Prüfer: Prof. Dr. Egbert Lehmann

Betreuer: Dr. Christian Rathke

begonnen am: 1. Oktober 1989

beendet am: 31. März 1990

CR-Klassifikation I.2.4, I.2.5

Entwicklung und Implementation eines Regelsystems für FrameTalk

Frank Merath

Studienarbeit Nr. 837

Fakultät Informatik
Universität Stuttgart
Azenbergstraße 12
D 7000 Stuttgart 1

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

Entwicklung und Implementation eines Regelsystems für FrameTalk

Frank Merath

März 1990

Diese Arbeit wurde in der Abteilung Intelligente Systeme am Institut für Informatik, Universität Stuttgart durchgeführt.

An dieser Stelle möchte ich mich für die Unterstützung und Betreuung durch Herrn Dr. Christian Rathke bedanken.

Inhalt

Abbildungsverzeichnis

Einleitung

Das funktional-applikative Paradigma hat eine lange Tradition in der Künstlichen Intelligenz. Die darauf basierende Programmiersprache LISP wurde auf Grund ihrer besonderen Eignung zur Verarbeitung symbolischer Daten zur Grundlage vieler Forschungen und Anwendungen auf dem KI-Sektor. Allerdings fehlen in LISP einige Ausdrucks- und Organisationsmöglichkeiten, über die objektorientierte Programmiersprachen verfügen. Die Prinzipien des Nachrichtenversands, der Datenkapselung und der Vererbungshierarchien sind hier nicht verwirklicht worden.

Daher wurden in einigen LISP-Dialekten objektorientierte Konzepte integriert (LOOPS, OBJTALK, CLOS), um einerseits die neuen Möglichkeiten dieses Paradigmas ausschöpfen zu können, andererseits aber auf die Vorteile der LISP-Welt (z.B. bestehende Hard- und Software) nicht verzichten zu müssen. Während sich jedoch einige Modelle der Wissensrepräsentation, z.B. Frames, in objektorientierten Umgebungen gut implementieren lassen, fehlt hier eine einfache Möglichkeit, Wissen über kausale oder empirische Zusammenhänge zwischen Teilen der so dargestellten Informationen darzustellen. Solche Zusammenhänge lassen sich dagegen einfach durch Produktionsregeln formulieren, wie sie in regelbasierten Systemen Anwendung finden. Dort ist es wiederum häufig sehr schwierig, Regeln nach ihrer Verwendung in Mengen zusammenzufassen oder auf bestimmte Gruppen von Objekten einzuschränken.

Eine Integration der Produktionsregelschematik in ein objektorientiertes System kann daher einige Nachteile der Einzelsysteme vermeiden und wesentliche Verbesserungen für die Ausdrucksfähigkeit und Verständlichkeit des Systems bewirken.

Diese Studienarbeit beschreibt den Entwurf und die Implementierung einer Regelsystemkomponente für das objektorientierte System FRAMETALK.

Teil 1

Grundlagen

1.1 Objektorientierte Programmiersysteme (OOPS)

Wie einige andere Programmierstile auch hat sich die objektorientierte Form aus der Idee entwickelt, das Programmieren der menschlichen Betrachtungsweise stärker anzunähern, um auf diese Weise Strukturen und Vorgänge in der wirklichen Welt einfacher modellieren zu können. Beispiele für diese Entwicklung sind u.a. SIMULA und SMALLTALK-80. Einige der Prinzipien, denen beim Design solcher Programmiersprachen und -systeme besondere Beachtung gewidmet wurde, sind:

- Einheitliche Grundstruktur
- Modularität
- Datenabstraktion
- Klassifizierbarkeit
- Faktorisierung

Die Umsetzung dieser Designprinzipien führte zu einem System mit folgendem Aufbau:

Grundbausteine des Systems sind *Objekte*. Dies sind Einheiten, die über lokale Daten und Funktionen verfügen. Die Daten werden in lokalen Variablen, auch *Slots* genannt, gehalten und sind meist nur vom Objekt selbst, nicht aber von außen, zugänglich. Der funktionale Teil des Objekts besteht aus sogenannten *Methoden*, in denen Operationen auf den lokalen Daten und/oder Kommunikation mit anderen Objekten festgelegt werden.

Diese Kommunikation zwischen Objekten beruht auf dem Versenden von *Nachrichten*. Darin werden über Selektoren die vom Empfänger der Nachricht durchzuführenden Aktionen spezifiziert, auch können Argumente übergeben werden. Vom Empfänger wird dann die entsprechende Methode ausgeführt und evtl. ein Ergebnis zurückgeliefert. Von einem traditionellen Funktions- oder Prozeduraufruf unterscheidet sich diese Art des Programmflusses dadurch, daß dem Nachrichtenversender nur das Protokoll bekannt ist, nicht aber die tatsächliche Struktur der Verarbeitung. Unterschiedliche Objekte können auf die gleiche Nachricht verschiedene Methoden zur Anwendung bringen. Änderungen in der Definition eines Objekts oder in der Art der verwendeten Objekte können somit ohne Auswirkungen auf andere Objekte durchgeführt werden.

Gemeinsamkeiten von Objekten werden durch die Zusammenfassung zu *Klassen* ausgedrückt. Diese sind Objekte, die „Baupläne“ für andere Objekte enthalten; nämlich die Methoden, die ein Objekt ausführen kann sowie den Aufbau der lokalen Daten. Jedes Objekt ist *Instanz* genau einer Klasse, d.h. seine Eigenschaften werden durch deren „Baupläne“ beschrieben.

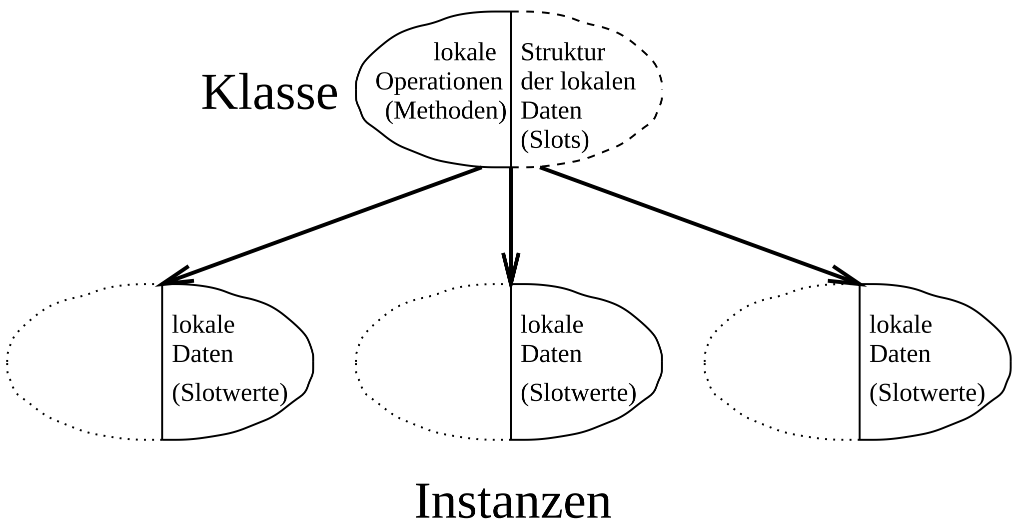

Abbildung 1.1: Struktur von Klassen und Objekten

Um dem Prinzip der Faktorisierung Genüge zu tun, werden Klassen hierarchisch angeordnet. Solch eine Vererbungshierarchie kann einfach (baumartig) oder mehrfach (graphartig) ausgeprägt sein. Eine Klasse ererbt von ihren Vorgängern in der Hierarchie, ihren Superklassen, alle dort definierten Methoden sowie die Datenstruktur für ihre Instanzen. Zusätzlich können weitere Eigenschaften (Methoden und Slots) hinzugefügt bzw. ererbte spezialisiert werden. Das Vererbungsprinzip erlaubt es, einfach neue speziellere Klassen von Objekten aus bestehenden zu kreieren, indem Veränderungen nur an den notwendigen Teilen vorgenommen werden, und vermindert somit die Redundanz an Funktions- und Datenstrukturvereinbarungen.

In einigen OOPS existieren bereits Programmierwerkzeuge wie Browser, Inspektoren und Debugger, die auf den objektorientierten und hierarchischen Aufbau dieser Systeme abgestimmt sind und eine textuelle oder graphische Darstellung der verschiedenen Strukturen und Abhängigkeiten zur Verfügung stellen bzw. eine direkte Manipulation derselben erlauben. Die Verfügbarkeit solcher Benutzeroberflächen, die übrigens meist auch in der objektorientierten Sprache geschrieben wurden und damit gut anpaßbar und erweiterbar sind, stellt einen weiteren Vorteil dieser OOPS dar.

Für eine genauere Beschreibung objektorientierter Programmiersysteme sei auf [GoldbergRobson 83] und [StefikBobrow 86] verwiesen.

1.2 Regelbasierte Programmiersysteme

Das Grundkonzept eines regelbasierten Systems besteht darin, aus bestehendem Wissen über Fakten mittels formalisierter Beziehungen zwischen hypothetischen Fakten neues Faktenwissen herzuleiten.

Die momentan bestehenden Fakten bilden dabei die *Datenbasis*. Deren Organisationsform kann, je nach den Anforderungen, unterschiedliche Ausprägung erfahren (Listen, Felder, semantische Netze); die Struktur der einzelnen Fakten ist jedoch grundsätzlich gleichartig. Die Existenz eines Fakts – in einigen Fällen statt dessen seine Wahrscheinlichkeit – geht aus der Datenbasis eindeutig hervor.

In der *Regelbasis* befinden sich die Informationen über Beziehungen zwischen Fakten, die Regeln. Diese bestehen aus einem Bedingungs- und einem Folgerungsteil. Die Bedingung stellt Annahmen über Fakten in der Datenbasis dar, während die Folgerung Anweisungen enthält, neue Fakten in die Datenbasis einzutragen. Ist die Bedingung erfüllt, d.h. die dort beschriebenen Annahmen treffen zu, so können die Anweisungen der Folgerung ausgeführt werden.

Bei diesen Regeln sind zwei Anwendungsrichtungen möglich: Es kann versucht werden, ausgehend von den bisher bekannten Fakten ungerichtet alle durch die Regeln bestimmbaren Fakten abzuleiten. Jeder in der Folge einer Regelanwendung neu in die Datenbasis eingetragene Fakt kann wiederum als Ausgangspunkt für weitere Regelanwendungen dienen. Diese Vorgehensweise wird *Vorwärtsverkettung* (Forward Chaining) genannt. Die andere Möglichkeit ist, einen angefragten aber bisher in der Datenbasis nicht vorhandenen Fakt durch Anwendung von Regeln gezielt herzuleiten. Auch dabei ist evtl. mehrfache Regelanwendung notwendig, wenn in der Bedingung einer Regel, die den gewünschten Fakt herleiten könnte, weitere bisher unbekannte Fakten auftauchen. Die Bezeichnung für diese Art der Regelanwendung ist *Rückwärtsverkettung* (Backward Chaining).

In beiden Richtungen der Verkettung von Regeln kommt es vor, daß mehrere Regeln gleichzeitig anwendbar sind (Forward Chaining) bzw. zur Herleitung des gesuchten Fakts dienen können (Backward Chaining). Die Lösung dieses Konflikts obliegt dem dritten Teil des regelbasierten Systems, der *Inferenzmaschine*. Diese stellt zunächst die Regeln fest, die zur Konfliktmenge gehören, und entscheidet dann anhand einer Konfliktlösestrategie, welche Regel angewandt bzw. weiterverfolgt werden soll. Die Festlegung dieser Strategie hat einen starken Einfluß auf die Funktionsweise (Durchschaubarkeit und Effizienz) des gesamten Systems.

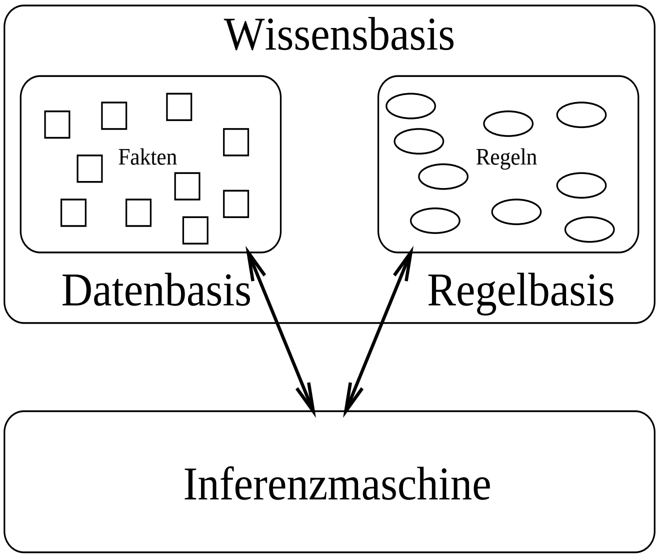

Abbildung 1.2: Aufbau eines regelbasierten Systems

Meistens wird in einem System entweder nur die vorwärts– bzw. die rückwärtsgerichtete Verkettungsart verwendet, oder die Anwendungen beider Verkettungsrichtungen werden deutlich voneinander abgegrenzt. Mischformen, bei denen beide Richtungen gleichzeitig angewandt werden, setzen einen hohen Aufwand zur Synchronisation zwischen den Teilen voraus, um z.B. erkennen zu können, wann das Forward Chaining einen vom Backward Chaining benötigten Fakt abgeleitet hat.

1.3 Integration eines Regelsystems in ein OOPS

Für die Einbettung von regelbasierten Konzepten in eine objektorientierte Umgebung spricht, daß viele Problemstellungen mit einem solchen multiparadigmatischen System natürlicher und effizienter gelöst werden können. Dies trifft vor allem zu, wenn die Ideen beider Stile etwa gleich stark in Anspruch genommen werden; wenn also z.B. die Modellierung des Problems durch hierarchisch geordnete Objekte durchgeführt werden kann und gleichzeitig eine Darstellung des Wissens über Verknüpfungen zwischen den Daten dieser Objekte mittels Regeln möglich ist.

Des weiteren sind bei traditionellen regelbasierten Systemen die Regel- und die Datenbasis meist sehr umfangreich gegenüber derjenigen Menge von Regeln, die zu einem Zeitpunkt in Betrachtung zu ziehen sind und den von diesen Regeln betroffenen Fakten. In einem objektorientierten System besteht die Möglichkeit, Regeln und Fakten objektlokal anzuordnen, wobei die lokalen Variablen (Slots) des Objekts die Datenbasis darstellen, während die Regeln in Analogie zu den Methoden bei der Klasse definiert sind. Durch eine solche Modularisierung kann das Verhältnis zwischen untersuchten und ausgeführten Regel (test–fire–ratio) auch ohne komplizierte Strukturen wie Rete–Netzwerke o.ä. verbessert werden.

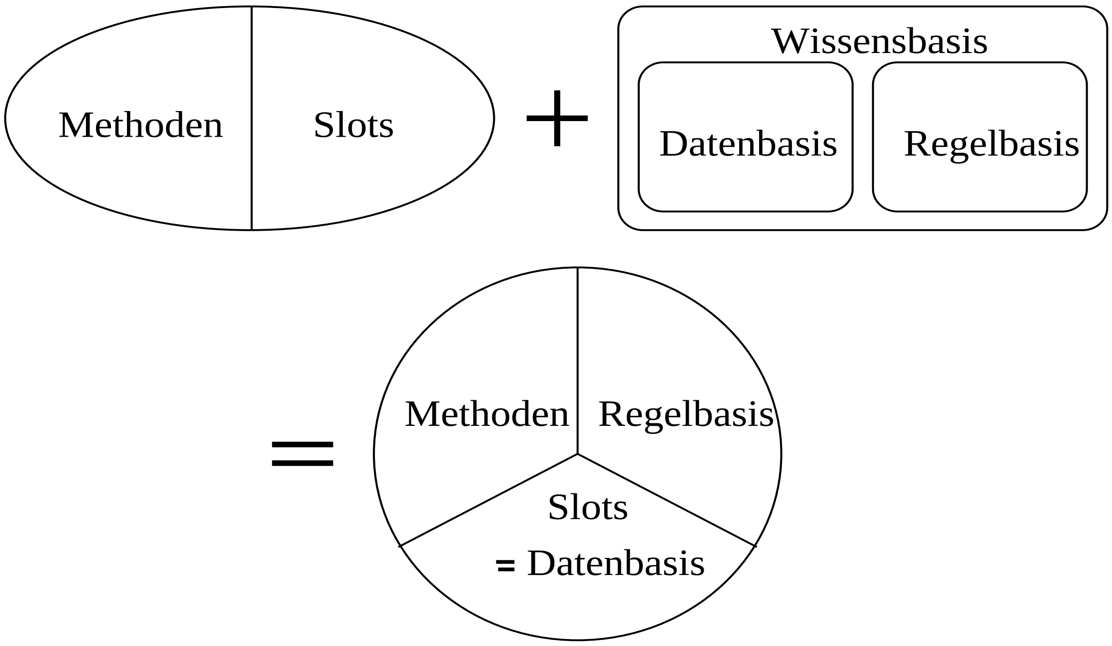

Abbildung 1.3: Integration einer Regelkomponente in ein Objekt

Demgegenüber steht als Nachteil einer solchen Integration der Verlust der Uniformität im Vergleich zu den einzelnen Programmierkonzepten. Inwiefern dieser Nachteil als gravierend zu bewerten ist, hängt stark von der Realisierung des Systems ab. Die Regelkomponente darf keinen Fremdkörper im objektorientierten System darstellen, sondern muß eng an dieses angebunden werden und selbst objektorientiert gestaltet sein. Dies kann auch Vorteile für die Regelkomponente mit sich bringen; so können z.B. Regeln als Objekte mit objektorientierten Werkzeugen manipuliert werden, sowie ähnlich wie Methoden innerhalb der Klassenhierarchie verfeinert und ergänzt werden.

Teil 2

Entwurf

2.1 Einführung – Clos und FrameTalk

Neben den Programmiersprachen, die wie SmallTalk-80 von vorneherein nur auf dem objektorientierten Paradigma aufbauen, gibt es auch Systeme, bei denen objektorientierte Konzepte in bestehende prozedurale (C^{++}) oder funktional-applikative (Loops, ObjTalk) Sprachen integriert wurden. Aufbauend auf dem Ansi Standard Common Lisp [Steele Jr. 84] wird momentan vom X3J13 Kommitee eine Spezifikation des Common Lisp Object System Clos erarbeitet ([Bobrow et al. 88]). Besonders hervorzuhebende Aspekte diese Systems sind seine Vererbungsstruktur und sein Methodenkonzept.

Clos unterstützt *multiple Vererbung*, d.h. eine Klasse kann mehrere direkte Superklassen besitzen. Um Konflikte zu vermeiden, wenn von mehreren Klassen unterschiedliche Ausprägungen eines Merkmals (z.B. einer Methode oder eines Slot–Defaultwerts) ererbt werden könnten, wird auf den Superklassen einer Klasse eine Totalordnung (class precedence list) definiert, die die Vorränge festlegt.

Operationen auf Objekten werden von Clos im Gegensatz zu anderen objektorientierten Systemen nicht durch Nachrichtenversendung sondern mittels *generischer Funktionen* realisiert. Diese unterscheiden sich im Aufruf nicht von normalen Lisp–Funktionen; anders als jene definieren sie aber nur das Aufrufprotokoll, nicht jedoch die durchzuführenden Aktionen. Bei der Methodendefinition können nun die Parameter der generischen Funktion spezialisiert werden, also angegeben werden, welcher Klasse diese angehören müssen, damit die jeweilige Methode ausgewählt wird. Durch dieses Konzept ist es möglich, die Methodenauswahl in Abhängigkeit von mehr als nur einem Parameter durchzuführen. Außerdem bietet Clos noch die Möglichkeit, zusätzlich zur primären Methode sekundäre vor, nach oder statt dieser auszuführen.

Auf der Basis des – teilweise noch im Entstehen begriffenen – CLOS–Standards wird zur Zeit von Dr. Christian Rathke am Institut für Informatik der Universität Stuttgart das System FRAMETALK entwickelt. Dieses soll einige zusätzliche Konzepte, die im Rahmen der Entwicklung von OBJTALK [Rathke 86] entstanden sind, in CLOS integrieren.

Speziell handelt es sich dabei um die Ergänzungen

- Slotbeschreibungen,
- Perspektiven,
- Regeln und
- Constraints.

Der Entwurf einer Regelkomponente für FRAMETALK und ihre Implementierung in PCL, eines Subsets von CLOS, das unter anderem auf SUN-Workstations zur Verfügung steht, wird im folgenden beschrieben.

2.2 Die Regelkomponente

2.2.1 Struktur der Wissensbasis

Wie bereits erwähnt, sollen in FrameTalk Daten– und Regelbasis der Regelkomponente nicht nur einmal, global, existieren. Statt dessen kann jedem Objekt eine solche Wissensbasis zugeordnet werden. Dies entspricht den objektorientierten Prinzipien Modularität und Klassifizierbarkeit.

2.2.1.1 Die Datenbasis

Die Rolle der Datenbasis wird dabei von den lokalen Variablen eines Objekts, den Slots, übernommen. In der Datenbasis enthaltene Fakten sind dabei wahre Aussagen über Slotinhalte; das Eintragen von neuen Fakten findet durch Belegen von Slots mit Werten statt. Da Slots eines Objekts wiederum Objekte enthalten können usf., zählen auch Beschreibungen von über Slotpfade erreichbaren Daten zu den Fakten. Ebenso ist es möglich, Konstanten und Werte von globalen Variablen als Bestandteil einer Faktbeschreibung zu verwenden. Die formale Definition von Faktbeschreibungen und Fakteinträgen lautet damit

$$\textit{fact-descr} ::= \textit{constant} \mid \textit{global-var} \mid \textit{slotpath}^1 \mid (\textit{fn}\ \textit{fact}_1,\cdots,\textit{fact}_n)$$
$$\textit{fact-entry} ::= (\textit{slotpath}\ \textit{fact-descr})$$

wobei *fn* eine beliebige seiteneffektfreie n-stellige Lisp-Funktion sein darf. Zur tatsächlichen Definition von Slotpfaden siehe Abschnitt 3.2.2.

2.2.1.2 Die Regelbasis

Auch die Regelbasis soll objektlokal gestaltet sein. Jedoch wird hier nicht bei jedem Objekt eine Regelbasis angelegt, sondern deren Definition bei der jeweiligen Klasse vorgenommen. Dies entspricht dem Vorgehen bei der Vereinbarung von Methoden und geschieht auch aus denselben Gründen: Sollten für zwei Objekte unterschiedliche Regeln gelten, so gehören sie ebensowenig in die gleiche Klasse wie wenn sie auf dieselbe Nachricht verschiedene Methoden ausführen sollen.

[1] Im Falle *fact-descr* = *slotpath* gilt die Lisp-Konvention, daß *nil* den Wahrheitswert falsch (false) darstellt; Werte, die nicht *nil* sind, dagegen wahr (true).

Die wichtigsten Bestandteile der Regeln sind die *Bedingung* und die *Folgerung*. Beide setzen sich aus einer oder mehreren Klauseln zusammen, die konjunktiv verknüpft werden. Die Klauseln der Bedingung sind Faktbeschreibungen, die der Folgerung Fakteintragungen. Die Semantik einer Regel ist also:

Wenn alle Aussagen über Slotinhalte, die durch die Faktbeschreibungen der Bedingung gemacht werden, wahr sind,
dann führe alle Slotbelegungen durch, die in der Folgerung stehen.

In manchen Fällen kann es wünschenswert sein, zusätzlich noch Seiteneffekte zu erzeugen, z.B. Ausgeben einer Meldung oder Durchführen von anderen Verarbeitungen. Zu diesem Zweck besitzen die Regeln einen Aktionsteil, in dem beliebige Funktionsaufrufe stehen können. Beim Anwenden einer Regel werden diese Aufrufe dann nach den Fakteinträgen in die Datenbasis evaluiert.

2.2.2 Die Inferenzmaschine

Die bisher beschriebenen Teile des Regelsystems sind dessen deskriptive Komponente. Sowohl die Datenbasis als auch die Regelbasis beschreiben nur, was für Fakten und Faktzusammenhänge es gibt, nicht aber, wie diese einzusetzen sind. Diese Steuerung zu übernehmen ist Aufgabe der Inferenzmaschine. Bei ihrem Entwurf ist zu berücksichtigen, welche Auswahlmöglichkeiten dem Benutzer für den Einsatz und den Ablauf des Inferenzprozesses gegeben werden sollen und welche Steuerungskriterien bei der Konfliktlösung eingesetzt werden können.

2.2.2.1 Verkettungsrichtung

Von grundlegender Bedeutung für die Struktur der Inferenzmaschine ist die Verkettungsrichtung. *Vorwärtsverkettung* (Forward Chaining) ist die Methode des ungerichteten Ableitens aus bestehenden Fakten. Der Eintrag neuer Fakten in die Datenbasis löst einen Inferenzprozess aus, der in folgendermaßen abläuft:

1. Schritt: Feststellen aller Regeln, die triggern (d.h. deren Bedingungen erfüllt sind)
Falls dies für keine Regel gilt, beenden der Inferenz
2. Schritt: Auswählen einer Regel aus der in Schritt 1 erstellten Menge
3. Schritt: Abfeuern der ausgewählten Regel
(d.h. Eintragen der Fakten aus der Folgerung in die Datenbasis)
danach wieder mit Schritt 1 beginnen

Dieser Arbeitszyklus wird auch als Recognize–Act–Cycle bezeichnet. Die Vorwärtsverkettung zeichnet sich dadurch aus, daß nach Beendigung des Inferenzprozesses alle ableitbaren Fakten in die Datenbasis eingetragen worden sind, da die im 3. Schritt eines Zyklusdurchgangs neu eingetragenen Fakten im 1. Schritt des nächsten Durchgangs die Bedingungen weiterer Regeln erfüllen können (s. Abb. 2.1). Der Inferenzprozess selbst ist relativ einfach, er erfordert jedoch einen hohen Rechenaufwand, um in jedem Zyklusdurchlauf aufs Neue alle triggernden Regeln zu ermitteln. Eine Verringerung dieses Aufwands ist möglich, indem eine Verbindung zwischen Fakten und davon betroffenen Regeln hergestellt wird, so daß Regeln, deren Bedingungen bisher noch nicht erfüllt waren und die von neu eingetragenen Fakten nicht berührt werden, gar nicht erst überprüft werden müssen (ähnlich dem Rete-Network in OPS-5).

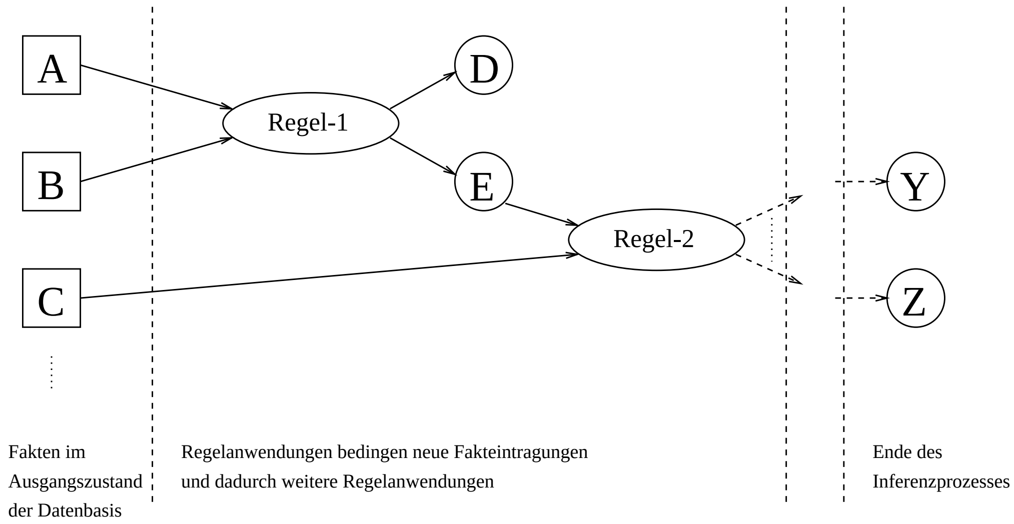

Abbildung 2.1: Inferenzvorgang bei der Vorwärtsverkettung

Vorwärtsverkettung eignet sich besonders für Konfigurationsprobleme, also für Fälle, bei denen abhängig von den Ausgangsbedingungen teilweise völlig verschiedene Endzustände erreicht werden können. Durch die Aktionen, die zusätzlich zur Folgerung durchgeführt werden können, ist es auch möglich, vorwärtsverkettende Regelsysteme z.B. zur Prozeßsteuerung zu verwenden.

Die andere Möglichkeit der Regelverkettung wird *Rückwärtsverkettung* (Backward Chaining) genannt. Wird nach einem Fakt gefragt, der nicht in der Datenbasis steht, so kann versucht werden, diesen aus dem bestehenden Wissen herzuleiten, indem ein Ableitungsbaum aufgebaut wird, dessen Wurzel der gesuchte Fakt ist. Dazu werden zunächst alle Regeln gesucht, deren Folgerung diesen Fakt eintragen könnten (hier: den betreffenden Slot belegen könnten). Ist die Bedingung einer dieser Regeln erfüllt, so wird der entsprechende Fakteintrag durchgeführt und der Inferenzprozess beendet. Wenn in Bedingungen wiederum unbekannte Fakten enthalten sind und der gesuchte Fakt anderweitig nicht herleitbar ist, dann wird rekursiv ein neuer Inferenzprozess gestartet, um diese Fakten herzuleiten. Die Herleitung für einen Fakt scheitert, falls keine Regeln ihn herleiten können bzw. die Bedingungen keiner dieser Regeln erfüllt werden können (s. Abb. 2.2).

Der Rechenaufwand für eine derartige Faktherleitung ist theoretisch ebensogroß wie der beim Forward Chaining nach einem Fakteintrag nötige. Der Vorteil besteht jedoch darin, daß der Inferenzprozeß gezielt nur dann angestoßen wird, wenn der Benutzer Interesse an unbekannten Fakten bekundet, während bei der Vorwärtsverkettung evtl. ständig uninteressante Fakten abgeleitet werden. In Kauf genommen werden muß dabei allerdings die größere Komplexität des Inferenzvorgangs durch den Aufbau des Ableitungsbaums, das nötige Backtracking beim Scheitern einer Teilbedingung und das Vermeiden von Zyklen im Ableitungsbaum.

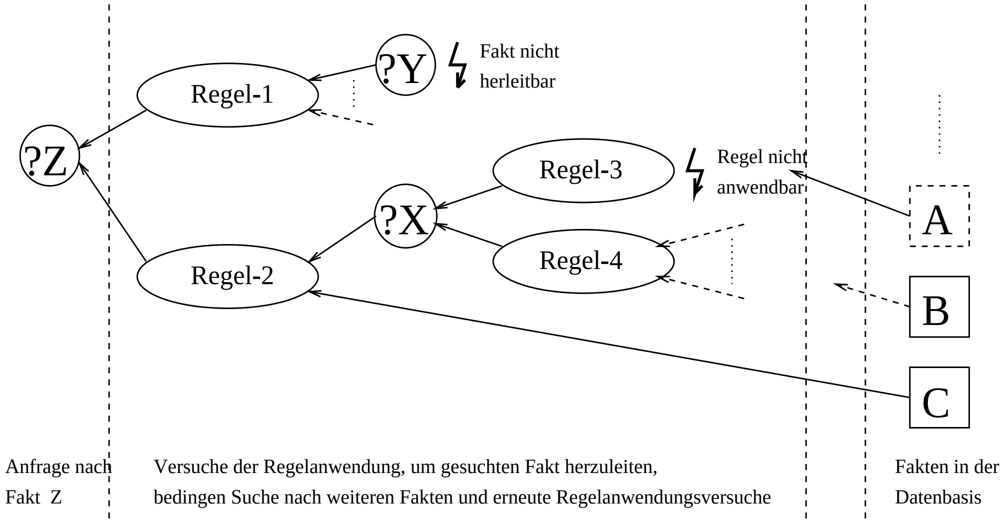

Abbildung 2.2: Inferenzvorgang bei der Rückwärtsverkettung

Daher werden rückwärtsverkettende Systeme hauptsächlich dann verwendet, wenn es darum geht, aus einer großen Menge von Daten mit jeweils vielen anwendbaren Regeln gezielt gewisse Informationen herzuleiten; etwa bei Auskunftssystemen.

Da die Wahl der zu verwendenden Verkettungsrichtung wohl hauptsächlich von der jeweiligen Problemstellung abhängt, erscheint die ausschließliche Verwendung einer Richtung nicht sinnvoll, zumal die Struktur der Regeln beide Möglichkeiten zuläßt. Den Vorrang bei der Implementierung soll jedoch die Vorwärtsverkettung haben, da hierbei dank des einfacheren Aufbaus der Inferenzmaschine die Designprinzipien der Integration in ein objektorientiertes System klarer darstellbar sind.

2.2.2.2 Automatische vs manuelle Inferenzauslösung

Nach der Betrachtung der Verkettungsrichtung stellt sich die Frage, wann, d.h. durch welchen Anlaß, der Inferenzprozeß gestartet werden soll. Dies kann z.B. geschehen, indem der Benutzer explizit den Befehl dazu gibt. Bei dieser Art der Steuerung, der manuellen Inferenzauslösung, verbleibt die Kontrolle über den Zeitpunkt, an dem die Inferenzmaschine startet, vollständig beim Benutzer. Er kann dadurch verhindern, daß die Regelanwendung vorgenommen wird, wenn es ihm aus Zeit- oder Datengründen nicht geraten erscheint; muß diese Kontrollmöglichkeit aber mit zusätzlichem Verwaltungsaufwand bezahlen. Beim Forward Chaining entspricht das manuelle Starten der Anweisung an die Inferenzmaschine, alle aus dem derzeitigen Zustand der Datenbasis ableitbaren Fakten zu bestimmen. Im Fall der Rückwärtsverkettung ist es der Befehl, einen gewünschten Fakt herzuleiten.

Im Gegensatz dazu wird bei der automatischen Inferenzauslösung die Regelanwendung unabhängig von expliziten Anweisungen dann versucht, wenn der Zustand der Datenbasis dies erfordert. Erfordernisse in diesem Sinn sind beim Backward Chaining Anfragen an die Datenbasis nach einem unbekannten Fakt bzw. bei der Vorwärtsverkettung Neueinträge von Fakten, die die Datenbasis potentiell unvollständig machen.

Betrachtet man das Arbeiten der Regelkomponente in einem OOPS, so ist zu erkennen, daß bei der Rückwärtsverkettung der Unterschied zwischen automatischer und manueller Inferenzauslösung sehr gering ist. Er besteht eigentlich nur in der formalen Struktur der Datenanforderung, d.h. der Spezifikation, ob bei nicht explizit vorhandenen Fakten, also unbelegten Slots, ein Inferenzversuch vorgenommen werden soll. Wesentlich stärker wirkt sich die Art der Inferenzauslösung aus, wenn Vorwärtsverkettung betrieben wird. Da die Regelanwendung hierbei nicht zielgerichtet vorgenommen wird, hängt die Vollständigkeit der Datenbasis von der Häufigkeit ab, in der die Inferenzmaschine aufgerufen wird. Andererseits kann in diesem Fall die automatische Inferenzauslösung zu Konsistenzproblemen führen, wenn sich ein geänderter Fakt aus den Werten mehrerer Slots zusammensetzt. Wird die Inferenz ausgelöst, wenn einige der betroffenen Slots noch die alten, andere dagegen schon die geänderten Werte haben, so können inkonsistente Werte in die Datenbasis eingetragen werden und vor allem durch die Aktionsteile der Regeln ungewollte Handlungen ausgelöst werden.

Für die objektorientierte Integration der Regelkomponente bietet sich folgende Synthese der Inferenzauslösungsmöglichkeiten an: Grundsätzlich arbeitet die Inferenzmaschine automatisch, also datengesteuert. Es ist jedoch den Objekten möglich, der Inferenzmaschine den Zugriff auf ihre Wissensbasis zu sperren. Leseoperationen auf unbelegte Slots und Änderungen von Slotwerten lösen dann keinen Inferenzprozeß mehr aus. Erst mit der Freigabe der Wissensbasis durch das Objekt sind diese Aktionen wieder möglich. Durch diese Konstruktion wird sowohl eine rein manuelle Inferenzauslösung möglich, indem die Wissensbasis jeweils nur kurz entsperrt wird, als auch die Erzeugung von Konstruktionen zur atomaren Änderung von aus mehreren Slotwerten bestehenden Fakten ermöglicht.

2.2.2.3 Regelselektion

Die Inferenzmaschine ist – wie in Abschnitt 2.2.2.1 beschrieben – dazu fähig, festzustellen, welche Regeln den momentan gestellten Ansprüchen genügen. Geht es darum, mittels Rückwärtsverkettung einen Fakt herzuleiten, so sind dies diejenigen Regeln, in deren Folgerung der dem Fakt entsprechende Slot belegt wird. Davon ausgehend wird dann ein *Ableitungsbaum* aufgebaut, und zwar mit dem herzuleitenden Fakt als Wurzel. Die nächste Stufe im Baum stellen dann die Regeln dar, die den Fakt herleiten könnten, an sie anschließend kommen Knoten, die wieder von herzuleitenden Fakten gebildet werden usw. Die Kanten zwischen herzuleitendem Fakt und Regeln können als disjunktiv verknüpft angesehen werden, da eine Regel zur Herleitung genügt. Die Kanten von einer Regel zu ihren Fakten dagegen sind konjunktiv verknüpft, weil alle Fakten bekannt sein müssen, damit die Regelbedingung überprüft werden kann. Das Aussehen eines solchen Baumes zeigt Abb. 2.3.

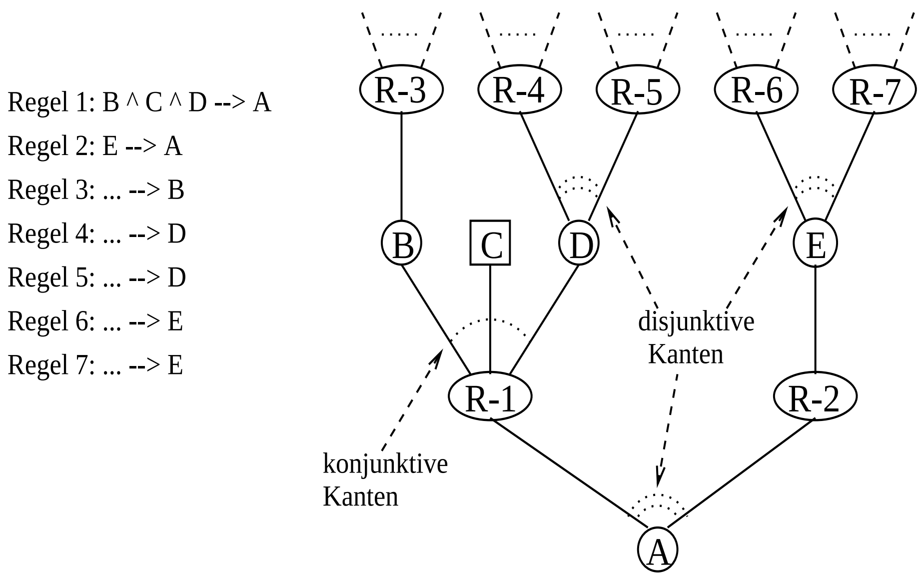

Abbildung 2.3: Beispiel für den Aufbau eines Ableitungsbaumes

Der Aufbau und das Durchlaufen eines solchen Ableitungsbaums ist nun nach verschiedenen Strategien möglich. Beispiele für derartige Strategien sind Tiefensuche (depth first), Breitensuche (breadth first) und bewertete Suche (best first). Tiefensuche bedeutet, daß immer zuerst die als letzte hinzugefügte zu einer Regel führende Kante des Baums weiterverfolgt wird, d.h. versucht wird, ihre Bedingung zu erfüllen. Bei der Breitensuche dagegen werden zuerst alle Regelknoten einer Ebene untersucht, ehe die der nächsten Ebene bearbeitet werden. In Abb. 2.4 werden diese beiden Verfahren illustriert. Das Best-First-Verfahren dagegen besteht darin, aus einer Menge von Knoten nach einer Bewertungsfunktion den weiterzuverfolgenden auszuwählen. Solch eine Bewertung kann z.B. sein, daß in der Folge des Regelknotens möglichst wenige Faktknoten zu bestimmen sind. Wie leicht einzusehen ist, können auch Breiten– und Tiefensuche als Spezialfälle der bewerteten Suches mit einer besonderen Bewertungsfunktion angesehen werden.

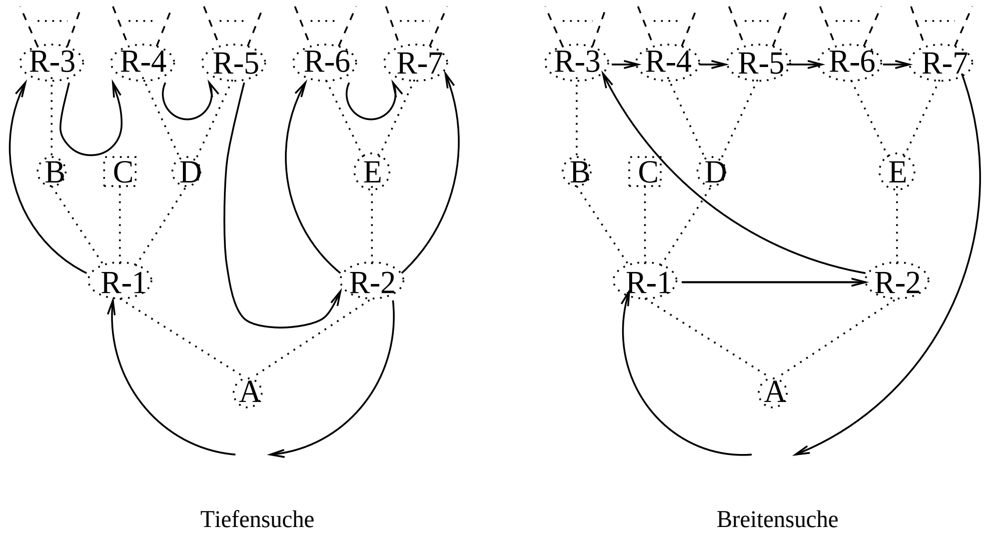

Abbildung 2.4: Durchlaufen eines Ableitungsbaumes

Handelt es sich bei der Regelselektion des Backward Chaining hauptsächlich um die Reihenfolge, in der die Regelknoten des Ableitungsbaums durchlaufen werden, während das im Endeffekt erreichte Ziel (der Wert, mit dem der Slot belegt wird) i.a. dasselbe ist und nur mehr oder weniger schnell erreicht wird[2], so treten bei der Vorwärtsverkettung andere Gesichtspunkte in den Vordergrund. Hier triggern auf Grund der Änderung eines Fakts meist mehrere Regeln und es ist die Aufgabe des *Regelselektors*, aus der *Konfliktmenge* dieser Regeln eine herauszusuchen, die dann feuern darf. Dadurch neu eingetragene Fakten können Regeln zur Konfliktmenge hinzufügen und andere wieder entfernen, daher kann die Regelauswahl von großem Einfluß auf das Ergebnis des Inferenzprozesses sein.

Aspekte, unter denen die Auswahl der zu feuernden Regel geschehen kann, sind z.B.

- Komplexität der Bedingung
  (Anzahl der Bedingungsklauseln)
- fester Gewichtungswert einer Regel
- „Neuheitswert“ (recency) der Fakten der Bedingung
- Zeitpunkt der letzten Ausführung der Regel
- Bekanntheitsgrad der Folgerung
  (Anzahl der neu einzutragenden bzw. zu ändernden Slotwerte)

---

[2] Im Gegensatz zu prädikativen Systemen wie Prolog, bei denen wegen der starren Reihenfolge der Regelsuche die Anordnung der Regeln benützt wird, um implizit Ausschlußbedingungen festzulegen.
Bsp.: `not(P) :- P, !, fail.`
`not(P).`

In beiden Verkettungsrichtungen ist die Frage nach der besten Strategie ohne Ansehen der Problemstellung nicht zu beantworten. Was für den einen Fall hervorragend geeignet ist, kann die Formulierung eines anderen Problems stark behindern. Eine gute Regelkomponente sollte daher dem Benutzer möglichst freie Hand bei der Auswahl seiner Regelselektionsstrategien lassen. Es ist sinnvoll, gewisse Strategiemodelle vordefiniert zur Verfügung zu stellen, jedoch muß auch die Definition eigener Strategien möglich sein.

2.2.3 Regelkategorien

Die Informationen über die Steuerung des Inferenzvorgangs, wie sie in Abschnitt 2.2.2 behandelt wurden, nämlich Verkettungsrichtung, Zugriffskontrolle auf die Wissensbasis und Konfliktlösung, müssen irgendwo festgehalten werden. Zur Auswahl stehen dabei in einem objektorientierten System prinzipiell drei Möglichkeiten:

1. **globale Festlegung:** Dies ist bei weitem zu unspezifisch – es wären z.B. im gesamten System nur eine Verkettungsrichtung und nur eine Konfliktlösungsstrategie möglich. Außerdem widersprechen solche globalen Festlegungen den objektorientierten Prinzipien.
2. **bei den Klassen:** Eine solche Zuordnung paßt gut in den objektorientierten Rahmen. Andererseits vergäbe man sich damit die Möglichkeit, Vorwärts– und Rückwärtsverkettung wahlweise zu betreiben oder verschiedene Teilgebiete der Wissensbasis mit unterschiedlichen Konfliktlösungsstrategien zu bearbeiten.
3. **bei den Regeln:** Die Nachteile aus Punkt 2 wären dabei vermieden, allerdings ist ein hoher Preis in Form von Redundanz dafür zu zahlen. Auch ist die Konfliktlösungsstrategie nicht eigentlich eine Regeleigenschaft, sondern eher ein Kennzeichen einer Menge von Regeln.

Aus der obigen Aufstellung geht hervor, daß der geeignetste Ort für die Festlegung der inferenzsteuernden Details irgendwo in der Mitte zwischen Klassen und den ihnen zugeordneten Regeln liegt. Aus diesem Grund werden *Regelkategorien* eingeführt. Bei einer Klasse von Objekten, auf die Regeln angewandt werden sollen, werden zunächst eine oder mehrere Regelkategorien definiert. Diese enthalten dann jeweils die Informationen über Verkettungsrichtung und Konfliktlösung, sowie eine beliebige Anzahl von Regeln, die nach diesen Festlegungen behandelt werden. Auch die Zugriffsberechtigung auf die Wissensbasis wird mit den Regelkategorien assoziiert. Bei der Klasse selbst muß nur noch vereinbart werden, wie die Kategorien verarbeitet werden.

2.2.4 Vererbung von Regeln

In OOPS spielt das Prinzip der Vererbung eine große Rolle. Vererbungshierarchien machen es möglich, Eigenschaften, die in Klassen definiert wurden, schrittweise zu spezialisieren bzw. neue Eigenschaften hinzuzufügen. So können z.B. bei teilweiser Überschneidung von Objekten gleiche Strukturen oder Methoden in einer gemeinsamen Oberklasse vereinbart werden, während in den verschiedenen Unterklassen nur noch die davon abweichenden Punkte definiert werden müssen. Dadurch wird eine erhebliche Verminderung der Redundanz in der Programmierung erreicht.

Ähnliche Überlegungen lassen sich nun auch für Regeln anstellen. Eine Menge von Regeln, die für die Objekte einer Klasse definiert wurde, wird generell auch für Objekte ihrer Unterklassen gelten, wobei natürlich nicht ausgeschlossen ist, daß gewisse Änderungen vorgenommen werden. An Abänderungen existieren dabei im Einzelnen folgende Möglichkeiten:

- Hinzufügen neuer Regeln
- Überdecken (shadowing) von bestehenden Regeln; dies schließt auch völlige Ausblendung der Regel ein
- Spezialisierung bestehender Regeln, d.h., es werden zusätzliche Klauseln in Bedingung, Folgerung oder Aktion eingefügt

Wie sich diese Möglichkeiten bei der Regelvererbung praktisch auswirken können, wird in Abb. 2.5 verdeutlicht.

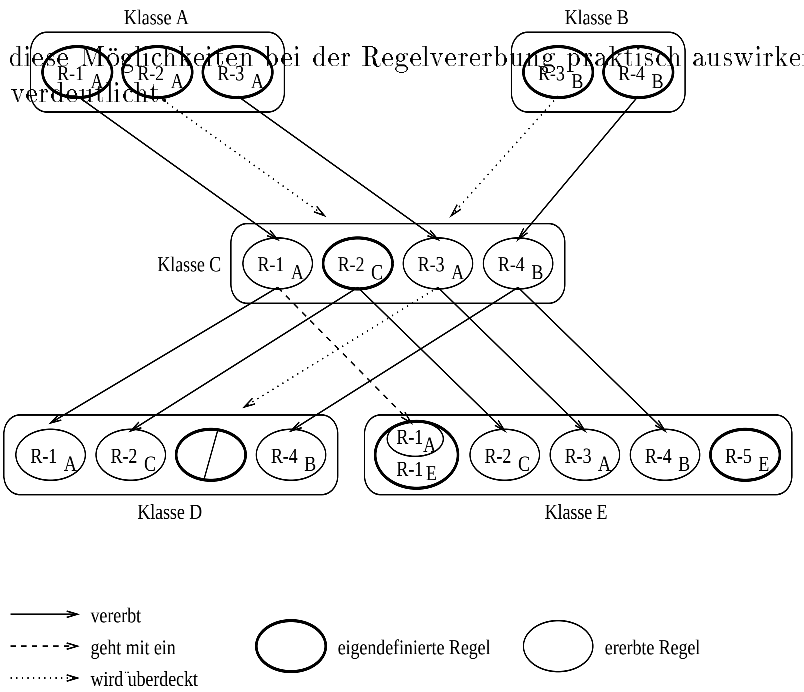

Abbildung 2.5: Beispiel für Regelvererbung

Die oben erwähnten Fälle, die bei der Vererbung von Regeln auftreten können, sind dieselben, die auch für Methoden gelten. Im Unterschied zu Methoden, die direkt vererbt werden, geschieht dies für Regeln jedoch auf dem Umweg über die Regelkategorien. Grundsätzlich erbt eine Klasse alle Kategorien der Vaterklassen mitsamt den darin enthaltenen Regeln. Auf die Menge der Regeln können die oben aufgeführten Änderungen angewandt werden, die Eigenschaften Verkettungsrichtung und Konfliktlösung der Kategorien können aber von den Unterklassen nicht mehr verändert werden.

2.3 Vergleich mit anderen Systemen

In diesem Abschnitt werden die Eigenschaften des bisher beschriebenen Systems mit denen anderer Systeme, bei denen ebenfalls Möglichkeiten zur Regelverarbeitung innerhalb einer objektorientierten Umgebung bestehen, verglichen. Besonders berücksichtigt wurden dabei die Anordnung der Wissensbasis im objektorientierten System, der strukturelle Aufbau der Regelbasis sowie eventuelle Möglichkeiten zur Vererbung von Regeln.

2.3.1 Objektorientierte Produktionsregeln in ObjTalk

Produktionsregeln im ObjTalk-Regelsystem [Girgensohn 86] sind in der Grundstruktur (Bedingung, Folgerung, Aktion) denen des FrameTalk-Systems sehr ähnlich. Die Bedingung einer Regel besteht dabei aus der logischen Verknüpfung von Faktbeschreibungen, die in disjunktiver Normalform abgelegt wird. Innerhalb der Regel ist die Verwendung von Variablen zugelassen, die an Objekte oder Slotwerte gebunden werden können.

Es findet allerdings keine objektlokale Anordnung der Wissensbasis statt, d.h. die Regeln sind globaler Natur und die Datenbasis besteht aus dem gesamten Objektraum. Spezialisierung von Regeln auf Objekte einzelner Klassen ist nur durch Formulieren dieser Einschränkungen in Klauseln der Bedingung möglich.

Regeln werden zu Regelmengen zusammengefaßt, innerhalb derer für beide Verkettungsrichtungen Konfliktlösungsstrategien definiert werden können. Regelmengen können explizit aufgerufen werden, indem an sie die Nachrichten `assert` oder `query` versandt werden, oder automatisch von Slotzugriffsdämonen aktiviert werden. Konfliktlösungsstrategien für den Fall, daß mehrere Regelmengen gleichzeitig automatisch aktiviert werden, sind nicht beschrieben.

2.3.2 Loops

In Loops [BobrowStefik 81] werden Regeln verwendet, die aus mehreren, implizit konjunktiv verknüpften Bedingungen und mehreren Folgerungsklauseln bestehen. Bei den Folgerungen wird keine Unterscheidung zwischen Datenbasiseinträgen und Seiteneffekten gemacht; damit ist keine Rückwärtsverkettung möglich.

Auch hier wird die Organisation von Regeln in Regelmengen vorgenommen. Diesen wird als Gültigkeitsbereich eine Klasse zugewiesen, die Datenbasis besteht damit hauptsächlich aus Slots von Objekten dieser Klasse. Spezialisierende Vererbung von Regelmengen ist jedoch nicht möglich.

Innerhalb von Regelmengen ist die Auswahl unter mehreren Kontrollstrukturen möglich. Diese haben jedoch einen eher prozeduralen Charakter, da sie die Reihenfolge, in der Regeln getestet und gegebenenfalls ausgeführt werden, nicht beeinflussen können. Regelmengen können sowohl prozedural als auch datengesteuert oder durch die Folgerung einer Regel aufgerufen werden, jedoch immer nur als Ganzes. Gezieltes Untersuchen einzelner Regeln ist nicht möglich.

2.3.3 Kee

Die Struktur des Regelsystems in Kee [KEE 85] entspricht weitgehend der des in Abschnitt 2.3.1 beschriebenen ObjTalk–Regelsystems. Auch in Kee muß die Spezialisierung von in Regelmengen zusammengefaßten Regeln nach Klassenzugehörigkeit des betrachteten Objekts explizit in der Regelbedingung geschehen.

Die Vorzüge dieses Systems liegen in der reichhaltig ausgestatteten Benutzeroberfläche. Objektorientierte Browser, mit denen auch Regeln bearbeitet werden können, sowie verschiedene grafische Werkzeuge z.B. zur Darstellung von Hierarchien ermöglichen es, mit diesem System komfortabel Wissensverarbeitung zu betreiben.

2.3.4 Orient84/K

Im Gegensatz zu den drei anderen in diesem Abschnitt betrachteten Systemen werden in Orient84/K [TokoroIshikawa 87] keine Produktionsregeln, sondern ähnlich wie in Prolog Hornklauseln verwendet. Deshalb ist von der Regelstruktur her ein Vergleich mit FrameTalk schlecht möglich.

Interessant ist jedoch, daß auch in Orient84/K die Wissensbasis objektlokal angeordnet ist. Jedes Objekt kann an seine private Wissensbasis, die aus Prädikaten und Fakten besteht, durch die Versendung der Nachrichten `unify:` *aClause* bzw. `forEachUnify:` *aClause* `do:` *aBlock* Prolog–ähnliche Anfragen richten, wobei in der Anfrage enthaltene Variablen gebunden werden. Ebenso ist es möglich, in Hornklauseln durch Verwendung des Prädikats `self` Methoden des Objekts ausführen zu lassen, die Wahrheitswerte zurückgeben müssen.

Ebenso wie Methoden können auch Klauseln der Wissensbasis von Klassen an ihre Subklassen vererbt werden, wobei die Identifikation der bei der Vererbung als Einheit zu betrachtenden Klauseln durch das herzuleitende Literal geschieht.

Teil 3

Implementierung

3.1 Übersicht

3.1.1 Umfang der Implementierung

Das im Teil 2 beschriebene Regelsystem ist auf Grund der Absicht, Benutzer durch systembedingte Einschränkungen so wenig wie möglich zu behindern, sehr umfangreich. Zur Implementierung gelangen konnte daher nur ein Ausschnitt der entworfenen Strukturen. Dabei wurden jedoch auch die nicht realisierten Teile nicht aus den Augen verloren, um eine spätere Erweiterung des Systems um diese Einheiten so einfach wie möglich zu machen. Dieses Vorgehen erschien sinnvoller, als sich von vorneherein auf ein kleineres System festzulegen.

Die leichteste Faktorisierung des Systems war nach der Richtung der Regelverkettung möglich. Hierbei ergeben sich die wenigsten Überschneidungen, daher war auch der Aufwand, der betrieben werden mußte, um nachträgliche Vervollständigungen nicht zu behindern, am geringsten. Die Wahl fiel aus den in Abschnitt 2.2.2.1 erwähnten Gründen auf die Vorwärtsverkettung. Damit lassen sich die Fähigkeiten des implementierten Systems wie folgt zusammenfassen:

- Vorwärtsverkettung von Regeln
- automatische Inferenzauslösung nach Veränderung von Slotwerten[1]
- Auswahl unter mehreren vordefinierten Konfliktlösungsstrategien;
  Eigendefinition neuer Strategien ist ebenfalls möglich

[1] Dabei ist es Objekten möglich, die Inferenzauslösung zu verhindern, indem sie Teile ihrer Wissensbasis sperren.

3.1.2 Beschreibung des Beispiels

In diesem Abschnitt soll das Anwendungsbeispiel vorgestellt werden, das im weiteren zur Verdeutlichung der Verwendung von bestimmten Eigenschaften des Regelsystems benützt wird. Es handelt sich dabei um eine einfache Konfigurationsaufgabe aus dem Bereich der Personal Computer. Diese Rechnertypen lassen sich, wie in Abb. 3.1 dargestellt, in einer hierarchischen Struktur anordnen.

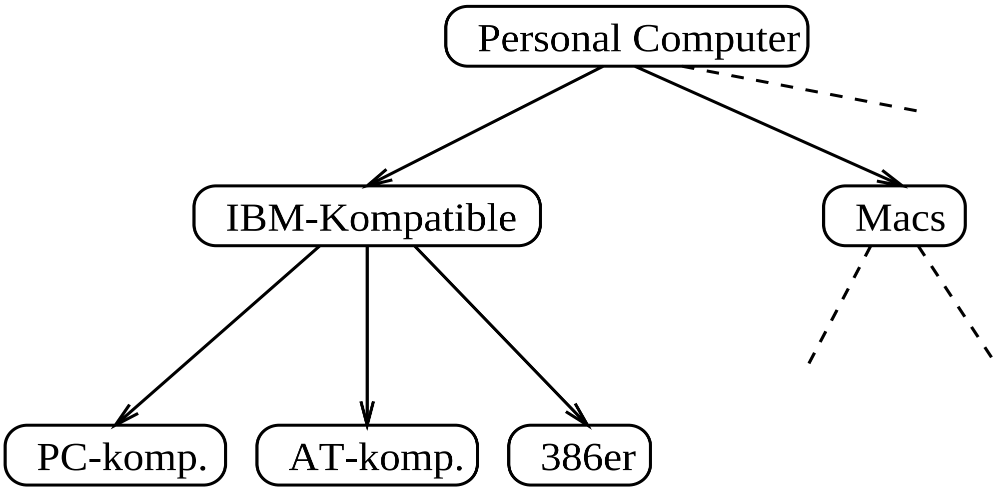

Abbildung 3.1: Hierarchische Anordnung von Personal Computern

Startinformationen für die Konfiguration sind Rechnertyp, Art der Anwendung und Angaben über die evtl. Netzanbindung. In diesem Beispiel vorgesehene Anwendungen sind

- Textverarbeitung,
- irgendeine Datenverarbeitung und
- CAD;

für die Netzanbindung stehen die Möglichkeiten

- Stand-Alone,
- Workstation und
- Fileserver

zur Verfügung. In Abhängigkeit von diesen Angaben soll dann die nötige Rechnerausstattung festgelegt werden.

3.2 Aufbau einer Wissensbasis

3.2.1 Klassen

Bei der Vereinbarung von Klassen, deren Objekte eine Wissensbasis besitzen sollen, müssen gegenüber herkömmlichen Klassen zusätzliche Angaben gemacht werden. Dies ist nötig, um folgende Aktionen durchführen zu können:

- Veränderung der Klassendefinition
  (automatische Einbindung von Mixin-Klassen, andere Metaklasse)
- Generierung von zusätzlichen Methoden,
  insbesondere Slotzugriffsdämonen
- Verarbeitung von Informationen über die Konfliktlösungsmodelle

Die Form der neuen Klassenvereinbarung lautet[2]:

> (`defrclass` *class-name* ({*superclass-name*}*)
>   ({*slot-specifier*}*)
>   ⟦↓*class-option*⟧)
>
> *slot-specifier* ::= *slot-name* | (*slot-name* ⟦↓*slot-option*⟧)

Gegenüber der Definition von Klassen in CLOS mittels `defclass` sind folgende Optionen hinzugekommen:

bei slot-option:

`:leave-out` *boolean*, um einen Slot als nicht zur Datenbasis des Objekts gehörig zu bezeichnen

`:all-writes` *boolean*, um festzulegen, ob bei allen Schreibzugriffen auf Slots die Inferenzmaschine gestartet wird oder nur bei Änderungen der dort enthaltenen Werte

bei class-option:

`:conflict-models` *conflict-model-list*, um die möglichen Konfliktlösungsmodelle anzugeben (s.a. Abschnitt 3.2.3)

[2]Der hier und in den folgenden Abschnitten sowie im Anhang verwendete Formalismus zur Beschreibung der Syntax entspricht der erweiterten BNF, wie sie in [Bobrow et al. 88] für die Definition der CLOS–Sprachelemente eingeführt wird.

Zu der in Abschnitt 3.1.2 vorgestellten Konfigurierungsaufgabe werden hier die zugrundeliegenden Klassen definiert. Die Struktur aus Abb. 3.1 wird durch die bei `computer` beginnende Klassenhierarchie realisiert. Der Slot `price` in der Klasse `system` wird als nicht zur Datenbasis gehörig gekennzeichnet.

```
(defrclass system ()
  ((computer  :documentation "Eigentlicher Rechner")
   (use       :documentation "Art der Nutzung")
   (net-env   :documentation "Netzanbindung")
   (price     :documentation "Der Preis spielt keine Rolle" :leave-out t))
  (:documentation "So sieht ein Computersystem aus"
   :conflict-models (simple-weight))
  )

(defrclass computer ()
  ((cpu         :documentation "CPU des Rechners")
   (memory-size :documentation "Groesse des Hauptspeichers")
   (add-ons     :documentation "Zusatzkarten")
   (periphery   :documentation "Peripheriegeraete"))
  (:documentation "Der Aufbau eines Computers"
   :conflict-models (simple-weight))
  )

(defrclass ibm-computer (computer)
  ((fpu   :documentation "Mathematischer Coprozessor")
   (8-bit-slots  :documentation "Anzahl der freien 8-bit-Slots"))
  )

(defrclass mac-computer (computer)
  ((fpu   :documentation "Mathematischer Coprozessor")
   (slots :documentation "Anzahl der freien Slots"))
  )

(defrclass pc-computer (ibm-computer)
  ()
  )

(defrclass at-computer (ibm-computer)
  ((16-bit-slots :documentation "Anzahl der freien 16-bit-Slots"))
  )

(defrclass 386-computer (ibm-computer)
  ((16-bit-slots :documentation "Anzahl der freien 16-bit-Slots")
   (32-bit-slots :documentation "Anzahl der freien 32-bit-Slots"))
  )
```

3.2.2 Regelkategorien und Regeln

Wie in Abschnitt 2.2.3 beschrieben, werden Regeln in Regelkategorien zusammengefaßt, die auch die Verwaltung von Steuerungsinformationen für die Inferenzmaschine übernehmen. Verwandte Regelkategorien, also Regelkategorien in verschiedenen Klassen, die den gleichen Namen haben, sind Instanzen derselben Kategorienklasse. Dies ist notwendig, damit die oben erwähnten Steuerungsinformationen zentral verwaltet werden können – für Kategorien gleichen Namens müssen diese wegen ihrer Vererbung identisch sein. Außerdem können so alle verwandten Regelkategorien gleichzeitig inaktiv geschaltet werden, wenn z.B. Änderungen oder Neueinträge von Regeln Folgeänderungen von Regelkategorien in Unterklassen bedingen. Kategorienklassen können explizit durch `defcategory` oder implizit durch die Vereinbarung einer Regelkategorie bei einer Klasse definiert werden. Die Verwendung von `defcategory` ist:

(`defcategory` *category-name* ({*superclass-name*}*) ⟦↓*category-option*⟧)

category-option ::= (`:documentation` *string*) |
(`:conflict-model` *conflict-model-name*)

Für die Konfiguration der Rechner kann also folgende Kategorienklasse definiert werden:

```
(defcategory computer-config ()
  (:documentation  "Konfiguration der Computer")
  )
```

Beim Erzeugen von Regelkategorien ist die Angabe der Klasse, der sie zugeordnet werden sollen, obligatorisch. Dadurch kann verhindert werden, daß für die Regelkategorien ein Konfliktlösungsmodell benutzt wird, das bei der Klassendefinition nicht vorgesehen wurde und evtl. zu den anderen Modellen inkompatibel ist. Optionale Angaben sind Initialisierungsargumente zu Slots der Regelkategorie (z.B. bei statisch gewichteten Regelkategorien `:weight` *number*) und Regeln, die sie enthalten soll. Falls für die Regelkategorie noch keine Kategorieklasse existiert, können außerdem das Konfliktlösungsmodell und die Klassenhierarchie festgelegt werden; damit wird dann automatisch eine Kategorieklasse erzeugt. Angelegt wird eine Regelkategorie durch:

(`defrulecat` *rulecat-name* ({*superclass-name*}*) *object-class*
(*rule-specifier**) ⟦↓*rulecat-option*⟧)

rulecat-option ::= (`:documentation` *string*) |
(`:conflict-model` *conflict-model-name*)
(*initargs*)

rule-specifier ::= (*rule-name rule-body* ⟦↓ *rule-option*⟧)

rule-body und *rule-option* s.u.

Im Beispiel wird die Zusammenstellung des Gesamtsystems von der Regelkategorie `system-config` übernommen, die hier einschließlich einiger ihrer Regeln festgelegt wird:

```
(defrulecat system-config () system
  ((network-rule-1
    (((member 'file-server #>net-env))
     ((#>(computer periphery)
       (union #>(computer periphery) '(harddisk tapedrive))))
     ()
     )
    )
   (network-rule-2
    (((member 'file-server #>net-env)
      (not (member 'bw-monitor #>(computer periphery)))
      (not (member 'color-monitor #>(computer periphery))))
     ((#>(computer periphery)
       (union #>(computer periphery) '(bw-monitor))))
     ()
     )
    )
   )
  (:documentation  "Systemkonfiguration")
  )
```

Die definierten Regeln bewirken, daß für einen Fileserver automatisch eine Festplatte und ein Bandlaufwerk hinzugefügt wird (`network-rule-1`) sowie daß, falls noch kein Monitor vorgesehen ist, ein Schwarzweißbildschirm verwendet wird (`network-rule-2`).

Außerdem muß für die Klasse `computer` noch eine Regelkategorie erzeugt werden. Diese wird dann automatisch an ihre Subklassen vererbt.

```
(defrulecat computer-config () computer)
```

Weiter Angaben über Konfliktlösungsmodell oder Superklassen sind hier nicht zulässig, da bereits eine Kategorienklasse mit diesem Namen existiert.

Der Aufruf von `defrulecat` löst folgende Vorgänge aus:

- Suchen oder Erzeugen der Kategorienklasse
- Erzeugen der Regelkategorie
- Einbinden der Regelkategorie in die Objektklasse
- Erzeugen der angegebenen Regeln und Zuordnen zur Regelkategorie
- Vererben der Kategorie

Statt innerhalb der Definition einer Regelkategorie können Regeln auch unter Verwendung des Macros `defrule` erzeugt werden. Da dabei die Angabe der Regelkategorie nicht zwingend ist, kann statt dessen festgelegt werden, zu welcher Unterklasse der Regelklasse `rule` die erzeugte Regel gehören soll. Stellt sich beim späteren Einbinden in eine Regelkategorie heraus, daß diese die Verwendung einer anderen Regelklasse verlangt, so wird die Umsetzung in die neue Klasse automatisch vorgenommen. Die Syntax von `defrule` ist:

(`defrule` *rule-name* ({*rule-class*}) ({*rule-allocation*}) *rule-body* ⟦↓ *rule-option*⟧)

rule-allocation ::= *rulecat-name class-name*

rule-body ::= (({*if-clause*}*) ({*then-clause*}*) ({*side-effect*}*))

rule-option ::= (`:documentation` *string*) |
(*initargs*)

Die Argumente *if-clause*, *then-clause* und *side-effect* entsprechen den Klauseln in Bedingungs-, Folgerungs- und Aktionsteil einer Regel, wie sie in Abschnitt 2.2.1.1 beschrieben wurde. Der Zugriff auf Slots und Slotpfade wurde mittels der Funktion `slot-path` realisiert. Das Argument dieser Funktion kann dabei der Name eines Slots oder eine Liste von Slotnamen sein. Anstelle von (`slot-path` *path*) kann auch, wie bereits in den Beispielen auf der vorigen Seite, `#>`*path* verwendet werden.

Dazu sollen hier einige weitere Konfigurationsregeln als Beispiel dienen: Um CAD zu betreiben, benötigt man einen Farbbildschirm und ein Graphiktablett, während man zur Textverarbeitung nur einen Schwarzweißbildschirm und eine Maus braucht. Ist jedoch schon ein Farbbildschirm vorhanden, so wird dieser benützt.

```
(defrule cad-rule-1 () (system-config system)
  (((member 'cad #>use))
   ((#>(computer periphery)
     (union #>(computer periphery) '(color-monitor graphics-tablet))))
   ())
  )

(defrule text-rule-1 () (system-config system)
  (((member 'text #>use)
    (member 'color-monitor #>(computer periphery)))
   ((#>(computer periphery)
     (union #>(computer periphery) '(mouse))))
   ())
  )

(defrule text-rule-2 () (system-config system)
  (((member 'text #>use))
   ((#>(computer periphery)
     (union #>(computer periphery) '(bw-monitor mouse))))
   ())
  )
```

3.2.3 Konfliktlösungsmodelle

Laut Abschnitt 2.2.3 wird die Steuerinformationen für die Inferenzmaschine generell von den Regelkategorien verwaltet, lediglich die Rangfolge der Regelkategorien ist bei den Klassen festgelegt. Da diese Steuerinformation aber aus einer Reihe von Einzelinformationen besteht, wäre es umständlich und außerdem fehlerträchtig, wenn diese jedesmal aufs Neue vollständig angegeben werden müßten. Die Überprüfung auf Kompatibilität zwischen Klassen- und Regelkategorieangaben sowie auf Widerspruchsfreiheit innerhalb dieser Deklarationen wäre äußerst aufwendig, teilweise sogar überhaupt nicht durchführbar. Daher wurde in dieses System die Verwendung von Konfliktlösungsmodellen aufgenommen. Ein solches Modell beinhaltet sämtliche Informationen, die zur Steuerung der Inferenzauslösung notwendig sind. Dies sind im einzelnen:

- Verkettungsrichtung
  (`:forward`, `:backward`)
- Zeitpunkt der Regelkategorie-Sortierung
  (`:at-definition-time`, `:before-rule-ordering`, `:after-rule-ordering`)
- Methode, die den Sortierwert der Regelkategorie zurückgibt
- Klassenzugehörigkeit der Regelkategorien[3]
  (als Liste der Superklassen)
- Zeitpunkt der Regelsortierung
  (`:at-definition-time`, `:before-rule-testing`, `:after-rule-testing`)
- Methode zum Vergleich zweier Regeln
- Klassenzugehörigkeit der Regeln[3]
  (als Liste der Superklassen)

Sollen innerhalb einer Klasse mehrere Konfliktlösungsmodelle zum Einsatz kommen, so gilt folgende Kompatibilitätsbedingung:

Zwei Konfliktlösungsmodelle sind kompatibel, wenn

1. die Verkettungsrichtung unterschiedlich ist

oder

2. bei gleicher Verkettungsrichtung auch der Zeitpunkt der Regelkategoriesortierung gleich ist.

Die Begründung für den ersten Fall ist, daß bei unterschiedlicher Verkettungsrichtung auch verschiedene Inferenzmaschinen eingesetzt werden, so daß hier keine Probleme auftreten

[3] Klassenzugehörigkeiten von Regelkategorien und Regeln ergeben sich bereits aus den anderen Angaben. Es ist jedoch möglich, in die Hierarchie noch weitere Klassen aufzunehmen.

können. Ist die Verkettungsrichtung jedoch die gleiche, so muß der Inferenzmaschine eindeutig mitgeteilt werden, ob die vorhandenen Regelkategorien überhaupt noch sortiert werden müssen (bei `:at-definition-time` ist dies nicht mehr nötig) und falls dies der Fall ist, ob dies vor oder nach der Regelsortierung zu geschehen hat (`:before-rule-ordering` bzw. `:after-rule-ordering`).

Konfliktlösungsmodelle werden folgendermaßen vereinbart (Defaults siehe Anhang):

(`define-conflict-model` *name direction* ⟦↓*option*⟧)

option ::= (`:documentation` *string*) |
(`:rulecat-sorttime` *rulecat-sorttime-option*) |
(`:rulecat-weightfn` *function*) |
(`:rulecat-supers` *superclass-list*) |
(`:rule-sorttime` *rule-sorttime-option*) |
(`:rule-sortfn` *function*) |
(`:rule-supers` *superclass-list*)

rulecat-sorttime-option ::= `:at-definition-time` |
`:before-rule-ordering` |
`:after-rule-ordering`

rule-sorttime-option ::= `:at-definition-time` |
`:before-rule-testing` |
`:after-rule-testing`

Mit den in diesem und in den beiden letzten Abschnitten aufgeführten Deklarationen läßt sich nun eine Gegenüberstellung zwischen dem operationalen Teil eines Objekts in CLOS und des regelbasierten Teils in FRAMETALK aufbauen:

CLOS	FRAMETALK–Regelsystem
Klassenvereinbarung `defclass`	erweiterte Klassenvereinbarung `defrclass`
Generische Funktionen `defgeneric`	Kategorienklassen `defcategory`
Methoden `defmethod`	Regelkategorien `defrulecat`
Methodenkombinationstypen `define-method-combination`	Konfliktlösungsmodelle `define-conflict-model`

Abbildung 3.2: Vergleich CLOS — FRAMETALK–Regelsystem

3.2.4 Ablauf der Vererbung

Wie aus dem bisher Gesagten bereits hervorgeht, findet die Vererbung der Wissensbasis im Regelsystem zweistufig statt: In der ersten Stufe wird vererbt, was den Klassen bekannt ist (also Informationen, die in Slots der Klassen stehen), wohingegen in der zweiten Stufe daraus erst hervorgehende Details weitergegeben werden. Zu den klassenresidenten Merkmalen gehören

die Datenbasis, d.i. die Menge der Slots, sowie

die Grobstruktur der Regelbasis mit den Teilen

- verfügbare Konfliktlösungsmodelle und
- Regelkategorien.

Die Vererbung der Slotinformationen wird bereits vom zugrundeliegenden System (CLOS) übernommen. Die Information über die Regelbasis, die in den Slots `conflict-models` und `rulecats` der Klassen gehalten wird, muß jedoch separat vererbt werden. Dabei kann das Problem auftreten, daß die Konfliktlösungsmodelle nicht verwandter Superklassen inkompatibel sind. In diesem Fall gelten die Vereinbarungen derjenigen Superklasse, die nach der Class Precedence List den Vorrang hat. (Innerhalb einer Klasse können grundsätzlich nur kompatible Konfliktlösungsmodelle vereinbart werden.) Regelkategorien werden im Folgenden nur dann vererbt, wenn sie eines der zulässigen Konfliktlösungsmodelle verwenden. Ebenso werden Neudefinitionen von Konfliktlösungsmodellen und Regelkategorien nur dann zugelassen, wenn sie mit den bisherigen kompatibel sind.

Eine andere Möglichkeit zur Behandlung der Vererbung von Konfliktlösungsmodellen wäre gewesen, beim Auftreten eines Widerspruchs die Klassenerzeugung einfach abzubrechen. Dies hätte aber zur Folge gehabt, daß eine einmal erfolgte Festlegung von Konfliktlösungsmodellen nicht mehr geändert werden könnte, da sonst evtl. Klassen gelöscht werden müßten. Bei der ersten Methode können schlimmstenfalls Regelkategorien gelöscht bzw. bei Nichtverträglichkeiten mit der Superklasse die Neudefinition als fehlerhaft zurückgewiesen werden. Da diese Folgen als weniger schwerwiegend anzusehen sind, wurde die zweite Variante nicht weiterverfolgt.

Um die Vererbung der Wissensbasis abzuschließen, fehlt nun noch

die Feinstruktur der Regelbasis, nämlich

- die Regeln.

Jede Regelkategorie besitzt zwei Slots zur Aufnahme von Regeln, nämlich `direct-rules` und `effective-rules`. In `direct-rules` werden die für die jeweilige Klasse neu definierten Regeln gehalten. Da bei der Klassenvereinbarung solche Regeln noch nicht existieren, bleibt dieser Slot vorerst unbelegt. `Effective-rules` soll die Gesamtmenge der in dieser Kategorie für die Klasse gültigen Regeln enthalten. Dazu werden zunächst die ererbten Regeln gesammelt. Dies ist die Vereinigung der `effective-rules` aller direkten Vaterklassen, wobei im Fall der Namensgleicheit von Regeln nach der Ordnung der Class Precedence List ausgewählt wird. Liegen nun keine `direct-rules` vor, so ist der Vererbungsvorgang für diese Regelkategorie abgeschlossen und dem Slot `effective-rules` werden die ererbten Regeln zugewiesen.

Existieren dagegen schon `direct-rules` – etwa, weil der Vererbungsvorgang durch Änderungen in einer Superklassse ausgelöst wurde –, so müssen die ererbten Regeln gegen diese noch abgeglichen werden. Ist unter einem Namen nur entweder eine ererbte oder eine direkt definierte Regel vorhanden, so besteht keine Konfliktsituation. Andernfalls wird zunächst der Slot `specialize` der direkt definierten Regel abgeprüft. Ist er *nil* (default), so überdeckt die direkt definierte Regel die ererbte; ansonsten werden die Klauseln beider Regeln verbunden. Der letztere Fall ist in etwa analog zur Verwendung von `call-next-method` im Anweisungsteil einer Methode in CLOS.

Bisher konnte der Eindruck entstehen, die beiden Phasen der Vererbung der Wissensbasis fänden zwar nacheinander, aber in jedem Fall direkt aufeinanderfolgend statt. Dies ist jedoch nicht prinzipiell so. Grundsätzlich wird beim Erstellen oder Ändern einer Klassendefinition sowie beim Einfügen oder Löschen von Konfliktlösungsmodellen oder Regelkategorien einer Klasse nur die erste Phase durchgeführt. Alle davon betroffenen Kategorien werden dabei als inaktiv markiert. Diese Markierung wird auch vorgenommen, wenn an den `direct-rules` einer Regelkategorie eine Änderung durchgeführt wird. Die Vervollständigung der Vererbung für eine Kategorie muß dann noch durch den Aufruf (`activate-rulecat` *rulecat*) ausgelöst werden. Nur falls der Slot `auto-activate` der Kategorie mit einem von *nil* verschiedenen Wert belegt wurde, erfolgt dieser Aufruf automatisch an Phase Eins anschließend. Diese Belegung ist aber nur dann sinnvoll, wenn jeweils nur wenige Änderungen vorgenommen werden sollen – z.B. beim Debuggen –, da dann nach jeder einzelnen Änderung (beispielsweise einer Regel) die gesamte Vererbungsstruktur für die betreffende Kategorie neu berechnet wird.

Im Abschnitt 3.2.2 wurde die Regelkategorie `computer-config` für die Klasse `computer` definiert. Durch die oben betrachteten Vererbungsmechanismen erhalten nun alle Subklassen von `computer` ebenfalls eine Regelkategorie mit diesem Namen. Nun seien in `computer-config` bei der Klasse `computer` folgende Regeln für die Notwendigkeit von Zusatzkarten eingeführt worden:

```
(defrule harddisk-rule-1 () (computer-config computer)
  (((member 'harddisk #>periphery))
   ((#>add-ons (union #>add-ons '(hd-controller))))
   ())
  )

(defrule mouse-rule-1 () (computer-config computer)
  (((member 'mouse #>periphery))
   ((#>add-ons (union #>add-ons '(mouse-interface))))
   ())
  )
```

Auch diese Regeln werden dann bei der Vervollständigung der Vererbung (Phase 2) an alle Subklassen von `computer` weitergegeben. Werden jedoch in einer dieser Subklassen gleichnamige Regeln definiert, so können diese die in `computer` vereinbarten überdecken oder spezialisieren. Bei der Klasse `mac-computer` ist zum Beispiel kein Maus–Interface nötig:

```
(defrule mouse-rule-1 () (computer-config mac-computer)
  (((member 'mouse #>periphery))
   ()
   ())
  (:documentation "Maus-Interface bereits eingebaut")
  )
```

Eine Spezialisierung für `ibm-Computer` wäre z.B., daß beim zusätzlichen Verwenden eines Maus–Interface die Anzahl der freien 8-bit-Slots um eins erniedrigt würde. Dies ließe sich unter Verwendung der Regeloption `:specialize` folgendermaßen formulieren:

```
(defrule mouse-rule-1 () (computer-config ibm-computer)
  (((not (member 'mouse-interface #>add-ons)))
   ((#>8-bit-slots (1- #>8-bit-slots)))
   ())
  (:documentation "Zusaetzlich Anzahl der freien Slots korrigieren")
  (:specialize t)
  )
```

Durch diese Deklaration würden die Klauseln der in der bei `computer` definierten Regel der hier festgelegten hinzugefügt.

3.3 Arbeit der Inferenzmaschine

3.3.1 Auslösung der Inferenz

Beim Forward Chaining findet die Inferenzauslösung datengetriggert statt, d.h. dann, wenn Schreibzugriffe auf einen Slot erfolgen. Dabei kann noch unterschieden werden, ob bereits das reine Beschreiben des Slots als Auslösefaktor zählt, oder ob zusätzlich gelten muß, daß der Slotwert geändert wurde. Diese Entscheidung erfolgt durch die Slot-Option `:all-writes` bei der Klassendefinition, wobei der Defaultwert *nil* ist. Die Überwachung der Auslösebedingung erfolgt durch Slotzugriffsdämonen[4], die dann folgende Aktionen durchführen:

1. Markieren des Slots als beschrieben
2. Auslösen des Inferenzprozesses für das betreffende Objekt

3.3.2 Zyklus der Inferenzmaschine

Objekte haben zwar eine eigene Wissensbasis, jedoch muß nicht jedes Objekt eine Inferenzmaschine für sich alleine besitzen. Statt dessen gibt es im System für jede Verkettungsrichtung eine allgemeine Inferenzmaschine, die allen Objekten zur Verfügung steht[5]. Sie ist ein Objekt einer Subklasse der Klasse `chainer`, bei der Vorwärtsverkettung ist dies in der Regel die Klasse `forward-chainer`. Instanzen dieser Klasse verwalten eine Warteliste der Objekte, für die Inferenz betrieben werden soll. Meldet ein Slotzugriffsdämon für ein Objekt das Zutreffen der Bedingungen für die Inferenzauslösung, so wird dieses Objekt nach der durch den Slot `queue-sortfn` festgelegten Methode in die Warteliste (queue) eingetragen. Defaultwert für die Eintragungsmethode ist `to-beginning`, d.h. das neue Objekt darf sich „vordrängeln".

Der Grund dafür liegt in der Möglichkeit, in Regeln eines Objekts Slots von über Slotpfade erreichbaren Objekten zu referenzieren. Wird nun durch Schreiben auf einen Slotpfad bei einem „tieferliegenden" Objekt ebenfalls ein Inferenzprozeß ausgelöst, so können weitere Slots dieses Objekts geändert werden. Dies wiederum könnte weitere Regelanwendungen im ursprünglichen Objekt bedingen. Ist zu diesem Zeitpunkt der Inferenzprozeß am ursprünglichen Objekt bereits abgeschlossen, so unterbleiben diese Regelanwendungen, da die Regeln des „tieferliegenden" Objekts nichts darüber aussagen können, daß es von einem anderen referenziert wird[6]. Wird aber zuerst das „tieferliegende" Objekt von der Inferenzmaschine bearbeitet, dann existiert für den Slot, über den es referenziert wurde,

[4] Diese sind als Around-Methoden der Slot-Accessoren realisiert.

[5] Es ist auch denkbar, daß verschiedene Objekte unterschiedliche Inferenzmaschinen besitzen. Dies kann durch Überdefinieren der Methode `forward-chainer`, die normalerweise `*system-forward-chainer` zurückgibt, realisiert werden.

[6] siehe dazu Teil 4, Ausblick.

noch ein „beschrieben“-Vermerk, der bewirkt, daß die dadurch betroffenen Regeln auf die Erfüllung ihrer Bedingung untersucht werden.

Ein solcher Fall kann auch im Beispiel der Rechnerkonfiguration vorkommen. Für den Fall, daß CAD betrieben werden soll, oder wenn der Rechner als Workstation im Netz arbeiten soll, wird festgelegt, daß der Hauptspeicher nicht kleiner als 1MB sein darf. Sonst wird eine Speichererweiterungskarte hinzugefügt. Diese Überprüfung wird durch die folgenden Regeln vorgenommen:

```
(defrule workstation-rule-1 () (system-config system)
  (((= 'workstation #>net-env)
    (< 1024 #>(computer memory-size)))
   ((#>add-ons (union #>add-ons '(memory-card))))
   ())
  )

(defrule cad-rule-2 () (system-config system)
  (((member 'cad #>use)
    (< 1024 #>(computer memory-size)))
   ((#>add-ons (union #>add-ons '(memory-card))))
   ())
  )
```

Bei der Klasse `computer` wird nun die Auswirkung des Speicherkarteneinbaus behandelt:

```
(defrule memory-rule-1 () (computer-config computer)
  (((member 'memory-card #>periphery))
   ((#>memory-size (+ #>memory-size 1024)))
   ())
  )
```

Würden nun zuerst die beiden Regeln bei einer Instanz der Klasse `system` verarbeitet, so würden zwei Speichererweiterungskarten eingebaut. Wenn aber die Instanz von `computer` von der Inferenzmaschine vorgezogen wird, dann hat sich die Speichererweiterung bereits auf die Speichergröße ausgewirkt, bevor die zweite Regel an die Reihe kommt. Infolgedessen sind die Bedingungen der zweiten Regel nicht erfüllt und der Speicher wird nur einmal vergrößert.

Die Inferenzmaschine durchläuft dann eine Schleife, in der jeweils das vorderste Objekt der Warteliste auf triggernde Regeln untersucht und gegebenenfalls eine dieser Regeln abgefeuert wird. Durch Slotbelegungen im Verlauf des Feuerns betroffene Objekte werden wiederum in die Warteliste eingetragen, wobei Doppeleinträge beseitigt werden. Sobald keine Objekte mehr auf Bearbeitung warten, gibt die Inferenzmaschine die Kontrolle an diejenige Funktion oder Methode zurück, in der die zu ihrem Aufruf führende Slotbelegung stattgefunden hat.

3.3.3 Konfliktmengenbildung und Regelselektion

Die Slotzugriffsdämonen markieren, wie in Abschnitt 3.3.1 beschrieben, die geänderten Slots bei jedem Objekt. Ist nun ein solches Objekt an der Reihe, von der Inferenzmaschine bearbeitet zu werden, so werden daraus zunächst in sämtlichen Regelkategorien alle Regeln berechnet, die auf Grund der geänderten Slots triggern könnten. Diese bilden – zusammen mit denjenigen Regeln, die evtl. noch aus vorangegangenen Zyklen der Inferenzmaschine stammen und noch nicht gefeuert haben – die Konfliktmenge[7]. Die im Konfliktlösungsmodell festgelegten Methoden für die Bewertung der Regelkategorien und die Sortierung der Regeln wählen daraus die abzufeuernde Regel aus. Je nach den voreingestellten Regel- und Regelkategorieauswahlzeitpunkten geschieht diese Auswahl in der in Abb. 3.3 dargestellten Weise. Dabei ist zu beachten, daß Regeln, die als nicht triggernd erkannt werden, sowie Regelkategorien, die keine möglichen Regeln mehr enthalten, aus der Konfliktmenge entfernt und daher bei der Auswahl nicht mehr berücksichtigt werden.

Regelsortier- Regel- zeitpunkt kategorie- sortierzeitpunkt	:at-definition-time	:before-rule-testing	:after-rule-testing
:at- definition time	solange Regelkat. vorhanden wähle 1. Regelkat. solange Regeln darin wähle 1. Regel falls diese triggert $\uparrow$diese Regel	solange Regelkat. vorhanden wähle 1. Regelkat. sortiere Regeln darin solange Regeln darin wähle 1. Regel falls diese triggert $\uparrow$diese Regel	solange Regelkat. vorhanden wähle 1. Regelkat. teste alle Regeln darin sortiere Regeln darin $\uparrow$1. Regel
:before- rule- ordering	sortiere Regelkategorien solange Regelkat. vorhanden wähle 1. Regelkat. solange Regeln darin wähle 1. Regel falls diese triggert $\uparrow$diese Regel	sortiere Regelkategorien solange Regelkat. vorhanden wähle 1. Regelkat. sortiere Regeln darin solange Regeln darin wähle 1. Regel falls diese triggert $\uparrow$diese Regel	sortiere Regelkategorien solange Regelkat. vorhanden wähle 1. Regelkat. teste alle Regeln darin sortiere Regeln darin $\uparrow$1. Regel
:after- rule- ordering	sortiere Regelkategorien solange Regelkat. vorhanden wähle 1. Regelkat. solange Regeln darin wähle 1. Regel falls diese triggert $\uparrow$diese Regel	sortiere Regeln in allen RK. sortiere Regelkategorien solange Regelkat. vorhanden wähle 1. Regelkat. solange Regeln darin wähle 1. Regel falls diese triggert $\uparrow$diese Regel	teste Regeln in allen RK. sortiere Regeln in allen RK. sortiere Regelkategorien wähle 1. Regelkat. $\uparrow$1. Regel daraus

Abbildung 3.3: Regelselektion in FRAMETALK

[7]Abweichend von der üblichen Definition wird hier nicht die Menge der triggernden Regeln als Konfliktmenge angenommen, sondern die der *möglicherweise* triggernden. Dies ist darin begründet, daß die Sortierung der Regeln *vor* ihrer Überprüfung stattfinden kann, um so ein heuristisches Vorgehen zu ermöglichen.

3.3.4 Regelanwendung

Die Regel, die auf Grund der Sortiervorschriften ausgewählt wurde, muß nun feuern. Dies geschieht, indem zuerst die im Folgerungsteil stehenden Zuweisungen von Werten an die Slots (bzw. Slotpfade) durchgeführt und danach die LISP-Ausdrücke des Aktionsteils evaluiert werden.

Dabei werden evtl. neue Objekte in die Warteliste des Chainers eingetragen, ihre Bearbeitung erfolgt aber erst nach der Beendigung des Feuerns. Dadurch ist es z.B. möglich, Breakpoints für einen Regeldebugger in den Aktionsteil der Regel einzubauen.

Nachdem die Regel abgefeuert wurde, wird sie aus der Konfliktmenge des bearbeiteten Objekts gelöscht und die Inferenzmaschine beginnt einen neuen Durchlauf ihrer Kontrollschleife.

3.3.5 Ausblenden von Teilen des Regelsystems

Manchmal ist es sinnvoll, den Inferenzprozeß in Teilbereichen des Systems unterbinden zu können. Beispiele dafür sind u.a.

- Debugging,
- Ändern mehrerer Slotwerte, die zu einem als atomar zu behandelnden Fakt gehören (vgl. Abschnitt 2.2.2.2) sowie
- unterschiedliche Regeln beim Betrachten eines Objekts unter verschiedenen Aspekten.

Um diesen Ansprüchen Rechnung zu tragen, wurden folgende Ausblendemechanismen eingerichtet:

1. Stillegung des gesamten Regelsystems
2. Deaktivieren einer Kategorie
   (wirkt auf sämtliche Klassen, die diese Kategorie verwenden)
3. Verhindern der Inferenzauslösung für ein spezielles Objekt
4. Ausblenden einer Regelkategorie eines Objekts

Die dazu zu verwendenden Methoden und Funktionen werden im Anhang behandelt.

3.4 Benutzerunterstützung

Unter diese Sammelbezeichnung fallen Implementierungsdetails, die zur Funktion des Regelsystems nicht unbedingt nötig wären, jedoch den Umgang damit erleichtern.

3.4.1 Erklärungskomponente

Dieser Teil der benutzerunterstützenden Funktionen dient dazu, eine Begründung für die Aktionen zu geben, die das Regelsystem vornimmt bzw. vorgenommen hat. Bei der Vorwärtsverkettung ist es am sinnvollsten, wenn dies zur Laufzeit durch Ausgeben der Systemteile (Inferenzmaschinenstatus, Objekte, Regelkategorien, Regeln), die das Regelsystem gerade bearbeitet, geschieht. Dabei kann die Ausführlichkeit dieser Ausgaben in folgender Abstufung verändert werden:

Stufe 0: keine Ausgabe

Stufe 1: Ausgabe des Inferenzmaschinenstatus und der Objekte, die jeweils bearbeitet werden.

Stufe 2: zusätzlich Ausgabe der untersuchten Regelkategorien

Stufe 3: zusätzlich Ausgabe der untersuchten Regeln und der Slotzuweisungen, wenn die Regel feuert wird.

Stufe 4: zusätzlich Ausgabe von Konfliktlösungsinformationen

Diese Ausgaben können statt auf die Standardausgabe auch auf Dateien oder auf einen Drucker umgeleitet werden. Die Angaben über Ausführlichkeit und Ziel der Ausgabe werden in den Slots `text-trace-level` und `text-trace-stream` der Inferenzmaschine verwaltet, die Ausgabe selbst wird durch für die enerische Funktion `text-trace` geschriebene Methoden vorgenommen.

3.4.2 Vordefinierte Regelmodelle

Um dem Benutzer die Möglichkeit zu geben, auch ohne längere Beschäftigung mit der Arbeit von Konkliktlösungsmodellen mit dem Regelsystem arbeiten zu können, wurden drei solche Modelle bereits implementiert. Diese können auch als Anregung und Grundlage für eigene Entwicklungen dienen.

3.4.2.1 Simple–Weight

Dabei handelt es sich um ein primitives Modell, bei dem sowohl die Regelkategorien als auch die Regeln innerhalb der Kategorien statisch, d.h. zur Definitionszeit geordnet werden, und zwar nach dem Gewichtswert, der ihnen vom Benutzer gegeben wird. Dieses Modell eignet sich hauptsächlich für erste „Gehversuche" mit dem Regelsystem sowie für sehr einfache Anwendungen. Die Definition für dieses Modell ist:

```
(define-conflict-model simple-weight :forward
      (:rulecat-sorttime :at-definition-time)
      (:rulecat-weightfn weight)
      (:rule-sorttime :at-definition-time)
      (:rule-sortfn greater-weight)
      )
```

Die nötigen Werte für `:rulecat-supers` und `:rule-supers` werden vom System auf Grund der Funktionsangaben berechnet und hinzugefügt.

3.4.2.2 LEX

Dies ist eine Art der Konfliktlösung, wie sie ähnlich in OPS-5 vorkommt. Da es Regelkategorien in OPS-5 nicht gibt, wird hier angenommen, daß nur eine Kategorie verwendet wird, bzw. falls mehrere verwendet werden sollen, daß die Anwendung der LEX-Strategie bei allen diesen Kategorien stattfindet. Deshalb werden zuerst in allen Regelkategorien die Regeln geordnet und dann aus den besten Regeln aller Regelkategorien die Beste ermittelt und abgefeuert.

Die LEX-Strategie geht folgendermaßen vor: Zuerst werden alle Regeln gesucht, die triggern. Dann werden bei jeder Regel die Slotwerte, die in der Regelbedingung benützt werden, nach ihrem „Neuheitswert“ (recency) sortiert, d.h. je kürzer die Slotbelegung zurückliegt, desto höher ist der Neuheitswert. Schließlich werden die Regeln danach sortiert, welche die neuesten Slots enthält. Die Definition für das LEX–Modell ist:

```
(define-conflict-model lex :forward
      (:rulecat-sorttime :after-rule-ordering)
      (:rulecat-weightfn rule-recency)
      (:rule-sorttime :after-rule-testing)
      (:rule-sortfn greater-data-recency)
      )
```

3.4.2.3 Most–Likely

Dabei handelt es sich um ein heuristisches Verfahren, bei dem zunächst die Regelkategorieauswahl danach getroffen wird, in welcher Regelkategorie die meisten möglicherweise anwendbaren Regeln stehen. Bei der gewählten Regelkategorie werden dann die Regeln zuerst nach der Anzahl der Klauseln in der Bedingung sortiert und dann getestet, ob sie triggern. Das Most–Likely–Modell ist folgendermaßen festgelegt:

```
(define-conflict-model most-likely :forward
      (:rulecat-sorttime :before-rule-ordering)
      (:rulecat-weightfn no-of-rules)
      (:rule-sorttime :before-rule-testing)
      (:rule-sortfn more-if-clauses)
      )
```

3.4.3 Möglichkeiten für die Anbindung von obektorientierten Werkzeugen

Nahezu alle Datenstrukturen, die bei der Implementation dieses Systems verwendet werden, sind als Objekte realisiert worden. Dies bedeutet, daß sie mit den üblichen objektorientierten Werkzeugen (Inspektoren, Browser etc.) bearbeitet werden können.

Zusätzlich wäre es aber von großem Interesse, wenn man die Arbeit im Regelsystem selbst, also z.B. Definieren von Klassen, Kategorien und Konfliktlösungsmodellen, Ändern von Regeln u.v.a.m. mit Hilfe eines spezialisierten Regelsystembrowsers vornehmen könnte. Die Deklarations–Macros (`defrclass`, `defrulecat` etc.) sind jedoch für den direkten Aufruf durch den Benutzer entworfen worden und enthalten daher einen zu großen Overhead an Plausibilitätsprüfungen und zu erledigenden Operationen, als daß sie für die meist sehr speziellen Änderungen, die normalerweise innerhalb eines Browsers durchgeführt weden, sinnvoll eingesetzt werden könnten. Daher sollte beim Erstellen eines Browsers auf diejenigen Funktionen und Methoden zurückgegriffen werden, die innerhalb eines solchen Macros zur Durchführung der einzelnen Aufgaben verwendet werden.

Ebenso wie ein Regelsystembrowser ist auch ein Regeldebugger für die Entwicklung von Anwendungen, die das Regelsystem benützen, von großem Nutzen. Um an den relevanten Stellen in die Auswahl und die Bearbeitung von Regelkategorien und Regeln eingreifen zu können, ist es am sinnvollsten, zusätzliche Methoden für die in Abschnitt 3.4.1 erwähnte generische Funktiom `text-trace` zu benützen, da diese zu den betreffenden Zeitpunkten aufgerufen wird.

Teil 4

Ausblick

Das FrameTalk–Regelsystem ist in der hier beschriebenen Implementierung lauffähig. Es sind jedoch noch verschiedene Vervollständigungs– bzw. Verbesserungsmöglichkeiten denkbar.

Zunächst einmal sollte die Rückwärtsverkettungskomponente noch implementiert werden. Bei der Betrachtung der dort zu verwendenden Erklärungskomponente können auch weitere Anregungen für die des Forward Chaining gewonnen werden.

Des weiteren ist eine Erweiterung des Faktzugriffsformalismus möglich, indem Fakten nicht nur über Slots und Slotpfade sondern auch über benutzerdefinierbare Methoden ausgedrückt werden können. Dabei kann auch das Problem des Backreferencing bearbeitet werden.

Eine andere Möglichkeit bietet sich in der Behandlung von Constraints. Es ist zwar bisher schon möglich, mehrere Regeln so zu schreiben, daß ihre Zusammenarbeit constraintähnliche Wirkungen hat, ein Formalimus zur direkten Deklaration solcher Strukturen fehlt jedoch noch.

Außerdem wäre eine grafische Unterstützung für Programmierer und Benutzer des Regelsystems sehr sinnvoll. Browser und Debugger könnten an den in Abschnitt 3.4.3 beschriebenen Punkten installiert werden. Bei der Entwicklung solcher grafischer Werkzeuge sollte jedoch auf Kompatibilität mit den für das zugrundeliegende objektorientierte System entwickelten Programmen geachtet werden.

Literaturverzeichnis

[Bobrow et al. 86] D. Bobrow, K. Kahn u.a. CommonLoops - Merging Lisp and Object-Oriented Programming. In *Proceedings of OOPSLA'86 - Object-Oriented Programming Systems, Languages and Applications*, S. 17–29, Portland, Oregon, September 1986.

[Bobrow et al. 88] D. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales und D. A. Moon. Common Lisp Object System Specification. Technischer Bericht 88-002R, X3J13 Committee, June 1988.

[BobrowStefik 81] D. Bobrow und M. Stefik. The LOOPS Manual. Technischer Bericht KB-VLSI-81-83, Knowledge Systems Area, Xerox Palo Alto Research Center, 1981.

[Forgy 81] C. Forgy. OPS5 User's Manual. Technischer Bericht CS-81-135, Carnegie-Mellon University, 1981.

[Girgensohn 86] A. Girgensohn. Objektorientierte Produktionsregeln - ein Ansatz für ein hybrides System. Studienarbeit Nr. 595, Institut für Informatik, Universität Stuttgart auch als WISDOM-Forschungsbericht FB-INF-87-13 erschienen, 1986.

[GirgensohnRathke 88] A. Girgensohn und C. Rathke. ObjTalk - Version 14.3. INFORM Manual, Institut für Informatik, Universität Stuttgart, January 1988.

[GoldbergRobson 83] A. Goldberg und D. Robson. *SMALLTALK-80, The Language and its Implementation.* Addison-Wesley, Reading, Ma., 1983.

[KEE 85] IntelliCorp. *KEE Software Development System User's Manual*, 1985.

[Keene 89] S. E. Keene. *Object-Oriented Programming in Common Lisp – A Programmer's Guide to CLOS.* Addison Wesley Publishing Company, Reading, Ma., 1989.

[Rathke 86] C. Rathke. ObjTalk. Repräsentation von Wissen in einer objektorientierten Sprache. Dissertation, Fakultät Mathematik und Informatik der Universität Stuttgart, Oktober 1986.

[Rathke 87] C. Rathke. Framebasierte Wissensrepräsentation mit Hilfe objektorientierter Programmierung. In H. Balzert, G. Heyer und R. Lutze (Hrsg.), *Expertensysteme '87. Konzepte und Werkzeuge, Nürnberg, April 1987*, S. 64–87, B.G. Teubner, Stuttgart, 1987. German Chapter of the ACM, Berichte 28.

[Rathke 89] C. Rathke. Extending Object-Oriented Programming by Descriptions. In *IJCAI-89 Workshop on Object-Oriented Programming in Artificial Intelligence*, Detroit, August 1989.

[Steele Jr. 84] G. L. Steele Jr. *Common LISP: The Language.* Digital Press, Digital Equipment Corporation, 1984.

[StefikBobrow 86] M. Stefik und D. G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[TokoroIshikawa 87] M. Tokoro und Y. Ishikawa. An Object-Oriented Concurrent Programming Language for Knowledge Representation. In A. Yonezawa und M. Tokoro (Hrsg.), *Object-Oriented Concurrent Programming*, S. 159–198. MIT Press, 1987.

Anhang A

Funktionen der Benutzeroberfläche

In diesem Anhang werden diejenigen Makros und Funktionen beschrieben, die zur Benutzerschnittstelle gehören. Die Beschreibung umfaßt jeweils den Zweck, die Aufrufsyntax, nähere Informationen über die Argumente sowie den zurückgegebenen Wert. Falls nötig, werden auch weitere Bemerkungen und Verweise auf andere Funktionen angegeben.

Die aufgeführten Funktionen lassen sich in folgende Kategorien unterteilen:

1. Vereinbarung von Einheiten des Regelsystems:
   - `add-rule`
   - `defcategory`
   - `define-conflict-model`
   - `defrclass`
   - `defrule`
   - `defrulecat`
   - `get-rule`
   - `rem-rule`
2. Steuerung der Inferenzauslösung:
   - `activate-category`
   - `active-category-p`
   - `allow-inference`
   - `allowing-inference-p`
   - `deactivate-category`

- `defer-inference`
- `unuse-rulecat`
- `unuse-rulesystem`
- `use-rulecat`
- `use-rulesystem`
- `using-rulecat-p`
- `using-rulesystem-p`

3. Programmierung im Regelsystem:

    - `as-atomic-in`
    - `slot-path`
    - `slot-path-missing-p`
    - `slot-path-unbound-p`

activate-category *Generische Funktion*

Zweck:

Falls eine Kategorie deaktiviert wurde – entweder vom Benutzer oder durch Änderungen z.B. ihrer Regeln – kann sie mittels `activate-category` wieder aktiviert werden. Dadurch können die in ihr enthaltenen Regeln wieder von der Inferenzmaschine bearbeitet werden.

Syntax:

(`activate-category` *category-name*)

Argumente:

Das Argument *category-name* ist der Name einer Kategorie.

Ergebnis:

Als Ergebnis wird *t* zurückgegeben.

Verweise:

`active-category-p`, `deactivate-category`

active-category-p *Generische Funktion*

Zweck:

Mittels `activate-category` kann erfragt werden, ob eine Kategorieklasse aktiviert ist, d.h., ob die in ihr enthaltenen Regeln von der Inferenzmaschine bearbeitet werden können.

Syntax:

(`active-category-p` *category-name*)

Argumente:

Das Argument *category-name* ist der Name einer Kategorie.

Ergebnis:

Als Ergebnis wird *t* zurückgegeben, wenn die Kategorieklasse aktiviert ist, sonst *nil*.

Verweise:

`activate-category, deactivate-category`

add-rule *Generische Funktion*

Zweck:

Mit dieser Funktion können Regeln zu einer Regelkategorie hinzugefügt werden.

Syntax:

(`add-rule` *class rulecat-name rule*)

Argumente:

Das Argument *rule* ist eine Regel, d.h. eine Instanz der Klasse `rule` oder einer ihrer Unterklassen. *rulecat-name* ist der Name einer Regelkategorie in der Klasse, die durch *class* angegeben wurde. *class* kann dabei entweder ein Klassenobjekt oder der Name eines Klassenobjekts sein.

Ergebnis:

Zurückgegeben wird die Regel.

Verweise:

`get-rule, rem-rule`

allow-inference *Generische Funktion*

Zweck:

Mit dieser Funktion läßt ein Objekt wieder zu, daß es von der Inferenzmaschine bearbeitet wird (Standardzustand). Notwendig ist dies, wenn vorher die Inferenz durch `defer-inference` verhindert wurde. Sind für das betreffende Objekt noch unbearbeitete Slotzugriffe gespeichert, so wird es sofort in die Warteliste der Inferenzmaschine eingefügt.

Syntax:

(`allow-inference` *object*)

Argumente:

Das bezeichnete Objekt muß Instanz einer mittels `defrclass` vereinbarten Klasse sein.

Ergebnis:

Ergebnis des Funktionsaufrufs ist *t*.

Verweise:

`allowing-inference-p, defer-inference`

allowing-inference-p *Generische Funktion*

Zweck:

Mit dieser Funktion kann festgestellt werden, ob ein Objekt von der Inferenzmaschine bearbeitet werden kann.

Syntax:

(`allowing-inference-p` *object*)

Argumente:

Das bezeichnete Objekt muß Instanz einer mittels `defrclass` vereinbarten Klasse sein.

Ergebnis:

Ergebnis des Funktionsaufrufs ist *t*, falls das Objekt von der Inferenzmaschine bearbeitet werden kann, sonst *nil*.

Verweise:

`allow-inference`, `defer-inference`

as-atomic-in *Makro*

Zweck:

Dieses Makro dient dazu, eine Reihe von Slotänderungen an einem oder mehreren Objekten vornehmen zu können, ohne daß dabei ein Inferenzprozeß ausgelöst werden kann.

Syntax:

(`as-atomic-in` ({*object*}*) `&body` *body*)

Argumente:

Diejenigen Objekte, bei denen während der Durchführung der Aktionen keine Inferenz stattfinden darf, sind als *object* anzugeben. Der Lisp–Ausdruck *body* wird evaluiert.

Ergebnis:

Zurückgegeben wird das Ergebnis der Evaluation von *body*.

Bemerkung:

Tritt während der Auswertung von *body* ein Fehler auf, so wird die Inferenzauslösung für die angegebenen Objekte erst wieder zugelassen, wenn zum Top Level zurückgekehrt wird.

Verweise:

`allow-inference, allowing-inference-p, defer-inference`

deactivate-category *Generische Funktion*

Zweck:

Durch Aufruf von `deactivate-category` wird eine Kategorie deaktiviert, d.h. die in ihren Instanzen enthaltenen Regeln in allen Objektklasse können nicht mehr bearbeitet werden.

Syntax:

(`deactivate-category` *category-name*)

Argumente:

category-name ist der Name einer Kategorieklasse.

Ergebnis:

Das Ergebnis des Aufrufs ist *nil*.

Verweise:

`activate-category, active-category-p`

defcategory *Makro*

Zweck:

Definition einer Kategorieklasse

Syntax:

(`defcategory` *category-name* ({*superclass-name*}*) ⟦↓*category-option*⟧)

category-option ::= (`:documentation` *string*) |
(`:conflict-model` *conflict-model-name*)

Ergebnis:

Zurückgegeben wird die neue Kategorieklasse.

Verweise:

Abschnitt 3.2.2

defer-inference *Generische Funktion*

Zweck:
Mit dieser Funktion kann der Inferenzprozeß für ein bestimmtes Objekt unterbunden werden.

Syntax:
(`defer-inference` *object*)

Argumente:
Das bezeichnete Objekt muß Instanz einer mittels `defrclass` vereinbarten Klasse sein.

Ergebnis:
Als Ergebnis wird *nil* zurückgegeben.

Verweise:
`allow-inference`, `allowing-inference-p`

define-conflict-model *Makro*

Zweck:

Mit diesem Makro können Konfliktlösungsmodelle definiert werden.

Syntax:

(`define-conflict-model` *name direction* ⟦↓*option*⟧)

option ::= (`:documentation` *string*) |
(`:rulecat-sorttime` *rulecat-sorttime-option*) |
(`:rulecat-weightfn` *function*) |
(`:rulecat-supers` *superclass-list*) |
(`:rule-sorttime` *rule-sorttime-option*) |
(`:rule-sortfn` *function*) |
(`:rule-supers` *superclass-list*)

rulecat-sorttime-option ::= `:at-definition-time` |
`:before-rule-ordering` |
`:after-rule-ordering`

rule-sorttime-option ::= `:at-definition-time` |
`:before-rule-testing` |
`:after-rule-testing`

Ergebnis:

Ergebnis des Aufrufs ist das neue Konfliktlösungsmodell.

Bemerkung:

Defaultwerte für die Optionen sind:

`:rulecat-sorttime`	`:at-definition-time`
`:rulecat-weightfn`	`weight`
`:rulecat-supers`	geht aus `:rulecat-weightfn` hervor.
`:rule-sorttime`	`:at-definition-time`
`:rule-sortfn`	`weight`
`:rule-supers`	geht aus `:rule-sortfn` hervor.

Verweise:

Abschnitt 3.2.3

defrclass *Makro*

Zweck:

Klassen, deren Objekte eine Wissensbasis besitzen sollen, müssen mit `defrclass` vereinbart werden.

Syntax:

(`defrclass` *class-name* ({*superclass-name*}*)
({*slot-specifier*}*)
⟦↓*class-option*⟧)

slot-specifier ::= *slot-name* | (*slot-name* ⟦↓*slot-option*⟧)

Ergebnis:

Zurückgegeben wird das neue Klassenobjekt.

Bemerkung:

Die Vereinbarung einer Klasse mittels `defrclass` entspricht weitgehend derjenigen, wie sie in CLOS durch `defclass` geschieht. Neue Optionen sind:

bei slot-option:

`:leave-out` *boolean*, um einen Slot als nicht zur Datenbasis des Objekts gehörig zu bezeichnen

`:all-writes` *boolean*, um festzulegen, ob bei allen Schreibzugriffen auf Slots die Inferenzmaschine gestartet wird oder nur bei Änderungen der dort enthaltenen Werte

bei class-option:

`:conflict-models` *conflict-model-list*, um die möglichen Konfliktlösungsmodelle anzugeben (s.a. Abschnitt 3.2.3)

Verweise:

Abschnitt 3.2.1 sowie [Bobrow et al. 88], pp. 2.20–2.25.

defrule

Makro

Zweck:

Mit diesem Makro können neue Regeln inner– und außerhalb von Regelkategorien definiert werden.

Syntax:

(`defrule` *rule-name* ({*rule-class*}) ({*rule-allocation*}) *rule-body* ⟦↓ *rule-option*⟧)

rule-allocation ::= *rulecat-name class-name*

rule-body ::= (({*if-clause*}*) ({*then-clause*}*) ({*side-effect*}*))

rule-option ::= (`:documentation` *string*) |
(*initargs*)

Ergebnis:

Als Ergebnis wird die neudefinierte Regel zurückgegeben.

Bemerkung:

Falls mit *rule-allocation* die Regelkategorie und die Klasse angegeben werden, in der die Regel stehen soll, so wird automatisch die betreffende Kategorieklasse deaktiviert. Nur falls ihr slot `auto-activate` einen von *nil* verschiedenen Wert hat, wird die Vererbung der neueingetragenen Regel durchgeführt und die Kategorieklasse wieder aktiviert. Ansonsten muß dies durch Aufruf von `activate-category` veranlaßt werden.

Verweise:

`activate-category` sowie Abschnitt 3.2.2

defrulecat *Makro*

Zweck:

Definieren einer Regelkategorie bei einer Klasse, es ist auch möglich, gleich Regeln mit anzugeben.

Syntax:

(`defrulecat` *rulecat-name* ({*superclass-name*}*) *object-class*
(*rule-specifier**) ⟦↓*rulecat-option*⟧)

rulecat-option ::= (`:documentation` *string*) |
(`:conflict-model` *conflict-model-name*)
(*initargs*)

rule-specifier ::= (*rule-name rule-body* ⟦↓ *rule-option*⟧)

rule-body ::= (({*if-clause*}*) ({*then-clause*}*) ({*side-effect*}*))

rule-option ::= (`:documentation` *string*) |
(*initargs*)

Ergebnis:

Das Ergebnis des Aufrufs von `defrulecat` ist die definierte Regelkategorie.

Bemerkung:

1. Existiert noch keine Kategorieklasse mit dem Namen *rulecat-name*, so wird diese erzeugt. Nur in diesem Fall sind die Angaben der Superklassen sowie des Konfliktlösungsmodells erlaubt.
2. Der Aufruf von `defrulecat` deaktiviert die gesamte (neu erzeugte bzw. schon bestehende) Kategorieklasse. Nur falls ihr slot `auto-activate` einen von *nil* verschiedenen Wert hat, wird die Vererbung von evtl. mitdefinierten Regeln durchgeführt und die Kategorieklasse wieder aktiviert. Ansonsten muß dies durch Aufruf von `activate-category` veranlaßt werden.

Verweise:

`activate-category` sowie Abschnitt 3.2.2

get-rule

Generische Funktion

Zweck:

Mit `get-rule` kann aus einer Regelkategorie eine namentlich angegebene Regel ausgelesen werden.

Syntax:

(`get-rule` *class rulecat-name rule-name*)

Argumente:

Die Argumente *rule-name* und *rulecat-name* sind Symbole, die die Namen der entsprechenden Regel bzw. Regelkategorie angeben. *class* kann sowohl ein Klassenobjekt als auch der Name eines solchen sein.

Ergebnis:

Zurückgegeben wird die in der Regelkategorie *rulecat-name* der Klasse *class* unter dem Namen *rule-name* abgelegte Regel;
bzw. *nil*, falls eine Regel mit diesem Namen nicht existiert.

Bemerkung:

Mit `get-rule` werden nur die `direct-rules` der Regelkategorie erreicht.

Verweise:

`add-rule, rem-rule`

rem-rule

Generische Funktion

Zweck:

Der Aufruf von `rem-rule` kann eine Regel aus einer Regelkategorie entfernen.

Syntax:

(`rem-rule` *class rulecat-name rule-name*)

Argumente:

Die Argumente *rule-name* und *rulecat-name* sind Symbole, die die Namen der entsprechenden Regel bzw. Regelkategorie angeben. *class* kann sowohl ein Klassenobjekt als auch der Name eines solchen sein.

Ergebnis:

Zurückgegeben wird *t*, falls in der Regelkategorie *rulecat-name* der Klasse *class* eine unter dem Namen *rule-name* abgelegte Regel entfernt wurde;
bzw. *nil*, falls keine Regel mit diesem Namen existierte.

Bemerkung:

Mit `rem-rule` können nur `direct-rules` der Regelkategorie entfernt werden.

Verweise:

`add-rule, get-rule`

slot-path *Funktion*

Zweck:

Die Funktion `slot-path` dient innerhalb von Regel dazu, auf Slots des gerade bearbeiteten Objekts oder von über Slotpfaden erreichbaren Objekten zuzugreifen.

Syntax:

(`slot-path` *path*)

Argumente:

path kann entweder der Name eines Slots des gerade bearbeiteten Objekts oder eine Liste von Slotnamen sein.

Ergebnis:

Das Ergebnis von `slot-path` ist der Wert des bezeichneten Slots.

Bemerkung:

1. Statt (`slot-path` *path*) kann auch #>*path* benützt werden.
2. Wird beim Absuchen des Slotpfades in der Bedingung einer Regel ein Slot nicht gefunden, so ergeht keine Fehlermeldung. Statt dessen gilt die Teilbedingung, in der der Fehler auftrat, als nicht erfüllt.

Verweise:

`slot-path-unbound-p, slot-path-missing-p`

slot-path-missing-p *Funktion*

Zweck:

Kann in einer Regel verwendet werden, um festzustellen, ob ein Slot des gerade betrachteten oder eines über Slotpfade erreichbaren Objekts nicht vorhanden ist.

Syntax:

(`slot-path-missing-p` *path-description*)

Argumente:

path-description muß von der Form (`slot-path` *path*) sein.

Ergebnis:

Das Ergebnis ist *t*, falls der durch *path-description* angegebene Slot fehlt, sonst *nil*.

Verweise:

`slot-path`, `slot-path-unbound-p`

slot-path-unbound-p *Funktion*

Zweck:

Kann in einer Regel verwendet werden, um festzustellen, ob ein Slot des gerade betrachteten oder eines über Slotpfade erreichbaren Objekts noch ungebunden ist.

Syntax:

(`slot-path-unbound-p` *path-description*)

Argumente:

path-description muß von der Form (`slot-path` *path*) sein.

Ergebnis:

Das Ergebnis ist *t*, falls der durch *path-description* angegebene Slot ungebunden ist, sonst *nil*.

Verweise:

`slot-path`, `slot-path-missing-p`

unuse-rulecat *Generische Funktion*

Zweck:

Diese generische Funktion dient dazu, innerhalb eines Objekts die Anwendung von Regeln einer Regelkategorie zu verhindern.

Syntax:

(`unuse-rulecat` *object rulecat-name*)

Argumente:

object muß Instanz einer durch `defrclass` definierten Klasse sein. *rulecat-name* ist der Name einer bei dieser Klasse festgelegten Regelkategorie.

Ergebnis:

Der Aufruf von `unuse-rulecat` liefert *nil* zurück.

Bemerkung:

Der Aufruf dieser Funktion hat keine Auswirkung auf den Aktivierungszustand der Kategorieklasse, der mittels `activate-category` bzw. `deactivate-category` beeinflußt wurde.

Verweise:

`use-rulecat, using-rulecat-p`

unuse-rulesystem *Funktion*

Zweck:
Dient dazu, das gesamte Regelsystem stillzulegen, d.h. die Arbeit aller im System vorhandenen Inferenzmaschinen zu verhindern.

Syntax:
(`unuse-rulesystem`)

Argumente:
keine

Ergebnis:
nil

Verweise:
`use-rulesystem`, `using-rulesystem-p`

use-rulecat *Generische Funktion*

Zweck:

Diese generische Funktion dient dazu, innerhalb eines Objekts die Anwendung von Regeln einer Regelkategorie (wieder) zu gestatten.

Syntax:

(`use-rulecat` *object rulecat-name*)

Argumente:

object muß Instanz einer durch `defrclass` definierten Klasse sein. *rulecat-name* ist der Name einer bei dieser Klasse festgelegten Regelkategorie.

Ergebnis:

Der Aufruf von `use-rulecat` liefert *t* zurück.

Bemerkung:

Der Aufruf dieser Funktion hat keine Auswirkung auf den Aktivierungszustand der Kategorieklasse, der mittels `activate-category` bzw. `deactivate-category` beeinflußt wurde.

Verweise:

`unuse-rulecat`, `using-rulecat-p`

use-rulesystem *Funktion*

Zweck:
Dient dazu, das Regelsystem wieder in Gang zu setzen.

Syntax:
(`use-rulesystem`)

Argumente:
keine

Ergebnis:
t

Verweise:
`unuse-rulesystem, using-rulesystem-p`

using-rulecat-p *Generische Funktion*

Zweck:

Diese generische Funktion dient dazu, festzustellen, ob innerhalb eines Objekts die Anwendung von Regeln einer Regelkategorie gestattet ist.

Syntax:

(`using-rulecat-p` *object rulecat-name*)

Argumente:

object muß Instanz einer durch `defrclass` definierten Klasse sein. *rulecat-name* ist der Name einer bei dieser Klasse festgelegten Regelkategorie.

Ergebnis:

Der Aufruf von `using-rulecat-p` liefert *t* zurück, falls die Anwendung von Regeln derer durch *rulecat-name* angegebenen Regelkategorie innerhalb des Objekts *object* gestattet ist, sonst *nil*.

Bemerkung:

Diese Funktion gibt keine Information über den Aktivierungszustand der Kategorieklasse, der mittels `activate-category` bzw. `deactivate-category` beeinflußt wurde, zurück.

Verweise:

`unuse-rulecat, use-rulecat`

using-rulesystem-p *Funktion*

Zweck:
Dient zur Feststellung, ob das Regelsystem benützt wird.

Syntax:
(`using-rulesystem-p`)

Argumente:
keine

Ergebnis:
t, falls das Regelsystem benützt wird, sonst *nil*.

Verweise:
`unuse-rulesystem, using-rulesystem-p`