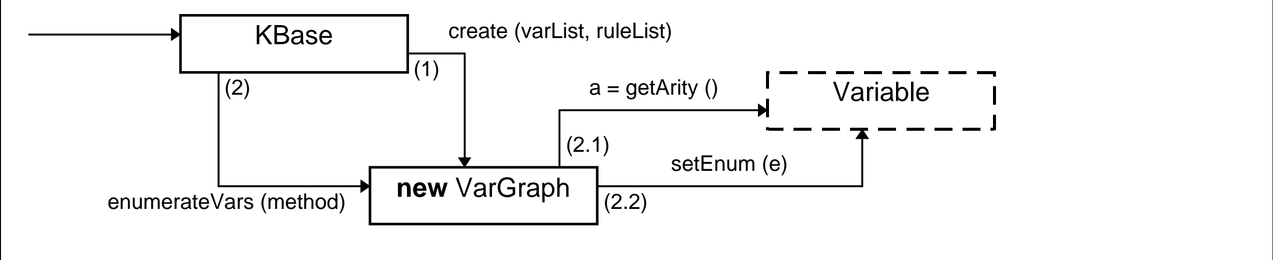
KBase.clone (withRules: boolean): KBase

Erzeuge neue (leere) KBase kb (1).
Erzeuge für kb neue VarList und fülle sie mit allen Variablen (2).
if ¬withRules **then** erzeuge neue (leere) RuleList **else** dupliziere die alte (mit Anpassung der Verbindungen zu Variablen) (3).
Erzeuge für kb neuen LEGForest als Kopie des LEGForests (mit Anpassung der Verbindungen zu Variablen und Regeln) (4).



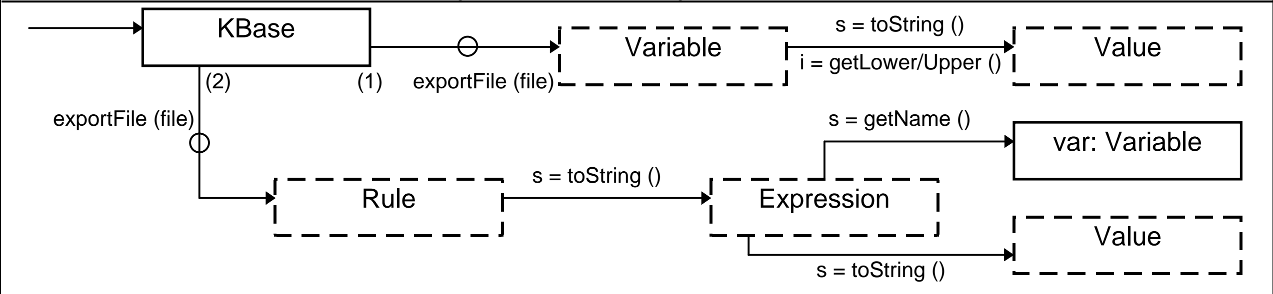
KBase.enumerateVars (method: EnumMethod)

Erzeuge temporären VarGraph (1).
Führe den mit method gewählten Algorithmus VarEnum... auf diesem Graphen aus (2).



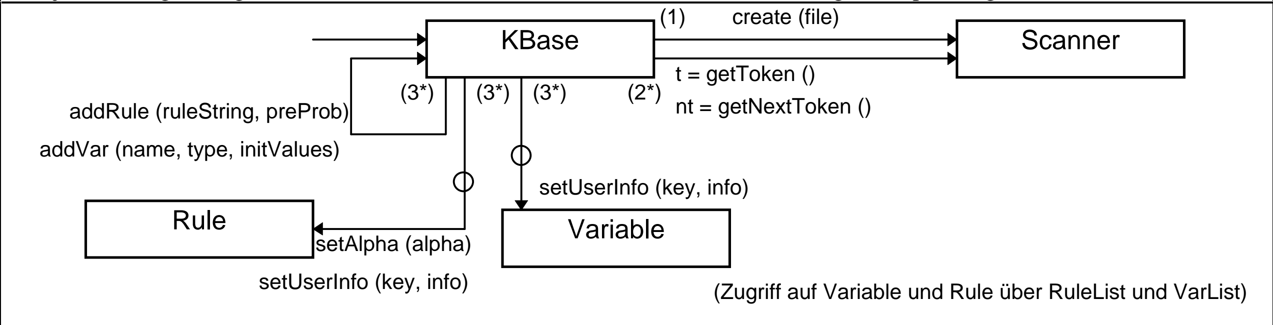
KBase.exportFile (file: File)

Schreibe nacheinander alle Variablen und Regeln in diese Datei (Zugriff über VarList und RuleList) (1), (2).



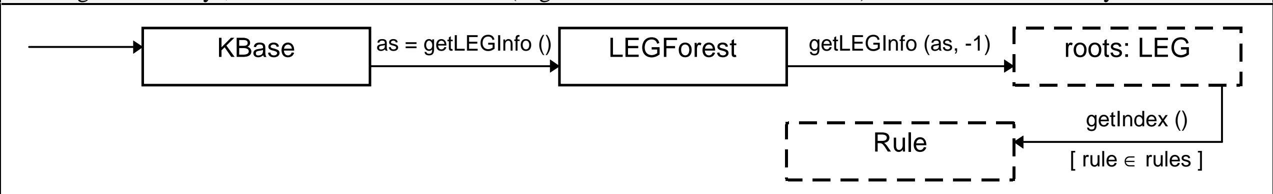
KBase.importFile (file: File)

Erzeuge Scanner für die Datei (1) und verwende ihn zum Einlesen der in der Datei befindlichen Tokens (2).
Je nach den in der Datei befindlichen Deklarationen erzeuge dann neue Variablen und Regeln (3).
Zu jeder erzeugten Regel und Variable können evtl. User-Informationen und (bei Regeln) Alphas zugeordnet werden (3).



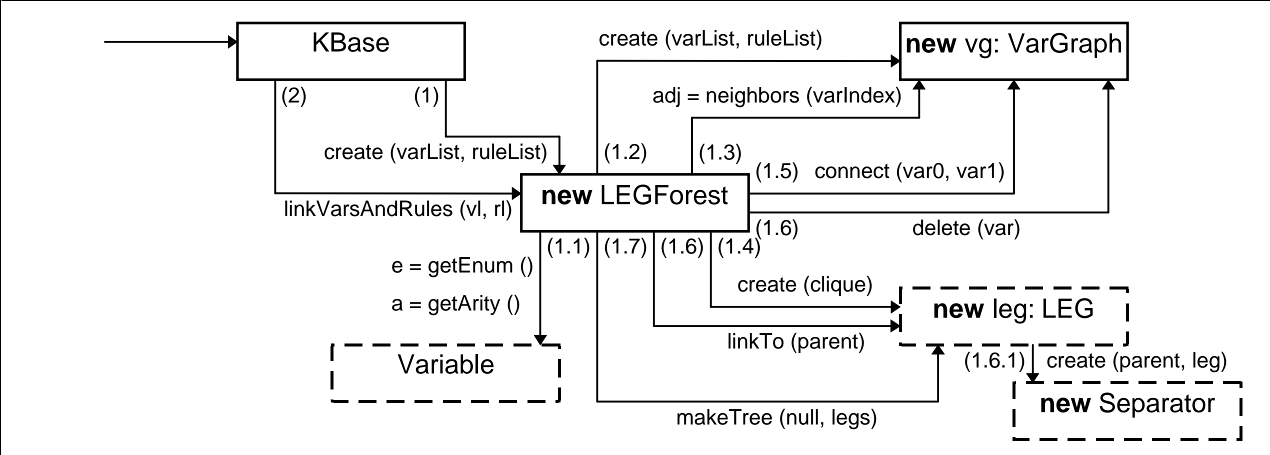
KBase.getLEGInfo (): Vector of (int [])

// Liefert zu jeder LEG ein Array $a[0..n_{leg}]$ mit a[0] = Index der Parent-LEG, a[1] = #Variablen in LEG,
// a[2]..a[2 + a[1]] = Indizes der Variablen in der LEG, a[3 + a[1]...] = Indizes der Regeln in LEG
Erzeuge leere Arrays, durchlaufe die LEG-Bäume (beginnend bei Wurzel mit Parent -1) und fülle dabei die Arrays.



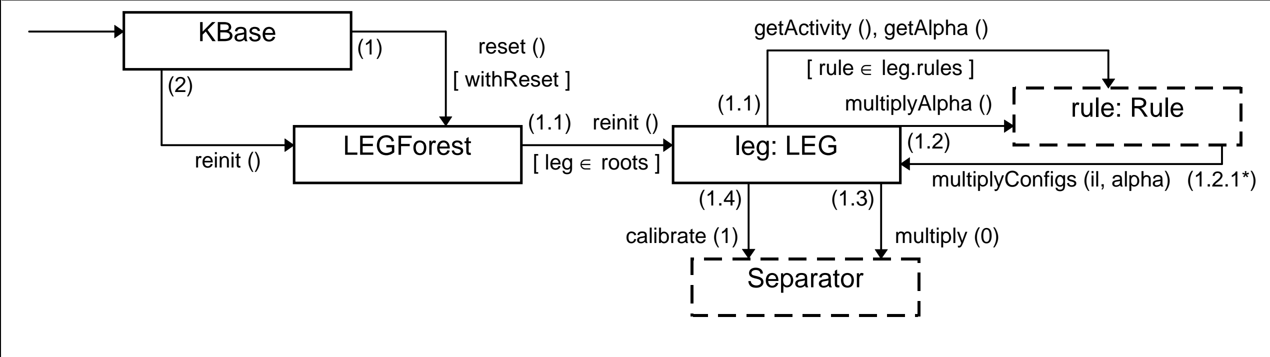
KBase.rebuild ()

Erzeuge neuen LEGForest (1).
Bilde Variablen-Liste gemäß enum-Werte der Variablen (und überprüfe dabei, ob Numerierung gültig ist) (1.1)
Erzeuge einen (temporären) VarGraph aus der aktuellen Variablen- und Regelmenge (1.2).
Gehe vor gemäß Algorithmus LEGForestBuild: Ermittle, verbinde und lösche Knoten im Variablengraphen (1.3), (1.5), (1.6).
Erzeuge dabei neue LEGs, verbinde sie über neue Separatoren und wandle diese Struktur in Bäume um (1.4), (1.6), (1.7).
Falls die neue Struktur erfolgreich erzeugt werden konnte, ersetze den alten LEGForest, ...
... und verknüpfe alle Variablen und Regeln mit den jeweils minimalen LEGs (2).



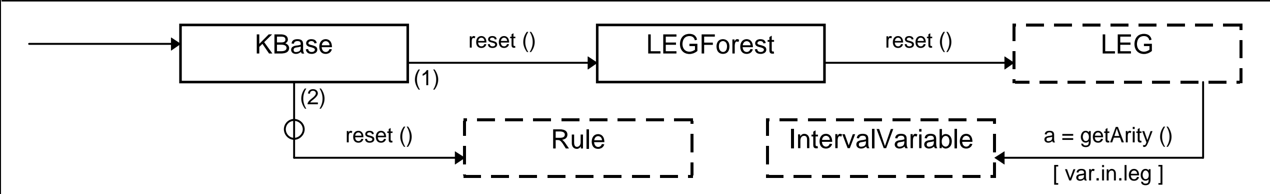
KBase.reinit (withReset: boolean)

if withReset **then** Setze LEGForest auf Gleichverteilung zurück (1)
Führe Re-Init gemäß Algorithmus LEGForestReinit durch (2):
Durchlaufe jeden LEG-Baum, beginnend mit der Wurzel (1.1).
In jeder neu erreichten LEG: Multipliziere die Alphas aller enthaltenen aktiven Regeln mit alpha ≠ 1 (1.2).
Für alle Sohn-LEGs: Multipliziere (1.3) und Rekursion.
Anschließend kalibriere top-down mit Normierung (1.4).



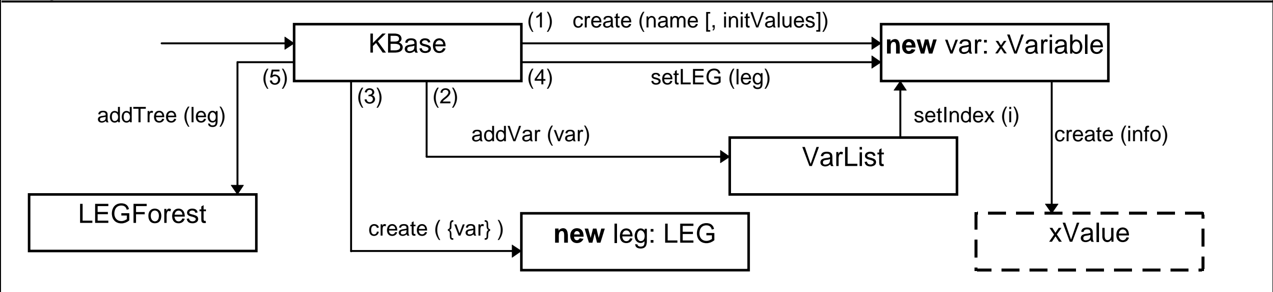
KBase.reset ()

Setze alle LEGs auf Gleichverteilung zurück gemäß Algorithmus LEGReset (1).
Setze alle Regel-Alphas zurück (2).



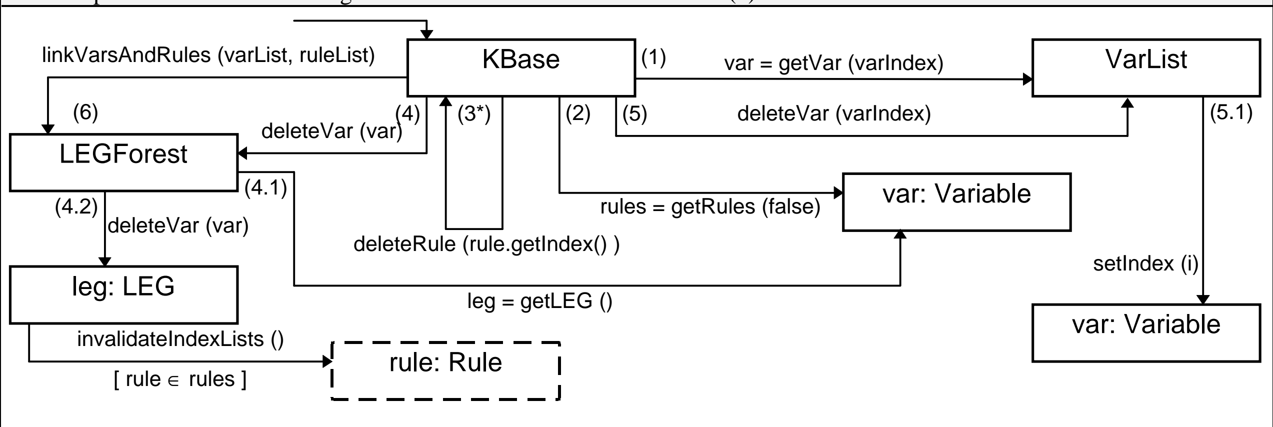
KBase.addVar (name: String, type: VarType, initValues: Object [])

Erzeuge eine neue Variable vom angegebenen Typen type (nicht dargestellt: eine Unterklasse von Variable) (1).
Zu jeder neuen Variable erzeuge Variablenwerte (Values) (nicht dargestellt: Unterklassen von Value).
Nimm diese neue Variable in die VarList auf und weise der Variable ihren Index zu (2).
Erzeuge eine neue LEG, die nur die neue Variable enthält (3).
Verknüpfe die Variable mit dieser LEG (4).
Füge die neue LEG als neuen Baum in den LEGForest ein (5).



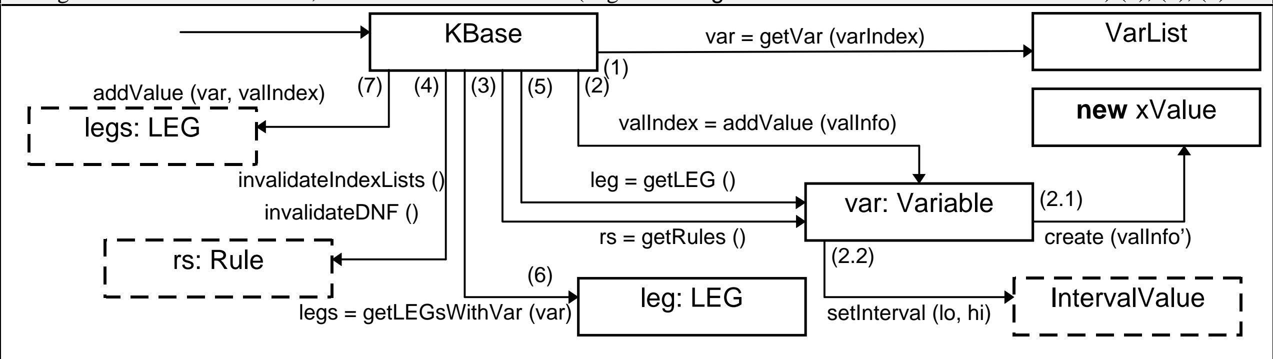
KBase.deleteVar (varIndex: int)

Ermittle die Liste der Regeln rs in der die Variable mit dem Index varIndex vorkommt (2).
Lösche alle Regeln der Liste rs (einzelner Aufruf für jede Regel aus rs) (3).
Verkleinere die LEGs, in denen die Variable vorkommt und lösche die dadurch evtl. leer gewordenen LEGs (4).
Lösche die Variable mit dem angegebenen Index aus der VarList und passe dabei die übrigen Indizes an (5).
Verknüpfe alle Variablen und Regeln mit den nunmehr minimalen LEGs (6).



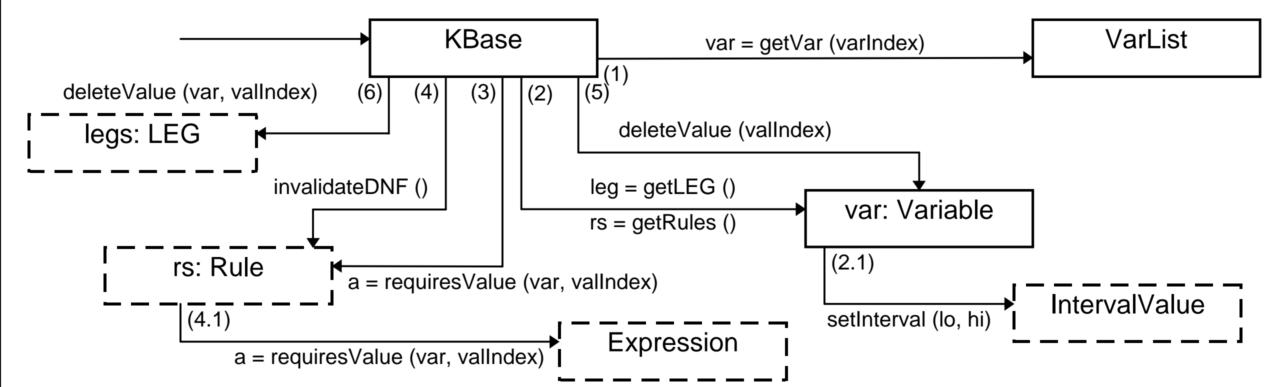
KBase.addValue (varIndex: int, valInfo: Object)

Erzeuge einen neuen Wert für die Variable mit dem Index varIndex und passe die anderen (Intervall-) Werte evtl. an (2).
Für aller Regeln, die die Variable enthalten (3): Mache die Index-Listen und ggfs. auch die DNF-Darstellung(en) ungültig (4).
Vergrößere die LEG-Tabellen, die die Variable enthalten (beginne in leg und betrachte sukzessive die Nachbarn) (5), (6), (7).



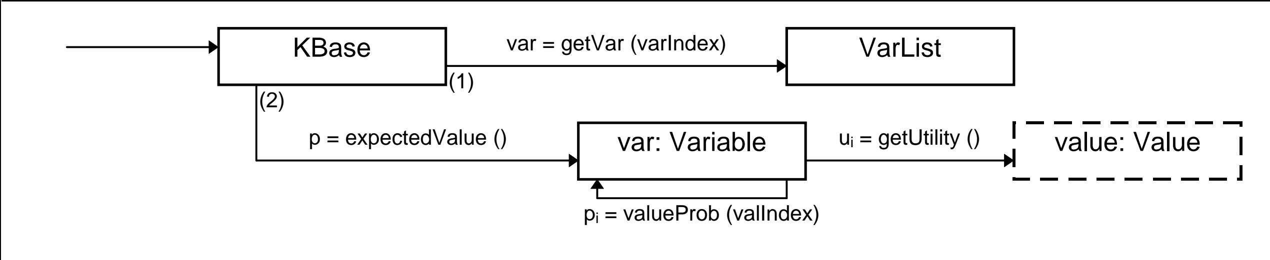
KBase.deleteValue (varIndex: int, valIndex: int)

Hole die Menge der Regeln, die die Variable verwenden (rs) und ein LEG, in der die Variable ist (als Einstiegspunkt) (2).
Lösche alle Regeln ⊆ rs, die auf diesen Wert zugreifen müssen (mit deleteRule) (3).
Mache die Index-Listen und ggfs. die DNFs der übrigen Regeln ungültig, die die Variable enthalten (4).
Lösche den valIndex'ten Wert der Variable mit dem Index varIndex und passe evtl. Intervall-Grenzen an (5).
Verkleinere die LEG-Tabellen, die die Variable enthalten (hier nicht dargestellt: leg.getLEGsWithVar (var)) (2), (6).



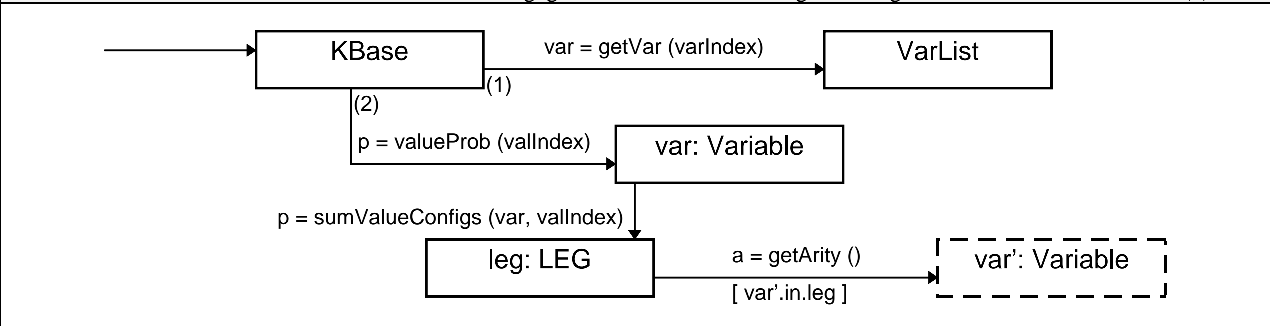
KBase.expectedValue (varIndex: int): double

Ermittle nacheinander alle aktuellen Wahrscheinlichkeiten (mittels valueProb, der letzte Wert ist 1-Summe der anderen)
Multipliziere die Werte jeweils mit dem *Utility* und summiere diese gewichteten Wahrscheinlichkeiten.



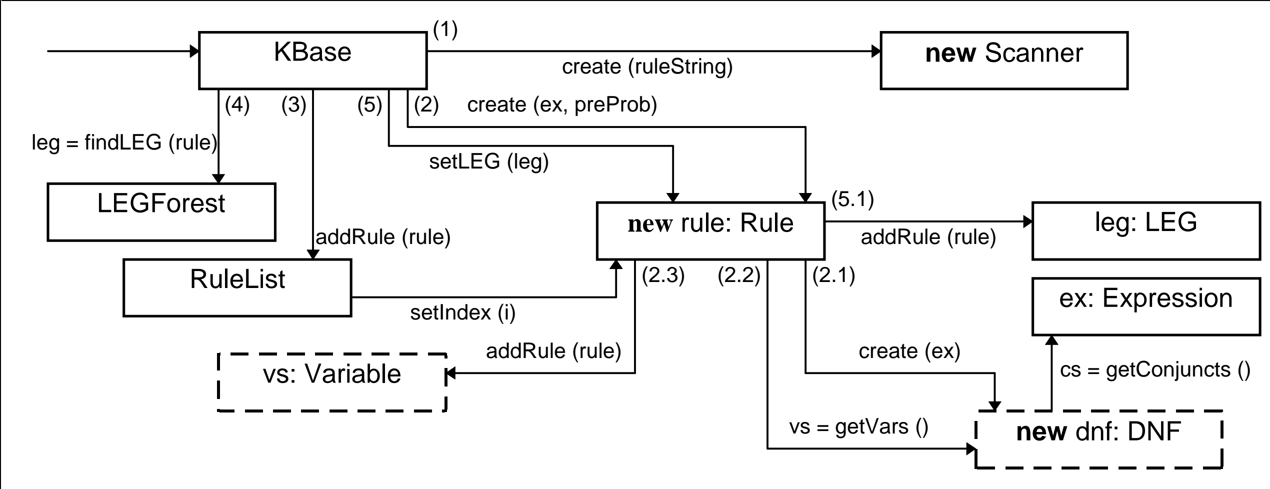
KBase.valueProb (varIndex: int, valIndex: int): double

Bilde in der minimalen LEG die Randsumme des angegebenen Variablenwertes gemäß Algorithmus VarValueActProb (2).



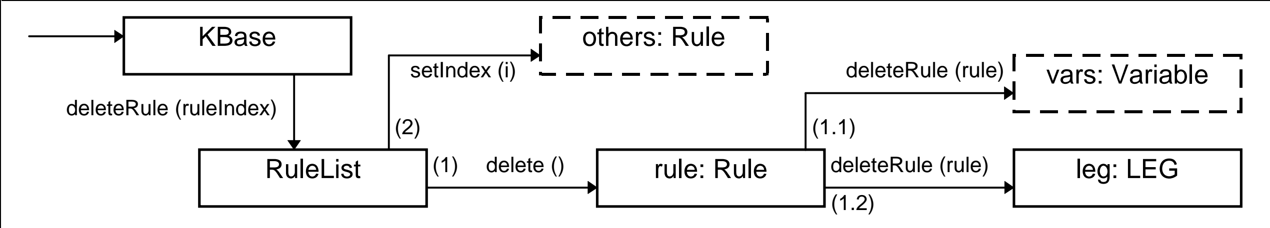
KBase.addRule (ruleString: String, preProb: PreProb)

Erzeuge einen Scanner, der aus dem angegebenen String eine Folge von Tokens bildet (1).
Forme dann einen Ausdrucksbaum aus den vom Scanner gelesenen Tokens (mit KBase.parseRule ()).
Erzeuge aus diesen Expressions und mit der angegebenen Wahrscheinlichkeit eine neue Regel (2).
Bilde aus den Ausdrucksbäumen eine DNF (2.1) (diese holt sich eine Liste der Konjunktionen aus dem Ausdrucksbaum).
Finde die Variable, die in der Regel genannt sind und trage die Regel in deren inLists ein (2.2), (2.3).
Trage die neue Regel in die RuleList ein und weise ihr ihren Index zu (3).
Suche die minimale passende LEG und (falls diese besteht) weise sie der Regel zu (4), (5).



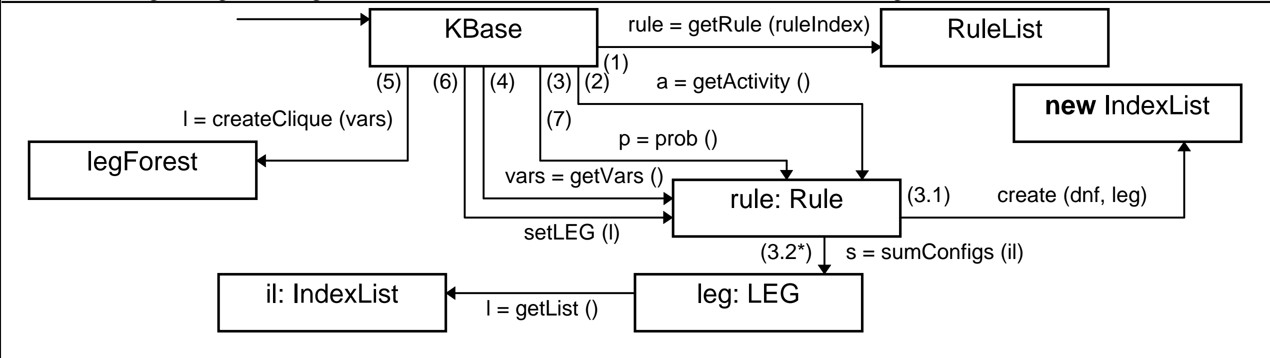
KBase.deleteRule (ruleIndex: int)

Streiche die Regel aus den inRules-Listen der Variablen, die sie enthält (1.1).
Lösche die Regel aus der Liste der LEG, die sie enthält (1.2).
Lösche die Regel aus der RuleList und passe die Indizes der übrigen Variablen an (2).



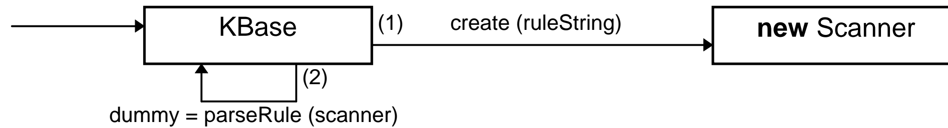
KBase.ruleProb (ruleIndex: int): double

if Regel.activity = passive **then** (2)
Erzeuge temporäre LEG zur Berechnung des Ergebnisses gemäß Algorithmus ruleActProb (...) (4), (5), (6), (7).
else Erzeuge (wenn nötig) die Index-Listen für den Zugriff auf die minimale LEG (3.1).
Errechne Ergebnis gemäß Algorithmus RuleActProb anhand der Index-Listen für die Regel (3.2).



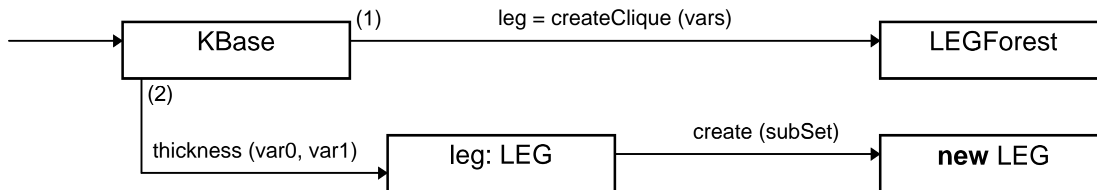
KBase.ruleStringCheck (ruleString: String)

Lege einen Scanner an, der den ruleString in Tokens zerlegen kann (1).
Versuche aus den Tokens einen Ausdrucksbaum aufzubauen (im Fehlerfall wird beschreibende Exception geliefert) (2).



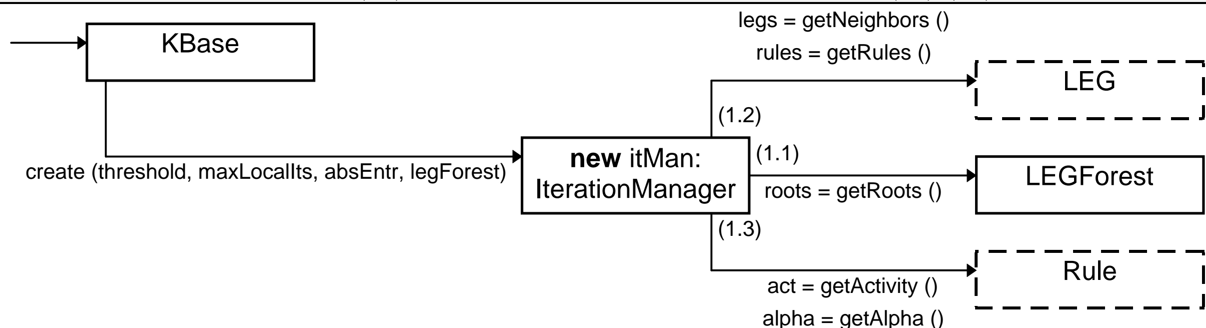
KBase.graphEdgeThickness (varIndex0: int, varIndex1: int): double

if var1 und var2 sind unverbunden **then return** 0 **else**
Erzeuge temporäre LEG für die Berechnung gemäß Algorithmus VarGraphEdgeThickness (1).
Wende auf diese LEG die Berechnungsformel an (2).
Erzeuge dazu drei temporäre LEGs mit Teilmengen.
Lege interne Kopie der Verteilung an, fülle sie neu gemäß Formel und berechne die relative Entropie.



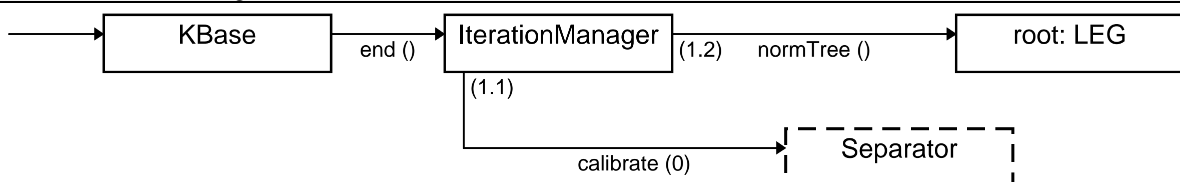
KBase.beginIteration (threshold: double, maxLocalIts: int)

Erzeuge einen neuen IterationManager (mit der zuvor ermittelten absoluten Entropie (KBase.absEntropy ()) als Startwert).
Hole die Wurzeln der LEG-Bäume (1.1) und bereite den Durchlauf des ersten Baumes vor (1.2), (1.3).



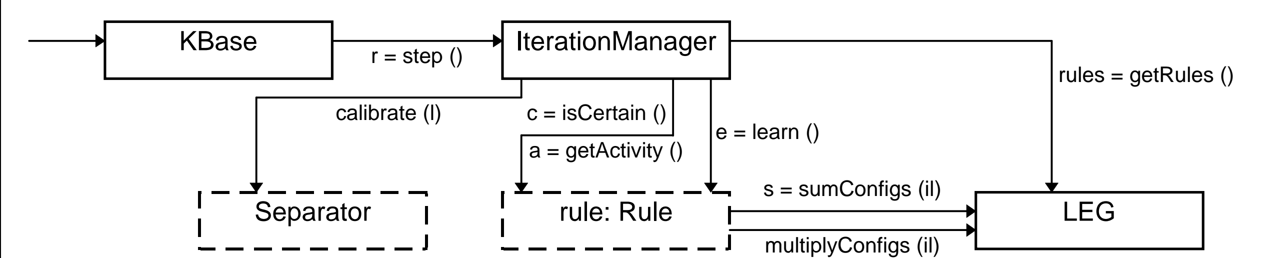
KBase.endIteration ()

if nicht am Anfang einer globalen Iteration **then**
Brich die aktuelle lokale Iteration ab und kalibriere den restlichen Baum, bis die Wurzel erreicht wird, (1.1) ...
... dann normiere den aktuellen LEG-Baum (1.2)
Lösche den IterationManager.



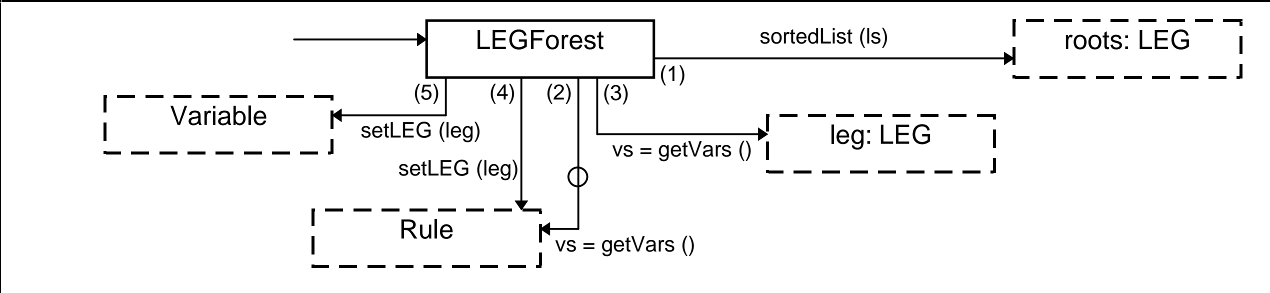
KBase.stepIteration (): boolean

Lerne die aktuelle Regel (falls noch eine existiert: sichere Regeln müssen nur einmal gelernt werden) in ihre LEG.
Falls damit eine lokale Iteration beendet ist, prüfe Entropieänderungen und wechsle zur nächsten Regel (evtl. andere LEG).
Falls damit eine globale Iteration beendet ist, prüfe Erfüllungskriterium (r) und gehe zur ersten Regel.
Falls ein Widerspruch festgestellt wurde, setze die gesamte Struktur auf Gleichverteilung zurück (LEGForest.reset ()).



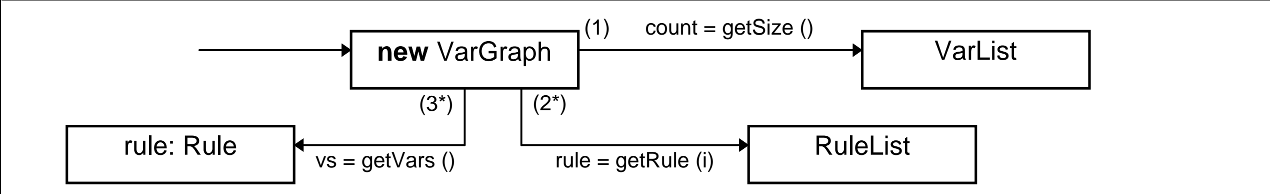
LEGForest.linkVarsAndRules (varList: VarList, ruleList: RuleList)

// Verknüpft jede Variable und jede Regel mit der jeweils minimalen passenden LEG.
Sortiere die LEGs nach ihrer Größe (1).
Suche in dieser Liste für jede Regel die erste passende LEG und weise diese der Regel zu (2), (3), (4).
Durchlaufe alle LEGs in der sortierten Liste und trage für jede neue Variable die LEG ein (3), (5).



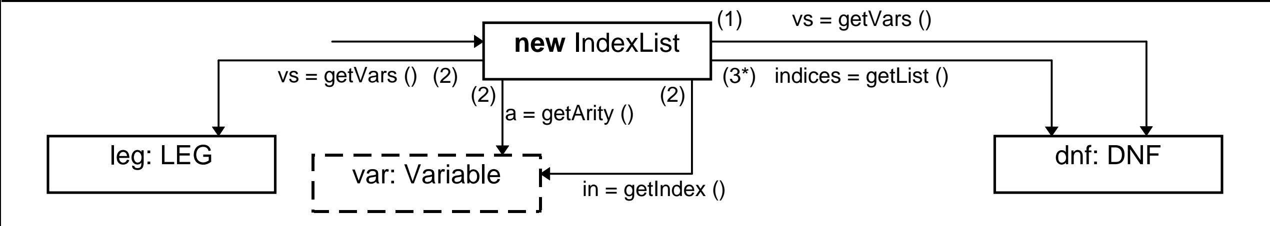
VarGraph.create (varList: VarList, ruleList: RuleList)

// Erzeugt einen neuen VarGraph (in Form von Adjazenzlisten), der die Nachbarschaftsrelation zwischen Variablen zeigt.
Starte mit einer leeren Liste pro Variable (1).
Durchlaufe alle Regeln und trage die vorkommenden Variablen jeweils in die Adjazenzlisten der anderen ein (2), (3).



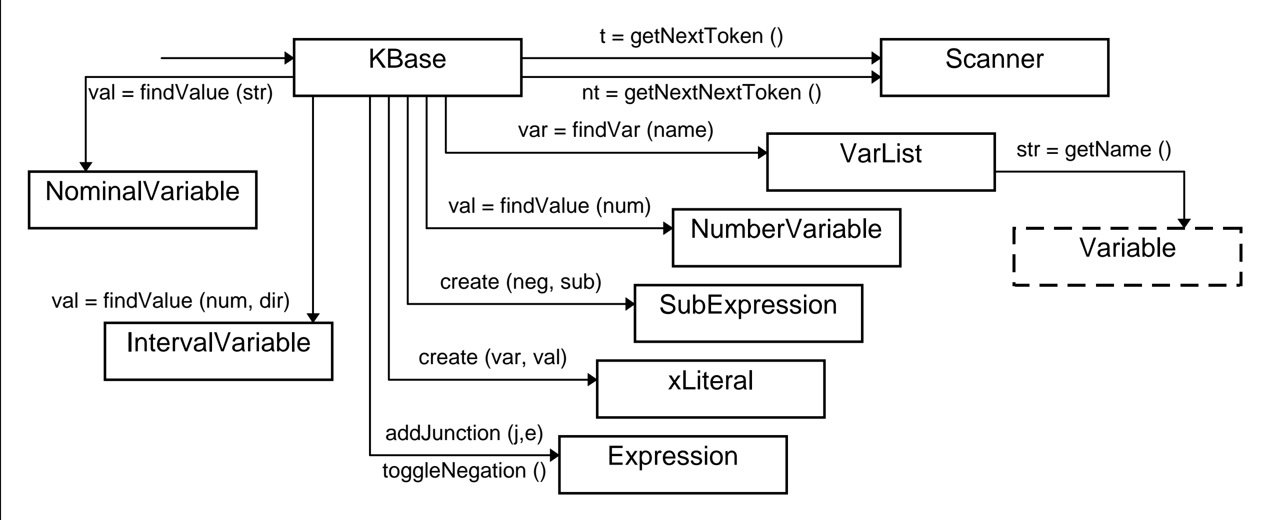
IndexList.create (dnf: DNF, leg: LEG)

// Bildet eine gegebene DNF-Darstellung auf eine Index-Liste der gegebenen LEG ab.
(Vorgehensweise hier leider zu umständlich zu beschreiben, Details im Code).



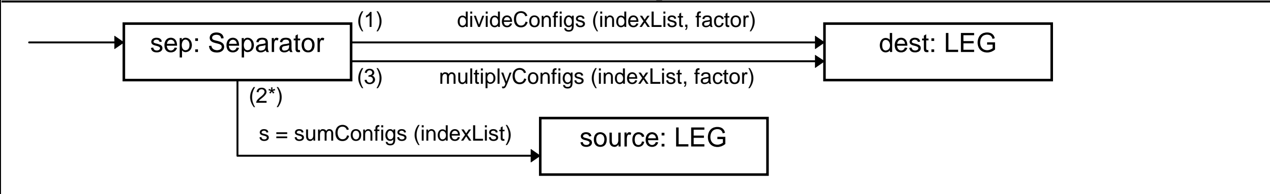
KBase.parseRule (scanner: Scanner): Expression

// Wandelt die Tokens aus dem Parser in einen Ausdrucksbaum gemäß SPIRIT-Regelsyntax um.
Lese Tokens aus dem Parser, suche nach Variablen und Werten und erzeuge daraus SubExpressions und Literals.



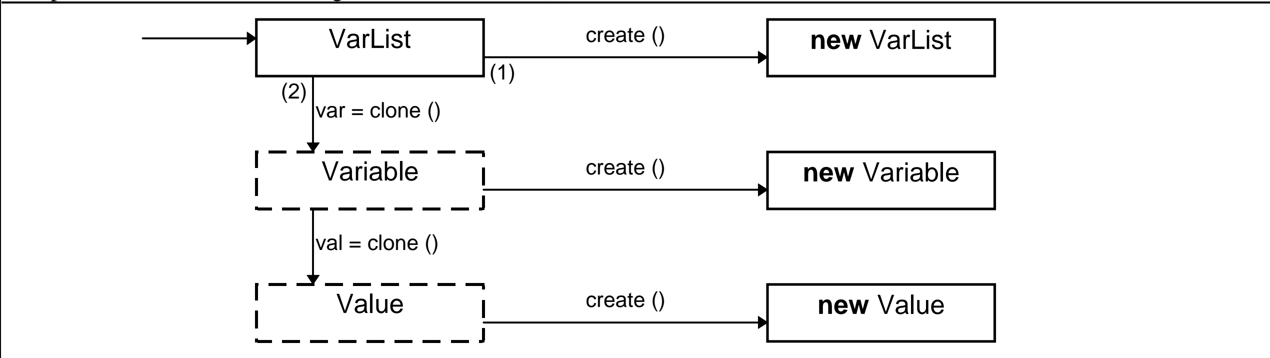
Separator.calibrate (destIndex: int ∈ { 0, 1 })

// Kalibriert eine LEG über den Separator mit einer benachbarten LEG gemäß Algorithmus LEGCalibrate.
Teile die Ziel-LEG mit den Wahrscheinlichkeiten aus dem Separator (1).
Fülle den Separator mit der Randverteilung aus der Quell-LEG (2).
Dividiere die Ziel-LEG mit den Wahrscheinlichkeiten aus dem Separator (3).



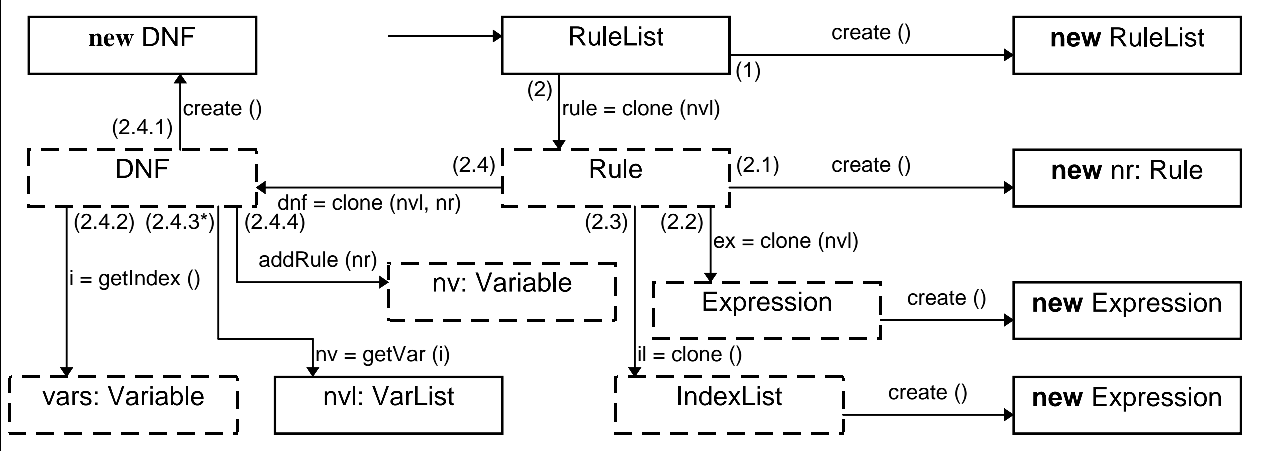
VarList.clone (): VarList

// Erzeugt eine neue, identische VarList, in der allerdings noch keine Verbindungen mit LEGs und Regeln eingerichtet sind.
Erzeuge neue (leere) VarList (1).
Dupliziere alle Variablen und füge sie in die Liste ein (2).



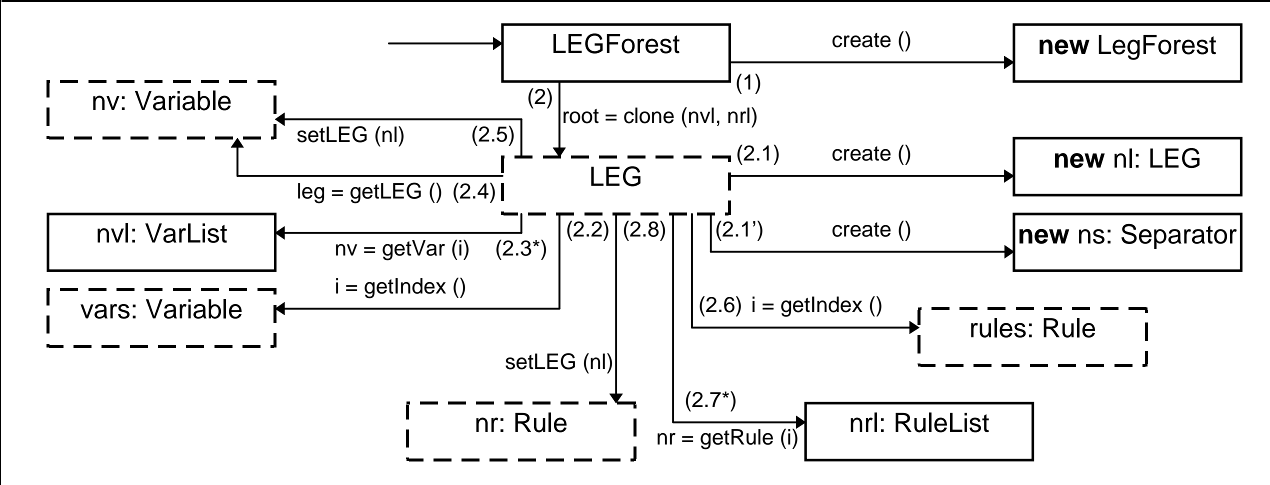
RuleList.clone (nvl: VarList): RuleList

// Erzeugt eine neue, identische RuleList, in der allerdings noch keine Verbindungen mit LEGs eingerichtet sind.
Erzeuge neue (leere) RuleList (1).
Dupliziere alle Regeln und füge sie in die Liste ein (2).
Dabei: Dupliziere Ausdrucksbäume (2.2), IndexListen (2.3) und DNFs (2.4).
In den DNFs: Tausche Verweise auf die "alten" Variablen mit Verweisen auf die entsprechenden der neuen VarList nvl...
... und sorge dafür, daß die neue Regel in den inRules-Listen der "richtigen" Variablen auftaucht (2.4.4).



LEGForest.clone (nvl: VarList, nrl: RuleList): LEGForest

// Erzeugt einen neue, identischen LEGForest mit jeweils neuen Verbindungen auf die Variablen- und Regelmengen
Erzeuge neuen (leeren) LEGForest (1).
Dupliziere jeden LEG-Baum und füge deren Wurzeln in die Wurzel-Liste ein (2).
Für jeden LEG-Baum: Rekursiver Baumabstieg. Dabei erzeuge und verbinde LEGs oder Separatoren (2.1), (2.2), ...
... und biege die Verweise auf die alten Variablen auf die neue Variablen-Liste nvl um (2.2), (2.3)...
... ebenso wie die Verweise auf die Regeln, wobei die neue Regel-Liste nrl durchaus leer sein kann (2.6), (2.7).
Für Variablen in der LEG: Falls die Variable die LEG als inLEG hat (2.4), dann setze diesen Verweis in der entsprechenden Variable in der neuen Liste nvl auf die neue LEG (2.5).
Für Regeln in der LEG: Falls die Regel die LEG als assignedLEG hat, dann setze diesen Verweis in der entsprechenden Regel in der neuen Regel-Liste nrl (falls diese nicht leer ist) auf die neue LEG (2.8).
Hier (aus Platzgründen) nicht abgebildet: Falls Separator erzeugt, dann clone auch die IndexListen.

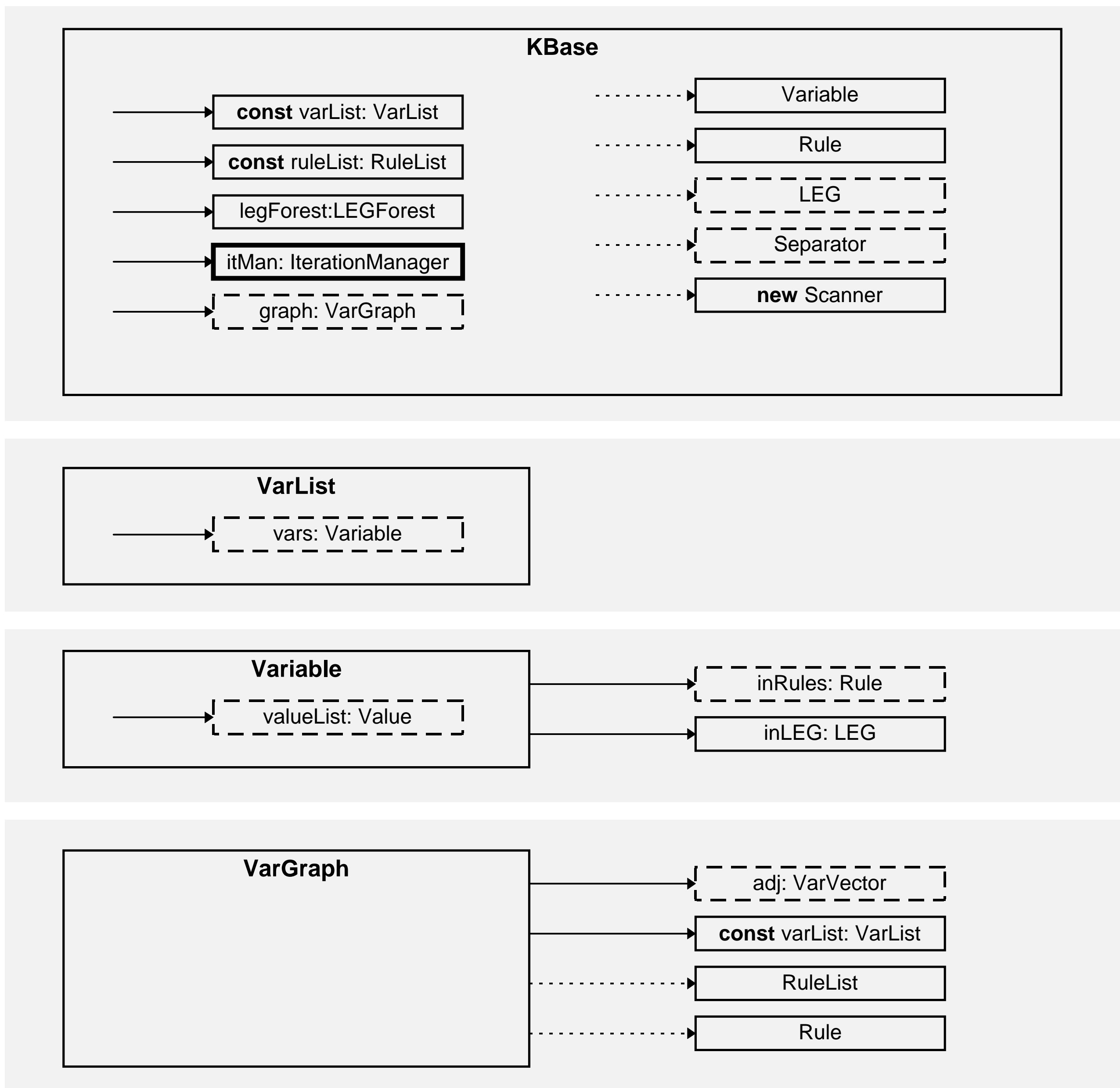


3.3 Visibility Graphs

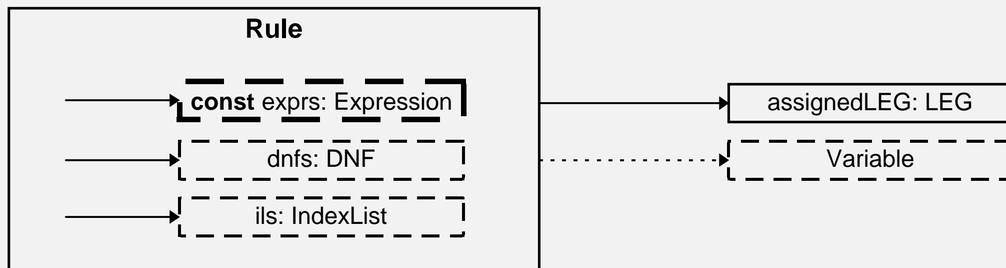
Der in FUSION folgende Schritt des Entwurfs ist die Entwicklung von *Visibility Graphs* (VGs). Zweck dieser Graphen ist die Modellierung der Zugriffspfade zwischen Instanzen der an den System-Operationen beteiligten Klassen. Insbesondere wird hierbei ermittelt, wie die *Relationships* aus dem Analyse-Modell auf die einzelnen Objekte abgebildet werden.

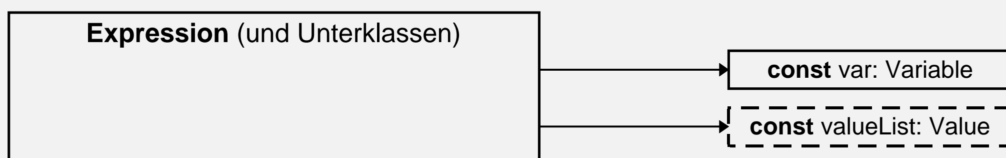
Die folgenden Graphen konnten recht schnell direkt aus den *Object Interaction Graphs* aus dem vorangegangenen Schritt gewonnen werden. Ich ging jede dort dargestellte Methode durch, prüfte die vorkommenden Kanten und erweiterte so sukzessive die Sichtbarkeitsgraphen.

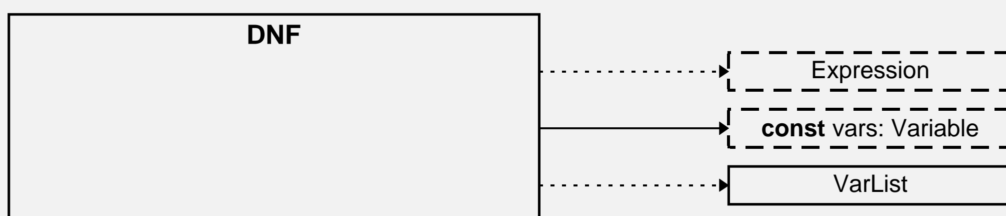
Aus drucktechnischen Gründen sind *Server*-Objekte (Vgl. [Col94, S.80]), die exklusiv (*exclusively*) einem *Client* zugeordnet sind, entgegen dem Standard von FUSION nicht mit einem doppelten, sondern mit einem fetten Rahmen dargestellt (z.B. bei KBase.itMan). Das Schlüsselwort *constant* wird mit *const* abgekürzt.

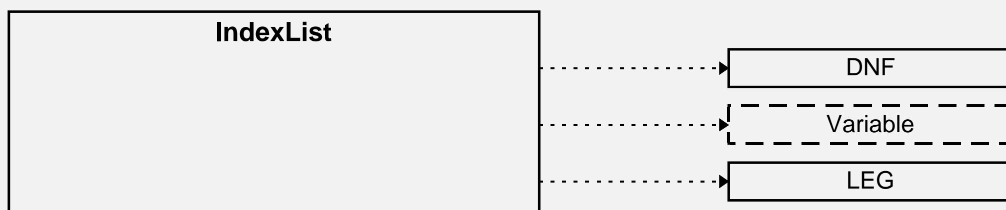


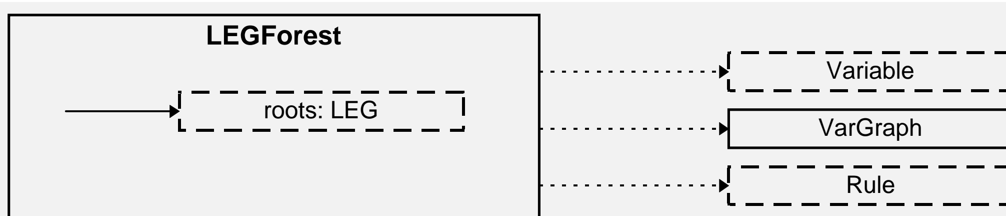
RuleList
rules: Rule

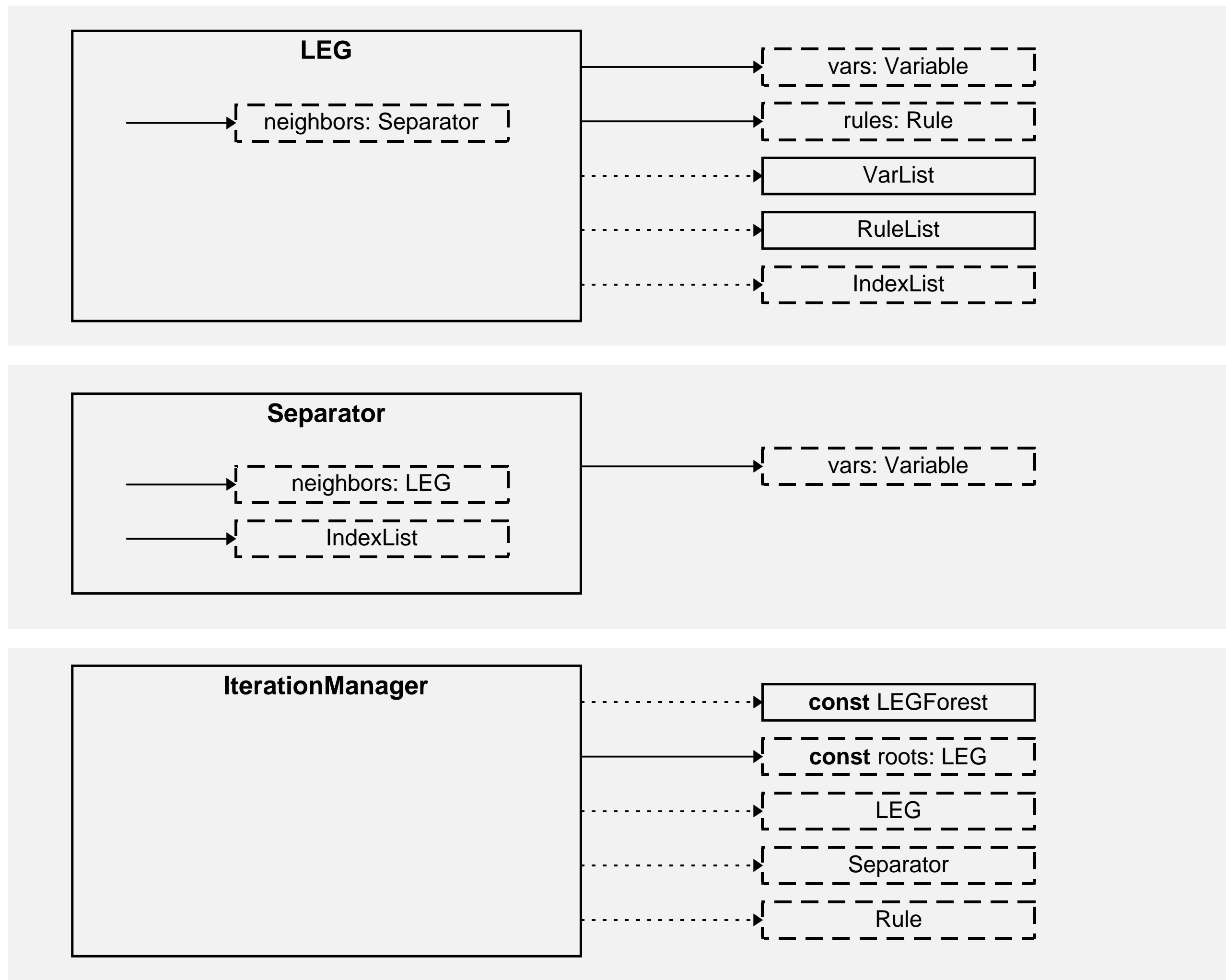

Rule
const exprs: Expression
dnfs: DNF
ils: IndexList
assignedLEG: LEG
Variable


Expression (und Unterklassen)
const var: Variable
const valueList: Value


DNF
Expression
const vars: Variable
VarList


IndexList
DNF
Variable
LEG


LEGForest
roots: LEG
Variable
VarGraph
Rule



Was sind nun aber die Ergebnisse dieser Graphen? Im Projekt SPIRIT hatte ich zahlreiche wichtige Entwurfsentscheidungen bereits in frühen Phasen (insbes. Abschnitt 3.1) getroffen. Daher ergaben sich die *Object Interaction Graphs* ebenso wie die *Visibility Graphs* relativ direkt. Die Sichtbarkeitsgraphen enthalten hier sehr wenig neue Informationen. Zumindest konnten sie später bei der Erstellung und Überprüfung der *Inheritance Graphs* und der *Class Descriptions* als weiteres Vergleichsmodell zur Korrektur herangezogen werden.

Zu einem etwas interessanteren Ergebnis kommt man, wenn man alle *Visibility-Graphs* zu einem neuen Graphen miteinander verbindet: Die Knoten dieses Graphen sind die vorkommenden Klassen und die Pfeile werden aus der Vereinigungsmenge der einzelnen Sichtbarkeitsgraphen genommen. In der so entstehenden Übersicht über die Klassen läßt sich erkennen, welche Klassen (=Module im späteren Programm) auf welche anderen Klassen zugreifen. Es ergibt sich (fast) eine Halbordnung auf den Klassen. Geht man bei der Implementierung *bottom-up-* oder *top-down-gemäß* vor (vgl. [Som96, S. 455f]), so muß man sich lediglich nach dieser Halbordnung richten, um eine schrittweise Umsetzung und Überprüfung der Module zu ermöglichen.

3.4 Inheritance Graphs

Dieser Abschnitt stellt die Vererbungshierarchien der entworfenen Klassen dar. Diese Hierarchien wurden gemäß FUSION parallel zu den in Abschnitt 3.5 erstellten *Class Descriptions* entwickelt, sind aber dennoch eine wichtige Grundlage für das Verständnis derselben und werden daher im folgenden als erstes vorgestellt. Zu den einzelnen Diagrammen werden kurze Erläuterungen gegeben, die den Entwurf noch einmal zusammenfassen.

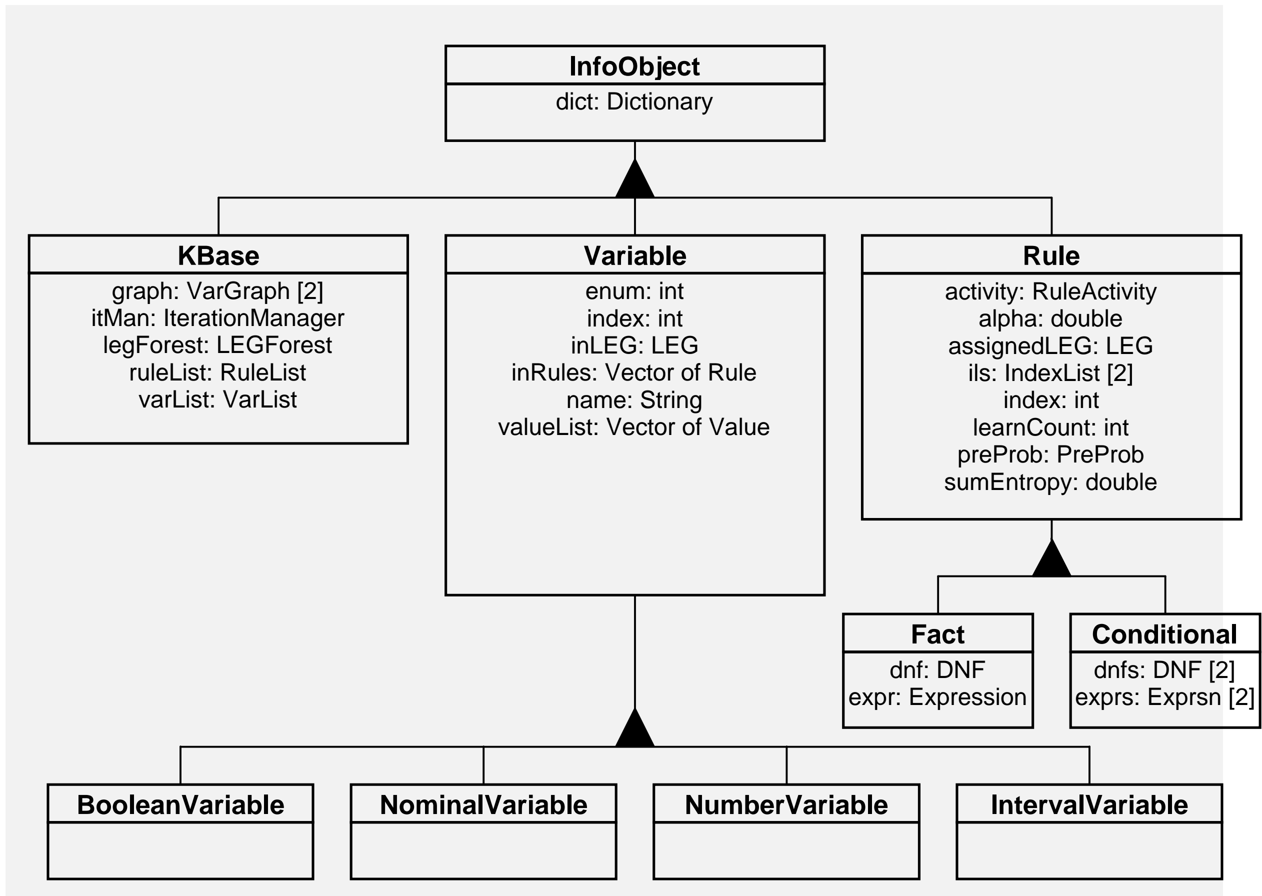


Abbildung 16: Klassenhierarchie von SPIRIT (Teil 1)

Die neu eingeführte abstrakte Klasse InfoObject stellt ein *Dictionary* bereit, in das *User*-Informationen (sh. Abschnitt 1.2.2 unten) eingetragen werden können. Diese Funktionalität wird von den Klassen KBase, Variable und Rule genutzt. Eine KBase ist eine Wissenbasis und besteht aus Variablen- und Regelmenge, der LEG-Struktur und eventuell einem IterationManager. Eine Variable muß entweder Boolean-, Nominal-, Number- oder IntervalVariable sein. Jede Variable verfügt über eine Liste von Verweisen auf die Regeln, in der sie vorkommt, und über einen Verweis auf die minimale LEG, die sie enthält. Vor allem aber besitzt sie eine Liste von Werten. Außerdem hat sie einen Namen, einen Index und einen Eliminationsindex. Regeln sind entweder Fakten oder Konditionale (mit Prämisse). Jede Regel verfügt i. w. über eine vorgegebene Wahrscheinlichkeit, einen Verweis auf die minimale passende LEG, einen Aktivitätszustand, einen Index und zum Lernen ein aktuelles Alpha. Facts bestehen aus einem Ausdrucksbaum und einer daraus abgeleiteten DNF-Darstellung, während Conditionals jeweils für jede Regelseite eine DNF und einen Ausdrucksbaum besitzen. Beide Klassen verfügen über verschiedene Index-Listen zur schnellen Durchführung von Berechnungen auf LEGs.

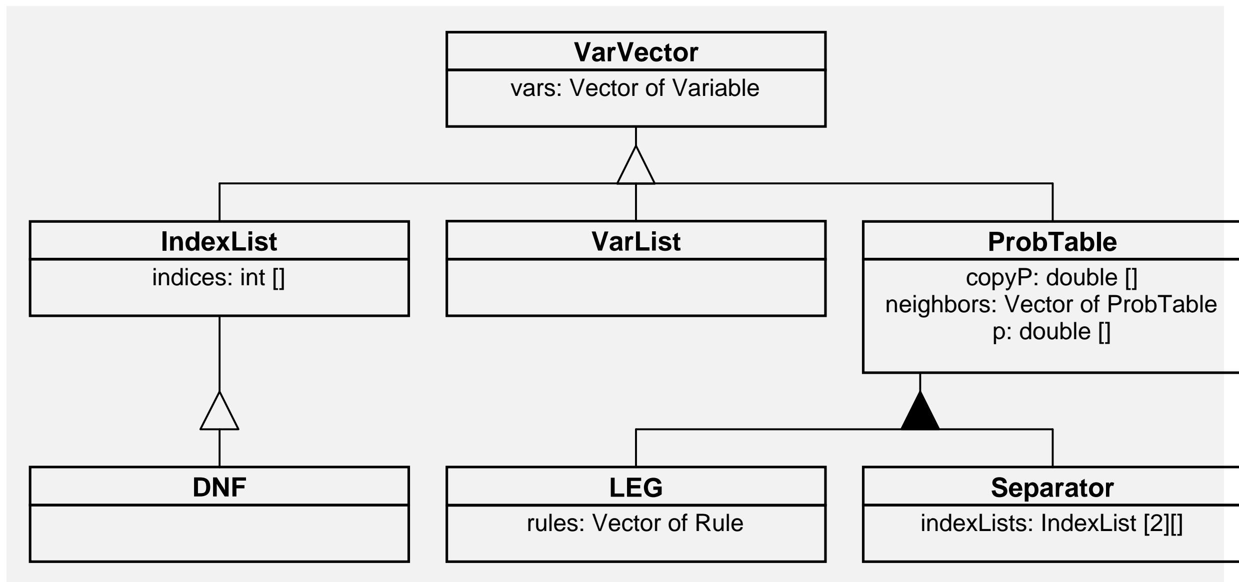


Abbildung 17: Klassenhierarchie von SPIRIT (Teil 2)

Ein VarVector ist eine dynamisch veränderbare, sortierte Menge von (Verweisen auf) Variablen. Indexlisten beziehen sich immer auf eine Variablenliste. Sie enthalten eine Liste von Indizes über eine (imaginäre) LEG aus diesen Variablen. Die Klasse DNF stellt eine über Indizes gespeicherte Regel (-seite) in disjunktiver Normalform dar und muß daher die enthaltenen Variablen in einer eindeutigen Reihenfolge benennen. Eine ProbTable ist eine Variablenmenge plus einer Tabelle von Wahrscheinlichkeiten. Diese Tabelle kann intern in einer Kopie zwischengespeichert werden. Außerdem sind ProbTables über eine Liste von Nachbarn untereinander verbunden. Die Klassen LEG und Separator sind spezielle ProbTables. Eine LEG kann zum Lernen verwendet werden, während Separatoren nur zur Performance-Steigerung eingesetzt werden. LEGs sind nur mit Separatoren benachbart und Separatoren nur mit LEGs. Zu jeder dieser LEGs besitzt ein Separator Index-Listen zur schnellen Ermittlung der Randwahrscheinlichkeiten. VarList schließlich ist eine Containerklasse für die Variablen. Im Gegensatz zu den übrigen VarVectors wird sie als "Besitzer" der Variablen angesehen, während die übrigen Unterklassen lediglich Verweise auf Variablen enthalten. Die Klasse VarVector wurde neu eingeführt, um Methoden zu ermöglichen, die auf DNFs, ProbTables und VarLists gleichermaßen arbeiten (z.B. Test ob eine DNF "Teilmenge" einer ProbTable ist). Insbesondere aber sollen alle Variablenlisten stets sortiert sein, damit Vergleiche, Einfügen und Löschen schnell möglich sind.

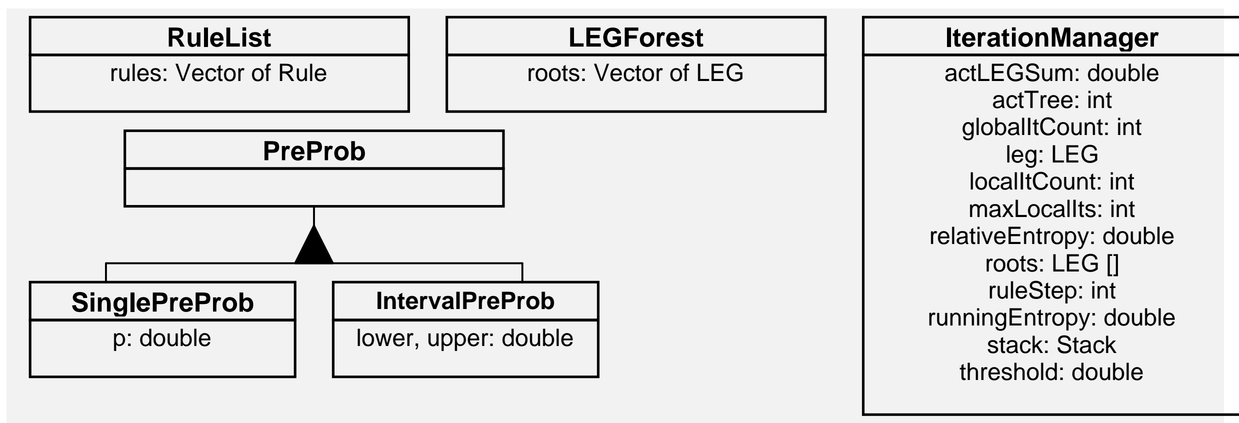


Abbildung 18: Klassenhierarchie von SPIRIT (Teil 3)

Eine RuleList ist eine dynamische Containerstruktur für Regeln und der LEGForest einer Wissensbasis verwaltet die dynamisch veränderbare Menge der LEG-Bäume. Die vorgegebene bedingte Wahrscheinlichkeit einer Regel kann entweder einfach oder ein Intervall sein. Zur Abstraktion wird die Klasse PreProb daher um zwei Unterklassen erweitert. Der IterationManager, der den Ablauf einer Iteration steuert verfügt über zahlreiche Attribute, die den aktuellen Zustand der Iteration angeben.

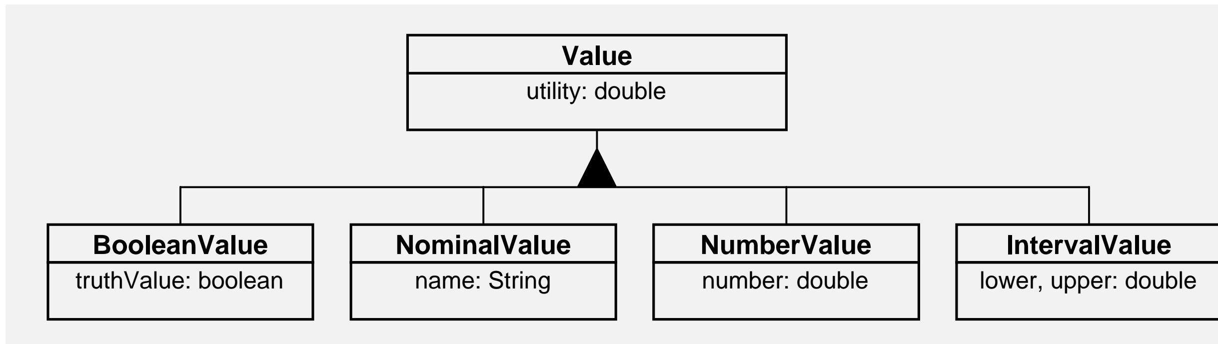


Abbildung 19: Klassenhierarchie von SPIRIT (Teil 4)

Variablenwerte werden von der abstrakten Klasse Value abgeleitet. Abhängig vom Typ der Variable, die den Wert besitzt, enthält jeder Wert anderen Inhalt. *Boolean*-Werte können wahr oder falsch sein, *Nominals* werden mit einem Namen bezeichnet, *Numbers* mit einer Zahl und *Intervals* mit einem Intervall aus reellen Zahlen. Alle Variablenwerte verfügen zur Berechnung des Erwartungswertes über einen Nutzen (*Utility*), der allerdings nur bei booleschen und nominalen Variablen veränderbar ist.

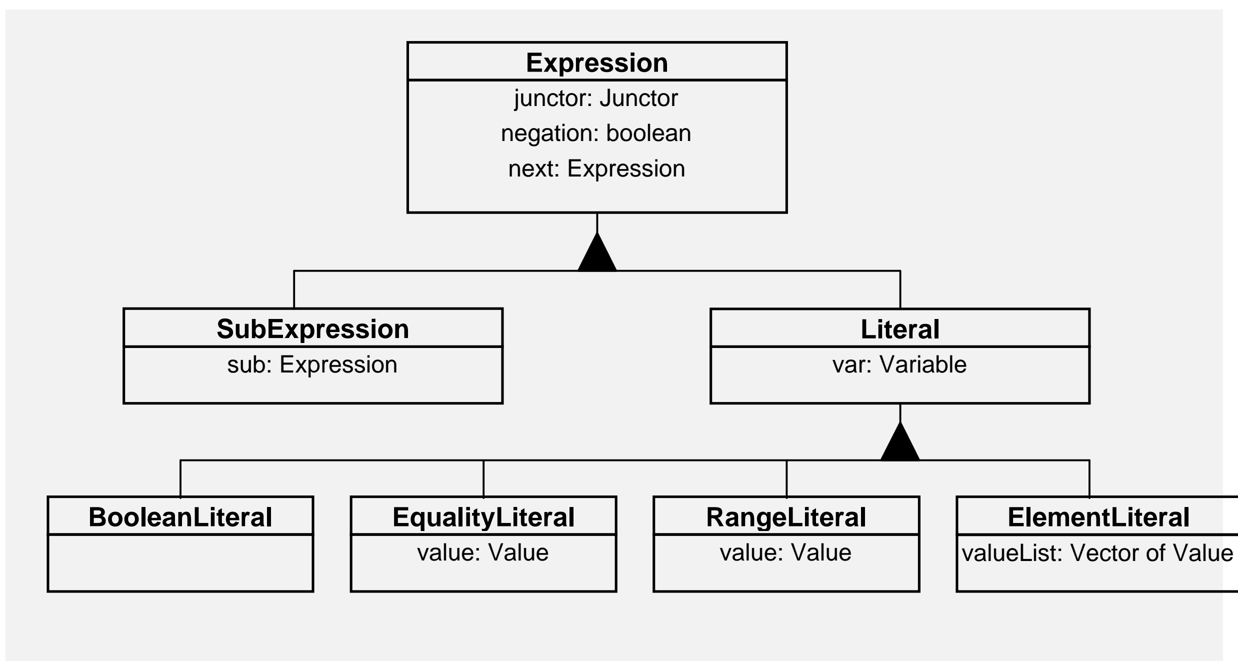


Abbildung 20: Klassenhierarchie von SPIRIT (Teil 5)

Die Unterklassen der Klasse Expression dienen zur Darstellung der Ausdrucksbäume zu Regeln. Expressions können (geklammerte) Unterausdrücke oder Literale sein. Literale setzen eine Variable mit ihren Werten in Beziehung. Entsprechend den zulässigen Operatoren in Regeln, gibt es verschiedene Literalklassen.

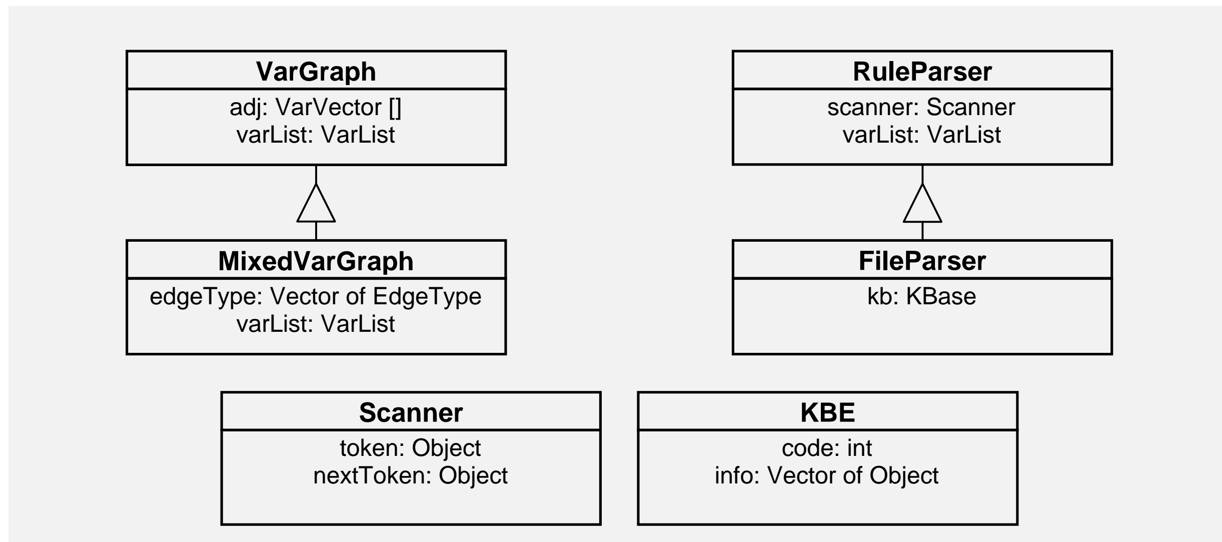


Abbildung 21: Klassenhierarchie von SPIRIT (Teil 6)

Die Klasse VarGraph speichert einen Graphen über der Variablenmenge. Zu jeder Variable wird eine Adjazenzliste gespeichert. Die Unterklasse MixedVarGraph, die gemischte Graphen darstellt, verfügt zusätzlich über eine Angabe der Art der jeweiligen Kanten. Ein Scanner ist ein Objekt, das Tokens aus einem Eingabestrom liefert und dabei einen Schritt voraussehen kann. An dieser Stelle werden zum Zwecke der Modularisierung zwei neue Klassen eingeführt, die den Übersetzungsvorgang von Ausdrucksbäumen und Dateien erledigen. Ein RuleParser erzeugt aus einer vom Scanner gelieferten Tokenfolge einen Ausdrucksbaum und ein FileParser liest Variablen und Regeln aus einer SPIRIT-Datei. Beide Klassen werden nur temporär benötigt und sollen verhindern, daß sich das Modul KBase unnötig aufbläht (da es sich um *Recursive-Descent-Parser* handelt, sind zahlreiche Hilfsfunktionen erforderlich). Die Klasse KBE schließlich dient zur Meldung von Fehlern. Tritt eine Ausnahmesituation auf (z.B. durch einen ungültigen Funktionsaufruf), so wird ein KBE-Objekt erzeugt, mit einem Fehlercode und evtl. zusätzlichen Informationen ausgestattet und an die aufrufende Umgebung geschickt (sh. Anhang C).

Die Erstellung dieser Graphen war sehr leicht mit den *Object Interaction Graphs* und den schrittweise vervollständigten *Class Descriptions* möglich. Zudem wurden die meisten Entscheidungen zur Vererbungsstruktur bereits in den frühesten Entwicklungsschritten (insbesondere in Abschnitt 3.1) getroffen. Zwar wurden einige neue Klassen (VarVector, RuleParser, FileParser, die Unterklassen von PreProb und von Rule) eingeführt, die das Projekt ordentlicher strukturieren, der Gehalt der *Inheritance Graphs* an neuen Informationen ist dennoch gering. Dafür stellen sie eine übersichtliche Zusammenfassung der Klassenstruktur dar, die insbes. dazu geeignet ist, andere Software-Entwickler in das Projekt einzuführen.

3.5 Class Descriptions

Der letzte Schritt eines Entwicklungsprozesses nach FUSION ist die Erstellung von Klassenbeschreibungen. Diese fassen die Ergebnisse der vorangegangenen Modelle in einer präzisen und codierten Form zusammen. Somit stellen sie den Ausgangspunkt der Implementierung dar.

Die folgenden *Class Descriptions* entstanden parallel zur gesamten Entwurfsphase. Ich begann damit bereits, als ich die *Object Interaction Graphs* entwarf, weil ich dadurch eine gute Übersicht über die vorhandenen Attribute und Klassen hatte (dafür hatte ich das *Data-Dictionary* nicht aktualisiert).

Ich habe mir erlaubt, die von FUSION vorgeschlagene Syntax für *Class Descriptions* (vgl. [Col94, S.288]) in ihrer Aussagekraft etwas zu erweitern. Hierzu führte ich folgende Konventionen ein:

- *Sichtbarkeit*: Ich unterscheide Sichtbarkeit auf drei Ebenen:
  - Mit public markierte Klassen, Attribute und Methoden sind von überall (insbes. von der Shell) sichtbar.
  - Alle Klassen, Attribute und Methoden ohne besondere Kennzeichnung sind nur innerhalb der Klassenbibliothek SPIRIT sichtbar.
  - Mit private versehene Methoden sind nur innerhalb ihrer Klasse (und Unterklassen) zugänglich. Alle Attribute sind automatisch private und somit nur über entsprechende Zugriffsfunktionen (wie z.B. Variable.getIndex () und Variable.setIndex ()) erreichbar.
- *Konstruktoren*: Da fast alle Objekte bei ihrer Erzeugung mit bestimmten Werten initialisiert werden können, sind die Klassenkonstruktoren (create mit dem Schlüsselwort constructor) ebenfalls aufgeführt. Fehlt der Konstruktor, so gilt die Default-Methode create ().
- *Abstrakte Methoden und Klassen*: Abstrakte Klassen, von denen es keine Instanzen geben kann, sind mit **abstract** markiert. Meistens verfügen sie über mindestens eine abstrakte (leere) Methode, die von den Unterklassen überschrieben werden muß. Abstrakte Methoden sind ***kursiv*** gedruckt.

Noch eine Anmerkung: Einige Funktionen liefern Ergebnisse vom Typ Vector oder Arrays zurück. Hierdurch kann das Geheimnisprinzip verletzt werden. In den meisten Programmiersprachen bedeutet dies nämlich, daß die aufrufende Funktion einen direkten Verweis auf einen Vektor oder ein Array aus einem anderen Objekt erhält. Dadurch kann sie theoretisch auch Änderungen vornehmen, die außerhalb der Kontrolle des Objektes liegen. In diesem Projekt werden *innerhalb der Klassenbibliothek* aus Effizienzgründen dennoch häufig die originalen Arrays und Vektoren zurückgeliefert, da alternativ aufwendige Kopien angelegt werden müssten.

In den folgenden *Class Descriptions* fehlen die Klassen Dictionary, File, Object, Stack, String und Vector, die i.a. aus Standard-Bibliotheken entnommen werden können.

```
class BooleanLiteral isa Literal
    constructor create (var: Variable)          // Erzeugt neues Literal mit der angeg. Variable
    method clone (nvl: VarList): Expression     // Erzeugt Kopie von this mit Verweisen nach nvl
    method toString (): String                  // Wandelt this in einen String um (zur Ausgabe)
endclass
```

```
class BooleanValue isa Value
    attribute truthValue: constant boolean      // Wahrheitswert des Values
    constructor create (truthValue: boolean)    // Erzeugt neuen Wert (true oder false)
    method toString (): String                  // Wandelt this in einen String um (zur Ausgabe)
endclass
```

```
class BooleanVariable isa Variable
    constructor create (name: String)           // Erzeugt neue Variable mit false und true
    method clone (): Variable                   // Erzeugt identische Kopie von this
endclass
```

```
class DNF isa IndexList
    constructor create (ex: Expression)         // Erzeugt eine DNF-Darstellung des
Ausdruckbaumes
    method toString (): String                  // Erzeugt String (zur Regelausgabe in DNF)
endclass
```

```
class Conditional isa Rule
    attribute dnfs: bound DNF [2]                               // DNF-Darstellungen der beiden Regelseiten
    attribute exprs: exclusive constant bound Expression [2]    // Ausdrucksbäume der beiden Seiten
    constructor create (prem, conc: Expression, preProb: PreProb)         // Erzeugt Regel mit den 2
Ausdrucksbäumen
    method clone (nvl: VarList): Conditional                    // Erzeugt Kopie von this mit Verweisen nach nvl
    method getDNFString (): String                              // Liefert eine Darstellung der Regel in DNF
    method getVars (): VarVector                                // Liefert die vorkommenden Variablen
    method getVars (side: int): VarVector                       // Liefert die Variablen einer Regelseite (0 oder 1)
    method invalidateDNF ()                                     // Berechnet die DNF neu
    method premProb (): double                                  // Ermittelt die aktuelle
Prämissenwahrscheinlichkeit
    method prob (): double                                      // Ermittelt die aktuelle bedingte
Wahrscheinlichkeit
    method requiresValue (var: Variable, valIndex: int): boolean          // Prüft, ob der Wert benötigt wird
    method toString(): String                                   // Erzeugt einen String aus den
Ausdrucksbäumen
endclass
```

```
class ElementLiteral isa Literal
    attribute valueList: Vector of Value                        // Liste der Werte nach dem Element-Operator
    constructor create (var: Variable, value: Vector of Value)  // Erzeugt neues Literal mit den angeg. Werten
    method clone (nvl: VarList): Expression                     // Erzeugt Kopie von this mit Verweisen nach nvl
    method requiresValue (var: Variable, valIndex: int): boolean          // Prüft, ob der Wert benötigt wird
    method toString (): String                                  // Wandelt this in einen String um (zur Ausgabe)
endclass
```

```
class EqualityLiteral isa Literal
    attribute value: constant Value                             // Wert mit dem die Variable verglichen wird
    constructor create (var: Variable, value: Value)            // Erzeugt neues Literal mit dem angeg. Wert
    method clone (nvl: VarList): Expression                     // Erzeugt Kopie von this mit Verweisen nach nvl
    method requiresValue (var: Variable, valIndex: int): boolean          // Prüft, ob der We rt benötigt wird
    method toString (): String                                  // Wandelt this in einen String um (zur Ausgabe)
endclass
```

```
abstract class Expression
    attribute junctor: Junctor                                  // Verknüpfungsoperator für next (falls next
existiert)
    attribute negation: boolean                                 // Gibt an, ob Expression negiert ist
    attribute next: exclusive bound Expression                  // Nächste Expression in Kette (oder null)
    constructor create (negation: boolean)                      // Erzeugt neue Expression (Methode zur
Inheritance)
    method addJunction (j: Junctor, next: Expression)           // Hängt neue Expression an Kette
    method clone (nvl: VarList): Expression                     // Erzeugt Kopie von this mit Verweisen in nvl
    method getConjuncts (vars: VarVector): Vector of (int [])   // Liefert die Konjunktionen als valIndizes
    method requiresValue (var: Variable, valIndex: int): boolean          // Prüft, ob der Wert benötigt wird
    method toggleNegation ()                                    // Kehrt negation um
    method toString (): String                                  // Wandelt den Ausdruck in einen String um
endclass
```

```
class Fact isa Rule
    attribute dnf: bound DNF                                    // DNF-Darstellung des Fakts
    attribute expr: exclusive constant bound Expression         // Wurzel des Ausdrucksbaums
    constructor create (expr: Expression, preProb: PreProb)     // Erzeugt Fakt mit dem AusdrucksBaum
    method clone (nvl: VarList): Conditional                    // Erzeugt Kopie von this mit Verweisen nach nvl
    method getDNFString (): String                              // Liefert eine Darstellung des Fakts in DNF
    method getVars (): VarVector                                // Liefert die vorkommenden Variablen
    method invalidateDNF ()                                     // Berechnet die DNF neu
    method prob (): double                                      // Ermittelt die aktuelle bedingte
Wahrscheinlichkeit
    method requiresValue (var: Variable, valIndex: int): boolean          // Prüft, ob der Wert benötigt wird
    method toString(): String                                   // Erzeugt einen String aus dem Ausdrucksbaum
endclass
```

```
class FileParser isa RuleParser
    attribute kb: constant KBase                                  // Verweis zum Einfügen von Regeln und
Variablen
    constructor create (file: File, kb: KBase, varList: VarList)  // Erzeugt neuen FileParser
    method parse ()                                               // Übersetzt die Datei in Variablen und Regeln
endclass

class IndexList isa VarVector
    attribute indices: constant int []                            // Liste der Indizes bzgl. einer Variablenmenge
    constructor create (dnf: IndexList, pt: ProbTable)            // Erzeugt neue IndexList aus der DNF bzgl. pt
    method getList (): int []                                     // Liefert die Liste (indices) als Array
endclass

public [abstract] class InfoObject
    attribute dict: constant exclusive bound Dictionary           // Dictionary zur Zuordnung von key zu info
    public method getUserInfo (key: String): String               // Ermittelt den Eintrag mit dem Schlüssel key
    public method getUserInfoKeys (): String []                   // Liefert alle im Dictionary befindlichen keys
    public method setUserInfo (key: String, info: String)         // Ändert den Eintrag mit dem Schlüssel key
endclass

public class IntervalPreProb isa PreProb
    attribute lower: constant double                              // Untergrenze des Intervalls
    attribute upper: constant double                              // Obergrenze des Intervalls
    public constructor create (lower: double, upper: double)      // Erzeugt neues IntervalPreProb
    method clone (): IntervalPreProb                              // Erzeugt identische Kopie von this
    public method getLower (): double                             // Liefert lower
    public method getUpper (): double                             // Liefert upper
    public method isCertain (): boolean                           // Prüft, ob es eine sichere PreProb ist (hier: false)
    public method toString (): String                             // Erzeugt einen String aus dem Intervall
endclass

class IntervalValue isa Value
    attribute lower: double                                       // Untere Intervall-Grenze
    attribute upper: double                                       // Obere Intervall-Grenze
    constructor create (lower, upper: double)                     // Erzeugt neuen Intervall-Wert mit den Grenzen
    method getLower (): double                                    // Liefert lower
    method getUpper (): double                                    // Liefert upper
    method setInterval (lower, upper: double)                     // Ändert lower und upper
    method toString (): String                                    // Wandelt this in einen String um (zur Ausgabe)
endclass

class IntervalVariable isa Variable
    constructor create (name: String, initBoundaries: double []) // Erzeugt Variable mit den Intervall-Grenzen
    method addValue (info: double)                                // Erzeugt neue Intervall-Grenze (Wert)
    method clone (): Variable                                     // Erzeugt identische Kopie von this
    method deleteValue (valIndex: int)                            // Löscht den valIndex'ten Wert (mit Anpassung)
endclass

public class KBE isa Exception
    attribute code: constant KBECode                              // Fehler-Code aus vordefinierter Fehlertabelle
    attribute info: bound Vector of Object                        // Zusätzliche Informationen zur Fehlerursache
    constructor create (code: int)                                // Erzeugt Fehler-Objekt mit angegebenem Code
    method addInfo (obj: Object)                                  // Fügt ein Objekt zu info hinzu
    public method getCode (): KBECode                             // Liefert code
    public method getInfo (): Vector of Object                    // Liefert info
    public method toString (): String                             // Liefert eine (englische) Textform des Fehlers
endclass
```

```
class IterationManager
    attribute actLEGSum: double                      // Summe der Konfigurationen in akt. LEG
    attribute actTree: int                           // Index des akt. LEG-Baumes bzgl. legLists
    attribute globalItCount: int                     // Anzahl der globalen Iterationen bisher
    attribute leg: LEG                               // Die aktuelle LEG
    attribute localItCount: int                      // Anzahl der lokalen Iterationen bisher
    attribute maxLocalIts: constant int              // Max. Anzahl lokaler Iterationen
    attribute relativeEntropy: double                // Zuletzt bewirkte relative Entropieänderung
    attribute roots: LEG[]                           // Verweise auf die Wurzeln der LEG-Bäume
    attribute rules: Rule[]                          // Die aktuell zu lernenden Regeln der LEG
    attribute ruleCount: int                         // Die Anzahl an Regeln in rules
    attribute ruleStep: int                          // Die aktuelle Position in rules
    attribute runningEntropy: double                 // Laufende Entropy
    attribute stack: Stack                           // Stack (verwendet zur Simulation der Rekursion)
    attribute threshold: constant double             // ε-Schwelle für Abbruch der Iterationen
    constructor create (threshold: double, maxLocalIts: int, absEntr: double, lf: LEGForest)
    method end ()                                    // Beendet die Iteration (Konsistenz herstellen)
    method getStatus (): double[]                    // Liefert Informationen über den aktuellen
Zustand
    method step (): boolean                          // Nächster Iterations-Schritt (wenn fertig: true)
endclass

class LEG isa ProbTable
    attribute rules: Vector of Rule                  // Liste der Regeln, die assigned sind
    constructor create (vars: Variable [])           // Erzeugt eine LEG mit den angeg. Variablen
    method addRule (rule: Rule)                      // Fügt rule zu rules hinzu
    method assignAndCollectEvidence (vs: Variable [], vals: int [])    // Schritt 1 von assignEvidence
    method clone (nvl: VarList, nrl: RuleList): LEG  // Erzeugt Kopie von this mit Verweisen auf nvl, nrl
    method deleteRule (rule: Rule)                   // Löscht rule aus rules
    method deleteVar (var: Variable)                 // Löscht die angegebene Variable aus der LEG
    method distributeEvidence ()                     // Führt Kalibrierung aller Unterbäume aus
    method getLEGInfo (as: Vector, parentIndex: int) // Füllt as mit einem Info-Block zu this
    method getRules (): Vector of Rule               // Liefert rules
    method linkTo (parent: LEG)                      // Verbindet this über Separator mit einer parent-
LEG
    method reinit ()                                 // Führt ein Re-Init durch
    method sortedList (legs: Vector)                 // Liefert eine sortierte Liste der LEGs unterhalb
    method thickness (var0, var1: Variable)          // Wendet die Formel für Kantendicke an
    method updateSeparators (direction: int)         // Aktualisiert die adjazenten Separatoren
endclass

class LEGForest
    attribute roots: bound Vector of LEG             // Einstiegspunkte der Bäume (Wurzeln)
    constructor create ()                            // Erzeugt neuen (leeren) LEGForest
    constructor create (varList: VarList, ruleList: RuleList)   // Erzeugt neuen LEGForest aus Vorgaben
    method absEntropy (): double                     // Ermittelt die absolute Entropie des Waldes
    method addTree (root: LEG)                       // Fügt den (entarteten) Baum ans Ende der roots
    method assignEvidence (vars: Variable [], vals: int[])   // Weist den angegeb. Variablen Evidenz zu
    method clone (nvl: VarList, nrl: RuleList): LEGForest    // Erzeugt Kopie von this mit Verweisen auf nvl, nrl
    method createClique (clique: VarVector): LEG     // Erzeugt Hilfs-LEG für temp. Berechnungen
    method deleteVar (var: Variable)                 // Löscht eine Variable aus allen LEGs
    method findLEG (rule: Rule): LEG                 // Sucht eine LEG, in die rule passt
    method getLEGInfo (): Vector of (int [])         // Liefert strukturelle Infos über den LEG-Wald
    method getRoots (): Vector of LEG                // Liefert roots
    method initCopy ()                               // Kopiert für jede ProbTable.p in
ProbTable.copyP
    method linkVarsAndRules (vl: VarList, rl: RuleList)   // Sucht für Variablen und Regeln die passende
LEG
    method reinit ()                                 // Führt ein Re-Init durch
    method reset ()                                  // Setzt alle LEGs auf Gleichverteilung zurück
    method restoreCopy ()                            // Stellt Kopie der ProbTable.p wieder her
endclass
```

```
public class KBase isa InfoObject
    attribute graph: exclusive bound VarGraph[2]                       // Die beiden Variablengraphen (oder null)
    attribute itMan: exclusive bound IterationManager                  // Aktueller IterationManager (falls existent)
    attribute legForest: bound LEGForest                               // LEG-Wald der Wissensbasis
    attribute ruleList: constant bound RuleList                        // Aktuelle Regelmenge
    attribute varList: constant bound VarList                          // Aktuelle Variablenmenge
    public method absEntropy (): double                                // Liefert die absolute Entropie der Verteilung
    public method addRule (ruleString: String, preProb: PreProb)       // Erzeugt eine neue Regel
    public method addValue (varIndex: int, valInfo: Object)            // Erzeugt neuen Variablenwert
    public method addVar (name: String, type: VarType, initValues: Object [])      // Erzeugt neue Variable
    public method assignEvidence (varIndexList: int [], valIndexList: int [])       // Evidenzzuweisung
    public method assignEvidenceReset ()                               // Rücknahme jeglicher Evidenzzuweisungen
    public method beginIteration (threshold: double, maxLocalIts: int) // Startet Iteration
    public method clone (withRules: boolean): KBase                    // Erzeugt eine Kopie von this (mit oder ohne Regeln)
    public method deleteRule (ruleIndex: int)                          // Löscht eine Regel
    public method deleteValue (varIndex: int, valIndex: int)           // Löscht einen Variablenwert
    public method deleteVar (varIndex: int)                            // Löscht eine Variable
    public method endIteration ()                                      // Beendet die Iteration (löscht itMan)
    public method enumerateVars (method: EnumMethod)                   // Numeriert die Variablen neu
    public method expectedValue (varIndex: int): double                // Liefert den Erwartungswert einer Variable
    public method exportFile (file: File)                              // Schreibt Variablen und Regeln in eine Datei
    public method getIterationStatus (): double []                     // Liefert Informationen zum Status der Iteration
    public method getLEGInfo (): Vector of (int [])                    // Liefert strukturelle Infos über den LEG-Wald
    public method getRuleActivity (ruleIndex: int): RuleActivity       // Liefert den Aktivitätszustand einer Regel
    public method getRuleAlpha (ruleIndex: int): double                // Liefert den Lagrange-Parameter einer Regel
    public method getRuleCount (): int                                 // Liefert die aktuelle Anzahl an Regeln
    public method getRuleLearnCount (ruleIndex: int): int              // Liefert die Zahl der Lernschritte einer Regel
    public method getRulePreProb (ruleIndex: int): PreProb             // Liefert die vorgeg. bed. W'keit einer Regel
    public method getRuleString (ruleIndex: int)                       // Liefert eine Regel als String
    public method getRuleStringDNF (ruleIndex: int)                    // Liefert eine Regel als String in DNF
    public method getRuleSumEntropy (ruleIndex: int): double           // Liefert die Entropiesummen einer Regel
    public method getRuleUserInfo (ruleIndex: int, key: String): String // Liefert die User-Infos einer Regel
    public method getUtility (varIndex: int, valueIndex: int): double  // Liefert den Utility eines Variablenwertes
    public method getValueName (varIndex: int, valIndex: int): String  // Liefert einen String zu einem Wert
    public method getVarArity (varIndex: int): int                     // Liefert die Anzahl der Werte einer Variable
    public method getVarCount (): int                                  // Liefert die aktuelle Anzahl an Variablen
    public method getVarEnum (varIndex: int): int                      // Liefert den Eliminations-Index einer Variable
    public method getVarName (varIndex: int): String                   // Liefert den Namen einer Variable
    public method getVarRules (varIndex: int, conc: boolean): int []   // Liefert Regeln mit Variable
    public method getVarType (varIndex: int): VarType                  // Liefert den Typ einer Variable (über typecheck)
    public method getVarUserInfo (varIndex: int, key: String): String  // Liefert User-Info einer Variable
    public method graphAdjacentVars (gt: GraphType, var: int): int []  // Liefert die Nachbarn einer Variable
    public method graphEdgeThickness (varIndex0: int, varIndex1: int): double // Liefert Kantendicke
    public method graphEdgeType (gt: GraphType, from: int, to: int): EdgeType      // Kantenart von from nach to
    public method importFile (file: File)                              // Liest Variablen und Regeln aus einer Datei
    public method rebuild ()                                           // Baut den LEgForest neu auf
    public method reinit ()                                            // Führt ein Re-Init auf dem LEGForest durch
    public method renameVar (varIndex: int, name: String)              // Nennt eine Variable um
    public method reset ()                                             // Setzt alle LEGs auf Gleichverteilung zurück
    public method ruleIsFact (ruleIndex: int): boolean                 // Prüft, ob eine Regel ein Fakt oder Conditional ist
    public method rulePremProb (ruleIndex: int): double                // Liefert die aktuelle Prämissenwahrscheinlichkeit
    public method ruleProb (ruleIndex: int): double                    // Ermittelt die aktuelle bed. W'keit einer Regel
    public method ruleStringCheck (ruleString: String)                 // Prüft einen Regel-String auf Korrektheit (sonst KBE)
    public method setRulePreProb (ruleIndex: int, preProb: PreProb)    // Ändert die vorgeg. bed. W'keit
    public method setRuleUserInfo (ruleIndex: int, key: String, info: String)      // Ändert User-Infos einer Regel
    public method setUtility (varIndex: int, valueIndex: int, utility: double)     // Setzt den Nutzen eines Wertes
    public method setVarEnum (varIndex: int, enum: int)                // Numeriert eine Variable neu
    public method setVarUserInfo (varIndex: int, key: String, info: String)        // Ändert Use r-Info einer Variable
    public method stepIteration ()                                     // Führt den nächsten Iterationsschritt aus
    public method toggleRuleActivity (ruleIndex: int)                  // Wechselt den Aktivierungszustand einer Regel
    public method valueProb (varIndex: int, valIndex: int): double     // Liefert die aktuelle W'keit des Wertes
endclass
```

```
abstract class Literal isa Expression
    attribute var: constant Variable                      // Variable, auf die sich das Literal bezieht
    method getVars (vars: VarVector)                      // Liefert die vorkommenden Variablen (füllt vars)
endclass

class MixedVarGraph
    attribute edgeType: (Vector of Integer) []            // Kantentyp zu jeder Kante
    attribute varList: VarList                            // Verweis auf die VarList der KBase
    constructor create (varList: VarList, ruleList: RuleList)   // Erzeugt neuen MixedVarGraph aus angeg.
Regeln
    method edgeType (from: int, to: int): EdgeType        // Liefert die Art der Kante zwischen zwei Knoten
endclass

class NominalValue isa Value
    attribute name: constant String                       // Bezeichnung des Wertes
    constructor create (name: String)                     // Erzeugt neuen Wert mit dem angeg. Namen
    method toString (): String                            // Wandelt this in einen String um (zur Ausgabe)
endclass

class NominalVariable isa Variable
    constructor create (name: String, initValues: String [])   // Erzeugt neue Variable mit den angeg. Werten
    method addValue (info: String)                        // Erzeugt neuen Variablenwert
    method clone (): Variable                             // Erzeugt identische Kopie von this
    method deleteValue (valIndex: int)                    // Löscht den valIndex'ten Wert
endclass

class NumberValue isa Value
    attribute number: constant double                     // Zahlenwert
    constructor create (number: double)                   // Erzeugt neuen Wert mit angeg. Zahlen-Wert
    method getNumber (): double                           // Liefert number
    method toString (): String                            // Wandelt this in einen String um (zur Ausgabe)
endclass

class NumberVariable isa Variable
    constructor create (name: String, initValues: double [])   // Erzeugt Variable mit den angeg. Werten
    method addValue (info: Double): int                   // Erzeugt neuen Variablenwert
    method clone (): Variable                             // Erzeugt identische Kopie von this
    method deleteValue (valIndex: int)                    // Löscht den valIndex'ten Wert
endclass

public abstract class PreProb isa Object
    public method toString (): String                     // Erzeugt einen String zur Ausgabe
endclass

abstract class ProbTable isa VarVector
    attribute copyP: double []                            // Temporäre Kopie der Konfigurationen
    attribute neighbors: bound Vector of ProbTable        // Nachbar-Tabellen im Baum (0=Parent)
    attribute p: double []                                // Liste der Konfigurationen
    constructor create (vars: Variable [])                // Erzeugt neue ProbTable mit Variablen
    method absEntropy (): double                          // Berechnung der abs. Entropie nach Formel
    method addValue (var: Variable, valIndex: int)        // Erweitert Tabellen und evtl. Rekursion zu
Nachbarn
    method clearConfigs (indexList: IndexList)            // Löscht die Konfigurationen der IndexList
    method deleteValue (var: Variable, valIndex: int)     // Verkleinert die Tabellen und evtl. Rekursion
    method divideConfigs (il: IndexList, f: double)       // Teile die p's aus der IndexList il durch Faktor f
    method getLEGsWithVar (var: Variable, pts: Vector of ProbTable, cameFrom: ProbTable)     // Füllt pts
    method getNeighbors (): Vector of ProbTable           // Liefert neighbors
    method initCopy ()                                    // Kopiert p in copyP
    method makeTree (parent: ProbTable, legs: Vector)     // Erzeugt einen Baum ab einer angeg. Wurzel
    method multiplyConfigs (il: IndexList, f: double)     // Multipliziert die p's aus der IndexList il mit
Faktor f
    method normTree (sum: double)                         // Normiert this und alle Söhne mit einem Faktor
    method reset ()                                       // Setzt this und alle Söhne auf Gleichverteilung
    method restoreCopy ()                                 // Kopiert copyP in p
    method sumConfigs (il: IndexList): double             // Summiert alle p's aus der IndexList
    method sumValueConfigs (var: Variable, valIndex: int): double     // Summiert alle p's mit var=valIndex
endclass
```

class **RangeLiteral** isa Literal
    attribute **value**: constant Value // Wert, der im Literal als Vergleichswert gilt
    constructor **create** (var: Variable, value: Value) // Erzeugt neues Literal mit dem angeg. Wert
    method **clone** (nvl: VarList): Expression // Erzeugt Kopie von this mit Verweisen nach nvl
    method **requiresValue** (var: Variable, valIndex: int): boolean // Prüft, ob der Wert benötigt wird
    method **toString** (): String // Wandelt this in einen String um (zur Ausgabe)
endclass

abstract class **Rule** isa InfoObject
    attribute **activity**: RuleActivity // Aktivierungszustand der Regel
    attribute **alpha**: double // Aktueller Lagrange-Parameter der Regel
    attribute **assignedLEG**: LEG // Die minimale LEG, in die this passt oder null
    attribute **ils**: IndexList [2] // Die beiden IndexListen für Lernen und Anfragen
    attribute **index**: int // Index in der RuleList der KBase
    attribute **learnCount**: int // Anzahl der Lernschritte in der Iteration
    attribute **preProb**: bound PreProb // Vorgegebene bed. Wahrscheinlichkeit
    attribute **sumEntropy**: double // Bisherige Summe der Entropien
    constructor **Rule** (preProb: PreProb) // Für Inheritance
    method ***clone*** (nvl: VarList): Rule // Erzeugt Kopie von this mit Verweisen auf nvl
    method **delete** () // Bereitet das Löschen der Regel vor
    method **exportFile** (file: File) // Schreibt die Regel in die angeg. Datei
    method **getActivity** (): RuleActivity // Liefert activity
    method **getAlpha** (): double // Liefert alpha
    method ***getDNFString*** (): String // Liefert die Regel als DNF
    method **getIndex** (): int // Liefert index
    method **getLearnCount** (): int // Liefert learnCount
    method **getLEG** (): LEG // Liefert assignedLEG
    method **getPreProb** (): PreProb // Liefert preProb
    method **getSumEntropy** (): double // Liefert sumEntropy
    method ***getVars*** (): VarVector // Liefert die vorkommenden Variablen
    method ***invalidateDNF*** () // Berechnet die DNF neu
    method **invalidateIndexLists** () // Löscht die IndexLists (Neuberechnung on
demand)
    method **learn** (): double // Lernt die Regel in die LEG und liefert Entropie
    method **multiplyAlpha** () // Multipliziert die passende LEG mit alpha
    method ***prob*** (): double // Ermittelt die aktuelle bed. Wahrscheinlichkeit
    method ***requiresValue*** (var: Variable, valIndex: int): boolean // Prüft, ob der Wert benötigt wird
    method **reset** () // Setzt alpha, learnCount und sumEntropy zurück
    method **setAlpha** (alpha: double) // Ändert alpha
    method **setIndex** (index: int) // Ändert index
    method **setLEG** (leg: LEG) // Setzt assignedLEG
    method **setPreProb** (preProb: PreProb) // Setzt preProb
    method **toggleActivity** () // Wechselt activity (nur active und activable)
    method ***toString*** (): String // Erzeugt einen String aus dem Ausdrucksbaum
endclass

class **RuleList**
    attribute **rules**: bound Vector of Rule // Liste aller Regeln (die *Regeln* sind bound!)
    method **addRule** (rule: Rule) // Fügt Regel ans Ende der Liste ein
    method **clone** (nvl: VarList): RuleList // Erzeugt Kopie von this mit Verweisen auf nvl
    method **deleteRule** (ruleIndex: int) // Löscht die angeg. Regel und paßt die Indizes
an
    method **getCount** (): int // Liefert die Zahl der Regeln in this
    method **getRule** (ruleIndex: int): Rule // Liefert die Regel mit dem Index ruleIndex
endclass

class **RuleParser**
    attribute **scanner**: constant Scanner // Scanner zur Lieferung der nächsten Tokens
    attribute **varList**: VarList // Verweis auf die VarList der Wissensbasis
    constructor **create** (varList: VarList, scanner: Scanner) // Erzeugt neuen RuleParser
    method **rule** (preProb: PreProb): Rule // Versucht eine Regel aus dem Scanner zu lesen
endclass

```
class Scanner
    attribute token: Object                         // String, Double oder Char
    attribute nextToken: Object                     // Für das Vorausschauen des Parsers
    constructor create (file: File)                 // File kann Datei, auch z.B. StringInputStream
sein
    method forward ()                               // Liest das nächste Token (token := nextToken)
    method getToken (): Object                      // Liefert token
    method getNextToken (): Object                  // Liefert nextToken
endclass
```

```
class Separator isa ProbTable
    attribute indexLists: bound IndexList [2][]     // Indexlisten zur schnellen Kalibrierung
    constructor create (leg0, leg1: LEG)            // Erzeugt neuen Separator zwischen zwei LEGs
    method calibrate (destIndex: int)               // Kalibriert die beiden adjazenten LEGs
    method clone (nvl: VarList, nrl: RuleList): Separator   // Erzeugt Kopie von this mit neuen Verweisen
    method multiply (destIndex: int)                // Multipliziert die angeg. adjazente LEG
    method update (sourceIndex: int)                // Aktualisiert mit Randsummen aus sourceIndex
endclass
```

```
public class SinglePreProb isa PreProb
    attribute p: constant double                    // Vorgegebene Wahrscheinlichkeit
    public constructor create (p: double)           // Erzeugt neue SinglePreProb
    public method getP (): double                   // Liefert p
    public method toString (): String               // Wandelt this in String (zur Ausgabe) um
endclass
```

```
class SubExpression isa Expression
    attribute sub: constant Expression              // Verweis auf das linke Ende des
Unterausdruckes
    constructor create (neg: boolean, sub: Expression)   // Erzeugt neue SubExpression
    method clone (nvl: VarList): Expression         // Erzeugt Kopie von this mit Verweisen nach nvl
    method requiresValue (var: Variable, valIndex: int): boolean     // Prüft, ob der Wert benötigt wird
    method toString (): String                      // Wandelt this in einen String um (zur Ausgabe)
endclass
```

```
abstract class Value
    attribute utility: double                       // Utility-Wert für die Berechnung d.
Erwartungswertes
    constructor create (utility: double)            // Erzeugt neuen Value mit Anfangs-Nutzen
(Utility)
    method getUtility (): double                    // Liefert utility
    method setUtility (utility: double)             // Setzt utility
    method toString (): String                      // Wandelt this in einen String um (zur Ausgabe)
endclass
```

```
class VarGraph
    attribute adj: VarVector []                     // Adjazenzlisten für jede Variable
    attribute varList: VarList                      // Verweis auf die VarList der KBase
    constructor create (varList: VarList, ruleList: RuleList)   // Erzeugt neuen VarGraph aus angeg. Regeln
    method connect (var0, var1: Variable)           // Fügt eine Kante zwischen var0 und var1 ein
    method delete (var: Variable)                   // Isoliert die Variable
    method edgeType (from: int, to: int): EdgeType  // Liefert die Art der Kante zwischen zwei Knoten
    method enumerateVars (method: EnumMethod)       // Numeriert die Variablen mit der angeg. Methode
    method getAdjacentVars (varIndex: int): int []  // Liefert die benachbarten Knoten einer Variable
    method neighbors (varIndex: int): VarVector     // Liefert die Adjazenzliste zu einer Variable
endclass
```

abstract class **Variable** isa InfoObject
attribute **enum**: int // Index in Eliminationsreihenfolge
attribute **index**: int // Index in der aktuellen VarList
attribute **inLEG**: LEG // Minimale LEG, die this enthält
attribute **inRules**: Vector of Rule // Liste der Regeln mit Vorkommen von this
attribute **name**: String // Bezeichnung der Variable
attribute **valueList**: Vector of Value // Liste der zulässigen Werte
constructor **Variable** (name: String) // Erzeugt neue Variable mit angeg. Namen
method **addRule** (rule: Rule) // Fügt eine Regel in die inRules-Liste hinzu
method ***clone*** (): Variable // Erzeugt identische Kopie von this
method **deleteRule** (rule: Rule) // Löscht die rule aus der inRules-Liste
method **expectedValue** (): double // Liefert den Erwartungswert (ähnlich valueProb)
method **exportFile** (file: File) // Schreibt die Variable in die angeg. Datei
method **getArity** (): int // Liefert die Zahl der aggregierten Variablenwerte
method **getEnum** (): int // Liefert enum
method **getIndex** (): int // Liefert index
method **getLEG** (): LEG // Liefert inLEG
method **getName** (): String // Liefert name
method **getRules** (conc: boolean): Vector of Rule // Liefert inRules (conc: Nur rechte Seite)
method **getValue** (valIndex: int): Value // Liefert den valIndex'ten Wert
method **getValueIndex** (value: Value): int // Liefert den Index eines Werte-Objektes
method **rename** (name: String) // Setzt name
method **setEnum** (enum: int) // Setzt enum
method **setIndex** (index: int) // Setzt index
method **setLEG** (leg: LEG) // Setzt inLEG
method **valueProb** (valIndex: int): double // Liefert die aktuelle bed. W'keit eines Wertes
endclass

class **VarList** isa VarVector
method **addVar** (var: Variable) // Fügt eine Variable hinzu
method **clone** (): VarList // Erzeugt identische Kopie von this
method **deleteVar** (varIndex: int) // Löscht die angeg. Variable (mit Anpassung
index)
endclass

class **VarVector**
attribute **vars**: Vector of Variable // Verweise auf die Variablen, sortiert nach
var.index
constructor **create** (vars: Variable []) // Erzeugt neuen VarVector
method **addVar** (var: Variable) // Fügt Variable sortiert in Liste ein (ggfs.
setIndex)
method **contains** (var: Variable): boolean // Prüft, ob die angeg. Variable in der Liste ist
method **deleteVar** (var: Variable) // Löscht die angegebene Variable
method **find** (name: String): Variable // Sucht die Variable mit dem Namen name
method **getVar** (index: int): Variable // Liefert die Variable an der index'ten Stelle
method **getVarCount** (): int // Liefert die Zahl der Variablen in this
endclass

Mit der Fertigstellung dieser *Class Descriptions* und den Tests der bisherigen Modelle auf gegenseitige Konsistenz und Vollständigkeit ist die Entwurfsphase des Projekts zunächst abgeschlossen. Die Entwurfsphase hatte die Aufgabe, die internen Abläufe zur Erfüllung der System-Operationen aus der *Analysephase* zu präzisieren. Dazu wurden zahlreiche Konzepte entwickelt und bis zu einem gewissen Abstraktionsgrad auch programmierbar formuliert. Aufbauend auf den *Class Descriptions* und den entwickelten Algorithmen kann nun in einer geeigneten objektorientierten Programmiersprache mit der Implementierung begonnen werden.

4 Implementierung, Test und Dokumentation

Dieses Kapitel beschreibt die Vorgehensweise bei der Implementierung von SPIRIT und meine dabei gemachten Erfahrungen. In der Implementierungsphase werden gemäß FUSION die aus dem Entwurf gewonnenen Modelle in ausführbaren Programmcode umgesetzt. Voraussetzung hierzu ist zunächst die Wahl einer geeigneten objektorientierten Programmiersprache, die im Abschnitt 4.1 erläutert wird. Dieser Abschnitt gibt eine kurze Übersicht über die ausgewählte Sprache JAVA und begründet dessen Eignung für ein mit FUSION entworfenes Software-Projekt. Die eigentliche Codierung des Systemkerns ist im anschließenden Abschnitt 4.2 dargestellt. Parallel zur Kodierung sollen üblicherweise Tests durchgeführt werden, die Korrektheit und Effizienz der Implementierung prüfen. Diese Tests und die dazu entwickelten Testtreiber werden in Abschnitt 4.3 behandelt. Schließlich geht Abschnitt 4.4 kurz auf die Dokumentation der Klassenbibliothek für Anwendungsprogrammierer ein.

4.1 JAVA als Programmiersprache im Software Engineering

Eine für die Implementierung der SPIRIT-Klassenbibliothek geeignete Programmiersprache sollte zur Erfüllung der Anforderungen aus Abschnitt 1.2 vor allem folgende Eigenschaften haben, bzw. unterstützen:

- *Effizientes Laufzeitverhalten*, da oft mit großen Zahlenmengen und Tabellen gearbeitet werden muß,
- *Ausfallsicherheit und Präzision*, da SPIRIT ein entscheidungsunterstützendes System ist und verläßliche Informationen liefern sollte,
- *Plattformunabhängigkeit und weite Verbreitung*, um SPIRIT möglichst vielen Nutzern zugänglich zu machen,
- *Vorhandensein brauchbarer Programmierwerkzeuge* zur effizienten Systementwicklung, und
- *Konsistenter Übergang von Entwurfsmodell (aus FUSION) zum Programm*, so daß der Entwurf als Grundlage für das Verständnis und die Wartung des Quelltextes herangezogen werden kann.

Da Analyse und Design des Projektes mit der objektorientierten Methode FUSION durchgeführt wurden, kamen nur objektorientierte Sprachen in Frage. Auch mußte abgeschätzt werden, welche Sprachen in Zukunft weite Verbreitung erlangen werden, wodurch allzu exotische Sprachen nur mit Vorsicht in die engere Auswahl genommen werden sollten. Hierzu zähle ich auch die in Fachkreisen hochgelobten Programmiersprachen EIFFEL und SMALLTALK, für die es im PC-Bereich praktisch keine brauchbaren (bzw. erschwinglichen) Entwicklungsumgebungen gibt. Auch das im PC-Sektor sehr populäre DELPHI von BORLAND schied für dieses Projekt aus, weil es zu keinem vergleichbaren System im UNIX- oder MACINTOSH-Bereich kompatibel ist. Somit blieb aus meiner Sicht nur die Wahl zwischen C++ und der neuen Sprache JAVA.

Die in den Vorgängerversionen verwendete Programmiersprache C++ kann die oben aufgeführten Anforderungen nur teilweise erfüllen. Insbesondere bietet diese Sprache wenig Unterstützung zur Entwicklung von (absturz-) sicheren Programmen, da es oft erforderlich ist, mit Zeigern u.ä. auf unterster Maschinenebene zu arbeiten. Auch liesse sich in C++ zwar eine relativ plattformunabhängige Klassenbibliothek aufbauen (wie in den Vorgängerversionen) - spätestens bei der Entwicklung von Oberflächen müßte man allerdings auf systemspezifische Module zurückgreifen.

1995 trat die Firma SUN mit der neuen Programmiersprache JAVA an die Öffentlichkeit. Innerhalb kürzester Zeit erlangte diese Sprache große Bekanntheit und löste sehr kontroverse Diskussionen aus. Auch jetzt (Anfang 1997), fast zwei Jahre nach ihrem Erscheinen, ist schwer abschätzbar, ob und in welcher Form sich die Sprache durchsetzen wird. Der folgende Abschnitt gibt einen kurzen Überblick über JAVA.

4.1.1 Kurze Übersicht über JAVA

JAVA wurde ursprünglich als spezielle Sprache für das Internet angesehen, da sie besondere Unterstützung zur Kommunikation über das Netz bereitstellt. Allerdings ist JAVA eine sehr allgemein einsetzbare, flexible Sprache, die für ähnliche Zwecke wie beispielsweise C++ verwendbar ist. Im folgenden will ich JAVA daher vor allem aus allgemeiner Sicht betrachten und seine Internet-Unterstützung vernachlässigen. Auch soll nur ein grober Überblick über die Sprache und den aktuellen Zustand des Marktes gegeben werden. Für weitere Informationen möchte ich auf die entsprechende einführende Literatur verweisen (sh. Literaturverzeichnis).

Aufgrund der Tatsache, daß JAVA erst in den 90er Jahren entwickelt wurde, konnten positive und negative Erfahrungen mit bestehenden Programmiersprachen für den Entwurf berücksichtigt werden. Allerdings machte SUN einige Anlehnungen an die Sprache C++ und ging somit einige Kompromisse ein, um die Sprache einem großen Personenkreis schmackhaft zu machen. Daher ähnelt die Syntax von JAVA der von C++. (Das kurze

Programmbeispiel aus Abbildung 22 (Seite 63) vermittelt einen groben Eindruck vom typischen Aussehen von JAVA-Quelltext.) Dies erleichert sowohl die Konvertierung von Quelltexten, als auch den Umstieg von C++-Programmierern auf JAVA. Während C++ allerdings nur als objektorientierte Erweiterung der älteren Sprache C angesehen werden kann, ist JAVA eine reine objektorientierte Sprache. Ich würde JAVA als eine optimale "Teilmenge" von C++ bezeichnen, da sie viele Vorteile von C++ übernimmt und dessen wesentliche Nachteile vermeidet.

Ein JAVA-Programm besteht aus einer Menge von Objekt-Klassen. Mehrere Klassen können zu sog. *Packages* zusammengefaßt werden. *Packages* sind Klassenbibliotheken, die ihrerseits über lokale Klassen und Unter-*Packages* verfügen können. Klassen bestehen aus Attributen und Methoden, wobei die Attribute entweder atomare Datentypen (wie int, char, double), Zeiger auf Objekte oder beliebig dimensionierte Arrays sein können. Methoden können die üblichen Kontrollstrukturen enthalten (while, for, if), über lokale Variablen verfügen und rekursiv sein. Attribute, Methoden und Klassen können verschiedene Sichtbarkeitsebenen haben. Im wesentlichen sind dies public (überall sichtbar), *friend* (innerhalb des Packages sichtbar), protected (von allen Unterklassen sichtbar) und private (nur innerhalb einer Klasse sichtbar. Jede Klasse erbt die Eigenschaften von genau einer Oberklasse, es fehlt also die Möglichkeit der Mehrfachvererbung. Statt dessen können sog. *Interfaces* verwendet werden. Ein *Interface* ist die Definition einer abstrakten Klasse, die nur aus abstrakten Methoden besteht. Eine Klasse *implementiert* ein Interface, wenn sie deren abstrakte Methoden überschreibt. Jede Klasse kann beliebig viele *Interfaces* implementieren, wodurch eine Art Mehrfachvererbung erreicht werden kann.

Da der JAVA-Standard eine umfangreiche Klassen-Bibliothek vorschreibt, ist eine sehr hohe Plattformunabhängigkeit gewährleistet. Die Sprache verfügt über einen recht kleinen Befehlssatz und wenige vorgegebene atomare Datentypen (u.a. int, double und char). Letztere sind (im Gegensatz zu C++) auch für alle Rechnertypen genormt. Zum Sprachstandard gehört auch eine String-Klasse und sogar eine Klassenbibliothek zur Programmierung von plattformunabhängigen graphischen Oberflächen (GUIs). Am Rande sei hierzu erwähnt, daß JAVA-Programme auch direkt in *WWW-Browsern* ablaufen können. Diese als sog. *Applets* bezeichneten Programme werden beim Öffnen einer WWW-Seite über das Netz auf den aufrufenden Rechner geladen und von der dortigen JAVA-Umgebung lokal ausgeführt. Die hierdurch induzierten Sicherheitsprobleme, die zu einer Einschränkung der Zugriffsmöglichkeiten von JAVA-*Applets* führten, will ich hier nicht weiter ausführen - auch weil deren Diskussion noch nicht abgeschlossen ist.

Der wesentlichste Punkt, der JAVA von den meisten anderen Sprachen unterscheidet, ist daß der Compiler nicht Maschinensprache erzeugt, sondern sog. *Byte-Code*, der zur Ausführung von einer virtuellen Maschine interpretiert wird. Der *Byte-Code* ist plattformunabhängig, so daß ein einmal übersetztes JAVA-Programm auf allen Systemen lauffähig ist, für die es eine JAVA-Laufzeitumgebung gibt (hierzu gehören inzwischen WINDOWS95, OS/2, UNIX, MACINTOSH und neuerdings sogar reine JAVA-Maschinen). Der Nachteil dieses Konzeptes ist, daß die Interpretation des *Byte-Codes* (verglichen mit Maschinensprache) deutliche Geschwindigkeitseinbußen nach sich zieht. Inzwischen gibt es sehr erfolgreiche Ansätze, die diesen Nachteil ausgleichen: Es gibt einige sog. *Just-In-Time-Compiler*, die den *Byte-Code* von Funktionen beim Funktionsaufruf ("*on the fly*") in Maschinensprache übersetzen und erst dann ausführen. Zwar geht dadurch beim jeweils ersten Funktionsaufruf etwas Zeit verloren, die Praxis zeigt aber, daß dieser Verlust kaum spürbar ist. Einige in Fachmagazinen und von mir durchgeführte Benchmark-Tests belegen, daß mit einem *Just-In-Time-Compiler* ausgeführte JAVA-Programme durchschnittlich etwa um den Faktor 1.7 langsamer sind als vergleichbare optimierte C++-Programme. Für extrem geschwindigkeitskritische Anwendungen stellt JAVA daher die Möglichkeit bereit, externe, in Maschinensprache vorliegende Methoden als sog. *Native-Calls* aufzurufen. Hierdurch wird allerdings die Plattformunabhängigkeit verletzt.

JAVA unterstützt übrigens *Multithreading* (die Ausführung von mehreren Prozessen für ein Programm gleichzeitig). Durch Klassen wie Thread, die ein Teil der Standard-Bibliothek von JAVA sind, können auf sehr elegante (und plattformunabhängige) Weise Probleme parallel gelöst werden.

Während es in C++ häufig zu Zugriffsverletzungen durch falsch gesetzte *Pointer* und überschrittene Speicherbereiche kommt, können in JAVA geschriebene Programme aufgrund der Verwendung der virtuellen Laufzeitmaschine nicht abstürzen. Da JAVA zudem keine Zeigerarithmetik unterstützt, können Zeiger nur auf *null* oder ein bestehendes Objekt verweisen: Objekte können nicht explizit gelöscht werden, sondern bestehen so lange, wie es eine Referenz auf sie gibt. Besteht kein Verweis mehr auf ein Objekt, so wird es von der nächsten *Garbage Collection* (einem bei Bedarf automatisch aufgerufenen Verfahren zur Freigabe von nicht mehr benötigten Resourcen) aus dem Speicher entfernt. Jeder andere Fehler (z.B. Zugriff auf einen *null*-Pointer, Überschreitung von Array-Grenzen, Datei-Fehler) löst eine *Exception* aus. Dadurch wird der normale Programmablauf unterbrochen und in die jeweils aufrufende Funktion zurückgekehrt, bis eine passende *Exception-Handling-Routine* die Behandlung des Fehlers übernimmt (vgl. hierzu Abschnitt 3.1.9).

Abschließend noch ein paar Worte zu den derzeitig (Anfang '97) vorhandenen Tools und Entwicklungsumgebungen. SUN stellte zusammen mit JAVA bereits ein einfaches *Software-Development-Kit* (das

JDK) vor, das aus Compiler, Laufzeitmaschinen, Debugger und kleineren Hilfsprogrammen besteht und für alle gängigen Plattformen frei verfügbar ist. Schon bald kamen kommerzielle Pakete auf den Markt, die sowohl im Bezug auf Performance als auch auf Benutzerfreundlichkeit deutliche Verbesserungen mit sich brachten. Ich verwendete auf meinem WINDOWS95-Rechner die integrierte Entwicklungsumgebung CAFÉ von der Firma SYMANTEC, die Editor, Compiler, Debugger und Projektverwaltung umfaßt und Mitte des Jahres 1996 erschien. Später brachten auch die etablierten Firmen BORLAND und MICROSOFT sehr leistungsfähige JAVA-Pakete auf den Markt. Auch wurde JAVA schon als integraler Bestandteil in das Betriebssystemes OS/2 eingebaut. Dort ist ab Version 4.0 eine JAVA-Maschine integriert und JAVA-*Applications* können genauso wie andere Programme gestartet werden. SUN ging noch einen Schritt weiter und realisierte einen Rechner, der *nur* aus einem JAVA-fähigen *WWW-Browser* und einem Internet-Anschluß besteht. Ob sich dieser mutige Ansatz gegen die Dominanz der traditionellen Systeme (insbes. im PC-Sektor) behaupten kann, läßt sich schwer abschätzen. Betrachtet man jedoch den Markt und die gewünschten Anforderungen in Stellenanzeigen aus der Industrie, so scheint sich ein Trend in Richtung JAVA abzuzeichnen.

4.1.2 Erfahrungen mit JAVA bei der Implementierung von SPIRIT

JAVA (sh. auch Abschnitt 4.1) erwies sich insgesamt als sehr geeignete Implementierungssprache für das mit FUSION entworfene Projekt SPIRIT. Die Sprache ist von Grund auf objektorientiert und verfügt über alle Konstrukte, die zur Umsetzung der Entwurfsmodelle nach FUSION nötig sind:

- *Klassen* zur genauen Realisierung von *Class Descriptions* und den Objekt-Modellen,
- *Modifikatoren* von Attributen, Methoden und Klassen zur Umsetzung der Sichtbarkeitsregeln aus den *Visibility Graphs* und *Class Descriptions*, wie private, protected, (friendly), public, abstract, static und final,
- *Vererbung* zur Umsetzung der Klassenhierarchie (JAVA unterstützt keine direkte Mehrfachvererbung, die im vorliegenden Projekt allerdings auch nicht benötigt wurde und allgemein selten unvermeidlich ist),
- *Kontrollstrukturen* zur Implementierung der Methoden aus den *Object Interaction Graphs,*
- *Zeiger (bzw. Referenzen)* auf Objekte zur Umsetzung von *Relationships* und Aggregation, und
- *Exceptions* für Fehlerbehandlung und Abfrage von Vor- und Nachbedingungen.

Somit liessen sich alle Modelle direkt und ohne umständliche Hilfskonstrukte codieren. Es sollte übrigens ohne weiteres möglich sein, die im Entwurfsprozeß verwendete Syntax (z.B. in den *Class Descriptions* und in den *Object Models*) direkt an die Syntax der Sprache JAVA anzupassen. Hierdurch würde der Zusammenhang zwischen Entwurf und Implementierung noch verstärkt und das Entwurfsdokument weiterhin allgemein verwendbar bleiben.

JAVA verfügt leider auch über einige konzeptuelle Mängel, deren Beseitigung aber für die kommenden JAVA-Standards im Gespräch ist. So ist z.B. die Verwendung von Funktionen als Datentypen (d.h. von Zeigern auf Funktionen) nicht möglich. Auch fehlt ein Konstrukt zur Formulierung von Aufzählungen (wie der beliebte enum-Befehl aus C++). Vor allem aber können keine eigenen Datentypen abgeleitet werden (z.B. über einen Befehl wie type VarIndex = int). Gerade dies ist ein großer Nachteil, weil in den Quelltexten nicht zwischen verschiedenen Interpretationsmöglichkeiten der atomaren Datentypen unterschieden werden kann. So mußten einige geplante Datentypen wie GraphType und EnumMethod durch das einfache int ersetzt werden. Funktionen, die eigentlich einen Variablen-Index vom "Typ" VarIndex erwarten, können somit mit jedem beliebigen int-Wert aufgerufen werden. Als Abhilfe hierzu achtete ich beim Schreiben des Quelltextes darauf, zumindest die Namen der betroffenen Variablen und Parameter so zu benennen, daß ein Rückschluß auf die zulässigen Werte möglich wird (z.B. int varIndex). Dennoch wäre natürlich wünschenswert gewesen, wenn schon der Compiler Typfehler durch ungültige Argumente und Werte melden könnte.

Weiterhin nachteilig erwies sich das Fehlen, bzw. frühe Stadium der zur Implementierung erforderlichen Tools (Compiler, Debugger, etc.). Mit dem verwendeten Programm CAFÉ ließ sich (nach einer Gewöhnungszeit an teilweise sonderbares Systemverhalten) dennoch sehr gut arbeiten. Vergleicht man dieses Tool mit ähnlichen Systemen für C++ (bspw. dem markführenden BORLAND C++), so wird ein enormer Pluspunkt von JAVA deutlich, nämlich die außerordentlich schnelle Übersetzung des Codes und der Wegfall des langsamen Link-Schrittes (der durch dynamische Bindung überflüssig wird, die von der JAVA-Laufzeitmaschine übernommen wird). Auf meinem durchschnittlichen PENTIUM-PC lag die Dauer der Neu-Übersetzung nach Änderung einiger Programm-Module zumeist unter 10 Sekunden, so daß eine sehr flotte Codierung und Fehlersuche möglich wurde und es nebenbei auch wirklich Spaß machte, mit JAVA zu arbeiten. Dennoch läßt der Stand der JAVA-Interpreter und Compiler für die Zukunft noch zahlreiche Optimierungen und Verbesserungen zu. Vor allem die Speicherverwaltung und die *Gargabe-Collection* schien mir bei großen Anwendungen noch recht dürftig zu sein, da der Speicher schnell fragmentiert und dadurch unbrauchbar wird.

Im Gegensatz zu C++ ergaben sich während der Implementierung kaum Probleme durch Abstürze und Zugriffsverletzungen. Dies lag insbesondere daran, daß JAVA-Programme über eine virtuelle Laufzeitmaschine

interpretiert werden (und somit sehr robust sind) und daran, daß Zugriffe auf nicht (mehr) existierende Objekte unmöglich sind. Die Fehlersuche wurde durch den gelungenen *Exception*-Mechanismus wesentlich unterstützt.

Für SPIRIT erwies sich JAVA als sehr geeignet, obwohl ich die eigentlichen Stärken dieser Sprache überhaupt nicht verwendet habe. Wohl keine andere Sprache ist besser für das immer wichtiger werdende Internet vorbereitet. Auch die Unterstützung von *Multithreading* und die plattformunabhängige Klassenbibliothek zur Grafik- und Oberflächenprogrammierung dürfte die Realisierung von Software-Projekten in Zukunft beschleunigen. Zweifellos ist JAVA weit davon entfernt, eine allen Ansprüchen genügende Programmiersprache zu sein. Es gibt z.B. Vorschläge aus der Wissenschaft, JAVA um funktionale Konstrukte zu erweitern. Um JAVA allerdings überhaupt erst einmal auf dem Markt etablieren zu können, fand SUN einen (meiner Ansicht nach gelungenen) Kompromiß zwischen dem theoretisch Wünschenswerten und dem wirtschaftlich Sinnvollen. Sollte JAVA in den kommenden Jahren die derzeitig etablierten Programmiersprachen verdrängen, so wäre erstmals das Problem der Plattformabhängigkeit gelöst und der Markt würde sich zugunsten der Anwender verschieben. Über die jetzige Leistungsfähigkeit von JAVA-basierten Programmen wird in den folgenden Unterabschnitte über die Implementierung von Systemkern und Testprogrammen, sowie der Erstellung von Dokumentationen zu SPIRIT und später im Abschnitt 5.2.2 berichtet.

4.2 Implementierung des Systemkerns

Die Implementierung des eigentlichen Systemkerns, also der SPIRIT-Klassenbibliothek, verlief (wie erhofft) außerordentlich schnell. Nachdem ich sehr viel Zeit in den Entwurf gesteckt hatte, stellte dessen Umsetzung zunächst wenig Probleme dar. Ich begann mit der Formulierung der Schnittstellen zwischen den einzelnen Klassen. Dazu mußte ich lediglich die *Class Descriptions* aus Abschnitt 3.5 in JAVA-Quelltext umwandeln. Dies war sehr leicht möglich. Da mein Entwurf von SPIRIT keine Mehrfach-Vererbung aufweist (die JAVA nicht ohne weiteres unterstützt), war eine direkte Umsetzung der Vererbungshierarchie möglich. Die Köpfe der Methoden liessen sich, ebenso wie die Klassen-Attribute, mit kleinen Anpassungen übernehmen. Lediglich die im Entwurf eingeführten neuen Datentypen, wie EnumMethod und GraphType, mußte ich an JAVA anpassen und auf int vereinfachen. Für die Rümpfe der einzelnen Methoden setzte ich anfangs *Dummy-Code* ein, so daß der Code bald zumindest syntaktisch korrekt war und ich die Module erstmals kompilieren lassen konnte.

Nun konnte ich die korrekten Funktionsaufrufe in den Testtreiber SPECTER (sh. Abschnitt 4.3.2) einbauen und war dadurch prinzipiell in der Lage, die neu zu implementierenden System-Operationen zu testen. Ich begann daraufhin mit der schrittweisen Umsetzung der zuvor konzipierten Algorithmen. Grundlage hierzu waren die *Object Interaction Graphs* aus Abschnitt 3.2, die Algorithmen aus Anhang D und auch eine vergleichende Betrachtung der Quelltexte der letzten Vorgängerversion. Aus dieser ließ sich allerdings nicht mehr viel brauchbarer Code gewinnen - zu unterschiedlich waren die neuen Ansätze. Wenn ich darin dennoch einen brauchbaren Ausschnitt fand (was etwa fünf mal vorkam), war dessen Umsetzung nach JAVA sehr einfach.

FUSION diktiert keine bestimmte Reihenfolge bei der Implementierung der Objektklassen [Col94, S. 103], schlägt aber einen "ereignisgesteuerten" Ansatz vor. Demnach ist es sinnvoll, die Funktionen *so* mit Inhalt zu füllen, wie sie in einem typischen Programmablauf benötigt werden. Dieser Ansatz ist leicht nachzuvollziehen, da die später aufrufbaren Programmteile ohne korrekte Implementierung der vorangehenden nicht getestet werden können. An den *Life-Cycles* aus der *Analysephase* kann abgelesen werden, welche System-Operationen üblicherweise als erstes benötigt werden. In Abschnitt 2.4 habe ich zwei solcher *Life-Cycles* entwickelt, deren zweiter allerdings praktisch ohne Informationsgehalt ist, da er lediglich besagt, daß fast alle Funktionen (wie bei Funktionssammlungen üblich) zu jedem Zeitpunkt aufrufbar sind. Betrachtet man hingegen den ersten *Life-Cycle*, der ein typisches Anwendungsbeispiel vorstellt, ergibt sich durchaus so etwas wie eine Halbordnung auf den System-Operationen. Geht man anhand dieser Halbordnung vor, werden die zuerst benötigten Methoden auch zuerst implementiert. Zu jeder neu angeforderten Methode werden alle von ihr verwendeten Unterfunktionen implementiert, und zwar zunächst so, daß ein minimal erforderlicher Rahmen gegeben ist. Dieser Ansatz entspräche einem *Top-Down*-Vorgehen, das in ähnlicher Form auch in [Som96, S. 453ff] beschrieben wird. Im Gegensatz dazu stünde ein *Bottom-Up*-Ansatz, der mit der Implementierung der Basis-Module beginnt. Meiner Ansicht nach war es zur Vereinfachung der Testphase für das vorliegende Projekt allerdings vielversprechender, i.w. dem *Top-Down*-Verfahren zu folgen. Ich habe bereits in Abschnitt 3.3 ein Verfahren vorgeschlagen, mit dem sich eine derartige Implementierungs- und Test-Reihenfolge auf einfache Art und Weise aus den *Visibility-Graphs* gewinnen läßt.

Somit begann ich mit der Implementierung der wichtigsten (d.h. zuerst benötigten) System-Operationen und programmierte die von diesen benötigten Unterfunktionen, dann die von diesen aufgerufenen Methoden, und so weiter. Allerdings sind viele Klassen des Entwurfes so klein und trivial, daß ich deren vollständige Implementierung als erstes vorziehen konnte. Dies waren z.B. die Klassen VarVector, die Value-Klassen, sowie die PreProb-Klassen. Da die *Variablen* die unbedingte Grundlage jeder Wissensbasis darstellen, setzte ich dann

die Funktionen zur Verwaltung der Variablenmenge um. Daraufhin entwickelte ich die (zahlreichen) Klassen zur Darstellung und Verwaltung von Regeln. Anschließend folgten die Methoden und Klassen zum Aufbau der LEG-Struktur. Abschließend implementierte ich den Iterationsprozeß und die Methoden zur Beantwortung von Anfragen. Diese Reihenfolge entspricht tatsächlich dem groben *Life-Cycle* aus Abschnitt 2.4.1.

Die Implementierung der einzelnen Funktionsrümpfe verlief zu großen Teilen recht gradlinig. Meist begann ich mit den *System Operation Models* der jeweiligen Methode und ergänzte die darin genannten Funktionsaufrufe um Kontrollstrukturen wie Schleifen und Bedingungen, sowie Variablendeklarationen. [Col94, S. 112ff] beschreibt ein Verfahren zur Umwandlung der Entwurfsmodelle in ausführbaren Quellcode. Abbildung 22 gibt ein (nicht untypisches) Beispiel für ein Vorgehen gemäß diesem Verfahren.

Ausgehend von den *Operation Schemata* aus dem *System Operation Model* der *Analysephase* werden zunächst die Vorbedingungen der Funktion geprüft und ggfs. mit einer Exception gemeldet. Auch andere Fehleingaben (hier: varIndex könnte einen ungültigen Wert haben) lösen KBE-Exceptions aus, die (wie im Funktionskopf hinter "**throws**" deklariert) von der Funktion gemeldet werden können.

Sind die Vorbedingungen bis dorthin erfüllt, kann der *Object Interaction Graph* der Methode direkt umgesetzt werden: Pfeile werden durch Funktionsaufrufe und Mehrfachausführungen (im Graphen: der Stern) durch for-Schleifen ersetzt.

Die neu eingeführten Funktionen checkNot... und invalidateGraphs sind private Hilfs-Methoden der Klasse KBase und haben nur für diese Klasse relevante Wirkung.

(JAVA-Schlüsselwörter sind **fett** gedruckt, Aufrufe von Methoden aus dem Entwurf unterstrichen).

```
public void deleteVar(int varIndex) throws KBE
{
    // Check the preconditions...
    checkNotIterating();
    checkNotEvidence();

    // Get the variable with the specified index...
    Variable var = varList.getVar(varIndex);

    // Delete the rules with an appearance of the Variable...
    Vector rules = var.getRules(false);
    for(int i = rules.size() - 1 ; i >= 0; i--) {
        Rule rule = (Rule) rules.elementAt(i);
        int ruleIndex = rule.getIndex();
        deleteRule(ruleIndex);
    }

    // Adjust the LEGForest to the change...
    legForest.deleteVar(var);

    // Delete the Variable from the varList...
    varList.deleteVar(varIndex);

    // Establish new links...
    legForest.linkVarsAndRules(varList, ruleList);

    // Invalidate the dependencies graphs...
    invalidateGraphs();
}
```

Abbildung 22: Umsetzung der System-Operation KBase.deleteVar in JAVA

Nicht immer verlief die Implementierung der Methoden so einfach und direkt, wie im genannten Beispiel. Meinen Beobachtungen zufolge versagt diese ideale von FUSION vorgeschlagene Methodik in folgenden Fällen:

- Hat eine Funktion hauptsächlich klassen-interne Wirkung, so enthalten die *Object Interaction Graphs* praktisch keine Informationen (da dort nur der Nachrichtenfluß *zwischen* den Objekten dargestellt wird und weder der Aufruf lokaler Funktionen noch die internen Kontrollstrukturen abgebildet werden). Man müßte die Graphen um sehr umfangreichen Text anreichern, um die zur Implementierung erforderlichen Details anzugeben.
- Oft ist der Entwurf nicht genau genug, verfügt also nicht über die zur direkten Implementierung ausreichende Tiefe. Zwar mag eine solche Detailtreue bei kleineren Projekten erreicht werden können, im vorliegenden Projekt wäre eine weitergehende Präzisierung allerdings *zu* komplex (und auch nicht sonderlich sinnvoll) gewesen.
- Die Aussagekraft der *Object Interaction Graphs* ist oft zu gering, um komplizierte Zusammenhänge darzustellen. Hierzu zählen u.a. Funktionsaufrufe an überladene Klassenfunktionen (die je nach Unterklasse unterschiedliche interne Wirkung haben können). Versucht man, eine hohe Detailtreue in die Graphen einzubringen, werden diese hoffnungslos unübersichtlich.

In SPIRIT war vor allem der erstgenannte Punkt von Bedeutung. Die Algorithmensammlung enthält teilweise sehr komplexe Verfahren, die am besten direkt mit Pseudo-Code (oder gleich Quelltext) dargestellt werden. In den meisten Fällen ließ sich mit dem Implementierungsverfahren nach FUSION aber sehr gut arbeiten. Immerhin benötigte ich für die Implementierung einer ersten, vollständigen Version von SPIRIT mit fast 10000 Zeilen JAVA-Code nur zwei Wochen (zugegebenermaßen mit 15 Stunden Arbeit pro Tag). Diese erste Version enthielt natürlich noch einige Fehler und ließ bzgl. ihrer Effizienz noch einige Wünsche übrig, war aber immerhin schon vollständig und konnte die wichtigsten Aufgaben erfüllen.

Während dieser intensiven Wochen deckte ich auch kleinere Mängel im Entwurf auf, die ich dort nachbesserte. Ich habe mir anschließend erlaubt, die fertige Version mit dem ursprünglichen, theoretischen Entwurf zu vergleichen. Dabei stellte ich folgende Unterschiede fest:

- 16 Methoden waren ursprünglich anders geplant. Größtenteils habe ich während der Implementierung Verbesserungen in der Vererbungshierarchie gefunden, also einige Funktionen von Ober- in Unterklassen verlagert oder umgekehrt.
- Fünf Methoden hatten andere Argumente, bzw. Rückgabewerte. Dies lag daran, daß ich die Sichtbarkeit von Klassenattributen nicht ausreichend vorhergesehen hatte und manche Attribute nicht dort zugänglich waren, wo sie später benötigt wurden.
- Vier Attribute waren unterschiedlich, und zwar vor allem in der Klasse IterationManager, die ich im Entwurf allerdings auch nicht weiter betrachtet, sondern der Implementierung überlassen hatte.
- Eine Klasse (MixedVarGraph) wurde neu eingeführt. Ich stellte bei ersten Testläufen fest, daß die ursprünglich geplante Umsetzung der graph-Funktionen zu langsam war. Dadurch mußte ich die Graphen mit einer eigenen Klasse verwalten und im Hintergrund speichern (vgl. Abschnitt 3.1.7).

Diese Änderungen habe ich anschließend in dem Entwurfsdokument nachgebessert, um Implementierung und Entwurf in Einklang zu bringen. Diese Nachbesserungen verliefen i.a. problemlos und schnell. Daß mein ursprünglicher Entwurf noch obige "Fehler" enthielt führe ich auf die Komplexität des Projektes zurück. Meiner Ansicht nach war der Entwurf ab einer bestimmten Tiefe nicht mehr sinnvoll, da viele Verbesserungen und kleinere Mängel am besten im Quelltext sichtbar sind. Es wäre sehr schwierig gewesen, hundertprozentig richtige *Class Descriptions* zu entwerfen. Es erwies sich hierbei als reine Ermessensfrage, zu welchem Zeitpunkt man mit dem Entwurf aufhört und die Implementierung beginnt. Möglicherweise wäre bei SPIRIT auch eine zeitliche Überlappung der beiden Phasen sinnvoll gewesen, um aus der Implementierung ein *Feedback* für die Güte des Entwurfs zu erhalten. Die Tatsache, daß nur sehr geringe, lokale Änderungen am Entwurf vorgenommen werden mußten, spricht meiner Meinung nach für die Leistungsfähigkeit von FUSION als objektorientierte Entwicklungsmethode.

4.3 Test

Nach und während der Implementierung des Systemkernes wurde die Software auf Korrektheit, Robustheit und Fehlertoleranz geprüft. Dieser Abschnitt beschreibt die Vorgehensweise beim Testen, die dazu eingesetzten Zusatzprogramme (Testtreiber) und nennt die wichtigsten Ergebnisse der Testphase.

4.3.1 Vorgehensweise in der Testphase

Zum Test der Korrektheit von SPIRIT boten sich mehrere Möglichkeiten an. Die Primärquelle zu FUSION ([Col94]) macht über geeignete Testverfahren kaum Aussagen, weil der Schwerpunkt dort auf Analyse und Entwurf liegt. Da die Implementierung ziemlich direkt aus dem Entwurf hervorgeht, sollte sie folglich kaum als Fehlerquelle in Erscheinung treten und Programmfehler sollten bereits in den früheren Entwicklungsphasen vermieden worden sein. Dies mag für einfache Projekte (wie Datenbankanwendungen) auch zutreffen; im vorliegenden Projekt gibt es aber eine fast unüberschaubare Anzahl von Anwendungsmöglichkeiten, so daß eine aufwendige Testphase notwendig war. Vor allem handelt es sich bei SPIRIT i.w. um eine Funktionssammlung, deren Kommandos von *Shells* in beliebiger Reihenfolge aufgerufen werden können, und somit auf eine Fülle von Fehlbedienungen vorbereitet sein müssen. Dies bedeutet auch, daß ein kompletter Test mit dem Ziel einer garantierten Fehlerfreiheit unmöglich ist und die Testergebnisse nicht überschätzt werden dürfen. Zumindest konnte ermittelt werden, ob die implementierten Algorithmen eine ausreichende Effizienz aufwiesen, und wo ggfs. optimiert werden mußte. Die neben Tests in der Qualitätssicherung von Software häufig eingesetzten *Reviews*, bei denen Entwurf und Code von anderen Systementwicklern überprüft werden, waren für das vorliegende Projekt übrigens nicht möglich. Dies lag einfach daran, daß ich als einziger Programmierer an SPIRIT mitwirkte.

Üblicherweise (vgl. [Som96, *Part Five*, insbes. Kapitel 22.3: *"Testing strategies"*]) werden u.a. zwei Strategien zum Testen von Software eingesetzt. Beim *Top-Down-Testing* wird das System zunächst von außen betrachtet und dessen Ausgaben bei gegebenen Eingaben untersucht. Dabei wird von den höheren Systemebenen in die tieferen, detaillreichen Komponenten hinabgestiegen, bis schließlich auf unterster Ebene keine Fehler mehr erkannt werden. Das alternative *Bottom-Up-Testing* beginnt auf eben diesen unteren Ebenen und testet die komplexen Zusammenhänge erst, wenn die Subkomponenten keine Fehler mehr enthalten. Im Projekt SPIRIT war eine Untersuchung einzelner Subkomponenten sehr schwierig. Dies lag insbes. daran, daß SPIRIT mit großen Zahlentabellen (den LEGs) rechnet, deren Inhalt durch sehr komplizierte Berechnungsalgorithmen verändert wird, deren Ergebnisse kaum lokal überprüfbar sind. Die Richtigkeit der LEG-Tabellen ist gleichzeitig

die wichtigste Eigenschaft von SPIRIT und die meisten anderen Komponenten (wie Regel- und Variablenverwaltung) stellen lediglich den Rahmen zur Ermittlung dieser Tabellen dar. Somit war *Bottom-Up-Testing* nicht sinnvoll und *Top-Down-Testing* i.w. auf die obersten Systemebenen beschränkt.

Es bot sich somit an, das System *als Ganzes* zu betrachten und (unter Kenntnis der internen Strukturen) anhand sinnvoller Testfälle zu prüfen. Dies hatte insbes. den Vorteil, daß die Testergebnisse mit den entsprechenden Resultaten der Vorgängerversionen verglichen werden konnten, die sich im praktischen Einsatz offensichtlich bewährt hatten. SPIRIT konnte somit als (scheinbar) fehlerfrei angesehen werden, wenn dieselben Operationen auf Wissensbasen mit äquivalenten Ausgangszuständen dieselben Ausgaben und Zustandsänderungen bewirkten wie in der Vorgängerversion. Unterstützt wurden diese Untersuchungen des Laufzeitverhaltens durch einige temporäre Erweiterungen des Codes:

- Sicherheitsüberprüfung der Vorbedingungen von einigen Funktionen, also der Korrektheit von Argumenten und Klassenattributen. Soweit diese Überprüfungen die Performance des Systems nicht beeinträchtigten, wurden sie in der Endfassung belassen.
- Einbau temporärer Funktionen für komplexe Test, deren Durchführung manuell und mit Taschenrechner unmöglich ist (z.B. eine Funktion, die den gesamten LEG-Wald durchläuft und alle benachbarten LEGs und Separatoren auf gegenseitige Konsistenz (=Gleichheit der Randsummen) überprüft).
- Vorübergehende globale Sichtbarkeit privater interner Klassenattribute zur Abfrage von Systemzuständen.

4.3.2 Entwicklung von Testtreibern

Ein Testtreiber ist ein Programm, das üblicherweise nur während der Programmentwicklung eingesetzt wird und eine simulierte Umgebung zum Test von Komponenten darstellt. Speziell für die SPIRIT-Klassenbibliothek sollte ein Testtreiber

- den Test *aller* öffentlich zugänglichen Funktionen der Bibliothek in *allen* möglichen Variationen, also in jedem Zustand der Wissensbasis und in jeder Reihenfolge, zulassen und nicht mehr "können" als andere Shells,
- die Funktionsaufrufe oder -ergebnisse nicht verfälschen, sondern einen *direkten* Zusammenhang zwischen Eingabe, Funktionsaufruf, Ausgabe und Fehlermeldung erkennen lassen,
- es ermöglichen, die Tests *effizient* zu gestalten (also möglichst eine interaktive Benutzerführung, aber auch automatisierte Tests mit Stapelverarbeitung unterstützen) und einigermaßen benutzerfreundlich sein,
- *einfach zu implementieren* sein und nicht unnötig Zeit kosten (d.h. eine einfache Benutzeroberfläche haben), und
- schrittweise an den Fortschritt der Programmierung des Systemkerns *anpaßbar* sein.

Bereits vor der Umsetzung des Entwurfes in eine ausführbare JAVA-Klassenbibliothek begann ich mit der Erstellung des Testtreibers SPECTER, um die jeweils neu entwickelten Methoden möglichst direkt auf Korrektheit überprüfen zu können. Gleichzeitig wollte ich noch etwas Programmiererfahrung in JAVA sammeln, da ich in dieser Sprache zuvor noch nicht programmiert hatte und nicht etwa SPIRIT selbst für solche Versuche mißbrauchen wollte.

SPECTER ist eine einfache, in JAVA geschriebene Kommandozeilen-Oberfläche zur Bearbeitung einer SPIRIT-Wissensbasis (der Klasse KBase). Der Benutzer kann vordefinierte Befehle und Parameter eingeben, die den direkten Aufruf einzelner SPIRIT-Funktionen bewirken. Die Ergebnisse dieser Funktionen werden, ebenso wie eventuelle Fehlermeldungen, in einem Log-Fenster protokolliert, das zur Rückverfolgung von Fehlerursachen verwendet werden kann. Über spezielle Status-Befehle kann der aktuelle Zustand der Wissensbasis angezeigt werden (u.a. die Liste der Variablen und Regeln, sowie die Struktur der LEGs). Außerdem lassen sich ganze Befehlssequenzen über Stapelverarbeitungsdateien (*Batch-Files*) automatisieren. Zur Messung der Effizienz des Codes ist es möglich, die zur Ausführung eines Kommandos benötigte Dauer (in Millisekunden) zu messen. Dadurch lassen sich verschiedene Varianten von Algorithmen bzgl. ihrer Effizienz miteinander vergleichen. Auch läßt sich so feststellen, welche Operationen unter welchen Rahmenbedingungen den Programmablauf merklich verzögern und somit optimiert werden sollten. Da einige Funktionen nur sehr kurze oder unregelmäßige Laufzeiten haben, unterstützt SPECTER auch die mehrfache Ausführung von Kommandos zur Messung der akkumulierten Laufzeit.

Ein sehr ähnlicher Ansatz wurde (mit dem Programm HOLYSPIRIT) bereits bei den Vorgängerversionen sehr erfolgreich erprobt. Damals machten wir den Fehler, direkt mit einer komplexen Oberfläche anzufangen, die für die Tests der einzelnen Systemoperationen viel zu umständlich und unübersichtlich war. Bei derartigen Oberflächen ist es schwer, den direkten Zusammenhang zwischen Benutzereingabe und Wirkung zu erkennen, da viele Operationen versteckt im Hintergrund ablaufen. Vor allem können dort keine Informationen ausgegeben werden, die eigentlich nur für interne Tests relevant sind. In einer Kommandozeilen-Version lassen sich demgegenüber beliebige Informationen ausgeben, ohne daß auf Layout und Gestaltung geachtet werden muß.

Die Entwicklung von SPECTER (und die damit verbundene Einarbeitung in JAVA) dauerte ca. ein Woche. Ich bereitete für jede System-Operation aus der *Design*-Phase ein Kommando vor, das i.w. im Aufruf einer KBase-Methode resultiert. Eine Auflistung dieser Befehle befindet sich in Anhang E. Hierbei ist zu sehen, daß die meisten Kommandos ähnliche Parameter erwarten, wie die induzierten System-Operationen (z.B. entspricht der SPECTER-Befehl addval genau der Methode KBase.addValue und erwartet dieselben Argumente). Die Implementierung der Befehle war zunächst leer, und wurde parallel zur Entstehung der SPIRIT-Bibliothek schrittweise mit Inhalt gefüllt.

Neben den positiven Erfahrungen und Erkenntnissen mit SPECTER sei aber auch ein wesentlicher Nachteil dieses Testtreibers erwähnt. Er ist praktisch nur für Leute zu gebrauchen, die sich recht gut mit dem zugrundeliegenden System auskennen. Im Projekt SPIRIT sollten viele Tests aber in der Praxis durchgeführt werden, insbes. von Menschen, die später mit einer auf SPIRIT basierenden *Shell* arbeiten sollen. Ein weiterer Nachteil von SPECTER ist außerdem, daß mit ihm i.w. nur auf die äußere Schale des SPIRIT-Systems zugegriffen werden kann, also auf die exportierten System-Operationen des System-Kernes. Somit sind keine Test einzelner Systemkomponenten und Klassen der Bibliothek möglich.

Nachdem ich eine erste Version von SPECTER fertiggestellt hatte, begann ich mit der Implementierung des eigentlichen Systemkerns. Als dann alle System-Operationen mit Hilfe von SPECTER und anderen Testverfahren auf einem stabilen Niveau implementiert waren, entwarf ich einen weiteren, wesentlich komfortableren Testtreiber, der auch von Nicht-Fachleuten verwendet werden können sollte. Es war also erforderlich, von der Befehls-Ebene zu einer grafischen Benutzerführung zu gelangen. Dazu entwarf ich das JAVA-*Applet* SPIRITSHELL, das durchaus als brauchbare Shell zur Bearbeitung von Wissensbasen angesehen werden kann. Zwar verfügt diese Shell nicht über die volle von SPIRIT bereitgestellte Funktionalität, aber sie ist dafür recht komfortabel von "Laien" benutzbar. Dadurch erst wurden richtige Praxistests möglich, die eine echte Bewährungsprobe für den Systemkern waren.

SPIRITSHELL (sh. auch Abbildung 23) ermöglicht die Bearbeitung und Befragung einer einzelnen Wissensbasis. Sie besteht aus verschiedenen Fenstern, die teilweise permanent sichtbar und teilweise zusätzlich aufklappbar sind:

- *Variablen-Fenster* zur Ansicht und Bearbeitung der aktuellen Variablenmenge, den aktuellen Wahrscheinlichkeiten der Variablenwerte und den Erwartungswerten, außerdem zur Zuweisung von Evidenz an ausgewählte Variablen,
- *Regeln-Fenster* zur Ansicht und Bearbeitung der aktuellen Regelmenge, den aktuellen und vorgegebenen Wahrscheinlichkeiten der Regeln, sowie weiteren Informationen zu den Regeln,
- *Graphen-Fenster* zur Visualisierung der Abhängigkeiten zwischen den Variablen,
- *Iterations-Fenster* zur Durchführung und Überwachung eines Iterationsprozesses zum Lernen der Regeln, und
- *Interface-Fenster* zur Ansicht und Änderung der Wissensbasis auf SPIRIT-Datei-Ebene.

Über diese Unterfenster kann der Benutzer sowohl strukturelle Änderungen an der Wissensbasis vornehmen, also Variablen und Regeln eingeben, als auch Anfragen stellen. Somit stellt SPIRITSHELL fast alle Funktionen bereit, die von SPIRIT unterstützt werden.

Eine erste brauchbare Version dieser Shell war innerhalb von zwei Wochen fertig. Ich legte keinen übermäßigen Wert auf optische Eleganz, sondern setzte den Schwerpunkt auf eine korrekte Funktionsweise. SPIRITSHELL ging zur ersten Begutachtung der neuen SPIRIT-Version an den auftraggebenden Lehrstuhl, wo sie umfangreichen Tests unterzogen wurde. Später verbesserte ich die SPIRITSHELL zu einer (auch optisch) ansprechenden graphischen Oberfläche.

Glücklicherweise lag aus den Vorgängerversionen noch ein großer Bestand an Wissensbasen (Anwendungsbeispielen aus der Praxis, sowie künstlich generierte Spezialfälle) vor, die als Fallbeispiele zur Verfügung standen. Ausgehend von diesen Beispielen wurden Laufversuche und systematische Tests durchgeführt und mit den Vorgängerversionen verglichen. Dabei waren "extreme" Wissensbasen als Grenzfälle besonders interessant, z.B. Wissensbasen mit besonders vielen Regeln oder Variablen und solche, die eine ungünstige LEG-Struktur mit hohem Speicherbedarf oder einen sehr langsam konvergierenden Iterationsprozeß erfordern.

Abbildung 23 zeigt einen typischen Bildschirmausdruck von SPIRITSHELL (hier in eine HTML-Seite eingebettet, direkt über das Internet gestartet). Eine Ansicht von SPIRITSHELL als *stand-alone-Application* befindet sich als Abbildung 25 auf Seite 72.

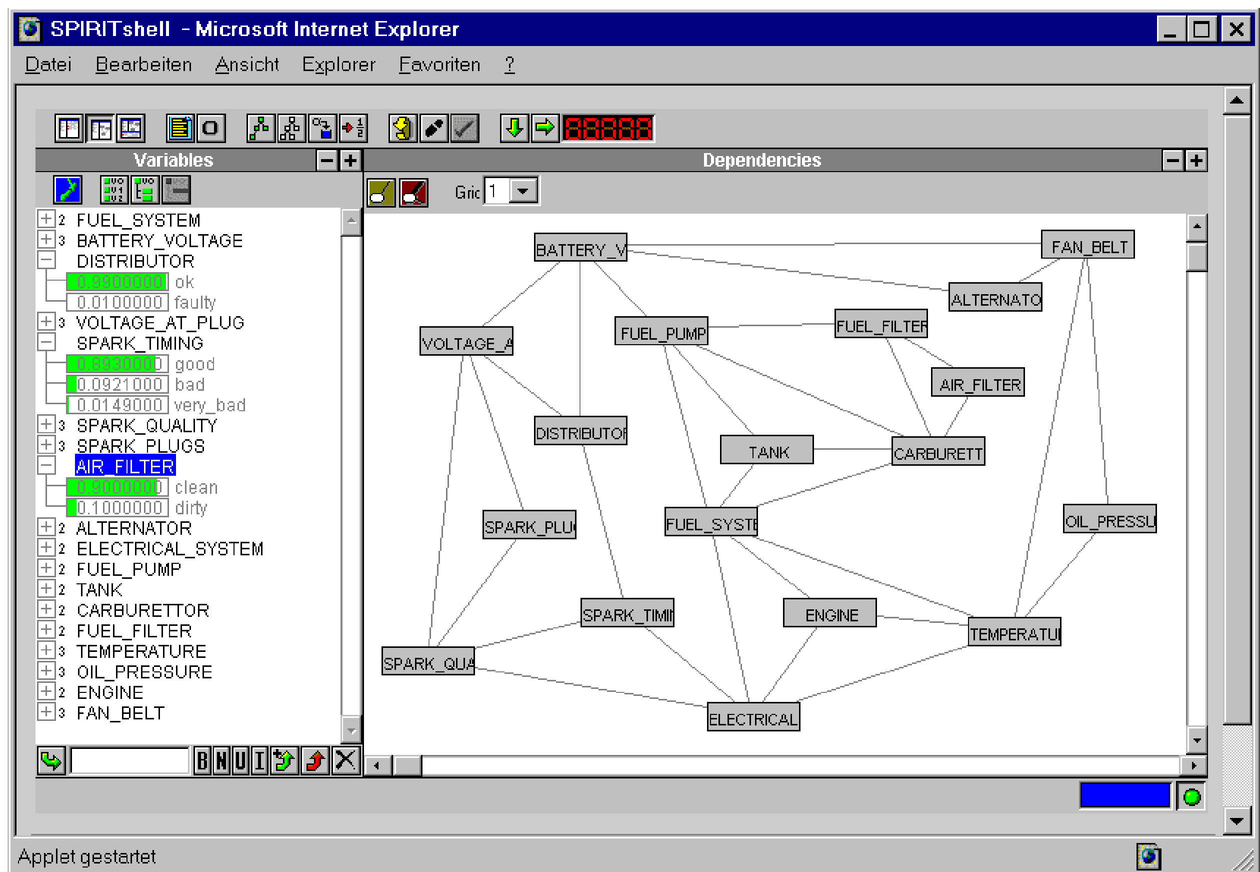


Abbildung 23: Eine Ansicht der "Weihnachtsversion" von SPIRITSHELL

4.3.3 Ergebnisse der Testphase

Insgesamt dauerte die Testphase recht lange und wird wohl auch in den nächsten Jahren fortgeführt werden, wenn sich SPIRIT in der Praxis bewähren soll. Neben den von Lehrstuhlmitarbeitern durchgeführten praxisnahen Tests, die natürlich größtenteils unsystematisch waren, führte ich Tests aller Systemfunktionen durch und erweiterte SPECTER um kleine Testprozeduren für besondere Belastungstests. Hierbei erwies sich der Code als fast fehlerfrei, da ich die üblichen "Flüchtigkeitsfehler" bereits durch Probeläufe während der Implementierungsphase ausgemerzt hatte.

Dennoch wurden während der ersten Wochen kleinere Fehler gefunden und behoben. Hierzu zählte z.B. die Entdeckung, daß der an der Vorgängerversion angelehnte Algorithmus zur Erzeugung der LEG-Struktur fehlerhaft umgesetzt war. Während der Test-Phase stellte sich auch heraus, daß SPIRIT bei sehr großen Wissensbasen (LEGs mit etwa 32000 Zuständen) Speicherprobleme bekommen kann. Daher wurde die Klasse KBase um eine "Modus"-Variable (sysMode) erweitert, mit der die Nutzung der speicheraufwendigen IndexLists deaktiviert werden kann. Da dies als reines Implementierungsdetail anzusehen ist, wurde hier der Entwurf nicht mehr nachträglich angepaßt.

Durch die Tests wurde auch festgestellt, daß die relative Entropie als Abbruchkriterium des Iterationsprozesses aufgrund der Rechenungenauigkeit von double-Werten (ein Fließkomma-Datentyp mit einer Breite von immerhin zehn Bytes) ungeeignet war, da durch die Logarithmierung in der Entropieformel sehr kleine Zahlen entstehen. Daher wurde die Abweichung zwischen vorgegebener und tatsächlicher Wahrscheinlichkeit einer Regel als neues Abbruchkriterium herangezogen. Dies erhöhte die maximale Genauigkeit der Berechnungen bis auf 10e-15.

4.4 Dokumentation für Anwendungsprogrammierer

SPIRIT ist eine Klassenbibliothek, die zur Programmierung von wissensbasierten Anwendungsprogrammen verwendet werden kann. Somit war es ein selbstverständlicher Bestandteil der *Requirements*, Anwendungsprogrammierern eine größtmögliche Unterstützung durch Anleitungen und Referenzhandbücher zu geben. Ein Programmierer sollte ohne Kenntnis der internen Abläufe die Funktionen verwenden können, aber wo nötig auch über die Auswirkungen der Operationen auf *Performance*, etc. unterrichtet werden.

Diese Aufgaben übernimmt ein *Online-Handbuch*, das über jeden beliebigen *WWW-Browser* angesehen und natürlich auch ausgedruckt werden kann. Das Handbuch besteht aus zwei Teilen:

- Das Programmierhandbuch (*Programming Guide*) führt in die erforderliche Theorie ein, beschreibt Einsatzgebiete und -techniken von SPIRIT und gibt eine Übersicht über die Klassenbibliothek.
- Das Referenzhandbuch (*Reference Guide*) dient als Nachschlagewerk für die von SPIRIT bereitgestellten Klassen, Methoden und Attribute.

Das Programmierhandbuch wurde, bzw. wird in Zusammenarbeit mit dem Lehrstuhl erstellt, während das Referenzhandbuch praktisch nebenbei entstand. JAVA stellt hierzu glücklicherweise das Zusatzprogramm JAVADOC bereit, das JAVA-Quelltexte automatisch in Dokumentationen umwandeln kann. Diese Dokumentationen werden im HTML-Format erzeugt (HTML ist die wichtigste Skriptsprache für WWW-Seiten) und enthalten alle nach außen sichtbaren Klassen, Methoden und Attribute. Der Programmierer der zu dokumentierenden Module kann zu jedem dieser Elemente zusätzliche beschreibende Informationen angeben, die er in Form besonders gekennzeichneter Kommentare direkt in den Quelltext einbaut. Über jede Methode schreibt er einen Kommentar-Block, dessen Inhalt durch JAVADOC in die Dokumentation aufgenommen wird. Damit JAVADOC die Kommentare richtig zuordnen kann, muß der Programmierer jeweils angeben, ob es sich um Eingabe-Parameter, Rückgabewerte, Fehlermeldungen oder auch Querverweise handelt. Da solche Kommentare ohnehin in jedes gute Programm gehören, ist es überhaupt kein Aufwand, die Quelltexte für JAVADOC vorzubereiten.

In SPIRIT wurde das Referenzhandbuch somit parallel zur Implementierung entwickelt und stets auf dem aktuellen Stand gehalten. Jede exportierte Methode der Bibliothek erhielt eine kurze textuelle Beschreibung, Informationen über Ein- und Ausgabeparameter und eine Liste der möglicherweise ausgelösten KBE-*Exceptions*. Da das Handbuch in *Hypertext*-Form vorliegt, kann es mit einem *WWW-Browser* bequem neben der Programmierung gelesen und mit der Maus bedient werden. Abbildung 24 zeigt zwei Ausschnitte aus diesem Referenzhandbuch.

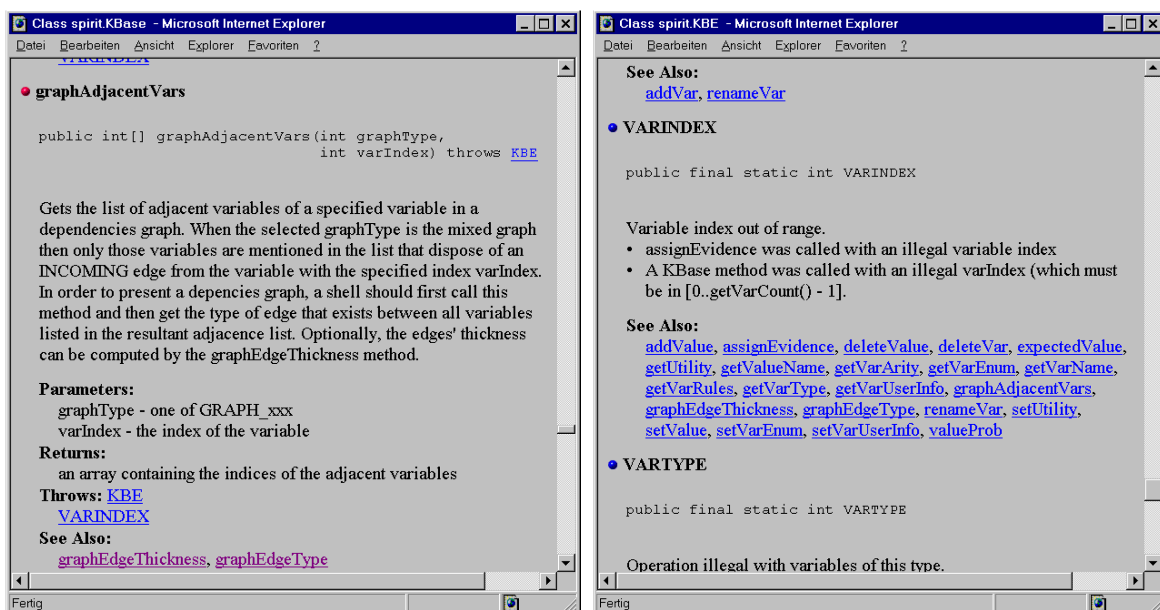


Abbildung 24: Ausschnitte aus dem SPIRIT-Referenzhandbuch, gestartet mit einem WWW-Browser

5 Fazit

Dieses abschließende Kapitel faßt die Ergebnisse der Arbeit zusammen. Zunächst beschreibe ich in Abschnitt 5.1 die positiven und negativen Erfahrungen, die ich während der Analyse- und Design-Phase von FUSION mit dem vorliegenden Projekt machte. Die aus der anschließenden Implementierungsphase gewonnenen Eindrücke wurden bereits im vorangegangenen Kapitel geschildert. Abschnitt 5.2 versucht eine Bewertung der neuen Version von SPIRIT gegenüber den Anforderungen und den Vorgängerversionen. Abschließend gibt Abschnitt 5.3 einen kurzen Ausblick auf die Zukunft von FUSION, JAVA und SPIRIT.

5.1 Erfahrungen mit FUSION

In diesem Abschnitt beschreibe ich meine mit FUSION gemachten Erfahrungen. Dabei vergleiche ich diese mit anderen in der Literatur auftauchenden Erfahrungsberichten, die FUSION kritisch untersuchen. Schließlich versuche ich, die Eignung dieser Methodik für ein Projekt wie SPIRIT zu beurteilen.

Das einleitende Kapitel von [Col94] nennt in erster Linie folgende positive Eigenschaften von FUSION (S. 8f):

- *FUSION sei ein systematischer Prozeß zur Software-Entwicklung, der einzelne Phasen mit klar abgegrenzten Aufgabenstellungen vorschlägt und den Entwickler dadurch durch das Projekt führt.* Meiner Erfahrung nach ist dies richtig, da ich recht zügig die einzelnen Teilschritte abarbeiten konnte. Mir war allerdings nicht immer klar, wann ich eine Phase abschließen und mit der nächsten fortfahren sollte. Dies war gerade bei den *Object Interaction Graphs* aus der Entwurfsphase der Fall, die fast beliebig tief und komplex sein können und es unklar ist, ob und wann eine ausreichende Genauigkeit erreicht ist. Auch der Schritt von der Analyse- zur Entwurfsphase war sehr groß, weil in der *Analysephase* nicht auf Fragestellungen der internen Umsetzung und der späteren *Performance* eingegangen wird (ähnliche Erfahrungen werden in [Mal95, S. 316 oben] geschildert). Ich habe daher den Zwischenschritt aus Abschnitt 3.1 eingebaut, der sowohl Implementierungsaspekte als auch Fragestellungen des Entwurfs frühzeitig beleuchtete und somit die Grundlage für einen ausreichend präzisen Entwurf darstellte.
- *FUSION sei eine durchgängige Methode, deren Schritte aufeinander aufbauen und direkt von der Analyse bis zur Implementierung führen.* Dies kann ich i.w. bestätigen. FUSION gibt allerdings keine Hilfestellung zur Erstellung eines Anforderungsdokumentes und macht auch wenige Angaben, wie man von den *Requirements* zu den ersten Analyse-Modellen gelangt (über Lösungsvorschläge hierzu sh. [Mal95, Kapitel 8]). Dieser Schritt zu den Analyse-Modellen ist meiner Ansicht nach einer der schwierigsten, da es hierbei keine formale Unterstützung oder Methoden gibt, sondern Intuition und Erfahrung des Entwicklers entscheidend sind. Nach Fertigstellung von *Object-* und *Operation Model* bauen aber alle Schritte aufeinander auf und vermeiden dadurch die Entstehung von Inkonsistenzen. Im Projekt SPIRIT habe ich verstärkt darauf geachtet, daß ich bereits in der Entwurfsphase Aspekte der späteren Implementierung (wie organisatorische Parameter und interne Optimierungen) mitberücksichtigt habe, was gemäß FUSION nicht unbedingt gemacht werden sollte. Ich bin der Meinung, daß erst dadurch der Unterschied zwischen Entwurfsdokument und Quelltext so gering wurde, der Code also recht gut anhand des Entwurfes erstellt und verstanden werden kann. Außerdem wurde der für die Implementierung erforderliche Zeitraum stark verkürzt.
- *Der Entwicklungsprozeß gemäß FUSION sei (z.B. seitens des Projekt-Managements) gut zu überblicken, liefere klar erkennbare Zwischenergebnisse und ermögliche die Prüfung der Konsistenz zwischen den einzelnen Modellen.* Meinen Beobachtungen zufolge legt FUSION großen Wert auf eine Überprüfbarkeit der Modelle und stellt die hierzu erforderlichen Mechanismen (wie in den Abschnitten 3.7, 4.2.4, 4.3.4, 4.4.3 und 4.5.3 aus [Col94] erläutert) nicht nur bereit, sondern hängt Konsistenzprüfungen als wichtige Teilschritte ans Ende jedes Teilschrittes. Für ein Ein-Personen-Projekt wie SPIRIT halte ich dies für absolut ausreichend, habe allerdings im Konzept von FUSION kaum Unterstützung für verteilte Software-Entwicklung durch große Teams gefunden. Es wurden keine Vorschläge gemacht, wie ein Projekt zu parallelisieren sei (lediglich das *Data-Dictionary* wurde zur Kommunikation und Konsistenzprüfung vorgeschlagen). Da die einzelnen Modelle aufeinander aufbauen und eine zeitliche Überlappung der Teilschritte kritisch ist, wäre nur eine "horizontale" Aufspaltung der Zwischenmodelle denkbar. Dieser Mangel soll in der kommenden Generation von FUSION einem Pressebericht aus dem Hause HEWLETT-PACKARD zufolge besondere Beachtung gefunden haben.
- *FUSION verfüge über präzise, aber verständliche und leicht erlernbare Modelle.* Auch dieser Meinung schließe ich mich (ebenso wie z.B. [Mal95, Abschnitt 11.3]) an. Anhand der formalen Beschreibung in [Col94] und einigen Beispielen war ich leicht in der Lage, die vorgeschlagenen Modelle auf SPIRIT anzuwenden. Auch können die Modelle ein hohes Maß an Präzision ausdrücken, auch wenn ich aufgrund der

Komplexität meiner Aufgabenstellung an vielen Stellen absichtllich etwas unscharf blieb und diese Präzision selten ausreizte.

- *FUSION sei anpassbar und flexibel, d.h. nicht alle Schritte müssten in der vorgeschlagenen Form durchgeführt werden*. Bei dieser Aussage gehe ich sogar so weit, daß FUSION hauptsächlich als *Rahmen* anzusehen ist, der für jedes spezielle Projekt anders genutzt werden sollte. Ich habe mich zwar zumeist an die von FUSION vorgeschlagene Methodik gehalten, mir aber an vielen Stellen erlaubt, vom Standard abzuweichen und eigene Ergänzungen einzubauen. Insbesondere habe ich die Syntax einiger Modelle (z.B. in *Operation Model* und *Object Interaction Graphs*) aufgelockert und erweitert. Die Schritte *Life-Cycle-Model* und *Visibility-Graphs* ergaben im SPIRIT-Projekt nicht viel Sinn und hätten ausgelassen werden können. Das wichtige Zwischenkapitel 3.1 war im Standard nicht vorgesehen, fügte sich letztlich aber sehr gut in das Gesamtkonzept von FUSION ein.

Somit kann ich die *allgemeinen* Erwartungen und Ansprüche der Verfasser von FUSION grundsätzlich bestätigen. Die beiden folgenden Unterabschnitte gehen nun noch etwas genauer auf die Analyse- und die Entwurfsphase ein.

5.1.1 Erfahrungen aus der *Analysephase*

Wie schon erwähnt, gibt FUSION wenig Hilfestellung bei der Umsetzung der *Requirements* in die Modelle der *Analysephase*. Die Modellierung des *Object-Models* ist meiner Ansicht nach sehr intuitiv und mit allgemeinem Verständnis des Problembereiches gut durchführbar. Vorteilhaft ist auch, daß eine große Ähnlichkeit zwischen *Object-Model* und den bekannten *Entity-Relationship-Diagrammen* aus der Theorie der Datenbanken vorliegt, die jedem durchschnittlich ausgebildeten Informatiker geläufig sein sollten. Im Projekt SPIRIT erforderte die Modellierung der Objekte allerdings einige vorausschauende Planung, die durch meine Erfahrungen mit den Vorgängerversionen wesentlich unterstützt wurde.

Die Syntax der *Object-Models* enthält meiner Beobachtung nach leider einige Ungereimtheiten (vgl. [Mal95, S.289]), die allerdings nicht sonderlich schwerwiegend sind. So gibt es bei Relationen einige redundante Verwendungsmöglichkeiten von *Total-Markers* und Kardinalitäten, die dieselbe semantische Bedeutung haben, bzw. auf unterschiedliche Weisen interpretiert werden können. Das Sternchen (*) zur Anzeige von 1-zu-N-Beziehungen kann meines Erachtens immer durch ein Plus (+) ersetzt werden, weil die Teilnahme einer Objektinstanz an *Relationships* ohnehin optional ist.

FUSION legt mit *System-Operation-* und *Life-Cycle-Model* ein großes Gewicht auf die *Schnittstelle*, die das zu entwickelnde System nach außen bereitstellen soll. Dies wird unterstützt von der klaren Trennung des Systemkerns von seiner Umgebung mit dem *System-Object-Model.* Hierdurch ist FUSION besonders für Klassenbibliotheken wie SPIRIT geeignet. Zu jeder exportierten *Interface*-Operation werden Vor- und Nachbedingungen, Ein- und Ausgabe, sowie die induzierten Seiteneffekte definiert, wodurch ein sicheres Verhalten des Systems garantiert werden soll. Durch die Entwicklung dieser Modelle konnten im vorliegenden Projekt einige Inkonsistenzen in den Anforderungen aufgedeckt und nachgebessert werden. Insofern kommt der *Analysephase* eine für die Qualität des späteren Produktes entscheidende Rolle zu und erfordert daher besondere Weitsicht.

Bei der Definition der System-Operationen hatte ich mir schon früh erlaubt, eigene Erweiterungen zum Standard, z.B. ein **delete**-Schlüsselwort zum Löschen von Objekten, einzuführen. Denselben Gedanken fand ich später bei [Mal95, S.295], wo ebenfalls auf die fehlende Löschoperation hingewiesen und sogar auch ein **delete**-Befehl vorgeschlagen wird. Auf einen weiteren kleineren konzeptionellen Mangel weist [Mal95, S. 292] hin. Es ist (auch im vorliegenden Projekt SPIRIT) nicht klar definiert, in welchem Anfangszustand sich die Objekt-Modelle und die später darauf erwachsenen Objekte beim Systemstart befinden. Es fehlt ein *Constructor*, der den Objekten einen Initialzustand zuweist und die ersten Verbindungen aufbaut. Daher wird ein spezielles *Operation-Schema* vorgeschlagen, das die Systeminitialisierung übernehmen soll. Bei SPIRIT war dieser Mangel (zufälligerweise) nicht aufgefallen, weil der Anfangszustand von Objekten der Klasse KBase trivial ist (es gibt weder Variablen noch Regeln und somit auch keine *Relationships*).

Die *Life-Cycles* aus der Analyse waren in SPIRIT praktisch redundant. Dies lag daran, daß die Systemoperationen fast ohne Beschränkung aufgerufen werden können. Dafür können sie zur Festlegung von Vor- und Nachbedingungen der System-Operationen eingesetzt werden. Gleichzeitig enthält FUSION hierbei eine gewisse Redundanz, weil sowohl *Life-Cyles* als auch *Operation-Model* dasselbe Systemverhalten ausdrücken können: Beide beschreiben Vor- und Nachbedingungen. Ein weiterer Mangel von *Life-Cycles* und *Operation-Model* ist, daß sie Zustandsinformationen zur Überprüfung dieser Vorbedingungen voraussetzen. Im Projekt SPIRIT ist dies zum Beispiel die Abfrage, ob Evidenz zugewiesen wurde oder eine Iteration läuft. Diese Zustandsinformationen werden in der *Analysephase* allerdings nicht berücksichtigt, sondern stellen *interne* Stati dar, die erst später in das Modell eingebaut werden dürfen.

5.1.2 Erfahrungen aus der Entwurfsphase

Meine wichtigste Kritik an FUSION richtet sich gegen den (fehlenden) Schritt von der Analyse- zur Entwurfsphase. [Col94] als FUSION-Lehrbuch geht selbst nur mit zwei kurzen Sätzen darauf ein, daß das Klassenmodell aus der *Analysephase* ggfs. erweitert werden muß: *"In addition to the objects listed explicitly in the [operation] schema, there may be other objects involved [in the object interaction graphs]. For example, new objects may be introduced to represent abstractions of computational mechanisms not identified in the analysis model."* ([Col94, S. 69]). Es wird dort kein systematisches Vorgehen zur Definition dieser neuen Klassen angegeben, sondern lediglich auf Beispiele verwiesen. Wie schon einige Male erwähnt, habe ich einen sehr umfangreichen und wichtigen Zwischenschritt eingeführt, um die folgenden Schritte der Design-Phase zu ermöglichen (Abschnitt 3.1). Dabei entstanden nicht nur die entscheidenden Konzepte für die spätere Umsetzung der Anforderungen, sondern auch zahlreiche neue Ideen, die sonst vermutlich erst während der Implementierung gekommen wären. Schließlich wäre der Unterschied von Entwurfsdokument und Programmcode sehr groß geworden. Auch [Mal95] weist auf Seite 316 auf diese Lücke zwischen Analyse und Design hin: *"In complex systems there may be significant transformations between analysis models and design."* .

Positiv überrascht war ich bei der Erweiterung des Analyse-Modells davon, wie gut sich das *Object-Model* ausbauen ließ. Ausgehend vom groben Diagramm der Klasse KBase (Abbildung 3, Seite 14) konnte ich schrittweise Verfeinerungen anbringen, ohne die alten Diagramme ungültig zu machen. Betrachtet man beispielsweise das Diagramm der Expression-Klassen (Abbildung 10, Seite 26), so sieht man, wie die ursprüngliche Relation appears dort nicht mehr mit der Klasse Rule, sondern mit der in Rule aggregierten Klasse Literal verknüpft ist und beide Diagramme dennoch ineinander passen. Auch ist es ohne weiteres möglich, Relationen zu erweitern, wie dies z.B. mit der assigned-Beziehung in Abbildung 13 (Seite 27) oder auch im Falle der adjacent-Relation in Abbildung 15 (Seite 29) vorgeführt wird. In allen Fällen zeigte sich, daß sich die anfänglichen sehr abstrakten Überlegungen aus der *Analysephase* sehr gut als Grundlage für konkrete, interne Strukturen heranziehen ließen. In der später gesichteten Literatur ([Mal95, S. 248]) fand ich übrigens meine Beobachtungen genau bestätigt. Dort wird insbesondere bemängelt, daß FUSION die hierarchische Zerlegung von Modellen kaum unterstützt und es wird sogar von *"design system object models"* gesprochen, die genau den erweiterten Klassenmodellen aus Abschnitt 3.1 entsprechen.

Die üblicherweise als erster Entwurfsschritt zu erstellenden *Object Interaction Graphs* (OIGs) konnten bei SPIRIT recht schnell entworfen werden. Da ich die wichtigsten Überlegungen zur Implementierung bereits vorher gemacht hatte und teilweise auch auf die Vorgängerversionen zurückgreifen konnte, war das größte Hindernis bei der Erzeugung der OIGs eher das Fehlen eines geeigneten CASE-Tools zum Zeichnen der Diagramme. Ich begann bei den einzelnen Funktionen mit der textuellen Beschreibung, zeichnete daraufhin die Graphen und nahm mit dem Fortschritt des Entwurfs kleinere bis mittelgroße Änderungen an den Diagrammen vor. Die OIGs bieten zwar eine recht anschauliche Darstellung der Funktionen, meiner Ansicht nach liesse sich der wesentliche Zweck dieses Entwurfsschrittes aber auch mit einer reinen Textform erreichen. Würde man anstelle der Graphen eine Art *Pseudo-Code* verwenden, könnte man außerdem den Kontrollfluß wesentlich präziser ausdrücken, als dies mit den etwas umständlichen Diagrammen der Fall ist. Versucht man nämlich, komplexere Abläufe mit vielen Abfragen und Schleifen durch einen OIG zu modellieren, wird dieser recht schnell unübersichtlich. Ein Vorteil der OIGs ist sicherlich, daß sie (wie der Name schon sagt) die Interaktionen zwischen den Objekten hervorheben und somit als Grundlage für die drei anschließenden Schritte des Design-Prozesses besonders geeignet sind. Insbesondere die *Visibility Graphs* können sehr schnell aus den OIGs erzeugt werden. Inwieweit also auf die OIGs verzichtet werden kann, hängt meiner Meinung nach von der Problemstellung und dem gegebenen Zeitrahmen für ein Projekt ab. Am besten ist sicherlich die Kombination beider Ansätze. Ich würde im Gegensatz zu [Col94, S. 63] die textuelle Beschreibung aber nicht ins *Data-Dictionary*, sondern direkt zu den OIGs verlagern (wie in Abschnitt 3.2 getan).

Bei der Entwicklung der OIGs gemäß FUSION wird von der Prämisse ausgegangen, daß alle Objekte aufeinander zugreifen können, also sichtbar sind. Dies bedeutet auch, daß sich erst die späteren Design-Schritte um eine Umsetzung der *Relationships* aus der *Analysephase* kümmern sollen (vgl. [Col94, S. 80]). Dies ist meiner Ansicht nach recht spät, weil dadurch der Abstand von Entwurfsdokument und Implementierung unnötig groß wird. Zumindest im Projekt SPIRIT war ich ohne weiteres in der Lage, die Zugriffspfade und die Beziehungen zwischen den Klassen schon frühzeitig (also in den OIGs) zu berücksichtigen. Dadurch wurde der für die Implementierung benötigte Zeitraum deutlich verkürzt, weil viele Funktionen direkt aus den OIGs gewonnen werden konnten. Demnach waren die *Visibility Graphs* für meinen Entwurf auch fast redundant und stellten lediglich ein weiteres Zwischenmodell dar, das zur Konsistenzprüfungen der übrigen Modelle herangezogen werden konnte.

Die *Inheritance Graphs* (IGs) sind ebenfalls nicht unbedingt erforderlich, wenn man über ein ausreichendes Abstraktionsvermögen verfügt und die *Class Descriptions* (CDs) zur Ermittlung der Vererbungsstruktur verwendet. Ich hatte mir erlaubt, schon früh mit der Erstellung der CDs anzufangen und konnte die Klassenhierarchie dadurch auch ohne IGs ermitteln. Insofern bin ich nicht in der Lage, die Notwendigkeit dieser

Graphen richtig einzuschätzen. Ich denke, daß die IGs zumindest eine sehr übersichtliche Darstellung der Klassen geben, die für interessierte projektfremde Personen als gute Einführung geeignet ist.

Auf Probleme stieß ich später bei der Implementierung von Methoden, die hauptsächlich *klasseninterne* Wirkung haben. Dies liegt daran, daß kein Analyse- und Entwurfsschritt von FUSION die Definition und Präzisierung lokaler Funktionen und interner Funktionsabläufe vorsieht und statt dessen zumeist ein *"Black-Box"*-Ansatz verfolgt wird. Besteht eine Methode aus einem komplexen Algorithmus, der allerdings größtenteils auf klasseninterne Attribute und Methoden zugreift, so enthalten weder *Object Interaction Graphs* noch die *Class Descriptions* genauere Informationen, wie die Methode später realisiert werden soll. Diese Detailinformationen werden meiner Erfahrung nach aber auch an einigen Stellen im Entwurf benötigt. Ich denke daher, daß zur Erleichterung der Implementierung wichtige und komplexe Algorithmen ebenfalls in den Entwicklungsprozess aufgenommen werden sollten. Hierzu wäre z.B. das *Data-Dictionary* geeignet oder etwa die Verwendung von halbformalen Algorithmen-Schemata (wie in Anhang D der vorliegenden Arbeit).

Zusammenfassend möchte ich festhalten, daß mir FUSION (in der von mir modifizierten Form) für Analyse und Entwurf sehr gut gefallen hat. Man sollte FUSION vor allem aber als *Möglichkeit* interpretieren und sich trauen, von den dort vorgeschlagenen Methoden und Formalismen abzuweichen und die gegebenen Freiräume im Sinne des speziellen Projektes zu nutzen. Sieht man von den kleineren (oben aufgeführten) Mängeln und Fehlern in FUSION ab, erhält man einen recht übersichtlichen Leitfaden zur Entwicklung komplexer Software-Systeme. Ich denke, der *kurze* für das Projekt SPIRIT benötigte Zeitraum spricht für FUSION.

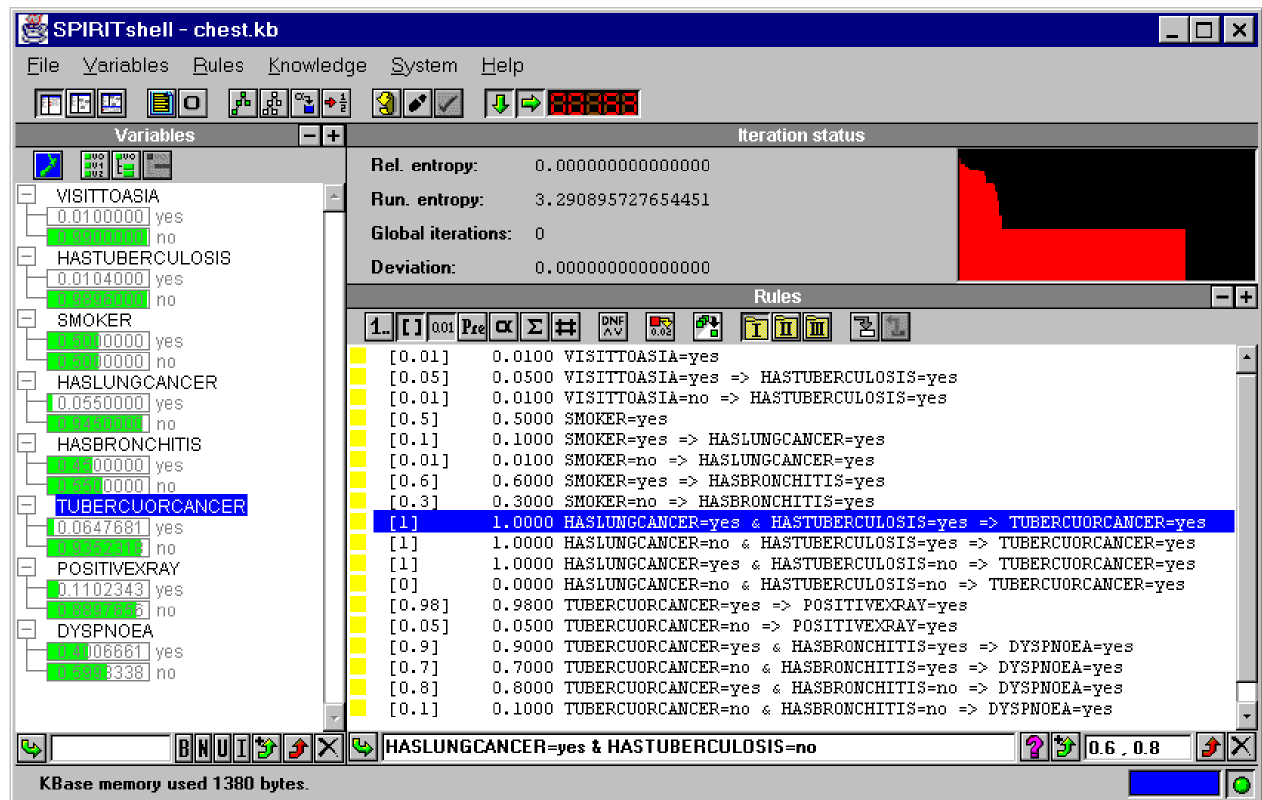


Abbildung 25: Ein Screenshot von SPIRITSHELL, als *Application* gestartet

5.2 Vergleich von SPIRIT mit den Vorgängerversionen

In diesem Abschnitt wird die neuentwickelte Klassenbibliothek SPIRIT (mit SPECTER) mit den Vorgängerversionen verglichen. Dieser Vergleich umfaßt die Klassenstrukturen der beiden Entwürfe sowie *Performance* und Laufzeitverhalten. Beide Tests sind allerdings mit Vorsicht zu genießen und sicherlich etwas unfair: Die Vorgängerversionen entstanden historisch und ohne jeglichen Entwurf, also mehr oder weniger spontan, und die Performancemessungen geraten durch die unterschiedlichen Programmiersprachen in Schieflage.

5.2.1 Vergleich der Klassenstrukturen

Nach der Fertigstellung des Entwurfs für die neue SPIRIT-Version habe ich mir erstmals erlaubt, die Klassenstruktur der Vorgängerversionen in eine ähnliche Form zu bringen. Dabei entstand das in Abbildung 26 gezeigte *Object Model*. Ich habe hierbei allerdings die Bezeichner so abgeändert, daß sie einen leichteren Rückbezug auf den neuen Entwurf zulassen.

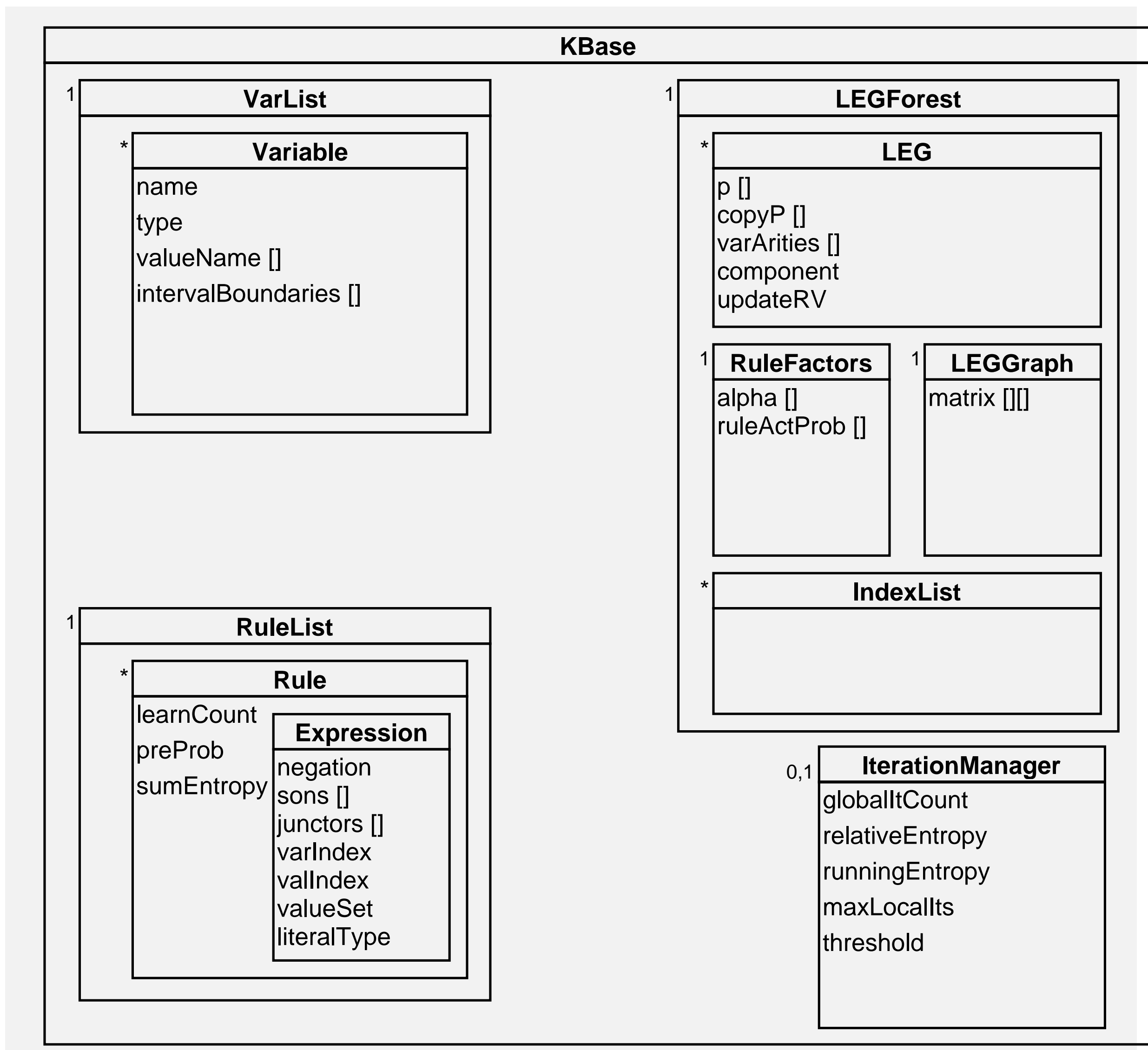


Abbildung 26: Übersicht über das "*Object Model*" der Vorgängerversion

Diese Abbildung enthält praktisch *alle* Klassen der Vorgängerversion. Die wesentlichen Unterschiede zur neuen Version offenbaren sich möglicherweise erst beim zweiten Hinsehen. Deutlich ist zu sehen, daß es damals wesentlich weniger Klassen gab. In der neuen Version habe ich hingegen verstärkt mit Vererbung gearbeitet. Während die alte Version schrittweise (und historisch) entstand, konnte ich beim neuen SPIRIT die gesamte Aufgabenstellung überblicken und dementsprechend besser auf einzelne Unterklassen verteilen. Als besonders

krasses Beispiel möchte ich hier die Klasse Expression aus der alten Version nennen, die jetzt durch eine Hierarchie aus *sieben* Unterklassen ersetzt ist.

Das offensichtliche Fehlen der Relationships im *Object Model* der Vorgängerversion ist übrigens nicht zufällig. Ich betrachtete für meinen damaligen Entwurf nur die Klassen an sich und weniger das Gefüge und die Beziehungen der Klassen untereinander. Die meisten Relationen wurden dementsprechend *dynamisch* (bei Bedarf) erzeugt und seitens der Implementierung nicht über Zeiger, sondern über Indizes verwaltet (es wurde nur über Regel- und Variablen-Indizes zugegriffen, was einen organisatorischen Overhead und unnötigen Verwaltungsaufwand nach sich zog). Ich denke daher, daß die Betonung der Relationships in Analyse und Entwurf eine sehr positive Eigenschaft von FUSION ist - gerade weil die Beziehungen zwischen den Klassen sonst häufig nicht erkannt oder falsch umgesetzt werden.

5.2.2 Vergleich des Laufzeitverhaltens

Ein Vergleich des Laufzeitverhaltens bzgl. Speichernutzung und Geschwindigkeit von den Vorgängerversionen und der neuen Version von SPIRIT ist nur mit großer Vorsicht möglich, weil beide Projekte in unterschiedlichen Programmiersprachen implementiert wurden. Während die Vorgängerversionen mit optimierten C++-Compilern übersetzt wurden und somit direkt in der schnellen Maschinensprache abliefen, wurde die neue SPIRIT-Version vollständig in JAVA geschrieben. JAVA-Quelltext wird (bisher) lediglich in plattformunabhängigen Zwischencode übersetzt und dann von einer Laufzeitumgebung interpretiert. Dadurch ist i.a. ein deutlicher Geschwindigkeitsverlust unvermeidlich.

Andererseits sind die in SPIRIT verwendeten Algorithmen (verglichen mit den alten Versionen) deutlich optimiert worden. An vielen Stellen wurde ein höherer Speicherverbrauch gegen eine höhere Ausführungsgeschwindigkeit "eingetauscht". Insbesondere wurden Index-Listen eingeführt (sh. Abschnitt 3.1.4.2), die zur Zwischenspeicherung von immer wieder benötigten Werten genutzt werden.

Die folgenden Performance-Tests (sh. Tabelle 2) wurden auf einem Pentium PC (90 MHz) mit 16 MB RAM und WINDOWS 95 durchgeführt. Die JAVA-Version wurde mit dem SYMANTEC JUST-IN-TIME-Compiler ausgeführt. Die genannten Wissensbasen sind sehr unterschiedliche Fallbeispiele, die jeweils verschiedene *Performance*-Aspekte, wie Speicherbedarf und Iterationsgeschwindigkeit besondes beleuchten.

Tabelle 2: Ergebnisse von Performance-Tests mit SPIRIT (links) und der besten Vorgängerversion (rechts)

Wissensbasis	Test 1		Test 2		Test 3		Test 4		Test 5		Test 6	
CarDiagnosis	12250	1360	3730	4890	660	6990	380	2100	9340	1650	17304	5990
ChestClinic	3900	590	220	250	170	670	170	300	220	110	1380	400
Bagang	128050	41340	32240	58210	32300	133480	5100	56300	114140	29550	918364	375850
Gut	10270	2750	4120	11540	1700	11720	600	5900	2960	1970	81200	31720
StudentFarm	8350	1880	660	640	270	810	220	22600	770	220	3952	1240

Test 1: 100 maliger Neuaufbau der LEG-Struktur (mit Variablennumerierung) (in ms)
Test 2: 10 maliger Aufruf von reset und einem vollständigen Iterationsprozess (Genauigkeit: 6 Stellen) (in ms)
Test 3: 100 malige Zuweisung und Zurücknahme von Evidenz (Wert 0 von Variable 0) (in ms)
Test 4: 100 malige Berechnung der aktuellen Wahrscheinlichkeiten aller Regeln (in ms)
Test 5: 10 malige Berechnung der Dicke einer Kante zwischen zwei Variablen im Abhängigkeitsgraphen (in ms)
Test 6: Benötigter Speicherplatz bei maximaler Geschwindigkeitsoptimierung (in Bytes)

Wie sich zeigt, unterscheiden sich die Testergebnisse beider Versionen beträchtlich (bis um den Faktor 10), allerdings hängt es von der gewählten Anforderung ab, welche Version besser abschneidet. Test 1 geht deutlich an die Vorgängerversionen - wie sich allerdings beim Neuentwurf von SPIRIT herausstellte, ist das damals implementierte Verfahren in einigen, seltenen Fällen fehlerhaft! Im sehr wichtigen Test 2 sind leichte bis mittelgroße Vorteile für die JAVA-Version zu erkennen. In diesem und in den Tests 3 und 4 wirken sich die Optimierungen mit Index-Tabellen besonders stark aus, die allerdings auf Kosten eines viel höheren Speicherbedarfs gehen (Test 6). Die Ergebnisse von Test 5 entsprechen in etwa dem "natürlichen" Unterschied zwischen der compilierten C++- und der interpretierten JAVA-Version.

Zusammenfassend läßt sich festhalten, daß die JAVA-Version - obwohl interpretiert - bei den wichtigsten und häufigsten Aufgaben der Vorgängerversion bzgl. Ausführungsgeschwindigkeit überlegen ist, teilweise sogar um einen zweistelligen Faktor! Ich denke, dies ist ein deutliches Zeichen, daß sich JAVA durchaus mit anderen Programmiersprachen messen kann, da man für die i.a. etwas geringere *Performance* andere Vorteile (wie Plattformunabhängigkeit und Absturzsicherheit der Programme) erhält. Außerdem sollte bedacht werden, daß JAVA noch relativ jung ist, die Compiler noch nicht den Zustand der etablierten C++-Systeme erreicht haben und

vielleicht sogar bald optimalen *Native-Code* erzeugende JAVA-Compiler auf den Markt kommen werden. Mir persönlich gaben die ausgeführten Testläufe das Gefühl, daß sich der Neuentwurf von SPIRIT in jeglicher Beziehung gelohnt hat.

5.3 Ausblick

Abschließend noch einige Bemerkungen zur Zukunft von FUSION, JAVA und SPIRIT.

FUSION wurde seit seiner ersten Vorstellung im Jahre 1993 in den HEWLETT-PACKARD-LABORATORIES weiterentwickelt. 1997 soll eine neue Generation der Methode der Öffentlichkeit präsentiert werden. Diese soll laut einem Pressebericht von HP vom Oktober 1996 einen besonderen Wert auf Software-Entwicklung durch Teams legen und hierzu spezielle Unterstützung bereitstellen. Auch wird hervorgehoben, daß das neue FUSION eine *"just-enough"*-Methode ist, die für jedes Projekt passend zugeschnitten werden kann. Der erste Schritt eines Projektes wird somit die Bestimmung der erforderlichen Präzision und Tiefe sein. Zudem soll FUSION die *Unified Modeling Language (UML)* verwenden, eine Modellierungssprache, die zum Standard in *Objektorientierter Analyse und Design (OOA/D)* werden soll. Hierdurch soll FUSION besser mit anderen Entwicklungsmethoden kombinierbar werden und von mehr CASE-Tools Unterstützung finden. Zudem soll die Entwicklung von Internet-bezogenen Systemen unterstützt und eine Anpassung an die Sprache JAVA in FUSION eingebaut werden.

Bei letzterer Programmiersprache ist schwer abzuschätzen, wie sie sich auf dem Markt entwickeln wird. Einige Prognosen sehen JAVA als Sprachstandard der nächsten Jahre. Nach den Plänen von SUN verfügen die Rechner der kommenden Generation nur noch über einen JAVA-Interpreter, einen *WWW-Browser* und einen Internet-Anschluß. Programme werden dann in Form von *Applets* direkt vom Hersteller über das Netz geladen und lokal ausgeführt. Doch bis es soweit kommt werden wohl noch einige Sicherheitsprobleme beseitigt und die Bandbreiten der weltweiten Computer-Netzwerke vergrößert werden müssen. Auch lassen sich solch radikale Umbrüche auf dem trägen Markt nur recht langsam durchsetzen, da die meisten derzeitig im Einsatz befindlichen Systeme völlig anders konzipiert sind. Meiner Ansicht nach wäre JAVA als Programmiersprache für die Zukunft eine gute Grundlage, da sie ein recht sauberes, objektorientiertes Konzept hat. Es wäre zweifellos kein Verlust, wenn die derzeitig populärste Sprache C++ langsam vom Markt verschwinden würde.

Die vorliegende Version von SPIRIT würde von einer Entwicklung zugunsten von JAVA sicherlich profitieren. In jedem Fall wird SPIRIT als Forschungsprojekt weiterhin in seiner Leistungsfähigkeit ausgebaut werden. Beispiele hierzu wurden am Ende der *Requirements* (in Abschnitt 1.2.5) bereits erwähnt. Ich habe beim Entwurf dieser Version besonderen Wert darauf gelegt, daß alle absehbaren Erweiterungen ohne größere Probleme in die Klassenbibliothek eingebaut werden können. Dadurch wird SPIRIT auf andere Arten mit Wissen ausgestattet werden können (z.B. durch Lernen aus Beobachtungen) und auch alternative Formen von Anfragen beantworten können (z.B. die Berechnung von Nutzen von Entscheidungen). Schließlich stehen verschiedene Kern-Algorithmen des Systems im Mittelpunkt intensiver weltweiter Forschung, insbes. die Erzeugung optimaler LEG-Strukturen. An der Erforschung und Nutzung des hier brachliegenden Optimierungspotentials wäre ich auch in Zukunft gerne beteiligt.

Anhang A: Syntax von Regeln und Fakten in SPIRIT

Abbildung 27 beschreibt die zulässige Syntax der Regeln und Fakten in SPIRIT. Die Beschreibung erfolgt in Form einer erweiterten BNF gemäß [Col94, S. 263].

Entgegen einer Konvention in der Aussagenlogik besteht in SPIRIT keine Rangfolge zwischen den Junktoren ∧ und ∨ (üblicherweise hat die Konjunktion Vorrang). Daher müssen bei der Deklaration von Regeln immer dann Klammern gesetzt werden, wenn eine Kette von Literalen gemischte Junktoren enthält.

Symbole (Tokens):

Number ::= ["-"] ("0".."9")$^{+}$ ["."("0".."9")$^{+}$]

String ::= ("A".."Z" | "a".."z" | "_" | "0".."9")$^{+}$

Implication ::= "⇒"

Negation ::= "¬"

Junction ::= "∧" | "∨"

EqualityOperator ::= "=" | "≠"

RangeOperator ::= "<" | ">"

ComparisonOperator ::= *EqualityOperator* | *RangeOperator*

ElementOperator ::= "∈" | "∉"

Grammatik:

Rule ::= *Fact Implication Fact*

Fact ::= *Expression*

Expression ::= ([*Negation*] (*Literal* | "(" *Expression* ")"))$^{++}$ *Junction*

Literal ::= *BooleanLiteral*

| $String_{Nominal\text{-}Variable}$ *NominalLiteral*

| $String_{Number\text{-}Variable}$ *NumberLiteral*

| $String_{Interval\text{-}Variable}$ *IntervalLiteral*

BooleanLiteral ::= $String_{Boolean\text{-}Variable}$

NominalLiteral ::= (*EqualityOperator String*) | (*ElementOperator* "{" (*String*$^{++}$ ",") "}")

NumberLiteral ::= (*ComparisonOperator Number*) | (*ElementOperator* "{" (*Number*$^{++}$ ",") "}")

IntervalLiteral ::= *RangeOperator Number*

Abbildung 27: Zulässige Syntax von Regeln und Fakten in SPIRIT

Zur einfacheren Darstellung der mathematischen Sonderzeichen gelten die in Tabelle 3 genannten Substitute.

Tabelle 3: Darstellung der Sonderzeichen in Regeln

Sonderzeichen	ASCII-Konvertierung
⇒	=>
¬	!
∧	&
∨	|
≠	!=
∈	@
∉	!@

Anhang B: Format von SPIRIT-Dateien

Abbildung 28 zeigt das zulässige Format von Export-/Importdateien in SPIRIT. Jede Datei beschreibt den aktuellen Zustand einer Wissensbasis, kann aber auch zur Erweiterung bestehender Wissensbasen in den Speicher geladen werden. Die folgenden Definitionen basieren auf Abbildung 27.

Zusätzlich gilt die Konvention, daß Text von "//" bis zum Zeilenende (als Kommentar) ignoriert wird.

AnyChar ::= **allbut**(";")
BeginBlock ::= "{"
EndBlock ::= "}"
NodeType ::= "boolean" | "nominal" | "number" | "interval"
Probability ::= ("1" ["." "0"$^+$]) | ("0" ["." *Digit*$^+$])
FreeDecl ::= *Name* "=" *AnyChar*
NetBlock ::= "net" *BeginBlock* (*FreeDecl* ";")* *EndBlock*
NodeTypeDecl ::= "type" "=" *NodeType* ";"
UtilitiesDecl ::= "utilities" "=" "(" Number^{++} ")" ";"
ValueList ::= "states" "=" "(" *ValueList* ")" ";"
EnumDecl ::= "enum" "=" *Number*
NodeDecl ::= *EnumDecl* | *FreeDecl*
NodeBlock ::= "node" *Variable BeginBlock*
NodeTypeDecl [*ValueListDecl*] [*UtilitiesDecl*] (*NodeDecl* ";")* *EndBlock*
RuleStringDecl ::= "string" "=" (*Rule* | *Fact*) ";"
ProbDecl ::= "prob" "=" *Probability* ";"
AlphaDecl ::= "alpha" "=" *PosNumValue*
RuleDecl ::= *AlphaDecl* | *FreeDecl*
RuleBlock ::= "rule" *BeginBlock RuleStringDecl ProbDecl* (*RuleDecl* ";")* *EndBlock*
File ::= (*NetBlock* | *NodeBlock* | *RuleBlock*)*

Abbildung 28: Format von SPIRIT-Dateien

Anhang C: Von SPIRIT gemeldete Fehler (*Exceptions*)

Tabelle 4 listet die Fehlercodes, die von den System-Operationen in Form einer KBE-*Exception* geliefert werden können. Zu einigen Fehlern werden zusätzliche Informationen mitgeliefert, die die Fehlerursache erläutern und im Vector KBE.info abgelegt sind. Diese Infos können beliebige Objekte sein (z.B. Strings und Zahlenwerte).

Die meisten Fehler werden durch ungültige Funktionsparameter ausgelöst, z.B. wenn eine Methode einen Integer-Wert aus einem bestimmten Intervall erwartet. Eine andere Fehlerursache ist die Verletzung einer Vorbedingung aus der *assumes*-Klausel im *Operation Model*.

Die in der linken Spalte der Tabelle aufgeführten Bezeichner sind Konstanten, die im Modul KBE.java definiert werden. Im Online-Handbuch ist zu jeder Methode aufgeführt, welche Fehlercodes durch sie ausgelöst werden können (sh. z.B. in Abbildung 24, Seite 68).

Tabelle 4: Fehlercodes in SPIRIT

Fehlercode (KBE.code)	Beschreibung und zusätzliche Informationen (KBE.info)
FATAL	Fataler (interner) Fehler, der nur in der Testphase auftreten sollte.
AMBIGUOUS	Im String einer Regel: Verwendung von Disjunktion und Konjunktion innerhalb einer Klammerebene ist nicht zulässig (explizite Klammerung erforderlich).
ARGUMENT	Funktion wurde mit einen ungültigen Parameter aufgerufen.
CONTRADICTION	Widerspruch im Iterationsprozeß: Mindestens zwei widersprüchliche Regeln.
EMPTY_RULE	Es wurde versucht, eine intern widersprüchliche Regel einzugeben.
EMPTY_SET	Eine Regel wurde (nach Umformung in DNF) als widersprüchlich erkannt.
ENUM	Unzulässige Variablennumerierung für rebuild.
EVIDENCE	Noch Evidenz zugewiesen: der Funktionsaufruf ist in diesem Zustand unzulässig (Verletzung einer Vorbedingung gemäß *assumes*-Klausel aus *Operation Model*).
EXPECTING	Im String einer Regel oder in SPIRIT-Datei: Unerwartetes Symbol.
EXPECTING_PROB	Wahrscheinlichkeit erwartet (rationale Zahl aus [0..1]).
FILE	Schreib- oder Lesefehler mit einer Datei.
ITERATING	Iterationsprozeß aktiv: der Funktionsaufruf ist in diesem Zustand unzulässig.
LEGSIZE	Operation würde LEG unzulässig vergrößern: Wertetabelle einer LEG wäre zu groß.
LEG_ZERO	Widerspruch im Iterationsprozeß: Eine LEG ist vollständig null.
NOT_ENOUGH_VALUES	Nicht genügend Werte bei Variablendeklaration.
OPIMPNOW	Operation in diesem Zustand nicht möglich (Verletzung einer *Precondition*).
RULEINDEX	Ungültiger Regel-Index (meistens Argument ruleIndex).
RULE_FACT	Operation unzulässigerweise auf ein Fakt angewendet.
RULE_PASSIVE	Regel ist passiv.
UNDEF_VALUE	Im String einer Regel: Undefinierter Variablenwert.
UNDEF_VARIABLE	Im String einer Regel: Undefinierte Variable.
VALINDEX	Unzulässiger Variablenwert (meistens Argument valIndex).
VALUE_EXISTS	Variablenwert konnte nicht hinzugefügt werden, da er bereits existiert.
VAR_EXISTS	Variable konnte nicht hinzugefügt werden, da sie bereits existiert.
VARINDEX	Unzulässiger Variablen-Index (meistens Argument varIndex).
VARTYPE	Operation mit Variablen dieses Types nicht zulässig.
WRONG_TYPE	Methode wurde mit Argument in ungültiger Objektklasse aufgerufen.

Anhang D: Einige Algorithmen

Zur Ergänzung der *Requirements* aus Abschnitt 1.2 und zur Präzisierung einiger programminterner Verfahren werden in diesem Anhang verschiedene Algorithmen und Formeln vorgestellt, die in SPIRIT Verwendung finden. Die Beschreibung erfolgt in Form von *Algorithmen-Schemata* gemäß Abbildung 29. Diese ähneln den *Operation-Schemata* aus der *Analysephase* von FUSION (sh. hierzu auch Abschnitt 2.3 und [Col94]).

Algorithm:	**Name des Algorithmus**
Description:	Kurze textuelle Beschreibung von Eingabe und Ergebnis.
Reads:	Eingabe des Algorithmus (Objekte und Werte)
Assumes:	Bedingungen, die vor der Ausführung des Algorithmus erfüllt sein müssen.
Result:	Rückgabewerte und Seiteneffekte durch die Ausführung des Algorithmus.
Method:	Beschreibung der Arbeitsweise mit Text, mathematischen Ausdrücken und Pseudo-Code.

Abbildung 29: Muster eines Algorithmen-Schemas

Die meisten der folgenden Algorithmen entstanden während der *Analysephase*. Dabei wurde noch eine sehr abstrakte und allgemeine Beschreibungsform verwendet. Mit dem Fortschritt der Design-Phase konnten viele Algorithmen allerdings dahingehend präzisiert werden, daß direkt auf die eingeführten Objektklassen und Attribute zugegriffen wird. Hierdurch wird zwar das Abstraktionsniveau deutlich gesenkt, aber gleichzeitig eine wesentlich bessere Verbindung zum Entwurfsdokument und dem Programmcode geschaffen.

Jeder Algorithmus hat einen Namen, über den er im Hauptteil der Arbeit referenziert wird.

Algorithm:	**AssignEvidence**
Description:	Weist einer Variablenmenge Evidenz zu.
Reads:	**supplied** varList: Vector of Variable, valList: Vector of Value **with** |varList| = |valList|
Assumes:	Auf der aktuellen Verteilung sind keine Evidenzzuweisungen vorgenommen.
Result:	**changes** LEG (Für alle angegebenen Variablen i gilt i.a.: $P(var_i = val_i) = 1$)
Method:	see below

```
foreach root: Wurzel-LEG von Bäumen des LEG-Waldes, in denen Variablen aus list vorkommt:
   root.assignAndCollectEvidence
   root.distributeEvidence

procedure LEG.assignAndCollectEvidence      // Lokale Evidenzzuweisung mit Angleichen
   foreach son of this do                   // Für alle Sohn-LEGs...
      son.assignAndCollectEvidence          //    Bottom-Up-rekursiver Aufruf
      LEGCalibrate(this, son)               //    Neue Informationen aus Sohn holen
   foreach var ∈ varList with var.inLEG=this do  // Für alle Variablen, die this zugeordnet sind
      lösche alle Konfigurationen mit var≠val
   normiere

procedure LEG.distributeEvidence            // Top-Down-Angleichen des Unterbaumes
   foreach son of this do                   // Für alle Sohn-LEGs...
      LEGCalibrate(son, this)               //    Neue Informationen an Sohn schicken
      son.distributeEvidence                //    Top-Down-rekursiver Aufruf
```

Optimierungen: Nur diejenigen LEGs betrachten und angleichen, die tatsächlich von den Änderungen betroffen sind, d.h. in der varList genannte Variablen enthalten.

Algorithm: **LEGAbsEntropy**
Description: Ermittelt die absolute Entropie einer LEG.
Reads: **supplied** leg: LEG
Result: entropy: double (die absolute Entropie der LEG bzgl. der Gleichverteilung)
Method:

```
entropy := 0                                   // Beginne bei null
foreach c: Konfiguration in leg do             // Summiere über alle Konfigurationen
   entropy = entropy - p[c] * log (p[c])       //    Prob * log(Prob)
```

Algorithm: **LEGCalibrate**
Description: Gleicht die Wahrscheinlichkeiten zweier benachbarter LEGs an (vgl. [JOA90, S. 645]).
Reads: **supplied** legD (Ziel), legS (Quelle): LEG, sep: Separator
Assumes: Die Legs haben die gemeinsame nichtleere Schnittmenge sep.
Im Separator stehen die aktuellen Randwahrscheinlichkeiten aus legD.
Result: **changes** ld (alle Variablen aus sep haben in legD dieselben Wahrscheinlichkeiten wie in legS)
changes sep (enthält die aktuellen Randwahrscheinlichkeiten aus legS)
Method:

legS → (2) → sep → (1, 3) → legD

```
1. Divide
   Teile legD konfigurationsweise durch sep.   // Teile durch alte Wahrscheinlichkeiten

2. Marginalize
   Fülle sep mit legS.                         // Bilde neue Randverteilung

3. Multiply
   Multipliziere legD mit sep.                 // Multipliziere mit neuen Wahrscheinlichkeiten
```

Algorithm: **LEGForestAbsEntropy**
Description: Ermittelt die absolute Entropie eines LEG-Waldes.
Reads: **supplied** legForest: LEGForest,
Result: entropy: double (die absolute Entropie von legForest bzgl. der Gleichverteilung)
Method:

```
// Summe der Einzel-Entropien der LEGs abzüglich der Entropien der Schnitte (Separatoren)
entropy := 0                                                 // Beginne bei null
foreach root @ legForest do                                  // Für jeden Baum im LEG-Wald...
   leg := root                                          (*)  //    Beginne bei Wurzel
   entropy = entropy + LEGAbsEntropy (leg)                   //    Addiere Entropie der LEG
   foreach son @ leg do                                      //    Für jeden Sohn der LEG...
      Rekursion zu (*) mit leg := son                        //       Rekursiver Baumabstieg
      entropy = entropy - LEGAbsEntropy (son ∩ leg)          //       Ziehe Entropie des Schnittes ab
```

Algorithm: **LEGReset**
Description: Setzt eine LEG unter Berücksichtigung der Intervall-Variablen auf Gleichverteilung zurück.
Reads: **supplied** leg: LEG; Variable.in.l
Result: leg wurde auf Gleichverteilung zurückgesetzt.
Method:

```
x := 1 / leg.ProbTable.size                            // Normaler Wert jeder Konfiguration
foreach c: Config in leg do ProbTable[c] := x          // Alle Konfigs gleichsetzen
foreach var.in.leg with var.type = interval do         // Für alle Intervall-Variablen...
   for i:=0 to |var|-1 do                              // Für alle Teilintervalle von v...
      f := n * abs(Intervallbreite(i) / Gesamtbreite)  // Faktor berechnen (Dichtefunktion)
      C_i := leg.IndexList (v=i)                       // Für alle Konfigurationen mit v=i...
      leg.multiplyConfigs (C_i, f)                     //    Multipliziere mit Faktor f
```

Algorithm:	**LEGForestBuild**
Description:	Baut eine LEG-Struktur zu einer gegebenen Variablen-Eliminationsfolge auf.
Reads:	Variable.enum, Rule
Assumes:	Variable.enum enthält eine *gültige* Variablen-Numerierung. Dazu müssen alle Indizes von 0..|VarList|-1 genau einmal vorkommen.
Result:	LEG-Struktur oder failure: Exception (ein LEG wurde zu groß)
Method:	

```
1. Erzeuge (unverbundene) Menge der LEGs
   Sei G = (V, E) der Abhängigkeitsgraph der Variablen, ermittelt durch die Regelmenge
   Sei v(e) = Variable with v(e).enum = e
   L := ∅                                                          // Menge der LEGs
   for e:=0 to |V|-1 do                                            // Numeriere ansteigend...
      Verbinde in G alle Nachbarn von v(e) vollständig untereinander   // Clique entsteht
      set := { v(e) } ∪ { v' | v' benachbart mit v(e) }            // set ist eine Clique
      if |set| > MaxLEGSize then                                   // LEG zu groß?
         throw Exception failure (KBECode: LEGSIZE)                // Fehlermeldung
      L = L ∪ { neue LEG mit der Variablenmenge set }              // Neue LEG
      Streiche v(e) aus G
   Streiche alle l ∈ L, die Teilmenge einer anderen LEG sind       // Redundanz
   return Menge der LEGs L                                         // Ergebnis: Liste von LEGs

2. Verbinde LEGs zu einem Wald (Kruskal-Algorithmus nach [JJ94] und [CLR90, S. 504ff])
   Sei G_pot = (L, E_pot) der LEG-Graph mit allen potentiellen Kanten
   Sei w(e) = Anzahl der Variablen im Schnitt der beiden über e ∈ E_pot verbundenen LEGs
   F := (L, ∅)                                        // der resultierende LEG-Wald
   for e ∈ E_pot absteigend sortiert nach w(e) do     // Starte bei Kante mit größtem Gewicht
      if F ∪ e zyklenfrei then                        // Wenn kein Zyklus entstünde
         F := F ∪ e                                   // Nimm neue Kante zum Baum hinzu
   return LEG-Wald F = (L, E_F)                       // Ergebnis: LEG-Wald

3. Erzeuge Separator-LEGs
   for e=(l_1,l_2) ∈ E_F do                           // Für alle Kanten des LEG-Waldes...
      s := l_1 ∩ l_2                                  // Bilde Schnittmenge der beiden LEGs der Kante
      L := L ∪ { s }                                  // Erzeuge neue LEG aus dieser Schnittmenge
      E_F := { E_F \ e } ∪ { (l_1,s) , (s,l_2) }      // Ersetze Kante durch zwei Kanten zum Separator
   return F                                           // Ergebnis: LEG-Wald mit Separator-LEGs
```

Algorithm:	**LEGRelEntropy**
Description:	Ermittelt die relative Entropie zwischen zwei LEGs.
Reads:	**supplied** leg_0, leg_1: LEG, beide LEGs haben dieselbe Struktur
Result:	entropy: double die relative Entropie von leg_1 bzgl. leg_0
Method:	

```
entropy := 0                                               // Beginne bei null
foreach c: Konfiguration in den legs do                    // Summiere über alle Konfigurationen
   entropy = entropy - p(c_1) * log (p(c_1) / p(c_0))      //    Prob * log(Prob_1 / Prob_0)
```

Algorithm: **Description:**	**LEGForestReinit** Aktualisiert die bedingten Wahrscheinlichkeiten in einer LEG gemäß den Regel-alphas.

Reads:
Assumes:
Result:
Method:
	LEG, Rule.alpha, etc.
	Gleichverteilung liegt vor.
	Eine neue Verteilung.
	
	// Baumdurchlauf mit Multiplikation der Alphas...
	foreach Wurzel-LEG root **do** // Für jeden Baum des LEG-Waldes...
	   root.reinitCollect // Neue Informationen aus Baum holen
	   distributeEvidence mit Normierung // Neue Informationen zurück in Baum schicken
	// (sh. AssignEvidence)
	
	procedure LEG.reinitCollect // Bottom-Up-rekursive Methode
	   Rekursion für alle Söhne
	   foreach active rule.in.this **do** // Für jede aktive Regel, die in die LEG paßt
	      LEGMultiplyRuleAlpha(rule, this) // Multipliziere Alpha in die LEG
	   Multipliziere Parent-LEG über Separator (ähnlich LEGCalibrate, aber ohne Division)

Algorithm: **Description:**	**RuleActPremProb** Ermittelt die aktuelle bedingte Prämissenwahrscheinlichkeit einer Regel
Reads: **Result:** **Method:**	**supplied** rule: Rule; leg: LEG **with** rule.assigned.leg p: double (aktuelle bedingte Prämissenwahrscheinlichkeit der Regel) wie RuleActProb, nur mit p := leg.sumConfigs($rule_{prem}$)

Algorithm: **Description:**	**RuleActProb** Ermittelt die aktuelle bedingte Wahrscheinlichkeit einer Regel.

Reads:
Result:
Method:
	supplied rule: Rule; leg: LEG, legForest: LEGForest, Variable
	p: double (aktuelle bedingte Wahrscheinlichkeit einer Regel)
	
	if es gibt keine passende LEG **then**
	   lf := legForest.clone () // Erzeuge temporäre Kopie des LEG-Waldes
	   vars := rule.getVars () // Ermittle die Menge der Variablen in rule
	   leg := **new** LEG (vars); lf.addLEG (leg) // Erzeuge neue LEG über diese Variablen
	   lf.reinit () // Fülle lf mit den aktuellen bedingten Probs
	else
	   leg := rule.getMinLEG // Wähle die kleinste passende LEG
	
	if rule ist Fakt **then**
	   p:= leg.sumConfigs (rule.C_{conc}) // Summe der Konfigurationen der Conclusio
	else
	   c_0 := $rule_{conc}$ // Hole die Index-Liste der Conclusio
	   c_1 := $rule_{prem}$ // Hole die Index-Liste der Prämisse
	   p:= leg.sumConfigs ($c_0 \cap c_1$) / // Summe nach bedingter Wahrscheinlichkeit
	      leg.sumConfigs (c_1)

Algorithm:	**RuleLearn**
Description:	Lernt eine Regel in eine passende LEG.
Reads:	**supplied** rule: Rule (die zu lernende Regel); leg: LEG **with** rule.assigned.leg
Result:	Die Regel ist in der LEG gültig oder Widerspruch. **Changes:** Summe der Konfigurationen s.
Method:	

```
if rule.isFact () then
   c_1 := rule                                  // "Regelwahrscheinlichkeit"
   c_0 := leg \ c_1                             // "Gegenwahrscheinlichkeit"
else
   c_1 := rule_conc ∩ rule_prem                 // "Regelwahrscheinlichkeit"
   c_0 := rule_conc ∩ (leg \ rule_prem)         // "Gegenwahrscheinlichkeit"

p_i := leg.sumConfigs (c_i), i = 0,1            // Summiere Konfigurationen der LEG

rule.α := 0, wenn rule ist sicher               // Sichere Regeln: nur einmal lernen
rule.α := 1, wenn p_0 = 0 oder p_1 = 0, Abbruch // Widerspruch
rule.α := f * p_0 / p_1, sonst                  // Neues Alpha nach Formel
   wobei f := rule.preProb / (1-rule.preProb)
x := RuleActProb (rule)                         // Hole aktuelle bedingte Wahrscheinlichkeit
aRule := rule.α / rule.α_old                    // Hilfsvariable
s := s + (aRule^(-x)-1 * p_0 + aRule^(1-x)-1 * p_1)  // Errechne neue Gesamtsumme der Configs

ξ := (x / (1-x)) * p_0 / p_1                    // Berechne Hilfs-Variable
f_0 := ξ^(-x) , f_1 := ξ^(1-x)                  // Berechne Faktoren für die Konfigurationen
leg.multiplyConfigs (c_i, f_i), i = 0,1         // Multipliziere die Konfigurationen
```

Algorithm:	**RuleLearnIteration**
Description:	Lernt alle Regeln in die aktuelle Verteilung.
Reads:	Menge der aktiven Regeln, LEG-Struktur
Result:	Die Regeln wurden (soweit möglich) in die LEGs gelernt.
Method:	

```
runningEntropy := LEGForestAbsEntropy ()     // Startwert für laufende Entropie
repeat                                       // Globale Iteration
   Für alle Bäume, in denen aktive Regeln sind
   leg := root                               //    Beginne an Wurzel
   loopCount := 0                            //    Zähle lokale Durchläufe
   repeat                                    //    Lokale Iteration
      loopCount := loopCount + 1             //       Nächste lokale Iteration
      relSum := 0                            //       Summiere relative Entropien
      foreach rule ∈ RestListe do   (s.u.)   //       Für alle noch zu lernenden Regeln
         relSum := relSum + RuleLearn(rule)  //          Lerne Regel und addiere rel. Entropie
      runningEntropy = runningEntropy-relSum //       Entropien aufsummieren
   until relSum <= threshold                 //    Bis Entropie konvergiert oder
      or loopCount >= maxLocalIts            //       ausreichend viele lokale Iterationen
   foreach Söhne s von leg do                //    Rekursion zu Söhnen...
      LEGCalibrate (s, leg)                  //       Änderungen an Nachbarn schicken
      Rekursion zur lokalen Iteration mit leg=s
      LEGCalibrate (leg, s)                  //       Änderungen vom Nachbarn holen
until runningEntropy unverändert             // Bis Entropie konvergiert ist
Normiere anhand der Variable s aus RuleLearn
```

RestListe: Alle Regeln, die leg zugeordnet sind, außer:

- sichere Regeln, die schon mindestens einmal gelernt wurden,
- schon geltende Regeln (bzw. mit Abweichung threshold geltende), und
- Regeln, die während der letzten lokalen Iteration keine Entropieänderungen bewirkten.

Algorithm: **Description:**	**RuleTransformToDNF** Wandelt eine Regel in kanonische Disjunktive Normalform (DNF) um.
Reads: **Result:** **Method:**	Regel in Form zweier Bäume (bei Fakten: ein Baum) aus Ausdrücken (Expressions) eine äquivalente Regel in DNF-ähnlicher Darstellung: Die Literale haben die Form "Variable=Wert" oder "Variable≠Wert". Der Algorithmus besteht aus sechs Teilschritten, die jeweils an den Wurzeln der Ausdrucksbäume starten und dann gemäß Tiefensuche bis zu den Blättern hinabsteigen. Der siebte Schritt sollte nicht mehr auf Ausdrucksbäumen durchgeführt werden, da dies einen großen Speicherbedarf bedeuten kann. Das Verfahren basiert auf ähnlichen Ansätzen mit booleschen Ausdrücken aus der Literatur, z.B. Uwe Schöning, *Logik für Informatiker* (3. üb. Aufl), BI-Wissenschafts-Verlag, 1992, Seite 27 ff.
	1. Wandle alle Literale zu EqualityLiterals um (nur die Operatoren = und ≠) Für jedes Literal werden folgende Umwandlungsgesetze angewendet. Die Negation wird dabei beibehalten (bei den Operatoren ∉ und >). • $n \in \{ v_{i1}, ..., v_{ik} \} \rightarrow (n = v_{i1} \vee ... \vee n = v_{ik})$ • $n < v_i \rightarrow (n = v_1 \vee ... \vee n = v_{i-1})$
	2. Forme den Ausdrucksbaum in einen "Binärbaum" um Beim Erreichen einer SubExpression im Baumdurchlauf wird folgendes durchgeführt: **while** (die Kette unter der SubExpression hat eine Länge > 2) **do** Erzeuge eine neue SubExpression aus den beiden ersten Objekten der Kette Ersetze die beiden Objekte durch diesen neuen Unterbaum
	3. Schiebe alle Negationen in die Blätter Für jede im Baumabstieg erstmals erreichte negierte SubExpression wende folgende Regeln (von *de Morgan*) an: • $\neg (A \wedge B) = (\neg A \vee \neg B)$ • $\neg (A \vee B) = (\neg A \wedge \neg B)$
	4. Sorge dafür, daß keine Disjunktion unterhalb einer Konjunktion auftritt Dazu werden in einem *post-order*-Durchlauf mit Anwendung der folgenden Distributivgesetze alle Disjunktionen aus den Unterbäumen nach oben gezogen (nach jedem Tauschvorgang muß der Schritt eine Blattebene tiefer rekursiv wiederholt werden). • $(A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C)$ • $C \wedge (A \vee B) = (C \wedge A) \vee (C \wedge B)$
	5. Entferne unnötige Klammerebenen • $A \vee (B \vee C) = A \vee B \vee C$ • $(A \wedge B) \wedge C = A \wedge B \wedge C$
	6. Entferne ungültige Konjunktionen Löscht alle Konjunktionen, in denen eine Variable mehrmals mit unterschiedlichen Werten auftritt. Falls dadurch keine Konjunktion übrig bleibt, ist die Regel widersprüchlich.
	7. Vervollständige alle Konjunktionen zur kanonischen DNF Dazu wird jede Konjuktion um alle nicht genannten Variablen vervollständigt, die in der Regel vorkommen. Dabei werden i.a. neue Konjunktionen mit allen Konfigurationen über die fehlenden Variablen in die Regel aufgenommen. Da dadurch sehr umfangreiche Strukturen entstehen können, bietet sich eine andere Speicherung, z.B. wie mit der Klasse DNF an.

Algorithm: **Description:**	**LEGMultiplyRuleAlpha** Multipliziert eine LEG konfigurationsweise mit dem Lagrange-Parameter α einer Regel.

Reads:
Result:
Method:
	supplied rule: Rule; leg: LEG **with** rule.assigned.leg
	changes leg (die entsprechenden Konfigurationen wurden multipliziert, aber **nicht** normiert!)
	
	if rule ist Fakt **then**
	   c_1 := $rule_{conc}$ // Konfigurationen des Fakts
	   c_0 := leg \ c_1 // Restliche Konfigurationen
	else
	   $c_1 := rule_{prem} \cap rule_{conc}$ // Konfigurationen Prämisse ∩ Conclusio
	   $c_0 := rule_{prem} \cap (leg \setminus rule_{conc})$ // Konfigurationen Prämisse ∩ ¬Conclusio
	
	x := rule.PreProb // Vorgegebene bedingte Wahrscheinlichkeit
	leg.multiplyConfigs(c_0, rule.alpha^{1-x}) // Multipliziere die Konfigurationen c_0
	leg.multiplyConfigs(c_1, rule.alpha^{-x}) // Multipliziere die Konfigurationen c_1

Algorithm: **Description:**	**VarEnumMaxCS** Variablennumerierung nach *Maximum Cardinality Search* gemäß [Kjæ90]

Reads:
Result:
Method:
	Variable, Rule
	Variable.enum beschreibt eine Permutation über die Variablenmenge
	
	Bilde Abhängigkeitsgraphen G = (V, E) aus Variablen und Regeln
	V.enum := undef // Anfangs ist keine Variable numeriert
	i := |V| - 1 // Numeriere absteigend
	repeat
	   Sei v ∈ V beliebig **with** v.enum = undef // Suche nächsten Baum
	   while v existiert **do** // Solange noch Nummern zu vergeben sind...
	      v.enum := i // Weise v den nächsten Index zu
	      i := i - 1 // Zähle Index weiter
	      v := derjenige Nachbar von v, der am wenigsten numerierte Nachbarn hat
	until i<0 // Bis alle Nummern vergeben

Algorithm: **Description:**	**VarEnumMinCF / VarEnumMinCS / VarEnumMinCW** Variablennumerierung nach den drei *Minimum Heuristiken* gemäß [Kjæ90]

Reads:
Result:
Method:
	Variable, Rule
	Variable.enum beschreibt eine Permutation über die Variablenmenge
	
	G = (V, E) // Bilde Abhängigkeitsgraphen
	V.enum = undef. // Anfangs ist keine Variable numeriert
	for i := |V| - 1 **downto** 0 **do** // Numeriere absteigend
	   v := nächste Variable gemäß Heuristik // Von gewählter Heuristik abhängig
	   v.enum := i // Weise Enumerationswert zu
	   Verbinde alle Nachbarn von v untereinander // Erzeuge Clique
	   Streiche v aus G // Verkleinere Graphen
	
	Heuristiken:
	• ***Minimum Clique-Fill-In***: Wähle diejenige nichtnumerierte Variable, für die zur Vervollständigung ihrer Clique am wenigsten zusätzliche Kanten nötig wären
	• ***Minimum Clique-Size***: Wähle diejenige nichtnumerierte Variable, für die eine Clique mit minimaler Größe (=Anzahl der Variablen) entstünde
	• ***Minimum Clique-Weight***: Wähle diejenige nichtnumerierte Variable, für die (nach Verbinden seiner verbliebenen Nachbarn) das minimale Cliquengewicht (=Zahl der Zustände) entstünde

Algorithm:	**VarGraphEdgeThickness**
Description:	Ermittelt die Dicke einer Kante im Variablengraphen.
Reads:	**supplied** var_1 , var_2: Variable; Rule
Assumes:	var_1.inRules ∩ var_2.inRules ≠ ∅, andernfalls ist t := 0
Result:	t: double (Dicke der Kante zwischen var_1 und var_2 in einem Variablengraphen)
Method:	

```
setFull := { var1, var2 } ∪ { alle Nachbarn von var1 und var2 }
lf := LEGForest.clone ()                  // Lege temporäre Kopie des LEG-Waldes an.
leg := new LEG (setFull)                  // Erzeuge neue LEG
lf := lf ∪ l                              // Nimm neue LEG hinzu
lf.reinit ()                              // Fülle temp. LEG-Wald mit alten Probs
Q1 := leg.getRV (setfull \ var1)          // Randverteilung ohne var1
Q2 := leg.getRV (setfull \ var2)          // Randverteilung ohne var2
Qn := leg.getRV (setfull \ { var1, var2 } )  // Randverteilung ohne var1 und var2
leg' := new LEG (setFull)                 // Bilde neue temp. LEG
l.prob := Q1*Q2/Qn                        // Fülle neue LEG nach Formel
t := LEGRelEntropy (leg, leg')            // Dicke := relative Entropie zwischen den LEGs
```

Algorithm:	**VarValueActProb**
Description:	Ermittelt die aktuelle bedingte Wahrscheinlichkeit eines Variablenwertes.
Reads:	**supplied** var: Variable, valIndex: int; leg: LEG **with** var.in.leg
Result:	p: double (aktuelle Wahrscheinlichkeit des Variablenwertes valIndex)
Method:	

```
// Summiere alle Konfigurationen mit var = valIndex
Sei v0..vn die Variablen in der LEG, var = vx
left := |v0| * ... * |vx-1|        // Produkt der Stelligkeiten links von var
right := |vx+1| * ... * |vn|       // Produkt der Stelligkeiten rechts von var
p := 0                             // Beginne bei null
i := left * valIndex               // Laufender Konfigurationen-Index
for j:=0 to right-1 do             // Für alle "Cluster" in der Tabelle...
   for k:=0 to left-1 do           //    Für alle Konfigurationen im Cluster...
      p := p + leg.p(i)            //       Addiere P-Wert der Konfiguration
      i := i + 1                   //       Nächste Konfiguration...
   i := i + left * (|var|-1)       //    Überspringe Konfigurationen
```

Algorithm:	**VarValueExpected**
Description:	Ermittelt den Erwartungswert einer Variable.
Reads:	**supplied** var: Variable; leg: LEG **with** var.in.leg
Result:	p: double (Erwartungswert der Variable var)
Method:	

```
p := 0                                       // Beginne bei null
for val := 0 to |var|-1 do                   // Für alle Variablenwerte...
   u := val.utility  (s.u.)                  //    Utility-Wert holen
   p := p + VarValueActProb(var,val) * u     //    Summiere gewichtete W'keiten

val.utility ergibt sich aus:
   if var.type = number then u := val.number              //    Fall 1: Number-Variable
   if var.type = interval then u := val.upper - val.lower //    Fall 2: Interval-Variable
   otherwise u := 0, bzw. ein frei definierter Wert       //    Fall 3: Boolean und Nominal
```

Anhang E: Kommandos des Testtreibers SPECTER

Dieser Abschnitt gibt eine Übersicht über die zulässigen Kommandos des Programms SPECTER, das als Testtreiber entwickelt wurde und u.a. zur Erprobung und Optimierung der System-Operationen diente. Zu diesem Zweck entsprechen die meisten Kommandos ziemlich genau den Operationen aus der *Analysephase* (sh. Abschnitt 2.3).

Mehrere Befehle können mit ";" getrennt eingegen werden. Eine Zahl n vor einem Kommando führt den Befehl n mal aus. Ein Fragezeichen vor dem Kommando mißt die für die Ausführung benötigte Zeit (in Millisekunden). Befehlswörter und Parameter sind üblicherweise über Leerzeichen getrennt. Werden andere Trenner (z.B. Kommata) verwendet, so werden auch die Trennzeichen selbst als Parameter interpretiert. Text zwischen zwei Anführungszeichen (") gilt als ein Wort (dies wird insbes. bei der Definition von Regeln benötigt).

Tabelle 5: Die zulässigen Kommandos von SPECTER

Befehl	Parameter	Funktion
absentr		entspricht absEntropy
addrule	str preprob	entspricht addRule (str, preprob)
addval	varIndex info	entspricht addValue (varIndex, info)
addvar	name <b,n,u,i> initval*	entspricht addVar (name, type, initValues)
batch	fileName	führt eine Batch-Datei mit SPECTER-Kommandos aus
begin	[threshold [mli]]	entspricht beginIteration (threshold, mli)
deleterule	ruleIndex	entspricht deleteRule (ruleIndex)
deleteval	varIndex val	entspricht deleteValue (varIndex, val)
deletevar	varIndex	entspricht deleteVar (varIndex)
end		entspricht endIteration ()
enum	<0,1,2>	entspricht enumerateVars
evi	(var val)*	entspricht assignEvidence
export		entspricht exportFile (logFenster)
expval	varIndex	entspricht expectedValue (varIndex)
help	[command]	listet alle Befehle oder gibt Hilfe zum Befehl *command*
i	[count]	entspricht stepIteration () count mal aufgerufen
import	fileName	entspricht importFile (file mit Namen fileName)
is		entspricht getIterationStatus ()
it		führt einen vollständigen Iterationsprozeß durch
leginfo		entspricht getLEGInfo
r	[ruleIndex]	listet alle Regeln oder die angegebene Regel
rebuild		entspricht rebuild
reinit		entspricht reinit
reset		entspricht reset
ruleact	ruleIndex	entspricht toggleRuleActivity ()
setrulep	ruleIndex preprob	entspricht setRulePreProb (ruleIndex, preprob)
setutility	varIndex utility	entspricht setVarUtility (varIndex, utility)
setvarenum	varIndex enum	entspricht setVarEnum (varIndex, enum)
thick	var0 var1	entspricht graphEdgeThickness (var0, var1)
v	[varIndex]	listet alle Variablen oder die angegebene Variable
varp	[varIndex]	entspricht valueProb (varIndex, ...)
varrules	varIndex [conc]	entspricht getVarRules (varIndex, conc)

Anhang F: Data-Dictionary

Das *Data-Dictionary* (gemäß FUSION) erläutert relevante Begriffe aus Analyse und Entwurf, sowie im Text auftauchende Fachtermini. Im gegebenen Rahmen ist ein nach [Col94] vollständiges *Data-Dictionary* leider nicht möglich, da dies wesentlich mehr Seiten erfordern würde. Ich habe mich daher auf die Definition der genannten Datentypen und einiger Grundbegriffe der Theorie beschränkt.

Begriff	Erläuterung
double *Datentyp*	Reelle Zahlen (die kleinste positive in JAVA darstellbare Zahl ist ca. 4.94e-324)
EdgeType *Datentyp*	{ none, undirected, arrow }, die Art einer Kante in den Variablengraphen
Entropie *Fachterm*	Im allg. Sinne ein Maß für Ordnung in einem System. In SPIRIT ein Wert, der den Zustand des gelernten Wissens in einer Wissensbasis angibt.
EnumMethod *Datentyp*	{ maxCS, minCF, minCS, minCW }, die verfügbaren Enumerations-Heuristiken
Evidenz (-zuweisung) *Fachterm*	Wird einem Variablenwert Evidenz zugewiesen, so wird er als "sicher", "wahr" oder "beobachtet" angenommen, d.h. er erhält die Wahrscheinlichkeit 1.0
GraphType *Datentyp*	{ undirected, mixed }, die beiden Variablengraphen
int *Datentyp*	Ganze Zahlen, gemäß JAVA aus dem Intervall $[-2^{31}..2^{31}-1]$.
Iterationsprozeß *Fachterm*	Ein iteratives Verfahren zum Lernen der Regeln in eine Wissensbasis. Die Regeln werden nacheinander gelernt, bis ein Abbruchkriterium erfüllt ist. Nach jedem Lernschritt ist die zuletzt gelernte Regel gültig (d.h. geforderte und tatsächliche Wahrscheinlichkeit stimmen überein). Diese Gültigkeit kann durch einen weiteren Lernschritt mit einer anderen Regel allerdings aufgehoben werden, so daß ein iteratives Wiederholen der Lernschritte zur Annäherung erforderlich ist.
Junctor *Datentyp*	{ conjunction, disjunction }, Verknüpfungen in SPIRIT-Regeln
KBECode *Datentyp*	Einer der in Anhang C definierten Fehlercodes (in KBE.code gespeichert).
Konfiguration *Fachterm*	Eine Belegung aller Variablen einer LEG. Enthält eine LEG die Variablen $v_1,...,v_n$, so gibt es $\|v_1\| * ... * \|v_n\|$ Konfigurationen, nämlich (0,...,0) bis $(\|v_1\|-1,...,\|v_n\|-1)$.
LEG (local event group) *Fachterm*	Eine Tabelle von Wahrscheinlichkeiten über das Kreuzprodukt über eine Menge von Variablen, also für jede *Konfiguration* ein Wert. Die LEGs bilden eine azyklische Struktur, den LEG-Wald, der den aktuellen Wissensstand repräsentiert.
Regel *Fachterm*	Eine logische Aussage über diskrete Variablen, die mit einer Wahrscheinlichkeit gewichtet wird. Regeln bestehen i.a. aus Prämisse und Conklusio.
RuleActivity *Datentyp*	{ active, activable, passive }, die Aktivierungszustände einer Regel
String *Datentyp*	Zeichenketten beliebiger Länge. In JAVA werden Strings in einer eigenen Klasse verwaltet. Die einzelnen Zeichen sind *Unicode*-Characters mit 16 Bit Breite.
Variable *Fachterm*	Eine diskrete Menge von Zuständen. Es gibt Boolean-, Nominal-, Number- und Interval-Variablen mit jeweils unterschiedlichen zulässigen Operationen.
VarType *Datentyp*	Typ einer SPIRIT-Variable. { boolean, nominal, number, interval }
Vector *Datentyp*	Eine dynamisch erweiterbare Liste von Objekten (in vielen Programmiersprachen und in [Col94] auch Collection genannt). Oft wird zu einem Vektor angegeben, von welcher Klasse die enthaltenen Objekte sind (z.B. Vector of Rule). Im Gegensatz zu Vektoren sind *Arrays* (die mit [] gekennzeichnet sind) nicht dynamisch erweiterbar.
Wissensbasis *Fachterm*	Eine Menge von Variablen, Regeln und einer LEG-Struktur zur Speicherung des aktuellen Wissensstandes. Eine Wissensbasis kann Regeln lernen, Variablen Evidenz zuweisen und Anfragen beantworten. Dadurch kann sie probabilistische Abhängigkeiten zwischen Variablen aufdecken.

Literaturverzeichnis

Zu den Themen FUSION und Software-Engineering:

[Col94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F.Hayes, and P. Jeremaes. 1994.
Object-oriented Development: The Fusion Method.
Eaglewood Cliffs, NJ: Prentice-Hall.

[Mal95] R. Malan, Reed Letsinger, Derek Coleman. 1995.
Object-oriented Development at Work: Fusion in the real World.
Eaglewood Cliffs, NJ: Prentice-Hall.

[Som96] I. Sommerville. 1996.
Software-Engineering, fifth Edition.
Wokingham, England: Addison-Wesley Publishing Company.

Zur Theorie der probabilistischen wissensbasierten Systeme (insbes. SPIRIT):

[JJ94] Finn V. Jensen, Frank Jensen. 1994.
Optimal Junction Trees.
Proc. 10th conference on Uncertainty in Artificial Intelligence, 360-366, Morgan Kaufmann Inc.

[JOA90] F. V. Jensen, K. G. Olesen, S. K. Andersen. 1990.
An Algebra of Bayesian Belief Universes for Knowledge-Based Systems.
Networks, Vol. 20 (S.637-659), John Wiley & Sons, Inc.

[KIMR96] G. Kern-Isberner, C.-H. Meyer, W. Rödder. 1996.
Analyse medizinisch-soziologischer Daten mittels eines probabilistischen Expertensystems.
Proc. Symposium on Operations Research (SOR95), Passau (1996), 347-352.

[Kjæ90] U. Kjærulff. 1990.
Triangulation of Graphs - Algorithms giving small total State Space.
Technical Report R90-09, Dept. of Mathematics and Computer Science, Aalborg University

[MR92] C.-H. Meyer, W. Rödder. 1992.
Propagation in Inferenznetzen unter Berücksichtigung des Prinzips der minimalen relativen Entropie.
Proc. 21. Jahrestagung der DGOR 1992, 446-453. Aachen: Springer (1993).

[Pea88] J. Pearl. 1988.
Probabilistic Reasoning in Intelligent Systems.
San Mateo, CA: Morgan Kaufmann

[RM96] W. Rödder, C.-H. Meyer. 1996.
Coherent Knowledge Processing at Maximum Entropy by SPIRIT.
in Proc. Twelfth Conf. on Uncertainty in Artificial Intelligence 1996, Portland, Oregon, USA.

[Röd94] W. Rödder. 1994.
Symmetrical Probabilistic Reasoning in Inference Networks in Transition.
in: Operations Research, Hrsg: B. Werners, R. Gabriel; 129-169. Berlin/Heidelberg: Springer.

Zu einigen verwendeten Algorithmen:

[CLR90] T. Cormen, C. Leiserson, R. Rivest. 1990.
Introduction to Algorithms.
Cambridge, MA: The MIT Press.

[WM92] R. Wilhelm, D. Maurer. 1992.
Übersetzerbau: Theorie, Konstruktion, Generierung.
Berlin/Heidelberg: Springer.

Einführende Literatur zur Programmiersprache JAVA:

[Fla96] D. Flanagan. 1996.
Java in a nutshell.
Cambridge, Mass.: O'Reilly.

[Hof96] A. van Hoff, et. al. 1996.
Hooked on Java.
Reading, MA: Addison-Wesley Publishing Company.

[New96] A. Newman, et. al. 1996.
Special Edition using JAVA.
Indianapolis, IN: Que Corporation.

[Rit95] T. Ritchey. 1995.
Java!
Indianapolis, IN: New Riders Publishing.

Zusammenfassung

Die Arbeit beschreibt die objektorientierte Neuentwicklung des Projektes SPIRIT. Bei diesem handelt es sich um ein wissensbasiertes System, das Expertenwissen aus probabilistischen Regeln über diskrete Variablen erhält und komplexe Anfragen beantworten kann. Aufgabe war die Erstellung einer flexibel einsetzbaren und effizienten Klassenbibliothek zur Realisierung solcher Wissensbasen in der Programmiersprache JAVA. Demnach enthält die Arbeit eine Vielzahl von Algorithmen und Datenstrukturen zur Umsetzung der theoretischen Konzepte.

Beim Entwicklungsprozeß wurde die objektorientierten Methode FUSION verwendet. Nach der Definition der Anforderungen wurden in der *Analysephase* verschiedene Modelle entwickelt, die das geforderte, nach außen sichtbare Systemverhalten darstellen. Aufbauend auf diesen Modellen wurden in der Entwurfsphase Software-Strukturen zur Umsetzung der Anforderungen herausgearbeitet und in der Implementierungs- und Test-Phase in ein JAVA-Programm umgesetzt. Hierbei entstanden auch einfache graphische Oberflächen für SPIRIT.

Der Text enthält neben den Ergebnissen der einzelnen Modellierungsschritte auch eine Beschreibung der Vorgehensweise nach FUSION. Dabei werden Vor- und Nachteile dieser Entwicklungsmethode aufgezeigt. Im wesentlichen wird FUSION für ein solches Projekt als geeignet befunden, es wird allerdings dargestellt, wann und warum von der üblichen Bearbeitungsmethode abgewichen wurde. Insbesondere wurde zwischen Analyse und Entwurf ein sehr komplexer Zwischenschritt eingeführt und recht früh auf Aspekte der späteren Implementierung geachtet. Dadurch konnte die eigentliche Codierung in sehr kurzer Zeit durchgeführt und eine hohe Konsistenz zwischen Entwurfsdokument und Quelltext erreicht werden.

Zudem wurde die Programmiersprache JAVA auf Tauglichkeit für ein softwaretechnisch entwickeltes Projekt untersucht. Dabei erwies sie sich als sehr geeignet, da sich die Modelle des Entwurfsdokumentes ohne größere Probleme umsetzen liessen. Die entstandene neue Version von SPIRIT konnte die in C++ entwickelten Vorgängerversionen bei den wichtigsten Teilaufgaben bzgl. Performance (teilweise deutlich) abhängen.