*Diplomarbeit*

# Konzeption und Implementierung einer Komponente zur regelbasierten Auswahl von Konfigurationsschritten

Werner Schirp

April 1995

*Betreuer:*

Prof. Dr. Michael M. Richter

Dipl. Inform. Jürgen Paulokat

AG Künstliche Intelligenz — Expertensysteme

UNIVERSITÄT KAISERSLAUTERN

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Die Lösung einer Konfigurationsaufgabe innerhalb einer technischen Domäne besteht in der Konstruktion eines komplexen Objektes, das sowohl alle in der Aufgabe gestellten Anforderungen bezüglich seiner Funktionalität erfüllt, als auch den innerhalb der Domäne vorhandenen Restriktionen vollständig genügt. Durch die als bekannt vorausgesetzte Struktur der Anwendungsdomäne wird ein Suchraum aufgespannt, in dem es eine in diesem Sinne korrekte — und wenn möglich besonders gute — Lösung zu finden gilt. Wegen der Größe des Suchraums wird zu diesem Zweck im allgemeinen ein Verfahren zur Tiefensuche eingesetzt.

Dieses Suchverfahren erzwingt in jedem Verzweigungspunkt eine Festlegung auf eine von mehreren möglichen Alternativen, obwohl für diese Entscheidung in der Regel kein ausreichendes Wissen vorhanden ist. Deswegen kann eine solche Auswahlentscheidung grundsätzlich nur vorläufiger Natur sein und muß später eventuell revidiert werden, was mit erheblichem Aufwand verbunden sein kann. Um diesen Aufwand zu reduzieren, sollte bei der Auswahl einer Alternative möglichst viel vorhandenes Wissen eingesetzt werden, damit immer diejenige Entscheidung getroffen werden kann, die einen besonders schnellen Erfolg des Suchprozesses oder eine besonders gute Lösung verspricht.

Ziel der vorliegenden Arbeit ist es, im Rahmen einer bestehenden Umgebung zur Entwicklung von Expertensystemen für Konfigurationsanwendungen in technischen Domänen eine Möglichkeit zu schaffen, das zur Auswahl von Konfigurationsschritten vorhandene Expertenwissen in Form von Präferenzregeln in den Suchprozeß einzubringen. Dabei soll berücksichtigt werden, daß in die Bevorzugung einer Alternative oft das Wissen um bereits getroffene Entscheidungen eingeht. Eine spätere Revision dieser Entscheidungen muß dann auch zu einer Überprüfung der auf sie gestützten Präferenzen führen. Besonderer Wert wird außerdem darauf gelegt, Entscheidungen eines menschlichen Experten oder Anwenders im Verlaufe einer Konfiguration homogen in den Suchprozeß einzubinden und das dabei eingesetzte Domänenwissen explizit zu repräsentieren.

Kenntnisse im Bereich der technischen Konfiguration mit Expertensystemen sind zum Verständnis dieser Arbeit selbstverständlich hilfreich, die wesentlichen Grundlagen werden aber in den Kapiteln 2 und 3 in kurzer Form zusammengefaßt. Der Kern der realisierten Komponente zu Verarbeitung von Präferenzregeln wird in den Kapiteln 4 und 5 beschrieben, spezielle Aspekte werden in den Kapiteln 6 und 7 ausgeführt. Gewisse Vorkenntnisse des Lesers über „Truth–Maintenance–Systeme“, wie sie von Jon Doyle in [Doy79] eingeführten wurden, sind dabei ebenso unverzichtbar wie rudimentäre Kenntnisse der Programmiersprache „Prolog“, die hier ohne weitere Erläuterung verwendet wird, da sie in zahlreichen Büchern, wie z.B. [CM84], ausführlich beschrieben ist.

# Kapitel 2

# Konfiguration in technischen Domänen

## 2.1 Definition des Konfigurationsproblems

Eine eigenständige, allgemein anerkannte Definition des Begriffs „Konfiguration“ hat sich in der Literatur bislang nicht herausgebildet. Ausgehend von der grundlegenden Aufteilung des Anwendungsbereiches von Expertensystemen in Analyse- und Syntheseaufgaben läßt sich die Konfiguration aber eindeutig in den Bereich der Synthese einordnen. Statt des Begriffs der Synthese wird in der deutschen Literatur oft auch synonym die Bezeichnung „Konstruktion“ benutzt.

Innerhalb des Bereichs der Konstruktion wird dann vor allem zwischen Planen und Konfigurieren unterschieden. Puppe grenzt in [Pup90] sogar drei Problemtypen voneinander ab:

- *Die* **Konfigurierung** *umfaßt Probleme, bei denen verfügbare Basiselemente ausgewählt, parametrisiert und zu einem Lösungsobjekt zusammengesetzt werden, das gewünschte Eigenschaften erfüllt.*
- *Bei der* **Zuordnung** *wird eine Menge von Objekten unter Berücksichtigung von Rahmenbedingungen auf eine andere Menge von Objekten abgebildet.*
- *Bei der* **Planung** *wird eine Sequenz von Operatoren gesucht, die einen gegebenen Ausgangszustand in einen gewünschten Zielzustand transformieren.*

Chandrasekaran liefert in [Cha90] eine detaillierte Definition für den englischen Begriff „Design Task“, der zwar allgemein als „Konstruktionsaufgabe“ übersetzt werden muß, in der betreffenden Quelle aber soweit eingeschränkt ist, daß er sehr genau einer „Konfigurationsaufgabe“ im hier gewünschten Sinne entspricht:

> *Eine Konfigurationsaufgabe ist festgelegt durch*
>
> 1. *eine Menge von funktionellen Eigenschaften (sowohl solche, die ausdrücklich vom Benutzer angegeben werden, als auch diejenigen, die implizit in der Anwendungsdomäne festgelegt sind), welche die Konfiguration besitzen soll, und eine Menge von Rahmenbedingungen, die erfüllt sein müssen, und*

2. *eine Technologie, d.h. ein Repertoire von Komponenten, die als verfügbar angesehen werden, und eine Beschreibung der Beziehungen zwischen diesen Komponenten.*

*Die Rahmenbedingungen können sich dabei auf die Parameter der Konfiguration selbst, auf das Herstellungsverfahren oder auch auf den Konfigurationsprozeß erstrecken. Die Lösung der Konfigurationsaufgabe besteht aus einer vollständigen Festlegung einer Menge von Komponenten und deren Beziehungen, die gemeinsam eine Konfiguration darstellen, welche die funktionellen Eigenschaften besitzt und die Rahmenbedingungen erfüllt. Von dieser Lösung erwartet man, daß sie zusätzlich eine Menge impliziter Anforderungen erfüllt; z.B. soll sie nicht wesentlich komplizierter oder teurer sein als mögliche Alternativen.*

Aus dieser Definition lassen sich drei wesentliche Aspekte einer Konfigurationsaufgabe zusammenfassend extrahieren:

- Die Menge der verfügbaren Komponenten spannt zusammen mit den zwischen diesen Komponenten erlaubten Beziehungen einen **Suchraum** auf, der alle im kombinatorischen Sinne möglichen Konfigurationen enthält.
- Innerhalb des Suchraumes definieren die in der Domäne geltenden Randbedingungen und die geforderten funktionellen Eigenschaften der Konfiguration den **Lösungsraum** einer Konfigurationsaufgabe.
- Es können implizite Anforderungen existieren, die von verschiedenen Elementen des Lösungsraumes mehr oder weniger gut erfüllt werden, weswegen eine **Optimierung** bezüglich einer oder mehrerer dieser Anforderungen erwünscht sein kann.

Formal läßt sich ein Konfigurationsproblem der soeben beschriebenen Art wie jedes andere Suchproblem im Prinzip durch systematische Tiefen- oder Breitensuche lösen. Ausgehend von der für Konstruktionsaufgaben typischen Annahme, daß der Lösungsraum verglichen mit dem Suchraum verschwindend klein ist, ist das wesentliche Bestreben bei der Entwicklung eines Konfigurationssystems, durch intensiven Einsatz von Domänenwissen den Suchraum insgesamt zu verkleinern und den Suchprozeß zusätzlich geschickt zu steuern.

## 2.2 Konfigurierung durch schrittweise Top–Down–Verfeinerung

Der Begriff der Konfiguration läßt sich im vorliegenden Zusammenhang weiter einschränken auf den Bereich der Routine–Konfiguration, was bedeutet, daß Methoden des innovativen oder kreativen Konfigurierens im folgenden nicht betrachtet werden. Eine Anwendungsdomäne, die in den Bereich des „Routine Design“ fällt, zeichnet sich nach [BC89] durch die drei folgenden Eigenschaften aus:

- Für jedes Konfigurationsproblem ist eine effektive Zerlegung in einfachere Teilprobleme bekannt.
- Für jedes Teilproblem sind vorgegebene Lösungsstrategien bekannt.
- Das Vorgehen für den Fall, daß die Lösung eines Teilproblems scheitert, ist explizit festgelegt.

Die Beschränkung auf diese Art von Konfigurationsproblemen erlaubt es, den Suchraum mit der bekannten Methode der schrittweisen Top–Down–Verfeinerung zu explorieren. Der Suchraum wird dabei durch einen gerichteten Und–Oder–Graphen repräsentiert, in dem als Lösung ein Hyperpfad, ausgehend vom Startknoten, der die gesamte Konfigurationsaufgabe darstellt, gesucht wird, der (mindestens!) einen terminalen Knoten enthält ([Ric92]). Die Lösung einer Konfigurationaufgabe mit Hilfe eines solchen Und–Oder–Graphen setzt voraus, daß

- für jede Konfigurationsaufgabe eine bekannte Menge alternativer Konfigurationsschritte existiert, aus der genau einer ausgewählt werden muß (Oder–Knoten).
- der ausgewählte Schritt die Konfigurationsaufgabe entweder löst (terminaler Knoten) oder in eine endliche Anzahl von Teilaufgaben zerlegt (Und–Knoten).
- dieser Prozeß rekursiv fortgesetzt werden kann, bis alle Teilaufgaben gelöst sind.

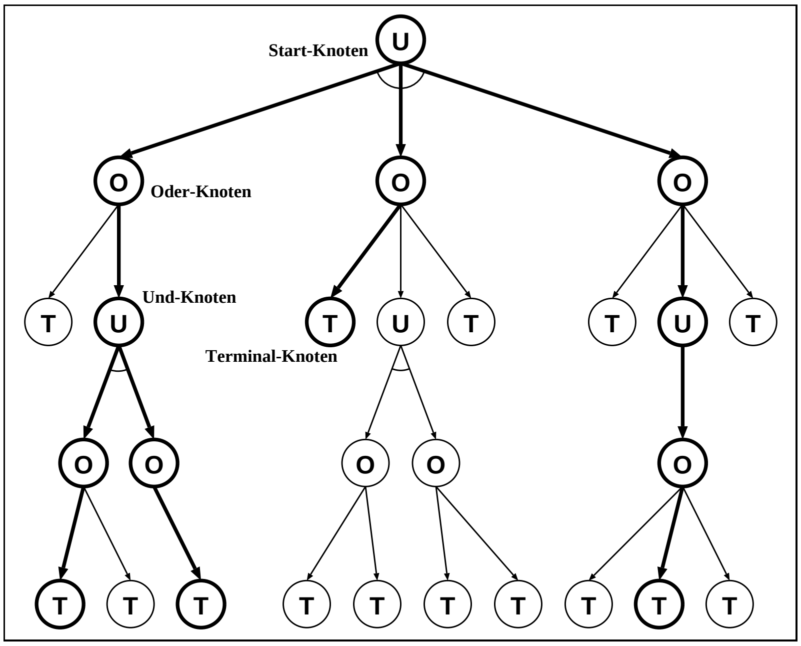


Abbildung 2.1: Und–Oder–Graph mit einem möglichen Hyperpfad (fett gedruckt)

Innerhalb eines Konfigurationsprozesses sind bei schrittweiser Top–Down–Verfeinerung abwechselnd zwei Arten von Auswahl–Entscheidungen zu treffen. Als erstes muß die zu bearbeitende Konfigurationsaufgabe gewählt werden, und als zweites muß für diese Aufgabe ein geeigneter Konfigurationsschritt gewählt werden.

Die Auswahl einer Konfigurationsaufgabe ist eine reine Reihenfolge–Entscheidung, da alle bekannten Teilaufgabe früher oder später bearbeitet werden müssen. Die Auswahl der zu bearbeitenden Teilaufgabe betrifft also nicht die Konfiguration an sich, sondern nur den Konfigurationsprozeß und stellt somit im Sinne von [CGS91] eine *Kontrollentscheidung* dar. Obwohl eine Kontrollentscheidung sich nicht unmittelbar auf das Konfigurationsobjekt auswirkt, kann sie dennoch wesentlichen Einfluß nicht nur auf die Effizienz des Konfigurationsprozesses haben, sondern auch auf dessen Ergebnis. Wenn es nämlich mehrere Lösungen gibt, kann es unter anderem von den Kontrollentscheidungen abhängen, welche dieser Lösungen zuerst gefunden wird.

Die Auswahl eines Konfigurationsschrittes hat im Gegensatz dazu direkte Auswirkungen auf das Konfigurationsobjekt und ist somit eine *Konstruktionsentscheidung*. Von diesen Entscheidungen hängt es ab, ob überhaupt eine, und wenn ja, ob eine gute bzw. optimale Lösung gefunden wird. Da bei der Wahl eines Konfigurationsschrittes in aller Regel nur unvollständiges Wissen zur Verfügung steht, kann es später erforderlich oder opportun sein, die bereits erfolgte Festlegung von Konfigurationsentscheidungen zu revidieren. Wegen des damit verbundenen Suchaufwandes ist es erstrebenswert, solche Revisionen so weit als möglich zu vermeiden, indem man den Wissenseinsatz bei der Auswahlentscheidung intensiviert. Um dieses Ziel zu erfüllen, wird in der vorliegenden Arbeit eine Komponente vorgestellt, die die Auswahl von Konfigurationsschritten durch Präferenzwissen steuert.

## 2.3 Präferenzwissen zur Auswahl von Konfigurationsschritten

Wenn ein menschlicher Experte eine Konstruktionsentscheidung treffen muß, für die er nicht mit Sicherheit eine einzige, korrekte Lösung angeben kann, so wird er üblicherweise nicht versuchen, alle möglichen Alternativen systematisch durchzuprobieren, sondern gezielt eine Auswahl treffen. In diese Auswahlentscheidung geht ein Teil seines Expertenwissens ein, der über die Kenntnis der Randbedingungen der Anwendungsdomäne und der funktionellen Anforderungen an das Konstruktionsobjekt hinausgeht. Zu diesem Teil des Wissens gehören beispielsweise

- Erfahrungswerte aus dem Erfolg oder Mißerfolg früherer Konstruktionsprozesse
- Kenntnisse über besondere Vor- oder Nachteile einzelner Konstruktionsschritte
- Vermutungen über implizite Abhängigkeiten in der Domäne
- „Gefühlsmäßige“, d.h. nicht erklärbare, Bevorzugung bestimmter Konstruktionsschritte

Dieses Wissen läßt sich unter dem Begriff *Präferenzen* zusammenfassen, weil es gemeinsam mit der Kenntnis der zu lösenden Konfigurationsaufgabe und der bereits erstellten Teillösung dazu führt, daß genau ein Konstruktionsschritt gegenüber allen anderen Alternativen bevorzugt wird.

Damit die Präferenzen eines Experten auch einem rechnergestützten Konfigurationssystem zugänglich werden, müssen diese Präferenzen sowohl explizit gemacht, als auch in geeigneter Form formalisiert werden. Präferenzen sind weniger leicht explizit zu formulieren als andere Teile des Domänenwissens, da sie nicht nur auf physikalischen Gegebenheiten, Vorschriften o.ä. basieren, sondern zum Teil auf den sehr persönlichen Einschätzungen eines Experten beruhen, die dieser oftmals sogar unbewußt in seine Entscheidungen einfließen läßt. Eine Möglichkeit, Präferenzen eines Experten explizit zu machen, besteht darin, ihn eine Vielzahl von Konfigurationsentscheidungen treffen zu lassen und dabei immer nach einer Begründung für die jeweilige Auswahl zu fragen. Aus diesen Begründungen, die leider oft zu speziell oder unvollständig sind, lassen sich dann Präferenzen ableiten (siehe dazu Abschnitt 7.2).

Oft werden in Begründungen oder Präferenzen Voraussetzungen genannt, die zu der jeweiligen Entscheidung geführt haben. Für die Formalisierung der Präferenzen bieten sich daher Regeln der Form „Wenn ..., dann ...“ an, mit denen die Auswahl eines bestimmten Konfigurationsschrittes („dann“–Teil) an gewisse Bedingungen („Wenn“–Teil) geknüpft wird. Dabei ist zu berücksichtigen, daß es auch Präferenzen geben kann, die nicht an Vorbedingungen geknüpft sind (siehe dazu Abschnitt 4.2). Die Verwendung von Regeln bietet den Vorteil, daß sie einerseits für den Menschen eine natürliche Form des Ausdrucks darstellen und andererseits logische Inferenzen erlauben.

Die in Form von Regeln vorliegenden Präferenzen kann ein rechnergestütztes Konfigurationssystem nun zur Festlegung einer Konstruktionsentscheidung einsetzen. Dazu muß aus der Menge der für die betreffende Entscheidung relevanten Regeln genau eine ermittelt werden, deren Bedingungen erfüllt sind. Dieser Vorgang kann sehr aufwendig werden, wenn viele Regeln mit komplizierten Bedingungen überprüft werden müssen. Unter der Annahme, daß die formulierten Präferenzen sinnvoll sind, wird dieser zusätzliche Aufwand aber dadurch gerechtfertigt, daß der präferierte Konfigurationsschritt mit großer Wahrscheinlichkeit zu einer schnelleren oder besseren Lösung führt, als ein zufällig ausgewählter (siehe dazu Abschnitt 4.1). Auf jeden Fall wird durch die Einbeziehung von Präferenzwissen aber erreicht, daß sich der rechnergestützte Konfigurationsprozeß stärker am Vorgehen eines menschlichen Experten orientiert.

# Kapitel 3

# Das Konfigurationssystem IDAX

Das Konfigurationssystem IDAX ([Pau95]) ist ein an der Universität Kaiserslautern entwickeltes Werkzeug zur interaktiven Lösung von Konfigurationsaufgaben in technischen Domänen. Sein Name steht als Abkürzung für „Intelligent Design Assistant based on Redux“. Der Konfigurationsprozeß in IDAX folgt dem Prinzip der Begriffshierarchie–orientierten Kontrolle ([CGS91]). Die Verwaltung von Konfigurationsentscheidungen und ihren Begründungen basiert auf dem problemklassenspezifischen Truth–Maintenance–System REDUX ([Pet91]). Das IDAX–System ist prototypisch in der objektorientierten Programmiersprache SMALLTALK–80 ([GR89]) implementiert.

## 3.1 Wissensrepräsentation in IDAX

### 3.1.1 Statisches Wissen

Das Wissen über eine technische Domäne, also über die vorhandenen Komponenten und Herstellungsverfahren, sowie das domänenspezifische Problemlösungswissen sind in der statischen Wissensbasis von IDAX ([Kau94]) repräsentiert, weil dieses Wissen nur selten modifiziert wird und daher bei der Bearbeitung einer einzelnen Konfigurationsaufgabe als unveränderlich betrachtet werden kann. Im Mittelpunkt der Wissensmodellierung stehen die Domänenobjekte, als welche nicht nur konkrete Objekte der Anwendungsdomäne, sondern auch abstrakte (Ober-)Begriffe bezeichnet werden, die funktionelle Eigenschaften repräsentieren. Das statische Wissen ist vollständig auf die Domänenobjekte aufgeteilt und wird dort durch folgende Aspekte dargestellt:

**Relationen:** Die Beziehungen zwischen Objekten werden durch Relationen beschrieben. Die beiden wichtigsten Relationen sind die Class-Of– bzw. Is-A–Beziehung, durch die eine taxonomische Hierarchie auf den Objekten definiert wird, und die Has-Parts– bzw. Part-Of–Beziehung, die eine kompositionelle Hierarchie definiert. Beide zusammen erzeugen die grundlegende Begriffshierarchie, an der sich der Konfigurationsprozeß orientiert (siehe Abschnitt 3.2). Die Has-Implementation– bzw. Implementation-Of–Relation beschreibt den Zusammenhang zwischen abstrakten Funktionen und deren konkreten technischen Realisierungen. Darüber hinaus können weitere, domänenspezifische Relationen definiert werden.

**Attribute:** Relevante Eigenschaften eines Objektes werden durch Attribute beschrieben. Bei abstrakten Begriffen wird in der Regel der Wertebereich, bei konkreten Objekten der Wert des Attributes spezifiziert.

**Constraints:** Einschränkende Abhängigkeiten zwischen Objekten, ihren Attributen und Relationen werden durch konzeptuelle Constraints ausgedrückt. Durch die Constraints werden im Sinne der Domäne inkorrekte Lösungen ausgeschlossen ([Pau90], [Mey94]).

**Phasen:** Die Strategie für die Durchführung der Konfiguration eines Objektes wird durch die Definition von Phasen festgelegt. Die Phasen stellen domänenspezifisches Problemlösungswissen dar ([Hoc94]).

Alle genannten Aspekte eines Domänenobjektes werden entlang der Class-Of–Relation zu spezielleren Objekten weitervererbt, wobei auch Mehrfachvererbung möglich ist. Die ererbten Aspekte können beim erbenden Objekt weiter spezifiziert werden. Durch die objektzentrierte Wissensrepräsentation wird eine starke Strukturierung des Wissens erzwungen. Durch die Vererbung können Redundanzen vermieden werden. Beides erleichtert Aufbau und Wartung der Wissensbasis.

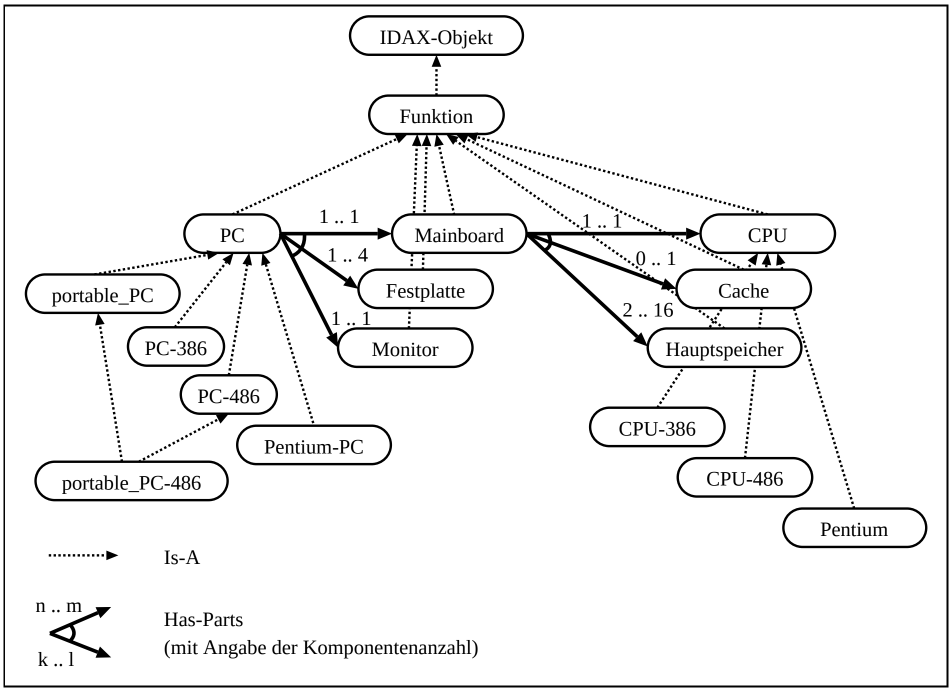


Abbildung 3.1: Ausschnitt aus der Begriffshierarchie der PC-Domäne (aus [Pau95])

Als Beispiel für die Konfiguration in technischen Domänen dient in IDAX derzeit die Konfiguration eines Personal–Computers. Diese Domäne ist lediglich zur Demonstration der Möglichkeiten des IDAX–Systems ausgewählt worden und stellt keine reale Anwendung dar, weswegen sie bislang nur unvollständig modelliert ist. Auch alle Beispiel innerhalb dieser Arbeit beziehen sich auf die PC–Domäne, deren Begriffshierachie in Abbildung 3.1 ausschnittweise dargestellt ist.

### 3.1.2 Dynamisches Wissen

Innerhalb eines Konfigurationsprozesses wird aufgabenspezifisches Wissen vom IDAX–System abgeleitet oder vom Benutzer eingegeben, um schrittweise eine Lösung aufzubauen. Dieses veränderliche Wissen wird in der dynamischen Wissensbasis verwaltet. Dazu wird ein problemklassenspezifisches Truth–Maintenance–System ([Rit92]) verwendet, das im wesentlichen der REDUX–Architektur entspricht, die von C. Petrie entwickelt wurde ([Pet91]). Da die Repräsentation des dynamischen Wissens im Kern auf einem JTMS ([Doy79]) basiert, ist es möglich, auch kausale

Abhängigkeiten zwischen den Informationen in der dynamischen Wissensbasis darzustellen. Außerdem werden neben der aktuell gültigen Teillösung auch alle bereits explorierten, ungültig gewordenen Lösungsteile gespeichert. Die dynamische Wissensbasis repräsentiert also stets vollständig die Resultate des Problemlösungsprozesses, ohne aber dessen Chronologie festzuhalten.

Da sich der Konfigurationsprozeß an der Begriffshierarchie der statischen Wissensbasis orientiert, werden in der dynamischen Wissensbasis Instanzen der Objekte, Relationen und Constraints aus der statischen Begriffshierarchie verwaltet. In diesem Zusammenhang sei insbesondere darauf hingewiesen, daß ein einzelnes Objekt der statischen Wissensbasis innerhalb eines Konfigurationsprozesses mehrfach instanziiert und in die dynamische Wissensbasis aufgenommen werden kann, da sowohl eine Komponente in mehreren Aggregaten vorkommen, als auch eine technische Realisierung mehrere Funktionen erfüllen kann. Zwischen den Objekten der statischen und der dynamischen Wissensbasis besteht also eine 1:n–Beziehung.

Eine spezielle Sicht auf das dynamische Wissen stellt die sogenannte Produkt–Datenbank dar. Dort sind alle Fakten zusammengefaßt, die unmittelbar Eigenschaften des zu konfigurierenden Objektes betreffen, wohingegen alle prozeßspezifischen Aspekte dort unsichtbar sind. Der Aufbau der Produkt–Datenbank ist deshalb — im Gegensatz zu demjenigen der dynamischen Wissensbasis — unabhängig von der technischen Struktur des Konfigurationsprozesses. Einem Domänenexperten, der mit IDAX arbeitet, sollte im wesentlichen nur die Produkt–Datenbank zugänglich sein, wohingegen der interne Ablauf des Konfigurationsprozesses ihm gegenüber weitgehend verborgen bleiben kann.

## 3.2 Konfigurationsprozeß

Der Konfigurationsprozeß in IDAX wird nach dem Prinzip der schrittweisen Top–Down–Verfeinerung entlang der in der statischen Wissensbasis definierten Begriffshierarchie durchgeführt. Eine Konfigurationsaufgabe wird initial durch ein oder mehrere offene Ziele beschrieben. Diese werden schrittweise reduziert, d.h. in weniger komplexe Teilziele zerlegt, bis diese Teilziele durch einen elementaren Konfigurationsschritt erfüllt werden können. Die Konfigurationsaufgabe ist gelöst, wenn alle Ziele erfüllt sind, es also keine offenen Teilziele mehr gibt. Alle offenen Ziele werden auf einer Agenda verwaltet, aus der jeweils die nächste zu bearbeitende Aufgabe ausgewählt wird. Diese Auswahl stellt eine wichtige Kontrollentscheidung dar, weil sie zwar nicht die Vollständigkeit, sehr wohl aber die Effizienz der Suche nach einer korrekten Konfiguration beeinflußt. Das notwendige Auswahlwissen wird durch die Definition von verschiedenen Konfigurationsphasen in den Konfigurationsprozeß eingebracht [Hoc94]. Der Bearbeitungsreihenfolge der einzelnen Teilaufgaben wird im folgenden keine weitere Beachtung geschenkt, da Thema dieser Arbeit die wissensbasierte Bearbeitung einer einzelnen, bereits ausgewählten Teilaufgabe ist.

Ein offenes Teilziel wird bearbeitet, indem ein geeigneter Konfigurationsschritt durchgeführt wird. Ein Konfigurationsschritt wird durch die Auswahl eines Operators festgelegt. Die Operatorauswahl stellt innerhalb des Konfigurationsvorgangs die zentrale Konstruktionsentscheidung dar und wird deswegen auch oft kurz als „Entscheidung" bezeichnet. Zu unterscheiden sind zwei Klassen von Konfigurationsschritten:

1. **Konfigurationsschritte, die neue Teilziele erzeugen.** Die zugehörigen Operatoren beschreiben eine Verfeinerung gemäß der in der Begriffshierarchie festgelegten Struktur. Es bestehen folgende Möglichkeiten:

   - Auswahl einer **Spezialisierung** des Konfigurationsobjekts entlang einer Is-A–Kante in der Begriffshierarchie

- Auswahl einer **Zerlegung** des Konfigurationsobjekts in seine Komponenten gemäß der Has-Parts–Kanten in der Begriffshierarchie

Da diese beiden Verfeinerungen auf ein Konfigurationsobjekt in IDAX nur alternativ angewendet werden können, muß vor dem Verfeinerungsschritt selbst eine Festlegung des Verfeinerungstyps erfolgen, was zu einer weiteren Art von Operatoren führt:

- Auswahl einer Verfeinerungsrelation für das Konfigurationsobjekt

2. **Konfigurationsschritte, die keine neuen Teilziele erzeugen,** also ein offenes Ziel unmittelbar erfüllen. Die entsprechenden Operatoren legen jeweils eine elementare Eigenschaft des Konfigurationsobjektes fest. Dazu gehören:

   - Festlegung des **Wertes eines Attributs** des Konfigurationsobjektes
   - Festlegung der **Kardinalität** (Anzahl des Auftretens) einer Komponente des Konfigurationsobjektes
   - Wahl einer **technischen Realisierung** für das Konfigurationsobjekt

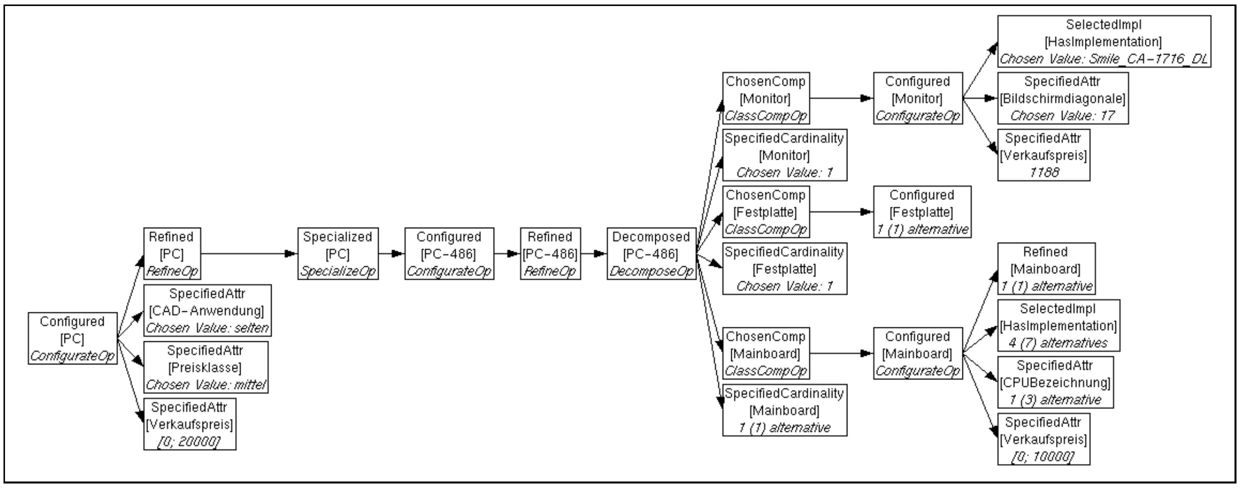


Abbildung 3.2: Beispiel für den Beginn eines Konfigurationsprozesses in der PC-Domäne

Durch das statische Domänenwissen ist eine maximale Menge möglicher, alternativ anwendbarer Operatoren für ein Ziel unabhängig vom aktuellen Stand des Konfigurationsprozesses gegeben. Diese Alternativenmenge wird bestimmt durch den erlaubten Wertebereich eines Attributes bzw. durch die von einem Objekt ausgehenden Relationen. In einer konkreten Entscheidungssituation stehen aber nicht immer alle diese Alternativen zur Verfügung:

- Operatoren können **unzulässig** sein. Damit werden externe, d.h. nicht aus dem Domänenwissen ableitbare Restriktionen ausgedrückt. Beispielweise könnte eine geeignete technische Realisierung zeitweilig nicht lieferbar sein. In diesem Fall wird der zugehörige Auswahl–Operator als „unzulässig“ markiert.

- Operatoren können **zurückgewiesen** sein. Damit werden Restriktionen ausgedrückt, die aus dem Domänenwissen heraus entstanden sind. Beispielsweise kann die Alternativenmenge durch Constraints eingeschränkt werden. Durch Constraints ausgeschlossene Operatoren werden als „zurückgewiesen“ markiert.

Die eingeschränkte Menge der auf ein Ziel anwendbaren Operatoren, die weder unzulässig noch zurückgewiesen sind, wird als Konfliktmenge des Zieles bezeichnet. Aus ihr muß zur Bearbeitung des Ziels ein Operator ausgewählt werden, wobei zu beachten ist, daß die Konfliktmenge sich im Lauf des Konfigurationsprozesses verändern kann. Einerseits kann ein bereits ausgewählter Operator später zurückgewiesen oder unzulässig werden, was eine Rücknahme der Entscheidung und aller ihrer Konsequenzen (Backtracking) notwendig macht. Die damit verbundenen Probleme werden im Abschnitt 3.3 besprochen. Andererseits kann ein zum Zeitpunkt der Bearbeitung des Ziels ausgeschlossener Operator später wieder zugelassen werden, wodurch sich eine neue Alternative bietet, welche ein Überprüfen der ursprünglichen Entscheidung bezüglich ihrer *Optimalität* notwendig macht. Diese Problematik wird im Abschnitt 3.4 behandelt.

## 3.3 Bedeutung der Operatorauswahl für den Verlauf des Konfigurationsprozesses

Wie bereits erwähnt, entspricht der Konfigurationsprozeß in IDAX einer Tiefensuche in einem Und–Oder–Graphen, wobei jede Teilaufgabe einem Oder–Knoten entspricht. Daraus folgt unmittelbar, daß die Bearbeitung einer Teilaufgabe durch Auswahl eines Operators genau diejenige Entscheidung darstellt, welche die weitere Suche auf einen Teil des Suchraumes einschränkt und damit möglicherweise das Auffinden einiger oder aller Lösungen verhindern kann. Deswegen kann jede Entscheidung für die Auswahl eines konkreten Operators, die gleichzeitig eine Entscheidung gegen alle anderen Alternativen darstellt, immer nur vorläufig sein, um nicht die Vollständigkeit des Suchprozesses zu gefährden.

Durch die Auswahl eines Operators, der mit keiner korrekten Lösung der Konfigurationsaufgabe verträglich ist, entsteht eine Inkonsistenz. Diese wird erkannt, sobald ein Teilziel auftritt, dessen Konfliktmenge leer ist, für das also kein anwendbarer Operator mehr existiert. Wenn diese Situation eintritt, kann der Suchprozeß nur dadurch fortgesetzt werden, daß ein oder mehrere zuvor ausgewählte Operatoren zurückgewiesen werden, bereits getroffene Entscheidungen also ungültig werden. Im einfachsten Fall wird dabei ein chronologisches Backtracking betrieben, d.h. die bisherigen Entscheidungen werden in umgekehrter Reihenfolge nacheinander ungültig gemacht, bis die Inkonsistenz behoben ist. Durch die Verwendung eines Truth–Maintenance–Systems kann stattdessen das Verfahren des *Dependency–Directed Backtracking* ([Doy79]) verwendet werden. Dadurch können gezielt diejenigen Entscheidungen zurückgezogen werden, die die Inkonsistenz verursacht haben, wenn geeignete Analyseverfahren zur Verfügung stehen, um die Ursache der Inkonsistenz zu ermitteln. Alle davon kausal unabhängigen Entscheidungen brauchen — im Gegensatz zum chronologischen Backtracking — nicht revidiert zu werden.

In jedem Fall aber stellt Backtracking einen zeitaufwendigen Umweg im Lösungsprozeß dar, der nach Möglichkeit dadurch vermieden werden soll, daß keine Operatoren ausgewählt werden, die zu Inkonsistenzen führen. In IDAX wird durch Constraint–Propagierung ([Pau90], [Mey94]) sichergestellt, daß alle Operatoren zurückgewiesen werden, die im Falle ihrer Auswahl unmittelbar eine Inkonsistenz erzeugen würden. Auf diese Weise können zahlreiche Situationen, die Backtracking erfordern würden, von vorneherein ausgeschlossen werden. Weil durch die Constraint–Propagierung jedoch wegen ihres hohen Aufwandes nur lokale, nicht aber globale Konsistenz garantiert wird, können Inkonsistenzen in Form von Teilzielblockaden nach wie vor entstehen.

Eine weitere Reduzierung des Auftretens von Inkonsistenzen kann durch eine Heuristik zur Auswahl eines in der vorliegenden Problemsituation besonders erfolgversprechenden Operators erzielt werden. Dazu muß zusätzliches Auswahlwissen in das System eingebracht werden, da die

Auswahl eines Operators in IDAX bisher nur zufällig oder durch eine Benutzereingabe geschieht. Das heuristische Auswahlwissen wird dabei in aller Regel unvollständig sein, also die Auswahl ungeeigneter Operatoren nie vollständig ausschließen, sondern nur reduzieren können. Das Vorgehen menschlicher Experten zeigt aber, daß in wissensintensiven Konstruktionsverfahren auf das Durchprobieren verschiedener Alternativen weitgehend verzichtet werden kann. Daher darf vom Einsatz zusätzlichen Auswahlwissens im Konfigurationsprozeß in vielen Fällen eine Verringerung des Backtracking–Aufwandes erwartet werden. In den Kapiteln 4 und 5 dieser Arbeit wird beschrieben, wie entsprechendes heuristisches Wissen in Form von Präferenzregeln in den Konfigurationsprozeß von IDAX integriert werden kann.

## 3.4 Bedeutung der Operatorauswahl für die Optimalität der Konfiguration

Falls mehrere korrekte Lösungen für die gestellte Konfigurationsaufgabe existieren, so möchte man in aller Regel nicht nur irgendeine, sondern eine möglichst gute, am liebsten sogar die optimale Lösung finden. Letzteres setzt allerdings voraus, daß es eine globale Optimierungsfunktion in der Domäne gibt. Diese Annahme ist für die technische Konfiguration im allgemeinen unrealistisch. Und selbst wenn ein globales Optimalitätskriterium existiert, bleibt das Problem, dieses für die lokal zu treffenden Konfigurationsentscheidungen zu operationalisieren, bestehen.

Die REDUX–Architektur unterstützt deswegen das schwächere Prinzip der *Pareto–Optimalität*, die ohne globales Optimierungsziel auskommt. Stattdessen wird eine Vielzahl von lokalen Optimalitätskriterien betrachtet, die sich jeweils auf genau einen Konfigurationsschritt beziehen. Es wird verlangt, daß keines dieser Kriterien besser erfüllt werden kann, indem nur ein einziger Konfigurationsschritt geändert wird. Hat man eine Lösung gefunden, für die dies gilt, so heißt diese Lösung pareto–optimal. Dabei kann es beliebig viele pareto–optimale Lösungen für eine Aufgabe geben.

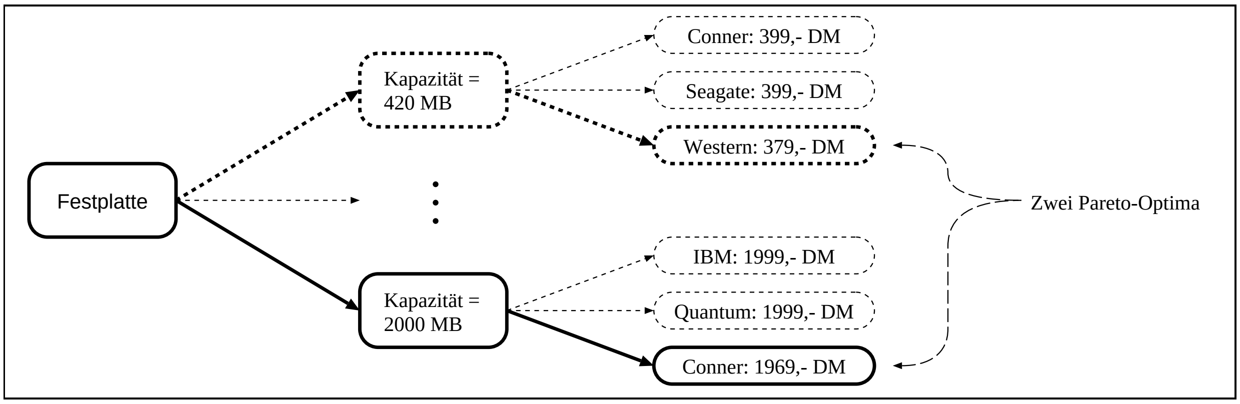


Abbildung 3.3: Auswahl einer Festplatte nach zwei lokalen Optimierungskriterien

**Beispiel:**

Bei der Auswahl einer Festplatte soll nach zwei lokalen Kriterien optimiert werden. Zum einen soll eine möglichst große, zum anderen eine möglichst billige Platte ausgewählt werden. Beide Kriterien lassen sich nicht gleichzeitig maximal erfüllen. In Abhängigkeit davon, ob zuerst über die Größe oder über den Preis entschieden wird, erhält man zwei unterschiedliche Lösungen, die aber gleichermaßen pareto–optimal sind (siehe Abb. 3.3).

Die Pareto–Optimalität kann bei der Konfiguration dadurch erreicht werden, daß bei jeder Entscheidung nur ein Operator ausgewählt werden darf, für den keine bessere Alternative zur Verfügung steht. Die Beurteilung der Alternativen kann dabei von Annahmen abhängen, die sich auf den aktuellen Stand des Konfigurationsprozesses beziehen. Die Pareto–Optimalität ist nun solange gewährleistet, wie diese Annahmen gültig sind und keine neuen Alternativen für die Entscheidung möglich sind. Um einen möglichen Optimalitätsverlust aufdecken zu können, müssen alle Annahmen, auf denen die Optimalität des Operators beruht, in die Rechtfertigung der Optimalität der Entscheidung aufgenommen werden. Ebenso müssen alle zum Zeitpunkt der Entscheidung ausgeschlossenen Alternativen in diese Rechtfertigung einfließen, da diese, falls sie später doch zur Verfügung stünden, eventuell eine bessere Entscheidung erlauben würden.

Mit Hilfe einer solchen Optimalitätsrechtfertigung kann der REDUX–Mechanismus nun jeden möglichen Optimalitätsverlust aufdecken. Wenn ein solcher Verlust bemerkt wird, so führt dies aber nicht automatisch zu einer Revision der betroffenen Entscheidung, da dies mit einem erheblichen Aufwand verbunden sein kann, der durch die zu erwartende Verbesserung der Lösung nicht zu rechtfertigen ist. Stattdessen wird der Optimalitätsverlust nur durch einen Eintrag auf einer speziellen Agenda angezeigt und erhält damit den Rang einer speziellen Teilaufgabe innerhalb des Konfigurationsprozesses (siehe Abb. 3.4). Es ist nun eine Frage der Strategie des Konfigurationssystems, ob und wann der Optimalitätsverlust behoben wird. Wenn die Optimalität einer Entscheidung wiederhergestellt werden soll, so muß unter den neuen Gegebenheiten erneut ein optimaler Operator bestimmt werden. Ist dies derselbe wie zuvor, erhält dieser lediglich eine neue Rechtfertigung für seine Optimalität. Andernfalls muß der bisherige Operator zurückgezogen werden, wodurch eine große Anzahl von abhängigen Entscheidungen ebenfalls ungültig werden kann. Das beschriebene Vorgehen garantiert die Pareto–Optimalität der Lösung, wenn keine Optimalitätsverluste (mehr) angezeigt werden.

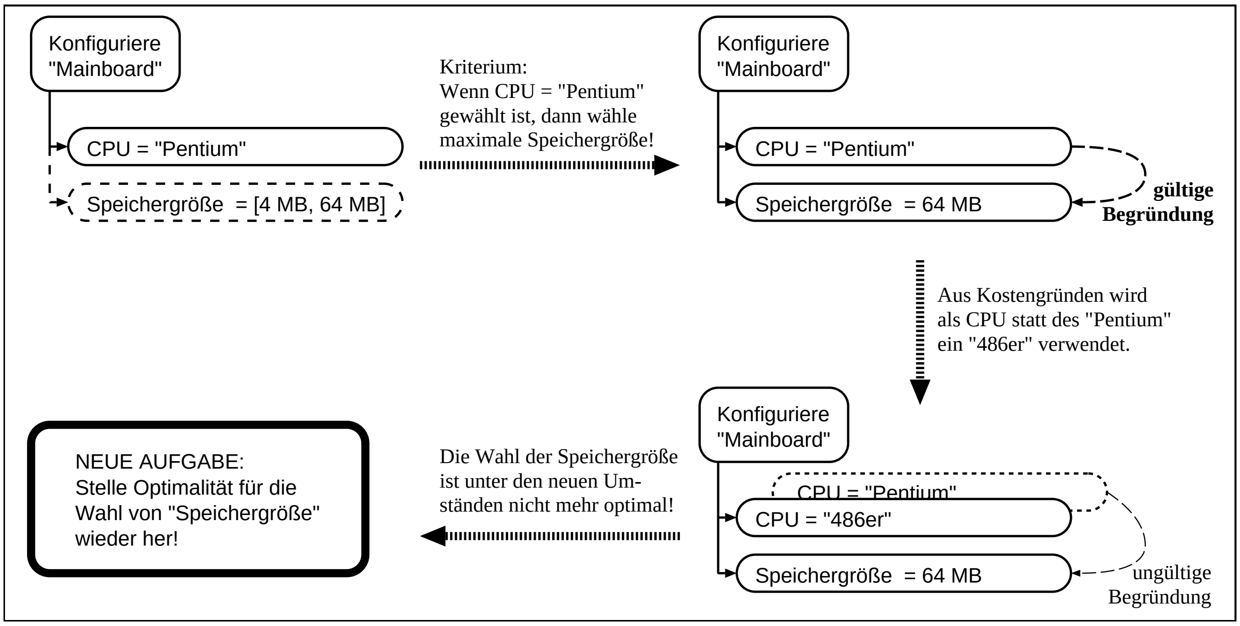


Abbildung 3.4: Beispiel für das Auftreten eines Optimalitätsverlustes

Das Konzept der Pareto–Optimalität setzt voraus, daß aus der jeweils gültigen Konfliktmenge stets ein Operator bestimmt werden kann, zu dem es keine bessere Alternative gibt. Allerdings wird nicht gefordert, daß ein solcher Operator eindeutig bestimmt ist. Um die Wahl des lokal besten Operators treffen zu können, ist Domänenwissen in Form von Güte- oder Präferenzkriterien erforderlich. Solches Wissen ist in IDAX bislang nicht verfügbar, weswegen die Operatorauswahl nur zufällig erfolgen kann. Bei diesem völlig uninformierten Vorgehen sind im oben beschriebenen Sinn alle Operatoren gleich gut, die Pareto–Optimalität ist also auf triviale Weise immer gegeben. Durch die Einführung von Präferenzregeln, wie sie in den Kapiteln 4 und 5 beschrieben wird, können dagegen lokale Gütekriterien formuliert und bei der Konfiguration berücksichtigt werden. Zur Einhaltung der Pareto–Optimalität muß dann in der Folge die lokale Optimalität der betroffenen Entscheidungen überwacht werden. Detaillierte Ausführungen hierzu finden sich in Kapitel 6.

Da IDAX als Werkzeug zur Unterstützung von interaktiven Konfigurationsprozessen konzipiert ist, kann die Auswahl eines Operators auch direkt durch den Benutzer, dem in diesem Fall Kenntnis der Domäne unterstellt wird, erfolgen. Trifft der Benutzer eine Entscheidung, so wird grundsätzlich angenommen, daß diese in der gegebenen Situation optimal ist. Das Konfigurationssystem sollte den Benutzer aber dann informieren, wenn eine Veränderung der Situation eingetreten ist, welche die Optimaliät seiner Entscheidung in Frage stellt. Dies geschieht bisher in IDAX genau dann, wenn eine zum Zeitpunkt der Benutzerentscheidung ausgeschlossene Alternative wieder gültig wird. Eine weitergehende Unterstützung des Benutzers kann deswegen nicht erfolgen, weil dieser seine Entscheidung dem System gegenüber nicht begründet. In Kapitel 7 wird eine Möglichkeit gezeigt, wie der Benutzer seine Entscheidungen in einer Form begründen kann, die es dem System ermöglicht, die Optimalität der Benutzerentscheidungen während des Konfigurationsprozesses zu überwachen.

## 3.5 Erweiterung der IDAX-Architektur für die regelbasierte Operatorauswahl

Eine Komponente, die eine regelbasierte Auswahl von Operatoren im Rahmen des IDAX-Systems ermöglichen soll, muß drei wesentliche Funktionen realisieren:

**Definition der Präferenzregeln:** Die Angabe von Präferenzen bezüglich der Auswahl alternativer Konfigurationsschritte stellt eine Erweiterung des bisherigen Domänenwissens dar. Deswegen muß die statische Wissensbasis so ausgebaut werden, daß sie zur Eingabe und Speicherung von Präfernzregeln genutzt werden kann (siehe Abschnitt 4.2).

**Auswertung der Präferenzregeln:** Da bislang im IDAX-System kein Teil des Domänenwissens in Form von Regeln dargestellt wird, fehlt auch ein Inferenzmechanismus zur Regelauswertung. Daher wird zu diesem Zweck ein Prolog-Interpreter an das IDAX-System angekoppelt. Um auch komplexe Bedingungen in Präferenzregeln leicht formulieren zu können, werden zahlreiche Prädikate in der Prolog-Datenbasis vordefiniert, die es zusammen mit speziellen Zugriffsmethoden erlauben, Anfragen an die dynamische Wissensbasis zu stellen (siehe Abschnitt 4.3).

**Kontrolle der Operatorauswahl mit Präferenzregeln:** Das durch Präferenzregeln gegebene Auswahlwissen muß in den Prozeß der Operatorauswahl integriert werden. Dazu wird die Kontrollkomponete von IDAX so erweitert, daß sie in der Lage ist, bei Bedarf die Regelauswertung in Gang zu setzen und deren Ergebnisse in den Konfigurationsprozeß einzubringen (siehe Abschnitt 4.4).

Abbildung 3.5 zeigt eine schematische Darstellung der Architektur des IDAX-Systems zusammen mit der im Rahmen dieser Arbeit entwickelten Komponente zur regelbasierten Auswahl von Konfigurationsschritten.

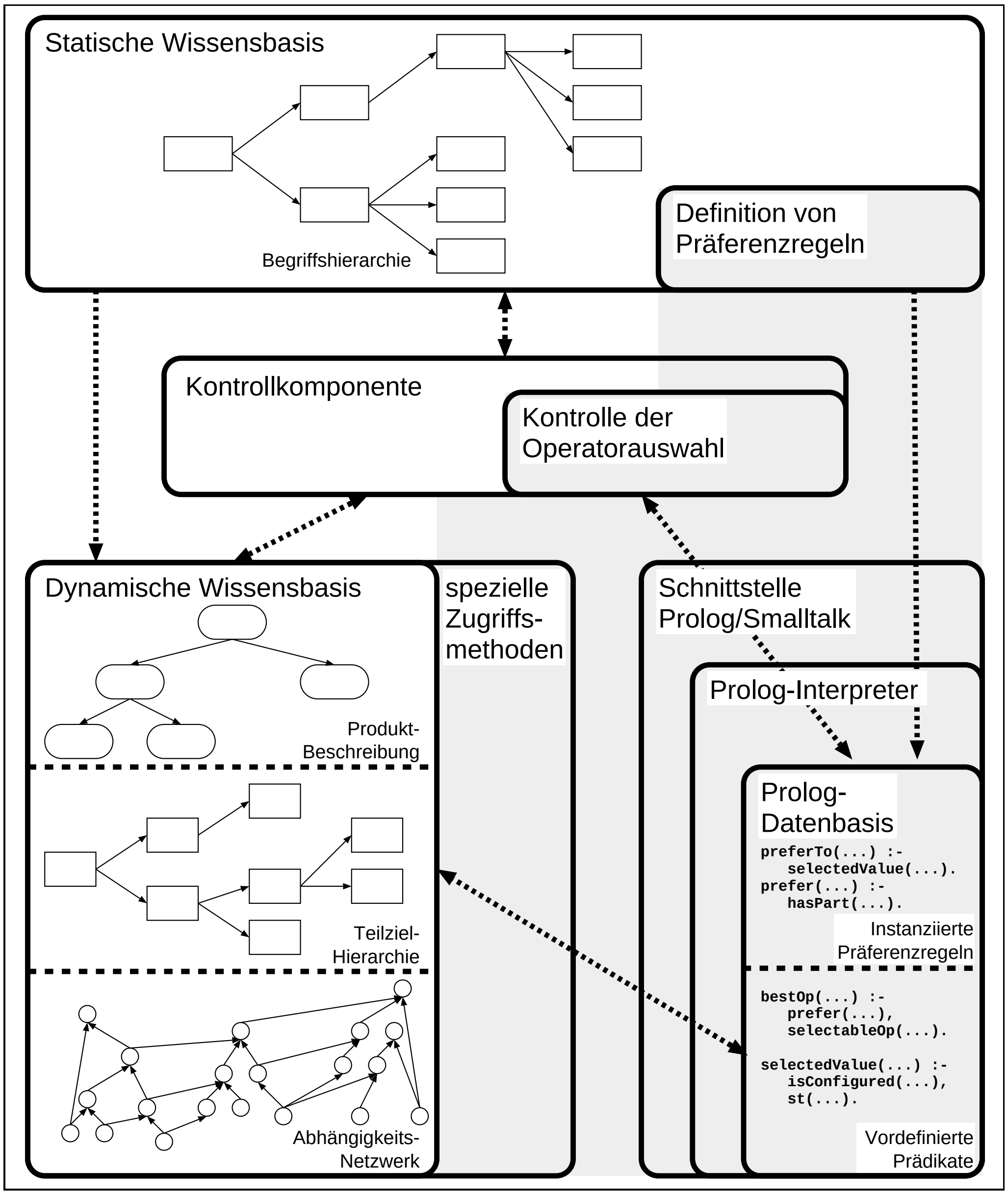


Abbildung 3.5: Architekur des IDAX-Systems mit der Regelkomponente (grau unterlegt)

# Kapitel 4

# Operatorauswahl durch Präferenzregeln

## 4.1 Zielsetzung

Durch die Anordnung der Konfigurationsobjekte in der Begriffshierarchie und durch die Definition von konzeptuellen Constraints ist bereits alles Wissen gegeben, das erforderlich ist, um die Korrektheit der Konfiguration zu gewährleisten. Zusätzliches Wissen in Form von Präferenzregeln für die Operatorauswahl kann also nur die Qualität der Lösung oder die Effizienz des Lösungsprozesses verbessern, da es im Suchraum weder Lösungsmöglichkeiten hinzufügt noch entfernt. Aus dieser Überlegung ergeben sich unmittelbar die beiden Ziele, die mit dem Einsatz von Präferenzregeln zur Operatorauswahl verfolgt werden:

**Heuristische Steuerung des Suchprozesses:** Die Auswahl eines Operators kann im Verlauf des Konfigurationsprozesses zu einer Inkonsistenz führen, die eine Rücknahme der Operatorauswahl und anderer davon abhängigen Entscheidungen erzwingen kann. Dieses zeitaufwendige Backtracking möchte man durch „geschickte" Operatorauswahl so weit als möglich vermeiden. Deshalb soll an die Stelle einer blinden Suche, also der zufälligen Auswahl eines Operators, ein besser informiertes Verfahren treten. Falls das vorhandene Wissen ausreichen würde, um mit Sicherheit über die Eignung einer Alternative zu entscheiden, so ließe sich ein dementsprechender Constraint formulieren, der die Auswahl auf die geeigneten Operatoren einschränkt. Unsicheres oder unvollständiges Auswahlwissen kann dagegen nicht durch einen Constraint dargestellt werden, weil dadurch die Vollständigkeit der Suche gefährdet würde. Durch die Formulierung von Präferenzregeln soll die Möglichkeit eröffnet werden, unsicheres Expertenwissen in den Konfigurationsprozess einzubringen, um durch eine Auswahlheuristik die Suche nach einer korrekten Lösung zu beschleunigen.

**Lokale Optimierung:** Um die in Abschnitt 3.4 beschriebene Pareto–Optimalität der Lösung der Konfigurationsaufgabe bestimmen zu können, müssen die zur Auswahl stehenden Alternativen hinsichtlich eines lokalen Gütekriteriums geordnet werden. Ein solches Kriterium kann sowohl Teil des allgemeinen Domänenwissens sein, als auch aus einem speziellen Wunsch des Benutzers resultieren. Beide Arten von Kriterien sollen durch Präferenzregeln ausgedrückt werden.

Um die Pareto–Optimalität im Verlauf des Konfigurationsprozesses überwachen zu können, müssen alle Voraussetzungen, die zur Anwendung einer Präferenzregel geführt haben, in die Rechtfertigung der Optimalität der dadurch getroffenen Entscheidung einfließen. Dann kann mit Hilfe der Abhängigkeitsverwaltung des REDUX–Systems erkannt werden, daß der Wegfall einer solchen Voraussetzung zu einem späteren Zeitpunkt eventuell die lokale Optimalität der betreffenden Entscheidung in Frage stellt.

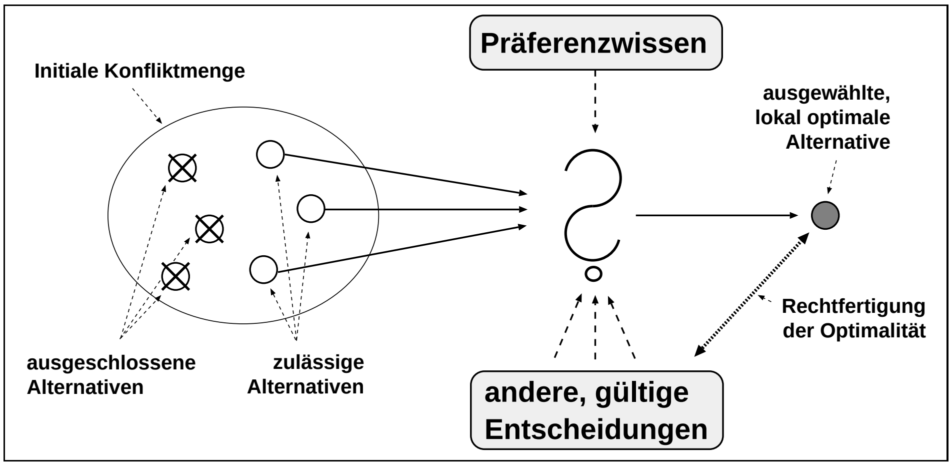


Abbildung 4.1: Entscheidungs–Situation bei der Operatorauswahl

Zu beachten ist, daß sich die beiden vorgenannten Ziele durchaus widersprechen können, da die (lokal) optimale Entscheidung nicht unbedingt auch die erfolgversprechendste ist. Beispielsweise würde eine lokale Optimierung mit dem Ziel, einen möglichst preiswerten PC zu konfigurieren, stets dazu führen, daß bei der Auswahl des Prozessors ein 386er–Chip bevorzugt wird. Andererseits könnte aber die Erfahrung gezeigt haben, daß ein 386er–Prozessor in den meisten Fällen zu wenig Rechenleistung bietet, um die gestellten Anforderungen zu erfüllen, weswegen die Auswahl eines Pentium–Chips erfolgversprechender scheint. Will man wie in diesem Beispiel den Preis des Konfigurationsobjektes optimieren, so muß man im Verlauf des Konfigurationsprozesses oft mehrere Alternativen — beginnend mit der billigsten — durchprobieren, bis eine Lösung gefunden wird, die auch den gestellten Anforderungen bezüglich der Leistung genügt.

In diesem Problem spiegelt sich der allgemeine Trade–Off zwischen der schnellen Suche nach einer beliebigen Lösung einerseits und der aufwendigeren Suche nach einer besonders guten Lösung andererseits wider. Es ist daher wichtig, sich bei der Modellierung der Präferenzregeln der gewünschten Zielsetzung bewußt zu sein. Da im Entscheidungsprozeß eines menschlichen Experten aber oft beide Ziele miteinander verschmolzen sind, darf man realistischerweise nicht erwarten, daß sich die Intention einer Präferenzregel eindeutig einem der beiden Ziele zuordnen läßt. Insbesondere kann man einer bestehenden Regel ohne Kenntnis des ihr zugrundeliegenden Expertenwissens nicht ansehen, welcher Zielsetzung sie dienen soll.

### 4.1.1 Einführendes Beispiel

Zu Beginn des Konfigurationsprozesses für einen Personal–Computer sei die in Abbildung 4.2 gezeigte Situation gegeben: Der Benutzer hat bereits festgelegt, daß der PC zur hohen Preisklasse gehören und nur selten für CAD-Anwendungen eingesetzt werden soll. Es stehen zwei Teilaufgaben zur Bearbeitung an, nämlich die Spezialisierung des abstrakten Begriffs „PC“und die Wahl eines Verkaufspreises.

Von diesen beiden Aufgaben wird zunächst die erste bearbeitet, d.h. es muß eine Spezialisierung des „PC“ ausgewählt werden. Dafür stehen in der Begriffshierarchie (siehe Abb. 3.1) die vier

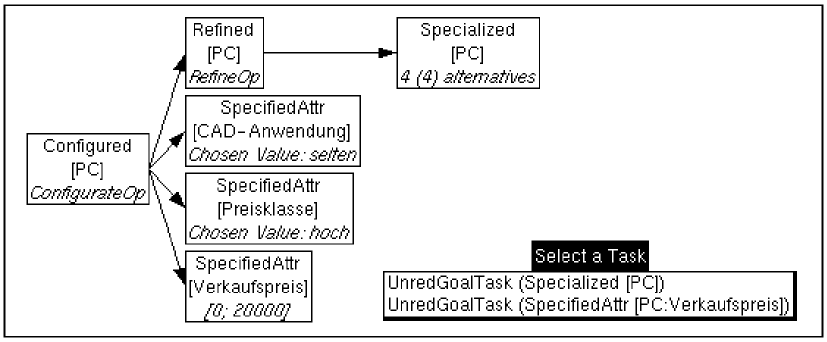


Abbildung 4.2: Beginn eines Konfigurationsprozesses

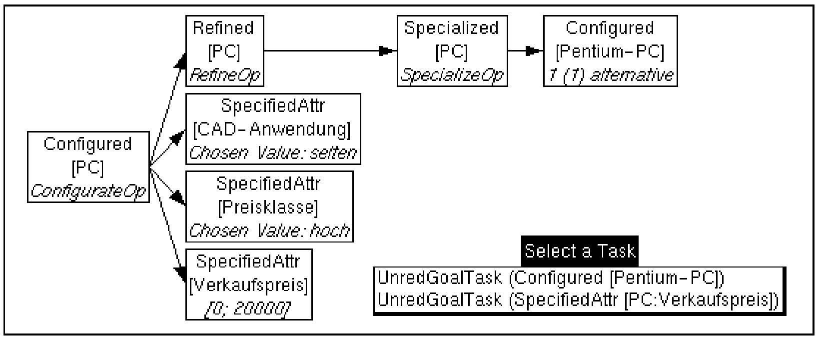


Abbildung 4.3: Auswahl einer Spezialisierung

Alternativen „portable_PC", „PC-386", „PC-486" und „Pentium-PC" zur Verfügung. Es sei nun eine Präferenz bekannt, die besagt:

**Wenn** als Preisklasse „hoch" gewählt ist,
  **dann** wähle als Spezialisierung „Pentium-PC".

**Wenn** als Preisklasse „niedrig" gewählt ist,
  **dann** wähle als Spezialisierung „PC-386".

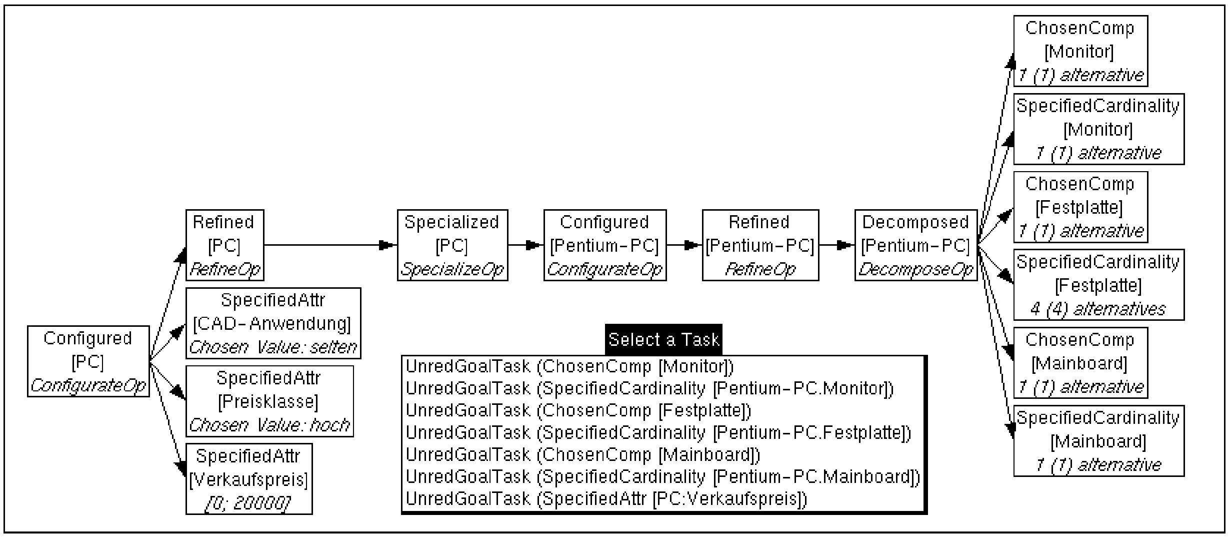


Abbildung 4.4: Fortführung des Konfigurationsprozesses

Die Anwendung der ersten Regel führt zu der in Abbildung 4.3 gezeigten Situation. Der Konfigurationsprozeß kann jetzt, ausgehend von dieser Spezialisierungsentscheidung Schritt für Schritt fortgesetzt werden (siehe Abb. 4.4).

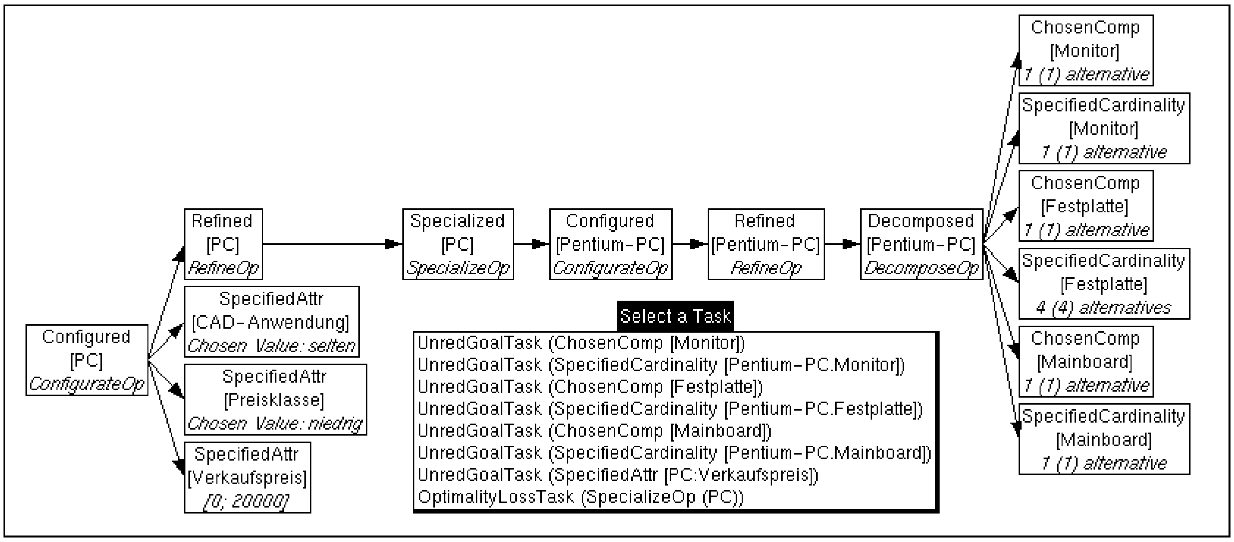


Abbildung 4.5: Änderung der ursprünglichen Anforderung

Es kann aber beispielsweise passieren, daß sich während des Fortgangs des Konfigurationsprozesses die Anforderungen des Benutzers ändern und der zu konfigurierende PC nicht mehr der hohen, sondern der niedrigen Preisklasse angehören soll (siehe Abb. 4.5). In diesem Fall ist die vorher getroffene Entscheidung, einen „Pentium-PC“ als Spezialisierung auszuwählen, nicht mehr optimal, da diese Auswahl nur schwerlich zu einer Konfiguration führen wird, die in der niedrigen Preisklasse angesiedelt ist. Die Abhängigkeit zwischen der Festlegung von Preisklasse und Spezialisierung des „PC“ hat also bei Änderung der einen Entscheidung zu einem Optimalitätsverlust der anderen Entscheidung geführt, der nun als neuer Eintrag auf der Liste der noch zu bearbeitenden Teilaufgaben vermerkt wird (letzte Zeile).

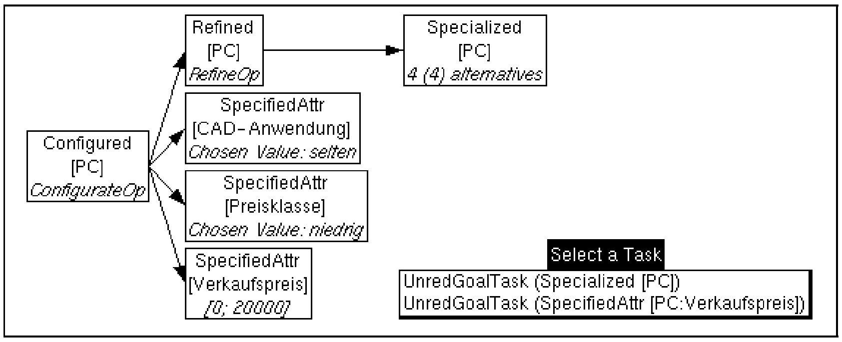


Abbildung 4.6: Rücknahme der supotimalen Entscheidung

Entweder kann nun der Konfigurationsprozeß mit der suboptimal gewordenen Entscheidung fortgesetzt oder aber der Optimalitätsverlust behoben werden. Letzteres geschieht, indem die bereits getroffene Spezialisierungs-Entscheidung zurückgenommen wird, wie es Abbildung 4.6 zeigt. Unangenehmer Effekt dieser Maßnahme ist, daß alle von der zurückgezogenen Entscheidung abhängigen Entscheidungen ebenfalls zurückgesetzt werden müssen. Die Auswahl einer Spezialisierung ist nun erneut eine zu bearbeitende Teilaufgabe. Die zweite der oben angegebenen Präferenzregeln führt dazu, daß nun als bestgeeignete Spezialisierung ein „PC-386“ ausgewählt wird (siehe Abb. 4.7).

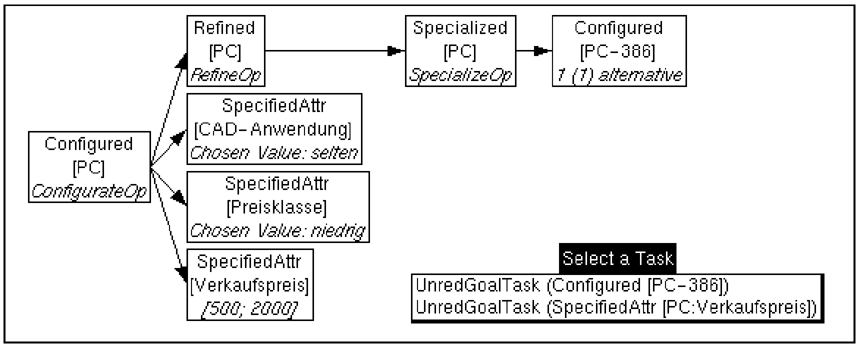


Abbildung 4.7: Auswahl einer anderen Spezialisierung

## 4.2 Definition der Präferenzregeln

Durch Präferenzregeln wird ausgedrückt, welchen von mehreren alternativen Konfigurationsschritten ein Experte unter bestimmten Voraussetzungen bevorzugt durchführen würde. Daher gehören Präferenzregeln eindeutig zum Experten- bzw. Domänenwissen und bilden deshalb einen Teil der statischen Wissensbasis. Genau wie das übrige statische Wissen in IDAX werden die Präferenzregeln dort als ein Aspekt der Domänenobjekte verwaltet, wodurch sich die Repräsentation des Präferenzwissens in die bestehenden Strukturen innerhalb der statischen Wissensbasis einfügt. (sieher Abb. 4.8). Jedem zu konfigurierenden Objekt kann ein Satz von Regeln zugeordnet werden, der das Präferenzwissen für diejenigen Entscheidungen beschreibt, die bei der Konfiguration des jeweiligen Objektes getroffen werden müssen. Diese Zuordnung bietet mehrere Vorteile:

**Kleine, modulare Regelmengen:** Die Verteilung der Präferenzregeln auf die Objekte der statischen Wissensbasis führt als solche bereits zu relativ kleinen Regelmengen. Diese können aber bei Bedarf noch weiter modularisiert werden, da die Anwendung einer Präferenzregel auf genau eine Konfigurationsentscheidung eingeschränkt ist. Wegen dieser strikten Lokalität sind die einzelnen Regelsätze voneinander unabhängig und können separat erstellt und gewartet werden. Dadurch bleibt das Regelsystem auch bei einer großen Gesamtzahl von Regeln überschaubar.

**Redundanzvermeidung durch Vererbung:** Präferenzregeln sollen in der Regel auch für die Spezialisierungen eines Domänenobjektes gelten. Durch die Ausnutzung des Vererbungsmechanismus entlang der Spezialisierungshierarchie kann auf die mehrfache Angabe solcher Regeln verzichtet werden. Da lokal definierte Regeln bevorzugt ausgewertet werden, ist es möglich, ererbte Regeln bei den erbenden Objekten zu überschreiben.

**Einbindung in Begriffshierarchie:** Es ist oft erforderlich, im Bedingungsteil von Regeln andere Objekte der aktuellen Teilkonfiguration zu referenzieren als dasjenige, auf welches sich der Regelkopf bezieht. Da Domänenobjekte während der Konfiguration mehrfach instanziiert werden können, reicht der Name der Domänenobjekte zur Identifikation von dynamischen Objekten nicht aus. Stattdessen müssen sie durch ihre Lage bezüglich der Dekompositions- und Spezialisierungshierarchien beschrieben werden, was durch die Einbindung der Präferenzregeln in die Begriffshierarchie entschieden erleichtert wird.

Die Präferenzregeln werden in Prolog–Notation angegeben, d.h. sie haben einen Regelkopf, der aus genau einem Prädikat besteht, und einen Bedingungsteil, der aus einer Liste von implizit konjunktiv verknüpften Prädikaten besteht, die auch leer sein kann. Im Regelkopf werden beschrieben

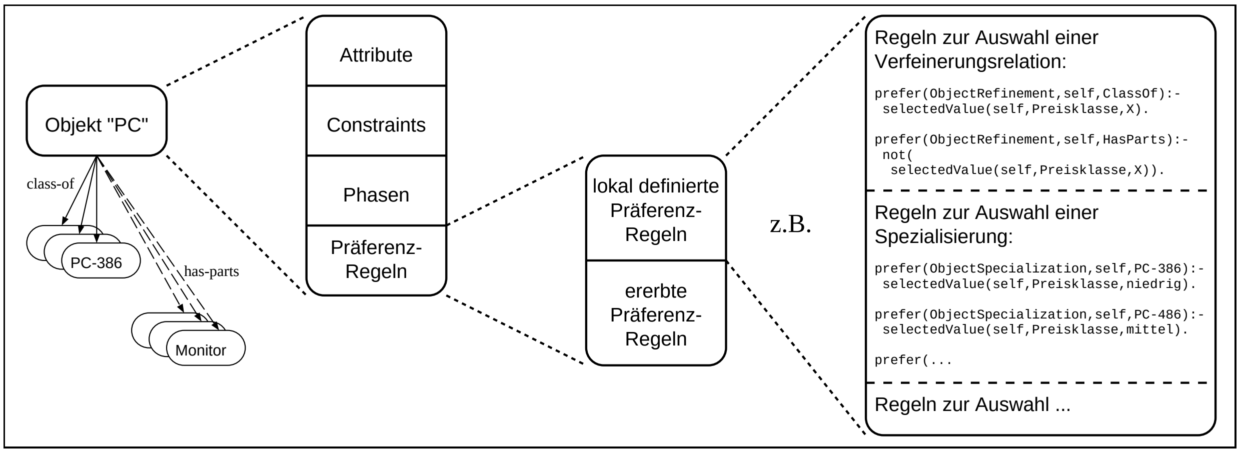


Abbildung 4.8: Einbindung von Präferenzregeln in die Strukturen der statischen Wissensbasis

- die betroffene Konfigurationsentscheidung,
- die Art der gewünschten Präferenz und
- die präferierte Entscheidungsalternative.

Im Bedingungsteil der Regel wird der Kontext der dynamischen Wissensbasis beschrieben, in welchem die im Regelkopf festgelegte Präferenz Gültigkeit erlangen soll. Zu diesem Kontext gehören

- die bereits erstellte Teilkonfiguration (die Produktbeschreibung),
- der Stand des Konfigurationsprozesses und
- die Anforderungen des Benutzers.

Die wesentliche Schwierigkeit beim Entwurf von Präferenzregeln liegt darin, den bedingenden Kontext exakt so allgemein zu beschreiben, daß alle Fälle, in denen die jeweilige Präferenz gelten soll, auch tatsächlich den Bedingungsteil erfüllen, und andererseits die Bedingungen eng genug zu fassen, damit die Präferenz nicht in unpassenden Situationen angewendet wird. Wenn mehrere verschiedene Präferenzen definiert werden, die in unterschiedlichen Kontexten gelten sollen, muß insbesondere darauf geachtet werden, daß sich die Bedingungsteile von Regeln, die zu unterschiedlichen Präferenzrelationen gehören, gegenseitig ausschließen, damit die Eindeutigkeit der Präferenz gewährleistet ist.

Zur Eingabe und Modifikation von Präferenzregeln steht dem Benutzer innerhalb der graphischen Oberfläche ein spezieller Editor zur Verfügung. Dieser unterstützt die Regeleingabe dadurch, daß auf Wunsch die Namen aller vordefinierten Prädikate sowie die Bezeichner für die verschiedenen Typen von Konfigurationsaufgaben aufgelistet werden, so daß der Benutzer lediglich den gewünschten Eintrag auswählen muß (siehe Abb. 4.9). Wählt der Benutzer ein vordefiniertes Prädikat aus, so wird zusammen mit dessen Namen auch ein Template eingefügt, das Anzahl und Bedeutung der Parameter anzeigt, die von dem jeweiligen Prädikat erwartet werden. Diese Hilfestellung erleichtert insbesondere solchen Experten, die IDAX nur sporadisch benutzen, die Eingabe von Präferenzregeln.

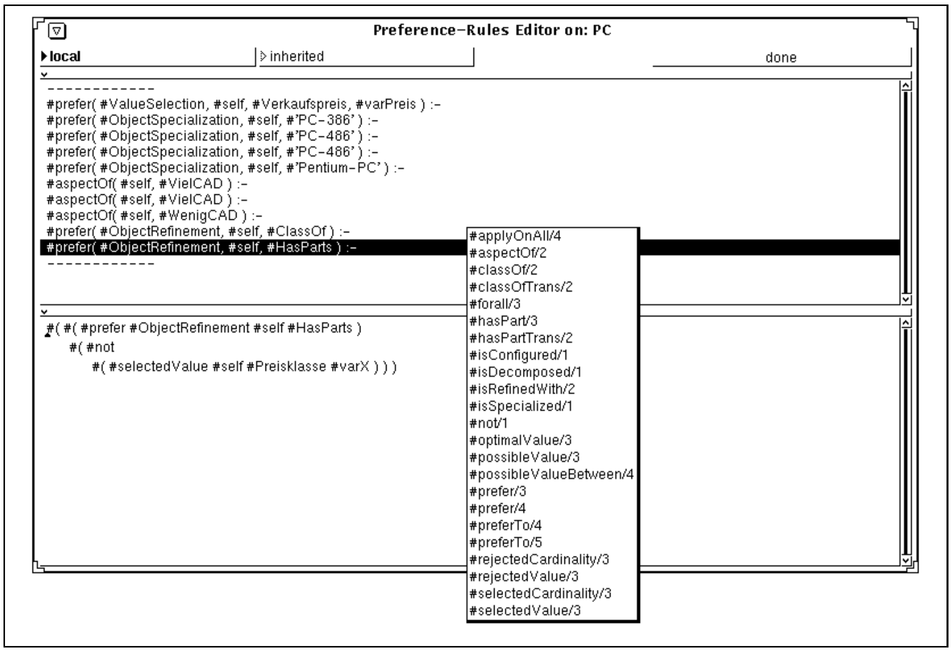


Abbildung 4.9: Der Editor zur Eingabe und Modifikation von Präferenzregeln

## 4.3 Auswertung der Präferenzregeln

Da die Anzahl der Fakten in der dynamischen Wissensbasis, die möglicherweise in Vorbedingungen von Regeln eingehen können, sehr groß ist im Vergleich zur Anzahl der alternativen Präferenzen in einer konkreten Entscheidungssituation, bietet es sich an, zur Verarbeitung der Präferenzregeln einen rückwärtsverkettenden Regelinterpretierer einzusetzen. Die Wahl fiel aus den folgenden drei Gründen auf einen Prolog–Interpreter:

**Effizienz:** Der Inferenzmechanismus von Prolog erlaubt eine effiziente Auswertung der Präferenzregeln. Ein in Smalltalk implementierter Regelinterpreter von ähnlicher Leistungsfähigkeit ist zumindest dem Autor nicht bekannt.

**Bekanntheit:** Wegen der großen Verbreitung von Prolog sind Syntax und Semantik dieser Sprache vielen Benutzern vertraut. Dies erleichtert die Definition von Präferenzregeln, da kein neuer Formalismus erlernt werden muß.

**Verfügbarkeit:** Das verwendete SWI–Prolog ([Wie93]) ist als Public–Domain–Software auf vielen gebräuchlichen Rechneranlagen verfügbar.

### 4.3.1 Ankopplung eines Prolog–Systems

Die Kopplung zwischen dem Prolog–Interpreter und dem ansonsten vollständig in Smalltalk implementierten IDAX–System erfolgt über eine speziell für KI–Anwendungen entwickelte Schnittstelle ([NL94]), die eine Kommunikation zwischen beiden Komponenten mit Hilfe von UNIX–Sockets realisiert. Um während der Bearbeitung einer Teilaufgabe die bestehenden Präferenzen auswerten zu können, muß der Prolog–Interpreter sowohl auf Teile des statischen als auch des dynamischen Wissens des IDAX–Systems zugreifen können. In der statischen Wissensbasis sind nämlich die Präferenzregeln gespeichert, die zur Anwendung kommen können, und in der dynamischen Wissensbasis sind die Fakten gespeichert, die zur Erfüllung der Bedingungsteile der Präferenzregeln abgefragt werden müssen. Da der Prolog–Interpreter nicht unmittelbar auf die Daten in den Wissensbasen des IDAX–Systems zugreifen kann, müssen diese in die Datenbank des Prolog–Systems übertragen werden. Die Wissensbasen von IDAX, und davon insbesondere die dynamische, stellen daher für den Prolog–Interpreter in diesem Sinne externe Datenbanken dar. Eine solche heterogene Architektur bringt zwei grundsätzliche Schwierigkeiten mit sich:

- Jede Anfrage des Prolog–Interpreters an die Wissensbasis verbraucht Zeit, nicht nur für den reinen Datenaustausch zwischen den beiden Komponenten, sondern auch für die Bearbeitung der Anfrage innerhalb der Wissensbasis.
- Die Speicherung von Daten aus der externen Wissensbasis in der Prolog–Datenbank führt zu einer Redundanz in der Datenhaltung, die nicht wegen des erhöhten Speicherbedarfes problematisch ist, sondern wegen der Notwendigkeit, die Konsistenz der kopierten Daten bei Änderungen in der Wissensbasis sicherzustellen.

Beide Probleme sind eng miteinander verbunden, da sich einerseits redundante Datenhaltung dadurch vermeiden läßt, daß vom Prolog–Interpreter mehrfach benötigte Daten jedesmal erneut von der IDAX–Wissensbasis erfragt werden, und andererseits die Zahl der Anfragen an die Wissensbasis gerade dadurch verringert werden kann, daß einmal erfragte Fakten in der Prolog–Datenbasis abgelegt werden.

In der derzeitigen Implementierung der Regelkomponente wird auf eine redundate Speicherung von Daten aus der dynamischen Wissensbasis in der Prolog–Datenbasis vollständig verzichtet, um Konsistenzproblemen aus dem Wege zu gehen. Da die Daten in der dynamischen Wissensbasis während des Konfigurationsprozesses ständig erweitert und geändert werden, scheint es nicht sinnvoll, Kopien dieser Daten auf Dauer in der Prolog–Datenbasis abzulegen, weil dann ein aufwendiges Update-Verfahren erforderlich würde. Speziell bei umfangreichen Präferenzbeweisen könnte sich aber eine Effizienzsteigerung dadurch erzielen lassen, daß aus der dynamischen Wissensbasis abgefragte Daten lediglich für die Dauer eines Beweises in der Prolog–Datenbasis zwischengespeichert werden, um mehrfache Anfragen nach einem identischen Datum zu vermeiden.

Die Probleme der Kopplung von (regelbasierten) Expertensystemen mit externen Datenbanken sind in der Literatur ausführlich behandelt worden (siehe z.B. [Reu87], [Ric92]). Allerdings lassen sich die dort vorgestellten Lösungsansätze nur sehr eingeschränkt auf die hier vorliegende Situation übertragen, da es sich bei der Wissensbasis von IDAX nicht um ein klassisches Datenbanksystem handelt. Im Hinblick auf das Retrieval sind vor allem die folgenden Unterschiede wichtig:

- Die Wissensbasis von IDAX ist verhältnismässig klein, d.h. sie kann vollständig im Hauptspeicher gehalten werden.
- Die Daten in der IDAX–Wissensbasis sind durch ein Abhängigkeitsnetzwerk miteinander verknüpft.
- Es existiert keine allgemeine Anfragesprache für die Wissensbasis, die über mächtige Zugriffsoperatoren verfügt.

Aus diesen Unterschieden ergibt sich, daß — anders als bei Anfragen an große Datenbanken — die Zeit, die zur Bearbeitung einer einzelnen Anfrage an die Wissensbasis benötigt wird, sehr kurz ist. Deswegen hängt der Aufwand für Anfragen an die Wissensbasis verhältnismäßig stärker von der Zeit ab, die für den Datenaustausch zwischen den beiden Komponenten verbraucht wird. Diese Zeit hängt aber im wesentlichen von der Anzahl und kaum von der Art der Anfragen an die Wissensbasis ab. Um nun die Zahl der erforderlichen Anfragen zu reduzieren, können mehrere einfache

Anfragen innerhalb der Auswertungsregeln (siehe Abschnitt 4.3.2) zu einer komplexeren Anfrage zusammengefaßt werden. Da es keine mächtige Anfragesprache für die Wissensbasis gibt, muß zur Abarbeitung einer neuen komplexen Anfrage aber zusätzlich eine passende Zugriffsmethode innerhalb der Wissensbasis programmiert werden.

### 4.3.2 Kommunikation mit dem Regelinterpreter

Für die Regelauswertung übernimmt der Prolog–Interpreter die Rolle eines Servers, an den das IDAX–System Anfragen bezüglich der Präferenz von Operatoren in der Form eines Prolog–Prädikates richten kann. Dieses Prädikat versucht der Prolog–Interpreter zu beweisen, wobei innerhalb des Beweises Rückfragen an das IDAX–System gestellt werden können. Auf diese Weise werden Fakten aus der dynamischen Wissensbasis von IDAX abgefragt, die im Bedingungsteil der Präferenzregeln angesprochen werden. Für diese Rückfragen an die dynamische Wissensbasis ist ein Satz von Prädikaten definiert, mit dessen Hilfe die relevanten Fakten der dynamischen Wissensbasis beschrieben werden können. Diese vordefinierten Prädikate werden wegen ihrer Funktion als *IDAX–Prädikate* bezeichnet. Syntax und Semantik der IDAX–Prädikate werden im Abschnitt 5.2 detailliert beschrieben.

Für jedes IDAX–Prädikat (und auch für jedes andere vordefinierte Prädikat) ist in der Prolog–Datenbasis mindestens eine Regel definiert, die die Auswertung des betreffenden Prädikates beschreibt. Besitzt ein vordefiniertes Prädikat Parameter, die beim Aufruf wahlweise freie Variablen oder feste Werte enthalten können, so existiert für jede dieser Möglichkeiten eine entsprechende Auswertungsregel. Bei der Abarbeitung des Bedingungsteils der Auswertungsregel eines IDAX–Prädikates erfolgt immer eine Anfrage an die dynamische Wissensbasis, um von dort die Gültigkeit der Aussage zu erfragen, die von dem IDAX–Prädikat repräsentiert wird. Eine solche Anfrage erfolgt, indem der Prolog–Interpreter über die Schnittstelle eine Smalltalk–Methode, also eine Art Unterprogramm, aufruft, das die gestellte Frage aufgrund der Fakten in der dynamischen Wissensbasis beantwortet. Zu jeder Auswertungsregel gehört genau eine bestimmte Methode im Smalltalk–Code von IDAX, so daß bei der Auswertung von IDAX–Prädikaten Prolog–Regeln und Smalltalk–Methoden — bis auf wenige Ausnahmen — stets als Paar eingesetzt werden.

Grundsätzlich lassen sich zwei Arten von Anfragen an die dynamische Wissensbasis unterscheiden, auf die auch in verschiedener Weise geantwortet wird:

- Es wird nach der Gültigkeit einer Bedingung, also der Existenz eines Faktums in der dynamischen Wissensbasis, gefragt. Beispielsweise könnte eine Anfrage lauten: „Ist der ausgewählte Prozessor ein 486er–Chip?“. Die Antwort auf eine solche Frage lautet immer „ja“ oder „nein“.
- Es wird nach dem Namen eines Objektes oder dem Wert eines Attributs gefragt, d.h. nicht nach der Existenz eines Faktums, sondern nach dem Faktum selbst. Eine solche Anfrage wäre zum Beispiel: „Welche Taktfrequenz verträgt der ausgewählte Prozessor?“. Als Antwort auf eine solche Frage wird der Wert oder Name zurückgegeben, der das entsprechende IDAX–Prädikat erfüllt. Es kann aber auch vorkommen, daß es keinen oder mehrere entsprechende Werte gibt, so daß auch diese Möglichkeiten bei der Weiterverarbeitung der Antwort in Prolog berücksichtigt werden müssen.

Bei beiden Anfrage–Typen gehört zur vollständigen Rückgabe an Prolog allerdings noch ein zweiter Teil. Denn außer der oben beschriebenen Antwort müssen auch noch diejenigen Knoten aus dem Abhängigkeitsnetzwerk des TMS ermittelt werden, die die Gültigkeit der Antwort garantieren. Bei der Auswertung eines IDAX–Prädikates wird also aus der dynamischen Wissensbasis immer ein Paar von Werten ermittelt, bestehend aus

1. der Antwort „ja" oder „nein" bzw. keinem, einem oder mehreren Fakten
2. einem oder mehreren TMS–Knoten.

Hat der Prolog–Interpreter einen Beweis erfolgreich abgeschlossen, so kann aus der Menge der zurückgelieferten TMS–Knoten eine Begründung erzeugt werden, die sich auf die Gültigkeit aller im Beweis verwendeten Fakten aus der dynamischen Wissensbasis stützt. Diese Begründung wird dann zur Rechtfertigung der Auswahlentscheidung genutzt, die aufgrund des Beweises getroffen wird.

## 4.4 Ablauf einer Operatorauswahl mit Präferenzregeln

Wenn die Kontrollkomponente eine Konfigurationsaufgabe bearbeitet, für die mehrere alternative Operatoren zur Auswahl stehen, so gibt es drei Möglichkeiten, eine Entscheidung zu treffen, nämlich entweder durch eine Eingabe des Benutzers, durch zufällige Selektion einer Alternative oder durch Auswertung von Präferenzregeln. Der Anwender kann bestimmen, welches dieser drei Auswahlverfahren angewendet werden soll. Insbesondere kann er die Komponente zur Auswahl von Präferenzregeln, die als weitgehend unabhängiges Modul implementiert ist, jederzeit aktivieren oder deaktivieren. Jedesmal, wenn eine Operatorauswahl mit Hilfe von Präferenzregeln durchgeführt werden soll, müssen die in den folgenden Unterabschnitten beschriebenen Schritte, die in Abbildung 4.10 zusammengefaßt sind, der Reihe nach abgearbeitet werden.

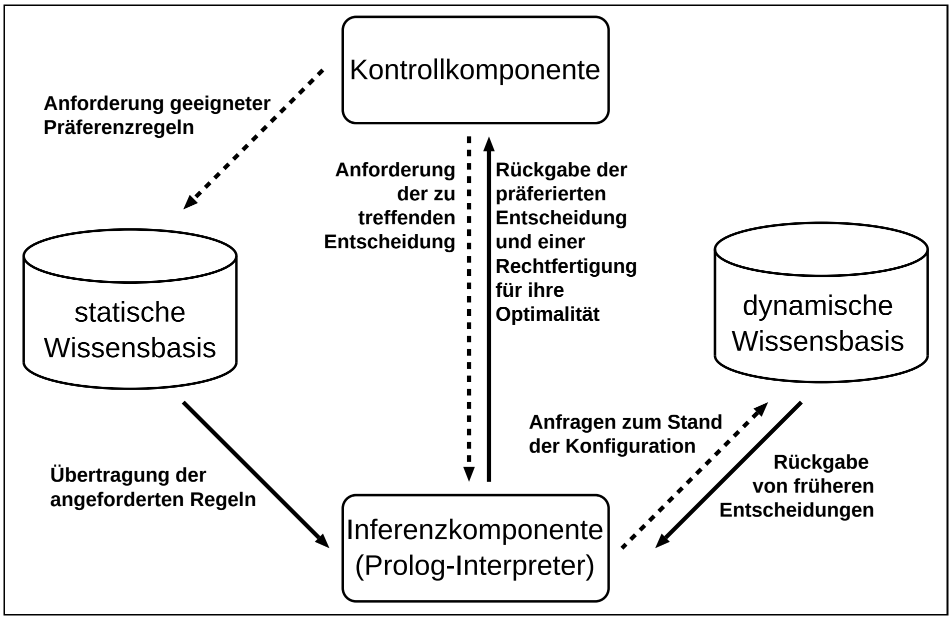


Abbildung 4.10: Schematische Darstellung einer Operatorauswahl mit Präferenzregeln

### 4.4.1 Auswahl geeigneter Regeln

Die konkrete Konfigurationsaufgabe bezieht sich immer auf ein bestimmtes Objekt der dynamischen Wissensbasis, das seinerseits Instanz eines Objektes der statischen Wissensbasis ist. Den Regelsatz dieses Domänenobjektes durchsucht die Kontrollkomponente nach Präferenzregeln, die sich auf die zu bearbeitende Konfigurationsaufgabe beziehen. Diese lassen sich anhand des ersten Parameters im Regelkopf ermitteln, der einen der folgenden Werte annehmen kann:

- ValueSelection zur Auswahl eines Attributwertes
- CardinalitySelection zur Auswahl der Kardinalität einer Komponente
- ObjectRefinement zur Auswahl zwischen alternativen Verfeinerungsrelationen
- ObjectSpecialization zur Auswahl einer Spezialisierung

Mit diesen Bezeichnern lassen sich alle Konfigurationsaufgaben innerhalb des IDAX–Systems spezifizieren, für die es beim gegenwärtigen Stand der Implementierung alternative Operatoren geben kann. Bei Präferenzregeln zur Auswahl eines Attributwertes oder einer Kardinalität ist im Regelkopf zusätzlich der Name des betreffenden Attributes bzw. der Komponente anzugeben.

**Beispiel:**

Die Regelmenge des Objektes PC enthalte die beiden folgenden Regeln:

```
prefer(ObjectRefinement, self, HasParts) :-
    not(selectedValue(self, Preisklasse, varX)).
prefer(ObjectSpecialization, self, 'Pentium-PC') :-
    selectedValue(self, Preisklasse, 'hoch').
```

Um die Aufgabe „Spezialisiere das Objekt PC“ zu bearbeiten, werden nur diejenigen Regeln ausgewählt, deren erster Parameter im Regelkopf den Wert ObjectSpecialization enthält, in diesem Beispiel also nur die zweite Regel.

### 4.4.2 Instanziierung der ausgewählten Regeln

Die ausgewählten Präferenzregeln müssen entsprechend der aktuellen Situation instanziiert werden. Dazu sind die folgenden Schritte erforderlich:

- Die Pseudovariable self bezeichnet dasjenige Objekt der dynamischen Wissensbasis, auf das sich die aktuelle Konfigurationsaufgabe bezieht. Die Variable self wird bei der Instanziierung durch den eindeutigen Namen dieses Objektes ersetzt.
- Es wird eine zusätzliche Variable als erster Parameter in den Regelkopf eingefügt, die bei der Abarbeitung der Regel mit dem offenen Ziel unifiziert wird, das zur Zeit bearbeitet wird. Dieses Ziel wird benötigt, damit Prolog dasjenige Smalltalk–Objekt identifizieren kann, an das eventuelle Rückfragen während des Beweises gestellt werden können.
- Es muß während des gesamten Beweises der Präferenzregel eine Liste mitgeführt werden, um für alle aus der dynamischen Wissensbasis abgefragten Fakten die zugehörigen TMS–Knoten abspeichern zu können. Dazu werden bei jedem IDAX–Prädikat (und anderen vordefinierten Prädikaten) zwei zusätzliche Parameter eingeführt, von denen der erste die Liste der bislang gespeicherten TMS–Knoten als Eingabe erwartet. Alle während des Beweises des Prädikates referenzierten TMS–Knoten werden an diese Liste angehängt, die dann im zweiten Parameter zurückgeliefert wird. Diese Parameter werden gemäß dem folgenden Schema verkettet, um eine Schachtelung von Regeln zu ermöglichen:

  prefer( ..., Variable$_0$, Variable$_n$) :-
  idax-prädikat$_1$( ..., Variable$_0$, Variable$_1$),
  idax-prädikat$_2$( ..., Variable$_1$, Variable$_2$),
  ⋮
  idax-prädikat$_n$( ..., Variable$_{n-1}$, Variable$_n$).

**Beispiel:**

Die ausgewählte Regel

```
prefer(ObjectSpecialization, self, 'Pentium-PC') :-
    selectedValue(self, Preisklasse, 'hoch').
```

wird instanziiert zu

```
prefer(varGoal, ObjectSpecialization, 'PC', 'Pentium-PC', var0, var1) :-
    selectedValue(varGoal, 'PC', Preisklasse, 'hoch', var0, var1).
```

wobei 'PC' der eindeutige Name des dynamischen Objekts, varGoal der Parameter für das anfragende Ziel und var0, var1 die Parameter für die Liste der TMS–Knoten sind. Diese instanziierte Darstellung der Regel dient nur zur internen Übergabe an die Prolog-Schnittstelle und bleibt für den Benutzer unsichtbar.

### 4.4.3 Übertragung der Regeln nach Prolog

Die instanziierten Regeln werden über die Schnittstelle in die Datenbasis des Prolog–Servers übertragen. Dazu wird eine modifizierte Version des asserta–Prädikates von Prolog verwendet, die dafür sorgt, daß alle mit var beginnenden Parameter in freie Prolog–Variablen umgewandelt werden, da eine direkte Übertragung von Variablen über die Schnittstelle nicht möglich ist. Alle anderen Parameter werden von ihr automatisch in Prolog–Konstanten übersetzt.

**Beispiel:**

Die im vorangehenden Beispiel instanziierte Regel steht nach der Übertragung in folgender Form in der Prolog–Datenbasis:

```
prefer(A, aObjectSpecialization, aPC, aPentiumPC, B, C) :-
    selectedValue(A, aPC, aPreisklasse, ahoch, B, C).
```

Dabei werden gemäß der Prolog-Syntax alle Bezeichner, die mit einem großen Buchstaben beginnen, als Variablen aufgefaßt und alle anderen als Konstanten. Da der Benutzer keinen unmittelbaren Zugriff auf die Prolog-Datenbasis hat, bleibt die obige Darstellung der Regeln in Prolog ihm gegenüber verborgen.

### 4.4.4 Aufruf der Regelauswertung

Nachdem die Präferenzregeln in die Prolog–Datenbasis übertragen worden sind, startet die Kontrollkomponente ihre Auswertung, indem sie über die Schnittstelle eines der beiden Anfrageprädikate findBestOp oder bestOp an den Prolog–Interpreter schickt. Beide erhalten dabei folgende Parameter:

- das offene Ziel, das die gerade bearbeitete Aufgabe repräsentiert
- den Bezeichner, der den entsprechenden Aufgaben–Typ beschreibt
- den eindeutigen Namen des dynamischen Objektes, auf das sich die Aufgabe bezieht
- gegebenenfalls den Namen des Attributs oder der Komponente, auf die sich die Aufgabe bezieht

Der Unterschied zwischen beiden Anfrageprädikaten ist, daß findBestOp als Antwort den präferierten Operator zurückliefern soll, wohingegen dem Prädikat bestOp als zusätzlicher Parameter ein bestimmter Operator mitgegeben wird, dessen Optimalität überprüft werden soll.

**Beispiel:**

Die folgende Anfrage sucht nach dem präferierten Operator für die in den bisherigen Beispielen betrachtete Aufgabe:

findBestOp(*SpecializeGoal*, ObjectSpecialization, 'PC')

Dagegen würde die nächste Anfrage dazu führen, daß nach einem Beweis für die Optimalität der Auswahl des Wertes 'hoch' für das Attribut Preisklasse gesucht wird:

bestOp(*AttributeGoal*, ValueSelection, 'PC', Preisklasse, 'hoch')

### 4.4.5 Regelauswertung in Prolog

In der Datenbasis des Prolog–Interpreters sind außer den in den vorangehenden Schritten übertragenen Präferenzregeln bereits verschiedene Prädikate vordefiniert, die die korrekte Abarbeitung der Präferenz–Prädikate, die in Abschnitt 5.1 beschrieben werden, gewährleisten. Während der Prolog–Interpreter nach dem Beweis für ein Präferenz–Prädikat sucht, also die Gültigkeit der Bedingungen einer oder mehrerer Präferenzregeln zu beweisen versucht, müssen die darin enthaltenen IDAX–Prädikate ausgewertet werden. Deren Auswertung führt dazu, daß Rückfragen an die dynamische Wissensbasis gestellt werden. Dazu wird an das offene Ziel, das beim Aufruf der Regelauswertung angegeben wurde, diejenige Methode geschickt, die in der Auswertungsregel des jeweiligen IDAX–Prädikates referenziert ist. Als Resultat dieser Methode wird das gewünschte Faktum aus der dynamischen Wissensbasis sowie der zugehörige TMS–Knoten an den Prolog–Server zurückgeliefert.

**Beispiel:**

Die folgende Regel ist ein Teil der Definition des IDAX–Prädikates selectedValue, das für den Beweis der Regel aus dem obigen Beispiel benötigt wird.

```
selectedValue(Goal, Object, Attribute, Value, InputList, OutputList) :-
    nonvar(Value),
    st(Goal, selectedValueOf:Object, attribute:Attribute, is:Value, Justification),
    append(InputList, [Justification], OutputList).
```

Ohne die technischen Details zu beachten, kann man an dieser Definition erkennen, wie mit dem Prädikat st die Smalltalk–Methode selectedValueOf:attribute:is: aufgerufen wird. Diese Methode muß bei dem im Parameter Goal angegebenen Ziel definiert sein. Sie überprüft das Vorhandensein eines bestimmten Attributwertes und liefert im Erfolgsfall den zugehörigen TMS–Knoten als Rechtfertigung zurück. Dieser wird dann an die Liste der im Beweis benötigten TMS–Knoten angehängt.

Ist es gelungen, die Präferenz eines Operators zu beweisen, wird dieser Operator zusammen mit der Liste aller zur Rechtfertigung seiner Optimalität benötigten TMS–Knoten vom Prolog–Server an die Kontrollkomponente zurückgegeben. Andernfalls wird ihr signalisiert, daß keine gültige Präferenz vorliegt.

### 4.4.6 Entfernen der nicht mehr benötigten Präferenzregeln

Nach Abschluß der Regelauswertung entfernt die Kontrollkomponente alle zuvor eingefügten Präferenzregeln wieder aus der Prolog–Datenbasis. Dieser Schritt ist für die korrekte Funktion der Regelverarbeitung nicht erforderlich, da jede Präferenzregel durch die zuvor erfolgte Instanzierung nicht fälschlicherweise bei der Regelauswertung für eine andere Konfigurationsentscheidung bewiesen werden kann. Die Regeln werden nur gelöscht, um die Prolog–Datenbasis klein zu halten, was die Suche nach geeigneten Präferenzregeln beschleunigt. Dieser Effizienzvorteil geht aber verloren, wenn — auf Grund von Backtracking oder Optimaliätsverlusten — mehrfach Präferenzbeweise für dieselbe Konfigurationsaufgabe geführt werden müssen. In diesem Fall müssen nämlich die benötigten Präferenzregeln vor jedem Beweis erneut in die Prolog–Datenbasis übertragen werden. Ob es effizienter ist, die Präferenzregeln in der Prolog–Datenbasis zu belassen oder nach jedem Auswertungszyklus wieder zu löschen, läßt sich deswegen nur nach Analyse der jeweiligen Anwendungsdomäne entscheiden.

# Kapitel 5

# Aufbau der Präferenzregeln

In diesem Kapitel werden die Prädikate beschrieben, aus denen sich Präferenzregeln aufbauen lassen. Neben den Präferenz–Prädikaten, mit denen die gewünschte Präferenzrelation beschrieben wird, und den IDAX–Prädikaten, die Fakten der dynamischen Wissensbasis darstellen, gehören dazu auch bestimmte Standard–Prädikate der Sprache Prolog sowie einige Kontrollstruktur–Prädikate, die es erlauben, spezielle Bedingungen zu formulieren, die zur Beschreibung von Präferenzen innerhalb der Konfiguration erforderlich sind. Außerdem wird der Begriff des *Aspektes* eingeführt, mit dem sich wiederkehrende Vorbedingungen und abstrakte Anforderungen des Benutzers auf natürliche Weise repräsentieren lassen.

Der im folgenden vorgestellte Satz der IDAX–Prädikate ist insofern vollständig, als er den Zugriff auf alle elementaren Entscheidungen, wie zum Beispiel die Auswahl eines Attributwertes, innerhalb des aktuellen Konfigurationsprozesses erlaubt. Darüberhinaus kann aber auch auf abgeleitete Fakten zugegriffen werden, wie beispielsweise die Zulässigkeit eines Attributwertes, die möglicherweise von mehreren Konfigurationsentscheidungen abhängt. Da sich aus der dynamischen Wissensbasis eine Vielzahl verschiedener Arten von Fakten ableiten läßt, muß in Abhängigkeit von der jeweiligen Domäne entschieden werden, für welche dieser abgeleiteten Fakten spezielle IDAX–Prädikate definiert werden müssen. Gleiches gilt auch für die zusätzlichen Kontrollstruktur–Prädikate, von denen hier nur zwei für die Konfiguration allgemein sinnvoll erscheinende vorgestellt werden. Es hat sich aber bereits während der Implementierung des jetzigen Satzes von Prädikaten gezeigt, daß eine Anpassung an geänderte Anforderungen mit wenig Aufwand möglich ist.

Für die Notation der vordefinierten Prädikate gelten für die kommenden Abschnitte diese Vereinbarungen:

- Parameter, die vollständig in kleinen Buchstaben geschriebenen sind, müssen beim Aufruf des Prädikates mit einem Wert belegt sein, sind also Eingabe–Parameter.
- Paramter, die vollständig in großen Buchstaben geschrieben sind, müssen beim Aufruf des Prädikates mit einer freien Variablen belegt sein.
- Paramter, die lediglich mit einem Großbuchstaben beginnen, können beim Aufruf sowohl an einen Wert als auch an eine freie Variable gebunden sein. Sie stellen je nachdem entweder Ein- oder Ausgabeparameter dar.
- Parameter, die in eckigen Klammer stehen, sind fakultativ.

## 5.1 Präferenz–Prädikate

Es gibt zwei Prädikate, mit denen im Kopf der Regel festgelegt wird, welche Präferenzrelation durch die Regel ausgedrückt wird:

1. prefer(operation, object, [attribute], Choice)

   erwartet die folgenden Parameter:

   **operation** beschreibt den Typ der Konfigurationsaufgabe, auf die die Regel angewendet werden kann. Hier ist einer derjenigen Bezeichner anzugeben, die im Abschnitt 4.4.1 aufgezählt sind.

   **object** muß den eindeutigen Namen des Objektes in der dynamischen Wissensbasis enthalten, auf das sich die Konfigurationsaufgabe bezieht. Bei der Definition der Regeln in der statischen Wissensbasis wird hier stets die Pseudovariable self eingesetzt, die dann bei der Instanziierung der Regel durch den Namen des dynamischen Objektes ersetzt wird.

   **attribute** : Falls die Konfigurationsaufgabe ValueSelection gewählt ist, wird hier der Name des zu belegenden Attributes übergeben. Falls die Konfigurationsaufgabe CardinalitySelection gewählt ist, wird der Name der Komponente übergeben, deren Kardinalität festgelegt werden soll. Bei allen anderen Konfigurationsaufgaben entfällt der Parameter.

   **Choice** enthält die präferierte Entscheidungsalternative. Hier kann bei der Regeldefinition auch eine Variable eingesetzt werden, die aber dann im Bedingungsteil der Regel mit einem Wert belegt werden muß.

   Das prefer–Prädikat kann dann verwendet werden, wenn in Abhängigkeit vom Stand des Konfigurationsprozesses, dessen relevante Fakten im Bedingungsteil der Regel beschrieben werden, genau eine Alternative bevorzugt wird. Damit die Präferenz eindeutig ist, müssen sich die Vorbedingungsteile mehrerer prefer–Regeln gegenseitig ausschließen, so daß immer höchstens eine Regel bewiesen werden kann. Wird dies bei der Regeldefinition nicht beachtet, so wird wegen der Beweisstrategie von Prolog diejenige Alternative ausgewählt, deren Präferenzregel zufällig als erste in der Prolog–Datenbasis steht.

2. preferTo(operation, object, [attribute], choice, second_choice)

   drückt die Bevorzugung einer Alternative gegenüber einer anderen aus und erwartet neben den Parametern, die vom Prädikat prefer bekannt sind, den zusätzlichen Parameter second_choice, in dem die weniger bevorzugte Alternative angegeben wird. Für beide Parameter müssen bei der Definition einer Regel konstante Werte festgelegt werden.

   Die Auswertung der preferTo–Regeln ist sehr viel aufwendiger als bei den einfachen prefer–Regeln: Es soll diejenige Alternative gefunden werden, die gegenüber allen anderen Alternativen bevorzugt wird. Dabei werden aber nicht alle Alternativen berücksichtigt, sondern nur solche, die erstens zulässig sind und zweitens im Kopf mindestens einer preferTo–Regel auftauchen. Diese zweite Bedingung führt dazu, daß die Konfliktmenge auf eine wahrscheinlich kleinere, in jedem Fall aber endliche Teilmenge eingeschränkt wird. Erst dadurch wird es möglich, preferTo–Regeln auch auf Aufgaben mit unendlicher Konfliktmenge, wie die Wahl des Wertes eines Attributes mit kontinuierlichem Wertebereich, anzuwenden. Aber auch bei großen, endlichen Konfliktmenge ergeben sich durch dieses Vorgehen die Vorteile, daß erstens weniger preferTo–Regeln definiert werden müssen und zweitens die Präferenzbeweise dadurch kürzer werden, wenn sich die Präferenz auf einige wenige Alternativen bezieht. Auf dieser eingeschränkten Konfliktmenge wird nun versucht, für eine Alternative zu beweisen, daß sie gegenüber allen anderen Alternativen bevorzugt wird, wobei für diesen Beweis der transitive Abschluß aller gültigen preferTo–Regeln verwendet wird.

Dieses Vorgehen ermöglicht es auch, eine vollständige Begründung für die Optimalität der Entscheidung zu liefern. In diese Begründung gehen alle Fakten ein, die erforderlich waren, um die Bevorzugung der getroffenen Entscheidung gegenüber allen zulässigen Alternativen der eingeschränkten Konfliktmenge zu beweisen. Darüber hinaus werden die Zurückweisungen aller unzulässigen Alternativen der eingeschränkten Konfliktmenge aufgenommen, da diese Alternativen eventuell besser sein könnten als die ausgewählte Entscheidung.

Abweichend von der hier vorgestellten Interpretation der preferTo–Regeln, nämlich daß eine Entscheidung genau dann ausgewählt wird, wenn sie gegenüber allen anderen zulässigen Alternativen bevorzugt wird, könnte man auch festlegen, eine Entscheidung immer dann auszuwählen, wenn es keine bessere zulässige Alternative gibt. In diesem Fall wäre es aber nicht möglich, eine vollständige Begründung für die Optimalität der Auswahl zu erzeugen. Ein entsprechender Beweis würde nämlich darauf gründen, daß keine Präferenzregel gilt, die eine andere Alternative der gewählten gegenüber bevorzugt. Aus dem Scheitern von Regeln lassen sich aber keine Begründungen ableiten, wie im Abschnitt 6.1.2 detailliert ausgeführt wird.

**Beispiel:**

Die folgenden beiden preferTo–Regeln beschreiben die Präferenz eines möglichst leistungsfähigen Prozessors ohne Abhängigkeit von weiteren Bedingungen:

```
preferTo(ObjectSpecialization, 'CPU', 'Pentium', 'CPU-486').
preferTo(ObjectSpecialization, 'CPU', 'CPU-486', 'CPU-386').
```

Wegen der Transitivität des Präferenzkonzepts wird durch diese Regeln auch ausgedrückt, daß ein 'Pentium' gegenüber einer 'CPU-386' zu bevorzugen ist. Stünden außer den drei explizit erwähnten Prozessortypen noch weitere zur Auswahl, so würden diese bei der Auswertung der Präferenzregeln nicht berücksichtigt.

Mit dem preferTo–Prädikat läßt sich leicht eine Präferenzordnung über der Konfliktmenge oder einer Teilmenge derselben definieren. Auch läßt sich von mehreren solchen Ordnungen über den Bedingungsteil der preferTo–Regeln diejenige auswählen, die zum aktuellen Stand des Konfigurationsprozesses paßt. Wegen der aufwendigen Beweise, die erforderlich sind, um die beste Alternative zu ermitteln, sollte die Zahl der preferTo–Regeln möglichst klein gehalten werden und — wenn möglich — das prefer–Prädikat verwendet werden. Werden beide Prädikate gemeinsam in einem Regelsatz benutzt, was inhaltlich nur selten sinnvoll sein dürfte, ist zu beachten, daß die prefer–Regeln bevorzugt ausgewertet werden.

## 5.2 IDAX–Prädikate

### 5.2.1 Produkt–Prädikate

Die im folgenden aufgeführten Prädikate erlauben es, Aussagen über den Zustand der aktuellen Teilkonfiguration zu machen. Diese Teillösung wird in einem speziellen Teil der dynamischen Wissensbasis, nämlich der Produkt–Datenbank, verwaltet. Auf diese Produkt–Datenbank — und wenn nötig auch auf andere Teile der dynamischen Wissensbasis — greifen die Produkt–Prädikate zu. Sie liefern dabei nicht nur ein Faktum aus der Wissensbasis zurück, sondern als Rechtfertigung für ihre Gültigkeit auch denjenigen Knoten aus dem Abhängigkeitsnetzwerk des Truth–Maintenance–Systems, der die Gültigkeit des betreffenden Faktums repräsentiert. Die Produkt–Prädikate sind die wichtigsten Prädikate, die zur Formulierung der Bedingungsteile von Regeln benötigt werden.

In einem wichtigen Punkt unterscheidet sich der Zugriff auf das Produkt, wie er über die Produkt–Prädikate erfolgt, von einem direkten Zugriff auf die Strukturen der dynamischen Wissensbasis: Wie bereits erwähnt, lassen sich Spezialisierung und Dekomposition in IDAX nur alternativ anwenden. Bei jedem Spezialisierungsschritt wird dabei eine neue Instanz eines Domänenobjektes in der dynamischen Wissensbasis erzeugt. Deswegen kann ein einzelnes Objekt der Domäne infolge von Spezialisierungen gleichzeitig durch eine Mehrzahl von Objekten in der dynamischen Wissensbasis repräsentiert werden. Da das Produkt keine Information über prozedurale und strukturelle Aspekte des Konfigurationsverlaufs enthalten soll, muß die interne Repräsentation eines nach außen hin einheitlichen Objektes durch mehrere dynamische Instanzen beim Zugriff auf das Produkt verborgen werden. Dies geschieht an zwei Stellen:

- Wenn auf ein Attribut eines dynamischen Objekts zugegriffen wird, so wird dieses Attribut nicht nur bei dem Objekt selbst, sondern auch bei allen seinen Generalisierungen in der dynamischen Wissensbasis gesucht.
- Wenn auf die Zerlegung eines dynamischen Objektes insgesamt oder auf eine einzelne Komponente zugegriffen wird, so wird nach einem entsprechenden Zerlegungsschritt nicht nur bei dem Objekt selbst, sondern auch bei allen seinen Spezialisierungen in der dynamischen Wissensbasis gesucht.

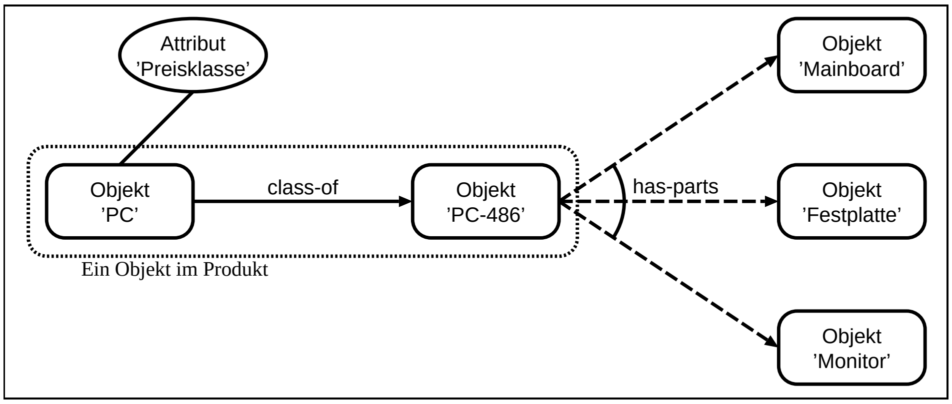


Abbildung 5.1: Vereinigung von Spezialisierungen zu einem Objekt innerhalb des Produktes

Diese Art des Zugriffs auf das Produkt läßt alle durch Spezialisierung innerhalb der dynamischen Wissensbasis entstandenen Instanzen nach außen hin wieder zu einem einzigen Objekt verschmelzen. Unmittelbare Folge dieser Sichtweise ist, daß ein Objekt innerhalb des Produktes auch gleichzeitig durch Zerlegung und durch Spezialisierung verfeinert sein kann.

**Beispiel:**

Abbildung 5.1 zeigt einen Ausschnitt aus einer dynamischen Wissensbasis, in der das Objekt PC erst zu PC-486 spezialisiert und diese Spezialisierung dann zerlegt worden ist. In den Strukturen der dynamischen Wissensbasis ist die Information über Attribute und Verfeinerungsrelationen disjunkt auf die beiden Instanzen PC und PC-486 verteilt. Im Produkt werden diese beiden Objekte aber als Einheit gesehen, so daß beim Zugriff auf PC auch die Zerlegung und beim Zugriff auf PC-486 auch das Attribut Preisklasse sichtbar ist.

**Prädikate, die auf Attributwerte zugreifen**

Folgende vier Prädikate stehen zur Verfügung, um den Wert eines Attributes zu beschreiben:

1. selectedValue(object, attribute, Value)
2. rejectedValue(object, attribute, Value)
3. optimalValue(object, attribute, Value)
4. possibleValue(object, attribute, Value)

Alle vier Prädikate erwarten dieselben drei Parameter:

object : Der eindeutige Name einer Objektinstanz in der dynamischen Wissensbasis

attribute : Der Name eines Attributes des Objektes

Value : Ein konkreter Wert aus dem Wertebereich des Attributes oder eine freie Variable.

Es gibt zwei Möglichkeiten, die Attribut–Prädikate aufzurufen:

**Geschlossener Aufruf:** Wenn im Parameter Value ein konkreter Wert übergeben wird, so wird überprüft, ob dieser Wert das Prädikat erfüllt. Wenn dies der Fall ist, hat der Aufruf Erfolg, andernfalls scheitert er.

**Offener Aufruf:** Wenn im Parameter Value eine freie Variable übergeben wird, so wird nach einem Wert gesucht, der das Prädikat erfüllt. Gibt es einen solchen, wird die Variable an ihn gebunden, und der Aufruf ist erfolgreich. Gibt es keinen solchen Wert, so scheitert der Aufruf.

Das Prädikat selectedValue ist erfüllt, wenn Value der für das Attribut ausgewählte Wert ist. Als Rechtfertigung wird im Erfolgsfall derjenige TMS–Knoten aus der dynamischen Wissensbasis geliefert, der die Zuweisung des betreffenden Attributwertes innerhalb der Produkt-Datenbank repräsentiert.

Das Prädikat rejectedValue ist erfüllt, wenn der Wert Value durch eine Zurückweisung aus der Konfliktmenge ausgeschlossen ist. Als Rechtfertigung wird im Erfolgsfall derjenige TMS–Knoten geliefert, der die Entscheidung repräsentiert, den betreffenden Wert zurückzuweisen.

Da mehrere Werte gleichzeitig zurückgewiesen sein können, liefert ein offener Aufruf kein eindeutiges Resultat, es wird irgendeiner der zurückgewiesenen Werte als Ergebnis zurückgeliefert. Scheitert der Beweisversuch mit diesem Rückgabewert, so wird Prolog im Rahmen seiner Backtracking–Strategie das Prädikat rejectedValue erneut aufrufen. In diesem Fall werden solange alternative Lösungen für das Prädikat zurückgeliefert, bis die Liste zurückgewiesener Werte erschöpft ist. In diesem Fall scheitert rejectedValue endgültig. Das Backtracking findet dabei nur innerhalb des Prolog–Interpreters statt, da von der dynamischen Wissensbasis schon bei der ersten Anfrage alle möglichen Lösungen an Prolog übergeben werden. Diese werden im Verlauf der Regelauswertung in einer Liste gespeichert, aus der dann im Falle des Backtrackings sukzessive die alternativen Werte entnommen werden können. Dieses Vorgehen wird in gleicher Weise bei allen anderen IDAX–Prädikaten praktiziert, die alternative Lösungen liefern können.

Das Prädikat optimalValue ist erfüllt, wenn Value der derzeit ausgewählte Attributwert ist und zusätzlich eine gültige Rechtfertigung für seine Optimalität besitzt. In diesem Fall wird derjenige TMS–Knoten als Rechtfertigung zurückgeliefert, der die Optimalität des gewählten Wertes repräsentiert. Es ist nicht möglich, Aussagen über die Optimalität von Attributwerten zu machen, die nicht aktuell ausgewählt sind, da für diese Werte im allgemeinen keine Optimalitäts–Rechtfertigungen erzeugt werden.

Das Prädikat possibleValue ist erfüllt, wenn noch kein Attributwert ausgewählt ist und Value sich in der gültigen Konfliktmenge befindet, also nicht zurückgewiesen ist. Als Rechtfertigung müssen in diesem Fall zwei TMS–Knoten zurückgeliefert werden, nämlich derjenige, der den Zustand repräsentiert, daß die Aufgabe, einen Attributwert zu wählen, noch unbearbeitet ist, und zusätzlich derjenige, welcher darstellt, daß es für Value keine gültige Zurückweisung gibt. Auch bei diesem Prädikat kann der offene Aufruf über Backtracking alternative Lösungen liefern, d.h. die gesamte Konfliktmenge aufzählen. Das Prädikat possibleValue gilt zusätzlich dann als erfüllt, wenn Value das Prädikat selectedValue erfüllt.

Die Aufzählung alternativer Lösungen durch einen offenen Aufruf der Prädikate rejectedValue und possibleValue ist bei Attributen mit kontinuierlichem Wertebereich nicht möglich, da sich weder ihre Konfliktmenge noch die möglicherweise durch Constraints ausgeschlossenen Werte aufzählen lassen. Daher ist der offene Aufruf in diesem Fall nicht zulässig. Es gibt allerdings ein zusätzliches Prädikat speziell für Attribute mit kontinuierlich–numerischem Wertebereich:

5. possibleValueBetween(object, attribute, lower_limit, upper_limit)

Das Prädikat possibleValueBetween ist erfüllt, wenn es noch mindestens einen zulässigen Wert innerhalb des Intervalls gibt, das durch die Grenzen lower_limit und upper_limit definiert ist. Um auch hier eine Optimalitäts–Rechtfertigung zu erhalten, muß im Erfolgsfall für einen beliebigen zulässigen Wert innerhalb des gegebenen Intervalls ein TMS-Knoten erzeugt werden, der die Tatsache repräsentiert, daß der betreffende Wert nicht zurückgewiesen ist. Dieser TMS-Knoten ist dann eine hinreichende Bedingung für die Gültigkeit des Prädikates, denn solange dieser eine Wert nicht zurückgewiesen wird, gibt es mindestens einen zulässigen Wert innerhalb des vorgegebenen Intervalls.

### Prädikate, die auf Objekt–Relationen zugreifen

Die in diesem Abschnitt beschriebenen Prädikate erlauben die Beschreibung der durch Dekomposition und Spezialisierung bestimmten Struktur des Produkts. Dazu ist ein Zugriff auf die hasParts– und die classOf–Relation zwischen den dynamischen Objekten erforderlich. Dieser Zugriff kann auch über mehrere Stufen der Relationen–Hierarchie — also transitiv — erfolgen, woraus sich unmittelbar die folgenden Prädikate ergeben:

1. hasPart(Super_object, Edge, Sub_object)
2. hasPartTrans(Super_object, Sub_object)
3. classOf(Super_object, Sub_object)
4. classOfTrans(Super_object, Sub_object)

Alle vier Prädikate erwarten zwei Parameter:

Super_object : Entweder der eindeutige Bezeichner des Objektes, dessen Komponenten oder Spezialisierungen bestimmt werden sollen, oder eine freie Variable.

Sub_object : Entweder der eindeutige Bezeichner des Objektes, dessen Aggregat oder Generalisierung bestimmt werden soll, oder eine freie Variable.

Da die Kanten der hasParts–Relation eigene Namen tragen, mit denen sich gezielt eine bestimmte Komponente ansprechen läßt, erwartet das hasPart–Prädikat als weiteren Parameter

Edge : Entweder der Name einer Kante der Dekompositions–Relation oder eine freie Variable.

Es gibt drei Möglichkeiten, die Relationen–Prädikate aufzurufen:

**Geschlossener Aufruf:** Wenn sowohl in Super_object als auch in Sub_objekt (und/oder Edge) ein konkreter Objekt–Bezeichner übergeben wird, so wird überprüft, ob die beiden Objekte das Prädikat erfüllen. Wenn dies der Fall ist, hat der Aufruf Erfolg, andernfalls scheitert er.

**Nach unten offener Aufruf:** Wenn in Sub_object (und ggf. auch in Edge) eine freie Variable übergeben wird, so wird nach einer Komponente bzw. einer Spezialisierung von Super_object gesucht. Gibt es ein solches Objekt, wird die Variable an seinen Bezeichner gebunden und der Aufruf ist erfolgreich. Gibt es kein solches Objekt, so scheitert der Aufruf.

**Nach oben offener Aufruf:** Wenn dagegen in Super_object eine freie Variable übergeben wird, so wird entsprechend nach dem Aggregat bzw. der Generalisierung von Sub_object gesucht.

Die Prädikate hasPart und classOf sind erfüllt, wenn Sub_object eine unmittelbare Komponente bzw. eine unmittelbare Spezialisierung von Super_object darstellt. Als Rechtfertigung für die Gültigkeit des Prädikates wird derjenige TMS–Knoten zurückgegeben, der die Existenz des untergeordneten dynamischen Objektes repräsentiert. Die Verfeinerungsrelation selbst fließt nicht explizit in die Rechtfertigung ein, da ihre Gültigkeit unmittelbare Vorbedingung für die Existenz aller untergeordneten dynamischen Objekte ist. Ein nach unten offener Aufruf des hasPart–Prädikats kann — wie oben beschrieben — durch Backtracking alternative Lösungen generieren.

Die Prädikate hasPartTrans und classOfTrans sind erfüllt, wenn man vom Sub_object durch mehrere Aggregations– bzw. Generalisierungs–Schritte innerhalb des Produktes zum Super_object gelangt. Auch hier wird als Rechtfertigung für die Gültigkeit des Prädikates nur der TMS–Knoten benötigt, der die Existenz des Sub_object repräsentiert, da es nicht möglich ist, übergeordnete Objekte zu entfernen oder auszutauschen, ohne daß die Existenz der ihnen untergeordneten Objekte ihre Gültigkeit verliert. Wegen der Transitivität der Beweisrelation können bei beiden Prädikaten sowohl nach oben als auch nach unten offene Aufrufe zur Erzeugung alternativer Lösungen führen.

Obwohl in der statischen Wissensbasis zu jedem Objekt mehrere alternative Spezialisierungen und mehrere Generalisierungen definiert werden können, kann innerhalb des Produktes jedes dynamische Objekt höchstens eine Spezialisierung bzw. Generalisierung besitzen, weil die Spezialisierungs–Entscheidungen eindeutig sind. Dagegen können auch die dynamischen Objekte eine Vielzahl von Komponenten besitzen. Diese Tatsache ist bei der Verwendung von transitiven Relationen–Prädikaten zu berücksichtigen, da deswegen der Beweis eines hasPartTrans–Prädikats sehr aufwendig werden kann.

Wenn man feststellen will, ob ein Objekt der dynamischen Wissensbasis (mittelbar) Instanz eines Begriffes der statischen Wissensbasis ist, so läßt sich dies mit dem oben beschriebenen classOf–Prädikat nicht zweifelsfrei feststellen, da dieses nur auf den erzeugten Instanzen innerhalb der dynamischen Wissensbasis operiert. Wenn aber ein Begriff aus der statischen Wissensbasis in einem Konfigurationsprozeß instanziiert wird, so setzt dies nicht voraus, daß auch alle seine Oberbegriffe bereits instanziiert worden sind. Deshalb gibt es ein zusätzliches Prädikat, daß eine unmittelbare Verbindung zur statischen Wissensbasis herstellt:

5. isA(object, Static_object)

Dieses Prädikat erwartet im Parameter Static_object einen Begriff aus der statischen Wissensbasis und ist erfüllt, wenn object eine Instanz dieses Begriffes ist. Das isA–Prädikat kann auch offen aufgerufen werden und liefert dann (über Backtracking) alle Oberbegriffe von object aus der statischen Wissensbasis. Da sich die Instanziierungsbeziehung zwischen Objekten der statischen und dynamischen Wissensbasis während des Konfigurationsprozesses nicht mehr verändern kann, muß isA keinen rechtfertigenden TMS-Knoten zurückgeben.

#### Prädikate, die auf die Kardinalität der Zerlegung zugreifen

Für jede Komponente eines Aggregates kann durch Angabe einer Kardinalität bestimmt werden, wie oft die betreffende Komponente verwendet wird. Da eine solche Kardinalität nichts anderes als ein Attribut der Kante einer Zerlegungsrelation darstellt, fallen Kardinalitäten zwar semantisch in den Bereich der Objekt–Relationen, syntaktisch wird auf sie aber genau wie auf andere Attribute zugegriffen. Zu diesem Zweck stehen die beiden folgenden Prädikate zur Verfügung:

1. selectedCardinality(object, edge, Cardinality)
2. rejectedCardinality(object, edge, Cardinaltiy)

Diese beiden Prädikate verhalten sich völlig analog zu den Prädikaten selectedValue bzw. rejectedValue mit dem einzigen Unterschied, daß im Parameter edge nicht der Name eines Attributes, sondern der einer Kante der HasParts–Relation übergeben wird. Beide Prädikate können selbstverständlich nur erfüllt sein, wenn object entlang der Zerlegungsrelation verfeinert ist und diese Verfeinerung eine Kante des Namens edge erzeugt.

### 5.2.2 Prozeß–Prädikate

Neben dem durch die zuvor beschriebenen Prädikate realisierten Zugriff auf die Eigenschaften des Produktes kann es auch erforderlich sein, den Stand des Konfigurationsprozesses zu beschreiben. Damit ist gemeint, daß man feststellen will, welche Konfigurationsaufgaben bereits bearbeitet sind und welche noch nicht. Da Aussagen über den Verlauf des Konfigurationsprozesses nicht zum Domänenwissen gehören, sondern nur aus dem Verständnis der speziellen Funktionsweise von IDAX heraus formuliert werden können, wird ein Domänenexperte die prozeßbeschreibenden Prädikate kaum für Präferenzregeln verwenden. Diese Prädikate sind aber von entscheidender Wichtigkeit für die internen Beweise der oben aufgeführten Produkt–Prädikate. So muß beispielsweise vor dem Versuch, die Komponenten eines Objektes zu bestimmen, erst überprüft werden, ob das betreffende Objekt überhaupt entlang einer Zerlegungsrelation verfeinert wurde.

In Entsprechung zu den in IDAX vorhandenen Konfigurationsaufgaben ergeben sich die folgenden vier Prozeß–Prädikate

1. isConfigured(object)
2. isRefinedWith(object, Relation)
3. isDecomposed(object)
4. isSpecialized(object)

mit den Parametern

object : Der eindeutige Bezeichner eines dynamischen Objektes

Relation : Entweder der Name einer Verfeinerungsrelation, also HasParts oder ClassOf, oder eine freie Variable

Die Prozeß–Prädikate sind erfüllt, wenn für das in object angegebene dynamische Objekt ein entprechender Konfigurationsschritt bereits durchgeführt wurde und auch noch gültig ist. Als Rechtfertigung für die Gültigkeit eines Prozeß–Prädiaktes wird immer der TMS–Knoten ausgewählt, der die Tatsache repräsentiert, daß die zugehörige Konfigurationsaufgabe abgearbeitet ist.

Das Prädikat isRefinedWith überprüft, ob der Name der gewählten Verfeinerungsrelation mit dem Wert von Relation übereinstimmt. Das Prädikat kann auch offen aufgerufen werden und liefert dann den Namen der gewählten Verfeinerungsrelation zurück. Als Rechtfertigung gibt isRefinedWith denjenigen TMS-Knoten zurück, der die Gültigkeit der Entscheidung für die gewählte Verfeinerungsrelation darstellt.

## 5.3 Kontrollstruktur–Prädikate

### 5.3.1 Beschränkte All–Quantifizierung

Für den Beweis eines Prädikates sucht Prolog immer eine Variablenbelegung, die das Prädikat erfüllt. Prolog führt also eine implizite Existenz–Quantifizierung der freien Variablen durch. Bei der Formulierung von Präferenzregeln möchte man aber auch Bedingungen angegben können, die die Gültigkeit eines Prädikates nicht nur für eine Variablenbelegung, sondern für alle erlaubten Belegungen einer Variablen garantieren. Ein typisches Beispiel sind Bedingungen der Art: „Wenn für alle Komponenten eines Aggregates gilt...“. Um solche Bedingungen ausdrücken zu können, ist eine beschränkte All–Quantifizierung erforderlich, die durch das Prädikat

forall(generator, predicate, VARIABLE)

ausgedrückt wird. Dabei haben die Parameter folgende Bedeutung:

generator : Eines der im Abschnitt 5.2.1 aufgeführten Produkt–Prädikate, das bei einer offenen Anfrage eine Anzahl alternativer Lösungen generieren kann. Das Prädikat muß also eine freie Variable enthalten. Als Werte von generator kommen somit in Frage:

- rejectedValue(object, attribute, VALUE)
- rejectedCardinality(object, edge, CARDINALITY)
- possibleValue(object, attribute, VALUE)
- hasPart(super_object, EDGE, SUB_OBJECT)

- hasPartTrans(super_object, SUB_OBJECT)
- hasPartTrans(SUPER_OBJECT, sub_object)
- classOfTrans(super_object, SUB_OBJECT)
- classOfTrans(SUPER_OBJECT, sub_object)

predicate : Ein beliebiges IDAX–Prädikat oder eines der nachfolgend beschriebenen Standard–Prädikate, das in seiner Parameter–Liste dieselbe freie Variable enthält wie das Generator–Prädikat.

VARIABLE : Die gemeinsame freie Variable der beiden vorangehenden Prädikate

Das Prädikat forall ist erfüllt, wenn alle Bindungen für VARIABLE, die das Prädikat generator erfüllen, auch das Prädikat predicate erfüllen. Die Rechtfertigung für die Gültigkeit des forall–Prädikates setzt sich aus zwei Mengen von TMS–Knoten zusammen: Zum einen werden alle Rechtfertigungen aus den Beweisen für das Prädikat predicate gesammelt. Damit ist die Gültigkeit des Prädikates für die von generator erzeugte Menge von Lösungen gewährleistet. Zusätzlich muß aber auch sichergestellt werden, daß später nicht weitere Lösungen von generator hinzukommen, für die die Gültigkeit von predicate bislang nicht gezeigt wurde. Deshalb müssen zusätzlich TMS–Knoten in die Rechtfertigung von forall aufgenommen werden, die die Nichtexistenz weiterer möglicher Lösungen des generator–Prädikates repräsentieren, falls es solche gibt. Diese Knoten werden in der dynamischen Wissensbasis ermittelt, wenn der Prolog–Server die Anfrage nach allen gültigen Lösungen des Generator–Prädikates stellt. Werden nämlich bei der Suche nach gültigen Lösungen auch derzeit ungültige gefunden, so werden die Gründe für deren Ungültigkeit gesammelt.

**Beispiel:**

Das Prädikat

```
forall(
      hasPart('Mainboard', X, Y),
      selectedValue(Y, Taktfrequenz, 66),
      Y )
```

soll bewiesen werden. Aus der Dekompositionshierarchie der statischen Wissensbasis ist bekannt, daß Mainboard aus maximal drei Komponenten besteht, nämlich CPU, Cache und Hauptspeicher. Es muß nun geprüft werden, welche dieser Komponenten tatsächlich in der aktuellen Teilkonfiguration enthalten sind. Angenommen, die optionale Komponente Cache sei bislang nicht vorgesehen. Dann wird nur für die anderen zwei Komponenten der Wert des Attributes Taktfrequenz überprüft. In der Rechtfertigung des forall–Prädikats würden dann die folgenden drei Entscheidungen aufgenommen:

- CPU.Taktfrequenz = 66 MHz
- Hauptspeicher.Taktfrequenz = 66 MHz
- Cache nicht vorhanden

Entscheidet man sich im weiteren Verlauf der Konfiguration, doch einen Cache einzubauen, würde diese Rechtfertigung ungültig. Das ist auch erforderlich, da man schließlich noch nicht weiß, ob der geplante Cache auch die gewünschte Taktfrequenz besitzt.

### 5.3.2 Aggregation von Attributwerten

Mit der beschränkten All–Quantifizierung läßt sich unter anderem beweisen, daß bei allen Komponenten eines Aggregates der Wert eines Attributes eine bestimmte Bedingung erfüllt. Dabei werden die Attributwerte aber nur einzeln betrachtet. Oft ist man jedoch an einer gemeinsamen Betrachtung aller Werte eines Attributes interessiert, das allen Komponenten gemein ist, um zum Beispiel die Summe aller dieser Werte zu bilden. Diese Aggregation numerischer Attributwerte ist mit Hilfe des Prädikates

applyOnAll(generator, attribute, functor, RESULT)

möglich, das die folgenden Parameter erwartet:

**generator** : Ein Produkt–Prädikat, das eine Menge von Komponenten spezifiziert. Also kommen nur in Frage:

- hasPart(super_object, EDGE, SUB_OBJECT)
- hasPartTrans(super_object, SUB_OBJECT)

**attribute** : Der Name des Attributes, dessen Wert aggregiert werden soll. Das Attribut muß bei allen durch generator ausgewählten Komponenten definiert sein und einen numerischen Wertebereich besitzen.

**functor** : Ein Symbol, das die Aggregierungs–Funktion beschreibt. Es sind folgende Funktoren vorgesehen:

sum , um die Summe der Attributwerte zu bilden

max , um das Maximum der Attributwerte zu finden

min , um das Minimum der Attributwerte zu finden

count , um die Anzahl der Attributwerte zu ermitteln

**RESULT** : Eine freie Variable

Bei der Auswertung von applyOnAll werden erst alle durch den Generator bestimmten Komponenten ermittelt, genau wie bei forall. Dann wird auf alle vorhandenen Werte des angegebenen Attributes die Aggregierungs–Funktion angewendet und ihr Ergebnis im Parameter RESULT zurückgegeben. In die Rechtfertigung des applyOnAll–Prädikats gehen ein: erstens die Entscheidungen für die bekannten Attributwerte, zweitens die Nichtexistenz der unbekannten Attributwerte und drittens die eventuelle Nichtexistenz weiterer optionaler Komponenten.

**Beispiel:**

Eine Anwendung der Aggregation könnte zum Beispiel in der Kalkulation des Gesamtpreises aus den Preisen der Komponenten bestehen, auf deren Summe 30% aufgeschlagen wird:

```
prefer(ValueSelection,'PC', Verkaufspreis, varA) :-
    applyOnAll(hasPart('PC', varB, varC), Verkaufspreis, sum, varD),
    varA is (varD * 1.3).
```

## 5.4 Erweiterte Ausdrucksmöglichkeiten

### 5.4.1 Verwendung von Standard–Prädikaten

Neben den speziellen, vordefinierten Prädikaten können auch Prädikate aus dem Standard–Sprachumfang von Prolog im Bedingungsteil von Präferenzregeln verwendet werden. Zu den Standard–Prädikaten, die sich in sinnvoller Weise zur Formulierung von Bedingungen einsetzen lassen, gehören unter anderem:

- Vergleichs– und Unifikations–Prädikate (=, \=, ==, \==, ...)
- Arithmetische Prädikate (between, >, <, is, ...)
- Arithmetische Funktionen (+, -, *, /, ...)
- Das Prädikat member

Bei der Verwendung von Standard–Prädikaten wird keine Rechtfertigung für die Gültigkeit des Prädikates erzeugt, da diese Gültigkeit immer vom Ergebnis anderer IDAX–Prädikate abhängt, so daß die Rechtfertigungen aller IDAX–Prädikate bereits ausreichen, um die Gültigkeit einer Bedingung zu beweisen, in der auch Standard–Prädikate verwendet werden. Da der Mechanismus, der die Listen mit den rechtfertigenden TMS–Knoten verwaltet, nicht auf die Standard–Prädikate ausgeweitet wurde, dürfen innerhalb eines Standard–Prädikates keine IDAX–Prädikate eingebettet werden. Wenn man derartige Konstrukte benötigt, ist es unumgänglich, ein neues Prädikat zu definieren, daß sich in den Mechanismus der hier vorgestellten Regelkomponente einfügt.

**Beispiele:**

Sinnvolle Regeln mit Standard–Prädikaten könnten folgendermaßen aussehen:

```
prefer(ObjectSpecialization, 'PC', 'PC-386') :-
    selectedValue('PC', Preis, varX),
    varX < 2000.

prefer(ObjectSpecialization, 'PC', 'PC-486') :-
    selectedValue('PC', Preisklasse, varX),
    member(varX, ['mittel', 'hoch']).
```

*Nicht* zulässig, obwohl syntaktisch korrekt, wäre dagegen eine Bedingung wie

```
..., findall( varX, possibleValue('PC', Preisklasse, varX), varY), ...
```

um alle möglichen Werte für das Attribute Preisklasse zu bestimmen, weil hier das IDAX–Prädikat possibleValue in das Standard–Prädikat findall eingebettet ist.

### 5.4.2 Aspekt–Definitionen

Eine zusätzliche Möglichkeit, Bedingungen in Präferenzregeln zu formulieren, ist durch den Zugriff auf sogenannte *Aspekte* gegeben. Unter einem Aspekt wird eine abstrakte Eigenschaft verstanden, deren Vorhandensein bei einem bestimmten Objekt durch das Überprüfen einer Bedingung aus den Fakten der dynamischen Wissensbasis abgeleitet werden kann oder die durch den Benutzer explizit im Sinne einer globalen Anforderung vorgegeben wird. Aspekte lassen sich dementsprechend formulieren als

- Aspekt–Regeln der Form

  aspectOf(object, aspect_name) :-
  *Bedingung*$_1$,
  ⋮
  *Bedingung*$_n$.

- Anforderungs–Aspekte der Form

  aspect(aspect_name).

Die Definition von Aspekt–Regeln geschieht — abgesehen natürlich vom Kopf der Regel — genau wie bei Präferenzregeln. Ebenso analog ist die Instanziierung und Auswertung der Aspekt–Regeln, die ebenfalls unter Mitführung der internen Listen für die Erzeugung von TMS–Rechtfertigungen geschieht. Einziger Unterschied in der Behandlung der Aspekt–Regeln ist, daß sie nur einmal in die Prolog–Datenbasis übertragen werden und dann dort solange verbleiben, wie das dynamische Objekt, dem sie zugeordnet sind, existiert. Auf diese Weise können Aspekte in mehreren Präferenzregeln, die sich auf unterschiedliche Konfigurationsaufgaben beziehen, referenziert werden, ohne daß die Aspekt–Regeln jedesmal neu instanziiert und übertragen werden müssen. Aspekt–Regeln erlauben es also, häufig wiederkehrende Bedingungen zu einem Aspekt zusammenzufassen und so den Aufwand bei der Defintion von Regelpaketen zu verkleinern, wie das folgende Beispiel zeigt.

**Beispiel:**

```
aspectOf('PC', 'universell-einsetzbar') :-
    selectedValue('PC', CAD-Anwendung, 'ja'),
    selectedValue('PC', Textverarbeitung, 'ja'),
    selectedValue('PC', DB-Anwendung, 'ja').

preferTo(ObjectSpecialization, 'PC', 'Pentium-PC', 'PC-486') :-
    aspectOf('PC', 'universell-einsetzbar'),
    selectedValue('PC', Preisklasse, 'hoch').

preferTo(ObjectSpecialization, 'PC', 'PC-486', 'Pentium-PC') :-
    aspectOf('PC', 'universell-einsetzbar'),
    selectedValue('PC', Preisklasse, 'mittel').

preferTo(ObjectSpecialization, 'PC', 'PC-486', 'PC-386') :-
    aspectOf('PC', 'universell-einsetzbar').
```

Durch die Formulierung von abstrakten Benutzerwünschen als Anforderungs–Aspekte lassen sich in Abhängigkeit von diesen unterschiedliche Präferenzen definieren. Damit kann der Benutzer durch Eingabe eines einzigen Anforderungs–Aspektes den Konfigurationsprozeß gleichzeitig an allen Stellen beeinflussen, wo dieser Aspekt im Bedingungsteil von Regeln auftaucht. Die Möglichkeit, globale Benutzeranforderungen in verschiedenen Abstraktheitsgraden zu spezifizieren und zu verarbeiten, ist derzeit innerhalb des IDAX–Systems nicht realisiert, ist aber für die Zukunft vorgesehen und wird daher von der Regelkomponente bereits jetzt unterstützt.

**Beispiel:**

Angenommen, der Benutzer könnte einen Aspekt 'Minimal–System' als Anforderung spezifizieren. Dieser könnte dann in den unterschiedlichsten Präferenzregeln auftauchen, wie z.B.:

```
prefer(ObjectSpecialisation, 'CPU', 'CPU-386') :-
    aspect('Minimal-System').
prefer(CardinalitySelection, 'Mainboard', 'Hauptspeicher', 2) :-
    aspect('Minimal-System').
    ⋮
```

# Kapitel 6

# Überwachung der Optimalität

## 6.1 Aufbau von Optimalitäts–Rechtfertigungen

Um das in Abschnitt 3.4 beschriebene Prinzip der Pareto–Optimalität innerhalb der REDUX–Architektur nutzen zu können, müssen die Voraussetzungen, die zur Auswahl eines Operators geführt haben, in Form einer Rechtfertigung explizit gemacht werden. Voraussetzungen werden durch die Angabe von Knoten aus dem Abhängigkeitsnetz, das den Zustand der dynamischen Wissensbasis darstellt, ausgedrückt und können durch den TMS–Mechanimus entweder als gültig oder als ungültig markiert sein. Eine Rechtfertigung besteht nun gemäß der in [Doy79] eingeführten Terminologie aus zwei Listen von Voraussetzungen: Die erste, die *In–Liste*, enthält TMS–Knoten, die gültig sein müssen, um die Gültigkeit der Rechtfertigung folgern zu können, und die zweite, die *Out–Liste*, enthält diejenigen TMS–Knoten, die gleichzeitig ungültig sein müssen.

Bei der Verwendung von Präferenzregeln zur Operatorauswahl wird der Bedingungsteil der Regeln mit Hilfe der im Abschnitt 5.2 beschriebenen IDAX–Prädikate durch Anfragen an die dynamische Wissensbasis ausgewertet. Alle Bedingungen werden also bis auf die Ebene der TMS–Knoten zurückgeführt. Dementsprechend besteht die Notwendigkeit, zu jeder erfolgreich angewendeten Regel eine Menge von TMS–Knoten zu ermitteln, die eine hinreichende Bedingung dafür ist, daß ein Beweis des Bedingungsteils der Regel möglich ist. Betrachtet man den Prolog–Beweis für eine Regel als Baum, so sind genau die Blätter dieses Baumes von Bedeutung, an denen IDAX–Prädikate stehen. Jedes IDAX–Prädikat liefert, wie bereits beschrieben, neben dem gewünschten Fakt aus der dynamischen Wissensbasis den (oder die) zugehörigen TMS–Knoten zurück. Sammelt man alle diese mit den Blättern des Beweisbaums assoziierten Knoten auf, so erhält man die gesuchte hinreichende TMS–Rechtfertigung für die bewiesene Präferenz.

Genaugenommen sind aber nicht immer alle zurückgelieferten TMS–Knoten für eine vollständige Rechtfertigung erforderlich. Unter Umständen bestehen nämlich zwischen diesen Knoten bereits Abhängigkeiten, die dazu führen, daß die Gültigkeit eines Knotens nur solange besteht, wie auch ein anderer Knoten innerhalb derselben Rechtfertigung gültig ist. In diesem Fall ist der letztere Knoten für die Rechtfertigung redundant und sollte aus Gründen der Effizienz nicht in selbige aufgenommen werden. Für die Nicht–Aufnahme solcher redundanten Knoten in die Rechtfertigung läßt sich keine allgemeine Regel angeben, sie kann nur im Einzelfall bei der Implementierung vordefinierter Prädikate berücksichtigt werden.

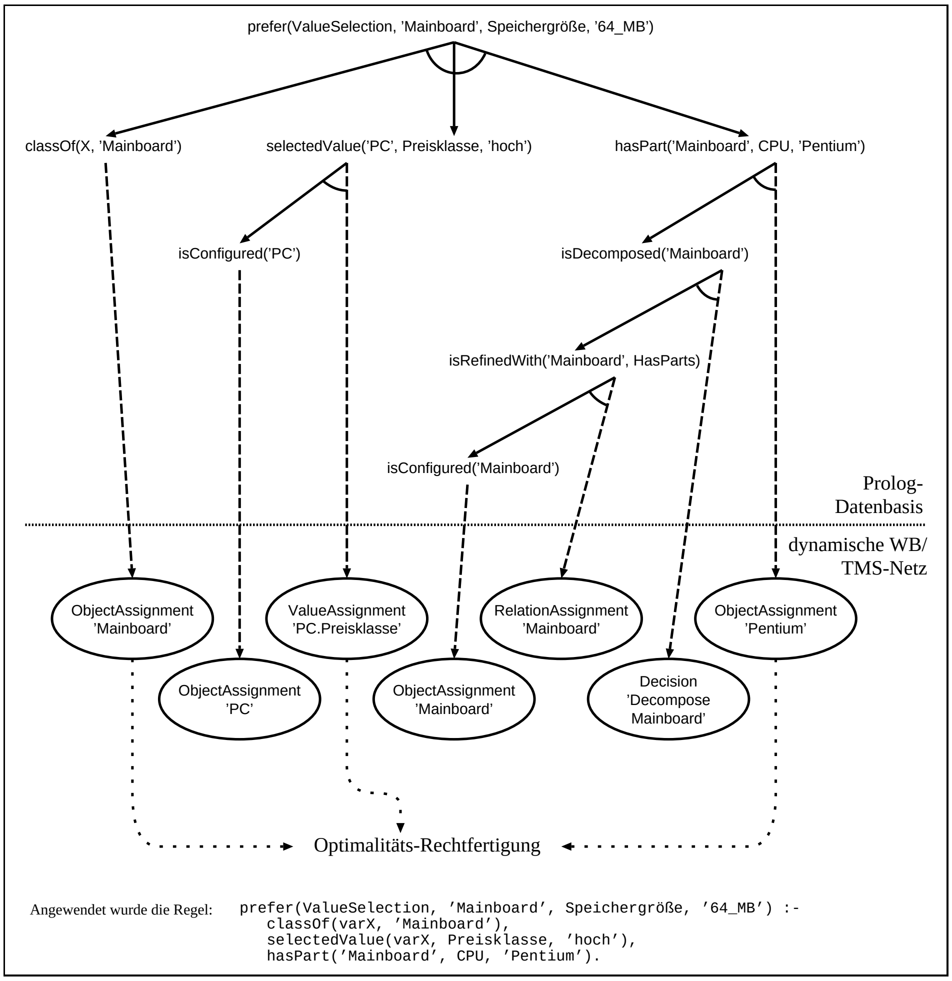


Abbildung 6.1: Beweisbaum für die erfolgreiche Auswertung einer Präferenzregel.

**Beispiel:**

```
hasPart(Goal, Object, Edge, Part, InputList, OutputList) :-
    ⋮
    isDecomposed(Goal, Object, [ ], _),
    st(Goal, partOf:Object, along:Edge, is:Part, Justification),
    append(InputList, [Justification], OutputList).
```

ist ein Ausschnitt aus der Definition des IDAX–Prädikates hasPart. Innerhalb des Beweises für dieses Prädikat wird zuerst geprüft, ob das betreffende Objekt überhaupt mit Hilfe der Dekomposition verfeinert wurde. Die Rechtfertigung aus dem Aufruf des entsprechenden Prozeß–Prädikates isDecomposed wird aber nicht in die Rechtfertigung des hasPart–Prädikates übernommen, weil in dieser im Erfolgsfall ein Knoten steht, der die Existenz der bezeichneten Komponente des Objektes darstellt. Solange diese Komponente existiert, muß aber auch eine Dekomposition des übergeordneten Objektes stattgefunden haben. Der die Dekomposition des Objektes darstellende TMS–Knoten wäre also in der Rechtfertigung des hasPart–Prädikates redundant.

Bei allen Optimaliäts–Rechtfertigungen ist zu beachten, daß sie zwar hinreichende, aber damit keineswegs notwendige Bedingungen für die lokale Optimalität eines durch Präferenzregeln ausgewählten Operators darstellen. Sie repräsentieren vielmehr jeweils genau eine Möglichkeit, einen

Beweis für die Präferenz eines Operators zu führen. Es kann aber außerdem noch viele andere Möglichkeiten für einen solchen Beweis geben, die entsprechend andere TMS–Rechtfertigungen erzeugen würden. Verliert also die ursprünglich erzeugte Optimalitäts–Rechtfertigung ihre Gültigkeit, kann man erneut versuchen, die Optimalität des ausgewählten Operators zu beweisen. Glückt dieser Beweisversuch, so erhält man eine neue, zusätzliche Optimalitäts–Rechtfertigung. Ein Operator kann also im Laufe des Konfigurationsprozesses mit einer Vielzahl von Rechtfertigungen für seine Optimalität versehen werden.

Die Erzeugung von Optimalitäts–Rechtfertigungen im Verlauf von Präferenzbeweisen hat neben der Kontrolle der lokalen Optimalität noch einen angenehmen Nebeneffekt: Da eine Optimalitäts–Rechtfertigung alle Vorbedingungen enthält, die zur Anwendung einer Präferenzregel und damit zur Auswahl des Operators geführt haben, kann ein Benutzer im Nachhinein aus der erzeugten Rechtfertigung leicht die Gründe ablesen, die zu einer Entscheidung des Systems geführt haben. Dadurch ist es möglich, im Verlauf des Konfigurationsprozesses oder nach seinem Abschluß alle Entscheidungen, die auf Grund von Präferenzwissen getroffen wurden, zu erklären. Ein menschlicher Experte kann an Hand der erklärenden Optimalitäts–Rechtfertigungen die Entscheidungen des Systems analysieren und gegebenenfalls Korrekturen des Präferenzwissens vornehmen.

### 6.1.1 Vergleich mit Erklärungsbasierter Generalisierung

Das oben skizzierte Vorgehen erinnert stark an die Erklärungsphase der *Explanation–Based Generalization (EBG)*, wie sie in [MKK86] beschrieben ist, und soll daher im folgenden kurz mit dieser verglichen werden: Ein EBG–Problem besteht aus einem Zielkonzept, für das mit Hilfe von Regeln und Fakten einer Domänentheorie und zusätzlichen Fakten eines Beispiels eine hinreichende Bedingung so formuliert werden soll, daß sie einem Operationalitäts–Kriterium entspricht. Folgende Zuordnungen erlauben es nun, den Beweis einer Präferenzregel in der Terminologie eines EBG–Problems auszudrücken:

**Zielkonzept** : Das zu beweisende Präferenz–Prädikat, also entweder prefer oder preferTo

**Domänentheorie** : Die Präferenzregeln, die vordefinierten Prädikate und die von den IDAX–Prädikaten aufgerufenen Smalltalk–Methoden

**Trainingsbeispiel** : Der aktuelle Zustand der dynamischen Wissensbasis

**Operationalität–Kriterium** : „Die Bedingung für das Zielkonzept darf nur aus TMS–Knoten bestehen.“

Die Erzeugung einer TMS–Rechtfertigung erfolgt dann völlig analog zur Erklärungsphase der EBG–Methode: Es wird ein Beweisbaum für das Zielkonzept gesucht, an dessen Blättern nur Voraussetzungen stehen, die dem Operationalitäts–Kriterium genügen. Diese Voraussetzungen werden zu einer hinreichenden Bedingung für das Zielkonzept zusammengefaßt, die als Erklärung bezeichnet wird. Diese Erklärung wird beim EBG–Verfahren nun in einem zweiten Schritt verallgemeinert, indem (soweit als möglich) die im Beweis verwendeten Konstanten durch Variablen ersetzt werden. Es wird also nicht nur nach einer beliebigen, sondern nach der schwächsten hinreichenden Bedingung gesucht.

Eine solche Generalisierung wird beim Beweis von Präferenzregeln nicht versucht, weil man nur an einer Erklärung der Optimalität, d.h. einer hinreichenden TMS–Rechtfertigung interessiert ist. Sie ließe sich auch technisch nicht erzeugen, weil ein TMS–Knoten nicht unmittelbar durch ein Prädikat dargestellt wird und dementsprechend nicht verallgemeinert werden kann. Aus diesem

Unterschied in der Zielsetzung resultiert auch die seltsam anmutende Verwendung der gesamten dynamischen Wissensbasis als „Trainigsbeispiel“: Während man bei der EBG–Methode aus einem Beispiel eine Erklärung gewinnen will, die auch auf andere ähnliche Beispiele zutrifft, soll die Erklärung der Optimalität nicht auf andere Operatoren übertragen werden. Gelernt wird also, wenn man das Wort überhaupt verwenden will, lediglich die Abhängigkeit der Optimalität eines konkreten Operators von anderen Konfigurationsentscheidungen, ohne daraus eine neue Regel abzuleiten.

### 6.1.2 Behandlung der Negation

In Prolog wird die Negation als sogenannte *negation as failure* interpretiert, d.h. die Negation eines Prädikates ist erfüllt, wenn alle Versuche scheitern, dieses Prädikat zu beweisen. Diese Form der Negation hat den Nachteil, daß sich für den Beweis einer Präferenzregel, in deren Bedingungsteil negierte Prädikate auftreten, nicht ohne weiteres eine Optimalitäts–Rechtfertigung finden läßt. Nachdem ein nicht–negiertes Prädikat erfolgreich bewiesen wurde, läßt sich nämlich genau feststellen, welche Fakten aus der dynamischen Wissensbasis für den Beweis benötigt wurden, so daß sich aus ihnen eine eindeutige Erklärung erzeugen läßt. Scheitert dagegen der Beweis eines Prädikates, so kann man anschließend nicht mehr feststellen, welche Fakten aus der dynamischen Wissensbasis für das Scheitern verantwortlich sind. Das liegt daran, daß es zum einen mehrere Regeln geben kann, deren Köpfe sich mit dem zu beweisenden Prädikat unifizieren lassen und die allesamt scheitern, und daß zum anderen für jede einzelne Regel eine große Zahl von Variablenbelegungen möglich ist, von denen jede einzelne zum Scheitern führt.

Theoretisch ließe sich eine Rechtfertigung für das Scheitern eines Prädikates zwar dadurch erzielen, daß man alle Fakten, die während des gesamten Beweisvorgangs abgefragt wurden, in diese Rechtfertigung aufnimmt. Da der Prolog–Interpreter innerhalb eines gescheiterten Beweisversuches intensives Backtracking betrieben haben kann, ist die Gesamtzahl der angefragten Fakten unter Umständen sehr hoch. Dadurch werden solche Rechtfertigungen aber nicht nur unverhältnismäßig groß, sondern enthalten oft auch zahlreiche Knoten, die Fakten repräsentieren, die für das Scheitern der Regel überhaupt keine Bedeutung besaßen. Wenn sich diese Fakten nun später ändern, wird die betreffende Rechtfertigung ungültig, obwohl es dafür keinen tatsächlichen Grund gibt.

**Beispiel:**

Angenommen, die Regel

```
prefer( ... ) :-
    possibleValue('Objekt','Attribut_1',X),
    possibleValue('Objekt','Attribut_2',Y),
    X > Y,
    Y > 0.
```

scheitert. Dies kann daran liegen, daß

1. es überhaupt keinen zulässigen Wert für Attribut_1 gibt.
2. es überhaupt keinen zulässigen Wert für Attribut_2 gibt.
3. kein zulässiger Wert für Attribut_2 kleiner als ein zulässiger Wert für Attribut_1 ist.
4. kein zulässiger Wert für Attribut_2 größer als 0 ist.
5. kein zulässiger Wert für Attribut_2 beide Bedingungen erfüllt.

Die beiden ersten Fällen lassen sich verhältnismäßig leicht erkennen. Für sie läßt sich auch eine sinnvolle Rechtfertigung für das Scheitern der Regel erzeugen. In diese müssen nämlich genau die Zurückweisungs–Entscheidungen aller alternativen Werte für das betreffende Attribut aufgenommen werden. Die drei letztgenannten Fälle lassen sich dagegen nicht voneinander unterscheiden. Deswegen müssen insbesondere im 4. Fall auch alle Zurückweisungs–Entscheidungen, die Attribut_1 betreffen, in die Rechtfertigung aufgenommen werden, obwohl diese nicht zum Scheitern der Regel beigetragen haben.

Weil es unmöglich ist, eine sinnvolle Rechtfertigung für das Scheitern eines Prädikates zu liefern, wird die Verwendung der Negation, also des Standard–Prädikates not, bei der Definition von Präferenzregeln im allgemeinen verboten. Aus demselben Grund darf man auch nicht ausnutzen, daß die Präferenzregeln von Prolog immer der Reihe nach abgearbeitet werden. Denn in diesem Falle nutzt man als implizite Vorbedingung einer Regel zusätzlich, daß alle vorangehenden Regeln mit passendem Regelkopf gescheitert sind. Das Scheitern dieser Regeln müßte dann aber in die Rechtfertigung für den Beweis der schließlich erfolgreichen Regel aufgenommen werden.

Auf der anderen Seite stellt die Negation ein unverzichtbares Hilfsmittel bei der Formulierung von Bedingungen dar, insbesondere wenn gefordert wird, daß sich die Bedingungen innerhalb eines Satzes von Präferenzregeln gegenseitig ausschließen, um die Eindeutigkeit der Regelauswertung zu garantieren. Deshalb wird die Negation in zwei besonderen Fällen ausdrücklich erlaubt:

- Jedes IDAX–Prädikat darf unmittelbar negiert werden, z.B.:

  not( selectedValue('PC', Preisklasse, 'niedrig') )

- Jedes Standard–Prädikat darf unmittelbar negiert werden, z.B.:

  not( between(3,5,X) )

In diesen beiden Fällen läßt sich eine Rechtfertigung für das Scheitern des negierten Prädikates unmittelbar formulieren. Wenn ein Standard–Prädikat scheitert, so ist dies unabhängig vom Zustand der dynamischen Wissensbasis, weil innerhalb des Standard–Prädikates kein Zugriff auf diese erfolgen kann. Also ist für ein Scheitern des Prädikates, d.h. den Erfolg des Prädikates not, in diesem Fall keine Rechtfertigung erforderlich. Wenn ein IDAX–Prädikat scheitert, so bedeutet dies, daß der durch dieses Prädikat beschriebene Fakt der dynamischen Wissensbasis nicht vorhanden ist. Dafür läßt sich aber in ähnlicher Weise eine Rechtfertigung formulieren wie für das Vorhandensein eines Fakts. Jedes IDAX–Prädikat muß also nicht nur im Fall eines Erfolgs, sondern auch im Falle eines Mißerfolgs einen oder mehrere TMS–Knoten zurückliefern, welche den entsprechenden Zustand der dynamischen Wissensbasis hinreichend beschreiben.

Die Menge der TMS–Knoten, die benötigt wird, um eine hinreichende Vorbedingung für das Scheitern eines Prädikates zu formulieren, ist schwieriger zu beschreiben als diejenige, die im Erfolgsfall zurückgeliefert wird. Das liegt daran, daß es für das Scheitern eines Prädikates sehr viel mehr verschiedene Ursachen geben kann als für seine Erfüllung, wie bereits das folgende einfache Beispiel zeigt. Allgemein läßt sich aber sagen, daß in die Begründung des Scheiterns sehr oft solche TMS–Knoten eingehen, die beschreiben, daß eine bestimmte Teilaufgabe noch nicht bearbeitet worden ist.

**Beispiel:**

Angenommen, daß Prädikat

classOf('CPU', 'CPU-386')

scheitert. Dafür gibt es allein vier mögliche Gründe:

- Für das Objekt CPU ist noch keine Verfeinerungsrelation ausgewählt worden. Als Rechtfertigung dient dann derjenige TMS–Knoten, der darstellt, daß die Aufgabe, eine Verfeinerungsrelation zu wählen, noch unbearbeitet ist.
- Für das Objekt CPU ist als Verfeinerungsrelation die Dekomposition ausgewählt worden. Als Rechtfertigung dient dann derjenige TMS–Knoten, der die Entscheidung für die Auswahl der Verfeinerungsrelation repräsentiert.
- Für das Objekt CPU ist zwar als Verfeinerungsrelation die Spezialisierung ausgewählt, ein Spezialisierungsschritt ist aber noch nicht durchgeführt worden. In diesem Fall dient als Rechtfertigung derjenige TMS–Knoten, der darstellt, daß die Aufgabe, eine Spezialisierung festzulegen, noch unbearbeitet ist.
- Das Objekt CPU ist bereits spezialisiert worden, und zwar zu 'Pentium'. Dann dient als Rechtfertigung derjenige TMS–Knoten, der die Entscheidung für die Auswahl der Spezialisierung repräsentiert.

Man kann sich leicht vorstellen, daß beispielsweise die Rechtfertigung für das Scheitern eines hasPartTrans–Prädikates sehr umfangreich wird, da nach dem Muster des obigen Beispiels alle TMS–Knoten in die Rechtfertigung aufgenommen werden müssen, die eine unbearbeitete Aufgabe darstellen, deren Bearbeitung mittelbar oder unmittelbar eine weitere Komponente erzeugen könnte.

Zusätzlich ergibt sich bei der Erzeugung von Rechtfertigungen für negierte Prädikate ein technisches Problem aus der Tatsache, daß der Prolog–Interpreter beim Scheitern eines Prädikates keine Variablenbindungen vornimmt. Damit ist die direkte Rückgabe einer Rechtfertigung durch ein gescheitertes IDAX–Prädikat nicht möglich; der im Abschnitt 4.4 geschilderte Mechanismus zum Sammeln der TMS–Rechtfertigung in einer durch den Beweisprozeß mitgeführten Liste kann also nicht unmittelbar verwendet werden. Stattdessen muß die Rechtfertigung für das Scheitern eines IDAX–Prädikates zunächst in der dynamischen Wissensbasis zwischengespeichert werden. Diese Rechtfertigung wird dann bei der Auswertung des not–Prädikates abgerufen, und zwar unmittelbar nachdem ein IDAX–Prädikat gescheitert ist. Dazu mußte ein spezielles Prädikat not vordefiniert werden, das sich vom gleichnamigen Standard–Prädikat dadurch unterscheidet, daß es statt einem vier Parameter besitzt. Denn genau wie bei einem IDAX–Prädikat müssen zusätzlich das aufrufende Ziel und die beiden Listen für die Rechtfertigungen übergeben werden. Da dies automatisch bei der Übertragung der Präferenzregeln in die Prolog–Datenbasis geschieht, sieht der Benutzer keinen Unterschied zum Standard–Prädikat not.

Ein negiertes IDAX–Prädikat kann bei der Definition von Präferenzregeln in gleicher Weise verwendet werden wie ein nicht–negiertes. Insbesondere darf auch das innerhalb der beschränkten All–Quantifizierung (Abschnitt 5.3.1) zu beweisende IDAX–Prädikat negiert sein. Mit dieser eingeschränkten Form der Negation lassen sich viele Bedingungen auf natürliche Weise formulieren, insbesondere solche, die sich gegenseitig ausschließen.

**Beispiel:**

Die folgenden beiden Regeln wählen eine Verfeinerungsrelation für das Objekt PC in Abhängigkeit davon aus, ob das Attribut Preisklasse bereits belegt ist oder nicht.

```
prefer(ObjectRefinement, 'PC', ClassOf) :-
    selectedValue('PC', Preisklasse, X).

prefer(ObjectRefinement, 'PC', HasParts) :-
    not( selectedValue('PC', Preisklasse, X) ).
```

## 6.2 Verwendung von Optimalitäts–Rechtfertigungen im Konfigurationsprozeß

### 6.2.1 Auswahl eines Operators

Jeder Operator besitzt einen speziellen TMS–Knoten, dessen Gültigkeit die Optimalität des zugehörigen Operators repräsentiert, den sogenannten bestOp–Knoten. Wird ein Operator ausgewählt, so muß er zum Zeitpunkt seiner Auswahl optimal sein, da es keinen Sinn machen würde, von vorneherein eine bekanntermaßen suboptimale Entscheidung zu treffen. Der bestOp–Knoten eines Operators muß dementsprechend bei dessen Auswahl bereits eine gültige Rechtfertigung besitzen oder eine solche erhalten. Die Auswahl eines Operators kann entweder selbständig durch die Kontrollkomponente von IDAX oder interaktiv durch den Benutzer erfolgen. Die zweite Möglichkeit wird ausführlich im Kapitel 7 behandelt, weswegen im folgenden nur die Operatorauswahl durch das IDAX–System betrachtet wird.

Bei der Auswahl eines Operators muß berücksichtigt werden, daß nicht alle Optimalitäts–Rechtfertigungen von derselben Qualität sind. Es wird unterschieden zwischen einer begründeten und einer unbegründeten Rechtfertigung, wobei letztere auch als Annahme bezeichnet wird. Damit wird ausgedrückt, daß unbegründete Optimalitäts–Rechtfertigungen sich — im Gegensatz zu begründeten — nicht auf explizites Domänen- bzw. Expertenwissen stützen, sondern durch eine — eventuell nur zufällige — Auswahl des Operators bei fehlender Information über die tatsächliche Optimalität zustande gekommen sind. Aus dieser Unterscheidung und dem Wunsch, bereits vorhandenes Wissen aus dem bisherigen Konfigurationsprozeß so weit als möglich zu nutzen, ergibt sich für die Kontrollkomponente folgende Priorität bei der Auswahl eines Operators aus einer Menge zulässiger Alternativen:

1. Wähle einen Operator, der bereits eine gültige, begründete Rechtfertigung für seine Optimalität besitzt. Diese Rechtfertigung könnte bereits aus einer früheren Auswahl des Operators stammen, die zwischenzeitlich revidiert wurde.

2. Versuche mit Hilfe von Präferenzregeln — falls solche vorhanden sind — die Optimalität eines Operators zu beweisen. Gelingt dies, so wird der präferierte Operator ausgewählt und gleichzeitig eine begründete Rechtfertigung für seine Optimalität erzeugt, wie in Abschnitt 6.1 beschrieben.

3. Wähle einen Operator, der bereits eine gültige, wenn auch unbegründete Optimalitäts–Rechtfertigung besitzt. Auch diese könnte wegen einer bereits früher erfolgten Auswahl des Operators bereits vorhanden sein.

4. Wähle zufällig einen Operator aus. Da kein Auswahlwissen vorhanden ist, kann die Optimalität des Operators nicht begründet werden. Sie wird also mit einer Annahme gerechtfertigt, die als solche durch einen speziellen TMS–Knoten (IDAXOptimalityAssumption) gekennzeichnet wird. Auf diese Weise können unbegründete Annahmen später leicht erkannt und — falls gewünscht — vom Benutzer korrigiert werden.

Durch dieses Vorgehen bei der Operatorauswahl wird sichergestellt, daß das vorhandene Wissen vollständig ausgenutzt wird. Da man aber davon ausgehen muß, daß dieses Wissen stets unvollständig ist, kann die lokale Optimalität eines Operators oft nicht eindeutig bestimmt werden. Es kann also durchaus mehrere Operatoren in einer Konfliktmenge geben, deren Optimalität in gleicher Weise gerechtfertigt ist, bzw. gerechtfertigt werden könnte. In diesem Fall bleibt nur die zufällige Auswahl eines dieser Operatoren, die entweder durch deren Reihenfolge in der internen Repräsentation der Konfliktmenge oder durch die Reihenfolge der zugehörigen Präferenzregeln in der Prolog–Datenbasis determiniert wird.

### 6.2.2 Behandlung von Optimalitätsverlusten

In einer Optimalitäts–Rechtfertigung sind alle Voraussetzungen enthalten, die dazu geführt haben, daß der zugehörige Operator bei seiner Auswahl als optimal angesehen wurde. Fällt auch nur eine dieser Voraussetzungen weg, so wird die Rechtfertigung ungültig. Werden alle vorhandenen Rechtfertigungen — es kann ja mehrere geben — für die Optimalität eines Operators ungültig, so verliert der zugehörige TMS–Knoten (bestOp) ebenfalls seine Gültigkeit. Dieser Zustandsübergang wird vom TMS erkannt und dem REDUX– bzw. IDAX–System mitgeteilt. Der erkannte Optimalitätsverlust führt dort zur Erzeugung einer neuen Konfigurationsaufgabe, eines sogenannten IDAXOptimalityLossTask. Diese Aufgabe kann jetzt — im Gegensatz zu den im Abschnitt 3.2 beschriebenen Aufgaben — nicht durch die Bearbeitung eines offenen Zieles gelöst werden, sondern durch die Wiederherstellung der lokalen Optimalität der betroffenen Entscheidung.

Das Auftreten eines Optimalitätsverlustes zwingt allerdings nicht dazu, die dadurch entstehende Konfigurationsaufgabe sofort zu bearbeiten. Diese steht vielmehr zusammen mit den übrigen ungelösten Aufgaben auf der Agenda, aus der die Kontrollkomponente beliebig auswählen kann. Es ist eine Frage der Konfigurationsstrategie, die entweder durch die definierten Konfigurationsphasen oder interaktive Eingriffe des Benutzers vorgegeben ist, wann Optimalitätsverluste behandelt werden. Der Konfigurationsprozeß kann insbesondere auch bis zum Ende fortgeführt werden, ohne die Optimalitätsverluste zu berücksichtigen. Diese Strategie ist immer dann angebracht, wenn man keine optimale, sondern nur eine korrekte Lösung erzielen will oder wenn Präferenzen nicht zum Zwecke der Lösungsoptimierung, sondern zur heuristischen Steuerung des Konfigurationsprozesses formuliert wurden.

Wenn sich aber die Kontrollkomponente oder der Benutzer entschließt, den Optimalitätsverlust einer Entscheidung zu beheben, werden dazu die folgenden Schritte der Reihe nach ausgeführt (siehe auch Abb. 6.2):

1. Es wird überprüft, ob die Entscheidung für den derzeit ausgewählten, als suboptimal erkannten Operator vom System oder vom Benutzer getroffen wurde. Handelt es sich um eine Benutzer–Entscheidung, so soll dieser auch über das weitere Vorgehen entscheiden, wie in Abschnitt 7.3 beschrieben.

2. Es wird erneut versucht, mit den vorhandenen Präferenzregeln die Optimalität des ausgewählten Operators zu beweisen. Da die Optimalitäts–Rechtfertigungen nicht notwendige, sondern nur hinreichende Bedingungen für die Optimalität eines Operators ausdrücken, ist es durchaus möglich, daß der Operator seine Optimalität überhaupt nicht verloren hat. Gelingt es, einen neuen Beweis für die Optimalität des ausgwählten Operators zu finden, so erhält dieser eine neue gültige Optimalitäts–Rechtfertigung, und der Optimalitätsverlust ist ohne Änderungen in der aktuellen Teilkonfiguration behoben.

3. Wenn die Optimalität des ausgewählten Operators nicht wiederhergestellt werden konnte, so

wird die Entscheidung für diesen Operator zurückgezogen, wodurch der Optimalitätsverlust behoben ist. Damit ist gleichzeitig das durch den Operator bislang erfüllte Ziel wieder offen, und die entsprechende Konfigurationsaufgabe wird zur erneuten Bearbeitung wieder in die Agenda aufgenommen.

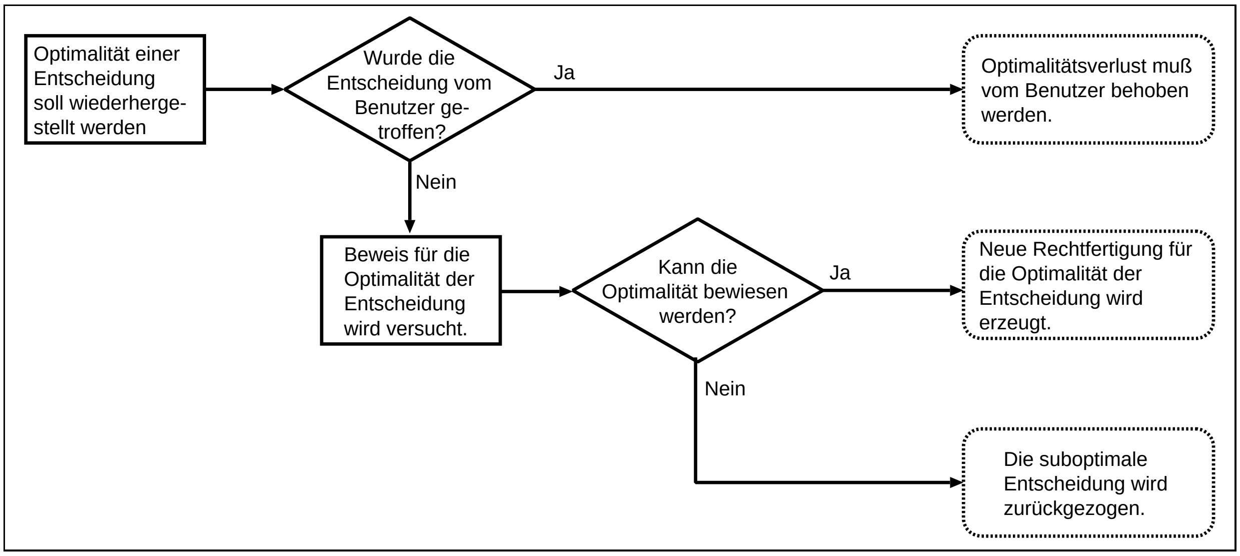


Abbildung 6.2: Behebung eines Optimalitätsverlustes

Der Rückzug einer suboptimalen Entscheidung kann für den Konfigurationsproz̈ß weitreichende Folgen haben, da möglicherweise eine Vielzahl von abhängigen Entscheidungen ebenfalls zurückgenommen werden muß. Darüberhinaus können auch weitere Entscheidungen in diesem Zusammenhang ihre Optimalität verlieren. Dieser zusätzliche Aufwand ist aber der Preis, den man für die Suche nach einer pareto–optimalen Lösung letztendlich zahlen muß.

Insbesondere gibt es Konfigurationsprobleme, bei deren Lösung der Versuch, alle Optimaliätsverluste zu beseitigen, in eine endlose Schleife führt. Bislang werden solche Schleifen innerhalb der Lösungssuche nicht erkannt, weswegen die Termination des Konfigurationsprozesses bei bevorzugter Bearbeitung der Optimalitätsverluste nicht garantiert werden kann.

**Beispiel:**

Die vier folgenden Regeln machen es unmöglich, die beiden Attribute Attribut_1 und Attribut_2 gleichzeitig so mit den Werten A und B zu belegen, daß kein Optimalitätsverlust auftritt. Wird der Optimalitätsverlust durch Rückzug eines Wertes behoben und im nächsten Schritt stattdessen der optimale Wert gewählt, so tritt erneut ein Optimalitätsverlust für die Belegung des anderen Attributes auf.

```
prefer(ValueSelection, 'Objekt', 'Attribut_1', 'A') :-
    not( selectedValue('Objekt', Attribut_2', 'A') ).

prefer(ValueSelection, 'Objekt', 'Attribut_1', 'B') :-
    selectedValue('Objekt', Attribut_2', 'A').

prefer(ValueSelection, 'Objekt', 'Attribut_2', 'A') :-
    not( selectedValue('Objekt', Attribut_1', 'B') ).

prefer(ValueSelection, 'Objekt', 'Attribut_2', 'B') :-
    selectedValue('Objekt', Attribut_1', 'B').
```

Prinzipiell lassen sich derartige Schleifen aber erkennen und dadurch aufbrechen, daß ein als suboptimal erkannter Operator eine zusätzliche Rechtfertigung erhält, in die alle übrigen an der Schleife beteiligten Entscheidungen eingehen. Auf diese Weise kann eine Fortführung des Konfigurationsprozesses ohne Optimalitätsverlust erzwungen werden. Auch bleibt die Pareto–Optimalität bei diesem Vorgehen erhalten, da es nach wie vor nicht möglich ist, einen Operator durch einen lokal besseren zu ersetzen, ohne daß an einer anderen Stelle ein Verlust der lokalen Optimalität entsteht.

**Beispiel:**

Im obigen Beispiel könnten beispielsweise sowohl Attribut_1 als auch Attribut_2 mit dem Wert A belegt werden. In diesem Fall wäre die Entscheidung für Attribut_2 optimal, für Attribut_1 jedoch nicht. Rechtfertigt man deren Optimalität jedoch zusätzlich mit der Entscheidung, daß für Attribut_2 der Wert A gewählt ist, so sind alle Optimalitätsverluste beseitigt. Muß die Belegung von Attribut_2 jedoch später aus einem anderen Grund zurückgezogen werden, so wird der Optimalitätsverlust der Wahl für Attribut_1 richtig erkannt.

# Kapitel 7

# Benutzer-Interaktion

## 7.1 Interaktive Operatorauswahl

Da IDAX als „Design–Assistent" konzipiert ist, der einen menschlichen Experten bei der Lösung einer Konfigurationsaufgabe unterstützen soll, hat der Benutzer zu jedem Zeitpunkt Gelegenheit, interaktiv in den Konfigurationsprozeß einzugreifen. Er entscheidet darüber, welche Konfigurationsschritte er selber festlegen will und welche er zur Bearbeitung an die Kontrollkomponente des IDAX–Systems delegiert. Der Benutzer kann dabei sowohl die Auswahl der zu bearbeitenden Aufgabe als auch die Entscheidung für den anzuwendenden Operator treffen.

Wenn sich der Benutzer entschließt, selbst einen Operator für die zu bearbeitende Aufgabe auszuwählen, so legt ihm das IDAX–System nur die derzeit möglichen Alternativen zur Auswahl vor. Der Benutzer wird also davor bewahrt, eine unzulässige Entscheidung zu treffen, was bereits eine wesentliche Hilfe darstellt. Nachdem der Benutzer sich für einen Operator entschieden hat, gibt es für ihn zwei Möglichkeiten:

1. Der Benutzer verzichtet auf eine Begründung für seine Entscheidung. In diesem Fall erhält der ausgewählte Operator eine unbegründete Optimalitäts–Rechtfertigung, deren Gültigkeit nur von dem bereits erwähnten speziellen TMS–Knoten IDAXOptimalityAssumption abhängt. Deswegen gilt der ausgewählte Operator so lange als optimal, bis der Benutzer seine Optimalität explizit für ungültig erklärt.

2. Der Benutzer teilt dem System die Gründe für seine Auswahl mit. Dies geschieht, indem er die Voraussetzungen seiner Entscheidung in Form von IDAX–Prädikaten explizit formuliert. Das System überprüft die Gültigkeit dieser Prädikate mit Hilfe des Prolog–Interpreters und ermittelt dabei gleichzeitig diejenigen TMS–Knoten, welche die Gültigkeit der Voraussetzungen bedingen. Diese Knoten werden dann zur Rechtfertigung der Optimalität des gewählten Operators verwendet.

Die Entscheidungen innerhalb eines Konfigurationsprozesses lassen sich somit an Hand von zwei Kriterien unterscheiden: Zum einen können sie vom Benutzer oder vom IDAX–System getroffen worden sein, und zum anderen sind sie entweder begründet oder unbegründet. Damit entstehen, wie in Abbildung 7.1 gezeigt, vier Arten von Konfigurationsentscheidungen, die sich in ihrer Bedeutung unterscheiden und daher entsprechend gekennzeichnet sind, damit sie gegebenenfalls unterschiedlich behandelt werden können.

Wenn der Benutzer seine Entscheidung gegenüber dem System begründet, so kann IDAX im

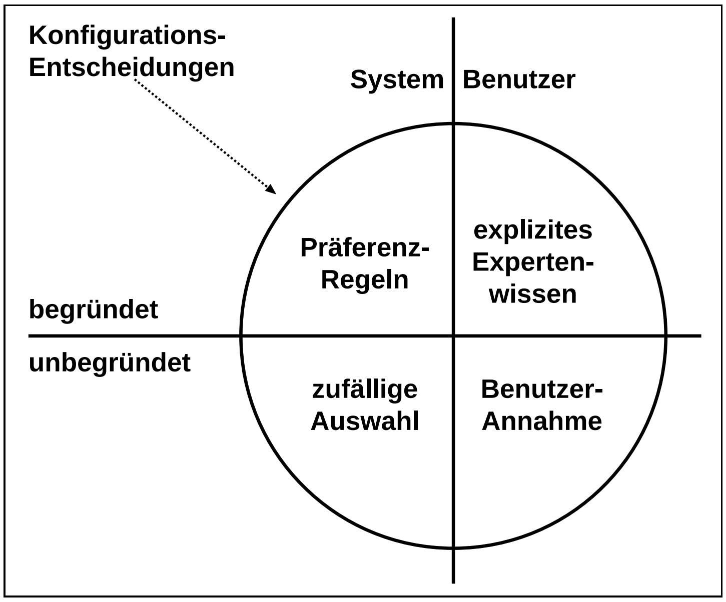


Abbildung 7.1: Die vier Arten von Konfigurationsentscheidungen

Verlauf des weiteren Konfigurationsprozesses überwachen, ob die Gründe für diese Benutzerentscheidung noch gültig sind. Sobald dies nicht mehr der Fall ist, wird ein Optimalitätsverlust registriert und dem Benutzer durch einen IDAXOptimalityLossTask angezeigt, daß der ursprüngliche Grund für seine Entscheidung weggefallen ist. Es steht dann erneut im Ermessen des Benutzers, wie er weiter verfahren möchte.

In einem komplexen Konfigurationsprozeß kann IDAX einen Experten auf diese Weise bei der Auswahl eines Operators in dreierlei Hinsicht unterstützen:

- Unzulässige Alternativen werden ausgeblendet, um Fehlentscheidungen des Benutzers zu verhindern.
- Unbegründete Annahmen des Benutzers werden registriert, um ihm später Ansatzpunkte für Korrekturen liefern zu können.
- Der Wegfall vom Benutzer angegebener Gründe für seine Entscheidungen wird angezeigt, um frühzeitig auf Verbesserungsmöglichkeiten hinzuweisen.

Durch die Verwendung derselben Prädikate zur Eingabe von Entscheidungsbegründungen und zur Formulierung von Bedingungen von Präferenzregeln ist es gelungen, einen einheitlichen Mechanismus für die Überwachung der Optimalität von Entscheidungen zu schaffen. Ausgehend von der Annahme, daß das in der statischen Wissensbasis repräsentierte Domänenwissen stets unvollständig ist, bietet sich so eine einfache Möglichkeit für einen Experten, während der Bearbeitung einer Konfigurationsaufgabe zusätzliches Wissen in den Lösungsprozeß einzubringen, um das vorhandene Domänenwissen im Bedarfsfall flüchtig zu ergänzen.

Für die Eingabe von Begründungen steht dem Benutzer ein spezieller Editor zur Verfügung, der das Formulieren von geeigneten IDAX–Prädikaten durch interaktive Hilfen unterstützt. So erhält der Benutzer auf Wunsch Listen mit allen vordefininierten IDAX–Prädikaten, mit allen zur Zeit existierenden dynamischen Objekten und mit allen Attributen sowie deren Werten, falls sie vorhanden sind. Aus diesen Listen kann er jederzeit die gewünschten Einträge in den Editor übernehmen. Sobald ein Prädikat eingegeben ist, ermittelt der Prolog–Interpreter den zugehörigen

TMS–Knoten, der dann im Editor angezeigt wird und vom Benutzer inspiziert werden kann. In Abbildung 7.2 hat der Benutzer beispielsweise eine von ihm getroffene Spezialisierungsentscheidung mit dem Wert des Attributes Preisklasse begründet, woraufhin als zugehöriger TMS–Knoten ein entsprechendes ValueAssignment ermittelt worden ist.

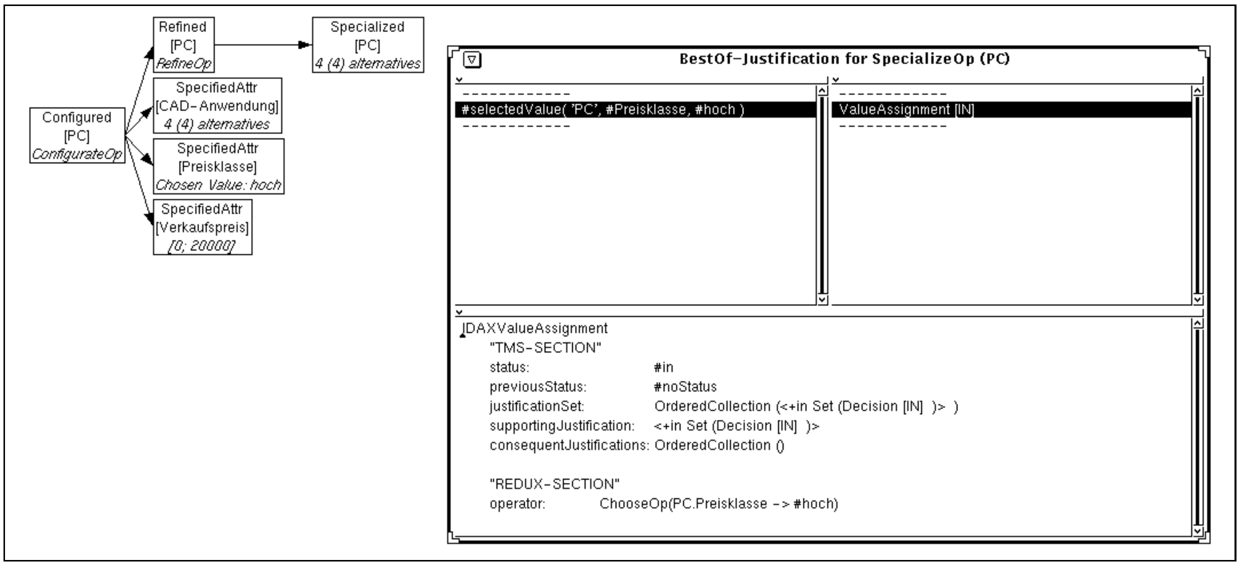


Abbildung 7.2: Editor zur Eingabe von Begründungen für Benutzer–Entscheidungen

## 7.2 Inkrementeller Erwerb von Präferenzwissen

Ausgehend von der Annahme, daß der Benutzer des IDAX–Systems ein Experte innerhalb der Domäne oder eines Teils derselben ist, ist es sinnvoll, dem Benutzer während eines Konfigurationsprozesses die Möglichkeit zu eröffnen, die statische Wissensbasis zu erweitern, wenn ihm dies notwendig erscheint. Interaktive Veränderungen des nicht ohne Grund als „statisch“ bezeichneten Wissens sind im allgemeinen problematisch, da sie die Konsistenz der aktuellen Teilkonfiguration mit der statischen Wissensbasis ebenso bedrohen wie die Konsistenz früher erzeugter Lösungen, deren Wiederverwendung bzw. Modifikation damit unmöglich würde. Dieses Problem tritt allerdings nicht auf, solange die Änderungen auf die Präferenzregeln beschränkt bleiben, da diese die Korrektheit der aktuellen oder früherer Konfigurationen nicht in Frage stellen können.

Immer, wenn der Benutzer eine Konfigurationsentscheidung trifft, setzt er dabei ein Stück seines Expertenwissens ein. Begründet er zusätzlich seine Entscheidung, so macht er dadurch das eingesetzte Wissen explizit und damit dem System zugänglich. Kommt der Benutzer zu dem Schluß, daß die von ihm gewählte Entscheidung samt ihrer Begründung nicht nur im konkreten Einzelfall, sondern auch in vergleichbaren Situation sinnvoll ist, so bietet es sich natürlicherweise an, aus dieser Erkenntnis eine dauerhaft gültige Präferenzregel zu formulieren.

Daher kann der Benutzer nach der Eingabe einer begründeten Entscheidung diese vom System zu einer Präferenzregel umformulieren lassen. Dabei kann das System auf Wunsch des Benutzers alle konkreten Namen dynamischer Objekte durch Variablen ersetzen und zusätzlich geeignete Relationen–Prädikate in den Bedingungsteil der Regel einfügen, um diese Variablen, ausgehend von demjenigen Objekt, für das die Regel definiert wird, korrekt zu belegen.

**Beispiel:**

Der Benutzer hat für die Kardinalität der Komponente Hauptspeicher des Objektes

Mainboard den Wert 32 gewählt und als Begründung angegeben:

selectedValue('Mainboard.Prozessor', Bezeichnung, Pentium) ,

wobei Mainboard.Prozessor der eindeutige Bezeichner für die Komponente Prozessor sein möge. Diese Begründung kann das System in die folgende Regel transformieren:

```
prefer(SelectedCardinality, self, Hauptspeicher, 32) :-
    hasPart(self, Processor, varX),
    selectedValue(varX, Bezeichnung, Pentium).
```

In dieser Regel tauchen keine konkreten Bezeichner von dynamischen Objekten mehr auf.

Die automatisch generierte Regel wird aber nicht unmittelbar in die statische Wissensbasis übernommen, sondern vorher dem Benutzer zu einer möglichen Überarbeitung vorgelegt. So kann dieser leicht eine Verallgemeinerung der speziellen Einzelpräferenz vornehmen, um so die Anwendung der Präferenzregel von der konkreten Situation auf ähnliche Situationen auszuweiten. Der Benutzer kann zu diesem Zeitpunkt auch zusätzliche Regeln eingeben, um den von ihm einmal erkannten Sachverhalt möglichst vollständig zu beschreiben. Erst wenn der Benutzer mit dem gesamten Regelsatz für die betreffende Entscheidung einverstanden ist, werden die Änderungen in die statische Wissensbasis aufgenommen. Auf Wunsch des Benutzers wird der geänderte Regelsatz danach sofort ausgewertet. Kann damit ein Präferenzbeweis geführt werden, so ersetzt der dabei gefundene Operator mitsamt einer entsprechenden Rechtfertigung die vorherige Benutzereingabe. Auf diese Weise ist es möglich, unmittelbar zu überprüfen, ob die Präferenzregeln zumindest im vorliegenden Beispiel zum gewünschten Resultat führen.

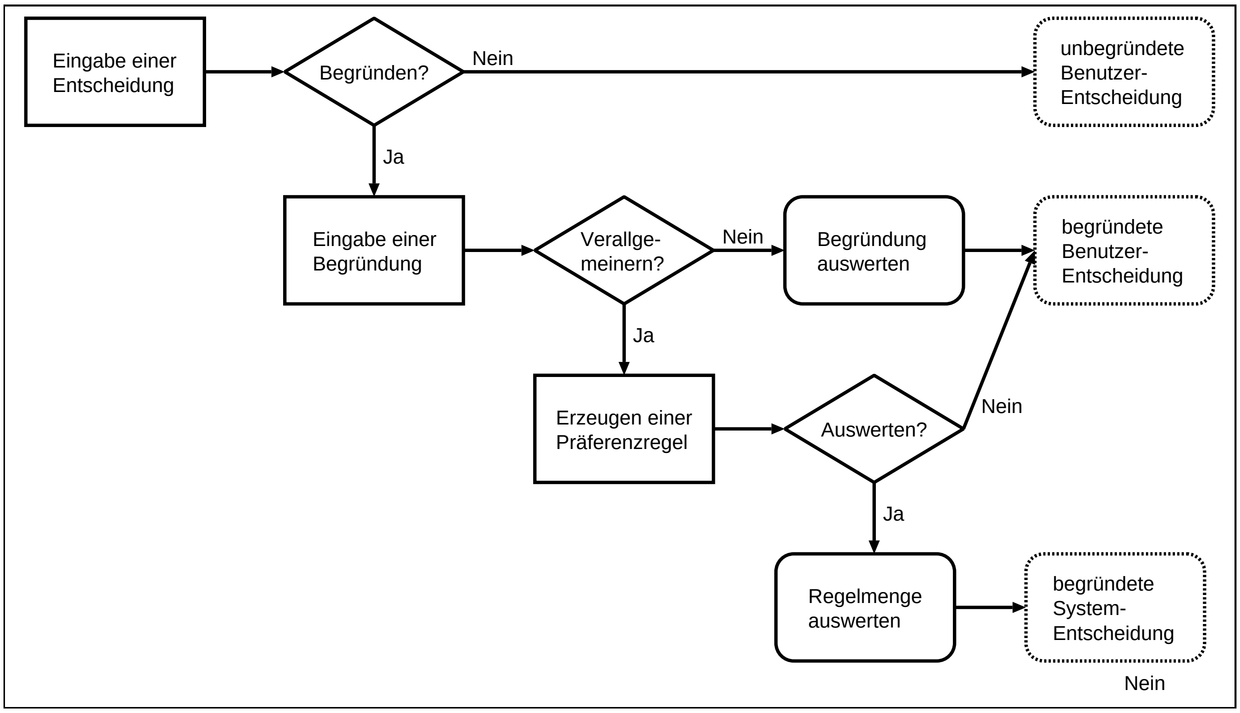


Abbildung 7.3: Schematischer Ablauf einer Benutzer–Entscheidung

Durch die interaktive Eingabe von Präferenzregeln ist es möglich, das Domänenwissen inkrementell zu erweitern. Auf diese Weise wird auch der Tatsache Rechnung getragen, daß sich

Expertenwissen nicht von vorneherein vollständig formalisieren läßt, sondern sich oftmals erst an konkreten Einzelfällen manifestiert. Das inkrementelle Erlernen von Präferenzregeln durch Benutzereingaben während eines konkreten Konfigurationsprozesses stellt damit eine Möglichkeit der Wissensakquistition dar, die im Vergleich zu einem abstrakten Entwurf großer Regelmengen aufgrund rein theoretischer Vorüberlegungen als sehr viel natürlicher erscheint.

## 7.3 Interaktive Auflösung von Optimalitätsverlusten

Wenn eine begründete Benutzer–Entscheidung ihre Optimalität verliert, weil die angegebene Begründung ungültig wird, so wird auch dieser Optimalitätsverlust durch einen IDAXOptimalityLossTask angezeigt. Diese Konfigurationsaufgabe darf nicht vom System abgearbeitet werden, da vermieden werden soll, daß eine Entscheidung des Benutzers ohne dessen ausdrückliches Einverständnis zurückgenommen wird, selbst wenn die Gründe für diese Entscheidung hinfällig geworden sind. Dementsprechend muß die Bearbeitung eines solchen Optimalitätsverlustes, wenn sie überhaupt erfolgt, vom Benutzer durchgeführt werden, der dazu vier Möglichkeiten besitzt:

1. Der Benutzer stellt die Optimalität des ausgewählten Operators wieder her, indem er eine neue, gültige Begründung eingibt. Dazu stehen ihm erneut alle Möglichkeiten offen, die in den Abschnitten 7.1 und 7.2 beschrieben sind.

2. Der Benutzer zieht seine Entscheidung zurück. Die zugehörige Konfigurationsaufgabe ist dann wieder offen und muß vom Benutzer oder durch das System zu einem späteren Zeitpunkt erneut bearbeitet werden.

3. Der Benutzer verzichtet — zumindest vorläufig — auf eine Änderung. In diesem Fall bleibt der Optimalitätsverlust selbstverständlich weiterhin bestehen.

4. Der Benutzer überträgt die Behandlung des Optimalitätsverlustes dem IDAX–System, das dann in der unter 6.2.2 beschriebenen Weise vorgeht.

# Kapitel 8

# Zusammenfassung und Ausblick

## 8.1 Zusammenfassung

In dieser Diplomarbeit wurde eine Komponente für das Konfigurationssystem IDAX vorgestellt, die es ermöglicht, Konfigurationsschritte in Abhängigkeit von domänenspezifischen Präferenzregeln auszuwählen. Diese Regeln können entweder Heuristiken zur Steuerung des Konfigurationsprozesses oder lokale Gütekriterien der Domäne ausdrücken. Der zur Verarbeitung von Präferenzregeln realisierte Ansatz wird durch folgende vorteilhafte Eigenschaften charakterisiert:

- **Einbindung der Präferenzregeln in die Begriffshierarchie:** Präferenzregeln werden lokal als Eigenschaft von Domänenobjekten definiert. Dies unterstützt den strukturierten Regelentwurf und führt zu verhältnismäßig kleinen Regelmengen, die gut zu warten sind.
- **Zugriff auf alle Fakten der dynamischen Wissensbasis:** Über vordefinierte Prädikate ist es möglich, vorangegangene Entscheidungen im Konfigurationsprozeß als Voraussetzungen für unterschiedliche Präferenzen zu nutzen. Durch begrenzte All–Quantifizierung und unmittelbare Negation können auch komplexe Vorbedingungen formuliert werden.
- **Regelauswertung mit einem Prolog–Interpreter:** Syntax und Semantik der Präferenzregeln folgen dem weithin bekannten Prolog–Standard. Die Verwendung eines Prolog–Interpreters als Inferenzmaschine erlaubt eine effiziente Regelverarbeitung, die in Smalltalk bisher nicht möglich ist.
- **Überwachung der Pareto–Optimalität:** Alle Voraussetzungen für die Anwendung einer Präferenzregel werden zur Rechtfertigung der Optimalität des ausgewählten Operators innerhalb eines TMS verwendet. Sobald eine dieser Voraussetzungen im weiteren Verlauf des Konfigurationsprozesses ihre Gültigkeit verliert, wird die Möglichkeit zur lokalen Optimierung der vorher getroffenen Entscheidung aufgezeigt.
- **Unterstützung des Benutzers:** Wenn der Benutzer die Gründe für eine von ihm getroffene Entscheidung dem System mitteilt, so kann die lokale Optimalität der Benutzerentscheidung auf die gleiche Weise überwacht werden.
- **Interaktive Regeleingabe:** Eine inkrementelle Erweiterung des statischen Präferenzwissens wird dadurch unterstützt, daß der Benutzer eine im Laufe des Konfigurationsprozesses getroffene und von ihm begründete Entscheidung in eine allgemeine Präferenzregel überführen kann. Auf diese Weise wird der Wissenserwerb deutlich erleichtert.

## 8.2 Ausblick

Die Implementation der vorgestellten Komponente zur Präferenzregel–Verarbeitung bei der Konfiguration ist bislang nur mit verhältnismäßig einfachen, eigens zu diesem Zweck konstruierten Beispielen getestet worden. Insbesondere die Effizienz der gewählten Architektur konnte auf diese Weise nicht untersucht werden. Was daher jetzt als dringlichstes Vorhaben ansteht, ist ein Test des Verfahrens in zumindest einer realen Anwendungsdomäne, besser noch in verschiedenen solcher Domänen. Erst durch einen derartigen Praxistest lassen sich die folgenden wichtigen Fragen klären:

- **Wie groß werden die einzelnen Regelmengen?** Davon hängt es ab, ob der Verwaltungsaufwand für eine weitere Aufteilung und Strukturierung der Regelsätze lohnend wäre. Beispielsweise könnte eine Regelmenge nach bestimmten Schlüsselbedingungen, die jeweils im Bedingungsteil mehrerer Regeln auftauchen, in Untermengen aufgeteilt werden. Aus diesen würde dann mit Hilfe der Schlüsselbedingungen, die durch Meta–Regeln ausgedrückt werden, der jeweils passende Regelsatz ausgewählt.

- **Wie sehen die Bedingungen für eine Präferenz aus?** Es muß sich noch herausstellen, ob die Ausdruckskraft der bislang vordefinierten Prädikate ausreicht, um den Anforderungen bei der Formulierung von Bedingungen zu genügen. Wahrscheinlich muß der Satz der vordefinierten Prädikate domänenspezifisch erweitert werden; eventuell kann auch auf einige Konstrukte verzichtet werden.

- **In welcher Hinsicht kann die Effizienz gesteigert werden?** An verschiedenen Stellen innerhalb der Regelkomponente sind unterschiedliche Strategien möglich, von denen die jeweils effizientere nur empirisch ermittelt werden kann. Insbesondere stellt sich dabei die Frage, ob einzelne Fakten und Regeln zwischenzeitlich in der Prolog–Datenbasis gespeichert oder ob sie stets aus der dynamischen Wissensbasis von IDAX erfragt werden sollen. Auch lassen sich komplexe Bedingungs–Prädikate sowohl durch die Kombination von einfacheren Prolog–Prädikaten als auch durch die Programmierung entsprechender Smalltalk–Methoden realisieren, was — in Abhängigkeit von den real auftretenden Anfragen — starke Auswirkungen auf die Effizienz der Abarbeitung haben kann.

- **Wie leicht fällt es, Regeln vor oder während eines Konfigurationsprozesses zu formulieren?** Von der Antwort auf diese Frage hängt es nicht nur ab, ob weitere Eingabehilfen erforderlich sind, sondern auch, wie vollständig oft schwer zu formalisierendes Expertenwissen überhaupt erfaßt werden kann.

- **Inwieweit läßt sich eine lokale Optimierung bei der Konfiguration erzielen?** Es muß zum einen geklärt werden, ob sich lokale Gütekriterien überhaupt ausreichend präzise mit Präferenzregeln formulieren lassen, und zum anderen, wie stark die konsequente Beachtung dieser Kriterien den Aufwand für die Konfiguration in die Höhe treibt.

Wenn sich die Verwendung von Präferenzregeln zur Auswahl von Konfigurationsschritten bewährt, kann man überlegen, ob sich diese Methode auch bei anderen Auswahlentscheidungen im Rahmen des Konfigurationsprozesses sinnvoll einsetzen läßt. Ein offenes Problem ist beispielsweise die Frage, wie sich Konfliktlösungswissen in den Konfigurationsprozeß von IDAX einbringen läßt. Ein Konflikt manifestiert sich dabei in einer Zielblockade, d.h. es gibt eine offene Konfigurationsaufgabe, für die alle Alternativen ausgeschlossen sind. Eine solche Zielblockade wird durch eine bestimmte Anzahl anderer Entscheidungen verursacht, die gemeinsam mit jeder möglichen Lösung der blockierten Aufgabe unverträglich sind.

Von diesen Entscheidungen müssen nun eine oder mehrere ausgewählt und zurückgezogen werden, um die Zielblockade aufzulösen. Damit liegt eine Auswahlentscheidung vor, für deren Lösung zusätzliches Wissen in Form einer Heuristik nützlich sein kann. Dies könnte ein Ansatzpunkt für die Einführung von Präferenzregeln sein, mit deren Hilfe sich gezielt ein besonders erfolgversprechender Entscheidungsrückzug finden ließe. Die Formulierung von Präferenzen ist allerdings sehr viel problematischer als bei der Operatorauswahl, da die Menge der an einer Zielblockade möglicherweise beteiligten Entscheidungen nicht so leicht einzugrenzen ist wie die Konfliktmenge für ein offenes Ziel. Trotzdem erscheinen Präferenzregeln noch am ehesten geeignet, um dieses Problem in den Griff zu bekommen. Sie dienen in diesem Zusammenhang aber nur zur Definition einer Auswahlheuristik; die Überwachung der Gültigkeit ihrer Vorbedingungen im Verlauf des Konfigurationsprozesses ist dementsprechend nicht erforderlich.

Eine weitere wesentliche Frage ist, ob es in Zukunft gelingen wird, Präferenzregeln durch die Analyse erfolgreich vom System erstellter Konfigurationen zu generieren oder sie ohne weiteres Zutun des Experten allein aus dessen exemplarischen Konfigurationsentscheidungen abzuleiten. Beides scheint mit den Methoden des maschinellen Lernens denkbar zu sein, wobei zumindest im Hinblick auf den letztgenannten Aspekt ein erster Schritt durch die interaktive Regelerstellung bereits getan ist.

# Literaturverzeichnis


[BC89] D. Brown, B. Chandrasekaran: *Design Problem Soving: Knowledge Structures and Control Strategies*. Research Notes in Artificial Intelligence, Pitman Publishing, 1989.

[Cha90] B. Chandrasekaran: *Design Problem Solving: A Task Analysis*. AI–Magazine, Vol.11 No.4 (S.59–71), 1990.

[CM84] W. Clocksin, C. Mellish: *Programming in Prolog*. Springer Verlag, 1984.

[CGS91] R. Cunis, A. Günter, H. Strecker (Hrsg.): *Das PLAKON–Buch*. Informatik–Fachberichte 266, Springer–Verlag, 1991.

[Doy79] J. Doyle: *A Truth Maintenance System*. Artificial Intelligence 12 (S.231–272), 1979.

[GR89] A. Goldberg, D. Robson: *Smalltalk-80 — The Language*. Addison–Wesley, 1989.

[Hoc94] H. Hoch: *Konzeption und Implementierung einer begriffshierarchieorientierten Steuerung des Konzeptions- und Konfigurationsprozesses in der Expertensystemshell IDAX*. Diplomarbeit, Universität Kaiserslautern, 1994.

[Kau94] U. Kaul: *Konzeption und Implementierung einer Komponente zur begriffshierarchieorientierten Repräsentation von strukturellem und heuristischen Konfigurationswissen*. Diplomarbeit, Universität Kaiserslautern, 1994.

[Mey94] H. Meyer auf'm Hofe: *Numerische Constraints als Spezialfall sortenspezifischer Constraintpropagierung*. Diplomarbeit, Universität Kaiserslautern, 1994.

[MKK86] T. Mitchell, R. Keller, S. Kedar–Cabelli: *Explanation–Based Generalization: A Unifying View*. Machine Learning 1 (S.47–80), Kluwer Academic Publishers, 1986.

[NL94] A. Niehaus, S. Lorscheid: *Eine Client–Server Kommunikationsschnittstelle zwischen Smalltalk und Prolog für KI–Anwendungen*. Projektarbeit, Universität Kaiserslautern, 1994.

[Pau90] J. Paulokat: *Ein System zur Verarbeitung und Relaxierung von Constraints*. Diplomarbeit, Universität Kaiserslautern, 1990.

[Pau95] J. Paulokat: *Entscheidungsorientierte Rechtfertigungsverwaltung zur Unterstützung des Konfigurationsprozesses in IDAX*. Expertensysteme 95: Beiträge zur 3. Deutschen Expertensystemtagung (S.19–36), infix, 1995.

[Pet91] C. Petrie: *Planning and Replanning with Reason Maintenance*. Dissertation, University of Texas, Austin, 1991.

[Pup90] F. Puppe: *Problemlösungsmethoden in Expertensystemen*. Studienreihe Informatik, Springer Verlag, 1990.

[Reu87] A. Reuter: *Kopplung von Datenbank- und Expertensystemen*. Informationstechnik it 29, Heft 3/1987 (S.164–175), 1987.

[Ric92] M. Richter: *Prinzipien der künstlichen Intelligenz*. B.G. Teubner, 1992.

[Rit92] H. Ritzer: *Konzeption und Implementierung einer TMS–basierten Komponente zur Verwaltung von Abhängigkeiten bei der Konfiguration und Planung*. Diplomarbeit, Universität Kaiserslautern, 1992.

[Wie93] J. Wielemaker: *SWI–Prolog 1.8 — Reference Manual*. Dept. of Social Science Informatics, University of Amsterdam, 1993.