Entwicklung eines regelbasierten GUI Validators

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science in Angewandter Informatik

(Bachelor of Science in Applied Computer Science)

vorgelegt im

Studiengang **Angewandte Informatik**

des Fachbereiches Elektrotechnik und Informatik

der Fachhochschule Münster

Von

Andre Heptner

Betreuer/Prüfer:

Prof. Dr. rer. nat. Nikolaus Wulff

Prof. Dr.-Ing. Uvo Hölscher

Münster, Februar 2007

Zusammenfassung / Abstract

Die vorliegende Arbeit untersucht die Fragestellung, ob es möglich und praktikabel ist eine deterministische, vorwärts verkettete, regelorientierte Inferenz Maschine zur automatischen Überprüfung eines GUI-Prototypens hinsichtlich ergonomischer Regeln einzusetzen.

Diese Arbeit stellt den Versuch dar, die vorherigen Arbeiten bestehend aus einer Normenuntersuchung nach ergonomischen GUI Regeln und einem Proof-of-Concept-Software-GUI-Layouter miteinander zu kombinieren, um eine automatische Regelüberprüfung auf einer Prototypzeichnung einer Benutzeroberfläche durchzuführen.

Schwerpunkt der Entwicklung lag in dem Entwurf einer XML Regel Syntax, der dazugehörigen XSD / XML Regel Grammatik Dateien, der Entwurf der beiden Regel Parsing Prozessen, und das Testen der Möglichkeiten der Inferenzmaschine hinsichtlich Umsetzung einzelner ergonomischer Regeln.

Keywords: Rule, Rules Engine, wissensbasiertes, regelbasiertes System, Inferenz Maschine, Regel Syntax, Parser, XML, Ergonomie, GUI

Ehrenwörtliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Arbeit ist noch nicht veröffentlicht worden. Sie ist somit weder anderen Interessenten zugänglich gemacht, noch einer anderen Prüfungsbehörde vorgelegt worden.

Münster, 13. Februar 2007

Andre Heptner

Danksagung

An dieser Stelle möchte ich die Gelegenheit wahrnehmen, mich bei allen zu bedanken, die mir während der Entstehung dieser Arbeit und während meines Studiums durch ihre Unterstützung hilfreich zur Seite standen.

Für die Ermöglichung dieser Bachelorarbeit möchte ich mich bei meinem Betreuer Prof. Dr. Nikolaus Wulff bedanken für die hilfreiche, lehrreiche und sehr angenehme Unterstützung und die Stellung dieses sehr interessanten Themas.

Meinem Betreuer in der Firma Use-Lab GmbH Herrn Prof. Dr. Uvo Hölscher für das Vertrauen in meine Arbeit und in der Umsetzung.

Meinem guten Freund Dipl. Betriebswirt Steffen Thalmann für das Korrekturlesen und mich mit Anregungen und konstruktiver Kritik unterstützt zu haben.

Mein besonderer Dank geht an meine Eltern Erika und Alfred ohne deren große Unterstützung mein Studium nicht möglich gewesen wäre.

Meiner Lebensgefährtin Lisa danke ich sehr für ihre Ausdauer und Geduld.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abb.	Abbildung
AI	Artificial Intelligence
API	Application Programming Interface (eine Programmierschnittstelle)
bspw.	beispielsweise
bzw.	Beziehungsweise
DFKI	**D**eutsches **F**orschungszentrum für **K**ünstliche **I**ntelligenz
DTD	Document Type Definition
et al.	et alteri oder et alii (und andere)
etc.	et cetera
FAW	**F**orschungsinstitut für **a**nwendungsorientierte **W**issensverarbeitung
FORWISS	**For**schungszentrum für **wis**sensbasierte **S**ysteme
GUI	**G**raphical **U**ser **I**nterface
HTML	**H**yper**t**ext **M**arkup **L**anguage
HTTP	Hypertext Transfer Protocol
KI	Künstliche Intelligenz
LAN	**L**ocal **A**rea **N**etwork
MIT	**M**assachusetts **I**nstitute of **T**echnology; Universität in Cambridge, USA
PDP	**P**arallel **D**istributed **P**rocessing (Com**p**uting)
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
URL	**U**niform **R**esource **L**ocator
W3C	World Wide Web Consortium
WAN	**W**ide **A**rea **N**etwork
WWW	World Wide Web
XAML	e**X**tensible **A**pplication **M**arkup **L**anguage
XML	e**X**tensible **M**arkup **L**anguage
XSD	XML Schema Definition
z.B.	zum Beispiel

1 Einleitung / Einführung

Informationsverarbeitungsprozesse lassen sich automatisieren, wenn die benötigten Daten maschinell verfügbar gemacht werden können und die Verarbeitungsschritte sich algorithmisch beschreiben lassen. Ein Algorithmus setzt sich dabei aus vielen sequentiellen Einzelschritten zur Verarbeitung der Eingabedaten zusammen, bis die gewünschten Ergebnisse vorliegen. Bevor ein Informationsverarbeitungsprozess aber automatisiert ablaufen kann, muss er in diese Einzelschritte zerlegt und mit Hilfe einer Programmiersprache präzise beschrieben werden. Lässt sich der Prozess nicht in solchen Einzelschritten formal beschreiben, dann ist er auch nicht automatisierbar, und auch der leistungsfähigste Computer ist dann unfähig, die Aufgabe zu lösen.

Menschliche Mitarbeiter lösen solche Aufgaben dennoch, indem sie aufgrund ihrer persönlichen Erfahrungen assoziieren, eine oder mehrere Lösungsstrategien auswählen, mit Kollegen oder Vorgesetzen diskutieren, urteilen und schließlich entscheiden. Sie benutzen also menschliche Intelligenz und Erfahrungen in einem jeweiligen Fachgebiet, um eine Problemlösung zu finden.

Die bereits viele Jahre andauernden Forschungen zur Künstlichen Intelligenz haben als ein konkretes Anwendungsgebiet Expertensysteme hervorgebracht. Ein Expertensystem verfügt über eine Wissensbasis auf einem speziellen Fachgebiet, sowie über Problemlösungstechniken, die von Experten als geeignet zur Lösung von Problemen des ausgewählten Fachgebiets eingestuft wurden [Vgl. SCHNUPP 1988].

Bei der softwaretechnischen Gestaltung von Expertensystemen stehen nicht die heutigen Möglichkeiten der Computertechnik mit ihrer sequentiellen und objektorientierten Verarbeitungstechnik im Vordergrund. Stattdessen wird der Versuch unternommen die menschliche Fähigkeit Probleme zu lösen zu simulieren, um auf maschinellem Wege ähnlich gute, wenn nicht bessere Ergebnisse, wie bei einem durchschnittlichen menschlichen Experten zu erzielen [Vgl. SCHNUPP 1988].

Expertensysteme haben bereits lange das Experimentierstadium verlassen; etliche professionelle und kommerziell genutzte Systeme zeigen dieses.

Die Entwicklung der theoretischen Grundlagen von Expertensystemen war und ist Forscheraufgabe, die Entwicklung kommerziell einsetzbarer Produkte ist die interdisziplinäre Aufgabe von Softwarespezialisten und „Knowledge Engineers“ - also Experten auf einem Sach- oder Fachgebiet, für das versucht wird ein Expertensystem zu entwickeln [Vgl. SCHNUPP 1988].

1.1 Motivation

Die Grundidee des Gesamtprojektes mit dem Arbeitstitel „DIN-Machine“ ist es Entwicklern von Benutzerschnittstellen für Medizingeräte bei den Gestaltungs- und Spezifikationsprozessen zu unterstützen und diesem Personenkreis ergonomisches Wissen zur Gestaltung, in Form von Grundsätzen und Normen, zur Verfügung zu stellen.

Dazu entstand die Idee ein Softwareprogramm zu entwickeln, das die Entwickler bereits während des Spezifikationsprozesses massiv unterstützt, aber dennoch so zurückhaltend sein sollte, dass der Anwender während des kreativen Arbeitsprozesses nicht zu sehr eingeschränkt oder zu häufig unterbrochen wird. Zum einen sollte diese Software ein Layouting bzw. Prototyping-Tool umfassen, das einfach zu bedienen ist und mit dem der Anwender in der Lage ist Prototypen von Layouts zu erstellen. Zum anderen ein beratendes Expertensystem, das dafür sorgt, dass die Entwickler bereits in diesem frühen Stadium des Entwurfs ergonomische Richtlinien einhalten.

Nachdem allgemeine Prozesse zur Entwicklung von grafischen Oberflächen hinreichend untersucht wurden, wurde eine interdisziplinäre Arbeitsgruppe bestehend aus Designern, Ingenieuren und Informatikern gebildet. Während Studenten des Fachbereichs Design Studien über den Umfang und das Aussehen der DIN-Machine-Software erstellten, wurden Ingenieursseitig die beteiligten Prozesse und die Projektplanung dokumentiert. Zudem wurde eine umfangreiche Literaturrecherche nach ergonomischen Regeln und Normen durchgeführt. Parallel wurde durch einen Informatikstudenten eine erste Proof-of-Concept-Software erstellt, die es ermöglichte Oberflächen-Prototypen zu erstellen und deren Bedienbarkeit durch einen Benutzer zu Testzwecken interaktiv navigierbar zu machen.

Die Ergebnisse aus der Literaturrecherche und des GUI-Layouter-Prototypen sollen nun in einem weiteren Modul miteinender kombiniert werden, sodass erstellte Benutzeroberflächen nach vorhandenen Ergonomieregeln und –normen untersucht werden und gegebenenfalls Verbesserungsvorschläge aufgezeigt werden. Womit im folgenden Abschnitt die Zielsetzung dieser Arbeit beschrieben werden kann.

1.2 Zielsetzung der Arbeit

Aus der unmittelbar vorher entstandenen Bachelorarbeit über eine GUI-Prototyp-Layouter-Software [BA FStolze] liegt eine Prototypzeichnung einer grafischen Benutzeroberfläche eines einfachen (medizinischen) Gerätes vor. Eine solche

Prototypzeichnung wird nach der Java-Bean-Konvention [JAVABEAN 1994] serialisiert (d. h. XML kodiert) und liegt in persistenter Form (d. h. als Datei) vor.

Die Aufgabe besteht darin, eine Software-Lösung zu entwickeln und sie daraufhin zu untersuchen, ob eine automatische Analyse und Bewertung eines Java-Bean XML Speicherformates praktikabel und umsetzbar ist.

Dazu ist es in dieser Arbeit nötig ein Verfahren zu entwickeln, welches speziell festgelegte ergonomische Richtlinien in computerverständliche Form bringt. Richtlinien, die hauptsächlich in Form vorliegen, wie nur ein menschlicher Experte sie anwenden könnte. Dabei bot es sich an, für eine in Teilen bereits existierende Inferenzmaschine eine Regelsprache zu entwerfen. Darauf abgestimmt soll ein Parser zum einlesen der Regeln entwickelt werden.

In diesem Zug wird der Inferenzmechanismus ganz allgemein für die Verarbeitung von Ergonomieregeln getestet. Damit wird die Frage geklärt, ob ein Inferenzmechanismus überhaupt praktikabel für diese Art von Regeln ist.

Durch diese Arbeit soll demnach in Erfahrung gebracht werden, ob, wie und welche Regeln sich mit dem Konzept eines regel- und wissensbasierten Expertensystems (einer Inferenzmaschine) eine Regelüberprüfung formulieren, durchführen und überprüfen lassen.

1.3 Aufbau der Arbeit

Die Arbeit ist im weiteren Verlauf wie folgt gegliedert:

Im Kapitel 2 werden die Grundlagen der künstlichen Intelligenz beschrieben. Es wird der Begriff „Künstliche Intelligenz“ erläutert und wie sich aus künstlicher Intelligenz wissensbasierten Systeme und Expertensysteme gebildet haben.

Das Kapitel 3 beschäftigt sich speziell mit den Grundlagen, Komponenten, Arbeits- und Funktionsweisen von Expertensystemen. Zudem werden Methoden zur Wissensdarstellung aufgezeigt; also die Frage geklärt, wie wird bekanntes, vorhandenes Wissen formuliert, sodass es von Computerprogrammen verstanden und verarbeitet werden kann.

Die konkreten Implementierungen der Wissensbasen werden ausführlich in Kapitel 4 dieser Arbeit beschrieben. Ebenso werden in diesem Kapitel die XML-Syntax der Regelbeschreibungssprache – für die Regelbasis - und die Einlesevorgänge (Regel-Parsing-Prozesse) vorgestellt. Darüber hinaus befasst sich dieses Kapitel mit der

Entstehung einer Fakten-Basis eines zu untersuchenden Oberflächenprototypen, auf der eine Regelbasis angewendet werden kann.

Das Kapitel 5 stellt einen möglichen Prototyp einer Rules Engine Benutzeroberfläche vor, mit der man die im vorherigen Kapitel erstellten Wissensbasen in eine Inferenzmaschine hineinsteuern, den Mechanismus starten und die daraus resultierenden Schlussfolgerungen einsehen kann.

Das Kapitel 6 ist den Tests bzw. der Verifikation der Implementierungen gewidmet. Hier werden JUnit Tests für einige relevante Klassen vorgestellt.

Im letzen Kapitel sind die Ergebnisse der vorliegenden Arbeit zusammenfassend dargestellt und es wird ein Ausblick auf weitere Entwicklungsperspektiven gegeben.

2 Grundlagen von wissensbasierten Systemen

Dieses Kapitel befasst sich mit den nötigen Grundlagen und Definitionen, die die allgemeine Arbeitsweise und Problemlösungsstrategien von Künstlicher Intelligenz erläutert.

Im Anschluss soll ein geschichtlicher Abriss über das Forschungsinteresse an künstlicher Intelligenz erfolgen. Darüber hinaus soll erläutert werden, welche Gemeinsamkeiten zwischen der künstlichen Intelligenz und wissensbasierten Expertensystemen existieren.

2.1 Künstliche Intelligenz

Bevor geklärt werden kann was künstliche Intelligenz bedeutet, muss erörtert werden was allgemein unter Intelligenz verstanden wird. Dazu folgt zuerst eine Definition aus einem Wörterbuch:

Intelligenz, abgeleitet von dem lateinischen Substantiv „intelligentia“, bedeutet „die Einsicht, das Erkenntnisvermögen“ oder von dem Verb „intellegere“ „verstehen“. Das Wort Intelligenz bezeichnet im weitesten Sinne die Fähigkeit zum Erkennen von Zusammenhängen und zum Finden optimaler Problemlösungen [Vgl. http://de.wikipedia.org/wiki/Intelligenz].

Mit Künstlicher Intelligenz (KI) wird heute eine Fähigkeit von Computerprogrammen bezeichnet, komplexe Probleme der realen Welt mit logischem Kalkül, also nicht funktionsanalytisch, zu lösen. KI-Methoden unterscheiden sich von traditionellen Programmen dadurch, dass sie versuchen Prinzipien des menschlichen Verhaltens nachzubilden: Versuch und Irrtum (try and error), experimentelles Lernen, logisches Schlussfolgern. Sie gehen vorwiegend mit Texten, Symbolen und logischem Kalkül und nicht so sehr mit Funktionen der klassischen numerischen Mathematik um.

Die KI wird zur Lösung von Problemen eingesetzt, für die es keine exakten oder nur mit sehr großem Aufwand berechenbare mathematische Lösungen gibt. Es ist die Umschreibung eines Forschungsgebietes zur Entwicklung leistungsfähiger Computer bzw. Algorithmen.

2.2 Geschichtliche Entwicklung wissensbasierter Systeme

Die Geschichte der Künstlichen Intelligenz ist nicht von der Geschichte der Logik zu trennen und müsste daher in der Antike mit Aristoteles beginnen. In diesem Rahmen soll sie jedoch im engeren Sinne in Verbindung mit ihrer Realisierung auf Computern

betrachtet werden und daher ist der Anfang in den Arbeiten von Alan Turing [TURING 1950] zu sehen, der die Frage behandelte: „Können Maschinen denken?“. In diesem Zusammenhang stellte er neben dem so genannten Turing-Test auch ein Konzept über die Programmierbarkeit diskreter Maschinen, die in der Lage sind, ihr eigenes Programm zu verändern, also zu lernen.

Im Jahr 1956 war die Grundsteinlegung der Entwicklung von „wissensbasierten Systemen“ am MIT durch John McCarthys und Marvin Minsky. Damals begannen die Forschungsarbeiten zur Artificial Intelligence (AI), heute im deutschsprachigen Raum bekannt als Künstliche Intelligenz.

Seither kann man fünf Phasen der Entwicklung unterscheiden [Vgl. KURBEL 1992]:

Die Gründungsphase von **1956 – 1965** war gekennzeichnet durch die ersten Ansätze der symbolischen, nicht numerischen Informationsverarbeitung, z. B. symbolische Berechnung unbestimmter Integrale. Der Computer berechnete die Stammfunktion (als Umkehrung der Ableitung) einer Funktion.

Die Zielsetzung in dieser Periode lag in der Findung einer zentralen Problemlösungsformel, bestehend aus der zielgerichteten Durchforstung des Suchraumes. Hierzu wurden leistungsstarke Heuristiken entwickelt, um eine effizientere Suche zu gestalteten.

Mit der listenorientierten Programmiersprache LISP schuf John McCarthy 1960 am MIT einen Meilenstein zur Erforschung der Künstlichen Intelligenz. Der Ausdruck „artificial intelligence“ wird Marvin Minsky vom MIT zugeschrieben, der ihn erstmals in seinem Artikel „Steps towards Artificial Intelligence“ 1961 verwendete.

Joseph Weizenbaum schrieb 1964 das inzwischen legendäre und berühmte Programm ELIZA, welches die Rolle eines passiven Gesprächstherapeuten karikierte und Dialoge wie in der folgenden Abbildung 2-1 führte [Vgl. WEIZENBAUM 1966].

ELIZA arbeitet nach dem Prinzip, Aussagen des menschlichen Gesprächspartners in Fragen umzuformulieren und so eine Reaktion auf die Aussage zu simulieren.

Benutzer: *„Ich habe ein Problem mit meinem Auto."*
ELIZA: *„Warum, sagen Sie, haben Sie ein Problem mit Ihrem Auto?"*

Außerdem ist es in der Lage, Schlüsselwörter (z.B. *Vater*) zu erkennen und darauf zu reagieren, z. B.

Benutzer: *„Ich habe ein Problem mit meinem Vater."*
ELIZA: *„Erzählen Sie mir mehr über ihre Familie!"*

In diesem Fall hat das Programm analysiert, dass *Vater* zu *Familie* gehört und dadurch eine sinnvolle Erwiderung gegeben.

Abb. 2-1: Prinzip und Beispieldialog ELIZA

Mittlerweise existieren im Internet etliche Derivate von ELIZA in den meisten bekannten Programmiersprachen [Vgl. http://de.wikipedia.org/wiki/Eliza#Weblinks], unter anderem eines als Java-Applet im Webbrowser lauffähiges ELIZA unter http://bs.cyty.com/menschen/e-etzold/archiv/science/rat.htm.

In der romantischen Epoche von **1966 – 1974** begannen Forschergruppen an führenden amerikanischen Universitäten zentrale Fragestellungen systematisch zu beantworten. Dazu zählten unter anderem die Sprachverarbeitung mit dem Computer, automatisches Problemlösen, z. B. automatisches Beweisen von mathematischen Sätzen und visuelle Szenenanalysen.

1970 wurde das erste kommerziell einsetzbare KI-Produkt an der Stanford University entwickelt, das Expertensystem MYCIN. Es sollte Physiologen in der Diagnose von Infektionskrankheiten unterstützen. Daraus bildete sich durch EMYCIN (Empty MYCIN) die erste leere Expertensystemschale, die für alle Arten von Regeln und Problemdaten, die mit LISP beschrieben werden konnten, universell einsetzbar war.

1972 entwickelte Alain Colmerauer die KI-Sprache PROLOG, die sich in den folgenden Jahren im europäischen Raum für die Formulierung von KI-Problemen durchsetzte.

Mitte der siebziger Jahre (**1975 – 1980**) begann die dritte Phase mit dem Entwurf erster integrierter Robotersysteme und expertenhaft problemlösender Systeme.

Seit Beginn der achtziger Jahre läuft die vierte Phase (moderne Epoche) (**1981 – 1995**), die vor allem durch eine umfassende Mathematisierung des Gebietes, eine Präzisierung des Konzepts der Wissensverarbeitung und das Aufgreifen neuer Themen gekennzeichnet ist, wie

- Verteilte Künstliche Intelligenz (distributed AI), d. h. das Wissen ist auf mehreren Rechnern oder Wissensbasen aufgeteilt und ein Mastersystem zieht dieses Wissen zusammen
- Neuronale Netze (NN), das sind im Computer nachgebildete Strukturen des menschlichen Gehirns, also der Neuronen-Netze
- Die Verarbeitung verschiedener Qualitätsstufen von Informationen, wie z. B. die Verarbeitung unscharfer Information. (Stichwort: Fuzzy-Logik)

1981 kündigten die Japaner ein nationales Forschungsprogramm zur KI unter Beteiligung bedeutender japanischer Computerfirmen an: the Fifth Generation Program. Programmierbasis wurde die Sprache PROLOG.

1982 bildete sich in den USA eine interdisziplinäre Forschergruppe aus Mathematikern, Physikern, Informatikern und Psychologen, die sich mit Parallelverarbeitung in neuronalen Rechnernetzen beschäftigte (Parallel Distributed Processing).

Fünf Bundesländer in Deutschland richteten Forschungsinstitute für KI ein, die Grundlagenforschung betrieben und immer mehr dazu übergingen, in Firmen Lösungen mit Hilfe der Techniken der Wissensbasierten Systemen zu erzeugen. Unter anderem Bayern mit dem FORWISS, Baden Württemberg mit dem FAW in Ulm und das Saarland mit dem DFKI.

In der postmodernen Epoche (**seit 1995**) gehörten wissensbasierte Komponenten zum Standard und wurden als ein kleines, aber sehr wichtiges Modul innerhalb größeren Softwarelösungen nicht mehr explizit erwähnt. Einen neuen Boom erlebten wissensbasierte Systeme Ende der neunziger Jahre durch das Internet in Form von Einkaufs- und Informationsfilter-Agenten und durch das Wissensmanagement.

Auch entstanden Rechner mit Nicht-Von-Neumann-Architekturen, wie der Neurocomputer, ein Rechner der auf Neuronalen Netzen fußt.

Die bisherigen Entwicklungen lassen deutliche Fortschritte erkennen und durch den Aufschwung werden neue Ideen zur Verwendung und Weiterentwicklung hervorgerufen. Somit wird auch diese Arbeit einen Teil zur zukünftigen Entwicklung von wissensbasierten Systemen beitragen.

2.3 Problemlösen mit allgemeinen KI-Systemen

KI-Systeme werden auch als Produktionssysteme, regelbasierte Systeme oder patterndirected inference systems bezeichnet. Sie bestehen im Allgemeinen häufig aus drei Komponenten: Wissensbasis (globale Datenbasis), Produktionsregeln und Kontrollsystem (Zielbedingungen).

Die Wissensbasis beschreibt einen, mehrere oder alle Zustände eines Systems auf dem Weg zu einer Lösung. Der Anfangszustand wird dabei durch eine Anfangsdatenbasis (initial global database) beschrieben.

Die Produktionsregeln operieren auf dieser Wissensbasis und verändern diese, falls die Daten entsprechende Bedingungen erfüllen. Können mehrere Regeln angewendet werden, so wählt das Kontrollsystem die am meisten geeignete Regel nach einer bestimmten Strategie aus. Das Kontrollsystem ist für die Überprüfung einer Zielbedingung zuständig [BRUNS 1990].

Die meisten bekannten KI-Systeme verwenden einen zentralen Prozessor, der die Regeln in einem sequentiellen Algorithmus abarbeitet. Die Systeme werden daher auch als regelbasiert und explizit bezeichnet. Diese Struktur hat einige Schwachstellen. Viele menschliche Tätigkeiten lassen sich nicht durch explizite Regeln beschreiben (z. B. das Erkennen eines Bekannten in einer Menge), sondern erfolgen ganzheitlich. Ein KI-System, das diese Fähigkeit ausklammert, wird für große Problemklassen unzureichend bleiben. Dreyfus & Dreyfus [DREYFUSS 1987] haben diesen Schwachpunkt in einer allgemeinen Kritik an der KI-Forschung hervorgehoben.

2.4 Methodologien der Wissensverarbeitung

Es gibt mehrere verschiedene Vorgehensweisen, um intelligente Eigenschaften in Systemen zu realisieren. Man unterscheidet je nach Ausgangspunkt zwischen

- Symbolischer Wissensverarbeitung
- Subsymbolischer Wissensverarbeitung

Bei der **subsymbolischen** Wissensverarbeitung soll mittels Neuronaler Netze intelligentes Verhalten erzwungen werden.

Dabei werden Netzwerke aus Schaltelementen mit geeigneten Verbindungen zwischen ihnen aufgebaut. Jedes Element kann im Prinzip mit jedem anderen Element verbunden sein. Jedes Element hat eine bestimmte Anzahl von Eingängen und genau einen Ausgang. Die Verbindungsstärke wird durch einen Gewichtsfaktor beschrieben. Das Element verhält sich wie ein Schwellenwert, das genau dann ein Signal an seinen

Ausgang weiterleitet, wenn an den Eingängen Signale anliegen, deren gewichtete Summe einen definierten Schwellenwert überschreitet [Vgl. ROJAS 1996].

Bei der **symbolischen** Wissensverarbeitung wird versucht, intelligente Phänomene auf der Begriffsebene zu lösen. Zum Beispiel werden zwei oder mehr Aussagen Symbole zugeordnet und versucht ihre Beziehung zueinander in Implikation, Äquivalenz oder Negationssymbolik auszudrücken.

Das Problem wird von oben her (top-down) angegangen. Man nimmt dabei an, dass sich eine symbolische Sicht der Begriffe, Wörter und sprachliche Gebilde sowie deren Bedeutung finden lässt. Weiterhin wird angenommen, dass sich das menschliche rationale Denken auf dieser symbolischen Ebene simulieren lässt.

Gegenstand der symbolischen Wissensverarbeitung sind nicht das Gehirn und Prozesse des Abrufs von Gedächtnisbesitz - wie bei neuronale Netzen -, sondern vielmehr die Bedeutung, die sich einem Prozess mit Hilfe symbolischer Beschreibung zuordnen lässt.

Durch diese Vorgehensweise werden logische Schlüsse mit Hilfe regelbasierter Systeme gezogen, also von Systemen, bei denen das Expertenwissen in Regelform vorliegt.

Logik ist ein fundamentales Werkzeug für Analysen auf der Wissensebene. Dementsprechend werden Logikformalismen vielfach benutzt, um eine explizite Menge von Überzeugungen (für wahr gehaltene Aussagen) zu beschreiben. Eine solche Menge von Überzeugungen, ausgedrückt in einer (symbolischen) Repräsentationssprache, wird **Wissensbasis** bezeichnet.

Das zentrale Paradigma der symbolischen KI wurde mit der Beschreibung des intelligenten Agenten 1972 in [NEWELL 1972] formuliert: Der intelligente Agent verfügt über Sensoren, zur Wahrnehmung von Informationen aus seiner Umgebung, und über Aktuatoren, mit denen er die äußere Welt beeinflussen kann. Bevor der Agent in der Welt handelt und sich dadurch möglicherweise irreversibel verändert, manipuliert er eine interne Repräsentation der Außenwelt, um den Effekt alternativer, ihm zur Verfügung stehender Methoden abzuwägen. [Vgl. NEWELL 1972]

Die besten Ergebnisse in der Simulation intelligenten Verhaltens wurden bei Projekten erzielt, in denen symbolische und subsymbolische Wissensverarbeitung kombiniert wurden.

2.5 Anwendungsgebiete der Künstlichen Intelligenz

Das Gebiet der Künstlichen Intelligenz umfasst Bereiche, in denen besonders komplexe Aufgaben behandelt werden, für die es entweder mehrdeutige oder nur mit sehr großem Aufwand berechenbare eindeutige Lösungen gibt.

Das Fachgebiet der künstlichen Intelligenz zerfällt in eine Reihe von Unterbereichen, die natürlich – vor allem in den praktischen Anwendungen – recht eng miteinander verwoben sind. Die wichtigsten Unterbereiche werden in den folgenden Absätzen kurz umschrieben [Vgl. BRUNS 1990].

Lösungssuche bei kombinatorischen Problemen

Die Lösungssuche bei kombinatorischen Problemen ist nicht mit der Suche in Datenbanken zu verwechseln. Vielmehr handelt es sich um das schrittweise Erzeugen eines möglichst optimalen Lösungsweges für ein Problem. Dazu zählen z. B. das Herausfinden eines möglichst kurzen Weges von A nach B, oder die Entwicklung von Programmen, die bei intellektuell anspruchsvollen Spielen wie Schach, Dame etc. das Niveau guter menschlicher Spieler erreichen oder gar übertreffen.

Verarbeitung natürlicher Sprache

Die Verarbeitung und das Verstehen von natürlicher Sprache ist eine sehr große Herausforderung. Ein Computersystem, das eine Meldung in natürlicher Sprache verstehen will, muss ähnlich dem Menschen über das Kontextwissen und die Fähigkeit verfügen, um damit aus einer Meldung Schlussfolgerungen zu ziehen. Die KI-Forschung steht hier erst am Anfang. Bescheidene Erfolge sind bereits erzielt worden. Diese erlauben es, eingeschränkte Wortmengen und Grammatiken zu interpretieren und so eine Kommunikation zwischen Mensch und Maschine zu ermöglichen.

Mustererkennung

Mustererkennung ist der Versuch, Maschinen mit einer Sensorik auszustatten, um die Umwelt zu erkennen. Hierunter fallen in erster Linie Sprach- und Bilderkennung. Aktuelle Beispiele sind das Erkennen von Bildern und Szenen, in einem statischen Foto oder in einem Fernsehbild, etwa zur Stauüberwachung oder für Ampelsensoriken.

Robotik

Fragestellungen aus der Robotertechnik hatten einen entscheidenden Einfluss auf die Entwicklung der KI. Wenn es nicht nur um die deterministische Steuerung von Roboterachsen in einer starren Umgebung geht, sondern um ein mehr oder weniger autonomes, intelligentes Aufgabenlösen in einer sich verändernden Umgebung, sind komplexe Methoden der Modellbildung von Wirklichkeit, Handlungsplanerzeugung und Erfolgsüberwachung erforderlich.

Maschinelles Lernen

Maschinelles Lernen handelt von Programmen, die aus ihren Erfahrungen lernen, die Hindernisse erkennen, sich merken können und die unterrichtbar sind. Konzentrierte sich der frühe KI-Weg auf die Wissenserweiterung durch Regelanhäufung von deklarativem Wissen, so versuchen neuere Ansätze auch assoziatives Wissen zu

erfassen. Dieses Ziel soll durch den Einsatz von großen künstlichen neuronalen Netzen erreicht werden.

Logik, automatisches Beweisen und Konstruieren

Logik und automatisches Beweisen beruhen auf der Umsetzung der Aussagenlogik in Computer-Algorithmen (Predicate Calculus, first-order-logic). Hiermit ist das automatische Ableiten mathematischer Beweise und logischer Schlüsse gemeint. Neu an den KI-Ansätzen ist die Möglichkeit, auch Elemente der Unsicherheit zu berücksichtigen. Die Realität ist selten durch eine einfache Modellbildung beschreibbar, die mit den exakten Regeln der Mathematik handhabbar ist. Um dieser Tatsache Rechnung zu tragen, wird in der KI bewusst das Element der Wahrscheinlichkeit eingeführt. Diese Logik, die auch unter Verwendung unsicherer Aussagen Schlussfolgerungen versucht, wird Fuzzylogik oder Quantenlogik genannt. In ihr gibt es neben den klassischen Zustandswerten „wahr“ und „falsch“ auch den Wert „unsicher“.

Schlussfolgernde Datenbank-Abfragen

Intelligente Datenbanksysteme ziehen aus ihrer Wissensbasis eigene Schlussfolgerungen und bilden Querverweise. Dadurch erlauben sie Benutzeranfragen, die in dieser Kombination nicht unbedingt bei der Eingabe eines Datensatzes berücksichtigt wurden. Stichwort: Data-Mining.

Automatisches Programmieren

Automatisches Programmieren ist das Erzeugen eines Computerhochsprachen-programms in einer jeweils höheren Sprache. Dies kann eine sehr präzise formale Sprache oder eine natürliche Sprache sein. Forschungsergebnisse dieses Gebietes gehen in die Entwicklung der Softwaretechnik ein: Programmverifikation, Compiling und Debugging. Auch die automatische Übersetzung von Texten aus einer Sprache in einer anderen Sprache ist ein Produkt solcher Forschungsergebnisse.

Spezialisierte Beratungssysteme (Expertensysteme)

Expertensysteme sind spezielle Computerprogramme, die auf einem Wissens- und Fachgebiet menschliche Experten unterstützen oder ersetzen können. Sie erfüllen allgemein gesprochen zwei Funktionen.

Erstens erlauben sie die Speicherung von Informationen über ein bestimmtes Sachgebiet in einer schrittweise erweiterbaren Wissensbasis.

Zweitens ermöglichen sie die Abfrage dieses Wissens in einer Weise, wie man es bei einem Experten dieses Faches erwarten würde.

Da das eigentliche Thema dieser Arbeit sich eben genau mit dieser Art von Systemen beschäftigt, ist den Grundlagen von Expertensystemen ein gesondertes Kapitel gewidmet.

Wenn im Folgenden also von Expertensystemen die Rede ist, dann ist zwangsläufig auch von Künstlicher Intelligenz die Rede.

3 Expertensysteme

Als Expertensysteme (XPS) wird eine Klasse von Software-Systemen bezeichnet, die auf Basis von Expertenwissen zur Lösung oder Bewertung bestimmter Problemstellungen dient. Es sind Programme, die eine besondere Softwarearchitektur besitzen und hauptsächlich zur Diagnostik, Analyse wissenschaftlicher Daten, Konstruktion (Festlegung optimaler Systemkonfiguration) und Simulation verwendet werden.

Die wesentliche Voraussetzung eines Expertensystems ist die Trennung von Steueralgorithmus und Wissensbasis. Um diese Unterscheidung zu verdeutlichen, wird in letzter Zeit auch häufig der Begriff „wissensbasiertes System“ verwendet.

Expertensysteme unterscheiden sich von konventionellen Programmen durch ihre hohe Modularität und Logikorientiertheit. Während bei letzteren nur eine Trennung zwischen Steuer- und Berechnungsalgorithmus einerseits und Problemdaten andererseits angestrebt wird, ist bei Expertensystemen grundsätzlich die Trennung zwischen Steueralgorithmus (Problemlösungsstrategie), Wissenskomponente (Regeln) und Datenbasis (Problemdaten) üblich. Die Trennung der Komponenten wird im Laufe dieses Kapitels noch weiter verfeinert werden.

3.1 Klassifizierung und Beispiele von Expertensystemen

Dieses Unterkapitel beschreibt die allgemeinen Aufgabenklassen von Expertensystemen und nennt einige bekannte realisierte Expertensysteme:

[Vgl. http://de.wikipedia.org/wiki/Expertensystem].

Dateninterpretation

Analyse von Daten mit dem Ziel einer Zuordnung zu Objekten oder Erscheinungen, insbesondere Signalverstehen.

Beispiele: Erkennung akustischer Sprache (HEARSAY), Identifizierung chemischer Strukturen anhand von Massenspektrometerdaten (DENDRAL), geologische Erkundung (PROSPECTOR), Proteinstrukturbestimmung aus Röntgendaten, Erdölbohrung, militärische Aufklärung, U-Boot-Ortung (SEPS, STAMMER).

Überwachung

Interpretation von Daten mit Aktionsauslösung in Abhängigkeit vom Ergebnis.

Beispiele: Produktionssicherung, Überwachung von Patienten in der „Eisernen Lunge“ (VM), Überwachung eines Kernreaktors (REACTOR).

Diagnose

Interpretation von Daten mit starker Erklärungskomponente.

Beispiele: vielfältig in der Medizin, zum Beispiel bei bakteriellen Infektionen (MYCIN), Rheumatologie, innere Medizin (INTERNIST), Pflanzenkrankheiten; außerdem zur Bestimmung und Lokalisation von Fehlern in technischen Systemen.

Therapie

Aktionen zur Korrektur fehlerhafter Systemzustände und Beseitigung der Ursachen (oftmals mit Diagnose gekoppelt).

Beispiele: siehe Diagnose, Fehlerdiagnose im Autogetriebe (DEX), Fehlerortung und Wartung bei Telefonnetzen (ACE), automatische Entwöhnung von Beatmungspatienten in der Intensivmedizin (SmartCare/PS).

Planung

Erzeugen und Bewerten von Aktionsfolgen zur Erreichung von Zielzuständen.

Beispiele: Versuchsplanung molekulargenetischer Experimente (MOLGEN), chemische Synthese (SECS), Finanzplanung (ROME), Produktionsplanung (ISIS), Steuerung des Flugbetriebs auf Flugzeugträgern (CAT), Handlungen autonomer Roboter (NOAH), beispielsweise Marsroboter.

Entwurf

Beschreibung von Strukturen, die vorgegebenen Anforderungen genügen.

Beispiele: unter anderem für Schaltkreisentwurf (SYN, DAA), Computerkonfiguration (R1/XCON), chemische Verbindungen (SYNCHEM), Konfiguration von Betriebssystemen bei Siemensrechnern (SICONFEX).

Prognose

Vorhersage und Bewertung erreichbarer Zustände zeitvarianter Systeme.

Beispiele: Beurteilung von Erdbebenauswirkungen (SPERIL), Erdbebenvorhersage, Hochwasservoraussage, Umweltentwicklung (ORBI).

Fazit für das Expertensystemmodul der DIN-Machine

Aus den oben beschriebenen Klassifizierungen der Anwendungsgebiete lässt sich die Erkenntnis gewinnen, dass das zu entwickelnde Rule-Engine-Modul für die DIN-Machine eine Mischung aus Diagnose, Planung und Entwurf sein wird. Denn es soll dazu dienen Prototypen von GUIs zu entwerfen und zu planen, und automatisiert eine Diagnose und Bewertung der Prototypen zu bekommen.

3.2 Komponenten eines Expertensystems

Wie bereits in der Einführung dieses Kapitels erwähnt, besteht bei einem klassischen Expertensystem die grundsätzliche Trennung der beteiligten Komponenten wie in folgender Abbildung 3-1 gezeigt wird. Ein Steueralgorithmus steuert die Wissenskomponente und wendet die dortigen Regeln auf vorhandene Problemdaten aus der Datenbasis an.

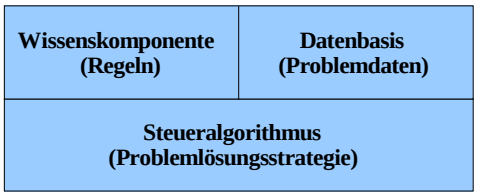

Abb. 3-1: Grundsätzlich getrennte Komponenten eines Expertensystems

In der folgenden Abbildung 3-2 ist die modernere Version [Vgl. ALTENKRÜGER 1992] eines Expertensystems zu sehen, die um einige Komponenten erweitert wurde. Darauf wurde der Übersicht halber die in Abbildung 3-1 noch getrennten dargestellten Komponenten „Regeln“ und „Problemdaten“ logisch zu einer gemeinsamen Komponente „Wissensbasis“ zusammengefasst.

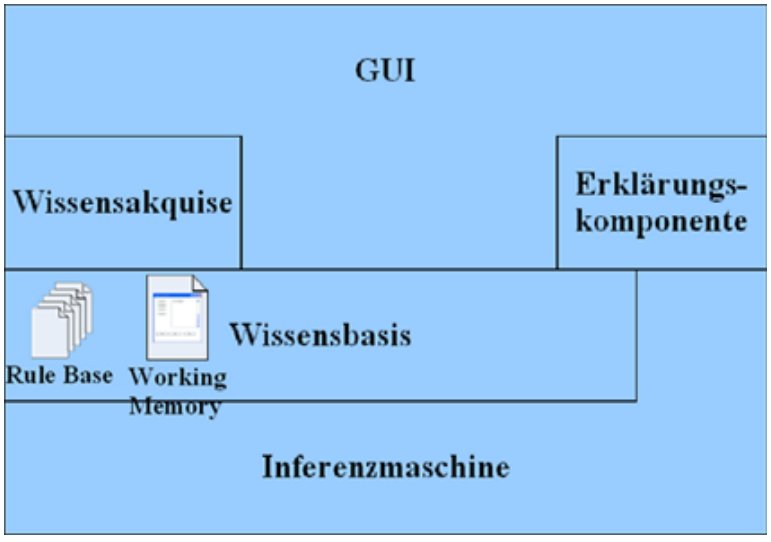

Abb. 3-2: Komponenten eines Expertensystems (erweitert)

Um eine Übersicht zu erhalten über die Funktionsweisen und Aufgaben dieser Komponenten wird nun jede einzelne buttom-up (von unten nach oben) anhand der Abbildung 3-2 erläutert.

3.2.1 Inferenzmaschine

Eine Inferenzmaschine ist eine Software aus dem Bereich der künstlichen Intelligenz, die durch Schlussfolgerung neue Aussagen aus einer bestehenden Wissensbasis ableitet. Damit sind Inferenzmaschinen Kernbestandteil von Expertensystemen und anderen wissensbasierten Systemen.

Die Inferenzmaschine (manchmal auch Schlussfolgerungssystem genannt) versucht aus den vorhandenen Regeln und Objekten dasjenige auszuwählen, das den Eingabebedingungen genügt. Man unterscheidet zwei Klassen von Inferenzmaschinen: deterministische und nichtdeterministische.

Deterministische Inferenzmaschinen berücksichtigen nur sichere Bedingungen und Aussagen. Unsicherheiten oder Wahrscheinlichkeiten können nicht verarbeitet werden. Ein Grundgerüst dieser Klasse von Inferenzmaschinen liegt bereits zu Projektbeginn vor und wurde parallel zu dieser Arbeit erweitert, getestet und weiterentwickelt.

Nichtdeterministische Inferenzmaschinen berücksichtigen die Tatsache, dass die meisten Dinge im praktischen Leben nicht absolut sicher sind. Auf die Frage: „Wie fertige ich am besten das Produkt A?“ kann häufig nur eine Antwort mit einer gewissen Unsicherheit gegeben werden. (falls die Fräsmaschine X nicht defekt ist oder falls Mitarbeiter Y nicht krank ist). Mit dieser Unsicherheit umzugehen und zu Schlussfolgerungen zu kommen, das ist Ziel dieser Klasse von Inferenzmaschinen.

In den jeweiligen Inferenzmaschinen werden je zwei unterschiedliche Entscheidungsstrategien zur Abarbeitung der Regeln eingesetzt: Die Vorwärtsverkettung (datengetrieben) und die Rückwärtsverkettung (zielgetrieben).

Die **Vorwärtsverkettung** (Forward Chaining) geht von der aktuellen Situation (Datenbasis) aus und arbeitet Regeln ab, deren Vorbedingungen durch die Datenbasis erfüllt sind. Die Aktionsteile der Regeln ändern und erweitern die Datenbasis, und der Vorgang wird wiederholt, bis keine Regel mehr anwendbar ist oder ein Abbruchkriterium erfüllt ist.

Bei der **Rückwärtsverkettung** (Backward Chaining) beginnt die Inferenzmaschine mit einer wahrscheinlichen Hypothese und prüft alle Regeln, deren Aktionsteil dieses Ziel enthält, bzw. die sich so zurückverfolgen lassen. Fehlende Informationen werden vom

Benutzer angefordert. Das System geht von einer angenommenen Zielhypothese aus und versucht eine Schlussfolgerungskette aufzubauen. Es wird dieses Schlussfolgerungssystem deshalb auch als zielgerichtetes Rückwärts-Verketten (goal-directed backward chaining) bezeichnet.

Beispiele für eine Rückwärtsverkettung findet man z. B. bei der Fehlersuche und bei Planungsvorgängen:

Das Auto bleibt stehen. Erste Hypothese ist: kein Benzin mehr im Tank, Man vergewissert sich, dass noch ausreichend Benzin im Tank ist. Die nächste Annahme ist: Zündung defekt. Man überzeugt sich, dass eine Zündkerze existiert, usw.

In der Produktionstechnik treten ähnliche Probleme in der Planung komplexer Fertigungsabläufe auf. Ein Endprodukt, das aus mehreren Zukaufteilen, Eigenfertigungsteilen und Montage entsteht, soll zu einem bestimmten Termin fertig sein. Wie müssen die einzelnen Arbeits- und Beschaffungstermine gelegt werden?

Die Ablaufsteuerung bei der Rückwärtsverkettung erfolgt implizit durch die Reihenfolge der Regeln und ihrer Vorbedingungen. Der Entwickler von Expertensystemen hat dadurch nur einen indirekten Einfluss auf den Programmablauf. Daher ist die Gestaltungsmöglichkeit bei umfangreichen Problemen beeinträchtigt, diesem Nachteil früher Expertensysteme wird in neueren Entwicklungen damit begegnet, prozedurale Steueralgorithmen neben deklarativem Wissen einzusetzen. [Vgl. BRUNS 1990].

3.2.2 Wissensbasis

Die Wissensbasis ist das Herzstück eines XPS. Sie besteht aus Informationen über Objekte und Regeln eines bestimmten Fachbereiches. Dabei ist ein Objekt ein Phänomen (Person, Gegenstand, GUI Element, Idee, Fehlerdiagnose), das durch eine Menge anwendbarer Regeln und Attribute (Regelwissen und Faktenwissen) definiert ist.

Eine Regel definiert Operationen, die mit oder von einem Objekt durchgeführt werden können, falls bestimmte Bedingungen erfüllt sind. Sie hat häufig den Aufbau:

WENN Bedingungsteil wahr, DANN Anweisungsteil.

Attribute sind Eigenschaften, die ein Objekt näher spezifizieren:

- z. B. Objekt: Auto, Attribut: Farbe, Wert: schwarz.
- Oder: Objekt: GUI Schaltfläche, Attribut: Schriftart, Wert: Arial.

Je nachdem, ob eine Wissensbasis stärker durch den Informationstyp Objekt oder Regel beschrieben wird, spricht man von objektorientierten oder regelorientierten Expertensystemen. In beiden Fällen erfolgt jedoch die Ableitung von Schlussfolgerungen nach kausalen Gesetzmäßigkeiten.

Eine Wissensbasis verfügt über eine leicht erweiterbare Datenbasis über ein spezielles, möglichst genau definiertes und scharf abgegrenztes Wissensgebiet, das im Idealfall auch noch mit unsicheren Daten umgehen kann [KURBEL 1992].

3.2.3 Wissensakquise Komponente

Eine Wissensakquise Komponente ist eine Erweiterung der Wissensbasis, die eine direkte Interaktion zwischen Experten und Expertensystem ermöglicht. Sie unterstützt die Aufnahme von neuem Wissen in die Wissensbasis, sowie die Löschung oder Manipulation von vorhandenem Wissen.

3.2.4 Erklärungskomponente

Eine Erklärungskomponente ist ein Modul eines Expertensystems, das dem Anwender auf Anfrage erklärt, durch welche Regeln und Fakten bestimmte Ergebnisse des Inferenzprozesses zustande gekommen sind. Sie schafft dem Benutzer also eine Transparenz, und zeigt warum bestimmte Entscheidungen gefallen und warum Alternativen ausgeschlossen wurden. Ausserdem könnte diese Komponente ausgebaut werden, um dem Anwender auf Wunsch Problemlösungs- und Verbesserungsvorschläge zu unterbreitet.

3.2.5 Grafische Benutzeroberfläche

Eine Benutzeroberfläche, über die die angrenzenden Komponenten gesteuert werden können und die es dem Benutzer und dem Programm ermöglicht, Fragen zu stellen und Antworten zu geben.

Bei Expertensystemen liegt ein besonderes Gewicht auf der Gestaltung einer benutzerbezogenen Mensch-Maschine-Schnittstelle mit ausgeprägter Verwendung sprachlicher und grafischer Elemente. Das Ziel ist, die Eingabe problembezogener Daten und die Rückfrage bzw. Diagnosen des Systems in einer Kommunikation (bis hin zu einer natürlich sprachlichen) zu ermöglichen.

3.3 Probleme beim Einsatz von Expertensystemen

Ein menschlicher Experte löst nicht nur Probleme, er erklärt die Ergebnisse in einer Weise, die nicht aus dem Aufzählen der Regeln besteht. Er lernt und strukturiert sein Wissen neu, weiß, wann er Regeln zu verletzen hat, wann eine Entscheidung besonders folgenschwer und wann unwichtig ist. Ein guter Experte erkennt insbesondere die Grenzen seines Könnens, wann er also Hilfe holen muss und wann er mit vertretbarem Risiko experimentieren darf oder das Problem von einer ganz anderen Ebene betrachten muss. Nach dem augenblicklichen Stand der Technik können Expertensysteme dies alles nicht [Vgl. BRUNS 1990].

Feigenbaum & McCorduck [FEIGENBAUM 1983] spekulieren über die zukünftigen Einsatzgebiete von Wissenssystemen und prognostizieren folgende Anwendungen:

- Den mechanischen Arzt, der streng methodisch Labordaten und Umgebungsbedingungen mit Symptomdaten verknüpft und Diagnosen stellt,
- Die intellektuelle Bibliothek, die den Informationssuchenden in einem Dialog zu den relevanten und benachbarten Wissensobjekten führt,
- Den intellektuellen Tutor, der sich auf das Vorwissen und die intellektuellen Fähigkeiten des Lernenden einstellen kann,
- Den Wissenssimulator, das Spiel zum Lernen,
- Das intellektuelle Nachrichtenblatt, das sich den Interessenten und Lesegewohnheiten des Lesers anpasst und dementsprechend eine Vorselektierung der Nachrichten vornimmt,
- Den Unterhalter, der geduldig und ermüdungsfrei die Konversation mit alten Menschen führt.

Diese überzogenen Erwartungen sind heute kritischeren und realistischen Einschätzungen gewichen, obwohl sich die technischen Probleme bei der Entwicklung leistungsstarker Expertensysteme nicht grundsätzlich unterscheiden von denen sehr großer Softwaresysteme, also ihrer Problematik der Unüberschaubarkeit und Unhandhabbarkeit. [Vgl. BRUNS 1990].

3.4 Techniken der Wissensrepräsentation

Der Gegenstand der Wissensrepräsentation ist die Formalisierung von Wissen, um eine maschinelle Verarbeitung durch den Computer überhaupt erst möglich zu machen. Generell unterscheidet man zwei Arten von Wissensrepräsentationen

- Prozedurales Wissen

- Deklaratives Wissen

Das Prozedurale Wissen lässt sich relativ einfach durch Programmiersprachen darstellen. Für deklaratives Wissen gibt es folgende Formalismen, von denen einige in den folgenden Abschnitten näher erläutert werden:

- Produktionsregeln
- Logik
- Semantische Netze
- Frames
- Spezielle Formalismen zur Repräsentation zeitlicher und räumlicher Aspekte
- Constraints

Dabei scheint Logik als das für Wissen am besten geeignete analytische Werkzeug zu sein [Vgl. CHOMSKY 1980], aus diesem Grund werden im Rahmen dieser Arbeit die Produktionsregeln in Verbindung mit der Aussagenlogik näher betrachtet.

3.4.1 Produktionsregeln

Eine Produktionsregel ist in einer formalen Sprache der Vorgang, bei dem eine bestimmte Menge von Zeichen durch eine andere bestimmte Menge von Zeichen ersetzt wird. Dieser Vorgang wird in der Mathematik symbolisch ausgedrückt als $A \rightarrow B$ (aus der Aussage A folgt die Aussage B). Eine Aussage kann sich dabei nach den Regeln der Aussagenlogik zusammensetzen. Produktionsregeln sind grundlegende Bestandteile von Bereichen, in denen mit formalen Sprachen gearbeitet wird. Diese Bereiche sind die moderne Linguistik und die Informatik (und damit auch die Mathematik). Noam Chomsky führte die Verwendung von Produktionsregeln in beiden Bereichen ein und etablierte so eine Ähnlichkeit zwischen Programmiersprachen und natürlichen Sprachen [Vgl. CHOMSKY 1980].

In der Informatik sind Produktionsregeln eine alternative Art, Algorithmen von Programmen darzustellen. Derartige Regeln werden etwa im Rahmen des Compilerbaus eingesetzt. Sie müssen regulär und kontextfrei im Sinne der Chomsky-Hierarchie sein. [Vgl. CHOMSKY 1963, S. 118-161] Produktionsregeln werden häufig in der Backus-Naur-Form dargestellt.

Eine Produktionsregel besteht dabei aus einem Prämissenteil, der die Vorbedingungen (manchmal auch Lefthandside (LHS) genannt) darstellt. Diese Vorbedingungen müssen im Sinne der Aussagenlogik wahr oder falsch sein, damit der Konklusionsteil ausgeführt wird. Der Konklusionsteil wird manchmal auch Aktionsteil, oder Schlussfolgerungsteil genannt.

Der Prämissenteil legt die Bedingungen für die Ausführung der Aktion oder der Aktionen im Aktionsteil fest.

3.4.2 Aussagenlogik

Die Aussagenlogik stellt den ältesten Zweig der formalen Logik dar. Die Aussagen sind grundsätzlich *wahr* oder *falsch* und können durch die Junktoren *und* und *oder* sowie die Verneinung *nicht* zu neuen Aussagen verknüpft werden [GOTTLOB et al. 1990]. Die Aussagenlogik entspricht der Schaltalgebra bzw., den dort angegebenen booleschen Ausdrücken [CLAUS 2003].

3.4.3 Prädikatenlogik

Die Prädikatenlogik wird auch **Logik erster Ordnung** genannt. Sie erweitert die Aussagenlogik um Variablen und die Quantoren ∀ („für alle….gilt…“) und ∃ („es existiert mindestens ein….für das gilt…“). Sie dient zur Formalisierung und zum Beweis von Eigenschaften von Programmen [CLAUS 2003].

3.4.4 Wissenserwerb oder Knowledge Engineering

Wissenserwerb umfasst alle Tätigkeiten zur Erfassung, Verwaltung und Transformation von Wissen. Wissenserwerb bezeichnet also den Prozess, bei dem eine Maschine Wissen erwirbt.

Die Grundprobleme sind technologischer Art:

- Wie sollte vorgegangen werden bei der Erhebung von Wissen, das für die Bewältigung bestimmter Anwendungsprobleme geeignet ist?
- Wie sollte dieses Wissen geordnet und von der Menge her bewältigt werden?
- Mit welchen Maßstäben oder Kriterien könnte ein Bestand an Wissen bewertet werden?

Das Wissen wird in geeigneter formaler Sprache repräsentiert (siehe Kapitel 3.4 Techniken der Wissensrepräsentation) und im Rechner gespeichert.

Unter der Annahme, dass Menschen und künstliche Problemlöser ähnlich beim Problemlösen vorgehen, wäre der menschliche Experte nur systematisch auszufragen und sein Wissen ohne weitere Veränderung in die Wissensbasis (als Frames oder Produktionsregeln) zu überführen.

Aber gerade Produktionsregeln ermöglichen kaum eine grobkörnige Ordnung, sondern entarten in großen Systemen zu einer unüberschaubaren Masse winziger Wissensatome.

Zudem haben Menschen keine Produktionsregeln im Kopf. Sie passen sich dem Verstehensniveau ihres Partners an und konstruieren beim Präsentieren ihres Wissens eine Ordnung, die dem Partner angemessen ist.

Auch wird durch das Sprechen und Vorstellen der Problemlösungskompetenz Wissen neu durchdacht und gewissermaßen neu erzeugt.

So ist Wissenserwerb ein Interationsprozess zwischen Wissensingenieur, Fachexperten und Software-Spezialisten [Vgl. BEHRENDT 1990].

4 Implementierung der Wissensdarstellung

In diesem Kapitel werden die Implementierungen beschrieben, die nötig sind um Wissen in computergerechte Regelsprache zu transformieren, um es anschliessend durch einen Inferenzmechanismus zu verarbeiten.

Wie in Abbildung 4-1 zu sehen ist, besteht die Wissensbasis eines Expertensystems aus zwei Komponenten. Das deckt sich mit den bereits erwähnten theoretischen Grundlagen. Es gibt einerseits die Problemdaten; die zu bewertenden und zu beurteilenden Daten, die manchmal auch Fakten Basis, Working Memory oder auch einfach Ist-Zustand genannt werden. In dem Fall der Din-Machine ist das ein gelayouteter GUI-Prototyp, der in der beschriebenen Java-Bean-Konvention in XML Format vorliegt. Später (Kapitel 4.7) wird ein Verfahren erläutert, mit dem es möglich ist, aus dem Java-Bean-XML-Format geeignete Fakten für das Working Memory zu ermitteln. Diese Erfassung von Faktenwissen wird Faktensensorik genannt.

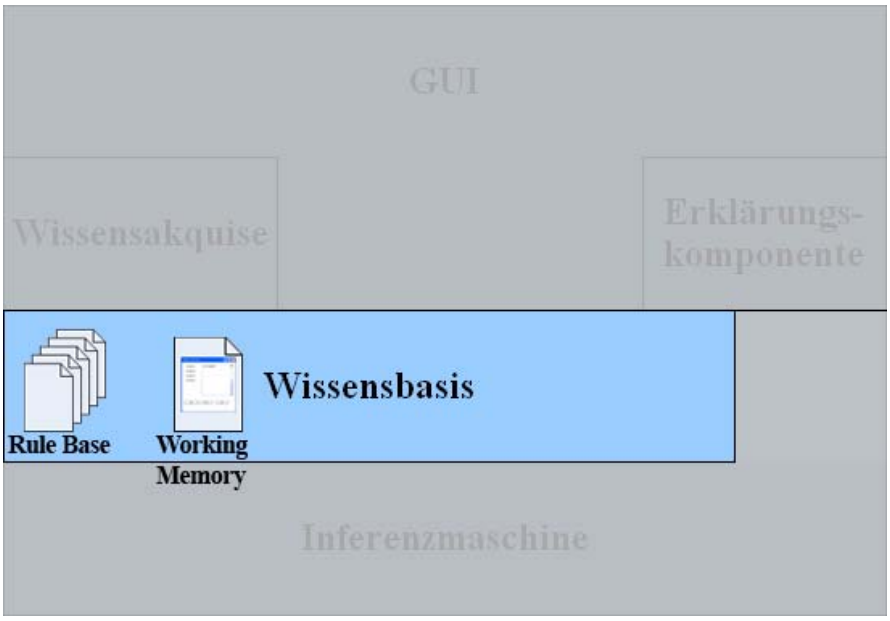

Abb. 4-1: Wissensbasis verfeinert

Andererseits existiert das eigentliche Wissen, das in Form von Regeln vorliegt. Dieses Wissen ist in der Abbildung 4-1 als Rule Base gekennzeichnet. Das Symbol der Rule Base soll bereits andeuten, dass sie aus einer oder mehreren Quellen zusammengefügt werden kann. Diese Quellen sind zu Beginn noch Dateien, in denen die Regeln in einer Regelsprache (in Kontext dieser Arbeit in XML formuliert) vorliegen; später wären

auch noch weitere Regelquellen wie z. B. Datenbanken, Entscheidungstabellen oder Regelbäume etc. denkbar.

4.1 Regelbasis (Rule Base)

Das allgemeine Schema und der Aufbau einer Regelbasis soll erst einmal anhand des UML Diagramms in der folgenden Abbildung 4-2 verdeutlicht werden. In dieser Abbildung ist zu sehen, dass eine Regelbasis aus mindestens einem Regelsatz, aber in der Praxis aus mehreren Regelsätzen (RuleSet) zusammengesetzt ist. Somit wird eine Komposition aus Regelsätzen „eine Regelbasis“ genannt.

Jedes RuleSet sollte einen systemweit (gemeint ist innerhalb der Rule Engine) eindeutigen und aussagekräftigen („sprechenden“) Namen besitzen. Vereinbarungen dieser Art sollten in einem Team beschlossen werden und werden im Folgenden „eine Nameskonvention“ genannt.

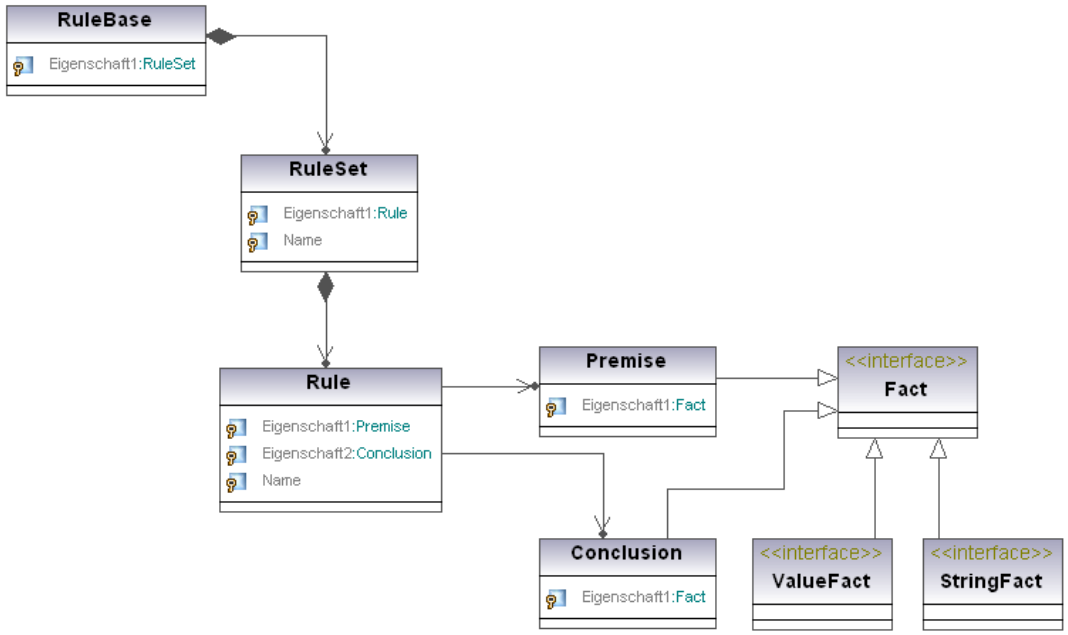

Abb. 4-2: UML Schema einer Rule Base mit Rule Sets und Rules…

Um eine bessere Übersicht über eine Vielzahl von Regeln zu bewahren, bietet es sich an, mehrere zusammengehörige oder verwandte Regeln zu clustern, also thematisch zusammen in einem Regelsatz zu sammeln. Daraus ergibt sich die Aussage, dass ein

Regelsatz ebenfalls eine Komposition darstellt. Ein Regelsatz ist eine Komposition aus Regeln. Für Regeln gelten dieselben Namenskonventionen wie für Regelsätze.

Eine Regel besitzt eine oder mehrere Bedingungen (die auch Prämissen genannt werden) und eine oder mehrere Schlussfolgerungen (die auch Konklusionen genannt werden).

Prämisse und Konklusion sind wiederum im einfachsten Fall jeweils ein Fakt; eine Aussage in Form eines Symbols der Aussagenlogik. Solch ein Fakt ist eine atomare Aussage, welche sich – wie die Bezeichnung schon vermuten lässt - nicht weiter in kleine Teilaussagen zerlegen lässt, ohne deren Bedeutung zu verlieren. Ausgedrückt werden solche einfachen Aussagen in Form eines **String-Fakts**, wie z. B. „Heute ist Freitag“ oder „Das Wetter heute ist regnerisch“.

Darüber hinaus könnte solch ein Fakt auch ein Wertefakt sein. Ein Wertefakt (**ValueFact**) ist ein Tupel bestehend aus einem Attribut (Keyelement) und einem zugehörigen Wert, wie z. B. Attribut „Tag“ mit Wert „Freitag“ oder Attribut „Hintergrundfarbe“ mit Wert „Rot“.

Ebenso kann ein Fakt eine der folgenden logischen Operationen sein:

- eine **Konjunktion**, also die ODER-Verknüpfung mindestens zweier Fakten (egal welchen Typs: StringFact oder ValueFact, Negation, Disjunktion),
- eine **Disjunktion**, die UND-Verknüpfung mindestens zweier Fakten, in Form und Kombination von StringFact, ValueFact, Konjunktion und Negation,
- oder eine **Negation**, die Verneinung eines Fakts oder einer logischen Operation.

Bevor nachfolgend diese Regel-Syntaxen (für Produktionsregeln und Aussagenlogik) und deren Darstellung in XML erläutert werden können, muss noch ein kurzes Kapitel über die Grundlagen von XML eingeschoben werden.

4.2 Die Extensible Markup Language (XML)

XML ist ein Standard zur Modellierung von strukturierten Daten in Form einer Baumstruktur. Dieser Standard ist vom World Wide Web Consortium (W3C) definiert worden und ist unter http://www.w3.org/XML/ einsehbar. XML definiert Regeln für den Aufbau von Dokumenten, die Daten enthalten, die einer fest vorgegebenen Struktur entsprechen.

XML als Rahmenkonzept lässt es offen, ob und wie ein konkretes XML Dokument automatisiert verarbeitet werden kann.

Jedes XML verarbeitende Programm wird „XML Anwendung“ genannt. Die Elemente der jeweiligen Dokumente müssen genau beschrieben werden. Dies betrifft insbesondere die Festlegung der Strukturelemente und ihre Anordnung innerhalb des Dokumentenbaums. Für diese Beschreibung stellt XML selbst Standards zur Verfügung: zum einen die bereits aus HTML bekannten Dokument Typ Definitionen (DTD) und zum anderen die wegen ihrer Flexibilität zukunftsträchtigeren XML Schema Definitionen (XSD), die ebenfalls in XML Sprache beschrieben werden.

Die Namen der Strukturelemente (XML Elemente) für eine XML Anwendung lassen sich frei wählen. Sie werden manchmal auch als Tags oder als Tagnamen bezeichnet. Ein XML Dokument wird laut W3C dann als **wohlgeformt** bezeichnet, wenn es sämtliche Regeln für XML einhält, beispielhaft seien hier folgende genannt:

- Das Dokument besitzt genau ein Wurzelelement
- Alle Tags mit Inhalt besitzen ein Beginn- (`<TagName>`) und ein End-Tag (`</TagName>`).
- Tags ohne Inhalt können auch mit den Zeichen `/>` abschließen.
- Nur Beginn-Tags dürfen (müssen aber nicht) Attribute enthalten.
- Die Beginn- und End-Tags sind korrekt verschachtelt.

Es ist grundsätzlich von Vorteil, wenn das Format einer XML Datei mittels einer Grammatik (einer Dokumenttypdefinition (DTD), eines XML Schemas (XSD) oder durch beide Grammatikarten parallel) definiert ist.

Der W3C-Standard definiert ein XML Dokument **dann** als **gültig (valide)**, **wenn** es **wohlgeformt** (den vorherigen Regeln entsprechend) ist und zusätzlich einen Verweis auf mindestens **eine Grammatik** enthält **und das** durch die Grammatik **beschriebene Format einhält.**

Programme oder Programmteile, die XML Daten auslesen, interpretieren und auf Gültigkeit prüfen, nennt man XML Parser; manchmal werden sie auch XML Prozessoren genannt. Prüft der Parser die Gültigkeit, so ist er ein validierender Parser.

Diese Kerntechnologien zur Verarbeitung von XML lassen sich grob in zwei APIs aufteilen, die im Laufe dieses Kapitels noch genauer im Rahmen des Regel-Parsing-Prozesses beschrieben und erläutert wird.

4.3 XML Syntax der Regelbasis

In diesem Kapitel wird beschrieben, wie mit Hilfe von XML eine Regelbasis erstellt werden kann. Zunächst wird das Grundgerüst samt Deklarationenteil zum Aufbau einer Regelbasis, wie sie in XML dargestellt werden kann, erläutert.

4.3.1 Grundgerüst der Regelbasis

Die folgende Abbildung 4-3 zeigt das Grundgerüst einer in XML formulierten Produktionsregel, wie sie nun definiert beschrieben werden kann. Auf die Syntax der Prämissen- und Konklusionsteile wird im Kapitel 4.4 konkreter eingegangen.

```
<Rule name="ein Regelname">
    <if>
        PRAEMISSENTEIL
        <then>
            KONKLUSIONSTEIL
        </then>
    </if>
</Rule>
```

Abb. 4-3: XML Syntax einer Regel (Rule)

Wie in der Abbildung zu sehen ist, beginnt eine Regel mit dem Start-Tag `<Rule>` in Verbindung mit dem `name` Attribut, das wieder nach einer Namenskonvention zu belegen ist. Produktionsregeln besitzen die Form: WENN Bedingung DANN Schlussfolgerung. Somit wird in dieser XML Syntax eine Produktionsregel mit einem `<if>`–Tag begonnen, in dem ein Prämissenteil und ein Schlussfolgerungsteil (eingeleitet mir einem `<then>`–Tag) eingefügt werden.

Die folgende Abbildung 4-4 zeigt die Syntax eines beispielhaften Regelsatzes. Dieser Regelsatz beinhaltet drei Regeln, deren Inhalt der Übersicht halber (durch Quellcode-Folding) ausgeblendet wurde.

Eingeleitet wird ein solcher Satz von Regeln durch das Tag-Element `<RuleSet>`, das genau wie eine Regel zwingend ein `name` Attribut erwartet. Laut hierarchischen und grammatikalischen Vorgaben können einem Regelsatz nur Regeln untergeordnet sein, nicht etwa weitere Regelsätze, geschweige denn Regelbasen oder direkt die Produktionen durch `<if>`– `<then>`–Tags.

```
<RuleSet name="ein RuleSet Name">

    <Rule name="ein Regelname">
    <Rule name="noch ein Regelname">
    <Rule name="und noch ein Regelname">

</RuleSet>
```

Abb. 4-4: XML Syntax eines Regelsatzes (RuleSet)

Eine Regelbasis setzt sich aus mehreren Regelsätzen zusammen, das wurde bereits zu Beginn dieses Kapitels erwähnt und wird, wie in Abbildung 4-5 zu sehen ist, nach folgender Syntax formuliert.

```
<RuleBase>

    <RuleSet name="ein RuleSet Name">
        <Rule name="ein Regelname">
        <Rule name="noch ein Regelname">
        <Rule name="und noch ein Regelname">
    </RuleSet>
    <RuleSet name="noch ein RuleSet Name">
    <RuleSet name="und noch ein RuleSet Name">

</RuleBase>
```

Abb. 4-5: XML Syntax einer Regelbasis (RuleBase)

Eingeleitet wird eine Regelbasis durch den Tag `<RuleBase>`. Es umschliesst erneut die bisher bekannte Struktur der Aufzählung von Regelsätzen. Das `<RuleBase>` Tag erwartet, entgegen den bisher kennen gelernten Tags, zurzeit noch keine Attribute; es könnte aber jederzeit um Attribute erweitert werden, falls das vom Team gewünscht und für sinnvoll gehalten wird.

In dieser beispielhaften Darstellung einer Regelbasis befinden sich drei Regelsätze, wobei in dem ersten Regelsatz immer noch die drei Beispielregeln angedeutet sind. Die

letzten beiden Regelsätze sind wieder durch Codefolding zu einer übersichtlichen Zeile verkleinert.

Die nötigen und am häufigsten benutzten Grundbausteine für eine Regelbasis wurden in den Abbildungen 4-1 bis 4-5 vorgestellt. Um allen XML Regeln aus dem vorherigen Kapitel gerecht zu werden sind noch zwei weitere wichtige Punkte zu erläutern, die in Abbildung 4-6 bereits angedeutet sind: Das eigentliche Wurzelelement, das jedes XML Dokument besitzt, und die Verweise auf XML Grammatiken, das auch allgemein Deklarationsteil genannt wird.

```
<?xml version='1.0' encoding='utf-8'?>
<!--
     EIN OPTIONALER DEFINITIONSTEIL
     FÜR NAMENSRÄUME UND XML-Regel-Schemata
     -->
<RuleEngine>
    <RuleBase>
        <RuleSet name="ein RuleSet Name">
        <RuleSet name="noch ein RuleSet Name">
        <RuleSet name="und noch ein RuleSet Name">
    </RuleBase>
</RuleEngine>
```

Abb. 4-6: XML-Syntax im Gesamtüberblick

Jedes XML Dokument besitzt zwingend ein Wurzelelement. Das Wurzelelement für diese hier beschriebene Regelsyntax ist der Tag `<RuleEngine>`. Diesem Tag muss zwingend ein `<RuleBase>`-Tag folgen; zusätzlich könnte dem Wurzelelement auch noch ein `<FactBase>`-Tag folgen, welches später näher erläutert wird.

Die erste Zeile in Abbildung 4-6 könnte, da es die erste XML Anweisung im Dokument ist, fälschlicherweise auch als Rootelement des XML Dokuments verstanden werden. Es ist aber nicht das eigentliche Rootelement, sondern vielmehr nur eine einfache Auszeichnung, mit der jede XML Datei beginnen sollte. Sie stellt lediglich den Bezug dieser Datei zu XML her. Es ist also der Beginn der Deklaration.

4.3.2 XML Deklarationsteil

Die XML Deklaration ist eine besondere, allein stehende Auszeichnung, deren erstes und letztes Zeichen innerhalb der spitzen Klammern < und > ein Fragezeichen ? ist. Unmittelbar hinter dem Anfangszeichen <? muss xml (kleingeschrieben) stehen. Dahinter können in Form von Attributen verschiedene Angaben folgen. Das Attribut „version“ bezieht sich dabei auf die Version der Sprachspezifikation von XML. Es gibt zwar bereits eine Version 1.1 von XML, wobei derzeit die Version 1.0 von XML maßgeblich ist, da die gegenwärtigen XML Parser normalerweise nur die Version 1.0 unterstützen.

„Da das Konzept von XML syntaktisch weitgehend ausgereift ist, ist auch nicht mit einer Versionsflut zu rechnen. Es sollte also, ausser in begründeten Ausnahmefällen, die Angabe version=“1.0“ benutzt werden.

[Vgl. http://de.selfhtml.org/xml/regeln/xmldeklaration.htm].

Es ist zu beachten, dass die XML Deklarationen vom Typ her aussehen wie Verarbeitungsanweisungen. Sie gehören jedoch nicht zu den eigentlichen Daten der XML Datei und werden auch nicht in der Baumstruktur der Daten repräsentiert.

Neben der Versionsangabe kann die XML Deklaration einer XML Datei zwei weitere Attribute enthalten: eines zur verwendeten Zeichenkodierung (encoding), und ein Ja/Nein-Attribut (standalone). Mit dem Attribut encoding wird angegeben, welcher Art der Zeichenkodierung zum Speichern der XML Datei verwendet wird.

Mit dem Attribut „standalone“ kann dem Parser vorab mitgeteilt werden, ob die vorliegende Datei sich auf eine externe DTD bezieht oder sich die DTD-Grammatik innerhalb der aktuellen Datei befindet. Mit standalone=“no“ wird mitgeteilt, dass sich die DTD in einer separaten Datei befindet. Die Quelle müsste in diesem Fall mit Hilfe der Dokumenttyp-Deklaration angegeben werden. Da für diese Regelgrammatik keine DTD, sondern eine moderne, zukunftsorientierte XML Schemadatei erstellt wurde, ist dieses Attribut zu vernachlässigen.

Letztlich ist noch zu beachten, dass zwar die beiden Attribute standalone und encoding optional sind, bei der Verwendung aber eine bestimmte Reihenfolge eingehalten werden muss. Genauer gesagt lautet die Reihenfolge:

version – encoding – standalone

Der Definitions- bzw. Deklarationsteil für Namensräume und XML Schemata ist in Abbildung 4-6 noch aus dem eigentlichen XML Daten auskommentiert und wird dort in grüner Schriftfarbe dargestellt.

An dieser Stelle im Dokument sollen die Dokumenttyp Definitionen geschrieben stehen. Da diese Dokumentgrammatikbeschreibungen sehr umfangreich werden können und dadurch ein Dokument sehr unübersichtlich machen würden, wäre an der Stelle aber in der Praxis allerhöchstens ein Verweis auf eine externe Datei zu finden, in der die DTD Beschreibungen notiert sind.

Dieser Bereich ist ausserdem für die weitere Variante der XML Sprachengrammatik vorgesehen. Hier könnten direkt die XML Schema Definitionen verfasst werden, das natürlich sehr unpraktikabel ist, da bereits die XSD für die bisher beschriebene Struktur der Regelbasis und die noch folgende Struktur der Produktionsregeln ca. 200 Zeilen lang sind. Aus diesem Grund sind die XSDs bereits in zwei Dateien aufgeteilt worden. Beide Dateien werden später im Kapitel 6 im Rahmen der Validierung erläutert und sind zusätzlich im Anhang dieser Arbeit einzusehen.

Anstatt die Grammatik in Form von XSD Anweisungen direkt in den Kopfbereich eines Dokumentes zu schreiben, wird man in der Praxis lediglich einen Verweis auf ein externes XSD Dokument verwenden. Diesen Verweis kann man entweder, wie in Abbildung 4-6 angedeutet, an der Stelle des grünen Kommentars platzieren, oder ihn als Attribut dem ersten definierten Element angeben. Im Fall dieser Arbeit wäre das erste definierte XML Element das bereits kennen gelernte Tag `<RuleEngine>`.

Die „lab4inf“ XML Regelsprache befindet sich allerdings noch im Aufbau- und Experimentierstadium. In den kommenden Monaten wird diese Regelsprache um einige bestimmte Sprachelemente erweitert. Somit existieren noch keine finalen Versionen der XSD Grammatik Dateien. Diese Dateien werden üblicherweise durch das Entwicklerteam in öffentlich zugänglichen Bereichen auf Webservern den Anwendern zugänglich gemacht.

In Rahmen dieser Arbeit wurden während der Entwicklung der Grammatiken und auch während der Regelerstellung die XSD Schema Dateien dem Parser direkt bekannt gemacht. Das Verfahren wird später genau erläutert.

4.4 XML Syntax innerhalb einer Regel

Nachdem nun also ausführlich in den letzten beiden Unterkapiteln die Grundelemente der Regelbasis und der Regelsätze behandelt wurden, widmet sich dieses Kapitel der XML Darstellung innerhalb der Regeln. Regeln setzen sich, wie in Kapitel 3 beschrieben, aus der Aussagenlogik zusammen. Mehrere Bedingungen, die über Kon- („und“) oder Disjunktion („oder“) verbunden wurden, finden sich durch Implikation („daraus folgt“) und einer oder mehreren Aktionen (Konsequenzen) zu Regeln

zusammen. Im Rahmen dieses Kapitels soll nun die XML Syntax der Produktionsregeln in all ihren Ausprägungen vorgestellt werden.

Eine Aussage in der Aussagenlogik ist im einfachsten Fall ein einzelnes Fakt in Form einer Aussage. Die Aussage „*Heute ist Freitag*“ ist ein solches Fakt. In diesem Kontext wird es weiter als StringFact bezeichnet, da es aus einer einfachen Aussage in Form eines String besteht. In XML Schreibweise würde diese Aussage wie folgt aussehen:

```
<fact>Heute ist Freitag</fact>.
```

Dieselbe Aussage könnte man in diesem Fall durch ein Tupel bestehend aus einem Key und einem zugehörigen Wert erreichen. Der Key wäre „*Tag*“ und der zugehörige Wert wäre dann „*Freitag*“. Diese Form wird weiter als ValueFact bezeichnet. Ein ValueFact existiert hauptsächlich für Wertzuweisungen und tatsächliche Wertvergleiche, unter die „ist gleich“, „ist ungleich“, „ist größer“, „ist kleiner“, „ist größer gleich“, „ist kleiner gleich“ Vergleiche fallen. In solchen Fällen setzt sich eine Bedingung aus insgesamt drei Teilen (Key, Vergleichsoperator, Value) zusammen, und sieht in XML Schreibweise folgendermassen aus:

```
<complexFact>Tag<eq />Freitag</complexFact>
```

Die Wahl des Tagnamen „complexFact“ rührt daher, dass sich ein Fakt aus Teilelementen zusammensetzt. Die Bezeichnung könnte nach belieben sehr schnell und einfach z. B. auf „valueFact“ oder ähnlichem geändert werden; dazu müssten lediglich die XML Schema Definitionsdatei FactBase.xsd (siehe Anhang) und die jeweiligen Content Handler (siehe Kapitel über Parsing-Prozesse) der XML Parser angepasst werden. Im Übrigen gilt diese Änderungsmöglichkeit für alle hier vorgestellten Tagnamen.

Diese beiden nun kennen gelernten Aussageformen gelten sowohl für den Bedingungsteil, als auch für den Aktionsteil einer Produktionsregel. Abbildung 4-7 zeigt dieses anhand des einfachen Beispiels „*Wenn heute Freitag ist, dann ist morgen Wochenende*“. Ebenfalls zeigt diese Abbildung, dass sich mithilfe eines zusammengesetzten Faktes in Verbindung mit dem „*equals*“-Operator eine nahezu bedeutungsgleiche Aussage formulieren lässt, wie durch einen einfachen Aussagesatz.

```
<Rule name="Freitag">
    <if>
        <fact>Heute ist Freitag</fact>
        <then>
            <fact>Morgen ist Wochenende.</fact>
        </then>
    </if>
</Rule>

<Rule name="key Tag value Freitag">
    <if>
        <complexFact>Tag<eq />Freitag</complexFact>
        <then>
            <fact>Morgen ist Wochenende.</fact>
        </then>
    </if>
</Rule>
```

Abb. 4-7: XML Syntax der Prämissen- und Konklusionsteile

Die Tabelle 4-1 ist eine Übersicht über die XML Syntaxen der einzelnen Vergleichsoperatoren, die in einem ValueFact als Operator verwendet werden und in das folgende Fakten Schema eingebaut werden können:

`<complexFact>`…Attribut…`<Operatoranweis. />`…Wert…`</complexFact>`

Tab. 4-1: XML Syntaxen der Vergleichsoperatoren.

Anweisung	**Bedeutung**	**Mathematisches Symbol**
`<eq />`	„ist gleich“, „is equal to“	=
`<neq />`	„ist ungleich“, „is not equal to“	≠
`<gt />`	„ist größer“, „greater than“	>
`<ge />`	„ist größer oder gleich“, „greater or equal“	≥
`<lt />`	„ist kleiner“, „is less than“	<
`<le />`	„ist kleiner oder gleich“, „less or equal“	≤

Um der tatsächlichen klassischen Aussagenlogik gerecht zu werden fehlen selbstverständlich noch die so genannten Junktoren, also die Verknüpfungen mehrerer Aussagen. Diese Junktoren sind in Ingenieurskreisen allseits bekannt als logische

Operatoren. Eine grundsätzliche logische Operation wurde bereits implizit durch die Wahl der Produktionsregeln (<if>–<then>) als Regelart erfasst: die „materiale Implikation“, auch Subjunktion oder Konditional genannt oder in Worten ausgedrückt „aus der Aussage A, folgt die Aussage B“, in mathematischen Symbolen ausgedrückt: A → B.

Die weiteren drei logischen Verknüpfungen wurden ebenfalls in der XML Regel Syntax berücksichtigt. Das logische Und, also die Konjunktion A und B, „sowohl A als auch B“; das logische Oder, also die Disjunktion A oder B, „entweder A oder B oder beide“; und die Negation, also eine Verneinung, „nicht A“.

Die entsprechenden Syntaxen in XML erklären sich fast von alleine. Dementsprechend wird bei der Negation einer Aussage, die jeweilige Aussage (ein StringFact, ein ValueFact, eine Oder-Verknüpfung oder eine Und-Verknüpfung) mit dem XML Tag <not>…eine Aussage…</not> umschlossen.

Ähnlich umschließende Tags werden für die Realisierung einer Konjunktion (<and>…Aussagen…</and>) und einer Disjunktion (<or>…Aussagen…</or>) verwendet. Wobei bei diesen beiden Implementierungen zwingend mindestens zwei Aussagen innerhalb der umschließenden Tags vorkommen müssen. Eine Höchstgrenze beteiligter Aussagen ist nicht gegeben.

Um eine bessere Übersicht über den Umfang der XML Syntax zu bekommen, befinden sich in der Abbildung 4-8 einige beispielhafte und aussagekräftige logische Formulierungen an ausgewählten ergonomischen GUI-Regeln.

```
<Rule name="MaxAmountOfColors">
    <if>
        <and>
            <fact>Farbdisplay wird verwendet</fact>
            <complexFact>AmountOfColors<gt />6</complexFact>
        </and>
    <then>
        <fact>es werden mehr als 6 Farben verwendet!</fact>
    </then>
    </if>
</Rule>

<Rule name="darkblue frame">
    <if>
        <or>
            <complexFact>frame.backgroundColor<eq />Color(r=0,g=0,b=67)</complexFact>
            <complexFact>frame.backgroundColor<eq />Color(r=0,g=0,b=69)</complexFact>
            <complexFact>frame.backgroundColor<eq />Color(r=0,g=0,b=70)</complexFact>
            <complexFact>frame.backgroundColor<eq />Color(r=0,g=0,b=77)</complexFact>
            <complexFact>frame.backgroundColor<eq />Color(r=0,g=0,b=89)</complexFact>
            <complexFact>frame.backgroundColor<eq />Color(r=0,g=0,b=126)</complexFact>
        </or>
        <then>
            <complexFact>frame.backgroundColor<eq />DARKBLUE</complexFact>
        </then>
    </if>
</Rule>

<Rule name="darkblue switch">

<Rule name="isPresentDarkblue" enable="true">
      <if>
        <or>
            <fact>TextColor<equals />DARKBLUE</fact>
            <complexFact>frame.backgroundColor<equals />DARKBLUE</complexFact>
            <complexFact>switch.backgroundColor<equals />DARKBLUE</complexFact>
        </or>
      <then>
            <fact>Darkblue wird verwendet</fact>
      </then>
      </if>
</Rule>
```

Abb. 4-8: XML Syntaxen: Beispiele mit logischen Operatoren

4.5 Regelsprachenvergleich anhand einer Ergonomieregel

Es gibt es noch einige weitere Regelsprachen auf dem Markt. Beispielhaft wird in diesem Kapitel ein Vergleich aufgezeigt zwischen der im Rahmen dieser Arbeit entstandenen, gerade vorgestellten „lab4inf“ Regel Syntax und einer weiteren am Markt bekannten Regelsprache.

Begonnen wird mit einer Ergonomieregel, die anhand der lab4inf-Regel Syntax formuliert, in Abbildung 4-9 vorliegt.

```
<Rule name="MaxAmountOfColors">
    <if>
        <and>
            <fact>Farbdisplay wird verwendet</fact>
            <complexFact>AmountOfColors<gt />6</complexFact>
        </and>
    <then>
        <fact>es werden mehr als 6 Farben verwendet!</fact>
    </then>
    </if>
</Rule>
```

Abb. 4-9: Ergonomieregel in lab4inf Regel Syntax

Diese Regel wird nun versucht in der Regelsprache des Produktes *Drools* zu formulieren.

Drools ist eine der im Internet frei erhältlichen Open Source Rules Engines. Die Internetpräsenz ist unter http://www.drools.org zu erreichen. Mittlerweile ist dieses Produkt von JBoss (einer Unterabteilung der Firma Redhat) übernommen worden und wurde dort ab der Version 2.5 an bis zur derzeit aktuellen Version 3.0.5 weiterentwickelt.

Drools verwendet intern ebenfalls eine Forward Chaining Regelmaschine. Für die interne Abbildung der Regeln verwendet Drools typische Javaobjektklassen, die miteinander kombiniert Bedingungen, Konsequenzen, Regeln, Regelsätze und Regelbasen bilden. Drools erlaubt es Regeln in Form von XML-Dateien abzulegen. Die Struktur der XML-Dateien ist dabei im Wesentlichen durch DTDs und XSDs vorgegeben, die Ausgestaltung und Syntax orientiert sich an bereits bestehenden Programmiersprachen. Das heißt der Benutzer kann Regelsätze in XML-Form ablegen und dabei auf die Syntax von Java, Python und Groovy zurückgreifen. Mittlerweile existiert auch eine Anpassung zur Formulierung von Regeln in c# (Sprich: „C Sharp“), d. h. auf Basis der Microsoft .NET-Sprachfamilie.

Eine gute Einführung in die Arbeitsweise und Syntax von Drools liefert [WUNDERLICH 2006, S. 63 ff.].

In Abbildung 4-10 ist exemplarisch der Versuch aufgezeigt die obige ergonomische GUI Regel (aus Abbildung 4-9) in der Regelsyntax von Drools zu formulieren. Es sei

erwähnt, dass die Regel zu jeder Zeit schematisch dargestellt sein soll, sie wurde nie produktiv implementiert und getestet, da dieses vorausgesetzt hätte als Rules Engine das Produkt Drools in die Prozesse der DIN-Machine zu integrieren. Diese Option wurde in der Projektplanung bis jetzt nicht berücksichtigt, könnte aber in der Zukunft wichtige Erkenntnisse für das DIN-Machine Rules Engine Modul liefern.

```
<?xml version="1.0">
<rule-set name="name of ruleset"
        xmlns="http://drools.org/rules"
        xmlns:java="http://drools.org/semantics/java"
        xmlns:xs="http://www.w3c.org/2001/XMLSchema-instance"
        xs:schemaLocation="http://drools.org/rules rules.xsd
    http://drools.org/semantics/java java.xsd">

    <import>java.lang.String</import>
    <import>java.lang.Double</import>

    <rule name="MaxAmountOfColors">
        <parameter identifier="displayinfo">
            <class>String</class>
        </parameter>
        <java:condition>displayinfo.equals("Farbdisplay")</java:condition>

        <parameter identifier="AmountOfColors">
            <class>Double</class>
        </parameter>
        <java:condition>AmountOfColors &gt; 6</java:condition>

        <java:consequence>
            System.out.println("es werden mehr als 6 Farben verwendet!")
        </java:consequence>
    </rule>

</rule-set>
```

Abb. 4-10: Ergonomieregel in Drools Regel Syntax

Auffällig ist auf den ersten Blick, dass sich XML-Elemente und Java Schreibweisen kombinieren (<java:condition>displayinfo.equals(….)</java:condition>) und dass die Drools Rules Engine bereits in der Regelsprache auf tatsächliche Objekte und deren Attribute zugreifen kann (Objekt *displayinfo* von Typ String); was ein erstrebenswertes Anliegen für die lab4inf Rules Engine darstellt.

Ausserdem fällt auf, dass die Drools Regelsprache für einen geübten Java Programmierer sehr leicht zu durchschauen ist, von einem Programmier Laien aber

nicht so einfach und intuitiv verstanden und gelesen werden kann, was ein Grund war, warum eine eigene einfache Regel XML Sprache entwickelt wurde.

Ausserdem sei noch erwähnt, dass es noch eine sehr verbreitete quasi universelle Regelsprache gibt, die unter dem Namen RuleML bekannt ist und deren Syntax sehr an die Sprache PROLOG angelehnt ist. Aktuell werden Bemühungen angestellt RuleML als Standardsprache für regelbasierte Programmierung in Form eines XML Dialektes beim W3C durchzusetzen. Nach einer Untersuchung und genaueren Prüfung dieser Sprache hinsichtlich ihrer Verwendung in der „lab4inf" Rules Engine könnte sich diese ebenfalls als brauchbar erweisen. Dafür müssten dann geeignete, auf die „lab4inf" Rules Engine und RuleML Syntax abgestimmte Parsing Routinen entwickelt werden.

4.6 Der Regel Parsing Prozess

In den vorherigen Kapiteln wurde erklärt, wie Regeln mit einer XML Syntax beschrieben werden können. Nachfolgend wird nun geklärt, wie diese XML Regel Strings in den Inferenzmechanismus gelangen. Um diese Frage genau zu beantworten, müssen zuerst die genauen Vorgaben der benutzen Inferenzmaschine bekannt sein.

Dazu genügen vorerst die Informationen, dass die **API der** verwendete **Inferenzmaschine** es vorsieht, mit **Regeln** in Form eines **`HashSet<Rule>`** initialisiert zu werden. Was ein Regelparser also erzeugen muss ist ein Satz (siehe Java Collection API Set) von gehashten Regeln. Der Hashwert errechnet sich dabei über den Namen der Regel und seiner Beschreibung. Dieser Hashwert sorgt also dafür, dass es nicht zwei Regeln mit gleichem Namen und gleicher Beschreibung zur selben Zeit in der Inferenzmaschine geben kann; sehr wohl aber zwei Regeln mit gleichem Namen und unterschiedlicher Beschreibungen oder umgekehrt. Zur Regelerzeugung sieht die API der Inferenzmaschine eine spezialisierte Factory-Klasse vor: die Klasse `RuleFactory`.

Für den zu bewertenden **Ist-Zustand** erwartet die Inferenzmaschine eine Zweidimensionale HashMap aus einem String und einem Fakt. Derselbe Datentyp wird nach Abarbeitung des Inferenzprozesses als Schlussfolgerungen zurückgeliefert; also ebenfalls eine **`Hashmap<String, Fact>`**. Genauere Informationen sind in einer separaten Dokumentation einsehbar [Verweis auf JavaDOC von Prof. Wulff].

Der Prozess aus einer auf der Festplatte persistent vorliegenden Datei Daten auszulesen nennt sich „Parsing Prozess". Ein Parser ist also für die Zerlegung und Umwandlung von XML Eingabestrings in ein für die Weiterverarbeitung brauchbares Format zuständig. Genauer gesagt sollen XML Regeln eingelesen werden, und eine Datenstruktur vom Typ `HashSet<Rule>` soll als Ergebnis erzeugt werden.

Grundsätzlich verwendet ein Parser zur Analyse eines Textes einen separaten lexikalischen Scanner (auch Lexer genannt). Dieser zerlegt die Eingabedaten (als simple Aneinanderreihung von Zeichen vorliegend) in Token, also bestimmte Wörter oder Eingabesymbole, die der Parser versteht. In Rahmen dieser Arbeit entspricht das den XML Elementen, die in den vorherigen Kapiteln vorgestellt wurden.

Das World Wide Web Consortium (W3C) definiert zwei Standard APIs zum einlesen von XML Dokumenten; zum einen eine Event-basierte API unter dem Kürzel SAX und zum anderen eine auf einer Baumstruktur basierende API, die bekannt ist unter dem Namenskürzel DOM. Beide Konzepte wurden im Rahmen dieser Arbeit implementiert. Was es bei Beiden zu beachten gab und warum tatsächlich auch beide Konzepte verwendet wurden, anstatt nur eines, wird in den folgenden Kapiteln erläutert.

4.6.1 SAX-Parsing Prozess

SAX (Simple API for XML) ist eine standardisierte Möglichkeit, wie eine XML Datei durch einen Parser bearbeitet wird. Die derzeit aktuelle Version SAX 2.0 [http://www.saxproject.org] ist Bestandteil von Java seit der Version 1.4 (Anmerkung: derzeit Aktuell ist Java Version 6), dementsprechend ist dieses Verfahren in Java-Entwickler-Kreisen bereits sehr etabliert.

SAX ist die Ereignisgesteuerte (Event-basierte) API zum Einlesen und Verarbeiten von XML Dateien. Event-basiert bedeutet hierbei, dass der Parser ein XML Dokument sequentiell von Anfang bis Ende einmal einliesst und die aufrufende Applikation bei jeder erkannten syntaktischen Struktur (Element, Kommentar, Text, etc.) benachrichtigt. Hierbei wird also ein Datei-Strom in einen Strom von Ereignissen umgewandelt. Programme bzw. bestimmte Verhaltensweisen oder Algorithmen können sich für einzelne Ereignisse registrieren, um bei Aufruf des jeweiligen Ereignisses ihre Arbeit zu verrichten. Die Eingabedaten werden hier rein sequentiell verarbeitet, wobei dabei der Vorteil von SAX ist, dass nicht die gesamte XML Datei im Arbeitsspeicher vorgehalten werden muss. Das könnte aber dann ein Nachteil sein, wenn man viele Informationen, die z. B. über die ganze Datei verstreut sind, zur Verarbeitung benötigt. Dieser Nachteil ist im Rahmen dieses Projektes zu vernachlässigen, da die Informationen für jeweils eine Regel örtlich sehr nah zusammen stehen.

Basierend auf dem SAX Konzept wurde ein Verfahren für den Einlesevorgang von XML-formulierten Produktionsregeln entwickelt. Dieses Verfahren wird im Weiteren nun erläutert.

Um eine XML Datei in einen für die lab4inf Inferenzmaschine gültigen Satz von Produktionsregeln zu transferieren, benötigt man lediglich zwei Klassen, und pro Klasse einen öffentlichen Methodenaufruf. Die erste beteiligte Klasse ist der

`RuleTreeBuilder`, dessen einzige öffentliche Methode `create()` als Argument die URI (Uniform Resource Identifier) in Form eines Strings erwartet. Ein URI ist eine Zeichenfolge zur Identifikation einer abstrakten oder physischen Ressource, also ein absoluter oder relativer Pfad zu einer Datei, entweder im lokalen Dateisystem oder auch im Internet. Nachdem diese Klasse ihre Arbeit erledigt hat, liefert sie eine Datenstruktur in Form einer Baumstruktur zurück. Genauer gesagt liefert sie einen Regelbaum in Form eines `javax.swing.tree.DefaultMutableTreeNodes` zurück. Dieser Regelbaum besitzt noch nicht die korrekte Form, damit er in die lab4inf Inferenzmaschine eingegeben werden kann. Um die richtige Form zu erhalten, muss die Baumstruktur noch durch die `RuleTreeWalker` Klasse verarbeitet werden. Die einzig öffentliche Methode (neben dem Konstruktor) der `RuleTreeWalker` Klasse lautet `walk()` und erwartet als Argument einen `DefaultMutableTreeNode`. Der Datentyp, der nach Verarbeitung von dieser Klasse zurückgeliefert wird ist ein `Set<Rule>`; um genau zu sein ein `HashSet<Rule>`. Und mit dieser Art Datentyp kann die Regelbasis der lab4inf Inferenzmaschine gefüllt werden.

Die folgende Abbildung 4-11 demonstriert die Verarbeitungsschritte.

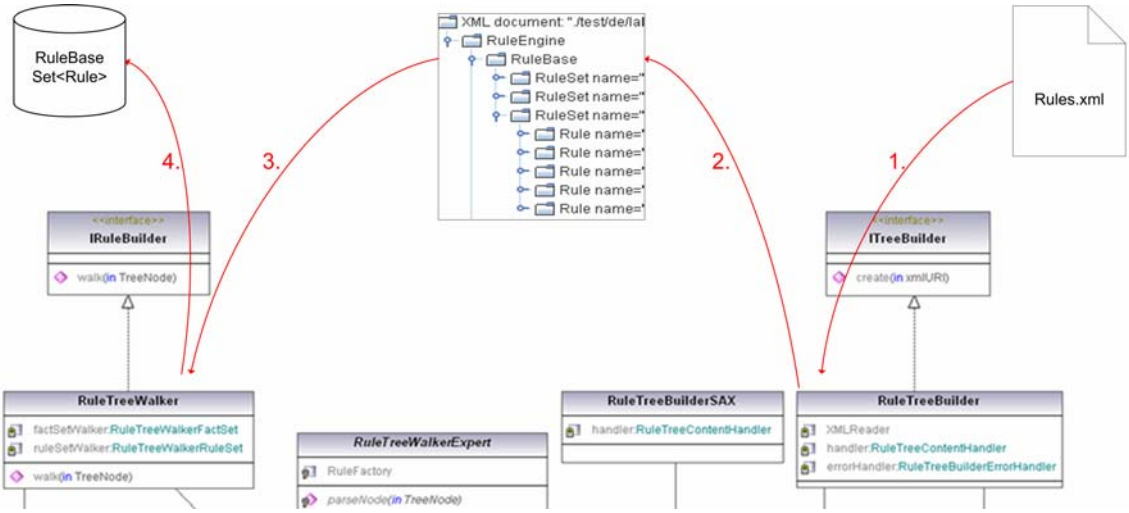

Abb. 4-11: Visualisierung des SAX-Parsing Prozesses

Die konkreten Implementierungen in Java werden nun im Detail erläutert. Die Erläuterungen beziehen sich auch das UML 2.0 Diagramm, das in Abbildung 4-12 dargestellt wird.

Um in der Klasse `RuleTreeBuilder` eine XML Datei einlesen zu können, besorgt man sich zu erst aus der `org.xml.sax.helpers.XMLReaderFactory` einen XML Reader. Das erfolgt durch den Aufruf

```
XMLReader xmlReader = XMLReaderFactory.createXMLReader();
```

Des Weiteren sieht die SAX API die Registrierung von so genannten Handlern vor. Die Struktur dieser Handler-Klassen wird durch Interface-Klassen von der API vorgegeben. Es wird dem entsprechend ein ErrorHandler an der `xmlReader`-Instanz registriert. Die konkrete Implementierung des Inferfaces `DefaultErrorHandler` ist die `RuleTreeBuilderErrorHandler` Klasse. In dieser Klasse existieren drei ausimplementierte Methoden, die auftretende Fehler während des Parsing-Prozesses aufzeigen würden. Dabei werden drei Arten von Fehlern unterschieden: Schlichte Warnungen, Fehler und Fatale Fehler. Darauf wird dann so reagiert, wie es in den jeweiligen Methoden implementiert wurde. In diesem konkreten Fall werden die Fehlerbenachrichtigungen einem Logger übergeben, wodurch die Nachrichten erst einmal auf der Konsole angezeigt werden. Durch die Übergabe an eine bestimmte Logger-Instanz könnten die gesamten Logger-Nachrichten in einer Log-Datei gespeichert und zu einem späteren Zeitpunkt analysiert werden. Das Registrieren einer Fehlerbehandlung ist essentiell, da eventuelle syntaktische oder grammatikalische Fehler in der Regel-XML-Datei so angezeigt würden.

Durch den Aufruf von

```
//register error handler
xmlReader.setErrorHandler( new RuleTreeBuilderErrorHandler() );
```

wird die Fehlerbehandlung an dem zuvor besorgten XMLReader registriert.

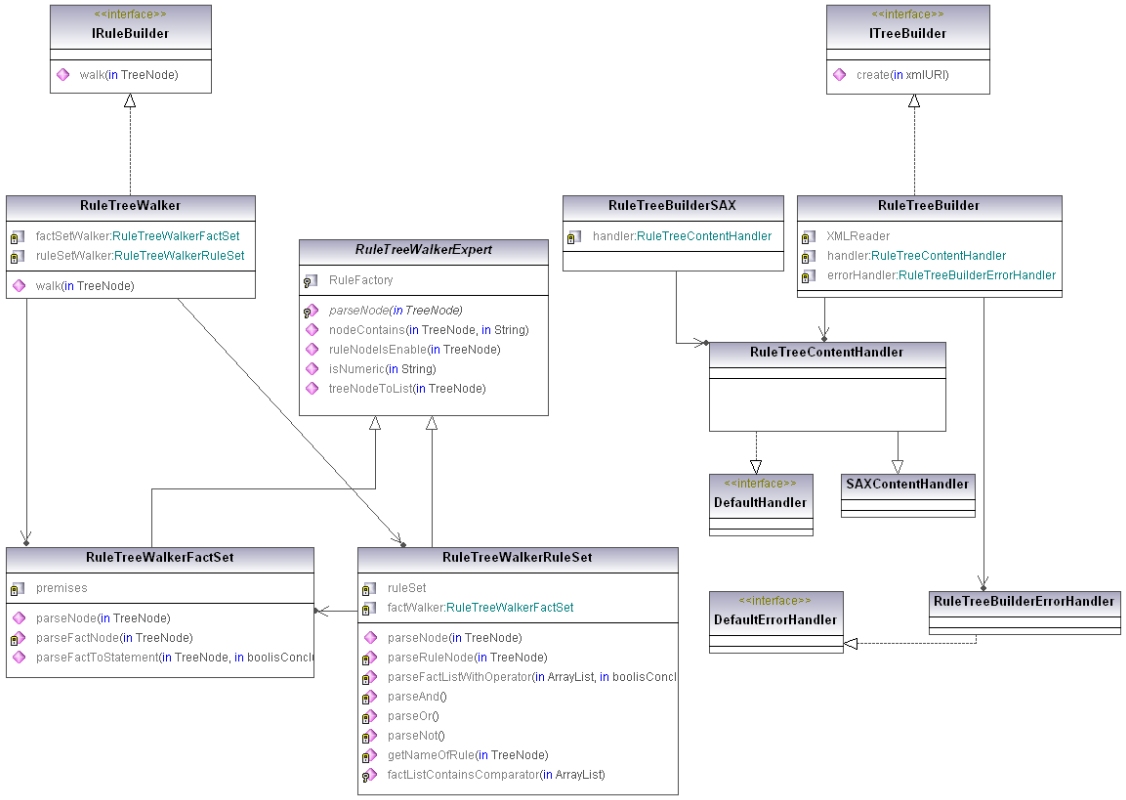

Abb. 4-12: UML 2.0 Diagramm: Konzept SAX-Parser

Des Weiteren wird die Instanz des XMLReaders mit dem Herzstück, das für die Verarbeitung zuständig ist, versorgt. Es wird ein so genannter ContentHandler registriert. Dieser ContentHandler hält sich wiederum an die Vorgaben der SAX API durch die Verwendung des Interfaces `DefaultHandler` und der abstrakten Klassen `SAXContentHandler`. Das Registrieren dieses besagten Handlers erfolgt durch folgende Quellcodepassage:

```
//register Content Handler
ContentHandler handler = new RuleTreeContentHandler( treeModel,
rootNode );
xmlReader.setContentHandler( handler );
```

Der Konstruktor des `RuleTreeContentHandler` bekommt als Argument ein `treeModel` (das nicht weiter erwähnenswert ist) und eine Datenstruktur von Typ `DefaultMutableTreeNode` übergeben. Von diesem Baumknoten wurde zuvor eine Instanz gebildet und er ist der Wurzelknoten (`rootNode`) der zu befüllenden Baumstruktur.

Nachdem nun die XMLReader-Instanz durch die Befehlsfolge

```
//Parsing
InputSource inputSource = new InputSource( xmlURI );
xmlReader.parse( inputSource );
```

gestartet wurde, wird die Baumstruktur durch die Algorithmen im ContentHandler durch die jeweiligen EventHandler mit der hierarchischen Regelstruktur aus der XML Datei gefüllt.

Das Konzept des SAX Parsers sieht in dem ContentHandler Interface insgesamt elf EventHandler vor, die in der Tabelle 4-2 aufgelistet sind.

Tab. 4-2: EventHandler eines SAX Parsers

EventHandler
DocumentLocator
StartDocument
EndDocument
StartPrefixMapping
EndPrefixMapping
StartElement
EndElement
Characters
IgnorableWhitespace
ProcessingInstruction
SkippedEntity

Ausimplementiert wurden in diesem Fall erst einmal nur die drei in der Tabelle fett gedruckten Events, da diese genügen um einen Regelbaum von XML Syntax in einen Java Swing Baum zu transferieren, wie an folgendem Beispiel und in Abbildung 4-14 gezeigt wird.

Der SAX Parser iteriert einmal über die Zeichenfolgen in dem XML Dokument und löst beim Auffinden von bekannten Token aus Tabelle 4-2 die jeweiligen Ereignisse (Events) aus.

Die `RuleTreeContentHandler` Klasse fügt nun bei jedem Auftreten eines Startelements den Tagnamen und dessen Attribute (falls vorhanden) als einen weiteren Knoten in die Baumstruktur ein.

Das Character-Event kann laut XML Grammatik nur nach einem `<fact>`- oder einem `<complexFact>`-Element gefeuert werden. Beim Auftreten dieses Elements werden

die gefundenen Zeichen wiederum als einen weiteren Kindknoten an den vorherigen Knoten angehängt.

Beim Auftreten eines Endelement-Events wird im Datenbaum eine Ebene zurückgesprungen; genauer gesagt wird vom gerade aktuellen Baumknoten zu dem Elternelement gewandert. Sehr viel deutlicher wird das, durch die folgende Java Quellcodepassage des Endelement-Events:

```
public void endElement(String arg0, String arg1, String arg2)
                throws SAXException {
     // Im Baum wieder nach oben wandern
     current = (DefaultMutableTreeNode) current.getParent();
}
```

Bei einem Parsing-Vorgang bewirkt der SAX Parser zu Beginn eines XML Dokumentes ein StartDocument()-Event. Danach werden Events getriggert, wie bereits in den vorherigen Absätzen beschrieben wurde und in der nachfolgenden Abbildung 4-13 anhand einer formulierten Ergonomieregel zu sehen ist. Als letztes wird am Ende des Dokumentes ein EndDocument()-Event ausgelöst.

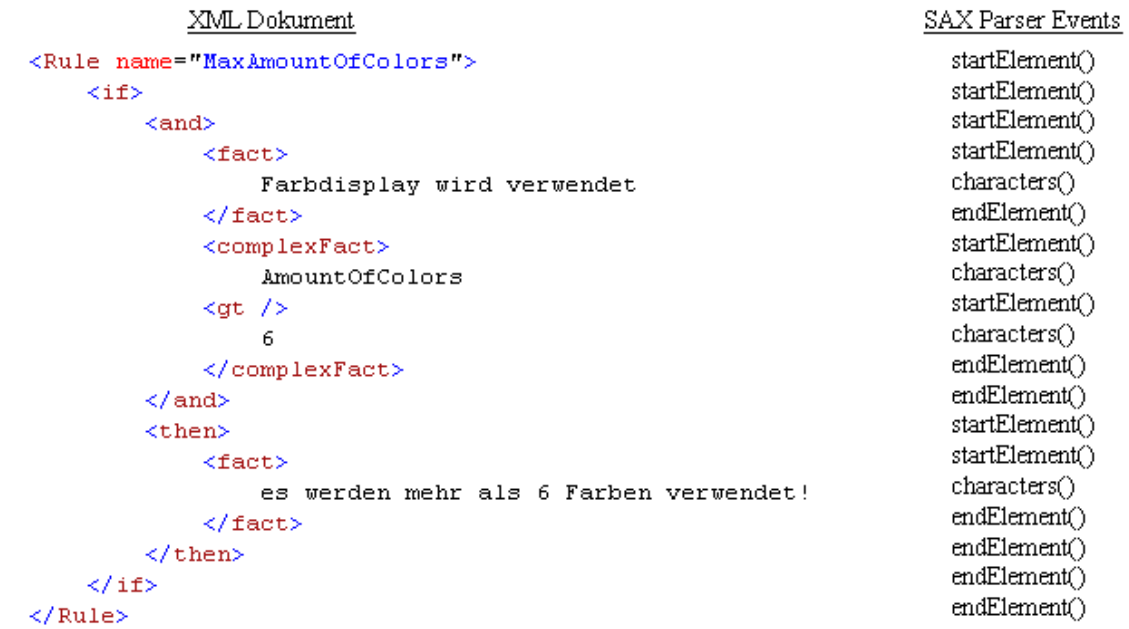

Abb. 4-13: XML Dokument vs. SAX Parser Eventkette

Die durch diesen Regel-Parsing-Prozess gewonnene Baumstruktur spiegelt sehr deutlich eine nahezu identische hierarchische Struktur wie bei dem XML Inhalt der Regeldatei

wieder. Die Abbildung 4-14 soll diese Gleichheit von einer Regel in XML und dem Regelbaum als `javax.swing.Tree` Datenstruktur verdeutlichen.

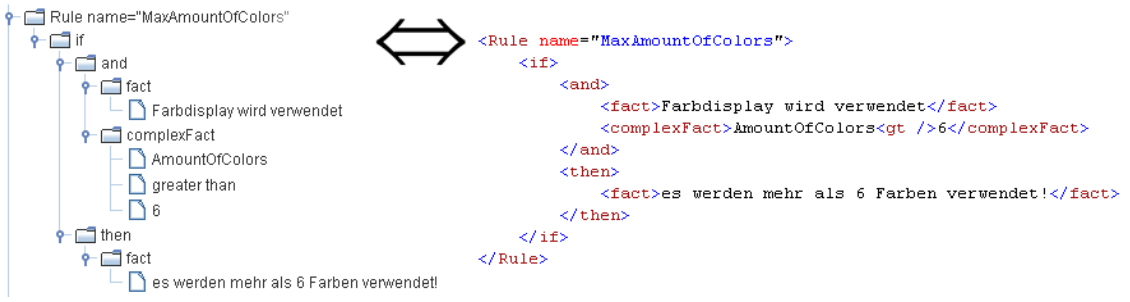

Abb. 4-14: Java Swing Tree vs. XML Rule

In der bis hierhin durch diesen ersten Parsingschritt gewonnenen Struktur lässt sich unter zur Hilfenahme von bekannten Baum-Algorithmen sehr komfortabel navigieren. Die Weiterverarbeitung der Baumstruktur durch die zweite beteiligte Klasse `RuleTreeWalker` zu der geforderten Datenstruktur `Set<Rule>` wird nun im Weiteren erläutert.

Dem Konstruktor der `RuleTreeWalker` Klasse wird zur Instanzbildung eine Instanz der Inferenzmaschine übergeben. Bei genauerer Betrachtung macht die direkte Übergabe einer Inferenzmaschinen-Instanz jedoch nur noch wenig Sinn. Eine deutliche Verbesserung würde durch eine Refaktorierung dieser und den abhängigen Klassen entstehen. Die Aufgabe dieser Klasse sollte sich lediglich darauf beschränken einen Satz von Regeln (`Set<Rule>`) zu erzeugen und zurückzuliefern. Mit diesem Satz könnte dann in die Inferenzmaschine gespeist werden, die daraufhin ihr Arbeit (das Finden von Schlussfolgerungen) erledigen wird. Leider wurde diese Erkenntnis erst zu einem sehr späten Zeitpunkt der Entwicklung gewonnen, aus diesem Grund wurde bisher keine Refaktorierung durchgeführt, allerdings wurde ein Verbesserungsvorschlag bereits in der Datei `IRuleParserSAX.java` als Interface formuliert und in das Package `de.lab4inf.rule.parser` gelegt. Eine konkrete Implementierung dieses Interfaces verbleibt als TODO im Projekt; denn so gravierend wird die Änderung nicht sein, da lediglich die gefundenen (geparsten) Regeln in einer `Set`-Datenstruktur gesammelt werden müssen, anstatt sie direkt der Inferenzmaschine zu übergeben, wie es bisher der Fall ist.

Die von der `RuleTreeBuilder.create()`-Methode zurück gelieferte Baumstruktur wird im Rahmen des SAX Regel Parsing Prozesses (Schritt Nr. 3 in Abbildung 4-11) nun der Klasse `RuleTreeWalker` – genauer gesagt der einzig öffentlichen

Methode `walk(DefaultMutableTreeNode anode)` zur Weiterverarbeitung übergeben.

Die `walk()`-Methode besteht im Wesentlichen aus einem Rekursiven Algorithmus, der an dem Verfahren der Tiefensuche bei Datenbäumen und Graphen angelehnt ist. Dieser Algorithmus prüft einen übergebenen Baumknoten ob er den String „ruleset“ beinhaltet und ob dieser RuleSet-Knoten aktiviert (`ruleNodeIsEnable(TreeNode anode)`) ist. Ist beides der Fall, delegiert der Algorithmus die Weiterverarbeitung an eine Instanz der Klasse `RuleSetWalker`.

Sollte die Überprüfung ergeben, dass es kein RuleSet-Knoten ist, wird kontrolliert, ob der Knoten einen Satz von Fakten (`FactSet`) repräsentiert und diesen Faktensatz ebenfalls aktiviert ist; trifft beides zu würde die Weiterarbeit an die Klasse `FactSetWalker` delegiert. Diese Klasse wurde zuvor mit der Instanz der Inferenzmaschine initialisiert und würde Fakten parsen und diese als Ist-Zustand in das Working Memory der Inferenzmaschine einfügen.

Falls weder ein Satz von Regeln, noch ein Satz von Fakten vorliegen, wird der `walk()`-Algorithmus vom überprüften Knoten ausgehen für jeden vorhandenen Kindknoten rekursiv eine erneute Überprüfung durchführen, solange bis ein Regelsatz gefunden wurde, oder der letzte Kindknoten des Baumes überprüft wurde.

Die beiden Klassen `RuleSetWalker` und `FactSetWalker` sind Ableitungen der abstrakten Klasse `RuleTreeWalkerExpert`. Diese abstrakte Klasse vereint einige (eigentlich alle) Hilfsmethoden, die von beiden abgeleiteten Klassen gemeinsam nutzbar sind. Ebenso gibt diese Klasse eine `abstract void parseNode(TreeNode anode)` Methode vor, die von jeder abgeleiteten Klasse implementiert werden muss. Dadurch ist jede abgeleitete Klasse ist für sich gesehen ein Experte für ein bestimmtes Gebiet.

Die Klasse `FactSetWalker` ist spezialisiert auf die Erstellung von Fakten, wie sie von der API der Inferenzmaschine vorgeschrieben werden. Dazu durchläuft der Algorithmus der `parseNode()`-Methode unter Zuhilfenahme der `javax.swing` API durch die ihm übergebenen Knoten und erstellt durch geeignete `create…Fact()`-Methoden der `RuleFactory` Klasse daraus Fakten der Form `Map<String, Fact>`. Die Factory Klasse wird im Package `de.lab4inf.rule` bereitgestellt. Die Fakten werden (sofern sie von einem `<FactSet>`-Tag eingeleitet wurden) direkt in das Working Memory der Inferenzmaschine gegeben. Das macht im Rahmen dieser Arbeit noch keinen Sinn, da in diesem Fall die betreffenden Daten für das Working Memory aus einem anderen XML Dokument und einem anderen Speicherformat (Java Bean) erfasst werden müssen. Dieser Prozess wird im Kapitel 4.7

genau beschrieben. In Zukunft kann der `FaktSetWalker` allerdings dazu genutzt werden Benutzer- und Hardwarespezifikationen einzulesen, und diese Informationen der Inferenzmaschine als Ausgangsfakten zugänglich zumachen.

4.6.2 DOM-Parsing-Prozess

DOM (Document Object Model) ist der zweite Weg, um XML-Dateien auszuwerten und er stellt ein vom W3C standardisiertes Objektmodell zur Verfügung. Mit dessen Hilfe kann der Inhalt der XML-Datei ausgewertet oder manipuliert werden. Zum Aufbauen des Objektbaumes muss jedoch zunächst die gesamte Datei eingelesen werden, wofür möglicherweise viel Speicher benötigt wird. Vorteilhaft ist hingegen, dass dann alle Elemente direkt in einer hierarchischen Struktur vorliegen und auf alle gleichermaßen zugegriffen werden kann. Die Elemente stehen zueinander in Beziehung (Eltern, Kinder, Geschwister). Als Nachteil von DOM kann sich ein hoher Speicherbedarf erweisen; er verhält sich proportional zur Größe der Eingabedatei.

Die folgende Abbildung 4-15 zeigt die am DOM Parsing Prozess beteiligten Klassen in Form eines UML Diagramms.

Abb. 4-15: UML 2.0 Diagramm: Konzept DOM-Parser

Die beiden Klassen `RuleTreeBuilderDOM` und `RuleParser` sind bewusst auf einer Hierarchiestufe angeordnet, da sie beide nahezu identische Aufgaben vertreten. Beide Klassen sind für das Regel Parsing verantwortlich, wobei die Klasse `RuleTreeBuilderDOM` lediglich für Testzwecke verwendet wird. Einzig die Klasse

`RuleParser` erfüllt dabei die vom JSR-94 vorgegebene Spezifikation. Diese Spezifikation ist in dem Interface `IRuleParser` formuliert und erfordert die Implementierung dreier `create()`-Methoden, die alle den gleichen Datentyp (`Set<Rule>`) zurückliefern. Der Unterschied in diesen drei Methoden besteht daran, dass sie jeweils einen unterschiedlichen Datentypen als Parameter erwarten. Somit kann man Regeln in Form eines Bytestromes (`InputStream`), eines Zeichenstroms (`Reader`) und in Form eines `W3C.Elementes` (also direkt das Rootelement des eingelesenen XML Dokumentes) zur Verarbeitung übergeben. Dabei laufen algorithmisch gesehen, intern alle drei `create()`-Methoden auf ein und dasselbe Verfahren hinaus.

Als erstes wird ein DocumentBuilder benötigt. Dieser wird von einer DocumentBuilderFactory angefordert, die im JDK mitgeliefert wird:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

Anschliessend wird diese `factory` Instanz auf die Zurücklieferung eines validierenden Parsers (DocumentBuilder) vorbereitet:

```
private String path2schemafile = "./ruleplugins/RuleSchema.xsd";

factory.setValidating(true);
factory.setAttribute("http://java.sun.com/xml/jaxp/properties/schema
Language", "http://www.w3.org/2001/XMLSchema");
factory.setAttribute("http://java.sun.com/xml/jaxp/properties/schema
Source", path2schemafile);
```

Diese Zeilen Quellcode sorgen dafür, dass der im Folgenden angeforderte Parser ein validierender Parser ist; der sich also an die XML Schema Sprache vom W3C aus dem Jahr 2001 hält, und zusätzlich auch noch die im Rahmen dieser Arbeit entstandene Regel Schema Grammatik einhält.

Letztlich wird der vorbereitete DocumentBuilder von der Factory angefordert und zur besseren Fehlerberichterstattung an ihm ein ErrorHandler registriert. Derselbe ErrorHandler wird auch bei dem SAX-Prozess verwendet.

```
/*
 * create new document builder
 */
DocumentBuilder docBuilder = factory.newDocumentBuilder();
/*
 * register Error Handler
```

```
 */
docBuilder.setErrorHandler( new RuleTreeBuilderErrorHandler() );
```

Werden nun der `create()`-Methode die Regeln in Form eines `InputStreams`, übergeben, wird intern aus dem InputStream mit Hilfe des `W3C.DocumentBuilder` ein `W3C.Document` erstellt. Anschliessend wird der Inhalt des erstellten Dokuments als ein W3C.Element repräsentiert (`doc.getDocumentElement()`) und der `create(in w3c.Element)`-Methode zur Verarbeitung übergeben.

Sehr ähnlich beläuft sich die Verarbeitung einer Regelbasis, die in Form eines `Readers` (Zeichenstrom) übergeben wird. Aus dem `Reader` wird intern ein `W3C.Document` erstellt, aus dem Dokument wird das Rootelement samt aller Kindknoten als `W3C.Element` extrahiert und danach ebenfalls der Methode `create(in w3c.Element)` übergeben.

Letztendlich läuft jede `create()`-Methode auf den Aufruf der Methode `create(in w3c.Element)` hinaus. Ein `W3C.Element` repräsentiert dabei die Regelbasis in einer sehr ähnlichen Baumstruktur wie sie bereits im Kapitel über den SAX-Parsing-Prozess durch die `RuleTreeBuilder` Klasse erstellt wurde. Mit dem Unterschied, dass der SAX Prozess auf einem „Java Swing Tree“ und der DOM Prozess auf einen „W3C Element Tree“ operiert. Aus diesem Grund wurde eine statische Klasse (`NodeUtil`) entwickelt in der benötige Hilfsmethoden implementiert sind, um geeignete Operationen auf der W3C Element Tree API durchzuführen. Diese Methoden werden im Einzelnen nicht näher erläutert.

Die statische Hilfsklasse `NodeUtil` wird von der Klasse `ElementParser` verwendet. An die Methode `parse(w3c.Element element)` der Klasse `ElementParser` wird das W3C Rootelement übergeben, das dann im Anschluss unter Zuhilfenahme von den `NodeUtils` und der `RuleFactory` der Inferenz-maschine die Regeln in der Form `Set<Rule>` erarbeitet. Die Erarbeitung der Regeln erfolgt durch eine sehr identische algorithmische Logik, wie sie bereits in der `RuleTreeWalker` Klasse und anhängenden Expertenklassen für den SAX-Parsing Prozess programmiert wurde. Einzig die Baumnavigations- und Manipulationsalgorithmen wurden auf die W3C Element API zugeschnitten.

Die folgende Abbildung 4-16 visualisiert grob die zwei Schritte, die für den DOM Parsing Prozess notwendig sind. Eine Regelbasis in Form eines XML Dokumentes wird der Klasse `RuleParser` übergeben, diese Klasse produziert Regeln in der Form eines

`Set<Rule>`, wie sie von der Inferenzmaschine verstanden und verarbeitet werden kann.

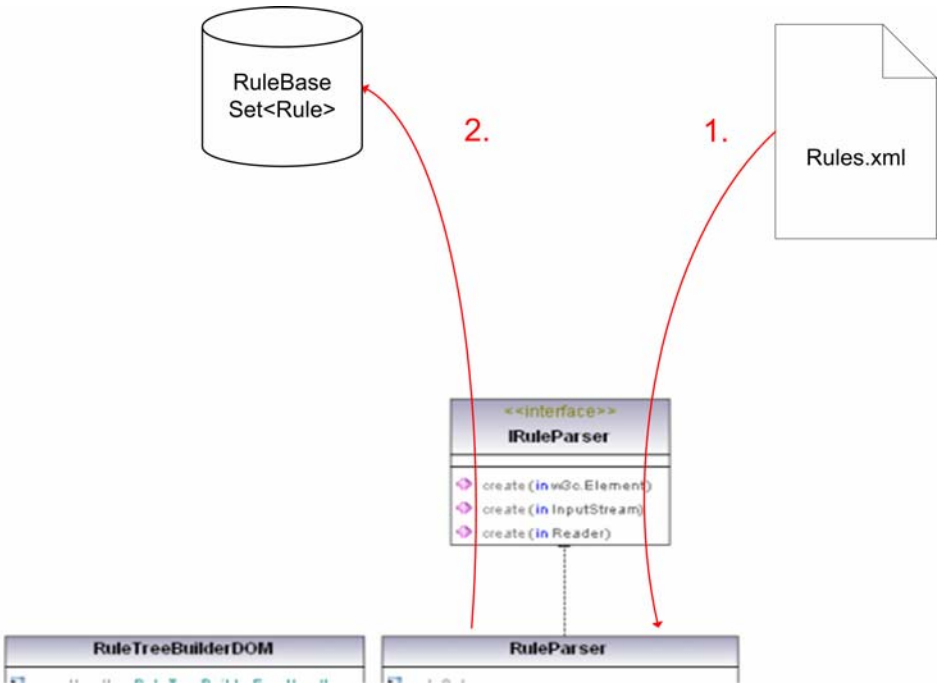

Abb. 4-16: Visualisierung des DOM-Parsing Prozesses

4.6.3 Java Specification Request 94 (JSR-94)

Der Java Community Process stellt mit dem JSR-94 die so genannte Java Rules API vor [http://jcp.org/en/jsr/detail?id=94]. Die Idee hinter dem JSR-94 ist, den Anschluss von Rules Engines an die eigene Software soweit es geht zu standardisieren.

Leider finden sich bereits in der Spezifikation im Unterabschnitt „Scope“ erste Grenzen des Geltungsbereiches der API. Dort heisst es im JSR-94:

Folgende Themengebiete liegen außerhalb dieser Spezifikation:

- Die binäre Repräsentation von Regeln und Regelausführungssätzen
- Die Syntax und das Dateiformat von Regeln und Regelausführungssätzen
- Die Semantik hinter der Interpretation und Ausführung von Regeln und Regelausführungssätzen

- Der Transformationsmechanismus, durch den Regeln und Regelausführungssätze zur Ausführung transformiert werden.

Tatsächlich verbirgt sich hinter diesen wenigen Sätzen eine ziemlich dramatische Konsequenz. Wenn Syntax und Regelbeschreibung nicht durch den JSR-94 definiert werden, ist ein Austausch von Regeln unter Regelmaschinen verschiedener Hersteller praktisch nicht denkbar; bzw. nur durch eine weitere Spezifikation an anderer Stelle. Dies bedeutet auch, dass ein erheblicher Zusatzaufwand entstehen kann, wenn man große Regelbasen nach einem langen Zeitraum der Zusammenstellung zu einem neuen Produkt migrieren will.

Bezogen auf den Rahmen dieser Arbeit liefert der JSR-94 aber auch Spezifikationen die bei der Implementierung zu berücksichtigen sind. So sieht die Spezifikation beispielsweise die drei - bereits zu Beginn des Kapitels 4.6 erwähnten - `create()`-Methoden zur Regelerstellung vor. Dieser Teil der Spezifikation ist in dem Interface `IRulePraser.java` im Package `de.lab4inf.rule.parser` formuliert und wird durch die Klasse `RuleParser` im DOM Parsing Prozess implementiert.

4.6.4 Die Notwendigkeit beider Regel-Parsing Prozessen

Beide Regelparserkonzepte besitzen ihre Berechtigung in der Anwendung. Auch wenn anhand der Abbildungen beider Parsing-Prozesse (Abb. 4-12 und Abb. 4-17) denkbar wäre, dass der DOM Prozess zeitlich gesehen schneller Regeln parsen würde als der SAX Prozess; zwei visualisierte Schritte bei DOM im Vergleich zu vier visualisierten Schritten bei SAX. Zeitmessungen des SAX und DOM Prozesses haben allerdings ergeben, dass der Unterschied beim einlesen der selben Regelbasis bestehend aus ca. 50 Regeln marginal (< 10 ms) ausfällt. Dieser Wert könnte sich in Zukunft bei einer größeren Anzahl Regeln allerdings schnell relativieren, da der Speicherbedarf für das Vorhalten sehr großer XML Dokumente bei dem DOM Prozess proportional ansteigt, und dadurch der DOM Prozess längere Zeit benötigen könnte um ein Dokument in eine W3C Element Tree Hierarchie zu transferieren. Aus diesem Grund wird sich der SAX Parsing Prozess eher für sehr große Regel Dateien eignen, da nicht das gesamte XML Dokument im Arbeitsspeicher vorgehalten werden muss und nur auf die wesentlichen Ereignisse (Events) reagiert wird.

Ausserdem ist für den SAX Prozess bereits ein Parsing Verfahren entwickelt worden, dass es ermöglicht z. B. Benutzer- und Hardware Spezifikationen als Faktenbasis in den Working Memory der Inferenzmaschine einzulesen. Diese Fakten könnten folgendermassen aussehen:

```
<fact>Farbdisplay wird verwendet</fact>
```

oder

```
<fact>Anwender soll bei Betrieb der Anzeige schlafen können</fact>
```

Dieses Spezifikationsfakten-Parsing-Verfahren wurde noch nicht auf den DOM Prozess migriert, es existiert also lediglich für den SAX-Prozess.

Ausserdem bietet der SAX-Prozess die Möglichkeit den eingeparsten Java Swing Baum in einer Java Swing JTree View anzuzeigen. Sollte eine Wissenserwerbskomponente entwickelt werden, könnte diese sehr leicht auf einer JTree View operieren.

Diese Möglichkeit bietet der W3C Element Tree – der vom DOM Parser produziert wird – nicht. Jedenfalls nicht so einfach; ausser man transferiert den W3C Baum in einen JTree und wieder zurück oder man entwickelt eine eigene geeignete Baumdarstellung (View) für einen W3C Baum.

Beide Modelle besitzen aus den genannten Gründen ihre Berechtigung in der Implementierung und Verwendung.

4.7 Faktenbasis (Working Memory)

In diesem Kapitel wird ein mögliches Verfahren erläutert, wie aus der zu bewertenden GUI (im Weiteren auch Testobjekt genannt) Fakten entstehen, mit denen die Inferenzmaschine arbeiten kann.

Ein Testobjekt wurde hierzu mit dem Proof-of-Concept System der DIN-Machine [BA FStolze] erstellt und liegt als eine XML Datei vor, die nach der Java Bean Spezifikation kodierte wurde.

Die Inferenzmaschine erwartet Fakten als eine `HashMap<String, Fact>`, wobei ein `Fact` unterschiedliche Ausprägungen (`StringFact`, `ValueFact`, …) besitzen kann. Näheren Aufschluss darüber gibt die `RuleFactory` Klasse aus dem Package `de.lab4inf.rule`. Diese Factoryklasse wird im Folgenden den Prozess des Faktenparsings bei der Erstellung der Fakten unterstützen.

Nebenbei sei erwähnt, dass es möglich ist den SAX-Parsing Prozess zum Einlesen einer Faktenbasis zu benutzen. Der SAX-Prozess würde Fakten, die innerhalb eines XML-Tag `<FactBase>` und eines XML-Tag `<FactSet>` angeordnet sind, in den Prämissenteil der Inferenzmaschine einlesen. Diese Fakten-Tags besitzen die bekannte Form eines StringFacts (einer einfachen Aussage) eingeleitet mit dem Tag `<fact>`,

oder eines (zusammengesetzten) ValueFact <complexFact>, der Wertzuweisungen angibt.

Eine Eingabe in dieser Form könnte beispielsweise für die Verwaltung der Benutzer- und Hardwarespezifikationen genutzt werden. Diese Art Spezifikationen werden während des zukünftigen Projektverlaufs essentiell, da sie viele weitere Grundvoraussetzungen darstellen, die im Kontext eines Testobjektes mit zu berücksichtigen sind. Zu einer besseren Veranschaulichung seien hier zwei dieser Spezifikationen in XML Syntax genannt:

```
<fact>Farbdisplay wird verwendet</fact>
<fact>Anwender soll bei Betrieb der Anzeige schlafen können</fact>
```

Die zugehörige XML Grammatik (XSD) der FactBase befindet sich im Anhang dieser Arbeit und im Projektordner auf dem lab4inf Sourcenserver.

Diese Art und Weise sei nur erwähnt; sie wird in der Gesamtheit des Projektes bis jetzt noch nicht verwendet, da zur Zeit noch nicht genügend Regeln formuliert sind, die auf derart Fakten sensitiv reagieren.

Vielmehr soll in diesem Kapitel erläutert werden, wie Fakten für die Inferenzmaschine entstehen, die aus Java-Bean-kodierten GUI Testobjekten stammen. Dieses Verfahren wird im weiteren Verlauf „Fakten-Parsing“ genannt.

4.7.1 Fakten-Parsing

Dem Quelltext oder auch der Dokumentation der DIN-Machine kann entnommen werden, dass in der Prototyp Software „DIN-Machine“ derzeit zwischen drei Grundobjekte einer GUI unterschieden wird. Diese Grundobjekte sind:

- Frame (eine Art Hardware in der Anzeigen und User Interface Module angeordnet werden können)
- Display (zur Anzeige von Daten)
- Switch (eine Art Schaltfläche, die Events auslösen kann; derzeit nur Textausgabe)

Jedes Objekt besitzt Attribute. Zurzeit sind das für ein Frame und das Display:

- Der Name
- Der Typ als numerischen Wert
- Die x- und y-Position in cm

- Die Höhe und Breite in cm
- Eine Schraffur (eher uninteressant)
- Eine Hintergrundfarbe
- Und eine Vordergrundfarbe (die derzeit nur von der Schraffur verwendet wird, in Zukunft auch als Farbe der Beschriftung etc.)

Ein Schalter (Switch) besitzt ebenfalls die gerade aufgezählten Attribute und bringt zusätzlich noch ein weiteres Attribut „Funktion“ mit sich, das sich derzeit aber noch auf die Ausgabe eines simplen Textes beschränkt.

Um diese Objekte und Attribute nun in inferenzmaschinenverständliche Fakten zu transferieren, wird die Quell XML Datei zuerst durch einen DOM DocumentBuilder in ein W3C Dokument überführt, wie es bereits in dem DOM Regel Parsing Prozess verwendet wurde. Dieses W3C Dokument ist das Rootelement der XML Quelldatei und repräsentiert die Struktur mit allen Kind- und Kindeskindknoten.

Dem anschliessenden Parser müssen diese Objekten und Attribute bekannt gemacht werden. Das geschieht durch die Datenstruktur einer Map, die zwei Parameter enthält: einen String (z. B. Frame oder Switch) und ein Array von Strings, das mit den zu parsenden Attributen gefüllt wird. Diese Bekanntgabe der Parameter sieht im Quelltext folgendermassen aus:

```
Map<String, String[]> factmap = new HashMap<String, String[]>();

String[] frameArray = {"backgroundColor", "heightCM", "widthCM",
"type", "x1CM", "y1CM", "serialNumber", "add"};
//  factmap<KeyWord, Properties>
//          <String, String[]>
factmap.put("frame", frameArray);

String[] switchArray = {"backgroundColor", "heightCM", "widthCM",
"type", "x1CM", "y1CM", "serialNumber"};
factmap.put("switch", switchArray);
```

Anschliessend wird der Methode `createAllFacts(Document doc, Map<String, String[]>)` einer Instanz des FactBuilderDOMs das zu parsende Testobjekt als W3C Dokument und die zu parsenden Argumente als Map übergeben.

```
factBuilder.createAllFacts(doc, factmap);
```

In der Methode wird über den Inhalt der Factmap iteriert und es werden Fakten in der Form von Aussagen und/oder Wertzuweisungen erzeugt, auf denen die Inferenzmaschine dann die zuvor eingelesenen Regeln anzuwenden versucht. Diese Fakten bestehen nach der Bearbeitung durch die `createAllFacts()`-Methode aus einem Key (z. B. „`switch.backgroundColor`“) und einem zugeordneten Wert „`Color(r=0,g=0,b=153)`“

In Verbindung mit den Produktionsregeln

```
<if>
   <complexFact>switch.backgroundColor<eq />Color(r=0,g=0,b=153)</complexFact>
<then>
     <complexFact>switch.backgroundColor<eq />DARKBLUE<complexFact>
</then>
</if>
```

und

```
<if>
     <complexFact>switch.backgroundColor<eq />DARKBLUE<complexFact>
<then>
     <fact>zu dunkles Blau wird verwendet<fact>
</then>
</if>
```

erzeugt die Inferenzmaschine eine Schlussfolgerung: „zu dunkles Blau wird verwendet“.

Allerdings können nicht alle Fakten so einfach erreicht werden. Für einige Regeln muss ein größerer Aufwand betrieben werden, was anschliessend zu der Erläuterung der Fakten-Sensorik führt.

4.7.2 Fakten-Sensorik

Fakten-Sensorik wird der Prozess genannt, Fakten für die verwendete Inferenzmaschine zu erzeugen, die nicht nach dem Verfahren aus Kapitel 4.7.1 zu erstellen sind. Das sind Fakten die nicht direkt aus dem Quelltext ersichtlich sind; bei denen unter Umständen komplexere Berechnungen erledigt werden müssen. Diese Berechnungen müssen in Java Klassen und somit in den Java Quellcode ausgelagert werden.

Eine Fakten-Sensorik soll anhand eines einfachen Beispiels verdeutlicht werden. Grundlage ist die Regel:

```
<if>
     <complexFact>AmountOfColors<gt />6<complexFact>
<then>
     <fact>es werden mehr als sechs Farben verwendet!<fact>
</then>
</if>
```

Was ein Algorithmus im Rahmen der Fakten-Sensorik erarbeiten muss, ist die Anzahl der auftretenden verwendeten Farben zählen und ein inferenzmaschinenverständliches Faktum erzeugen, bestehend aus einem Key „`AmountOfColors`“ und dem zugeordneten Wert der Anzahl der Farben.

Dieses Ziel kann durch eine relativ kurze Methode erreicht werden, die innerhalb der `createAllFacts()`-Methode aufgerufen wird. Eine Methode `countAmountOfColors()` eignet sich aus dem W3C Dokument des Testobjektes eine Liste aller Knoten an (inklusive aller ihre Kindknoten), bei denen Farbwerte vorhanden sind. Das sind Knoten, die das Attribut `class=“java.awt.Color“` besitzen. Dazu wird eine Hilfsmethode verwendet, die eine Liste von Knoten zurückliefert falls sie ein bestimmtes Klassenattribut besitzen. Als Verarbeitungsparameter werden ein W3C Dokument und einen String übergeben:

```
List<Node> listOfNodes = getElementsByClass(doc, "java.awt.Color");
```

Über den Inhalt dieser Liste wird im Anschluss iteriert und die beinhalteten Farbwerte werden in einer HashMap gespeichert. Eine HashMap sorgt implizit dafür, dass identische Farbwerte nur einen Eintrag produzieren. Abschliessend wird unter Zuhilfenahme der RuleFactory ein ValueFact erzeugt mit dem Key „`AmountOfColors`“ und der Anzahl der gezählten Farben, die sich aus der Größe der HashMap ablesen lässt:

```
factory.createValueFact("AmountOfColors", new Double(
colorHashMap.size() )));
```

Somit hat der hier beschriebene Algorithmus das Faktum erzeugt, sodass die Regel aus dem Ausgangsbeispiel feuern kann.

5 Prototyp Rules Engine Anwenderschnittstelle

In diesem Kapitel wird ein möglicher Prototyp der Rules Engine Anwenderschnittstelle vorgestellt. Dieses soll anhand eines praktischen Beispieles aus der Sicht des Gestaltungsprozesses eines zukünftigen Anwenders des Expertensystems verdeutlicht werden.

Die Ausgangsbasis eines Anwenders ist der gestaltete und gelayoutete Prototyp einer beliebigen Anwenderschnittstelle eines beliebigen (medizinischen) Gerätes; dieser Prozess wird in dem Bespiel mit dem Tool erstellt, das unter dem Arbeitstitel „DIN-Machine“ entwickelt wurde. Der Anwender möge folgende Oberfläche in Abbildung 5-1 generiert haben. Dieses beispielhafte Testobjekt ist bewusst sehr einfach und sehr konstruiert gewählt.

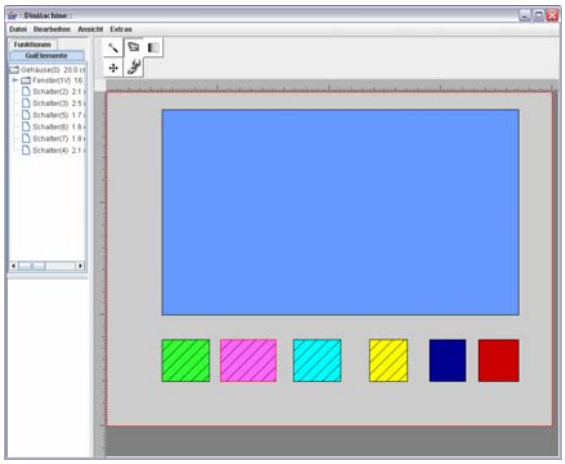

Abb. 5-1: Arbeitsbereich der DIN-Machine einschließlich einer „bunten“ GUI

Nach der Gestaltung eines Testobjektes muss der Prozess zur Regelüberprüfung vorbereitet werden. Dem Rule Engine Modul müssen die Regeln und die Fakten bekannt gemacht werden. Dazu wird der Prototyp der Rule Engine Anwenderschnittstelle gestartet, welches zurzeit noch in einer separaten Anwendung dargestellt wird, also noch nicht mit der Prototypgestaltungssoftware „DIN-Machine“ vereint wurde.

In Abbildung 5-2 ist die Oberfläche dargestellt. Das Hauptaugenmerk sei auf die Statusleiste der Anwendung gelegt, da diese sämtliche benötigten Funktionalitäten bietet um Testdurchläufe von Regeln und Fakten durchzuführen. Ausserdem ist damit

auch eine Alternative zu den automatischen Tests (die in Kapitel 6 erläutert werden) geschaffen worden.

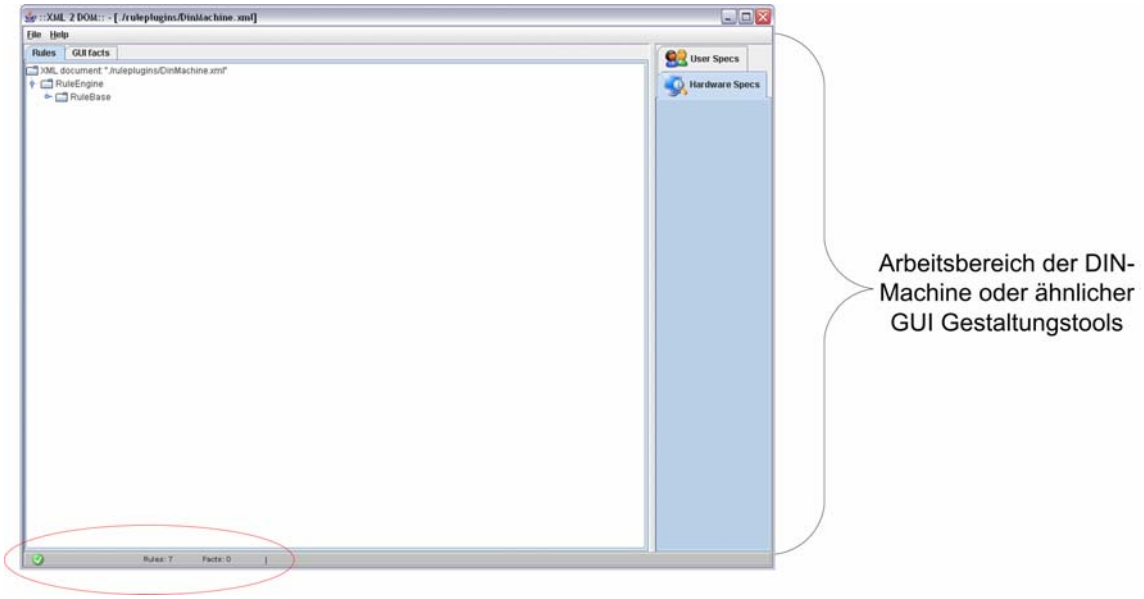

Abb. 5-2: Benutzeroberfläche Rule Engine

Dieses User Interface (UI) ist zu Simulations-, Demonstrations- und Proof-of-Concept-Zwecken entstanden. Die Aufgabe dieser Anwenderschnittstelle beschränkt sich darauf eine benutzerfreundliche Steuerung der grundlegenden Funktionen zu erzielen. Zusätzlich sollte eine alternative Möglichkeit der Ergebnisdarstellung von Regeleinlesevorgänge und GUI Verifikationen geschaffen werden.

Die zugehörigen Quelltexte der Implementierungen befinden sich im Projektordner in dem Package `de.lab4inf.rule.gui` für das Gesamtgerüst der Oberflächen und des Arbeitsbereiches und im Package `de.lab4inf.rule.status` für die Ansicht in der Statusleiste und deren Funktionalität die Rule Engine zu bedienen.

Der Regelbaum - in Abbildung 5-3 auf der linken Seite dargestellt - dient zurzeit lediglich der Darstellung der eingelesenen Regeln. Die Implementierung von Editiermöglichkeiten der Regeln im Baum sowie eventuell die Möglichkeit von Regelassistenten (ähnlich der Formelassistenten von gängigen Tabellenkalkulations-programmen) Gebrauch zu machen, fallen unter den Gesichtspunkt der Wissensakquise, das genug Stoff für ein eigenständiges Teilprojekt bietet. Ebenso fallen in dieses Thema die beiden Dialog-Reiter auf der rechten Seite der Arbeitsfläche. Diese Dialoge sind leer, da die Eingabemöglichkeit für Benutzer- und Hardwarespezifikationen lediglich angedeutet ist. Sie bietet aus diesem Grund noch keine Funktionalität, weil noch nicht

genügend Spezifikationen definiert wurden und ausserdem noch keine genügend grosse Anzahl Regeln in der Regelsprache formuliert und transferiert wurden.

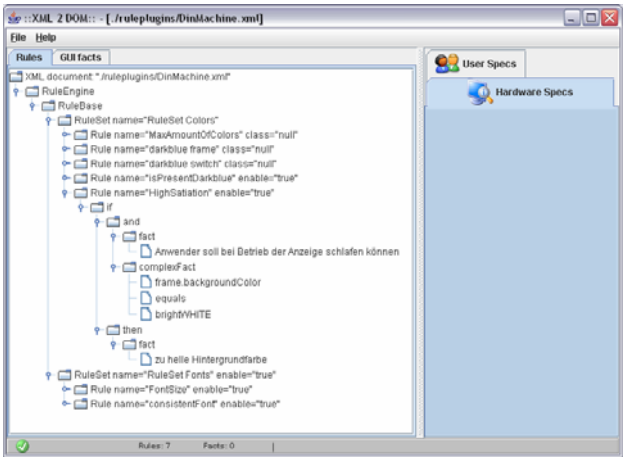

Abb. 5-3: Regelbaum als Beispiel für Wissensakquise

Allgemeine Informationen über den Status der Regel Engine liefert die Statusleiste indem man mit der Maus über die Leiste fährt. Dieses Ereignis löst die Erstellung und Anzeige eines Popup Fensters aus, welches bei weiteren Mausbewegungen automatisch wieder geschlossen wird. Die allgemeinen Informationen bestehen aus der Anzahl der eingelesenen Regeln, aus der Anzahl der erzeugten Fakten aus dem Testobjekt und im weiteren Verlauf aus der Anzahl der getroffenen und gefundenen Schlussfolgerungen nach der Verifikation des Testobjektes. Diese allgemeinen Statusinformationen sind in Abbildung 5-4 zu sehen.

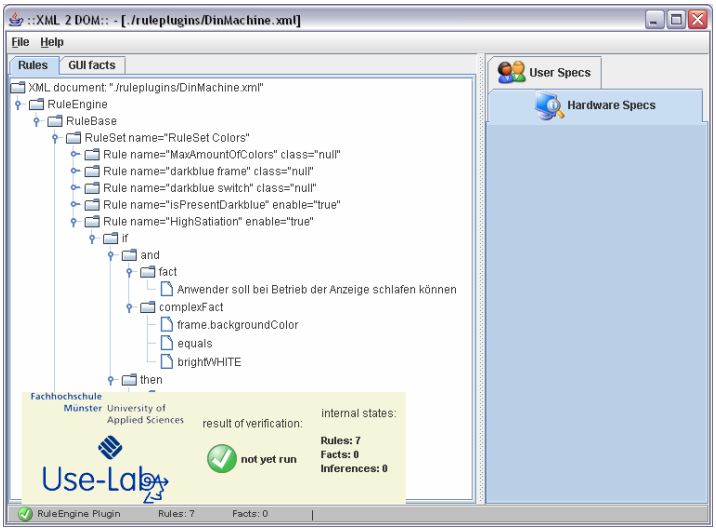

Abb. 5-4: Popupfenster Rule Engine Informationen

Jede Funktionalität ist über das Kontextmenü erreichbar, dessen Inhalt in Abbildung 5-5 zu sehen ist.

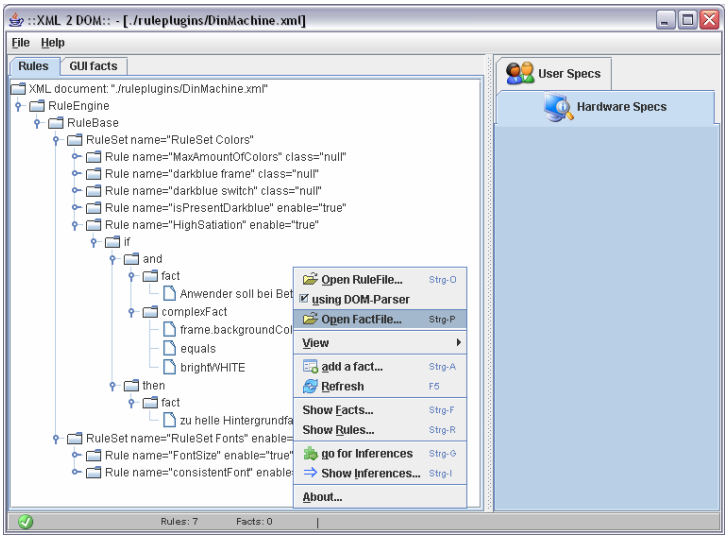

Abb. 5-5: Kontextmenü der Rule Engine UI

Über das Kontextmenü und den Menüpunkt „Open Fact File“ wird nun das Testobjekt aus Abbildung 5-1 in Form einer Java Bean kodierten XML Datei der Rule Engine Schnittstelle bekannt gegeben. Dieses Testobjekt wird durch den Controller (Klasse

`InferenceAdapter.java`) des Rule Engine Status Panel an die Faktensensorik (Kapitel 4.7) weitergeleitet und verarbeitet. Durch diesen Schritt entstehen Fakten für eine Faktenbasis auf die Regeln angewendet werden können.

Die Faktenbasis kann manuell durch den Anwender erweitert werden. Dieses ermöglicht der Menüpunkt „add a fact“ aus dem Kontextmenü. Daraufhin öffnet sich das Dialogfenster, das in Abbildung 5-6 zu sehen ist.

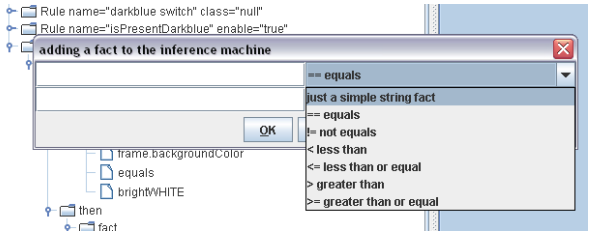

Abb. 5-6: Dialogfenster zum expliziten Hinzufügen von Fakten

In Abbildung 5-7 ist zu sehen, wie ein Anwender die Aussage dass ein Farbdisplay verwendet wird, über das Dialogfenster als Fakt der Faktenbasis hinzufügt.

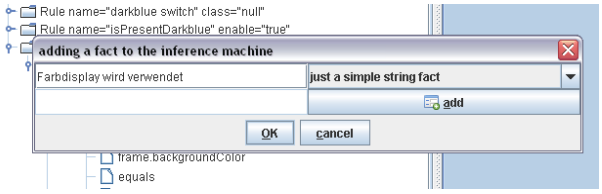

Abb. 5-7: Dialogfenster: Hinzufügen einer Aussage als StringFakt

Nach diesem Schritt sind sowohl eine Regelbasis als auch eine Faktenbasis vorhanden. Allerdings sind noch keine Ergebnisse einer Überprüfung einsehbar, wie in Abbildung 5-8 zu erkennen ist.

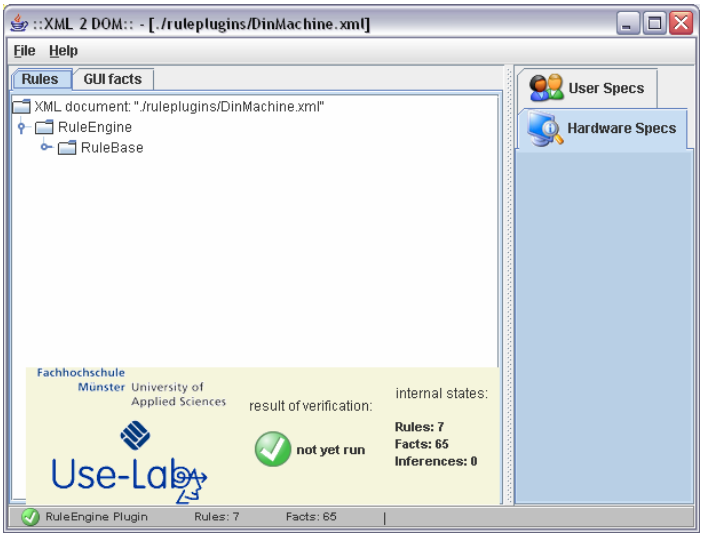

Abb. 5-8: Visualisierung des DOM-Parsing Prozesses

Die Durchführung einer Prüfung muss von dem Anwender zurzeit noch explizit gestartet werden, welches durch zwei Menüpunkte möglich ist. Durch den Menüpunkt „Show Inferences“ wird zuerst der Inferenzalgorithmus gestartet und das Ergebnis wird in Form von einfachen Aussagen in einem Dialogfenster angezeigt. Dieses Dialogfenster repräsentiert eine Art Erklärungskomponente, in der allerdings zurzeit noch keine Regelschlussfolgerungen erläutert werden. Diese Komponente bedarf ebenfalls einer grossen Regelmenge und darauf abgestimmter Erläuterungen. Dieses Dialogfenster mit den Ergebnissen der Überprüfung ist in Abbildung 5-9 zu sehen.

Abb. 5-9: Übersicht der Ergebnisse in einer Pseudoerklärungskomponente

Alternativ können die ersten fünf gefundenen Schlussfolgerungen auch in dem Popupfenster „Informationsübersicht“ durch ein MouseOver-Ereignis über den Statusleistenbereich der Anwendung eingesehen werden, wie in Abbildung 5-10 zu sehen ist.

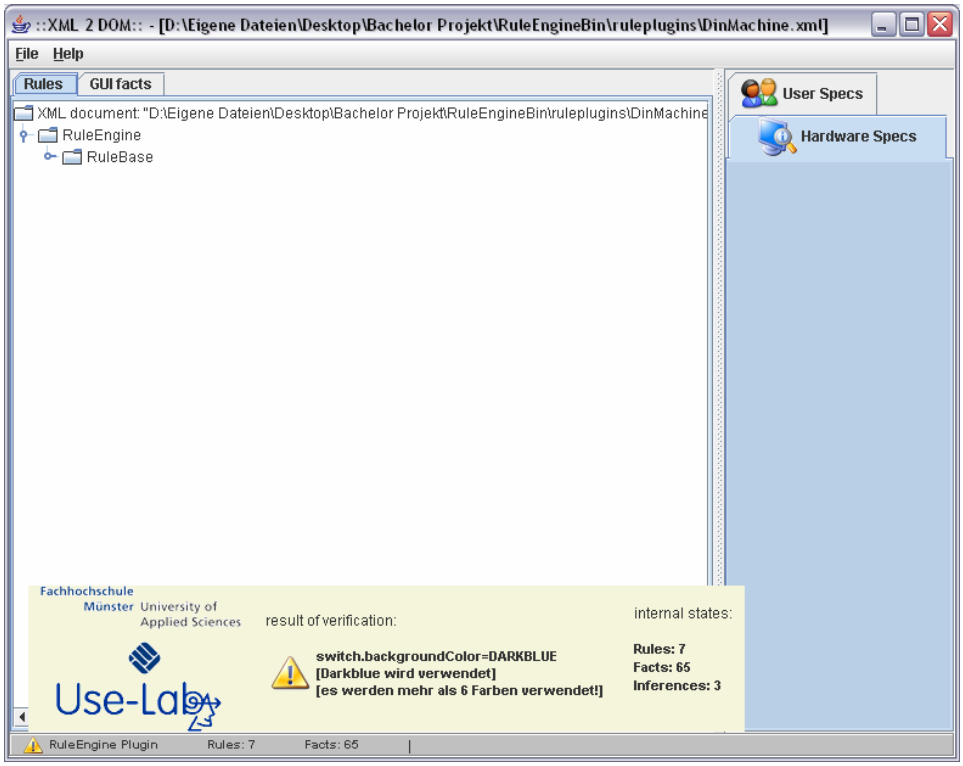

Abb. 5-10: Übersicht und Ergebnisse dargestellt als Popup

Später könnte eine Regelüberprüfung zum Beispiel bei jedem vom Benutzer durchgeführten Speichervorgang automatisch durchgeführt werden; oder als weitere Variante könnte sie gestartet werden, wenn der Anwender eine Art Vorschaufenster öffnet.

Schlussfolgerungen lassen sich klassifizieren, indem man in der XML-Syntax der Regelsprache dem Konklusionstag `<then>` ein Attribut anhängt. Dadurch wird es möglich Schlussfolgerungen einer Kategorie zuzuordnen. Die XSD Regelgrammatik einer Konklusion besitzt, wie im Anhang dieser Arbeit zu sehen ist, folgende Kategorien: „`notice`“, „`warning`“, „`error`“ und „`fatal error`“. Das sind die Werte, die das Attribut `declare` annehmen kann. Daraufhin könnte der Inferenzmechanismus aus den Schlussfolgerungen eine priorisierte Ausgabe erzeugen, sodass immer zuerst die fatalen Fehler, danach die Fehler, die Warnungen und zum Schluss nur einfache Hinweise angezeigt werden.

Zurzeit wird lediglich, sobald Schlussfolgerungen gefunden wurden, eine Ikone dargestellt. Diese soll durch ein gelbes Warndreieck mit einem Ausrufezeichen eine Warnung repräsentieren. Anhand der priorisierten Ausgabe könnten dann auch unterschiedliche Ikonen angezeigt werden um den Ergebnisstatus zu visualisieren.

Die Funktionalität der Klassifizierung wird zurzeit lediglich in der XSD Regelsprachengrammatik berücksichtigt, und wird weder von der Inferenzmaschine noch von der Rules Engine Plugin Darstellung verarbeitet.

Die Optik der Rules Engine Benutzerschnittstelle sowie die Verwendung der Klassifizierungen orientieren sich bewusst an einer bereits sehr verbreitete Software, die zu Validierungszwecken von HTML Internetseiten von vielen Webdesignern verwendet wird. Diese Software ist bekannt unter dem Namen „Tidy“ [http://users.skynet.be/mgueury/mozilla/] und kann als Plugin in den Firefox Browser integriert werden. Dieses Tool überprüft Regeln, zeigt Schlussfolgerungen, Begründungen und demonstriert Lösungsvorschläge. Es könnte ebenfalls nach dem Konzept eines regelbasierten Expertensystems entwickelt worden sein, wobei in diesem Fall die Regeln selbstverständlich einfacher zu formulieren sind, da nur harte Fakten (W3C HTML Regeln) formuliert werden müssen.

6 Test und Verifikation der Wissensbasis Implementierung

Dieses Kapitel soll zeigen das und wie die getätigten Implementierungen gegenüber den Anforderungen verifiziert wurden.

Dazu sei erwähnt, dass in dem Quellcode Projektordner zu jedem Package (Ansammlung von Klassen) ein zugehöriges Testpackage existiert. In den Testpaketen ist für jede real existierende, implementierte Javaklasse eine JUnit Testklasse vorhanden. Testklassen werden von einer JUnit Collectorklasse gesammelt und danach ein Test nach dem anderen durchgeführt. JUnit Tests sind kleine atomare Tests, die Algorithmen von Methoden in Java Klassen testen. Das Ergebnis eines dieser Durchläufe sei in Abbildung 6-1 dargestellt.

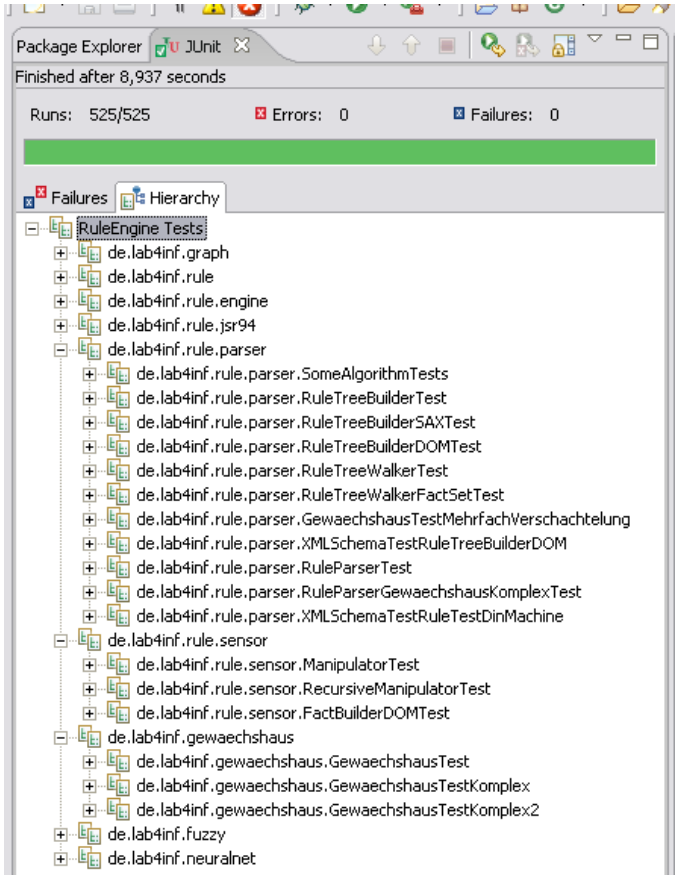

Abb. 6-1: Ergebnis eines JUnit Testdurchlaufes

Zusätzlich zu den atomaren JUnit Tests existieren komplexere ganzheitliche Tests an denen die Funktionalität der Rules Engine getestet wurde. Einer dieser ganzheitlichen Tests wurde ausführlich in Kapitel 5 dieser Arbeit erläutert und durchgeführt.

Weitere ganzheitliche Tests sind in dem Package `de.lab4inf.gewaechshaus` implementiert. In diesen Tests wird gezeigt, dass die vorliegende Inferenzlogik auch für Regeln aus anderen Kontexten praktikabel und einsetzbar ist. In den Gewächshaustests sind Regeln formuliert wie „*Wenn es heiss ist und eine hohe Feuchtigkeit vorliegt, dann muss nur wenig gegossen werden*“. Von dieser Art Regeln existieren beispielhaft 25 Ausprägungen, die durch zehn weitere Regeln ergänzt wurden, die einen gegebenen Temperaturwert auf die Aussagen „*es ist heiss*“ und Feuchtigkeitswerte auf „*es ist feucht (sehr feucht, trocken, etc)*“ herunterbrechen. Anhand dieser Tests liess sich sehr anschaulich die Funktionsweise der Inferenzmaschine und der Parsingroutinen testen; und sie zeigen ausserdem, dass Geschäftslogiken, die sich durch Produktionsketten formulieren lassen, durch die vorhandene Rules Engine korrekt verarbeitet werden können.

Sicherlich muss zugegeben werden, dass derart ganzheitliche Tests für ein zügiges Debugging und eine schnelle Fehlerfindung eher kontraproduktiv sind, sie aber eine sehr anschauliche Ergänzung zu den atomaren JUnit Tests darstellen.

Eine Verifikation der erstellten ergonomischen Regeln lässt sich sicherlich nur im Kontext eines GUI Prototypen zeigen, was die Demonstration im Kapitel 5 darstellt. Eine Validierung der XML Syntax findet quasi im Hintergrund während des DOM Parsing Prozesses statt, da dieser einen validierenden Parser benutzt, der die Regelsyntax mit den in XSD formulierten Regelgrammatiken vergleicht.

Eine Validierung der XSD Regelgrammatik Dateien findet ebenfalls implizit durch den DOM Parsing Prozess statt. Es kann aber auch explizit validiert werden, wozu auf diversen Internetseiten Möglichkeiten angeboten werden; unter anderem bietet das World Wide Web Consortium einen Validierungsdienst auf ihrer Internetpräsenz unter http://www.w3.org/2001/03/webdata/xsv an. Hier kann man XSD Schema Dateien hoch laden und nach deren strikten Regeln überprüfen lassen. Die Ergebnisse dieser Durchläufe seien in den Abbildungen 6-2 und 6-3 für die erstellten Fakten und Regel Schemata dargestellt.

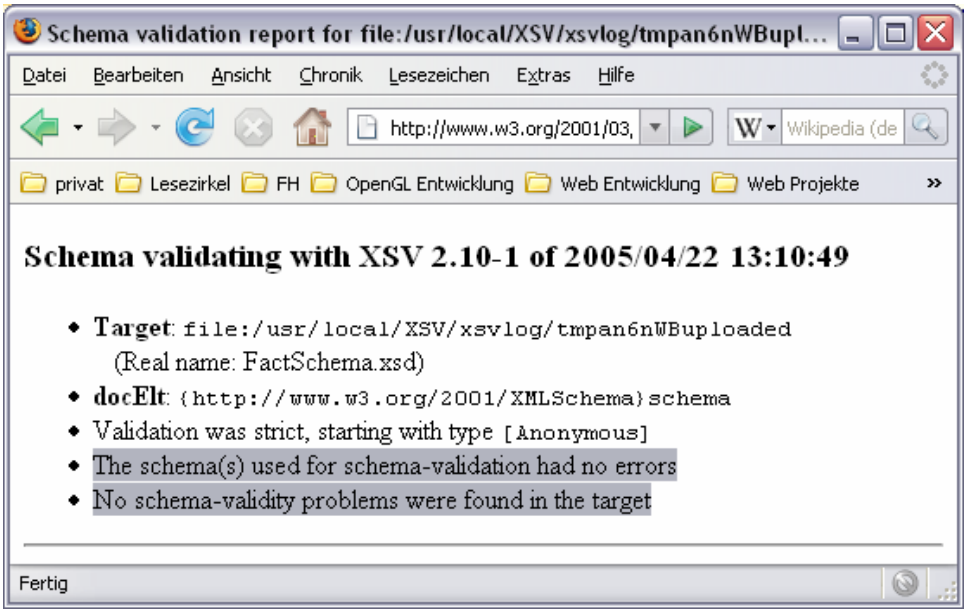

Abb. 6-2: W3C Ergebnis Validierung der XSD Fakten Schema Grammatik

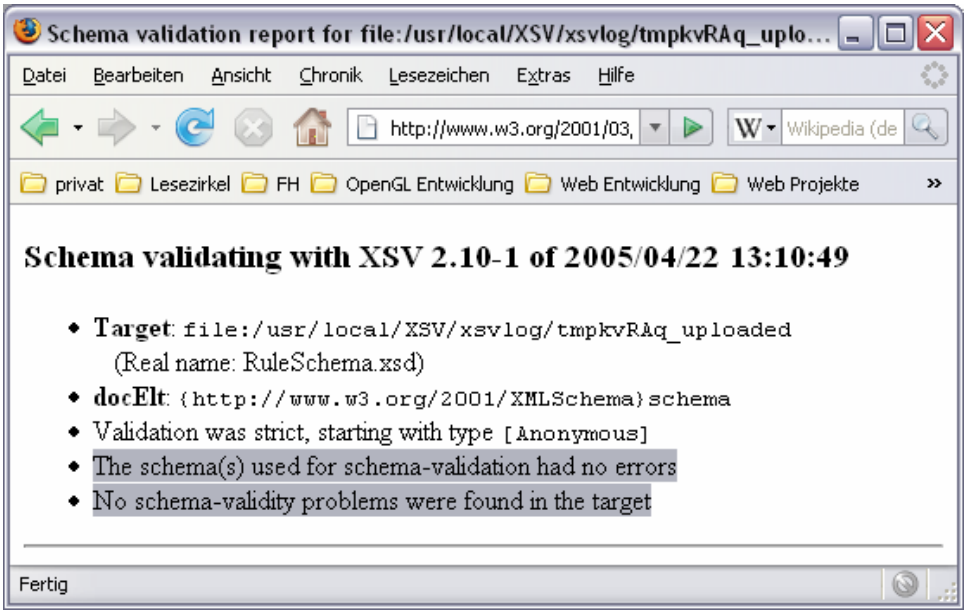

Abb. 6-3: W3C Ergebnis Validierung der XSD Regel Schema Grammatik

Einen Schritt weiter geht der Validierungsdienst der Firma DecisionSoft Limited auf deren Internetpräsenz man unter http://tools.decisionsoft.com/schemaValidate/ nicht nur die reinen Schema Dateien validieren lassen kann, sondern zusätzlich auch eine vorhandene XML Instanz (die im Kontext dieser Arbeit entstandenen ergonomischen GUI Regeln aus der Datei `DinMachine.xml`) angeben kann. Dazu muss allerdings erwähnt werden, dass aus den beiden XSD Grammatiken `FactSchema.xsd` und `RuleSchema.xsd` eine gemeinsame Datei `SchemaKomplett.xsd` erstellt wurde, da nur eine Schema Datei angegeben werden kann.

Das Ergebnis der positiven Validierung sei abschliessend in Abbildung 6-4 dargestellt.

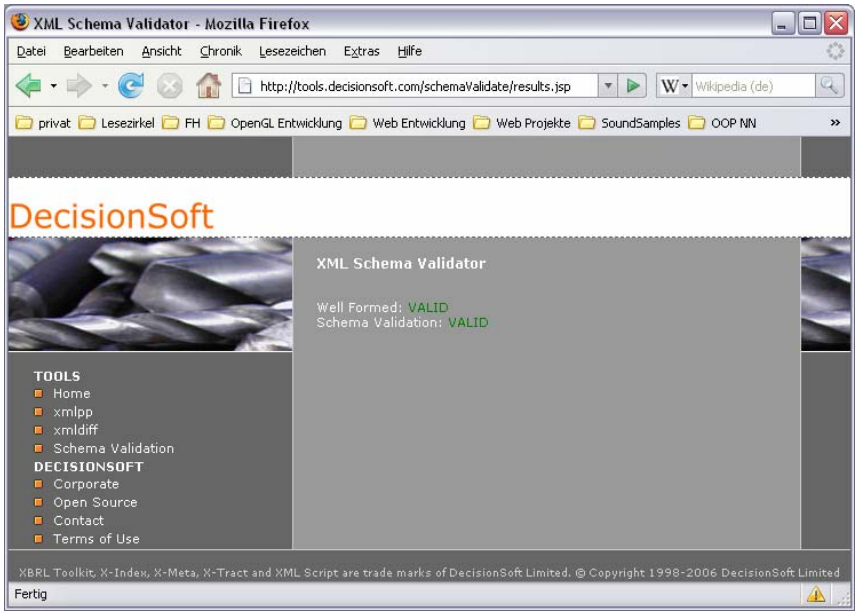

Abb. 6-4: Ergebnis Validierung Schema und Instanz durch DecisionSoft

7 Ergebnisse der Arbeit / Ausblick und Fortführung

Abschliessend lässt sich sagen, dass die Verwendung eines Inferenzalgorithmus und einer XML Regelsprache eine elegante Art und Weise darstellen um Regeln automatisch überprüfen zu lassen und sie durch einen Anwender zur Laufzeit zu verändern, anzupassen und erweitern zu können. Diese Erweiterungen der Regelbasis lassen sich annähernd natürlich sprachlich und ohne grosse Programmierkenntnisse durchführen. Voraussetzung für die Eingabe von Regeln in einer natürlich sprachlichen Beschreibung ist die vorherige Schaffung einer darauf ausgelegten Faktenbasis, wodurch sich dann die Hauptarbeit auf die Erstellung einer geeigneten Faktensensorik verschiebt, da Berechnungen für komplexere Zusammenhänge in diese Sensorik ausgelagert werden müssen. Diese Faktensensorik ist nicht zu vernachlässigen und nicht zu unterschätzen und erfordert in der zukünftigen Weiterentwicklung des geplanten Expertensystems einen enormen Fokus.

Die Anforderungen des Projekts konnten mit Hilfe der vorgestellten Techniken und Komponenten prototypisch erfüllt werden. Ebenso gilt es als erwiesen, dass das Verfahren regelbasierte Expertensysteme mit Feed-Forward-Driven (vorwärts verkettender) Inferenzalgorithmen als Lösungsverfahren für eine automatische Überprüfungen ergonomischer Normen an grafischen Benutzeroberflächen eingesetzt werden kann.

Eine Verbesserung der Resultate könnte vor allem dadurch erzielt werden, wenn die Regelsyntax und die Inferenztechnologie um einige Elemente erweitert werden. So könnte von großem Vorteil sein, wenn die Regelsyntax die Logik ersten Grades unterstützt, vor allem der Gebrauch von Prädikaten und dem Allquantor ∀ wäre für eine Vielzahl von Regeln sehr hilfreich, wie zum Beispiel „Für alle (Button.Font)… existiert (Font<eq />Arial)“, das wäre eine sehr einfache Beschreibung für die Styleguide-Regel: „Einheitliche Schriftart soll verwendet werden“.

Ebenfalls wäre die Möglichkeit erstrebenswert eine Objekt-(Entity-)-Attribut-Modellierung in der Regelsyntax und auch in der Inferenztechnologie nutzen zu können, wie es das Open Source Produkt Drools ermöglicht, um vorhandene Objekthierarchien sehr genau darstellen zu können. Diese Implementierungen befinden sich aber bereits in der Planung und Entwicklung und konnte aus Zeitmangel nicht mehr getestet und integriert werden.

Aufbauend auf eine Objekthierarchie sollte sich der Inferenzformalismus bei gefeuerten Regeln und gefundenen Schlussfolgerungen merken können, aufgrund welcher Regel

eine bestimmte Schlussfolgerung gefunden wurde. Ausserdem sollte die Inferenzmaschine angeben können aufgrund welchen Faktes an welchem Objekt eine Schlussfolgerung stattgefunden hat. Diese Informationen sollten dann an eine Erklärungskomponente geleitet werden, die diese Daten dann sinnvoll ausgibt, sodass dem Anwender Erläuterungen und Problemlösungsstrategien angeboten werden.

Ein wichtiger Schritt bei der Weiterentwicklung des Gesamtprojektes „DIN-Machine inklusive einer Rules Engine“ wird es sein im Projektteam ein geeignetes GUI Gestaltungsprogramm zu verabschieden oder ein eigenes Tool entwickeln (lassen). Das sollte eine Software sein, die erst einmal den Anforderungen einer Benutzeroberfläche für medizinisch-technische Geräte gerecht wird. Später wäre es auch noch denkbar, die „DIN-Machine“ um Regeln zu erweitern, die allgemeine Software Dialogfenster bewertet. Eine zur Entstehungszeit dieser Arbeit aktuelle Beschreibungssprache für grafische Oberflächen bietet eine an XML angelehnte Sprache mit dem Namen XAML. Eine kurze Einführung ist unter http://www.medical-usability.de/aufsaetze/einfuehrung_in_xaml.pdf einsehbar. Diese Sprache ist aktuell von der Firma Microsoft entwickelt worden für die Beschreibung und Erstellung von Software Oberflächen für die Windows Presentation Foundation und ist ein Grundpfeiler der neuen WinFX-API des Betriebssystems Windows Vista, kann aber auch von Windows XP verwendet werden, sofern das .NET Framework 3.0 installiert ist.

Diese Sprache könnte für die Gestaltung von medizinisch-technischen Oberflächen geeignet sein, da sie alle bekannten und gängigen GUI Elemente und deren Attribute bereits kennt. Zusätzlich ist es jedoch möglich jegliche Kombinationen unter den Elementen zu bilden, dadurch könnten eigene Elemente entstehen die medizinisch-technischen Baugruppen entsprächen. Für diese GUI Beschreibungssprache existieren bereits einige Zeichenprogramme (Layouter), bei denen es nur noch eine Frage der Zeit sein wird, dass sie auf alle gängigen Betriebsysteme und Plattformen portiert werden.

Auf diese XAML Syntax ausgelegt könnte sich die Softwareentwicklung auf eine geeignete Fakten-Sensorik und die Umsetzung und Formulierung weiterer ergonomischer GUI Regeln konzentrieren, sowie die Erklärungskomponente (vielleicht mit einer angebundenen Datenbank) ausbauen.

Literaturverzeichnis

[ALTENKRÜGER 1992] Altenkrüger, Doris; Büttner, Wilfried:
Wissensbasierte Systeme: Architektur, Entwicklung, Echtzeit-Anwendungen – Eine Praxisgerechte Einführung
Vieweg, 1992

[BEHRENDT 1990] Behrend, Reinhard (IBM Deutschland GmbH):
Angewandte Wissensverarbeitung: Die Expertensystemtechnologie erobert die Informationsverarbeitung
Oldenbourg Verlag München Wien, 1990

[BRUNS 1990] Bruns, F. Wilhelm:
Künstliche Intelligenz in der Technik: Eine praxisnahe Einführung,
Hanser Verlag München Wien, 1990

[CHOMSKY 1963] Noam Chomsky, Marcel P. Schützenberger:
The algebraic theory of context free languages, Computer Programming and Formal Languages
(P. Braffort and D. Hirschberg ed.), North Holland, Amsterdam, 1963, 118-161.

[CLAUS 2003] Claus, V. und A. Schwill
Duden Informatik – Ein Fachlexikon für Studium und Praxis.
Dudenverlag, Mannheim, 2003

[DREYFUSS 1987] Dreyfuss, H.L. & S.E.:
Künstliche Intelligenz, Von den Grenzen der Denkmaschine und dem Wert der Intuition
Rowohlt Taschenbuchverlag, 1987

[GOTTLOB 1990] Gottlob, G., T. Frühwirth und W. Horn
Expertensysteme.
Springer Verlag, Wien 1990

[HARTMANN 1990] Hartmann, Dietrich; Lehner, Karlheinz:
Technische Expertensysteme: Grundlagen, Programmiersprachen, Anwendungen,
Springer-Verlag Berlin Heidelberg 1990

[JAVABEAN 1994] http://java.sun.com/products/javabeans/docs/spec.html

[KURBEL 1992] Kurbel, Karl:
Entwicklung und Einsatz von Expertensystemen: Eine anwendungsorientierte Einführung in wissensbasierte Systeme
Springer-Verlag, 1992

[NEWELL 1972] Newell, Allen; Simon, Herbert:
Human Problem Solving
Englewood Cliffs, Prentice-Hall, 1972

[ROJAS 1996] Rojas, Raúl:
Theorie der neuronalen Netze. Eine systematische Einführung
Springer-Verlag Berlin, 1996

[SCHNUPP 1988] Schnupp, Peter; Leibrandt, Ute:
Expertensysteme: Nicht nur für Informatiker,
Springer-Verlag Berlin Heidelberg, 1988

[TURING 1950] Turing, Alan:
Computing machinery and intelligence

[WEIZENBAUM 1966] Weizenbaum, Joseph:
„ELIZA - A Computer Program For the Study of Natural Language Communication Between Man And Machine“ in: Communications of the ACM, 9(1).
(1966)

[WUNDERLICH 2006] Wunderlich, Lars:
Java Rules Engines: Entwicklung von regelbasierten Systemen
entwickler.press, 2006

Anhang

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ************************************************ -->
<!-- *   general stuff                                * -->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:include schemaLocation="FactSchema.xsd" />

<xsd:annotation>
    <xsd:documentation xml:lang="DE">
        Rule Schemata / Rules Language
        fuer de.lab4inf.rule.engine
        Copyright 2006 de.lab4inf. Alle Rechte vorbehalten.
    </xsd:documentation>
    <xsd:documentation xml:lang="EN">
        Rule Scheme / Rules Language
        for the de.lab4inf.rule.engine
        Copyright 2006 de.lab4inf. All rights reserved.
    </xsd:documentation>
</xsd:annotation>


<!-- ************************************************ -->
<!-- *   the root element of a xml rules file         * -->

<xsd:element name="RuleEngine">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="RuleBase" type="RuleBaseType" minOccurs="1" />
            <xsd:element name="FactBase" type="FactBaseType" minOccurs="0" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>


<!-- ************************************************ -->
<!-- *   scheme for the RuleBase, RuleSet and Rules   * -->

<xsd:element name="RuleBase" type="RuleBaseType" />
    <xsd:complexType name="RuleBaseType">
        <xsd:choice>
            <xsd:element name="RuleSet" type="RuleSetType" minOccurs="1" maxOccurs="unbounded" />
            <xsd:element name="Rule" type="RuleType" minOccurs="1" maxOccurs="unbounded" />
        </xsd:choice>
    </xsd:complexType>

<xsd:element name="RuleSet" type="RuleSetType" />
    <xsd:complexType name="RuleSetType">
        <xsd:sequence>
            <xsd:element name="Rule" type="RuleType" minOccurs="1" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="name"   type="xsd:string"  use="required" />
        <xsd:attribute name="enable" type="xsd:boolean" use="optional" />
    </xsd:complexType>

<xsd:element name="Rule" type="RuleType" />
    <xsd:complexType name="RuleType">
        <xsd:sequence>
            <xsd:element name="if" type="ifType" minOccurs="1" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="name"   type="xsd:string"  use="required" />
        <xsd:attribute name="enable" type="xsd:boolean" use="optional" />
    </xsd:complexType>
```

```
<xsd:element name="if" type="ifType" />
    <xsd:complexType name="ifType">
        <xsd:sequence>
            <xsd:choice>
                <xsd:element name="fact" type="factType" />
                <xsd:element name="simpleFact" />
                <xsd:element name="complexFact" type="complexFactType" />
                <xsd:element name="and" type="andType" />
                <xsd:element name="or"  type="orType" />
                <xsd:element name="not" type="notType" />
            </xsd:choice>
            <xsd:element name="then" type="thenType" minOccurs="1" maxOccurs="1" />
        </xsd:sequence>
    </xsd:complexType>

<xsd:element name="then" type="thenType" />
    <xsd:complexType name="thenType">
        <xsd:choice>
            <xsd:element name="fact" type="factType" minOccurs="1" maxOccurs="1" />
            <xsd:element name="simpleFact" />
            <xsd:element name="complexFact" type="complexFactType" minOccurs="1" maxOccurs="1" />
        </xsd:choice>
        <xsd:attribute name="declare" use="optional">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="notice" />
                    <xsd:enumeration value="error" />
                    <xsd:enumeration value="fatal error" />
                    <xsd:enumeration value="warning" />
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>

<xsd:element name="and" type="andType" />
    <xsd:complexType name="andType">
        <xsd:choice minOccurs="2" maxOccurs="unbounded">
            <xsd:sequence>
                <xsd:element name="fact" type="factType" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="simpleFact" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="complexFact" type="complexFactType" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="or" type="orType" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="not" type="notType" minOccurs="0" maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:choice>
    </xsd:complexType>

<xsd:element name="or" type="orType" />
    <xsd:complexType name="orType">
        <xsd:choice minOccurs="2" maxOccurs="unbounded">
            <xsd:sequence>
                <xsd:element name="fact" type="factType" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="simpleFact" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="complexFact" type="complexFactType" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="and" type="andType" minOccurs="0" maxOccurs="unbounded" />
                <xsd:element name="not" type="notType" minOccurs="0" maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:choice>
    </xsd:complexType>

<xsd:element name="not" type="notType" />
    <xsd:complexType name="notType">
        <xsd:choice>
            <xsd:element name="simpleFact" />
            <xsd:element name="complexFact" type="complexFactType" minOccurs="1" maxOccurs="1" />
            <xsd:element name="fact" type="factType" />
            <xsd:element name="and"  type="andType" />
            <xsd:element name="or"   type="orType" />
            <xsd:element name="not"  type="notType" />
        </xsd:choice>
    </xsd:complexType>


</xsd:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ************************************************ -->
<!-- *   general stuff                                * -->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:annotation>
    <xsd:documentation xml:lang="DE">
        Fact Schemata / Rules Language
        fuer de.lab4inf.rule.engine
        Copyright 2006 de.lab4inf. Alle Rechte vorbehalten.
    </xsd:documentation>
    <xsd:documentation xml:lang="EN">
        Fact Scheme / Rules Language
        for the de.lab4inf.rule.engine
        Copyright 2006 de.lab4inf. All rights reserved.
    </xsd:documentation>
</xsd:annotation>

<!-- ************************************************ -->
<!-- *   scheme for the FactBase and the FactSet(s)   * -->

<xsd:element name="FactBase" type="FactBaseType" />
    <xsd:complexType name="FactBaseType">
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="FactSet" type="FactSetType" maxOccurs="unbounded" />
            <xsd:element name="fact" type="factType" maxOccurs="unbounded" />
            <xsd:element name="simpleFact" maxOccurs="unbounded" />
            <xsd:element name="complexFact" type="complexFactType" maxOccurs="unbounded" />
        </xsd:choice>
    </xsd:complexType>

<xsd:element name="FactSet" type="FactSetType" />
    <xsd:complexType name="FactSetType" >
<!--        <xsd:group ref="differentFacts" minOccurs="1" maxOccurs="unbounded" /> -->
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="simpleFact" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="fact" type="factType" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="complexFact" type="complexFactType" minOccurs="0" maxOccurs="unbounded" />
        </xsd:choice>
        <xsd:attribute name="name"   type="xsd:string"  use="required" />
        <xsd:attribute name="enable" type="xsd:boolean" use="optional" />
    </xsd:complexType>


<!-- ************************************************ -->
<!-- *   scheme for all different kinds of facts      * -->


<xsd:group name="differentFacts">
    <xsd:choice>
        <xsd:sequence>
            <xsd:element name="simpleFact" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="fact" type="factType" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="complexFact" type="complexFactType" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:choice>
</xsd:group>

<xsd:element name="simpleFact" type="xsd:string" />

<xsd:element name="fact" type="factType" />
    <xsd:simpleType name="factType">
        <xsd:restriction base="xsd:string"></xsd:restriction>
    </xsd:simpleType>

<xsd:element name="complexFact" type="complexFactType" />
    <xsd:complexType name="complexFactType" mixed="true">
        <xsd:sequence>
<!--            <xsd:element type="xsd:string" /> -->
            <xsd:choice>
                <xsd:element name="eq" />
                <xsd:element name="neq" />
                <xsd:element name="lt" />
                <xsd:element name="le" />
                <xsd:element name="gt" />
                <xsd:element name="ge" />
            </xsd:choice>
<!--            <xsd:element type="xsd:string" /> -->
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Konzept zur Berechnung der Gridness eines Benutzerdialoges.

Quelle: Dissertation Hamacher. RWTH Aachen.

Gridedness (*G*)

Erklärung: Ist ein Maß der *Angeordnetheit* der Displayelemente und wird hauptsächlich zur Konsistenzberechnung benutzt. Es gibt an, in welchem Maße die Displayelemente aneinander ausgerichtet sind. Für alle Displayelemente werden jeweils die linken, rechten, oberen und unteren Kanten aufgenommen; Kanten mit gleichen Werten werden nur einmal gezählt. Die Summe dieser Werte entspricht der Anzahl der vorkommenden Kanten. Die *Gridedness G* des Dialogs *Dlg* einer Dialogfolge berechnet sich anschließend wie folgt:

$$G(S) = \frac{|Dlg|}{|L|+|R|+|T|+|B|} \cdot 100 \qquad (3.5)$$

mit
$$L = \{l | l = d^{[x]}, \forall d \in D\}$$
$$R = \{r | r = d^{[x]} + d^{[w]}, \forall d \in D\}$$
$$T = \{t | t = d^{[y]}, \forall d \in D\}$$
$$B = \{b | b = d^{[y]} + d^{[h]}, \forall d \in D\}$$

Abbildung 3.7 verdeutlicht die Berechnung. Im Dialog D_1 (links) ergeben 4 Displayelemente mit insgesamt 10 Kanten (6 vertikal und 4 horizontal) eine *Gridedness* von $G(D_1) = 40$. Im Dialog D_2 (rechts) sind die Elemente gegeneinander verschoben, was sich in einer deutlich niedrigeren *Gridedness* von $G(D_2) = 28,5$ widerspiegelt.
Elemente, die überdeckt sind, verfälschen die *Gridedness* und sollten daher vor der Berechnung entfernt werden.

Vorgabe: Je größer die Werte im Vergleich, desto mehr sind die Elemente aneinander ausgerichtet. Aufgrund fehlender objektiver Vorgaben eignet sich dieses Maß nur für eine vergleichende Bewertung.

Ergebnis: Zahlwert

Quelle: (MAHAJAN und SHNEIDERMAN 1997)

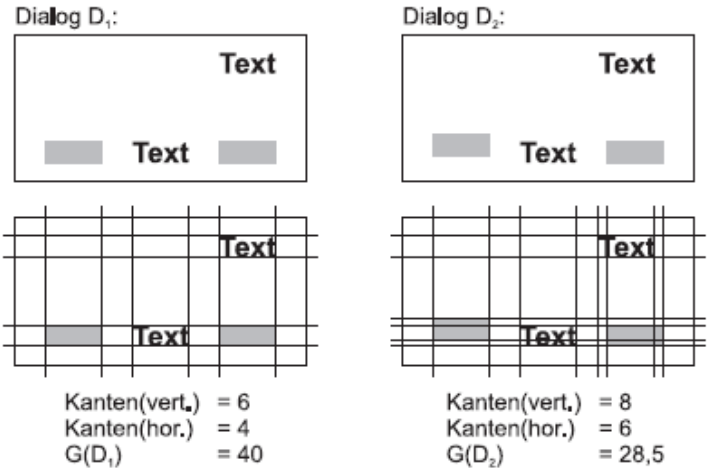

Abbildung 3.7: Beispiel der Berechnung der *Gridedness (Angeordnetheit)* von Displayelementen unter Einbeziehung aller im Bild vorkommenden Kanten. Ein höherer Wert entspricht einem konsistenteren Layout.

MAHAJAN, R. und B. SHNEIDERMAN (1997). *Visual and Textual Consistency Checking Tools for Graphical User Interfaces*. IEEE Transactions on Software Engineering, 23(11):722–735.