# Debugging und maschinelles Lernen von PROLOG-Programmen

Der Technischen Fakultät der
Universität Erlangen–Nürnberg

zur Erlangung des Grades

D O K T O R – I N G E N I E U R

vorgelegt von

**Gabriella Kókai**

Erlangen — 1998

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

| | |
|---|---|
| Tag der Einreichung: | 11. Mai 1998 |
| Tag der Promotion: | 1. September 1998 |
| Dekan: | Prof. Dr. G. Herold |
| Berichterstatter: | Prof. Dr. H. J. Schneider |
| | Prof. Dr. G. Görz |

**Entdecken heißt sehen, was jeder gesehen hat,**
**und dabei denken, was niemand gedacht hat.**
**Albert Szent-Györgyi**

Für alle, die an mich geglaubt haben!

## Zusammenfassung


Kernpunkt dieser Arbeit ist eine integrierte Methode zum funktionalen Testen und zur Fehlersuche („Debugging“) in logischen Programmen, die in Form des Systems *IDTS* (*Interactive Debugging, Testing and Slicing System*) implementiert wurde. *IDTS* basiert auf dem von Shapiro vorgeschlagenen Debugging-Verfahren, das den Programmierer durch gezielt gestellte Fragen zu Programmfehlern hinführt. *IDTS* ergänzt Shapiros Verfahren um zwei weitere Methoden („*Category Partition Testing*“ und „*Slicing*“) und verbessert die Effizienz der Fehlersuche durch die Verwendung von Testspezifikationen und Testdatenbanken. *IDTS* reduziert die Zahl der beim Debuggen gestellten Fragen dadurch, daß die Benutzereingaben kategorisiert und mit den Testfällen der Testdatenbank verglichen werden. Außerdem wird der Programmabhängigkeitsgraph genutzt, um fehlerfreie Programmteile von vornherein von der Fehlersuche auszuschließen. *IDTS* ist vollständig in PROLOG implementiert und kann mittels einer graphischen Benutzeroberfläche bedient werden.

Weiterhin wird das *SPECG*-System (*Specialized Prolog-based Electrocardiogram*) beschrieben. Das *SPECG*-System kombiniert einen Elektrokardiogramm-Wellenklassifikator mit dem Debuggingsystem *IDTS*. Mit Hilfe dieses Verfahrens wird ein Fehler in einer Klassifikation gefunden, wenn das Klassifikatorprogramm die eingegebenen *EKG*-Wellen nicht analysieren kann.

Die *IDTS*-Methode wird außerdem in eine maschinelle Lernmethode *IMPUT* (*Interactive Multiple Predicate Learner Using Unfolding Transformation*) eingebaut. Das *IMPUT*-System kann eine interaktive Revision der vielfältigen Prädikate eines logischen Programms durchführen. Die Hauptidee beim *IMPUT*-Verfahren ist, eine Verbesserung des Spezialisierungsprozesses durch die Anwendung von *IDTS* zu erreichen.

Schließlich wird gezeigt, daß die Integration des *IMPUT*-Verfahrens in das *SPECG*-System ein Programm ergibt, das dem Benutzer hilft, ein Klassifikatorprogramm zu verbessern, wenn es eine vorgegebene Eingabe nicht klassifizieren kann. Das System kann damit sowohl Syntax- als auch Semantikregeln von *EKG*-Grammatiken lernen. Die hierbei verwendeten Regeln braucht der Anwender, um sogenannte *QRS*-Komplexe von den anderen *EKG*-Wellenformen trennen zu können.

## Danksagung

An erster Stelle möchte ich mich bei Herrn Prof. Dr. H. J. Schneider dafür bedanken, daß er es mir ermöglicht hat diese Arbeit an seinem Lehrstuhl anzufertigen, sowie für die Erstellung des Erstgutachtens.
Mein besonderer Dank gilt weiterhin Herrn Prof. Dr. G. Görz für die Übernahme des Zweigutachtens.
Besondere Anerkennung verdient Dr. Tibor Gyimóthy, der mich bei meiner Entscheidung, dieses Gebiet zu wählen, maßgeblich beeinflußt hat. Er hat sehr viel Zeit auf sich genommen, um meine Arbeit zu betreuen.
Mein Dank gilt meinen Kollegen Zoltán Alexin und László Harmath in Szeged und Christian Niss in Erlangen für die wertvolle Zusammenarbeit und vor allem für die vielen Diskussionen.
Ich schulde meinen Dank den Kollegen Tamás Horváth und Gyöngyi Szilágyi von der József Attila Universität Szeged, die mit mir im ILP-Projekt zusammengearbeitet haben, sowie Herrn Prof. Dr. Attila Kuba, Herrn Prof. Dr. János Csirik und Herrn Prof. Dr. Árpád Makay, die meinen Aufenthalt in Erlangen ermöglicht und unterstützt haben.
Bei allen Kolleginnen und Kollegen am Lehrstuhl für Programmier- und Dialogsprachen sowie Compiler Friedrich-Alexander Universität Erlangen-Nürnberg möchte ich mich für die angenehme Arbeitsatmosphäre und für die rege Diskussionsbereitschaft bedanken.
Frau Ricarda Dormeyer danke ich für das sehr gründliche Korrekturlesen, das mir eine große Hilfe bei der Ausarbeitung der Endfassung dieser Arbeit war. Den Herren Dr. C. Jacob, Dr. M. Minas und Dr. J. Nilson gebührt mein Dank für ihre kritischen und konstruktiven Anmerkungen.
Schließlich schulde ich meinem Freund aufrichtigen Dank dafür, daß er mich in schwierigen Situationen immer wieder zu dieser Arbeit ermutigt hat.

Erlangen, im September 1998 Gabriella Kókai

## Köszönetnyilvánítás

Ezúton szeretném köszönetemet kifejezni Dr. Gyimóthy Tibornak, aki érdeklődésemet ezen terület felé irányította és munkámat az elmúlt évek során messzemenően figyelemmel kísérte.
Hálával tartozom Alexin Zoltánnak és Harmath Lászlónak, akikkel különösen öröm volt együttmüködni.
Köszönöm munkatársaim, Horváth Tamás és Szilágyi Gyöngyi, támogatását, akikkel az ILP projektben együtt dolgozhattam.
Külön hálámat fejezem ki Dr. Kuba Attilának, Dr. Csirik Jánosnak és Dr. Makay Árpádnak, akik erlangeni tartózkodásomat lehetővé tették és messzemenően támogatták.
Végül szeretném megköszönni családom minden tagjának a támogatásukat és a belém helyezett bizalmukat.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

> The presence of bugs in programs can be regarded as a fundamental phenomenom; the bug-free program is an abstract theoretical concept like the absolute zero of thermodynamics, which can be envisaged but never attained.
>
> Jacob Schwartz, in [Sch71]

Alle, die sich schon einmal mit Programmen beschäftigt haben, wissen, daß das *Testen* und *Debuggen* von Software ein komplexes Problem ist. Mehr als fünfzig Prozent der Kosten der Systementwicklung werden durch diese beide Tätigkeiten verursacht ([CR88] und [Bö88]). Testen ist die Durchführung von Programmen mit der Absicht, Fehler zu finden ([Mye79]), und Debugging ist ein Verfahren, bei dem mit Hilfe einer Analyse des Programms die Fehler lokalisiert werden sollen.

Egal, wie vorsichtig und wie gut ausgebildet die Programmierer sind, sie können nicht verhindern, daß sie bei der Programmierung Fehler machen. Shapiro beschreibt dies in seiner Arbeit ([Sha83]) so:

> No matter what language we use to convey, we are bound to make mistakes. Not because we are sloppy and undisciplined, as advocates of some program methologies may say, but because of a much more fundamental reason: we cannot know at any finite point in time, all the consequences of our current assumptions. A program is a collection of assumptions, which can be arbitrarily complex; its behaviour is a consequence of these assumptions; therefore we cannot, in general, anticipate all possible behaviours of a given program. This principle manifests itself in numerous undecidability results, that cover most interesting aspects of program behaviour for any non-trivial programming. [Rog67]

Eine Lösung zur Vermeidung von Fehlern wäre, sicherzustellen, daß diese überhaupt nicht eingebaut werden. Es wird behauptet, daß *Programmverifikation* (*proving program correctness*) die Notwendigkeit von Testen und Debuggen beseitigen könnte. Programmverifikation benötigt das Verhalten des Programms in der Form von Zusicherungen über die Ein- und Ausgabe. Dann kann überprüft werden, ob die Zusicherungen für die Ausgabe zutreffen, wann immer das Programm mit einer Eingabe, die den Zusicherungen für die Eingabe entspricht, durchgeführt wird. Dieser Ansatz funktioniert gut in der Theorie. In der Praxis treten mehrere Probleme auf:

- Für viele in praktischen Anwendungsfeldern auftretende Probleme existiert keine mathematisch exakte Beschreibung.

- Es ist aufwendig, eine Spezifikation einzugeben, die korrekt ist.
- Die Konstruktion eines Programmbeweises kann eine mühsame Aufgabe sein. Im allgemeinen ist es viel schwieriger, einen Beweis zu konstruieren, als das Programm selbst zu verstehen.

Im Fall automatisierter Programmverifikation soll durch mathematische Methoden nachgewiesen werden, daß ein Programm bezüglich einer vorgegebenen Problemspezifikation korrekt ist. Hierbei muß sowohl die Bedeutung der Problemstellung als auch die des Programms mit mathematischen Mitteln exakt festgelegt sein. Jedoch tritt dann das Problem auf, ob die Spezifikation vollständig und fehlerfrei ist. Automatisierte Programmverifikation ist nur für exakt abgegrenzte Anwendungsgebiete oder Lehrsysteme möglich. Mit Hilfe der Spezifikation kann nur die Konsistenz zwischen der Spezifikation und dem tatsächlichen Programm festgestellt werden. Der Programmierer muß beides, die Spezifikation und den Quellcode, überprüfen, um den Fehler zu finden.

Eine andere Möglichkeit wäre, wie Agrawal ([Agr91]) beschreibt, die strukturierte Programmierung. Während sie oft hilft, die Verständlichkeit der Programme zu verbessern, wäre es falsch zu denken, daß diese Methode die Notwendigkeit des Testens und Debuggens vermeidet. Diesen Ansatz formuliert Parnas [Par85] folgendermaßen:

> Even in highly structured systems, surprise and unreliability occur because the human mind is not able to fully comprehend the many conditions that can arise because of the interactions of the components.

Vor diesem Hintergrund wird die Notwendigkeit für ein Debuggingsystem deutlich: Die geschriebenen Programme müssen getestet werden und die so gefundenen Fehler müssen mit Hilfe des Debuggings aufgespürt und behoben werden.

Wie beim Debugging gibt es auch beim Testen verschiedene Methoden: Man kann zum Beispiel verifizieren oder mit Hilfe von Sprachwissen überprüfen. Aber diese Methoden sind ziemlich zeitaufwendig und können damit nicht auf größere Programme angewendet werden. Deswegen ist die Durchführung des Programms mit zutreffenden Testdaten geeignet, um Indizien für die Korrektheit feststellen zu können. Diese Vorgehensweise setzt aber die systematische Generierung von Testdaten voraus, um möglichst viele Fehler sichtbar zu machen.

Besonders bei PROLOG und anderen logischen Programmiersprachen müssen eine Programmierumgebung und Testmöglichkeiten angeboten werden, damit die Sprache als *wirkliche* Programmiersprache akzeptiert wird.

## 1.1 Aufgabenstellung und Lösungsansatz

Unter Berücksichtigung der oben erwähnten Aspekte wird eine integrierte Methode zum Debuggen und funktionalen Testen logischer Programme ausgearbeitet. Die Entwicklungskriterien sind die folgenden:

- Ein vielschichtiges System, das viele Techniken verwendet
- Ein effizientes Debuggingverfahren auf Grund eines hohen Automatisierungsgrades
- Einfache Bedienung
- Anwendbar für reale Programme

- Integration in komplexe Systeme möglich

Unser *IDTS* System kombiniert Shapiros algorithmische Debuggingmethode mit funktionalem Testen (*CPM*) und mit der Methode des Programmslicing. Das Ziel besteht darin, ein effektives System für das Debugging und Testen von PROLOG-Programmen zu entwickeln. Bei der Entwicklung haben wir mit dem Diagnosealgorithmus von Shapiro begonnen, weil die meisten bislang zum Testen und Debuggen von PROLOG-Programme entwickelten Systeme diesen Algorithmus benutzen (APROPOS [Loo88], EDS [Fer87], DED [Llo87a] und ADAss [DNTM88a]).

Das algorithmische Debugging von Shapiro kann drei verschiedene Formen von Fehlern identifizieren, wenn das Programm vorgegeben ist:

- *Terminierung mit inkorrekter Ausgabe*: Das Programm hält, aber das Ergebnis ist falsch.
- *Terminierung ohne Ausgabe*: Das Programm beendet seine Arbeit ohne Ausgabe.
- *Nichtterminierung*: Das Programm läuft möglicherweise in eine Endlosschleife.

Die Algorithmen durchlaufen den Ableitungsbaum des Programms in unterschiedlicher Weise und stellen dem Benutzer Fragen. Diese Fragen beziehen sich auf das vermutete Verhalten der jeweiligen Klauseln. Der Nachteil dieses Debuggingverfahrens ist, daß der Benutzer eine große Anzahl von Fragen beantworten muß.

Unser erster Verbesserungsvorschlag ist die Verwendung einer Testdatenbank. Das Programm kann nicht mit allen möglichen Eingabeparameterbelegungen getestet werden. Deswegen ist das Ziel der *CPM*-Methode, eine Testdatenbank zu generieren und zu benutzen, um die Anzahl der dem Benutzer gestellten Fragen zu reduzieren. Also sind zunächst die *kritischen Eigenschaften* (*categories*) der Parameter der Prädikate zu definieren. Nur die kritischen Eigenschaften werden beim Testen überprüft. Diese Kategorien können außerdem in *Wahlmöglichkeiten* (*choices*) aufgeteilt werden. Aus Kategorien und Wahlmöglichkeiten erzeugt *IDTS* alle möglichen *Testrahmen* (*test frames*), die die eigentliche Testdatenbank bilden.

Als zweiter Verbesserungsvorschlag wird das Slicing in *IDTS* integriert. Das Slicing-Verfahren ist eine allgemeine Methode, mit der versucht wird, die Programmteile, die vermutlich den Fehler enthalten, zu finden. Um die Slice, das Ergebnis des Slicing-Verfahrens, berechnen zu können, wird zuerst statisch der Programmabhängigkeitsgraph erzeugt. Dann wird ein Ableitungsbaum anhand eines gegebenen Zieles generiert. Im Slicing selbst wird bei der vom Benutzer festgestellten fehlerhaften Position begonnen und nach Argumenten gesucht, die aus dieser Position erreicht werden können.

Durch Integration dieser drei Methoden kann ein hoher Automatisierungsgrad erreicht werden, weil die von Shapiros Diagnosealgorithmen gestellten Fragen nur vom Anwender beantwortet werden müssen, wenn sie in der Slice sind (also bei der Berechnung des falschen Ergebnisses eine Rolle spielen) und keine Antwort in der Datenbank vorhanden ist.

Um das System einfach bedienen zu können, wird eine graphische Bedienoberfläche Graphical Interactive Debugging, Testing and Slicing System (*GIDTS*) entwickelt, die alle von *IDTS* angebotenen Funktionen unterstützt. *GIDTS* bietet dem Anwender eine Umgebung an, die während des gesamten Debuggings, beispielsweise durch die Visualisierung des Ableitungsbaums, hilft.

Um die Leistungsfähigkeit des *IDTS*-Systems unter Beweis zu stellen, haben wir das Anwendungssystem *SPECG* (PROLOG for ECG) entwickelt. Dieses System kann *EKG*-Wellenformen klassifizieren. Das Ziel war hier, Regeln zu definieren, die die sogenannten *QRS*-Komplexe von den anderen Mustern einer *EKG*-Wellenformen trennen können. Dieses System hilft dem Benutzer, das Klassifikatorprogramm zu verbessern, wenn es die vorgegebene Eingabe nicht klassifizieren kann.

*IDTS* wird in ein komplexes System, in die *ILP*-Lernmethode *IMPUT* (Interactive Multiple Predicate Learner Using Unfolding Transformation) integriert. Das *IMPUT*-System besteht aus zwei Hauptteilen: Der Spezialisierungsalgorithmus stammt aus dem *SPECTRE*-System ([BIA94]), während der interaktive Debuggerteil das *IDTS*-System ist. So kann *IMPUT* als eine Erweiterung von *IDTS* betrachtet werden. Außerdem ist dieses System ein Teil des von uns entwickelten *SPECG*-Systems, das *EKG*-Wellenformen klassifizieren und lernen kann.

## 1.2 Inhaltsübersicht

Die vorliegende Arbeit gliedert sich in drei Hauptteile:

- *Die Bausteine von* IDTS : Im ersten Teil (Kapitel 2-6) werden die Grundlagen behandelt und das Debugging & Testing System selbst ausführlich beschrieben.

- IDTS *im Vergleich* : Im zweiten Teil (Kapitel 7-8) wird gezeigt, wo unser *IDTS*-System im Vergleich zu anderen Systemen steht: Zuerst werden andere ebenso für das Debugging von PROLOG-Programmen entwickelte Systeme vorgeführt. Danach wird gezeigt, wie gut *IDTS* für die Fehlersuche in realen Programmen anwendbar ist.

- *Integration in komplexe Systeme*: Im dritten Teil (Kapitel 9-10) wird *IDTS* erweitert und als Teil eines Lernprogramms verwendet.

Betrachten wir nun den Inhalt der einzelnen Kapitel etwas genauer:

In Kapitel 2 werden die mathematischen Grundlagen der logischen Programmierung und von PROLOG zusammengefaßt, die für diese Arbeit benötigt werden.

Das von Shapiro entwickelte Debuggingsystem (*APD*) wird in Kapitel 3 vorgestellt; in diesem Kapitel werden die drei verschiedenen in PROLOG auftretenden Fehlerarten (Terminierung mit inkorrekter Ausgabe oder ohne Ausgabe beziehungsweise Nichtterminierung) erläutert. Zusätzlich werden im Fall der Terminierung mit inkorrekter Ausgabe die verschiedenen Traversierungsmethoden (Top-Down, Bottom-Up, Divide-and-Query) des Ableitungsbaums beschrieben. Anhand eines fehlerhaften Beispielprogramms (*compute*) werden diese Traversierungsmethoden bezüglich der Anzahl der dem Benutzer gestellten Fragen miteinander verglichen.

Eine der in dieser Arbeit genannten Erweiterungsmöglichkeiten, *CPM* (Category Partition Method), wird im Kapitel 4 vorgestellt. Zuerst wird die formale Beschreibung des Verfahrens angegeben. Im Rest des Kapitels werden die im Kapitel 3 vorgestellten Diagnosealgorithmen im Fall von Terminierung mit inkorrekter Ausgabe oder ohne Ausgabe erweitert. Das Beispiel *compute* wird auch hier verwendet, um zu zeigen, daß die Anzahl der Fragen reduziert werden kann.

Kapitel 5 enthält eine formale Einführung in das Slicing-Verfahren. Ferner wird eine Erweiterung der Algorithmen von Shapiro um Slicing beschrieben. Hier wird wiederum das Beispiel *compute* als Testbeispiel verwendet.

Kapitel 6 ist eine Zusammenfassung der in den vorherigen Kapiteln beschriebenen Techniken. Außerdem wird hier eine Beschreibung der zum *IDTS*-System entwickelten graphischen Bedienoberfläche angegeben. Das Kapitel enthält zusätzlich ein Unterkapitel, in dem anhand von drei verschiedenen Beispiele (*compute*, *isort* und *convert*) die von den verschiedenen Algorithmen benötigten Fragenanzahlen verglichen werden.

Im zweiten Teil der Arbeit, in Kapitel 7, werden andere, ebenfalls für das Debuggen und Testen von PROLOG-Programmen entwickelte Systeme vorgeführt und kurz beschrieben.

Kapitel 8 demonstriert die Leistungsfähigkeit von *IDTS* anhand eines realen Programms zur Klassifizierung von Elektrokardiogrammwellenformen. Zuerst wird kurz eingeführt, wofür die *EKG*-Diagnostik verwendet werden kann. Die Wellenformen werden zunächst in den Rechner übertragen und digitalisiert. Durch die anschließende Segmentierung der Daten erhält man eine textuelle Darstellung, mit deren Hilfe es möglich ist, eine Grammatik in Form eines PROLOG-Programms zu erzeugen. Das Kapitel schließt mit einem Beispiel, das erläutert, wie *IDTS* in der Grammatik Fehler finden kann.

Das Lernsystem *IMPUT*, das aus *IDTS* und dem Lernverfahren *SPECTRE* besteht, wird in Kapitel 9 untersucht. Dieses Kapitel gibt zuerst einen Überblick über eine neue maschinelle Lernstrategie: Induktive Logische Programmierung (*ILP*). Dann folgt die Beschreibung des Lernverfahrens *IMPUT*, mit dessen Hilfe fehlerhafte Klauseln nicht nur gefunden, sondern auch verbessert werden können. Auf diese Weise können korrekte Programme gelernt werden. Ein Beispiel hierfür wird im letzten Drittel des Kapitels gezeigt. Dieses Verfahren wird hier nicht nur zum Klassifizieren von *EKG*-Wellenformen (siehe Kapitel 8), sondern auch zum Lernen der dazu notwendigen Grammatik verwendet.

In Kapitel 10 werden zum Abschluß die Ergebnisse der Arbeit zusammengefaßt und bewertet. Einige weiterführende Ideen, die im Rahmen dieser Arbeit nicht untersucht werden konnten, werden dargelegt.

# Teil I

# Die Bausteine von IDTS

# Kapitel 2

# Logische Programmierung und PROLOG

Das Interactive Testing, Debugging and Slicing System (*IDTS*) ist ein in PROLOG implementiertes System zum Debugging und Testen von PROLOG-Programmen. In diesem Kapitel werden die theoretischen Grundlagen der logischen Programmierung und der in PROLOG verwendeten Formalismen und Konzepte kurz erläutert. In den folgenden Kapiteln beschreiben wir die Funktionsweise des Systems.

## 2.1 Logische Programmierung

Die Grundidee der logischen Programmierung besteht in der Beschreibung von Relationen zwischen Objekten und in der Verwendung von Regeln, um eine Schlußfolgerung abzuleiten. In der logischen Programmierung werden wohldefinierte Formeln (*well formed formulas*) und Logik erster Ordnung (*first order logic*) benutzt, um eine Lösung für ein Problem zu finden. Eine umfassende Beschreibung des Konzeptes der logischen Programmierung kann man beispielweise in [Llo87b] und [NM90] nachlesen. Hier erklären wir nur die für diese Arbeit notwendigen Begriffe und Notationen. Wir geben zuerst eine formale Beschreibung der Programmiersprache PROLOG an und gehen dabei insbesondere auf die Syntax und Semantik von Klauseln ein.

### 2.1.1 Die Syntax der Formeln

**Definition 1** Ein *Alphabet* $\mathcal{A}$ ist eine Klasse von Symbolen, die folgende Unterklassen enthält:

- *Variablen* sind Bezeichner und werden gemäß der PROLOG-Notation mit großen Anfangsbuchstaben geschrieben.
- *Konstanten* sind Bezeichner oder Zahlen und beginnen mit kleinen Anfangsbuchstaben.
- *Funktoren* besitzen eine Stelligkeit $n$ ($n > 0$) und beginnen mit kleinen Anfangsbuchstaben.
- *Prädikatsymbole* besitzen eine Stelligkeit $n$ ($n \geq 0$) und beginnen mit kleinen Anfangsbuchstaben.

- Die Menge der *logischen Junktoren* umfaßt $\wedge$ (Konjunktion), $\neg$ (Negation), $\leftrightarrow$ (logische Äquivalenz), $\rightarrow$ (Implikation) und $\vee$ (Disjunktion).
- Als *Quantoren* sind $\forall$ (universell) und $\exists$ (existentiell) zugelassen.
- Als *zusätzliche Symbole* werden Komma $(','$) und runde Klammern $('(',')')$ verwendet.

**Definition 2** Die Menge der *Terme* $\mathcal{T}_{\mathcal{A}}$ über einem Alphabet $\mathcal{A}$ ist die kleinste Menge, für die gilt:

- Alle Variablen aus $\mathcal{A}$ sind in $\mathcal{T}_{\mathcal{A}}$.
- Alle Konstanten aus $\mathcal{A}$ sind in $\mathcal{T}_{\mathcal{A}}$.
- Wenn $f$ ein $n$-stelliger Funktor aus $\mathcal{A}$ ist und $t_1, \ldots, t_n \in \mathcal{T}_{\mathcal{A}}$, dann $f(t_1, \ldots, t_n) \in \mathcal{T}_{\mathcal{A}}$.

Um die Schreibweise für Terme einfacher zu halten, verzichten wir im folgenden auf die explizite Angabe des Alphabets und schreiben kurz $\mathcal{T}$ anstatt $\mathcal{T}_{\mathcal{A}}$.

**Definition 3** Die Menge der *wohldefinierten Formeln* (*well formed formulas*) $\mathcal{F}$ wird auf den Termen $\mathcal{T}$ über $\mathcal{A}$ definiert:

- Ist $p$ ein $n$-stelliges Prädikatsymbol aus $\mathcal{A}$ und $t_1, \ldots, t_n \in \mathcal{T}$, so gilt $p(t_1, \ldots, t_n) \in \mathcal{F}$.
- Wenn $F \in \mathcal{F}$ und $G \in \mathcal{F}$, dann sind auch die folgende Ausdrücke Formeln $(\neg F)$, $(F \vee G), (F \wedge G), (F \rightarrow G)$ und $(F \leftrightarrow G)$.
- Wenn $F \in \mathcal{F}$ und $X \in \mathcal{A}$ eine Variable ist, dann $(\forall X F) \in \mathcal{F}$ und $(\exists X F) \in \mathcal{F}$.

Die Formeln in der Form $p(t_1 \ldots t_n)$ heißen *atomare Formeln* oder *Atome*. Um eine PROLOG-ähnliche Notierung zu behalten, schreiben wir $G \leftarrow F$ anstatt $F \rightarrow G$.

**Definition 4** Sei $F$ eine Formel. Eine Variable $X$ in $F$ heißt *gebunden*, wenn die Variable entweder direkt hinter einem Quantor steht oder in einer Teilformel auftritt, die mit einem Quantor $\forall X$ oder $\exists X$ beginnt. Ansonsten wird $X$ als *freie* Variable bezeichnet.

**Definition 5** Eine Formel ohne freie Variablen heißt *geschlossen* (*closed*). Eine Formel beziehungsweise ein Term ohne Variablen heißt *gebunden* (*ground*).

**Definition 6** Seien $X_1, \ldots, X_n$ die in der Formel $F$ vorkommenden Variablen. Ist $F$ von der Form $(\forall X_1), (\forall X_2) \ldots (\forall X_n) F'$ heißt $F$ *universell geschlossen* (*universal closure*). Ist $F$ von der Form $(\exists X_1), (\exists X_2) \ldots (\exists X_n) F'$ heißt $F$ *existentiell geschlossen* (*existential closure*).

**Definition 7** Ein *Literal* ist ein Atom oder ein negiertes Atom. Eine *Programmklausel* notieren wir als A $\leftarrow L_1, \ldots, L_n$, wobei $A$ ein Atom ist und $L_i$ Literale sind. Die rechte Seite der Klausel wird dabei als Konjunktion von Literalen aufgefaßt. Wenn alle $L_i$ Atome sind, dann heißt die Klausel *definit* oder *Horn-Klausel*. Wenn keine $L_1, \ldots, L_n$ angegeben sind, dann wird die Klausel als *Fakt* (*unit-clause*) bezeichnet.

**Definition 8** Ein *logisches Programm* **P** ist eine endliche Menge von Horn-Klauseln $C_k$ der Form $A_k \leftarrow L_{k_1}, \ldots, L_{k_n}$, $0 \leq n$, wobei $A_k$ und $L_{k_i}$ $(1 \leq i \leq n)$ Atome sind. $A_k$ heißt der *Kopf* der Klausel $C_k$, und die $L_{k_i}$ bilden den *Rumpf* der Klausel.

**Definition 9** Ein *Ziel* (*goal*) ist eine Klausel mit leerem Kopf,

$$\leftarrow G_1, \ldots, G_n.$$

wobei die $G_i$ $(1 \leq i \leq n)$ Literale sind. Wenn sämtliche $G_i$ Atome sind, dann heißt das Ziel *definit.*

Nach der oben angegebenen Definition kann man $A \leftarrow L_1, \ldots L_n$ auch lesen als: Wenn alle $L_i$ als Voraussetzungen gelten, dann gilt $A$ als Konsequenz.

Wir illustrieren die Definitionen durch ein kurzes Beispiel.

**Beispiel 1** Die Klausel

$$\mathrm{daughter(X, Y)} \leftarrow \mathrm{female(X), mother(Y, X).}$$

ist eine definite Klausel, aber die Klausel

$$\mathrm{daughter(X, Y)} \leftarrow \mathrm{not\ male(X), father(Y, X).}$$

ist eine (normale) Klausel. Zusammen beschreiben diese Klauseln das Prädikat *daughter/2*, wobei das Prädikat *daughter* zwei Argumente hat. Sie bilden also ein (normales) Programm. Die erste Klausel ist eine abgekürzte Repräsentation der Formel:

$$\forall X \forall Y \ \mathrm{daughter}(X, Y) \vee \overline{\mathrm{female}(X)} \vee \overline{\mathrm{mother}(Y, X)}$$

und wird häufig geschrieben als

$$\{\mathrm{daughter}(X, Y),\ \overline{\mathrm{female}(X)}, \overline{\mathrm{mother}(Y, X)}\}$$

### 2.1.2 Die Semantik der Formeln

In diesem Kapitel definieren wir die Begriffe *Interpretation* und *Herbrand-Modell* logischer Programme. Ausführlicher kann man die Definitionen in [Llo87b] nachlesen.

**Definition 10** Eine *Interpretation* $\mathcal{I}$ über dem Alphabet $\mathcal{A}$ besteht aus einem nicht leeren Definitionsbereich $\mathcal{D}$ und einer Abbildung, die folgende Eigenschaften besitzt:

- Für jedes Konstantensymbol $c \in \mathcal{A}$ gibt es ein Element $c_{\mathcal{I}} \in \mathcal{D}$.
- Für jedes $n$-stellige Funktionssymbol $f \in \mathcal{A}$ gibt es eine Funktion $f_{\mathcal{I}} : \mathcal{D}^n \rightarrow \mathcal{D}$.
- Für jedes $n$-stellige Prädikatsymbol $p \in \mathcal{A}$ gibt es eine Relation $p_{\mathcal{I}} \subseteq \mathcal{D}^n$.

Damit können wir weiter die *Bedeutung* (*meaning*) eines Terms und schließlich die Bedeutung einer Formel definieren. Die Bedeutung der Formel wird als eine Funktion der Komponenten festgelegt. Die Bedeutung einer Formel kann kurz wie folgt geschrieben werden: $\models_{\varphi}^{\mathcal{I}} Q$ ($Q$ ist wahr in $\mathcal{I}$ bezüglich $\varphi$), wobei $Q$ eine Formel, $\mathcal{I}$ eine Interpretation und $\varphi$ eine Auswertung ist.

Weil die Terme Variablen enthalten, muß eine Funktion (*Auswertung, valuation*) zu ihnen definiert werden. Diese Zuordnung ist eine Abbildung der aus dem Alphabet stammenden Variablen auf den Definitionsbereich der Interpretation. Diese Zuordnung ist also eine Funktion $\varphi$, die Objekte aus der Interpretation den Variablen zuordnet. Natürlich können nur geschlossene Formeln eine Bedeutung besitzen.

**Definition 11** Seien $\mathcal{I}$ eine Interpretation, $\varphi$ eine Auswertung und $t$ ein Term. Dann kann die *Bedeutung* $\varphi_{\mathcal{I}}(t)$, die ein Element aus der Interpretation liefert, wie folgt definiert werden:

- Wenn $t$ eine Variable $X$ ist, dann $\varphi_{\mathcal{I}}(t) = \varphi(X)$.
- Wenn $t$ eine Konstante $c$ ist, dann $\varphi_{\mathcal{I}}(t) = c_{\mathcal{I}}$.
- Wenn $t$ die Form $f(t_1 \cdots t_n)$ hat, dann $\varphi_{\mathcal{I}}(t) = f_{\mathcal{I}}(\varphi_{\mathcal{I}}(t_1), \cdots, \varphi_{\mathcal{I}}(t_n))$.

**Definition 12** Seien $\mathcal{I}$ eine Interpretation und $\varphi$ eine Auswertung. Für die geschlossenen wohldefinierten Formeln kann die Bewertung, die eigentlich ein Wahrheitswert ist, folgendermaßen definiert werden:

- Wenn $F$ ein geschlossenes Atom ist, dann ist der Wahrheitswert von $\mathcal{I}$ definiert.
- Wenn $\neg F$, $F \vee G$, $F \wedge G$, $F \rightarrow G$ oder $F \leftrightarrow G$, dann ergibt sich der Wahrheitswert aus folgender Tabelle:

| $F$ | $G$ | $\neg F$ | $F \wedge G$ | $F \vee G$ | $F \rightarrow G$ | $F \leftrightarrow G$ |
|---|---|---|---|---|---|---|
| *true* | *true* | *false* | *true* | *true* | *true* | *true* |
| *true* | *false* | *false* | *false* | *true* | *false* | *false* |
| *false* | *true* | *true* | *false* | *true* | *true* | *false* |
| *false* | *false* | *true* | *false* | *false* | *true* | *true* |

- Wenn $F$ in der Form $\exists X F$ gegeben ist, dann ist der Wahrheitswert *true*, wenn ein $d \in \mathcal{D}$ existiert, so daß $F\delta$ eine geschlossene Formel mit dem Wahrheitswert *true* ist, wobei $\delta = \{X/d\}$ eine Substitution ist, bei der jedes Vorkommen von $X$ durch $d$ ersetzt wird (zum Begriff der Substitution siehe Definition 21).
- Wenn $F$ in der Form $\forall X F$ gegeben ist, dann ist der Wahrheitswert *true*, wenn für alle $d \in \mathcal{D}$ die Formel $F\delta$ eine geschlossene Formel mit dem Wahrheitswert *true* ist, wobei $\delta = \{X/d\}$.

### 2.1.3 Modell und logische Konsequenz

Wenn eine Menge von geschlossenen Formeln $\mathcal{P}$ und eine Interpretation $\mathcal{I}$ gegeben sind, dann kann die Bewertung von $\mathcal{P}$ in $\mathcal{I}$ als die logische Konsequenz festgelegt werden.

**Definition 13** Eine Interpretation $\mathcal{I}$ heißt genau dann ein *Modell* von $\mathcal{P}$, wenn alle Formeln von $\mathcal{P}$ *wahr* in $\mathcal{I}$ sind.

Ganz einfach kann ein Beispiel konstruiert werden, bei der die Menge von Formeln unerfüllbar (*unsatisfiable*) ist. Wenn $\mathcal{P}$ die Formeln $(F \wedge \neg F)$ enthält, wobei $F$ eine beliebige Formel ist, dann gibt es kein Modell von $\mathcal{P}$.

**Definition 14** Eine geschlossene Formel $F$ heißt *erfüllbar* (*satisfiable* oder *consistent*), wenn eine Interpretation $\mathcal{I}$ existiert, die $F$ als *true* bewertet.

**Definition 15** $F$ ist *gültig* (*valid*), wenn alle Interpretationen von $F$ *true* sind.

**Definition 16** Sei $\mathcal{P}$ eine Menge von geschlossenen Formeln. Eine Formel $F$ ist genau dann die *logische Konsequenz* von $\mathcal{P}$ (*logical consequence*, $\mathcal{P} \models \mathcal{F}$), wenn $F$ in allen Modellen von $\mathcal{P}$ *true* ist: Seien $G_1, \ldots, G_n \in \mathcal{P}$ Formeln, dann ist $F$ eine logische Konsequenz von diesen, wenn in allen Interpretationen, für die $G_1 \wedge \ldots \wedge G_n$ erfüllbar sind, $F$ ebenso erfüllbar ist.

**Definition 17** Die Menge $\mathcal{U}_{\mathcal{A}}$ aller gebundenen Terme über dem Alphabet $\mathcal{A}$ heißt *Herbrand-Universum*, wenn $\mathcal{A}$ wenigstens ein Konstantensymbol enthält und $\mathcal{U}_{\mathcal{A}}$ ausschließlich mit Hilfe der Funktions- und Konstantensymbole konstruiert wird.

**Definition 18** Die Menge von gebundenen atomaren Formeln über $\mathcal{A}$ heißt *Herbrand-Basis* $\mathcal{B}_{\mathcal{A}}$ von $\mathcal{A}$.

Natürlich können Herbrand-Universum und -Basis ebenso auf Programmen definiert werden.

**Definition 19** Die *Herbrand-Interpretation* ($\mathcal{I}_{\mathcal{A}}$) eines logischen Programms $\mathbf{P}$ ist eine Interpretation von $\mathbf{P}$, für die gilt:

- Der Definitionsbereich ist das *Herbrand-Universum* $\mathcal{U}_{\mathcal{P}}$.
- Alle Konstanten in $\mathcal{U}_{\mathcal{P}}$ sind sich selbst zugeordnet.
- Für alle Funktionen $f$ mit der Stelligkeit $n$ ist die Abbildung von $(\mathcal{U}_{\mathcal{P}})^n$ auf $\mathcal{U}_{\mathcal{P}}$ die folgende: $(t_1 \ldots t_n) \rightarrow f(t_1 \ldots t_n)$. Also stellt $f$ $n$ gebundene Terme $t_1, \cdots, t_n$ in der Interpretation $\mathcal{I}$ zusammen.
- Für alle Prädikate $p$ mit Stelligkeit $n$ ist $p_{\mathcal{I}}$ eine Teilmenge von $(\mathcal{U}_{\mathcal{P}})^n$.

**Definition 20** Ein Herbrand-Modell einer Menge von Formeln ist eine Herbrand-Interpretation, die ein Modell für alle Formeln in der Menge ist.

**Satz 1** (Lövenheim-Skolem) Sei $\mathcal{I}_{\mathcal{P}}$ ein Modell des definiten Programms $\mathbf{P}$. Die Menge $\mathcal{I} = \{A \in \mathcal{B}_{\mathcal{A}} | \models_{\mathcal{I}} A\}$ ist ein Herbrand-Modell von $\mathbf{P}$.

Dies bedeutet, daß eine logische Formel in der Form $A \leftarrow L_1, \cdots, L_n$ ein Modell hat, wenn sie ein Herbrand-Modell besitzt. Wir können also unsere Untersuchungen auf Herbrand-Modelle beschränken.

Der nächste Satz wurde von van Emden und Kowalski ([vEK76]) bewiesen, die gezeigt haben, daß die Schnittmenge von Herbrand-Modellen wieder ein Herbrand-Modell ist.

**Satz 2** Sei $\mathrm{Mod}_{\mathbf{P}} = \{I_1, I_2, \cdots\}$ eine nicht leere Reihe von Herbrand-Modellen des definiten Programms $\mathbf{P}$. Dann ist die Schnittmenge $I = I_1 \cap I_2 \cap \cdots$ auch ein Herbrand-Modell.

**Satz 3** Das kleinste Herbrand-Modell $\mathcal{M}_{\mathcal{P}}$ eines definiten Programms ist die Menge aller gebundenen atomaren logischen Konsequenzen des Programms, also $\mathcal{M}_{\mathcal{P}} = \{\mathcal{A} \in \mathcal{B}_{\mathcal{P}} | \mathcal{P} \models \mathcal{A}\}$.

Van Emden und Kowalski haben gezeigt, wie das kleinste Herbrand-Modell konstruiert werden kann (siehe [vEK76]). Eine Funktion $T_P(X) : \mathcal{P}(\mathcal{B}_{\mathcal{P}}) \rightarrow \mathcal{P}(\mathcal{B}_{\mathcal{P}})$ wird wie folgt definiert:

$$T_P(X) = \{A | A \leftarrow L_1, \cdots, L_n \in \mathbf{P} \wedge \{L_1, \cdots, L_n\} \subseteq X\}$$

Es kann gezeigt werden, daß eine kleinste Herbrand-Interpretation $\mathcal{I}_{\mathcal{P}}$ existiert, so daß $T_P(\mathcal{I}) = \mathcal{I}$ (also der kleinste Fixpunkt von $T_P$) und $\mathcal{I} = \mathcal{M}_{\mathcal{P}}$, also die kleinste Herbrand-Interpretation mit dem kleinsten Herbrand-Modell übereinstimmt.

### 2.1.4 Substitution

**Definition 21** Eine *Substitution* ist eine endliche (möglicherweise leere) Menge von Paaren in der Form $\{X_1/t_1, \ldots, X_n/t_n\}$, wobei $X_i$ $(1 \leq i \leq n)$ eine Variable und $t_i$ ein Term ist. Zusätzlich gilt $X_i \neq t_i$ und $X_i \neq X_j$, wenn $i \neq j$ ist. Die leere Substitution wird mit $\varepsilon$ bezeichnet.

**Definition 22** Die Klausel $C$ ist eine *Instanz* der Klausel $D$, wenn eine Substitution $\theta$ existiert, so daß $C = D\theta$, wobei $D\theta$ bedeutet, daß auf D die Substitution $\theta$ angewendet wird.

**Definition 23** Wenn eine Substitution $\theta$ die Form $\{X_1/Y_1, \ldots, X_n/Y_n\}$ besitzt, wobei $Y_i$ $(1 \leq i \leq n)$ neue, bisher noch nicht vorkommende, voneinander verschiedene Variablen sind, dann heißt $\theta$ *Umbenennungs-Substitution.*

**Definition 24** Wenn für ein $t \in \mathcal{T}$ $t\theta$ gebunden ist, dann heißt $\theta$ *Grundsubstitution* oder *Instantiierung* für $t$, wobei $t\theta$ eine mit $\theta$ gebundene Instanz von $t$ ist.

**Definition 25** Seien $A$ und $B$ zwei Terme, so daß $A\theta = B\theta$, dann heißt $\theta$ *Unifikator.*

**Definition 26** Eine Substitution $\theta$ ist eine *allgemeinere Substitution* (*more general substitution*) als $\theta'$, wenn ein $\theta''$ existiert, so daß $\theta' = \theta\theta''$, wobei $\theta''$ auch eine Substitution ist $(\theta \succ \theta')$.

**Definition 27** Es gibt den *allgemeinsten Unifikator* (*most general unifier*, $mgu(A, B)$), der abgesehen von Variablenumbenennungen eindeutig ist, für den also gilt: $\theta = mgu(A, B) \Rightarrow (\forall\theta' : A\theta' = B\theta' \Rightarrow \theta \succ \theta')$.

Wie dieser $mgu(A, B)$ berechnet werden kann, erklären wir im Kapitel 2.2.1. Um die oben gegebenen Begriffe besser verstehen zu können, betrachten wir das schon eingeführte *daughter*-Beispiel.

**Beispiel 2** Sei $C$ eine Klausel:

$$\mathrm{C : daughter(X, Y) :\!- \; parent(Y, X).}$$

$\theta = \{X/mary, Y/ann\}$ ist eine Substitution, bei der jeweils $X$ mit $mary$ und $Y$ mit $ann$ ersetzt werden. Man bekommt, wenn $\theta$ auf beide Literale angewendet wird:

$$\mathrm{C}\theta : \mathrm{daughter(mary, ann) :\!- \; parent(ann, mary).}$$

**Definition 28** Eine Klausel $C$ *$\theta$-subsumiert* (*$\theta$- subsumes*) eine Klausel $D$ $(C \geq_\theta D)$, wenn es eine Substitution $\theta$ gibt, so daß $C\theta \subseteq D$. Zwei Klauseln $C$ und $D$ sind *$\theta$-subsumptionsäquivalent* $(C \equiv_\theta D)$, wenn $(C\theta \subseteq D)$ und $(D\theta \subseteq C)$.

**Beispiel 3** Die Klausel $C$ $\theta$-subsumiert die Klausel

$$\mathrm{D : daughter(X, Y) :\!- \; female(X), parent(Y, X).}$$

mit der leeren Substitution $\theta = \varepsilon$, weil die Menge

$$\{\mathrm{daughter(X, Y)}, \overline{\mathrm{parent(Y, X)}}\}$$

eine Teilmenge von

$$\{\mathrm{daughter}(X,Y), \overline{\mathrm{female}(X)}, \overline{\mathrm{parent}(Y,X)}\}$$

ist. Die Klausel $C$ $\theta$-subsumiert die Klausel

$$\mathrm{E} : \mathrm{daughter}(X,X) \text{ :- } \mathrm{female}(X), \mathrm{parent}(X,X).$$

mit der Substitution $\theta = \{Y/X\}$.
Die Klausel $C$ $\theta$-subsumiert die Klausel

$$\mathrm{daughter}(\mathrm{mary},\mathrm{ann}) \text{ :- } \mathrm{female}(\mathrm{mary}), \mathrm{parent}(\mathrm{ann},\mathrm{mary}), \mathrm{parent}(\mathrm{ann},\mathrm{tom}).$$

mit der Substitution $\theta = \{X/mary, Y/ann\}$.

**Definition 29** $C$ ist eine *gemeinsame Instanz* (*common instance*) von $A$ und $B$, wenn es Substitutionen $\theta_1$ und $\theta_2$ gibt, so daß $C = A\theta_1 = B\theta_2$.

**Definition 30** Sei $\theta := \{X_1/Y_1, \ldots X_n/Y_n\}$ eine Umbenennungssubstitution Dann heißt $\{Y_1/X_1, \ldots Y_n/X_n\}$ *Inverse* von $\theta$ und wird mit $\theta^{-1}$ bezeichnet.

**Satz 4** Wenn $\theta$ eine Umbenennungssubstitution ist, dann ist $\theta^{-1}$ auch eine Umbenennungssubstitution und $\theta^{-1}\theta = \theta\theta^{-1} = \varepsilon$.

## 2.2 Die Programmiersprache PROLOG

Traditionelle imperative Programmiersprachen können wie folgt formuliert werden:

PROGRAMM = ALGORITHMUS + DATENSTRUKTUR.

Entgegen der oben erwähnten imperativen Programmierung kann der Grundgedanke von PROLOG wie folgt beschrieben werden:

ALGORITHMUS = LOGIK + STEUERUNG.

Die LOGIK in dieser 'Gleichung' gibt an, welches Wissen und welche Zielsetzung der Algorithmus hat. Die Steuerung definiert, wie diese einzusetzen ist ([Kow79]). Die LOGIK sagt, worin das Problem besteht, und die STEUERUNG beschreibt, wie es gelöst werden kann.

Die Realisierung neuer Systeme aus dem Bereich der künstlichen Intelligenz hat zur Entwicklung neuer Sprachen, wie beispielsweise LISP, PROLOG oder OPS5 geführt. Diese Sprachen gehören zu den *logischen Programmiersprachen*, die auch als *nicht-deterministische* oder *deklarative* Programmiersprachen bezeichnet werden können.

Die Sprache PROLOG (Programming in Logic) wurde Anfang der Siebziger Jahre von einer Reihe von Wissenschaftlern unter der Leitung von A. Colmerauer an der Universität Marseille ausgearbeitet und implementiert ([RGdA75]). Natürlich wurden weitere Implementierungen für neu entwickelte Rechner, wie auf DEC-10 die Edinburgh-Implementierung von Pereira (et al [PBB+78]) oder die erste UNIX-Implementierung von Bundy ([Bun79]), erarbeitet.

Die theoretischen Grundlagen der Sprache, die ihre Prinzipien von der logische Programmierung bekommen hat, wurden unter anderem von R. Kowalski [Kow74] durchgeführt. Die Grundideen (zum Beispiel Theorem Proving) wurden als Folge der Arbeit von Robinson [Rob65] entwickelt. Mehr über die Geschichte von PROLOG und eine Einführung in die Sprache kann in [Cla82], [Coh85], [SS86],[Bra86] und [NM90] nachgelesen werden.

Die Bausteine von PROLOG-Programmen sind die folgenden:

- Klauseln (*non unit clauses*),
- Fakten (*unit clauses*),
- Anfragen (*goals*).

Wie wir früher definiert haben, ist ein logisches Programm eine endliche Menge von *(Horn-)Klauseln.* Eine Klausel oder *Regel* (*rule*) ist ein universell quantifizierter logischer Satz (*universally quantified logical sentence*) und hat die folgende Form:

$$F \leftarrow G_1, \ldots, G_n.$$

dabei sind $F$ ein Atom und $G_i$ Literale.

Obige Zeile kann so gelesen werden, daß $F$ durch die Konjunktion der $G_i$ implementiert wird, oder daß man, um $F$ beantworten zu können, die Konjunktion von $G_1, \ldots, G_n$ beantworten muß.

Wenn $n = 0$ ist, dann sprechen wir von einem *Fakt.* Fakten können als allgemeine Tatsachen interpretiert werden. Sie enthalten eine beliebige Zahl von gebundenen Variablen und ein Prädikat.

Eine *Anfrage* besitzt die Form:

$$G_1, \ldots, G_n. \text{ für } n > 0.$$

Sie ist eine Konjunktion von Zielen $G_i$ $(1 \leq i \leq n)$. Eine Anfrage ist eine Horn-Klauseln, die kein positives Literal enthält. Wenn $n = 1$ ist, dann spricht man von einer *einfachen Anfrage* (*simple goal*). Für $n \geq 2$ nennen wir eine Anfrage *zusammengesetzte* Anfrage. Wenn Variablen in der Anfrage vorkommen, dann sind sie implizit existentiell quantifiziert. Eine Anfrage beantworten zu können (wenn ein Programm schon vorhanden ist) bedeutet, daß diese Anfrage logische Konsequenz des Programms ist. Wenn alle $G_i$ gebundene Ziele sind, dann wird die Anfrage mit *yes* beantwortet. Wenn alle $G_i$ gebunden und mit Hilfe des Programms **P** ableitbar sind, dann wird mit *yes* geantwortet. Eine oder mehrere Variablen können in mehreren Zielen vorkommen (*gemeinsame*, *shared* Variable). Dies bedeutet, daß eine oder mehrere Variablen in zwei verschiedenen Zielen vorkommen können. In diesem Fall ist die Anfrage logische Konsequenz des Programms **P**, wenn alle Ziele in der Konjunktion logische Konsequenz sind und die verteilten Variablen mit dem gleichen Wert instantiiert werden. Formal formuliert: Die Lösung der $G_1, \ldots, G_n$ bei Verwendung von **P** ist die Suche nach der Substitution $\theta$, so daß $G_1\theta, \ldots, G_n\theta$ gebundene Vorkommen von Fakten in **P** sind.

Diese Substitution kann durch Verwendung von *Ableitungsregeln* (*deduction rules*) bestimmt werden. Die einfachste Regel ist die *Identifizierung.* Die nächste Ableitungsregel ist die *Generalisierung.* Diese Regel wird im nächsten Kapitel näher erklärt. Eine Anfrage $G_1, \ldots, G_n$ ist eine logische Konsequenz ihrer Instanz $G_1, \ldots, G_n\theta$ bei einer beliebigen Substitution $\theta$. Eine nicht gebundene Anfrage mit Hilfe des Programms beantworten zu können, bedeutet, diejenigen Fakten zu finden, die Instanzen der Frage sind. Die Antwort sind diese Instanzen. Falls keine solchen Instanzen existieren, ist die Antwort *no.* Es kann vorkommen, daß mehrere Lösungen existieren.

PROLOG berechnet die Antwort so, daß es den sogenannten *SLD-Baum* ([Rob65], siehe nächstes Unterkapitel) durchläuft. Wenn das gerade bearbeitete Ziel zu verschiedenen Klauselköpfen paßt, dann verzweigt sich der SLD-Baum. Um alle Lösungen finden zu können, wird die Sackgassenbehandlung (*Rücksetzen* und *Wiederanlauf*, *backtracking*) der sequentiellen Abarbeitung verwendet.

### 2.2.1 SLD-Resolution

In der SLD-Resolution wird der allgemeinste Unifikator (mgu, siehe Definition 27) benutzt. Deswegen wird zunächst definiert, wie dieser berechnet werden kann. Dazu führen wir einige neue Definitionen ein. Wie wir schon früher erwähnt haben, wurde das *Resolutions-Prinzip* von Robinson [Rob65] ausgearbeitet und dessen Korrektheit von Hill [Hil74] bewiesen. Die *Ableitungs-Prozedur* wurde als erstes von Kowalski [Kow74] beschrieben. Die hier kurz erläuterte *SLD-Resolution* (*linear resolution with selection function for definite clauses*) macht es möglich, eine korrekte Konsequenz eines Programms zu bekommen, was die Basis für die operationelle Semantik eines Programms liefert. Mehr über Automatic Theorem Proving ist bei [Bib82], [CL73], [Lov69] oder [Man74] nachzulesen.

**Definition 31** Das *Unterscheidungspaar* $< t_1, t_2 >$ (*disagreement*) von Strukturen $s_1$ und $s_2$, die entweder Terme oder Atome sind, kann wie folgt definiert werden:

- Wenn $s_1$ und $s_2$ verschiedene Anfangssymbole haben, dann ist $< t_1, t_2 >=< s_1, s_2 >$ das einzige Unterscheidungspaar von $s_1$ und $s_2$.

- Wenn $s_1$ und $s_2$ das gleiche Anfangssymbol haben und die $i$-te Struktur von $s_1$ und $s_2$ verschieden ist, aber alle vorhergehenden Strukturen $(1 \cdots i-1)$ gleich sind, dann ist das Unterscheidungspaar dieser Unterstrukturen in der $i$-ten Komponente auch das Unterscheidungspaar von $s_1$ und $s_2$.

Wenn $\theta$ ein $mgu$ für $s_1$ und $s_2$ und $< t_1, t_2 >$ ihr Unterscheidungspaar ist, dann muß $\theta$ auch ein Unifikator für $t_1$ und $t_2$ sein. Also ist der $mgu$ von $s_1$ und $s_2$ eine Komposition aus dem $mgu$ $\delta$ für $t_1$ und $t_2$ und dem $mgu$ für $s_1\delta$ und $s_2\delta$. Wenn das Unterscheidungspaar $< t_1, t_2 >$ betrachtet wird, dann muß einer der folgenden Fälle auftreten:

- *Fail*: weder $t_1$ noch $t_2$ ist eine Variable, dann bedeutet dies nach Definition 31, daß die Anfangssymbole von $t_1$ und $t_2$ verschieden sind. Daraus folgt: Es gibt keinen Unifikator.

- *Occur check*: Entweder $t_1$ oder $t_2$ ist eine Variable ($< t_1 t_2 >=< X, t >$ oder $< t_1, t_2 >=< t, X >$), und $X$ kommt als eine Teilstruktur in $t$ vor. Dann gibt es keinen Unifikator.

- *Success*: Es gibt eine Substitution $\{X/t\}$, die den allgemeinsten Unifikator $mgu(t_1, t_2)$ liefert.

Im folgenden beschreiben wir den Algorithmus, der den allgemeinsten Unifikator $mgu$ liefert. Den allgemeinsten Unifikator braucht man, um eine SLD-Ableitung konstruieren zu können. Das Verfahren wird immer mit $mgu(s_1, s_2, \varepsilon)$ aufgerufen.

**Algorithmus 1** Die Berechnung des allgemeinsten Unifikators

```
Eingabe: Terme s1 und s2
Ausgabe: Der allgemeinste Unifikator Θ, falls dieser existiert, oder fail
procedure mgu(s1, s2, Θ)
begin
  while (s1 ≠ s2) do
    θ := mgu_of_disagreement_pair(s1, s2)
    if θ = fail then return(fail)
    else
      Θ' := Θ ∪ θ;
      s1' := s1θ;
      s2' := s2θ;
      return mgu(s1', s2', Θ')
    endif
  endwhile
  return(Θ)
end
```

**Algorithmus 2** Suche nach der Substitution, wenn $s_1$ und $s_2$ das Unterscheidungspaar $t_1$ und $t_2$ haben

```
Eingabe: Terme s1 und s2
Ausgabe: eine Substitution θP für t1 und t2
function mgu_of_disagreement_pair(s1, s2) : θP
  begin
    let < t1, t2 > disagreement_pair of s1 and s2
    case < t1, t2 >
        Fail: θP := fail return(θP);
        Occurcheck: θP := fail return(θP);
        Success:
          if variable(t1) then θP = {t1/t2}; return(θP)
          else θP = {t2/t1}; return(θP)
          endif
    endcase
end
```

Jetzt können wir die Grundtechnik der SLD-Resolution, die die Basis für logische Programme bildet, erklären. Dabei werden die *Inferenzregeln* (*resolution principle*), die im Fall von definiten Programmen definiert werden, verwendet. Um diese Methode formal einführen zu können, definieren wir einige Begriffe.

**Definition 32** Seien **P** ein Programm, $G_0$ ein Ziel in der Form $\leftarrow A_1, \ldots, A_n$ und $C : A \leftarrow B_1, \ldots, B_m$ eine Klausel in **P**. Eine *SLD-Resolution* ist die Berechnung eines neuen Ziel $G_1$, wobei die folgenden Bedingungen erfüllt sein müssen:

- $A_1, \ldots, A_n$, und $A, B_1, \ldots, B_m$ sind Atome.
- $\theta$ ist der allgemeinste Unifikator von $A$ und $A_k$ (ausgewähltes Atom) für ein $k$ ($1 \leq k \leq n$), wobei eine Berechnungsregel eindeutig definiert, welches $k$ ausgewählt wurde.
- $G_1$ hat die Form $\leftarrow A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n$.
- $G_1$ wird auch als *Resolvent* von $G_0$ und $C$ bezeichnet.

**Definition 33** Seien $\mathbf{P}$ ein definites Programm und $G$ ein definites Ziel. Eine *SLD-Ableitung* von $\mathbf{P} \cup \{G\}$ ist eine (endliche oder unendliche) Sequenz von Zielen $G_0, G_1, \ldots$, eine Sequenz von Varianten von Klauseln $C_1, C_2 \ldots$ sowie eine Sequenz allgemeinster Unifikatoren $\theta_1, \theta_2, \ldots$, (siehe Algorithmus 1), so daß $G_{i+1}$ aus $G_i$ und $C_{i+1}$ unter Verwendung von $\theta_{i+1}$ (i = 1, 2, $\cdots$) berechnet wird.

Es gibt zwei verschiedene Gründe, warum $G_{i+1}$ nicht mehr berechenbar sein könnte:

- Wenn das ausgewählte Teilziel mit keinem Kopfelement einer Klausel aus dem Programm unifiziert werden kann (es gibt keinen Resolventen), heißt dies *endlicher Fehlschlag* (*finitely fail.*)
- Wenn $G_i = \Box$ (leeres Ziel) ist, dann ist die *Ableitung* (*refutation*) erfolgreich und heißt SLD-Ableitung des Anfangszieles $G_0$. Die Länge der Ableitung ist $i$.

Eine SLD-Ableitung kann grafisch dargestellt werden (vgl. Abbildung Seite 42 in [Llo87b]).

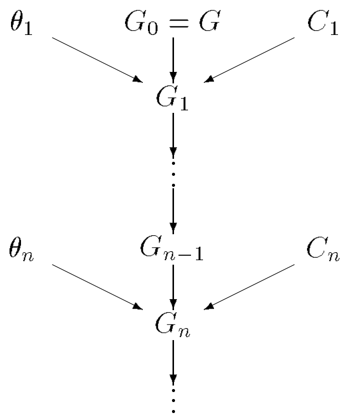



Abbildung 2.1: Eine SLD-Ableitung

Um das Resolutions-Prinzip zu illustrieren, benutzen wir wieder das Familien-Beispiel.

**Beispiel 4** Die folgenden Fakten sind vorgegeben:
$F_1$: `female(mary).`

$F_2$: `parent(ann, mary).`

und die Klausel C
$C$: `daughter(X,Y)  :-  female(X), parent(Y,X).`

Nehmen wir an, daß die gestellte Frage die Form hat:
$G$: `?- daughter (mary, ann).`

Um das Ziel ableiten zu können, gehen wir wie folgt vor:

- Zuerst wird der Resolvent $G_1 = res(G, C)$ mit Hilfe der Substitution $\theta_1 = \{X/mary, Y/ann)$ berechnet. Das bedeutet, daß die Substitution $\theta$ auf die Klausel $C$ angewendet wird, und so entsteht:

  $G_1$:`female(mary), parent(ann,mary).`

  die schließlich mit $F_1$ resolviert.

- Der nächste Resolvent $G_2$ ist dann $G_2$ `= res(`$G_1, F_1$`) = parent (ann, mary).`

- Das Ziel $G_2$: `parent(ann, mary)` und der Fakt `parent(ann,mary)` haben als Resolvent $e =$ res $(G_2, F_2) = \Box$.

Den Resolutionsbaum zeigt die Abbildung 2.2. Das Anfangsziel ist `daughter (mary, ann)`. Dies kann mit der linken Seite von $C$ unter der Substitution $\theta_1 = \{X/mary, Y/ann\}$ unifiziert werden. Dies bedeutet, daß das nächste Ziel die linke Seite der Klauseln wird, wobei $X$ mit $mary$ und $Y$ mit $ann$ ersetzt werden. Dann kann festgestellt werden, daß das erste Atom genau den Fakt $F_1$ liefert, also reduziert sich das Ziel auf `parent(ann, mary)`. Dies entspricht genau dem zweitem Fakt. So haben wir $\Box$ als Ergebnis der Ableitung bekommen, was bedeutet, daß die Ableitung erfolgreich war.

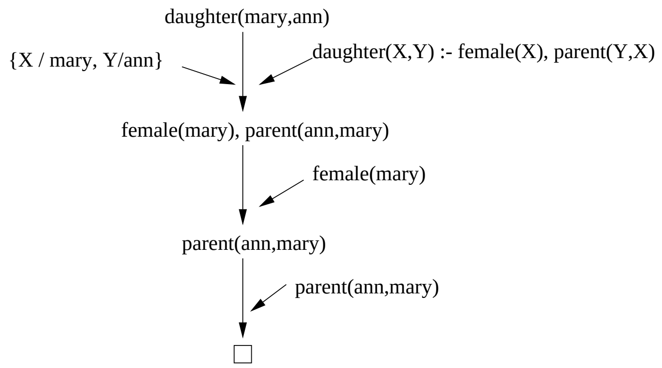



Abbildung 2.2: Eine SLD-Ableitung für das Familienbeispiel

### 2.2.2 Die Korrektheit der SLD-Resolution

**Definition 34** Sei **P** ein definites Programm. Die *Erfolgsmenge* (*success*) von **P** ist die Menge aller $G \in \mathcal{B}_\mathcal{P}$, so daß $\mathbf{P} \cup \{G\}$ eine SLD-Ableitung hat.

Diese Menge ist die genaue prozedurale Entsprechung des letzten Herbrand-Modells (Beweis siehe [Llo87b]). Aus der *Gültigkeit* (*soundness*) und der *Vollständigkeit* (*completeness*) der SLD-Ableitung folgt, daß die Erfolgsmenge und das letzte Herbrand-Modell eines Programms $\mathbf{P}$ gleich sind. Gleiches kann für die korrekte Antwort definiert werden.

**Definition 35** Seien $\mathbf{P}$ ein definites Programm und $G$ ein definites Ziel. Die *berechnete Antwort* $\theta$ (*computed answer*) für $\mathbf{P} \cup \{G\}$ ist die Substitution, die man durch Beschränkung der Komposition von $\theta_1, \ldots, \theta_n$ auf die Variablen von $\mathbf{P}$ erhält, wobei $\theta_1, \ldots, \theta_n$ die Sequenz der mgu's aus der SLD-Ableitung von $\mathbf{P} \cup \{G\}$ sind.

Es kann gezeigt werden, daß die SLD-Resolution *korrekt* (*correct*), *gültig* (*sound*, wenn nur gültige Formeln ableitbar sind) und *vollständig* (*complete*, wenn alle gültigen Formeln ableitbar sind) ist.

**Satz 5** Seien $\mathbf{P}$ ein definites Programm und $G$ ein definites Ziel. Dann sind alle für $\mathbf{P} \cup \{G\}$ berechneten Antworten gültig für $\mathbf{P} \cup \{G\}$.

**Satz 6** Seien $\mathbf{P}$ ein definites Programm und $G$ ein definites Ziel. Für alle korrekten Antworten $\theta$ in $\mathbf{P} \cup \{G\}$ gibt es eine für $\mathbf{P} \cup \{G\}$ berechnete Antwort $\delta$ und eine Substitution $\gamma$, so daß $\theta = \delta\gamma$ ist.

Der vollständige Beweis dieser zwei Sätze kann in [Llo87b] (Seite 47 ff.) nachgelesen werden. Die Korrektheit der SLD-Resolution ist direkt ableitbar aus der Korrektheit der von Robinson [Rob65] definierten Resolution. Das gleiche gilt jedoch nicht für die Inkorrektheit: Die SLD-Resolution ist inkorrekt für universelle Klauseln, aber sie ist korrekt für Horn-Klauseln. SLD-Ableitungen sind lineare Beweise, während im allgemeinen Beweise Resolutionsbäume sind. Dies bedeutet, daß die SLD-Resolution sehr effizient ist, jedoch negative Informationen in Horn-Klauseln nicht ausgedrückt werden können.

Unser Ziel ist, eine Strategie zu definieren, mit der eine Ableitung in einem Programm gefunden werden kann. Es bedeutet, daß eine Suchstrategie definiert werden muß. Der Suchraum (*search space*) ist der *SLD-Baum*.

**Definition 36** Seien $\mathbf{P}$ ein definites Programm und $G$ ein definites Ziel. Ein *SLD-Baum* für $P \cup \{G\}$ ist ein Baum, der die folgenden Bedingungen erfüllt:

- Alle Knoten im Baum sind definite Ziele (kann leer sein).
- Die Wurzel ist $G$.
- Sei $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ $(n \leq 1)$ ein Knoten im Baum und nehmen wir an, daß das ausgewählte Atom $A_k$ ist, dann hat für alle Programm-Klauseln $A \leftarrow B_1, \ldots, B_n$, wobei $A$ und $A_k$ durch den mgu $\theta$ unifizierbar sind, dieser Knoten einen Nachkommen $\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\theta$.
- Knoten, die die leere Klausel enthalten, haben keine Kinder.

Jeder Zweig im SLD-Baum ist eine Ableitung. Zweige, die zu einer erfolgreichen Ableitung führen, heißen *erfolgreiche Zweige* (*success branches*), diejenigen, die unendliche Ableitungen enthalten, heißen *unendliche Zweige* (*infinite branches*), und Zweige, die mit fail enden, heißen *erfolglose Zweige* (*failure branches*).

Wie früher erwähnt wurde, muß eine sogenannte *Berechnungsregel* (*computation rule*), die das ausgewählte Atom bestimmt, definiert werden.

**Definition 37** Seien **P** ein definites Programm und $G$ ein definites Ziel. $R$ ist eine *Berechnungsregel* des SLD-Baums, mit der das Atom in dem SLD-Baum für $\mathbf{P} \cup \{G\}$ selektiert wird.

Wie Lloyd bewiesen hat, ist die Existenz der Ableitung eines Programm mit einem Ziel von der Berechnungsregel unabhängig (siehe [Llo87b], Seite 51f).

**Satz 7** Seien **P** ein definites Programm und $G$ ein definites Ziel. Nehmen wir an, daß die SLD-Ableitung von $\mathbf{P} \cup \{G\}$ die Antwort $\delta$ zurückliefert. Dann gibt es für alle Berechnungsregeln $R$ eine SLD-Ableitung von $\mathbf{P} \cup \{G\}$ durch $R$, wobei die berechnete Antwort $\delta'$ ist, und es gilt, daß $G\delta'$ eine Variante von $G\delta$ ist.

Lloyd hat auch gezeigt, welchen Einfluß die Auswahl der Berechnungsregel auf die Struktur und Länge des SLD-Baums hat.

**Beispiel 5** Um die oben definierten Begriffe zu erklären, nehmen wir hier ein modifiziertes Familien-Beispiel. Die Klauseln sind die folgenden:

$C_1$: `ancestor(X,Y)  :-  parent(X,Y).`
$C_2$: `ancestor(X,Y)  :-  ancestor(X,Z), ancestor(Z,Y).`

und zusätzlich definieren wir die folgenden Fakten:

$F_1$: `parent(katherina,frank).`
$F_2$: `parent(katherina,bertrand).`
$F_3$: `parent(bertrand,kate).`
$F_4$: `parent(bertrand,john).`

Der zu diesem Beispiel und dem Ziel `ancestor(katherina,Y)` gehörende SLD-Baum wird in Abbildung 2.3 gezeigt.

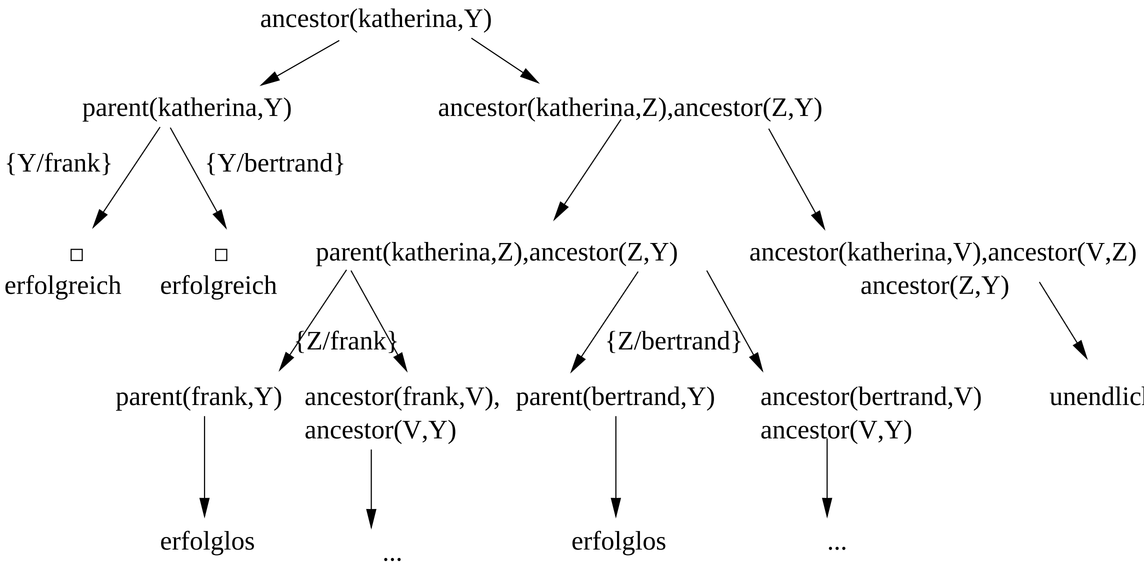



Abbildung 2.3: Der SLD-Baum zu dem Programm *ancestor*

Ein Blatt im SLD-Baum ist entweder das leere Ziel oder ein endlicher Fehlschlag, weil das ausgewählte Atom mit keiner Klausel aus dem Programm unifizierbar ist. Als dritter Fall kann vorkommen, daß die Ableitung unendlich wird.

Ein allgemeines Ergebnis ist, daß für $\mathbf{P} \cup \{G\}$ entweder alle SLD-Bäume unendlich viele erfolgreiche Zweige haben, oder alle haben die gleiche Anzahl endlicher Zweige (siehe [Llo87b] Seite 56f).

### 2.2.3 Ableitungsbaum in einem PROLOG-Programm

Bei der logischen Programmierung wurde der SLD-Baum als ein Werkzeug definiert, das gegebene Ziel auszuwerten. Der Nachteil dieser Methode ist, daß es keine unmittelbare Korrespondenz zwischen dem Teilziel im Körper der Klausel und dem selektierten Teilziel im Zweig des SLD-Baums gibt. Eine besser geeignete Vorgehensweise für die Beschreibung von Fragen könnte der von Clark ([Cla82]) eingeführte *Ableitungsbaum* sein. Der Vorteil dieser Repräsentation ist, daß der Baum genau zeigt, wie die Ableitung erfolgt. Die folgenden Definitionen sind aus der Arbeit von Shapiro übernommen (siehe [Sha83] Seite 34f).

**Definition 38** Das Tripel $< C, x, y >$ ist eine *Instanz* des Kopfelementes $C$ einer Klausel, die mit $x$ als Eingabe aufgerufen $y$ als Ausgabe liefert.

**Definition 39** Sei $C$ eine beliebige Klausel in einem Programm $\mathbf{P}$, so daß $C$ mit der Eingabe $x$ terminiert und die dazugehörige Ausgabe $y$ ist. Zusätzlich sei $C$ von der Form $C \leftarrow C_1, \ldots, C_n$. Das *top level trace* des Tripels $< C, x, y >$ ist eine geordnete Menge $\{< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, \ldots, < C_n, x_n, y_n >\}$. Wenn $C$ mit $x$ als Eingabe aufgerufen wird, dann wird zuerst $C_1$ mit $x_1$ aufgerufen und gibt als Ausgabe $y_1$ zurück. Es folgt $C_2$ mit Eingabe $x_2$ und so weiter. Am Ende ruft $\mathbf{P}$ $C_n$ mit Eingabe $x_n$ auf und gibt $y_n$ zurück, so daß $\mathbf{P}$ mit $y$ als Ausgabe endet.

**Definition 40** *Ein partieller Ableitungsbaum* von $\mathbf{P}$ (*partial computation tree*) ist ein geordneter Baum. Alle Knoten in diesem Baum sind mit einem Tripel $< C_j, u, v >$, $j = 1, 2, \ldots$ markiert. Die Menge der direkten Nachkommen eines inneren Knotens ist der *legal top level trace* dieses Knotens. Der Baum $\mathbf{T}$ ist *ein vollständiger Ableitungsbaum* von $\mathbf{P}$, wenn er ein partieller Ableitungsbaum ist und alle Blätter von $\mathbf{T}$ die leere Menge sind.

**Definition 41** Ein *Programm* $\mathbf{P}$ ist *partiell korrekt* in der Interpretation $\mathcal{I}$, wenn die Wurzeln aller vollständigen Ableitungsbäume in $\mathcal{I}$ sind.

**Definition 42** Ein *Programm* $\mathbf{P}$ ist *vollständig* in der Interpretation $\mathcal{I}$, wenn alle Tripel in $\mathcal{I}$ Wurzel eines kompletten Ableitungbaums von $\mathbf{P}$ sind.

**Definition 43** Ein *Programm* $\mathbf{P}$ *terminiert immer*, wenn kein *unendlich erreichbarer* (*infinite reachable*) Ableitungsbaum dafür existiert, ansonsten heißt der Ableitungsbaum divergierend (*diverging*).

**Definition 44** Ein *Programm* ist *vollständig korrekt*, wenn es partiell korrekt und vollständig ist und immer terminiert.

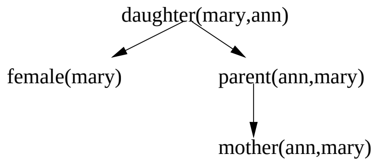



Abbildung 2.4: Der zum Ziel *daughter(mary, Y)* gehörende Ableitungsbaum

**Beispiel 6** Um sich einen Ableitungsbaum grafisch vorstellen zu können (siehe Abbildung 2.4), nehmen wir die in Beispiel 1 schon verwendeten, aber hier leicht modifizierten Klauseln:

$C_1$: `daughter(X,Y)  :-  female(X), parent(Y,X).`
$C_2$: `parent(Y,X)  :-  mother(Y,X).`
$C_3$: `parent(Y,X)  :-  father(Y,X).`

und die Fakten:

$F_1$: `female(ann).`
$F_2$: `female(eve).`
$F_3$: `female(mary).`
$F_4$: `female(sue).`
$F_5$: `mother(eve,sue).`
$F_6$: `mother(ann,mary).`
$F_7$: `father(tom,mary).`
$F_8$: `father(tom,sue).`

und versuchen das Ziel

$G$:`daughter(mary,Y).`

abzuleiten.

Wenn wir aber die zwei Klauseln $C_2$ und $C_3$ vertauschen, also

$C_2$: `parent(Y,X)  :-  father(Y,X).`
$C_3$: `parent(Y,X)  :-  mother(Y,X).`

und mit dem gleichen Ziel das Programm aufrufen, bekommen wir ein anderes Ergebnis, wie uns der Berechnungsmechanismus von PROLOG demonstriert. PROLOG nimmt immer das am weitesten links stehende Atom, und hierzu wird die erste Möglichkeit, mit der dieses das aktuelle Ziel (das selektierte Atom) unifizieren kann, gesucht, und setzt damit die Tiefensuche fort. Den neuen Ableitungsbaum zeigt Abbildung 2.5.

Um eine in der Praxis gut verwendbare Programmiersprache zu implementieren, die auf dem Berechnungsmodell der logischen Programmierung basiert, wurde dieses Modell um einige *metalogische* und *extra-logische* Eigenschaften erweitert. Außerdem wurden einige Eigen-

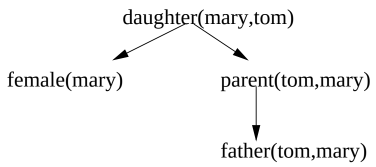



Abbildung 2.5: Der neue aber auch zum Ziel *daughter(mary, Y)* gehörende Ableitungsbaum

schaften eingebaut, die von Rechnern unterstützt werden, wie zum Beispiel Arithmetik und schnelle Ein- und Ausgabe. Schließlich wurde in PROLOG ein Ableitungsmechanismus als konkrete Realisierung eines abstrakten Berechnungsmodells implementiert. Somit wurde der Ableitungsmechanismus von der logischen Programmierung auf eine konventionelle Maschine übertragen.

Wenn PROLOG die in Algorithmus 1 beschriebene Ableitung durchführt, wird immer das am weitesten links stehende Ziel ausgewählt, anstatt ein beliebiges auszusuchen. PROLOG ersetzt die nichtdeterministische Wahl einer Klausel durch eine sequentielle Suche nach Klauseln, die unifizierbar sind. Außerdem verwendet es *Backtracking*. Mit anderen Worten: Die Berechnung des Zieles $G$ mit Hilfe des PROLOG-Programms $\mathbf{P}$ ist die Generierung aller Lösungen von $G$ in $\mathbf{P}$ mittels der *left-most Strategie*. Die Suchregeln werden mit Hilfe eines Stacks von Zielen implementiert. Die Instanzen der Ziele im Stack sind die gerade untersuchten Zweige. Die Berechnung ist eine Sequenz von *push* und *pop* auf diesem Keller. Push kommt vor, wenn es gelingt, das selektierte Atom und das oberste Element des Kellers erfolgreich zu identifizieren. Dann wird der Resolvent auf dem Keller abgelegt. Es wird ein Element aus dem Keller genommen, wenn es keine Klausel gibt, deren Kopfelement mit dem ersten Atom des Ziels des obersten Kellerelements identifiziert werden kann. Dieses Ziel wird dann entfernt und die nächste Wahl der passenden Klausel für das oberste Element des Kellers untersucht. Es gibt nur eine Möglichkeit, von diesem Mechanismus abzuweichen, nämlich wenn das eingebaute Prädikat *cut* verwendet wird und damit das Backtracking unterbrochen wird.

### 2.2.4 CUT

*Cut* ist eine oft benutzte Kontrollmöglichkeit in PROLOG. Meistens ist es in Programmen mit "!" gekennzeichnet. Worin besteht die Wirkung von *cut* ? Ist *cut* ein weiteres Atom im Körper der Klausel, oder ist es nur eine nicht logische Vergrößerung des Programms, um einige Kontrollinformationen für das System zu übermitteln? Es kann als Atom, das sofort nach dem Aufruf *yes* zurückliefert, betrachtet werden.

Jedoch wird die deklarative Semantik durch *cut* nicht verändert. Wie funktioniert der *cut*-Mechanismus? Um dies erklären zu können, benutzen wir einige Begriffe von Lloyd [Llo87b]:

> Sei das Ziel, das den Anruf der Klausel mit *cut* verursacht, Elternziel (ein Atom aus dem Elternziel wird mit dem Kopfelement der Klausel unifiziert). Nach der Selektion von *cut* ist dieses sofort erfolgreich. Wenn das Backtracking später zu *cut* zurückkehrt, dann unterbricht das System die Durchsuchung des Teilbaums, dessen Wurzel das Elternziel ist. Im SLD-Baum bedeutet das, daß auch der Rest dieses Teilbaums abgeschnitten wird.

**Beispiel 7** Um diese Vorgehensweise besser zu veranschaulichen, betrachten wir das folgende Beispiel:

$A$ :- $B, C$
$\vdots$
$B$ :- $D, !, E$
$D$ :- $\ldots$
$\vdots$

Die Abbildung 2.6 zeigt den SLD-Baum für den Aufruf dieses Programms mit dem Ziel `A`.

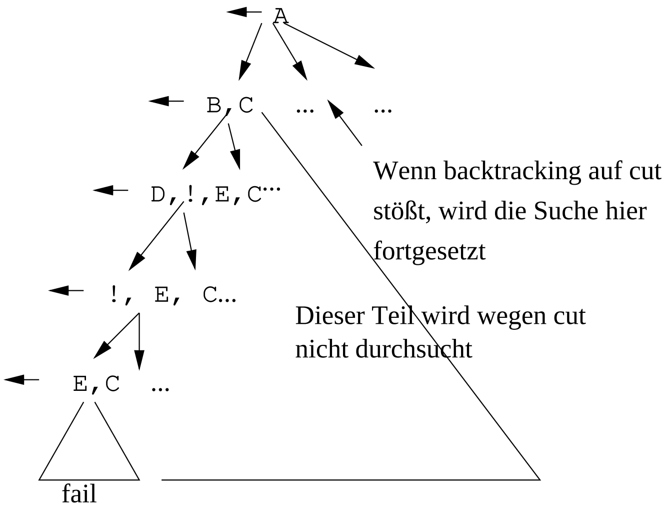



Abbildung 2.6: SLD-Baum für ein Programm mit *cut*

Der Effekt von *cut* ist der folgende: $A, B, C, D$ und $E$ sind Atome. Wenn wir der left-most Strategie folgen und nach A fragen, wird $B$ selektiert. Nehmen wir an, daß das als nächstes selektierte Atom $D$ erfolgreich ist und $E$ fehlschlägt. Dies führt das Backtracking zu *cut* zurück. Hier hört das System mit der Durchsuchung des Teils, dessen Wurzel $B, C$ ist, auf und fährt mit dem nächsten Ziel in :-$A$ fort.

Ein *cut* ist *sicher* (*safe*), wenn es keine Antwort im weggeschnittenen Teilbaum gibt (wenn es also keinen erfolgreichen Zweig gibt), ansonsten ist es *unsicher* (*unsafe*).

### 2.2.5 Logische Programmierung und PROLOG

Es existieren bereits PROLOG-Implementationen. Unter anderem IC-PROLOG ([CM79]), MU-PROLOG ([Nai85a], [Nai85b]), NU-PROLOG ([Nai86], [TZ86]) und die PROLOG-10-Familie, die DECsystem-10, C-PROLOG, POPLOG, RT11 und York Portable PROLOG (siehe [PBB$^+$78] [CM81]) enthält. Die heute meistverwendeten Systeme sind *Quintus* ([Qui87]) und *Sicstus* ([CWA$^+$]). Sie unterscheiden sich in der Syntax sowie in einigen kleinen Verfahrensweisen und Programmiermöglichkeiten. Die meisten Systeme folgen der *Edinburgh-Konvention.*

Hier möchten wir noch einige Unterschiede zwischen *logischer Programmierung* und *PROLOG* auflisten:

- In PROLOG bestimmt die Reihenfolge der Regeln (Klauseln), in welcher Reihenfolge die Lösungen gefunden werden. Daher müssen einige Kontrollmechanismen (zum Beispiel *cut*) zur Verfügung gestellt werden, während in der logischen Programmierung diese syntaktischen Beschränkungen nicht nötig sind.

- PROLOG's depth-first Ableitungsmechanismus hat das Problem, daß, wenn ein zu dem gegebenen Ziel gehörender Ableitungsbaum unendliche Zweige besitzt, die Berechnung nicht terminiert. PROLOG kann keine Lösung des Zieles finden, obwohl das Ziel eine endliche Ableitung hat.

- Die Anordnung der Atome in den Klauseln determiniert den Ableitungsbaum und die Menge notwendiger Suchvorgänge im Programm, um das Ziel beantworten zu können.

- Die Redundanz in einem PROLOG Programm bedeutet, daß das Ziel durch verschiedene Wege ableitbar ist, was eine Verlängerung der Laufzeit aufgrund des Backtracking zur Folge hat. Die Laufzeit ist in der logischen Programmierung irrelevant.

Natürlich gibt es mehrere Versuche, eine logische Programmiersprache mit anderen Programmierparadigmen, zum Beispiel mit funktionaler Programmierung, zu kombinieren. Es gibt zwei verschiedene Arten der Integration:

- Integration der logischen Programmierung mit schon existierenden funktionalen Sprachen, wie zum Beispiel Logusp ([RS82]), QLOG ([Kom82]), POPLOG ([MH84]) und APPLOG ([Coh86]).

- Entwicklung einer neuen Sprache, die es erlaubt, Funktionen und Relationen zu definieren, wie zum Beispiel SLOG ([Fri85]), EQLOG ([GM86]), LEAT ([BBLM86]), K-LEAF ([BGL$^+$87]) und BABEL ([MNRA92]).

In diesen oben erwähnten Sprachen können sowohl funktionale als auch logische Komponenten definiert werden. Es können aber auch externe Bibliotheken aus schon existierenden funktionalen oder prozeduralen Sprachen, wie *C* oder Fortran, verwendet werden. Diese Eigenschaft besitzen die zwei meistverwendeten Programmiersprachen Quintus und Sicstus PROLOG auch. Eine Beschreibung der möglichen Weiterentwicklung logischer Programmierung kann in Buch [AdBR94] gefunden werden. Die neuen Ideen für die Sprache PROLOG werden im Buch [Lev94] beschrieben.

Nachdem in diesem Kapitel wichtige Grundbegriffe der logischen Programmierung und PROLOG erläutert worden sind, folgt in den nächsten Kapiteln der wichtigste Teil der Arbeit: Die durch die Verwendung der *CPM*-Methode (*Category Partition Method*) und der *Slicing*-Technik verbesserten, von Shapiro entwickelten Methoden für *IDTS* (*Interactive Debugging, Testing and Slicing System*) werden eingeführt.

# Kapitel 3

# Shapiros Diagnose-Algorithmus zum Finden von Fehlern in PROLOG-Programmen

Unser *IDTS*-System kombiniert Shapiros algorithmische Debuggingmethoden mit funktionalem Testen (*CPM*) und mit der Methode des Programmslicing. Das Ziel besteht darin, ein effektives System für das Debugging und zum Testen von PROLOG-Programmen zu entwickeln. Das algorithmische Debugging von Shapiro kann eine fehlerhafte Klausel identifizieren, wenn das Programm und das Ziel vorgegeben sind. Der Nachteil dieses Debuggingverfahrens ist, daß der Benutzer eine große Anzahl von Fragen beantworten muß. Diese Fragen betreffen die Korrektheit der Zwischenergebnisse von Klauselaufrufen. Während des Debuggings des Programms muß der Benutzer mehrere Fragen beantworten. Häufig sind dies *ja/nein*-Fragen über die Korrektheit von Prädikatinstanzen. Manchmal müssen auch Variablenbelegungen für die Prädikatinstanzen angegeben werden.

Die Idee von *IDTS* war die folgende: Wenn das Programm bereits getestet wurde, dann können diese Testergebnisse direkt im Debugging verwendet werden, anstatt den Anwender zu fragen. Die im Kapitel 5 beschriebene Slicing-Methode berechnet den relevanten Teil des Ableitungsbaums und reduziert darüber hinaus die Mengen der Fragen.

Um unsere Erweiterungen beschreiben zu können, erklären wir zuerst näher, wie die von Shapiro vorgeschlagenen Debuggingverfahren arbeiten.

Wie wir schon erwähnt haben, liefert die Methoden von Shapiro eine fehlerhafte Klausel, wenn das Programm und das Ziel gegeben sind. Die Algorithmen durchlaufen den Ableitungsbaum des Programms in unterschiedlicher Weise und stellen dem Benutzer Fragen. Diese Fragen beziehen sich auf das vermutete Verhalten der jeweiligen Klauseln.

Die Methoden von Shapiro können drei verschiedene Formen von Fehlern erkennen:

- *Terminierung mit inkorrekter Ausgabe*: Das Programm hält, aber das Ergebnis ist falsch.
- *Terminierung ohne Ausgabe*: Das Programm beendet seine Arbeit ohne Ausgabe.
- *Nichtterminierung*: Das Programm läuft in eine Endlosschleife.

In diesem Kapitel verwenden wir die schon vorher eingeführten Definitionen für logische Programme:

- Unter einem *logischen Programm* **P** verstehen wir eine Menge von Klauseln $C$: $A \leftarrow B_1, \ldots, B_n$, wobei der Kopf von $C$ *(head (C))* $A$ ist und der Rumpf von $C$ (*body*($C$)) die Menge $\{B_1, \ldots, B_n\}$ ist.

- Eine Klausel ist genau dann *definit*, wenn alle $B_i$ atomar sind und $n \geq 0$.

- Wenn $\mathcal{I}$ eine Interpretation von **P** ist, $A'$ ein Atom ohne Variablen bezeichnet und $A \leftarrow B_1, \ldots, B_n$ eine beliebige Klausel im Programm **P** ist, dann können wir sagen, daß $A \leftarrow B_1, \ldots, B_n$, das Atom $A' \in \mathcal{I}$ genau dann *abdeckt (cover)*, wenn es eine *Substitution* $\theta$ gibt, so daß $A\theta = A'$ und für alle $i$ $(1 \leq i \leq n)$ $B_i\theta \in \mathcal{I}$ gilt.

- Eine beliebige Klausel $C \in$ **P** ist genau dann *korrekt* in $\mathcal{I}$, wenn alle Grundatome (Atome, die keine Variablen enthalten) von $C$ in $\mathcal{I}$ substituiert wurden. Sonst ist $C$ *inkorrekt* in $\mathcal{I}$.

- Sei $C$ eine beliebige Klausel in einem Programm **P**, wobei $C$ für die Eingabe $x$ terminiert und die dazugehörige Ausgabe $y$ ist. Das *top level trace* des Tripels $< C, x, y >$ ist eine geordnete Menge $\{< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, \ldots < C_n, x_n, y_n >\}$ für die gilt: Wenn $C$ mit $x$ als Eingabe aufgerufen wird, dann wird zuerst $C_1$ mit $x_1$ aufgerufen und gibt $y_1$ als Ausgabe zurück. Es folgt $C_2$ mit Eingabe $x_2$ und so weiter. Am Ende ruft $C$ die Klausel $C_n$ mit Eingabe $x_n$ auf und gibt $y_n$ zurück, so daß $C$ mit $y$ als Ausgabe endet.

Alle in diesem Kapitel vorkommenden Algorithmen veranschaulichen wir mit dem Beispiel *compute*, das das Komplement (*complement*) zweier Listen und die Summe der Listenelemente (*sigma*) berechnet. Das von uns verwendete Programm enthält einen Fehler: Die Zeile

```
delete(X, [Y|Z], [Y|U]): - delete(X, Z, U).
```

müßte korrekt lauten:

```
delete(X, [Y|Z], [X|U]): - delete(X, Z, U).
```

Der vollständige Programmtext ist der folgende:

**Beispiel 8** Das fehlerhafte *compute*-Programm

```
program(L1,L2,C,S)     :- compute(L1,L2,C,S), print(C), print(S).
compute(L1,L2,C,S)     :- sigma(L1,L2,S), complement(L1,L2,C).

complement(L,[],L).
complement(L,[H|T],U)  :- member(H,L), delete(H,L,L1),
                          complement(L1,T,U).

member(X,[X|T]).
member(X,[Y|T])        :- member(X,T).
```

```
delete(X,[X|T],T).
%delete(X,[Y|Z],[X|U]) :- delete(X,Z,U). die korrekte Klausel
delete(X,[Y|Z],[Y|U])  :- delete(X,Z,U).

sigma(L1,L2,S)         :- sum(L1,S1), sum(L2,S2), S is S1+S2.

sum([],0).
sum([X|T],S)           :- sum(T,S1), S is S1+X.
```

Wenn wir das oben gegebenen Programm mit der Anfrage:

```
program([1,2,3,4],[1,2,4],C,S)
```

aufrufen, bekommen wir für $C$ die Ausgabe [4] und für $S$ das Resultat 17, wobei das erste Ergebnis offensichtlich falsch ist, weil nicht [4] das Komplement der beiden Listen ist. sondern [3]. Den zum Programm und zum vorgegebenen Ziel gehörenden Ableitungsbaum zeigt Abbildung 3.1.

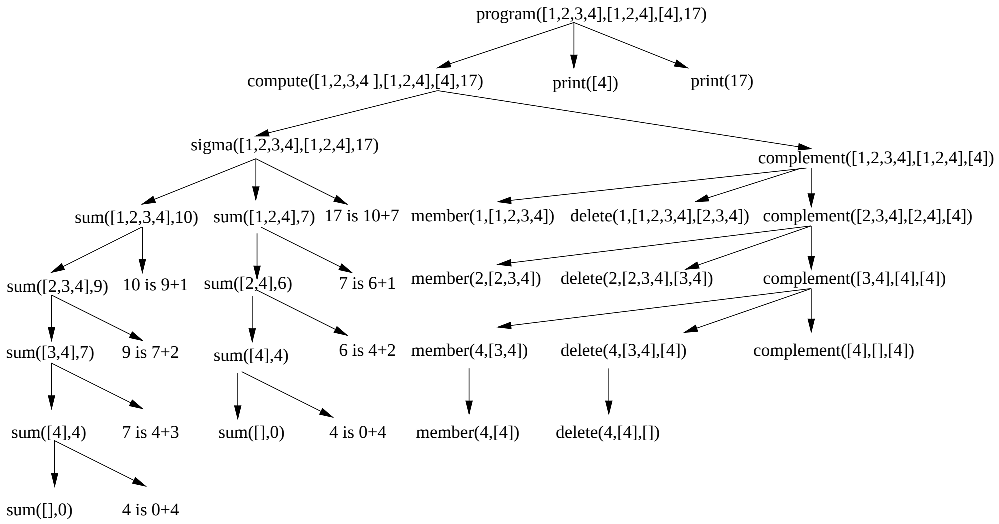



Abbildung 3.1: Der Ableitungsbaum für das fehlerhafte Programm *compute* und das Ziel `program([1,2,3,4],[1,2,4],C,S)`

Offensichtlich terminiert unser Beispiel-Programm mit einer inkorrekten Ausgabe. Im nächsten Abschnitt werden wir zeigen, wie man unter Verwendung verschiedener Traversierungsstrategien den Fehler finden kann.

## 3.1 Terminierung mit inkorrekter Ausgabe

Zunächst nehmen wir an, daß **P** *frei von Seiteneffekten* (*side effects*) ist. Sei $C$ eine beliebige Klausel in **P**, die mit $x$ als Eingabe aufgerufen wird und mit der Ausgabe $y$ terminiert, wobei $< C, x, y > \notin \mathcal{I}$. Das heißt, daß es wenigstens eine falsche Klausel in **P** gibt. Um diese *inkorrekte Klausel*, die den Fehler verursacht, zu finden, wird der Ableitungsbaum auf verschiedene

Weisen durchlaufen. Im Fall einer Terminierung mit inkorrekter Ausgabe hat Shapiro drei verschiedene Methoden entwickelt:

- *Bottom-Up*
- *Top-Down*
- *Divide-and-Query*

In allen drei Fällen überprüft der Diagnose-Algorithmus die Knoten des Ableitungsbaums und stellt dem Benutzer *ja/nein*-Fragen. In der *Bottom-Up*-Methode wird der Ableitungsbaum in *Postorder*-Weise traversiert. Hier wird gefragt, ob die berechneten Werte korrekt oder inkorrekt sind. Wenn das Ergebnis in einem Knoten inkorrekt ist, aber alle Nachkommen dieses Knotens das korrekte Ergebnis liefern, dann identifiziert dieser Algorithmus die Klausel, deren Kopfelement in diesem Knoten vorkommt, als fehlerhaft.

Der zweite Durchlaufmechanismus überprüft die Knoten im *Top-Down*-Durchgang. Wenn die in einem Knoten berechnete Antwort korrekt ist, dann braucht das Verfahren nicht den Unterbaum zu diesem Knoten zu untersuchen, sondern kann mit dem nächsten, rechts von diesem Atom stehenden Knoten fortfahren.

Die leistungsfähigste Methode ist die *Divide-and-Query*-Strategie. Diese Methode berechnet Gewichte für alle Knoten im Baum und zerlegt diesen auf der Basis der berechneten Gewichte in zwei ungefähr gleich gewichtete Teile. Dann stellt sie eine Frage für den Zerlegungsknoten. Wenn der Knoten eine inkorrekte Bewertung liefert, dann traversiert der Algorithmus rekursiv den Unterbaum, dessen Wurzel dieser Knoten ist. Im anderen Fall, wenn die Antwort bei diesem Knoten also korrekt ist, wird der Teilbaum entfernt und ein neuer Mittelknoten berechnet.

Schon aus dieser Beschreibung wird klar, wie wichtig eine Visualisierung der Bäume ist. Diese kann die Umgebung der in einem Knoten gestellten Frage zeigen und damit helfen, diese zu beantworten. Auf die Details, die zu *IDTS* entwickelte graphische Benutzerschnittstelle (*GIDTS*), wird in Kapitel 6.4 eingegangen.

### 3.1.1 Bottom-Up-Durchlaufen des Ableitungsbaums

Bei der *Bottom-Up*-Methode wird das Traversieren der Baumknoten in *Postorder*-Art durchgeführt (siehe Algorithmus 3). Das heißt, daß beim Durchlaufen des Baums bei allen Knoten $< C, x, y >$ eine Zugehörigkeitsfrage gestellt wird, wobei $C$ eine Klausel, $x$ die Eingabe, und $y$ die gelieferte Ausgabe sind. Nehmen wir an, daß die erste *false*-Antwort bei dem Knoten $< C, x, y >$ zurückgegeben wird. Die direkten Nachkommen von $< C, x, y >$ seien $< C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >$. Weil die Postorder-Strategie benutzt wird, gilt: $\forall j \quad (1 \leq j \leq n) \quad < C_j, x_j, y_j > \in \mathcal{I}$. Daraus folgt, daß die Klausel $C \leftarrow C_1, \ldots, C_n$, die von $< C, x, y >$ repräsentiert wird, nicht in $\mathcal{I}$ ist.

Der *Bottom-Up*-Algorithmus hält bei dem Knoten $< C, x, y >$ und gibt eine Instanz der Klausel $< C, x, y > \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >$ aus. Die fehlerhafte Klausel kann immer identifiziert werden, wenn die Antworten zu den Zugehörigkeitsfragen korrekt sind.

**Algorithmus 3** Shapiros *Bottom-Up*-Methode

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \bot$ und inkorrekt in $\mathcal{I}$
**Ausgabe:** eine Instanz der fehlerhaften Klausel $C_{buggy}$,
ein Wahrheitswert für $found$,
wobei $found$ =*true*, wenn $C_{buggy}$ gefunden wurde, sonst $found$ =*false*

```
procedure bottom_up(< C, x, y >, C_buggy, found)
begin
  let {< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, ... < C_n, x_n, y_n >} be the top level trace of < C, x, y >;
  C_buggy:=ε; i:=1; found:=false
  while (i ≤ n) and (not found) do
    bottom_up(< C_i, x_i, y_i >, C_buggy, found);
    i := i + 1
  endwhile
  if not found then
    answer := Query(< C, x, y >)
    if answer = false then
      found := true;
      C_buggy := (< C, x, y >←< C_1, x_1, y_1 >, ..., < C_n, x_n, y_n >)
    endif
end
```

Die Funktion *Query* bekommt als Eingabe $< C, x, y >$ und liefert als Ausgabe einen logischen Wert zurück. Der Benutzer wird gefragt, ob das gerade untersuchte Atom mit der gegebenen Eingabe die korrekte Ausgabe liefert oder nicht.

**Algorithmus 4** Das Fragestellen an den Anwender

**Eingabe:** ein Atom $< C, x, y >$
**Ausgabe:** oracle-answer
true, wenn $< C, x, y > \in \mathcal{I}$
false, wenn $< C, x, y > \notin \mathcal{I}$

```
function Query(< C, x, y >):oracle-answer
  begin
    oracle-answer := "Is the goal < C, x, y > correct? (y/n)";
    return(oracle-answer)
  end
```

Wenn man diese Funktion *Query* betrachtet, kann man noch nicht erkennen, warum es sinnvoll ist, eine eigene Funktion zu definieren. Später wird diese Funktion so erweitert, daß wir, anstatt den Benutzer zu fragen, die Antwort in einer Testdatenbank zu finden versuchen (siehe Kapitel 4).
Der oben angegebene Algorithmus stellt die folgenden Fragen für das Beispiel 8, wenn das Programm mit dem Ziel `compute([1,2,3,4],[1,2,4],X,Y)` aufgerufen wird.

**Beispiel 9** Die bei der *Bottom-Up*-Methode gestellten Fragen und die Ausgabe der falschen Klausel:

```
Is the goal [sum([],0)] correct? (y/n) y
Is the goal [sum([4],4)] correct? (y/n) y
Is the goal [sum([3,4],7)] correct? (y/n) y
Is the goal [sum([2,3,4],9)] correct? (y/n) y
Is the goal [sum([1,2,3,4],10)] correct? (y/n) y
Is the goal [sum([2,4],6)] correct? (y/n) y
Is the goal [sum([1,2,4],7)] correct? (y/n) y
Is the goal [sigma([1,2,3,4],[1,2,4],17)] correct? (y/n) y
Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n) y
Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n) y
Is the goal [member( 2 , [2,3,4] )] correct? (y/n) y
Is the goal [delete( 2 , [2,3,4] , [3,4] )] correct? (y/n) y
Is the goal [member( 4 , [4] )] correct? (y/n) y
Is the goal [member( 4 , [3,4] )] correct? (y/n) y
Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n

The false clause:
delete(4 , [3,4] , [4]) :- delete(4 , [4] , []).
```

Der Ableitungsbaum des Programms *compute* in Abbildung 3.2 zeigt uns, wie PROLOG das Ergebnis berechnet und welche Fragen von der *Bottom-Up*-Methode Shapiros nicht gestellt werden.

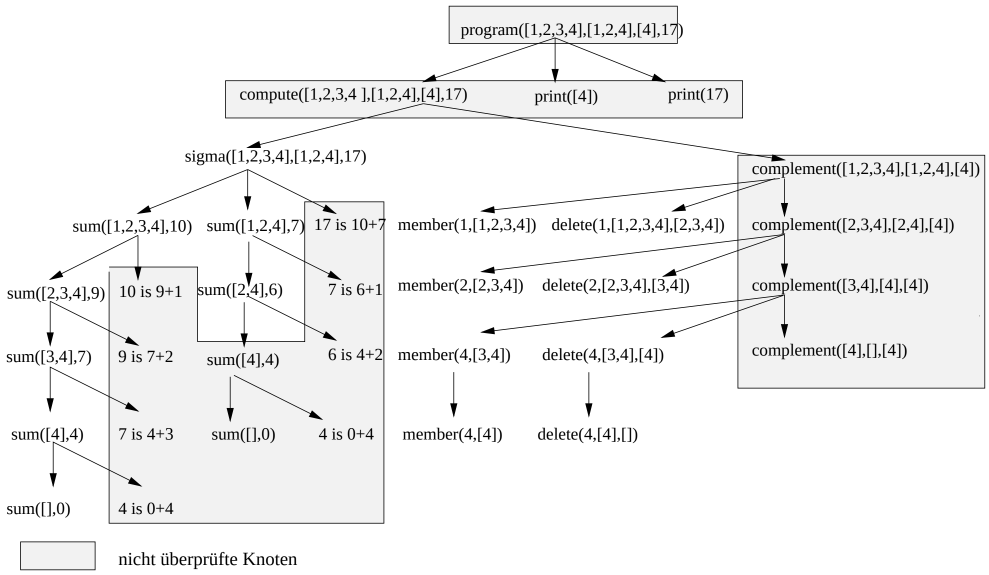



Abbildung 3.2: Die von Shapiros *Bottom-Up*-Methode gestellten und nicht gestellten Fragen für das *compute*-Beispiel

### 3.1.2 Top-Down-Durchlaufen des Ableitungsbaums

Eine mögliche Verbesserung des Algorithmus besteht darin, wenn der Ableitungsbaum *Top-Down* zu durchlaufen. Dies funktioniert wie folgt: Bei allen Knoten, angefangen mit der Wurzel des Ableitungsbaum wird versucht, den falschen Kindknoten zu finden. Wenn es bei keinem gelingt, dann enthält die Wurzel des Teilbaums die Instanz der falschen Klausel. Wenn alle Kindknoten eines solchen Knotens korrekt sind, hat man die Instanz der falschen Klausel gefunden. Der Algorithmus 5 liefert die falsche Klausel zurück.

**Algorithmus 5** Shapiros *Top-Down*-Methode

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$
**Ausgabe:** eine Instanz der fehlerhaften Klausel $C_{buggy}$,
ein Wahrheitswert $found$,
wobei $found$ =*true*, wenn $C_{buggy}$ gefunden wurde sonst $found$ =*false*
**procedure** $top\_down(<C,x,y>, C_{buggy}, found)$
**begin**
  let $\{<C_1,x_1,y_1>, \ldots, <C_n,x_n,y_n>\}$ be the top level trace of $<C,x,y>$;
  $C_{buggy}$:=$\varepsilon$; $i$:=1; found:=false
  **while** $(i \leq n)$ **and** (not found) **do**
    answer := $Query(C_i, x_i, y_i)$
    **if** answer = false **then** $top\_down(<C_i,x_i,y_i>, C_{buggy}, found)$
    **else** $i := i+1$
    **endif**
  **endwhile**
  **if** not found **then**
    $C_{buggy} := (<C,x,y>: - <C_1,x_1,y_1>, \ldots, <C_n,x_n,y_n>)$;
    found := true
  **endif**
**end**

Der Algorithmus stellt zuerst eine Frage zur Korrektheit eines Atoms. Wenn die Antwort *no* ist, dann liegt der Fehler in dem Unterbaum, dessen Wurzel dieser Knoten ist. Das bedeutet, daß das Verfahren mit dem am weitesten links stehenden Nachkommen fortgesetzt wird. Wenn aber die Antwort *yes* war, dann ist es unnötig, diesen Teilbaum zu überprüfen. Also kann man mit dem rechts von diesem Knoten stehenden Atom fortfahren.

Das nächste Beispiel illustriert die von dieser Methode gestellten Fragen, und die Abbildung 3.3 zeigt den zum Ziel `program([1,2,3,4],[1,2,4],X,Y)` gehörenden Ableitungsbaum. Während das *Bottom-Up*-Verfahren 16 Fragen brauchte, um den Fehler zu finden, kann der *Top-Down*-Algorithmus mit Hilfe von 12 Fragen den Fehler identifizieren.

**Beispiel 10** Die bei der *Top-Down*-Methode gestellten Fragen und die Ausgabe der falschen Klausel:

```
Is the goal [compute([1,2,3,4],[1,2,4],[4],17)] correct? (y/n) n
Is the goal [sigma([1,2,3,4],[1,2,4],17)] correct? (y/n) y
```

```
Is the goal [complement([1,2,3,4],[1,2,4],[4]]) correct? (y/n) n
Is the goal [member(1,[1,2,3,4])] correct? (y/n) y
Is the goal [delete(1,[1,2,3,4],[2,3,4])] correct? (y/n) y
Is the goal [complement([2,3,4],[2,4],[4])] correct? (y/n) n
Is the goal [member(2,[2,3,4])] correct? (y/n) y
Is the goal [delete(2,[2,3,4],[3,4])] correct? (y/n) y
Is the goal [complement([3,4],[4],[4])] correct? (y/n) n
Is the goal [member(4,[3,4])] correct? (y/n)? y
Is the goal [delete(4,[3,4],[4])] correct? (y/n)? n
Is the goal [delete(4,[4],[])] correct (y/n) y

The false clause:
delete(4 , [3,4] , [4]) :- delete(4 , [4] , []).
```

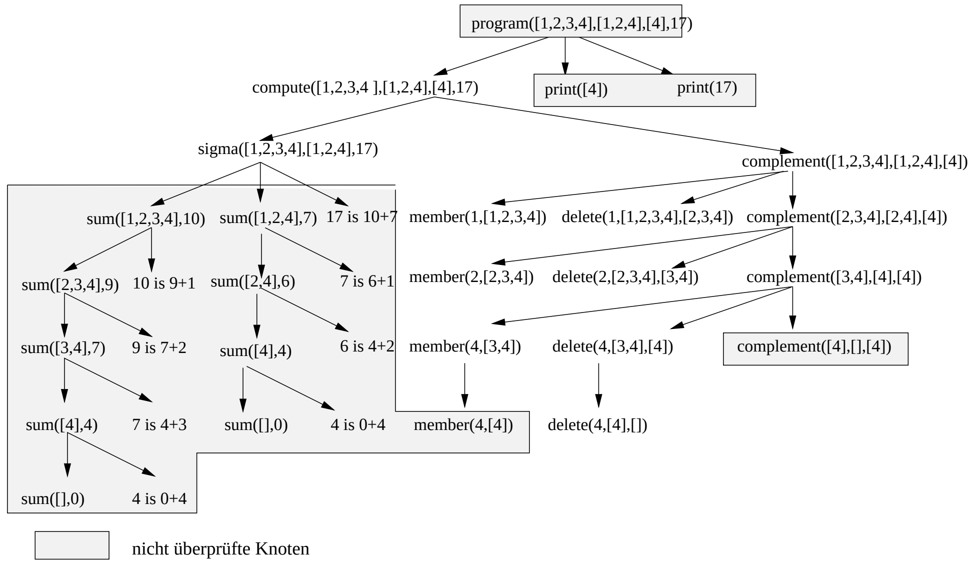



Abbildung 3.3: Die von Shapiros *Top-Down*-Methode gestellten Fragen für das *compute*-Beispiel

### 3.1.3 Die Divide-and-Query-Methode

Shapiro selbst hat eine Methode vorgeschlagen, bei der die Anzahl der Fragen optimiert wird, die *Divide-and-Query*-Methode. Seine Idee war die folgende: Ein mittlerer Knoten $< C_j, x_j, y_j >$ wird gesucht, und dieser Knoten trennt den Ableitungsbaum in zwei ungefähr gleich große Komponenten auf. Wenn $< C_j, x_j, y_j > \in \mathcal{I}$ ist, dann lassen wir den Teilbaum, dessen Wurzel dieser Knoten ist, weg und fahren rekursiv auf den verbleibenden Knoten fort, ansonsten wenden wir die Methode rekursiv im aktuellen Teilbaum, dessen Wurzel $< C_j, x_j, y_j >$ ist, an. Zuerst muß beschrieben werden, was man unter dem Gewicht eines Knotens und der Zerteilung des Ableitungsbaums versteht und wie diese berechnet werden können. Fangen wir damit an, daß wir das Gewicht eines Knotens definieren. Diese Definition wird dann benutzt, um den Mittelpunkt eines Ableitungsbaums zu berechnen, wobei $\mathcal{I}'$ eine Menge von Prädikaten ist, die von dem Algorithmus als korrekt akzeptiert werden.

**Definition 45** Sei $\mathcal{I}' \leq \mathcal{I}$ und der Ableitungsbaum besitze die Wurzel $< C, x, y >$. Das Gewicht $g^{\mathcal{I}'}$ des Knotens $< C', x', y' >$ kann wie folgt definiert werden:

$$g^{\mathcal{I}'}(< C', x', y' >) = \begin{cases} 0 & \text{falls } < C', x', y' > \in \mathcal{I}' \\ 1 & \text{falls } < C', x', y' > \text{ ein Blatt} \\ 1 + \sum_{< C'', x'', y'' >} g^{\mathcal{I}'}(< C'', x'', y'' >) & \text{sonst} \end{cases}$$

Hierbei sind $< C'', x'', y'' >$ die Nachfolgeknoten von $< C', x', y' >$.

Der nächste Algorithmus zeigt uns, wie das Gewicht für den gesamten Baum festgestellt werden kann. Das Gewicht des Baums ist das Gewicht der Wurzel. Der Algorithmus arbeitet wie folgt: Das Gewicht wird von unten nach oben berechnet. Es bedeutet, daß zuerst für die Blätter des Baums das Gewicht zugeordnet wird. Das Gewicht eines Elternknotens wird dann als die Summe der Gewichte der Kindknoten plus eins erzeugt. Diese Vorgehensweise setzt voraus, daß die *Postorder*-Strategie beim Überprüfen der Knoten im Ableitungsbaum verwendet wird. Das Verfahren sucht nach dem am weitesten links unten stehenden Knoten im Baum, so daß man mit Hilfe des *top level trace* eine Ebene tiefer geht. Wenn das *top level trace* leer ist (keine Ableitung mehr möglich), dann haben wir ein Blatt des Baums erreicht, und diesem Knoten wird das Gewicht 1 zugeordnet, wenn dieser Knoten kein Systemprädikat enthält. Wenn er ein Systemprädikat enthält, wird er mit 0 gewichtet. Wenn alle Kindknoten ein Gewicht erhalten haben ($i = n$), dann wird die Summe der Gewichte der Kindknoten um eins inkrementiert und dem Knoten zugeordnet.

**Algorithmus 6** Berechnung der Gewichte im Baum

**Eingabe:** $< C, x, y >$, $\mathcal{I}' \subseteq \mathcal{I}$
**Ausgabe:** das Gewicht der Wurzel *weight*

```
procedure compute_weight(< C, x, y >, I', weight)
begin
  let {< C_1, x_1, y_1 >, ..., < C_n, x_n, y_n >} be the top level trace of < C, x, y >
  if C ∉ I' then
    i := 1
    while i ≤ n do
      compute_weight(< C_i, x_i, y_i >, I', sumweight);
      i:=i + 1;
      weight := weight+sumweight
    endwhile
    if n = 0 then
      if (C is a system predicate) then weight := 0
      else weight := 1
      endif
    else
    weight := weight +1
    endif
  else weight := 0
  endif
end
```

**Beispiel 11** Die Abbildung 3.4 zeigt den Ableitungsbaum des schon oft verwendeten Beispiels *compute*. Für alle Knoten ist zusätzlich das Gewicht angegeben.

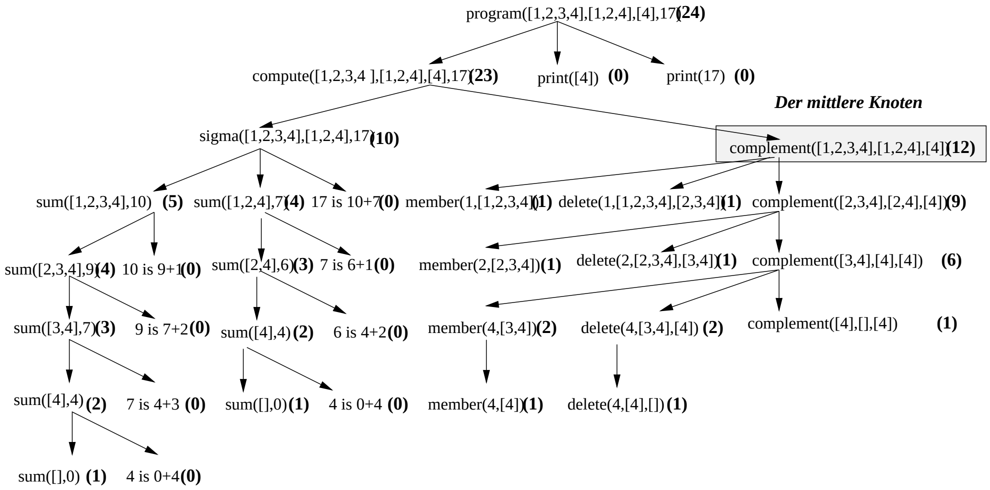



Abbildung 3.4: Der gewichtete Ableitungsbaum des *compute*-Beispiels

**Definition 46** Sei **T** ein Ableitungsbaum, dessen Gewicht $g^{\mathcal{I}'} = w$ ist. Der *mittlere Knoten mp* ist der am weitesten links liegende schwerste Knoten im Baum, wobei $g(mp)^{\mathcal{I}'} \leq \lceil w/2 \rceil$ gilt.

Die Berechnung von $mp$ kann, wenn das zum Ableitungsbaum gehörende Gewicht $w$ bekannt ist, linear in $w$ durchgeführt werden (siehe [Sha83] S. 45). Als nächstes formulieren wir einen Algorithmus, der beschreibt, wie dieser Knoten berechnet werden kann.

**Algorithmus 7** Berechnung des mittleren Knotens und seines Gewichts

**Eingabe:** Eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ $y$ zurückgibt, wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
das Gesamtgewicht *tree_weight* des Ableitungsbaums zu $C$

**Ausgabe:** Der mittlere Knoten $< C_j, x_j, y_j >$,
das Gewicht des mittleren Knotens *middle_point_weight*,
das Gesamtgewicht *weight* des Ableitungsbaums
mit dem mittleren Knoten als Wurzel

```
procedure middle_point(< C, x, y >, < Cj, xj, yj >, middle_point_weight, weight,tree_weight)
begin
  let {< C1, x1, y1 >, ..., < Cn, xn, yn >} be the top level trace of < C, x, y >;
  < Cj, xj, yj >:= ε; found:=false; i := 1; node_weight:=0;
  local_max_weight:= 0; position := ε
  if C ∉ I' then
    while (i ≤ n) and (not found) do
      middle_point(< Ci, xi, yi >, < Cj, xj, yj >, middle_point_weight, weight,tree_weight)
      if local_max_weight < weight then
        local_max_weight := weight;
        position :=< Ci, xi, yi >
      endif
      node_weight := node_weight + weight;
      i := i + 1
    endwhile
    if not found then
      if n = 0 and (C is not system predicate) then weight:= 1
      else
        if n = 0 then weight:= 1
        else node_weight := node_weight +1
            if node_weight > (tree_weight +1)/2 then
              < Cj, xj, yj >:= position;
              middle_point_weight := local_max_weight;
              found := true
            else weight := node_weight
            endif
        endif
      endif
    endif
  else weight := 0
  endif
end
```

Dieser Algorithmus sucht nach dem mittleren Knoten ($< C_j, x_j, y_j >$) und dessen Gewicht (*middle_point_weight*). Er überprüft die Knoten und berechnet die Knotengewichte. Diese Berechnung wird so lange fortgesetzt, bis sie in den unteren Teil des Ableitungsbaums gelangt: (*node_weight* $>$ (*weight* $+ 1)/2$). Mit anderen Worten, diese Knoten haben ein Gewicht, das kleiner als die Hälfte des Gesamtgewichtes des Baums (*weight*) ist. Das Verfahren findet den Knoten mit dem größten Gewicht von all diesen Knoten mit kleinem Gewicht. Wenn dieser Knoten gefunden wurde, dann wird die boolesche Variable *found* auf *true* gesetzt, und damit wird verhindert, daß der Algorithmus weitere Knoten überprüft.

**Beispiel 12** Betrachten wir die vorher erläuterte Abbildung 3.4. In dem ersten Schritt der *Divide-and-Query*-Methode ist das Gesamtgewicht des Baums 24. Also verwenden wir den Knoten `[complement( [1,2,3,4] , [1,2,4] , [4] )]` als mittleren Knoten, weil dessen Gewicht 12 ist, damit ist er der schwerste Knoten von den Knoten, deren Gewicht nicht größer als die Hälfte des Baumgewichtes ist. Der nächstschwerere Knoten hat schon das Gewicht 23. In der Abbildung 3.4 ist der mittlere Knoten grau gekennzeichnet.

Jetzt sind wir in der Lage, den Hauptalgorithmus zur *Divide-and-Query*-Methode zu formulieren.

---

**Algorithmus 8** Shapiros *Divide-and-Query*-Methode

---

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
$\mathcal{I}' \subset \mathcal{I}$, enthält am Anfang die Systemprädikate
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel
**procedure** $divide\_and\_query(< C, x, y >, C_{buggy}, \mathcal{I}')$
**begin**
  $compute\_weight(< C, x, y >, \mathcal{I}', weight)$
  $find\_error(< C, x, y >, \mathcal{I}', weight, C_{buggy})$
**end**

---

Der Algorithmus 8 ist der Hauptalgorithmus der *Divide-and-Query*-Methode und ruft zuerst die Prozedur *compute_weight* auf, die das Gewicht des ursprünglichen Ableitungsbaums *weight* liefert. Damit arbeitet dann das Verfahren *find_error* (siehe Algorithmus 9) weiter.

**Algorithmus 9** Suche nach dem Fehler im gewichteten Ableitungsbaum

**Eingabe:** eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$ ,
$\mathcal{I}' \subset \mathcal{I}$, wobei $\mathcal{I}'$ leer sein kann,
das Gewicht des zu $C$ gehörenden Ableitungsbaums *weight*

**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel

**procedure** $find\_error(< C, x, y >, \mathcal{I}'$, weight, $C_{buggy})$
**begin**
  middle_weight = 0; $C_{middle}{:=}\varepsilon$
  **while** weight $\neq 1$ **do**
    $middle\_point(< C, x, y >, C_{middle}$, middle_weight, weight);
    answer := Query$(< C, x, y >)$
    **if** answer = yes **then**
      $find\_error(< C, x, y >, \mathcal{I}' \cup C_{middle}$, weight-middle_weight,$C_{buggy})$
    **else**
      $find\_error(< C_{middle} >, \mathcal{I}'$, middle_weight,$C_{buggy})$
    **endif**
  **endwhile**
  $C_{buggy} := C_{middle} \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >,$
**end**

Der Algorithmus 9 versucht rekursiv, die falsche Klausel zu finden. Dabei benutzt er das schon vorher für den ganzen Baum berechnete Gewicht *weight*. Mit Hilfe dieses Werts wird das Gewicht des mittleren Knotens und das der übrigen Knoten bestimmt. Für den mittleren Knoten stellt der Algorithmus dem Anwender eine Frage. Wenn die Antwort *yes* lautet, dann liegt der Fehler nicht im Teilbaum dieses Knotens, deswegen wird die Prozedur rekursiv mit der gleichen Wurzel, aber mit der Differenz des Original-Gewichts und des mittleren Gewichts als Gewicht *weight* aufgerufen. Dabei wird der mittlere Knoten mit allen Unterknoten zu $\mathcal{I}'$ hinzugefügt. $\mathcal{I}'$ enthält die Menge der Klauselinstanzen, von denen angenommen wird, daß sie korrekt sind. Andernfalls, wenn die Funktion *Query* das Ergebnis *no* liefert, wird wieder rekursiv fortgefahren, aber jetzt bleibt $\mathcal{I}'$ unverändert. Es wird der am mittleren Knoten hängende Teilbaum betrachtet. Der Beweis der Korrektheit dieses Algorithmus kann bei Shapiro (siehe [Sha83] S. 45f) nachgelesen werden.

Um den Vergleich zwischen den verschiedenen Methoden weiterführen zu können, betrachten wir wieder das Beispiel 8 und nehmen das gleiche Ziel:

```
program([1,2,3,4],[1,2,4],C,S).
```

Das Gewicht des Ableitungsbaums ist 24. Verwendet man die *Bottom-Up*-Methode, so sind 16 Fragen zu beantworten. Im Fall *Top-Down* werden 12 Fragen gestellt. Die *Divide-and-Query*-Methode braucht nur 5 Fragen, um die falsche Klauselinstanz zu finden. Wir sehen, daß die Anzahl der Fragen erheblich reduziert wurde. Dieses Ergebnis bestätigt auch die grafische Abbildung des Ableitungsbaums (siehe Abbildung 3.5).

**Beispiel 13** Die bei der *Divide-and-Query*-Methode gestellten Fragen und die Ausgabe der falschen Klausel:

```
Is the goal [complement( [1,2,3,4] , [1,2,4] , [4] )] correct? (y/n) n
Is the goal [complement( [3,4] , [4] , [4] )] correct? (y/n) n
Is the goal [member( 4 , [3,4] )] correct? (y/n) y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y

The false clause:
delete(4 , [3,4] , [4]) :- delete(4 , [4] , [])
```

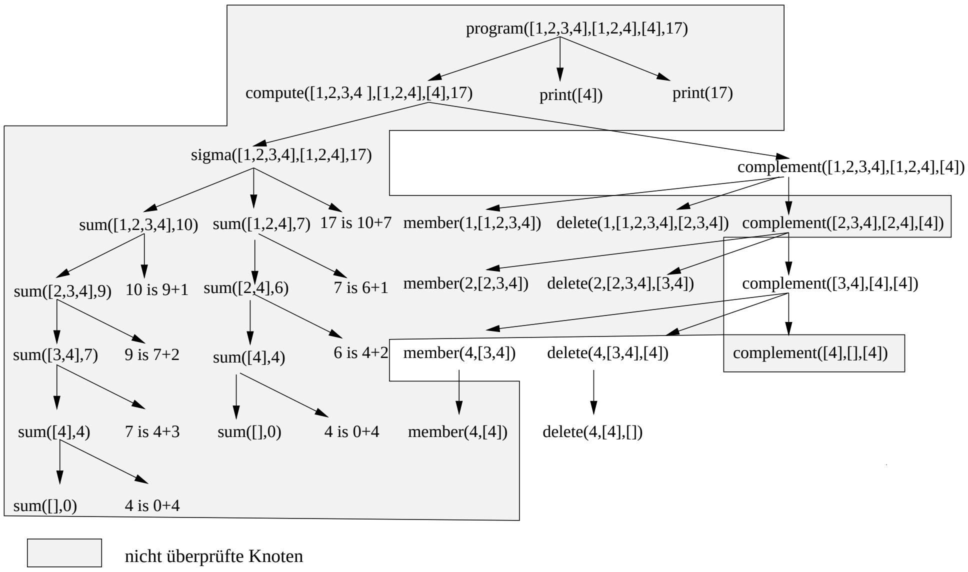



Abbildung 3.5: Die von Shapiros *Divide-and-Query*-Methode gestellten Fragen für das *compute*-Beispiel

### 3.1.4 Bewertung der Algorithmen

Um die in diesem Kapitel vorkommenden Algorithmen zu bewerten, kann man entweder die *Berechnungskomplexität* oder die *Anzahl der gestellten Fragen* überprüfen.

Ersteres bedeutet, zu berechnen, wieviele maschinelle Mittel der Algorithmus braucht. Um diese festzustellen, können drei verschiedene Maße verwendet werden:

- *Länge*: Die Anzahl der Klauselaufrufe, die bei der Berechnung verwendet werden, bestimmt sich aus der Knotenanzahl im Baum.
- *Tiefe* : Die maximale Tiefe der Klauselaufrufe (stack depth) entspricht der Tiefe des Ableitungsbaumes.
- *Größe*: Die Anzahl der Symbole in der textuellen Darstellung.

Formal kann man die Komplexität mit Hilfe der folgenden Definitionen beschreiben, wobei die SLD-Ableitung benutzt wird. (siehe Kapitel 2.2.1).

**Definition 47** Die *Länge einer Ableitung* ist die Anzahl der Knoten ihres Ableitungsbaums. Die *Tiefe einer Ableitung* ist die Tiefe ihres Ableitungsbaums. Die *Zielgröße einer Ableitung* ist die maximale Zeichenanzahl in einem Knoten ihres Ableitungsbaums.

Bei der zweiten und meistens verwendeten Bewertungsmöglichkeit fragt man nach der Anzahl und dem Typ der dem Benutzer gestellten Fragen. Unser Ziel ist es, die Anzahl der gestellten Fragen zu minimieren. Die maximale Anzahl der Fragen ist durch die Länge der Berechnung beschränkt, und die untere Schranke kann mit dem folgenden Satz beschrieben werden:

**Satz 8** Sei $DA$ ein Diagnosealgorithmus, mit dessen Hilfe man den Fehler im Fall inkorrekter Klauseln identifizieren kann. Außerdem sei $\mathbf{P}$ ein Programm, $\mathcal{I}$ eine Interpretation und $< C, x, y >$ ein Tripel, dessen Ableitungsbaum eine Länge kleiner oder gleich einer Zahl $n$ besitzt. Für einen solchen $DA$, der auf $< C, x, y >$ arbeitet, ist die Fragenkomplexität wenigstens $log_2 n$.

Den Beweis kann man bei Shapiro ([Sha83], S. 44) nachlesen.

Der *Single-stepping*-Algorithmus ist eigentlich eine *Bottom-Up*-Methode, Er findet den Fehler, indem er den Ableitungsbaum *postorder* traversiert. Nehmen wir an, daß der Algorithmus schon für jeden Nachkommen eines Knotens eine Frage gestellt hat, und diese wurden als gültig befunden. Dann wird eine Frage über den Knoten selbst gestellt. Wenn dieser Knoten auch korrekt ist, dann wird der Algorithmus das Postorder-Durchlaufen des Ableitungsbaums fortführen. Sein Nachteil ist, daß die Komplexität der Fragen im schlechtesten Fall gleich der Anzahl der Knoten im Ableitungsbaum ist.

Der *Divide-and-Query*-Algorithmus ist insofern eine Verbesserung, als die Fragenkomplexität abgesehen von einem konstanten Faktor (Verzweigungsfaktor $b$) optimal ist. Die Idee bei diesem Algorithmus ist die folgende: Er sucht nach dem mittleren Knoten im Baum. Zu diesem Knoten wird dann eine Frage gestellt. Wenn die Antwort die Korrektheit dieses Knotens bestätigt, dann wird ein neuer Mittelpunkt gesucht, aber ohne diesen Teilbaum erneut zu berücksichtigen. Wenn dieser Knoten nicht gültig ist, dann wird der Algorithmus rekursiv auf den Teilbaum, dessen Wurzel der gerade gefragte Knoten ist, aufgerufen. Shapiro hat gezeigt, daß diese Berechnung logarithmische Komplexität hat:

**Satz 9** Seien $\mathbf{P}$ ein Programm und $\mathcal{I}$ eine Interpretation. Es kann gezeigt werden (siehe [Sha83] S. 45f), daß folgendes gilt: Wenn $C \in \mathbf{P}$ eine Berechnung mit der Eingabe $x$ ist, wobei die Länge $n$ und die Tiefe $d$ ist, und der zurückgegebene Wert $y \neq \perp$ inkorrekt ist, diese auf $< C, x, y >$ und $\mathcal{I}' = \{\}$ angewendete Methode die Länge $cn$ für eine beliebige Konstante $c > 0$, Tiefe $d + 1$ und braucht höchstens $log_2 n$ Fragen, um das Tripel $< C_j, x_j, y_j > \notin \mathcal{I}$ zu finden.

Meistens ist es nicht möglich, den Baum genau in der Mitte zu zerlegen. Oft gibt es nur einen Knoten, dessen Gewicht dem mittleren Gewicht am nächsten kommt. Dies relativiert die Leistung des *Divide-and-Query*-Algorithmus. Wenn der Baum $n$ Knoten hat und der Verzweigungsfaktor $b$ ist, dann ist die Fragekomplexität im schlechtesten Fall $b{\cdot}\log n$.

Zusammenfassend läßt sich sagen, daß diese Methode sicherstellt, daß die Länge und die Tiefe der Berechnung linear ist. Aber das kann man nicht von der Laufzeit des Algorithmus behaupten, weil sehr viele Backtrackingschritte benötigt werden, um die neuen Gewichte des Ableitungsbaums zu berechnen.

Der *Top-Down-Algorithmus* sucht den Ableitungsbaum ab wie folgt: Zuerst wird eine Frage zur Wurzel gestellt, die vermutlich falsch ist. Wenn die bei allen Vorfahren gestellten Fragen mit *yes* beantwortet werden, dann wurde die falsche Klausel gefunden. Sonst geht der Algorithmus mit dem am weitesten links stehenden Vorfahren rekursiv weiter, der inkorrekt ist. Wenn der Baum den Verzweigungsfaktor $b$ hat und die Tiefe des Baums $h$ ist, dann ist die Fragekomplexität im schlimmsten Fall $bh$. Also ist die Anzahl der gestellten Fragen des *Top-Down*-Algorithmus im schlimmsten Fall nicht zur Tiefe des Ableitungsbaums linear, sondern zu dessen Länge.

Es gibt ein paar interessante Fälle, für die man die Fragekomplexität der letzten zwei Algorithmen vergleichen kann:

- *Top-Down kann eindeutig schlechter sein als Divide-and-Query*: Nehmen wir an, daß der Baum nur eine Liste ist und der Fehler am Ende der Liste ist. Der *Top-Down*-Fall muß bei allen Knoten Fragen stellen, um den Fehler zu finden, aber die andere Methode hat nur den Logarithmus dieser Anzahl an Fragen.

- *Top-Down kann eindeutig besser sein als Divide-and-Query*: Nehmen wir an, daß unser Baum in zwei Teilbäume zerlegt werden kann, wobei der rechte viel größer als der linke ist und der Fehler im linken Teilbaum liegt. Die *Divide-and-Query*-Methode sucht lange im rechten Teilbaum, bevor sie einen Knoten aus dem linken überprüft. In diesem Fall wird der *Top-Down*-Algorithmus besser bewertet, weil er sofort mit dem Teilbaum, wo der Fehler liegt, anfängt.

- *Top-Down kann gleichwertig zu Divide-and-Query sein*: Nehmen wir an, daß der Ableitungsbaum fast vollständig ausgeglichen ist. Es bedeutet, daß alle Knoten $b > 1$ Vorfahren haben, und die Blätter sind auf der gleichen Ebene. Die Tiefe des Baums ist $h$. In diesem Fall wird der berechnete mittlere Knoten der am weitesten links stehende Kindknoten der Wurzel sein. Wenn dieser Knoten korrekt ist und $b > 2$ ist, dann geht der Algorithmus weiter und der nächste berechnete Knoten wird der zweite Nachfolger von links. Wenn der Fehler in dem Knoten ist, der ganz rechts steht, dann werden alle Nachkommen zuerst gefragt, bis ein falscher Knoten gefunden wird. Es bedeutet, daß das Verhalten der beiden Algorithmen in diesem Fall gleichwertig ist, beide haben nämlich die Fragekomplexität $bh$.

Der Vorteil der *Divide-and-Query*-Methode besteht darin, daß die Fragekomplexität im schlechtesten Fall für beliebige Bäume logarithmisch ist. Dabei ist die Auswahl des nächsten zu überprüfenden Knoten durch Berechnungsregeln definiert und völlig vom Fehler unabhängig. Demgegenüber überprüft die *Top-Down*-Methode die Knoten in einer von Fehlern gesteuerten Reihenfolge.

## 3.2 Terminierung ohne Ausgabe

Shapiro hat einen Algorithmus für die Diagnose von Fehlern entwickelt, bei denen es keine Ausgabe gibt. Diese Art von Fehlern kann nur in nichtdeterministischen Sprachen auftreten. Wie wir schon wissen, ist PROLOG eine nichtdeterministische Programmiersprache. Obwohl alle Berechnungen von $C \in \mathbf{P}$ mit $x$ als Eingabe terminieren und die gelieferten Ausgaben (sofern sie existieren) korrekt sind ($\in \mathcal{I}$), gibt es trotzdem keine Ableitung, die $y$ als Ausgabe liefert. So ein Verhalten eines Programms heißt *endliches Fehlschlagen* (*finitely fail*) und benötigt eine spezielle Behandlung (vergleiche [AvE82], [Cla82]).

**Definition 48** Ein Programm **P** *schlägt* nach endlich vielen Schritten (*finitely fail*) auf $< C, x, y >$ *fehl*, wenn alle Berechnungen von $C$ mit Eingabe $x$ terminieren und die Ausgabe korrekt in $\mathcal{I}$ ist, aber keine Berechnung $y$ zurückgibt.

Um die von Shapiro verwendete Methode zu erklären, benötigen wir hier noch einige Begriffe.

**Definition 49** Eine *Klausel* $C$ ist *vollständig* (*complete*) in Bezug auf $\mathcal{I}$, wenn für alle Tripel $< C, x, y >$ in $\mathcal{I}$ das Programm **P** das Tripel $< C, x, y >$ in Bezug auf $\mathcal{I}$ abdeckt.

**Definition 50** Ein *Programm* **P** ist *vollständig* in $\mathcal{I}$, wenn es für alle Tripel $< C, x, y > \in \mathcal{I}$ eine Berechnung von $C \in \mathbf{P}$ mit Eingabe $x$ gibt, die $y$ zurückliefert. Wenn ein Programm endlich fehlschlägt, dann ist es nicht vollständig in $\mathcal{I}$.

Mit Hilfe dieser Definitionen hat Shapiro den folgenden Satz formuliert:

**Satz 10** Seien **P** ein Programm und $\mathcal{I}$ eine Interpretation. Wenn **P** auf $< C, x, y >$ in $\mathcal{I}$ nach endlich vielen Schritten fehlschlägt, dann enthält **P** eine unvollständige Klausel, oder es fehlt eine Klausel in **P**.

Der Beweis beruht auf der Induktion über die maximale Tiefe des Ableitungbaums mit der Wurzel $< C, x, y >$ und kann bei Shapiro [Sha83] nachgelesen werden. Mit anderen Worten, wenn ein Programm **P** mit einem Ziel $G$ in $\mathcal{I}$ endlich fehlschlägt, dann gibt es ein anderes Ziel $G_1$ in $\mathcal{I}$, so daß **P** mit $G_1$ sofort fehlschlägt (*immediately fail*). Sofort Fehlschlagen bedeutet, daß es keine Klausel $A \leftarrow B$ in **P** gibt, so daß $A$ mit $G_1$ unifizierbar ist.

Der hier beschriebene Algorithmus stellt eine andere Art von Fragen als die, die wir bisher benutzt haben. Eine *Existenzfrage* ist ein Paar $< C, x >$. Die Antwort auf diese Frage in einer Interpretation $\mathcal{I}$ ist eine endliche Menge $\{y| < C, x, y > \text{ ist in } \mathcal{I}\}$.

**Algorithmus 10** Eine unvollständige Klausel finden

**Eingabe**: $< C, x, y >$ in $\mathcal{I}$, wobei $C$ endlich fehlschlägt
**Ausgabe**: $< C_j, x_j, y_j >$ in $\mathcal{I}$, wobei $C_j$ nicht abgedeckt ist
**procedure** $incomplete\_procedure(< C, x, y >, < C_j, x_j, y_j >)$
**begin**
  found:=false; $i := 1$
  let $\{< C_1, x_1, y_1 >, \ldots < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$,
  where $C : A_0 \leftarrow A_1, \ldots A_n$ and $\forall i\ 1 \leq i \leq n\ A_i = C_i\theta$
  **if** $\exists\ t$ **then**
    $satisfiable(t$ , success, Instances)
    **if** success = false **then** $< C_j, x_j, y_j > := < C, x, y >$
    **else**
      **while** $(i \leq n)$ and (not found) **do**
        **if** call($< C_i, x_i, y_i >$) = fail
        **then** $incomplete\_procedure(< C_i, x_i, y_i >, < C_j, x_j, y_j >)$; found := true
        **else** i:= i+1
        **endif**
      **endwhile**
    **endif**
  **else** $< C_j, x_j, y_j > := < C, x, y >$ {endliches Fehlschlagen}
  **endif**
**end**

Die Prozedur *incomplete-procedure* (siehe Algorithmus 10) arbeitet wie folgt: Sie traversiert die Ableitung, die zu *fail* führt, vorausgesetzt, daß das Ziel wahr ist und endlich fehlschlägt. Der Fehler kann auf zwei verschiedene Weisen gefunden werden: Entweder findet die Prozedur keine Klausel mehr, mit der die Ableitung fortgesetzt werden kann (endliches Fehlschlagen), oder *satisfiable* (siehe Algorithmus 11) liefert *false*, weil es keine Lösung der Klausel gibt. Wenn aber *satisfiable true* liefert, wird die darunterliegende Ebene des Ableitungsbaums mit einem rekursiven Aufruf der Prozedur durchsucht. Der Hauptalgorithmus enthält zwei weitere Prozeduren, deren Beschreibung jetzt folgt.

**Algorithmus 11** Suche nach einer Instanz der Klausel, wenn diese existiert

**Eingabe**: top level trace $t$
**Ausgabe**: success: *false*, wenn es keine Instanz gibt
wenn es *true* ist, dann gibt es die Instanzen zurück
**procedure** $satisfiable$(t,success,Instance)
**begin**
$i := 1$; Instance := {}; $n$:=length(t)
**while** $(i \leq n)$ **and** (Query $(< C_i, x_i, y_i > \in t)$ = yes ) **do**
Instance := Instance $\cup\ y_i$; $i := i + 1$
**endwhile**
**if** $i > n$ **then** success := true
**else** success := false
**endif**
**end**

Die Funktion *Query* (siehe Algorithmus 12) wird mit dem Tripel $< C, x, y >$ aufgerufen, das auch Variablen enthalten kann, und liefert als Ausgabe einen booleschen Wert, der angibt, ob $< C, x, y >$ aus $\mathcal{I}$ ist oder nicht. Sollte $< C, x, y >$ aus $\mathcal{I}$ sein und ungebundene Variablen enthalten, so werden die Variablenwerte vom Benutzer erfragt und die Variablen damit gebunden. Diese Funktion ist eine modifizierte Version von Algorithmus 4.

**Algorithmus 12** Stellen einer Existenzfrage

**Eingabe**: $< C, x, y >$ ein Atom, das auch ungebundene Variablen enthalten kann
**Ausgabe**: *yes* und $< C, x, y >$ wird gebunden, wenn $< C, x, y > \in \mathcal{I}$
*no*, wenn $< C, x, y > \notin \mathcal{I}$
**function** $Query < C, x, y >$):{*yes* and *Instance no*}
**begin**
answer := "Is the goal $< C, x, y >$ correct? (y/n)"
**if** answer = false **then return**(answer)
**else**
**if** variable($< C, x, y >$) **then**
write('Which value');
*Instance* = list of values of the variables;
**return**(answer);
bind variables in $< C, x, y >$ with Instance;
**else return**(answer)
**endif**
**endif**
**end**

**Beispiel 14** Nehmen wir unser *compute*-Beispiel, aber mit einer modifizierten Version der Anfrage:

```
program([1,2,3,4],[3,2,4],C,S).
```

Das Programm liefert mit diesem Ziel *fail* zurück:

```
This fact fails. Is it correct or not (y/n/c) n
```

Das ist natürlich nicht korrekt. Also versuchen wir, den Fehler im Programm zu finden.

```
error: missing solution program([1,2,3,4],[3,2,4],X,Y). diagnosing...
```

Den Ableitungsbaum zu dem Programm und dem neuen Ziel sieht man in Abbildung 3.6. Der Original-algorithmus kann keinen Ableitungsbaum erzeugen, wenn es keine vollständige Ableitung gibt. Wir haben den Originalalgorithmus so modifiziert, daß in jedem Fall ein Baum generiert wird. Es können in der Ableitung auch Variablen auftreten. Der Algorithmus von Shapiro ersetzt die Repräsentation der Variablenbezeichner in PROLOG durch Variablennamen aus einer vordefinierten Liste, anstatt die ursprünglichen Variablennamen aus dem Programm beizubehalten. Wenn das Verfahren im aktuellen Atom eine Variable findet, ersetzt es die erste Variable mit dem ersten Element der Liste und so weiter. Weil es immer von vorne beginnt, kann es vorkommen, daß die Variablen im Ableitungsbaum nicht weitergeleitet werden, sondern die gleiche Variable mit einem anderem Namen identifiziert wird, wie zum Beispiel im Atom `compute([1,2,3,4],[3,2,4],X,Y)`. Hier werden zwei Variablen `X` und `Y` eingesetzt, aber im nächsten Schritt der Ableitung im Atom `sigma([1,2,3,4],[3,2,4],X)` wird statt der Variablen `Y`, die aus der Sicht der Ableitung korrekt wäre, wieder der Name `X` verwendet, weil dieses Atom nur eine Variable enthält und daher die erste Bezeichnung aus der Liste `X, Y, Z, U, V, W ...` genommen wird.

Wie sich dieses Verhalten korrigieren läßt und andere Verbesserungen beschreiben wir im Kapitel 4.4.
Der oben beschriebene Algorithmus stellt dann die folgenden Fragen:

```
This fact fails. Is it correct or not? (y/n/c) n

error: missing solution program([1,2,3,4],[3,2,4],X,Y). diagnosing...

Query:program([1,2,3,4],[3,2,4],X,Y)? y
Which X ?[1].
Which Y ?19.
```

Bei dieser Methode kann es vorkommen, daß das gefragte Atom Variablen enthält. Wenn dies der Fall ist, dann sammelt das Verfahren die Variablen und erwartet vom Anwender, daß er diese Variablen mit einem Wert bindet. Der Original-algorithmus kann nur eine einzige Variable pro Atom behandeln. Wir haben das System so erweitert, daß beliebig viele Variablen behandelt und unifiziert werden können.

```
Query:compute([1,2,3,4],[3,2,4],[1],19)? y

Query:sigma([1,2,3,4],[3,2,4],19)? y

Query:complement([1,2,3,4],[3,2,4],[1])? y
```

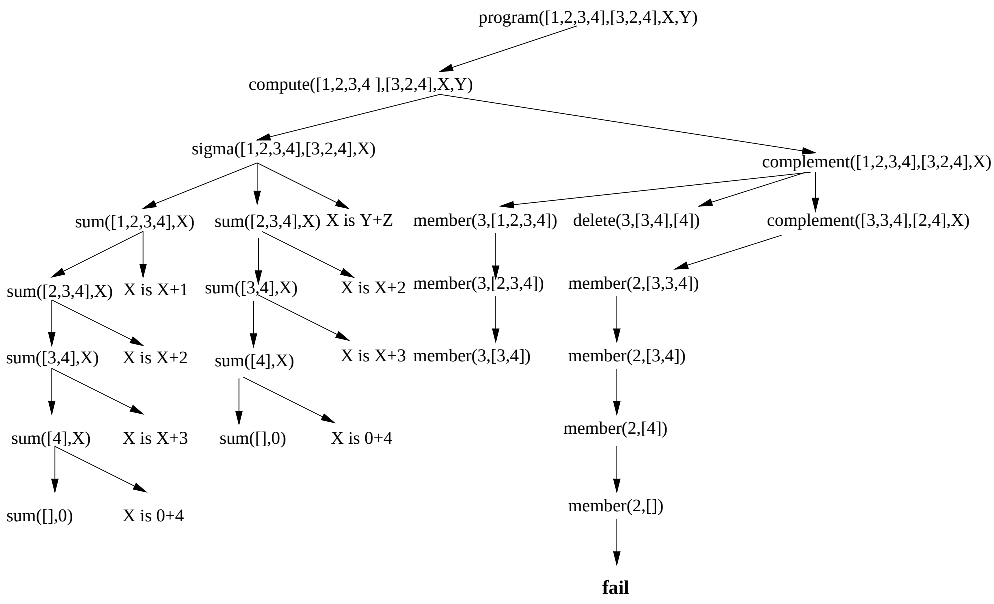



Abbildung 3.6: Der Ableitungsbaum zu dem Ziel `program([1,2,3,4],[3,2,4],C,S)`

```
Query:member(3,[1,2,3,4])? y

Query:delete(3,[1,2,3,4],X)? y
Which X ?[1,2,4].

[Error diagnosed:,delete(3,[1,2,3,4],[1,2,4]), is uncovered.]
Listing of delete(X,Y,Z):
delete(X,[X|Y],Y):-true.
delete(X,[Z|U],[X|Y]):-delete(X,U,Y).
```

Der Algorithmus hat festgestellt, daß es keine Möglichkeit gibt, die Klausel `delete` mit der Eingabe `3` und `[1,2,3,4]` abzuleiten, wenn die Ausgabe `[1,2,4]` sein soll, obwohl der Anwender die Frage mit $y$ beantwortet hat, was bedeutet, daß die Klausel korrekt sein muß. Die Ableitung der Klausel `delete` enthält keine weitere Klausel, mit der der Prozeß fortfahren könnte. Daraus folgt, daß der Fehler in der Definition von `delete` liegt. Daher listet das Verfahren die Klauseln auf, die im Kopf den Prädikatnamen `delete` haben. Es kann nicht genau lokalisieren, welche der beiden Klauseln fehlerhaft ist.

Was die Komplexität des Algorithmus betrifft, seien $n$ die Länge, $d$ die Tiefe des Ableitungsbaums mit der Wurzel $< C, x, y >$. $b$ ist die maximale Größe der Vereinigung aller *top level traces* aller $< C_j, x_j, y_j >$, die aus $< C, x, z >$ ableitbar sind, wobei z beliebig ist. Shapiro hat durch ein Beispiel gezeigt, daß die Durchführung des Algorithmus höchstens die Länge $dn+1$ und die Tiefe $d+1$ hat, und er braucht $b(d-1)+1$ Fragen, um die Tripel $< C_j, x_j, y_j >$ zu bestimmen, die in $\mathcal{I}$ bei $C_j$ nicht abgedeckt sind, wenn es mit $< C, x, y >$ in $\mathcal{I}$ aufgerufen wird.

Der Beweis beruht auf der vollständigen Induktion über $d$ unter der Voraussetzung, daß es nur eine endliche Menge von Ableitungsbäumen mit der Wurzel $< C, x, y >$ gibt, weil $\mathbf{P}$ keinen unendlichen Ableitungsbaum mit der Wurzel $< C, x, z >$ hat.

## 3.3 Nichtterminierung

Wie können wir feststellen, daß sich das Programm in einer Endlosschleife befindet? Eine Möglichkeit wäre natürlich, wenn die Berechnung die von dem Rechner zur Verfügung gestellten Speicher- oder Zeitressourcen verbraucht, wobei man aber nicht sicher sein kann, daß das Programm tatsächlich nicht terminiert. In den meisten Fällen ist die Geduld des Benutzers erschöpft, und er bricht die Berechnung ab. Das hier vorgestellte Verfahren verwendet die Programmterminierungs-Methode von Floyd ([Flo67]). Viele andere Artikel beschäftigen sich ebenfalls mit diesem Problem, unter anderen [Cov85a], [Cov85b] und [Nut85].

**Definition 51** Sei $S$ eine nicht leere Menge. Eine *wohldefinierte Ordnung* (*well founded ordering*) auf $S$ ist eine partielle Ordnung auf $S$, so daß es keine unendlich absteigenden Sequenzen gibt. $\succ$ ist die asymmetrische, irreflexive und transitive binäre Relation dieser Ordnung.

Für Programme kann der folgende Satz definiert werden.

**Satz 11** Ein Programm $\mathbf{P}$ *terminiert* genau dann immer, wenn es eine wohl fundierte Ordnung $\succ$ auf den Prozeduraufrufen gibt, so daß für alle Berechnungen in $\mathbf{P}$ gilt: Wenn $< C, x >$ das Tupel $< C_j, x_j >$ aufruft, dann ist $< C, x >\succ< C_j, x_j >$ .

Es kann definiert werden, was es bedeutet, wenn eine Klausel divergiert.

**Definition 52** Seien $\mathcal{I}$ eine Interpretation, $\succ$ eine wohlfundierte Ordnung auf den Klauselaufrufen. Eine Klausel $C$ *divergiert* in Bezug auf $\succ$ und $\mathcal{I}$, wenn sie ein Tripel $< C, x, y >$ mit dem *top level trace* $S$ besitzt, für das gilt:

- $(\exists < C_j, x_j, y_j >\in S)(< C, x >\not\succ< C_j, x_j >)$
- $(\forall < C_i, x_i, y_i >\succ< C_j, x_j, y_j >)(< C_i, x_i, y_i >\in \mathcal{I})$

Aber wenn $C \in \mathbf{P}$ die Ordnung $\succ$ verletzt, bedeutet es nicht unbedingt, daß $C$ fehlerhaft ist. Allein die Ordnung zu kennen, reicht nicht aus. Man muß auch die beabsichtigte Interpretation kennen.

**Definition 53** Eine Klausel befindet sich in einer *Endlosrekursion*, wenn sie sich immer wieder selbst mit der gleichen Eingabe aufruft.

**Satz 12** Seien $\mathbf{P}$ ein Programm, $\succ$ eine wohldefinierte Ordnung über den Prozeduraufrufen und $\mathcal{I}$ eine Interpretation. Wenn $\mathbf{P}$ divergiert, dann enthält es eine Klausel, die entweder inkorrekt ist oder mit $\succ$ und $\mathcal{I}$ divergiert.

Jetzt sind wir so weit, daß wir den Algorithmus von Shapiro für die Feststellung der Nichtterminierung beschreiben können. Wir nehmen dabei an, daß der Anwender eine grobe Vorstellung davon hat, wie die Aufrufrelation in seinem Programm aussehen sollte. Mit anderen Worten, er kann für alle Klauseln in $\mathbf{P}$ entscheiden, ob $< C, X >\succ< C_j, x_j >$ gilt. Außerdem wird angenommen, daß es für alle Berechnungen von $C$, die mit $x$ aufgerufen werden, eine obere Schranke der Tiefe $d$ gibt, die angibt, wie oft die Klausel mit der gleichen Eingabe aufgerufen werden kann. Die wesentlichen Schritte des von Shapiro beschriebenen Algorithmus sind die folgenden:

- Suche nach einem Segment, das sich in einer Endlosrekursion befindet: $< C, x >, < C, x >, \cdots$

- Wenn so etwas gefunden wurde, dann wird überprüft, ob es in diesem Segment zwei aufeinander folgende Aufrufe von Klauseln gibt, die $\succ$ verletzen. Hier muß dann natürlich der Benutzer gefragt werden.

---

**Algorithmus 13** Simulierung der Durchführung eines PROLOG-Programms

---

**Eingabe**: eine Klausel $C \in \mathbf{P}$, eine Eingabe $x$,
die Tiefe $d$, so daß die Berechnung von $C$ mit $x$ diesen Wert nicht überschreitet,
eine Interpretation $\mathcal{I}$,
der Anfangsaufrufstack *StackIn*
**Ausgabe:** ein Wahrheitswert *succeed*,
der Aufrufstack *Stack*
**procedure** $simulate(< C, x, \_ >$, d, $\mathcal{I}$, StackIn, Stack, succeed)
**begin**
  **if** $d = 0$ **then** Stack := push(StackIn, $< C, x, \_ >$); succeed := false
  **else**
    let $\{< C_1, x_1, y_1 >, \ldots < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$;
    succeed = true
    **while** $i \leq n$ **and** succeed **do**
      $simulate(< C_i, x_i, y_i >$, $d - 1$, $\mathcal{I}$,push(StackIn, $< C, x, y >$), Stack, succeed)
      $i := i + 1$
    **endwhile**
  **endif**
**end**

---

Algorithmus 13 versucht, $C$ mit einer Eingabe $x$ zu berechnen, aber wenn diese Klausel häufiger als die vordefinierte Schranke $d$ aufgerufen wird, dann wird die Variable *succeed* auf *false* gestellt. Wenn $d$ nicht überschritten wird, dann bleibt *succeed true*.

**Algorithmus 14** Der Algorithmus im Fall Nichtterminierung

**Eingabe**: eine Klausel $C \in \mathbf{P}$, eine Eingabe $x$,
die Tiefe $d$, so daß die Berechnung von $C$ mit $x$ diesen Wert nicht überschreitet,
eine Interpretation $\mathcal{I}$
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
so daß $C_j$ mit $x_j$ als Eingabe $y_j$ liefert oder
zwei Aufrufe von Klauseln $< C_j, x_j >$ und $< C_i, x_i >$,
die $\succ$ verletzen, oder der Text *no divergence found* oder der Aufrufstack *stack*
**function** $diverging(< C, x,_> , d, \mathcal{I})$:{$C_{buggy}$ oder
$< C_j, x_j >$ und $< C_i, x_i >$ oder *no divergence found*} *oder stack*
**begin**
$j := 1$; $k := 1$; found := $false$;
$simulate(< C, x, \_ >$, d, $\mathcal{I})$, [], $Stack$, succeed)
**if** succeed **return**(stack)
**else**
length := $|stack|$
**while** ($j \leq length$) **and** $\neg found$ **do**
**if** $< C, x > \not\succ < C_j, x_j >$ **then** found := $true$
**else** $j$:=$j$+1
**endif**
**endwhile**
**if** found = $false$ **then** **return**(*no divergence found*)
**else**
**while** ($k \leq j$) **do**
answer := Query$< C_k, x_k, y_k >$
**if** answer = $false$ **then** $divide\_and\_query(< C_k, x_k, y_k >, C_{buggy}, \mathcal{I}')$; **return**($C_{buggy}$)
**else** $k := k + 1$
**endwhile**
**if** $k = j + 1$ **return**($< C, x >, < C_j, x_j >$)
**endif**
**endif**
**endif**
**end**

Algorithmus 14 ruft zuerst Algorithmus 13 auf. Abhängig vom Wert von *succeed* arbeitet er dann weiter: Wenn *succeed false* ist, dann beendet sich Algorithmus 14 und liefert den Aufrufstack zurück. Wenn $d$ nicht überschritten wird, also *succeed* := $true$, dann überprüft das Verfahren im nächsten Schritt den berechneten Keller, um solche Paare zu finden, die $\succ$ verletzen. Wenn so ein Paar existiert, dann wird dem Benutzer für alle vorher aufgetretenen Aufrufe die Frage gestellt, ob die gelieferte Antwort korrekt oder inkorrekt ist. Um diese Frage zu stellen, wird die Funktion *Query* (siehe Algorithmus 4) aufgerufen. Wenn die Antwort *no* ist, dann wird der Fehler mit Hilfe des Algorithmus 8 weiterermittelt. Sonst wird das Paar ($< C, x >, < C_j, x_j >$), das $\succ$ verletzt, zurückgegeben. Wenn überhaupt kein solches Paar existiert, dann liefert die Methode den Text *no divergence found* zurück.

**Beispiel 15** Nehmen wir eine modifizierte Version unseres *compute*-Beispiels:
Jetzt wird die Klausel *delete* korrekt definiert:

```
delete(X,[X|T],T).
delete(X,[Y|Z],[Y|U]) :- delete(X,Z,U).
```

Aber die Beschreibung für *member* enthält einen Fehler, statt

```
member(X,[Y|T]) :- member(X,T).
```

schreiben wir:

```
member(X,[Y|T]) :- member(X,[Y|T]).
```

Wir rufen das Programm mit der Anfrage auf:

```
program([1,2,3,4],[1,2,4],C,S).
```

Die Ableitung bei `member(4,[3,4])` läuft in eine Endlosrekursion. Wenn nämlich das erste Argument in *member* (4) nicht in der Liste ist, die als zweites Argument in *member* angegeben wurde, dann überprüft es nicht den Rest der Liste, sondern nimmt wieder die gesamte Liste. Die Konstante $d$ wird in diesem Fall auf 25 gesetzt. Das bedeutet, nach fünfundzwanzig Aufrufen des Atoms `member(4,[3,4])` bricht die Berechnung ab. Wo die Ableitung scheitert, zeigt die Abbildung 3.7.

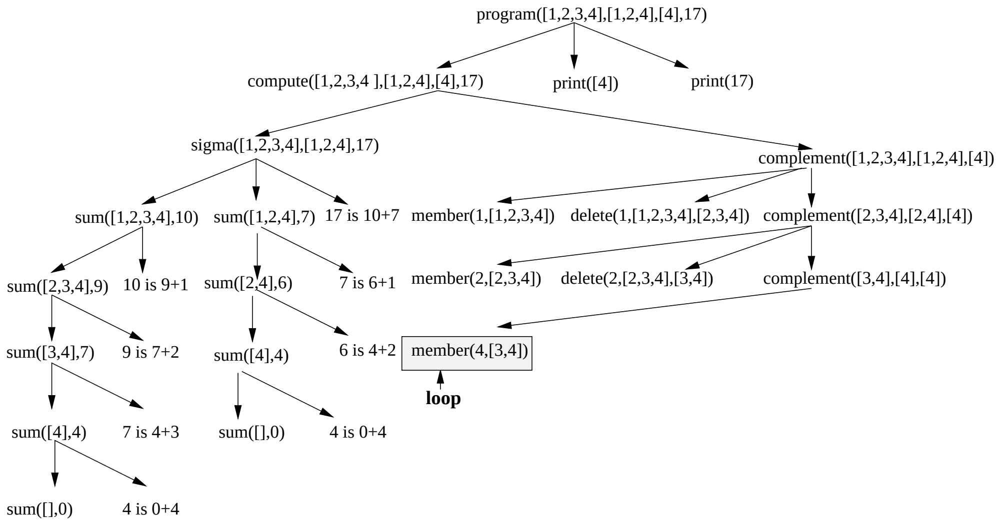



Abbildung 3.7: Die Nichtterminierung mit dem Ziel `compute([1,2,3,4],[1,2,4],C,S)`

In diesem Kapitel haben wir die drei verschiedenen Fehlermöglichkeiten (Terminierung mit inkorrekter Ausgabe, Terminierung ohne Ausgabe und Nichtterminierung), die Shapiro definiert hat, formal und durch Beispiele vorgeführt. Im Fall der Terminierung mit inkorrekter Ausgabe haben wir die drei verschiedenen Durchlaufarten des Ableitungsbaums (Bottom-Up, Top-Down, Divide-and-Query) beschrieben. Wir haben dabei festgestellt, daß die Anzahl der Fragen sehr groß werden kann, deswegen beschreiben wir im nächsten Kapitel unseren ersten Verbesserungsvorschlag: die Verwendung einer Testdatenbank.

# Kapitel 4

# Category Partition Testing Methode (CPM)

In der Regel kann ein PROLOG-Programm nicht mit allen möglichen Eingabeparameterbelegungen getestet werden. Daher muß der Suchraum in sinnvoller Weise eingeschränkt werden. Eine Möglichkeit hierzu ist, die möglichen Parameterbelegungen in eine im allgemeinen kleine Zahl von Klassen zu unterteilen. Die *CPM*-Methode (*Category Partition Testing Method*) nimmt eine solche Einteilung pro Parameter vor und nennt die Klassen *kritische Eigenschaften* (*categories*), die vom Benutzer definiert werden müssen. Nur diese kritischen Eigenschaften werden beim Testen überprüft. Diese Kategorien können außerdem in *Wahlmöglichkeiten* (*choices*) aufgeteilt werden. Aus Kategorien und Wahlmöglichkeiten erzeugt *IDTS* alle möglichen *Testrahmen* (*test frames*). Alle Testrahmen enthalten genau eine Wahlmöglichkeit aus jeder Kategorie. Dabei wird vorausgesetzt, daß das Verhalten der Elemente im Testrahmen identisch ist. Im allgemeinen werden auch Testrahmen generiert, die sinnlos sind. Diese Testrahmen können ausgeschlossen werden, wenn man sogenannte *Selektionsausdrücke* (*selector expressions*) mit dem Testrahmen verbindet. Eine Kombination von Wahlmöglichkeiten ist nur dann ein Testrahmen, wenn der dazugehörige Selektionsausdruck wahr ist. Zusätzlich kann man einer Wahlmöglichkeit noch einen *Property-Namen* (*properties*) zuordnen, der einen Selektionsausdruck enthält. Ein Property-Name ist eine Art logische Variable, deren Wert wahr ist, wenn ein Testrahmen mit dieser Wahlmöglichkeit generiert werden soll. Eine längere, nicht formale Beschreibung von *CPM* kann man in [OB88] finden.

## 4.1 Formale Beschreibung der Category Partition Testing Methode (CPM)

Im folgenden wird der Formalismus dieser Methode beschrieben. Genauer gesagt, geben wir eine an logische Programmierumgebungen angepaßte formale Beschreibung von *CPM* an. Sei $\mathbf{P} = \{C_1, \ldots, C_k\}$ ein logisches Programm, wobei $C_i$ $(1 \leq i \leq k)$ eine Klausel des Programms ist. Wir nehmen an, daß der Wertebereich der Interpretation $\mathcal{I}$ die Menge $\mathcal{D}$ ist. Dann können alle Klauseln $C_i$ in $\mathbf{P}$ als eine Abbildung (mapping) aus $D^n$ in $D^m$ aufgefaßt werden, wobei $n$ und $m$ die Eingabe- und Ausgabestelligkeiten von $C_i$ sind.

**Definition 54** Seien $C_i$ $(1 \leq i \leq k)$ eine Klausel in $\mathbf{P}$ und $n$ die Eingabestelligkeit von $C_i$. Eine Äquivalenzrelation über $D^n$ heißt *Kategorie (category)* von $C_i$.

**Definition 55** Sei $C_i$ $(1 \leq i \leq k)$ eine Klausel in $\mathbf{P}$ und $c$ eine Kategorie von $C_i$. Eine Äquivalenzklasse von $c$ heißt Wahlmöglichkeit (*choice*) von $c$.

Jede Kombination von Wahlmöglichkeiten der Kategorien bestimmt eine Klasse von $D^n$. In *CPM* wird die Wahlmöglichkeit dadurch festgelegt, daß ein Ausdruck (*Suchfunktion*, *searching function*) definiert wird, der von einer Teilmenge der Eingabevariablen abhängig ist. Mit Hilfe dieses Ausdrucks kann automatisch entschieden werden, in welche Wahlmöglichkeit ein beliebiges Element aus $D^n$ gehört. Allen Klauseln wird jeweils eine endliche Menge von Kategorien zugeordnet, und die Kategorien können als endliche Mengen von Wahlmöglichkeiten betrachtet werden.

**Beispiel 16** In diesem Beispiel definieren wir Kategorien und Wahlmöglichkeiten für die Klausel *delete* aus dem von uns oft verwendeten Beispiel *compute* (siehe Beispiel 8).

```
delete(X,[X|T],T).
delete(X,[Y|Z],[X|U]) :- delete(X,Z,U).
```

Es gibt zwei Kategorien. Die erste `number_of_elements` beschreibt, wieviele Elemente die Liste im zweiten Argument dieses Prädikates hat. Aus dieser Liste wollen wir ein Element löschen. Die Liste kann entweder leer sein oder ein oder mehrere Elemente enthalten. Die zweite Kategorie `relationship` beschreibt das Verhältnis zwischen dem ersten Argument des Prädikates, das das Element enthält, das wir löschen wollen, und dem ersten Element der Liste. Die hier vorstellbaren Wahlmöglichkeiten sind: *none* (die Liste ist leer, so daß es keinen Sinn ergibt, etwas zu vergleichen), *greater* (das erste Argument ist größer als das erste Element der Liste), *less* (das erste Argument ist kleiner als das erste Element der Liste) und *equal* (das erste Argument ist gleich dem ersten Element der Liste).

Eine *CPM*-Beschreibung für die Klausel *delete*:
**Test specification:** delete
**Category:** number_of_elements
  choice: empty
  choice: one
  choice: more
**Category:** relationship
  choice: none
  choice: greater
  choice: less
  choice: equal
**End of specification**

Im folgenden definieren wir die *CPM*-Methode formal für PROLOG-Programme.

**Definition 56** Sei $C_i$ eine beliebige Klausel und seien $\Sigma_{C_i} = \{\delta_{C_i 1}, \ldots, \delta_{C_i k}\}$ die Kategorien der Klausel $C_i$, wobei $\delta_{C_i j} = \{c_{C_i j_1}, \ldots c_{C_i j_n}\}$ ( $c_{C_i j l}$ Wahlmöglichkeiten) sind. $\Sigma_{C_i}$ ist die *Testspezifikation der Klausel* $C_i$. Eine *Testspezifikation des Programms* $\mathbf{P}$ ist die Menge $\Sigma_{\mathbf{P}} = \{\Sigma_{C_i} | C_i \in \mathbf{P}\}$.

Um die Bedeutung der Definition 56 zu erläutern, geben wir eine formale Beschreibung der Testspezifikation der Klausel *delete* aus dem Programm *compute* an.

**Beispiel 17** Die formale Beschreibung der Testspezifikation der Klausel *delete*: Eine nicht formale Beschreibung der Suchfunktionen ist bei den Wahlmöglichkeiten in geschweiften Klammern mit angegeben.

$\Sigma_{delete} = \{\delta_{number\_of\_elements}, \delta_{relationship}\}$
$\delta_{number\_of\_elements} = \{c_0, c_1, c_2\}$
$c_0 =$ {die Liste ist leer} empty
$c_1 =$ {die Liste hat genau ein Element} one
$c_2 =$ {die Liste hat mehrere Elemente} more
$\delta_{relationship} = \{d_0, d_1, d_2, d_3\}$
$d_0 = \{(X, [])\}$ none
$d_1 = \{(X, [Y|Z]), X, Y \in D, Z \in D^*$ und $X < Y\}$ greater
$d_2 = \{(X, [Y|Z]), X, Y \in D, Z \in D^*$ und $X > Y\}$ less
$d_3 = \{(X, [Y|Z]), X, Y \in D, Z \in D^*$ und $X = Y\}$ equal

Es ist klar, daß $\delta_{number\_of\_elements}$ und $\delta_{relationship}$ Äquivalenzrelationen über $D \times D^*$ definieren.

**Definition 57** Seien $C_i$ eine Klausel in $\mathbf{P}$ und $\Sigma_{C_i} = \{\delta_{C_i1}, \ldots, \delta_{C_ik}\}$ eine Testspezifikation von $C_i$. Sei $F(\Sigma_{C_i})$ eine Teilmenge des kartesischen Produkts $\delta_{C_i1} \times \ldots \times \delta_{C_ik}$. $F(\Sigma_{C_i})$ heißt *die Menge der Testrahmen* von $C_i$ und wir nehmen an, daß $F(\Sigma_{C_i})$ das ganze $D^n$ abdeckt. Die Menge $F(\Sigma_{C_i})$ ist eine Klassifizierung von $D^n$. Deswegen kann $F(\Sigma_{C_i})$ als eine Äquivalenzrelation über $D^n$ betrachtet werden. Ein Element in $F(\Sigma_{C_i})$ heißt Testrahmen von $C_i$. Die Menge der *Testrahmen* des *Programms* $\mathbf{P}$ ist die Menge $F(\Sigma_{\mathbf{P}}) = \bigcup_{C_i \in \mathbf{P}} F(\Sigma_{C_i})$.

Nach der Definition 57 können wir die Testrahmen für die im Beispiel 17 definierte Testspezifikation angeben.

**Beispiel 18** Testrahmen der Klausel *delete*

$$F(\Sigma_{delete}) = \{(c_0, d_0), (c_0, d_1), (c_0, d_2), (c_0, d_3), (c_1, d_0), (c_1, d_1), (c_1, d_2), (c_1, d_3),\\ (c_2, d_0), (c_2, d_1), (c_2, d_2), (c_2, d_3)\}$$

**Definition 58** Seien $C_i$ eine Klausel in $\mathbf{P}$, $\Sigma_{C_i} = \{\delta_{C_i1}, \ldots, \delta_{C_ik}\}$ eine Testspezifikation von $C_i$ und $f = \{c_{C_i1}, \ldots, c_{C_ik}\} (c_{C_ij} \in \delta_{C_ij})$ ein Testrahmen von $C_i$. Ein *Testfall* von $f$ ist ein Element $(d_{C_i1}, \ldots, d_{C_ik})$, wobei $d_{C_ij}$ in der Wahlmöglichkeit (*choice*) $c_{C_ij}$ $(1 \leq j \leq k)$ ist.

Jetzt sind wir so weit, daß wir aus den definierten Kategorien und Wahlmöglichkeiten die Testrahmen generieren können. Dabei muß beachtet werden, daß es in manchen Fällen keinen Sinn ergibt, bestimmte Kombinationen zu generieren. Wenn zum Beispiel die Liste leer ist, dann kann nicht überprüft werden, welcher Zusammenhang zwischen dem Element und der Liste besteht. Wie diese Testrahmen mit Hilfe von Selektionsausdrücken reduziert werden können, beschreiben wir später.

**Beispiel 19** Die generierten Testrahmen für die Klausel *delete* mit jeweils einem Beispiel.

| | |
|---|---|
| $(c_0, d_0)$ | $(2, [])$ |
| $(c_0, d_1)$ | sinnlos |
| $(c_0, d_2)$ | sinnlos |
| $(c_0, d_3)$ | sinnlos |
| $(c_1, d_0)$ | sinnlos |
| $(c_1, d_1)$ | $(1, [2])$ |
| $(c_1, d_2)$ | $(3, [2])$ |
| $(c_1, d_3)$ | $(2, [2])$ |
| $(c_2, d_0)$ | sinnlos |
| $(c_2, d_1)$ | $(0, [1, 2, 4])$ |
| $(c_2, d_2)$ | $(2, [1, 2, 4])$ |
| $(c_2, d_3)$ | $(1, [1, 2, 4])$ |

Es kann vorkommen, daß eine Klasse aus $D^n$ leer ist, also darin keine Testfälle existieren. Ein Testrahmen kann als ein Element aus dem Produkt der Äquivalenzrelation über $D^n$ betrachtet werden. Die *CPM*-Methode überprüft das Verhalten der $C_i$ für den gegebenen Testrahmen durch die Überprüfung der Ergebnisse von $C_i$ an einem repräsentativen Element (*Testfall, testcase*) aus diesem Testrahmen. Mitte des Debuggens wird es angenommen, daß die Menge der Testrahmen teilweise bewertet ist, es gibt also Testrahmen, die noch nicht bewertet sind, das ist im allgemeinen die Situation in der Praxis. Deshalb ordnen wir der Menge der Testrahmen eine Bewertungsfunktion zu, die die Testrahmen in die Menge $\{$*correct, incorrect, undefined*$\}$ abbildet.

**Definition 59** Seien $C_i$ eine Klausel in $\mathbf{P}$ und $\Sigma_{C_i}$ eine Testspezifikation von $C_i$. Ein *CPM-Test von $C_i$* durch $\Sigma_{C_i}$ ist eine Zuordnung $\varphi_{C_i} : F(\Sigma_{C_i}) \to \{correct, incorrect, undefined\}$. Seien $f$ ein beliebiger Testrahmen von $C_i$ und $t$ ein Testfall aus $f$, dann ist $\varphi_{C_i}(f)$ genau dann *correct/incorrect*, wenn $C_i$ auf $t$ angewendet wird und die vom Anwender angegebene Antwort *true/false* ist. Sonst ist $\varphi_{C_i}(f)$ *undefined*.

**Definition 60** Seien $C_i$ eine beliebige Klausel in $\mathbf{P}$ und $\Sigma_{C_i}$ eine beliebige Testspezifikation von $\mathbf{P}$. Sei $\chi_{C_i} : D^n \to F(\Sigma_{C_i})$ eine Zuordnung, so gilt $\chi_{C_i}(x) = f$ genau dann, wenn $f$ der entsprechende Testrahmen für $x$ ist. Aus den früheren Definitionen folgt, daß $\chi_{C_i}$ eine Funktion, nämlich eine *Suchfunktion* (*searching function*), ist.

**Definition 61** Sei $f$ ein beliebiger Testrahmen von $C_i$ aus $\Sigma_{C_i}$. Wir sagen, daß der *CPM-Test* von $f$ genau dann *konsistent* ist, wenn $\varphi_{C_i}$ nicht *undefined* und $\varphi_{C_i}(f)$ unabhängig von der Wahl des Testfalls ist.

**Definition 62** Sei $C_i$ eine beliebige Klausel in $\mathbf{P}$ und $\Sigma_{C_i}$ eine Testspezifikation von $C_i$. Der *CPM*-Test von $C_i$ ist genau dann *konsistent*, wenn er für alle Testrahmen von $C_i$ über $\Sigma_{C_i}$ konsistent ist. Der *CPM*-Test von $\mathbf{P}$ ist genau dann *konsistent*, wenn alle $C_i$ in $\mathbf{P}$ konsistent sind.

Mit dem Fall inkonsistenter Testrahmen beschäftigen wir uns im Kapitel 4.3.2.

**Definition 63** Eine *CPM*-Konfiguration eines Programms $\mathbf{P}$ über der Menge der Testrahmen $F(\Sigma_{C_j})$ ist eine endliche Teilmenge von $T(F(\Sigma_{C_j})) = \{(C_j, a, f, i, o, e) | C_j \in \mathbf{P}\}$, wobei gilt:

- $a$ ist die Stelligkeit von $C_j$,
- $f \in F(\Sigma_{C_j})$ ist ein Testrahmen von $C_j$,
- $o$ ist die Ausgabe von $C_j$, wenn $i$ die Eingabe war
- $e \in \{$ correct, incorrect, undefined$\}$ ist die Bewertung des Testrahmens $f$ $(e = \varphi_{C_j}(f))$.

Eine *CPM*-Testkonfiguration kann als eine Testdatenbank betrachtet werden. Für die Klauseln $C_j \in \mathbf{P}$, die eine *CPM*-Konfiguration haben, ist $C_j$ die erste Komponente aller Elemente von $T(F(\Sigma_{C_j}))$.

Wir geben jetzt für unser Beispiel die *CPM*-Testkonfiguration $T(F(\Sigma_{delete}))$ an und zeigen, wie $\varphi$ und $\chi$ funktionieren. Wir rufen unser Beispiel *compute* mit der Anfrage `program([1,2,3,4],[1,2,4],C,S)` auf. Das Ergebnis `program([1,2,3,4],[1,2,4],[4],17)` ist inkorrekt. Wir beschreiben die Schritte, bei denen der Benutzer über die Klausel *delete* befragt wird, und wie die Datenbank abhängig von der Antwort geändert wird.

**Beispiel 20** Die *CPM*-Testkonfiguration $T(F(\Sigma_{delete}))$ enthält am Anfang keine Ein- und Ausgabe, und alle Bewertungen sind *undefined.*

$T(F(\Sigma_{delete}))$ :
$(delete, (empty, none), \_, \_, undefined)$
$(delete, (one, greater), \_, \_, undefined)$
$(delete, (one, less), \_, \_, undefined)$
$(delete, (one, equal), \_, \_, undefined)$
$(delete, (more, greater), \_, \_, undefined)$
$(delete, (more, less), \_, \_, undefined)$
$(delete, (more, equal), \_, \_, undefined)$

Wenn die Top-Down-Methode verwendet wird, werden beim Debugging die folgenden Fragen gestellt:

| | |
|---|---|
| delete(1,[1,2,3,4],[2,3,4]) | $\chi_{delete}$(1,[1,2,3,4]) = (more,equal) |
| | answer = $\varphi_{delete}(more, equal)$ = *true* |
| delete(4,[4],[]) | $\chi_{delete}$(4,[4]) = (one,equal) |
| | answer = $\varphi_{delete}(one, equal)$ = *true* |
| delete(4,[3,4],[4]) | $\chi_{delete}$(4,[3,4]) = (more,greater) |
| | answer = $\varphi_{delete}(more, greater)$ = *false* |

Die gegebenen Antworten führen zu den folgenden Änderungen in der Datenbank:

$T(F(\Sigma_{delete}))$ :
$(delete, (empty, none), \_, \_, undefined)$
$(delete, (one, greater), \_, \_, undefined)$

$(delete, (one, less), \_, \_, \mathit{undefined})$
$(delete, (one, equal), (4, [4]), ([]), \mathit{correct}\ )$
$(delete, (more, greater), (4, [3, 4]), ([4]), \mathit{incorrect})$
$(delete, (more, less), \_, \_, \mathit{undefined})$
$(delete, (more, equal), (1, [1, 2, 3, 4]), ([2, 3, 4]), \mathit{correct})$

Bislang haben wir formal beschrieben, wie unsere Testspezifikation aufgebaut wird. Im nächsten Beispiel zeigen wir, wie sie in PROLOG vom Anwender eingegeben werden muß. Wir beschränken uns hierbei nur auf die Beschreibung des *delete* Prädikates. Die Beschreibung der anderen Prädikate folgt dem gleichen Schema.

**Beispiel 21** Die PROLOG-Repräsentation einer *CPM*-Beschreibung lautet:

```
%       test description:
globaltype(all).
type(delete(+,+,+)).
```

Hier kann sowohl global als auch für jedes Prädikat einzeln eingestellt werden, ob bestimmte Parameterpositionen von *IDTS* ausgegeben werden sollen oder nicht. Im Detail ist die Bedeutung der oberen zwei Zeilen die folgende: Bei der Entwicklung des Systems haben wir festgestellt, daß bei einigen Programmen, in denen wir den Fehler suchen wollen, die Parameterliste ziemlich groß sein kann (siehe *EKG*-Beispiel 8.5, Kapitel 8). Außerdem haben hinsichtlich des Debuggings nicht alle Parameter die gleiche Bedeutung für die Beantwortung der gestellten Fragen. Wir haben eine Möglichkeit in das System eingebaut, daß der Anwender selbst definieren kann, welche Parameterinstanzen er sehen möchte und welche nicht. Das geschieht dadurch, daß er entweder "+" oder "−" eingibt. Bei + an der Parameterstelle wird der gerade aktuelle Wert dieses Parameters erscheinen. Bei "−" kennzeichnet das System beim Debugging den Parameter nur mit einem Fragezeichen (?), statt seinen Wert auszugeben. Für diejenigen Prädikate, die keine solche Einschränkung haben, gilt entweder `globaltype(all)` oder `globaltype(nothing)`, wobei das erste bedeutet, daß alle Parameterinstanzen erscheinen werden, und das zweite, daß nur die Prädikatnamen aufgelistet werden.

```
% first category for delete/3 : number_of_elements
 choice_of(delete,3, number_of_elements, empty).
 choice_of(delete,3, number_of_elements, one).
 choice_of(delete,3, number_of_elements, more).

% second category for delete/3 : relationship
 choice_of(delete,3, relation, none).
 choice_of(delete,3, relation, greater).
 choice_of(delete,3, relation, less).
 choice_of(delete,3, relation, equal).
```

Hier kann der Benutzer die Kategorien und die dazugehörenden Wahlmöglichkeiten definieren. Das PROLOG-Fakt beginnt immer mit dem Prädikatnamen `choice_of`. Diesem folgt der Name und die Stelligkeit des Prädikats, zu dem die Kategorien und Wahlmöglichkeiten gehören. Die Stelligkeit wird benötigt, weil man in PROLOG zwei verschiedene Prädikate mit gleichem Namen, aber unterschiedlicher Stelligkeit definieren kann. An dritter Stelle steht der Kategorienname und diesem folgt die Wahlmöglichkeit.

```
%   searching functions:
 empty(delete,3,_,[],_).
```

```
one(delete,3,_,X,_)      :- length(X,L), L ==1.
more(delete,3,_,X,_)     :- length(X,L), L >1.

none(delete, 3, _,[],_).
greater(delete, 3, X,[Y|_],_) :- X > Y.
less(delete, 3, X,[Y|_],_)    :- X < Y.
equal(delete,3, X,[Y|_],_)    :- X = Y.
```

Hier werden die Suchfunktionen angegeben. Mit Hilfe dieser Funktionen kann entschieden werden, welche Wahlmöglichkeit das Prädikat mit der aktuellen Parameterbelegung enthält. Diese Funktionen sind als PROLOG-Klauseln implementiert. Im Kopf steht der Wahlmöglichkeitenname als Prädikatname. Die Argumente sind der Prädikatname und seine Stelligkeit. Dahinter stehen die Argumente des Prädikats. Auf der rechten Seite steht das PROLOG-Programm, das durch Verwendung von Variablen des Prädikats entscheiden kann, ob das Prädikat mit der aktuellen Parameterbelegung in diese Wahlmöglichkeit fällt.

```
%   properties:
 p_empty_del(delete, 3, empty, X) :- member(empty, X).
```

Einige Testrahmen können ausgeschlossen werden, wenn man die sogenannten Selektionsausdrücke mit kritischen Eigenschaften verbindet. Es werden nur die Testrahmen erzeugt, bei denen die Selektionsausdrücke für die beteiligten kritischen Eigenschaften wahr sind. Zusätzlich kann man einer kritischen Eigenschaft noch einen Property-Namen zuordnen, der einen Selektionsausdruck enthält. Property-Namen sind eine Art logische Variable, deren Wert wahr ist, wenn ein Testrahmen die kritische Eigenschaft enthält.

Aus diesen Angaben wird die Testdatenbank generiert. Diese Datenbank enthält später die Informationen, die wir beim Testen und Debugging modifizieren und benutzen. Hier werden automatisch alle möglichen Kombinationen aller Wahlmöglichkeiten aller Kategorien gebildet. In dem Beispielprogramm besitzt *delete* zwei Kategorien mit je drei und vier Wahlmöglichkeiten, also werden daraus die folgenden Kombinationen gebildet:

**Beispiel 22** Die generierten Testrahmen

```
The generated frames for delete/3 are:
  [empty,equal]
  [empty,greater]
  [empty,less]
  [empty,none]
  [more,equal]
  [more,greater]
  [more,less]
  [more,none]
  [one,equal]
  [one,greater]
  [one,less]
  [one,none]
```

Wie bereits erwähnt wurde, sind einige dieser Kombinationen sinnlos. Wenn beispielweise eine Liste leer ist, dann kann man nicht davon sprechen, ob das Element, das gelöscht werden soll, größer als, kleiner als oder gleich dem ersten Element in der leeren Liste ist. Diese Kombinationen werden mit *Bedingungen* ausgefiltert:

```
%  conditions:
 cond( delete, 3, none, X)    :- !, p_empty_del(delete, 3, _, X).
 cond( delete, 3, greater, X) :- !, \+ p_empty_del(delete, 3,_, X).
 cond( delete, 3, less,    X) :- !, \+ p_empty_del(delete, 3, _, X).
 cond( delete, 3, equal,   X) :- !, \+ p_empty_del(delete, 3,_, X).
```

Das Ergebnis der Filterung ist die folgende:

**Beispiel 23** Die selektierten Testrahmen

```
 The screened frames for delete/3 are:
  [empty,none]
  [more,equal]
  [more,greater]
  [more,less]
  [one,equal]
  [one,greater]
  [one,less]
```

Aus dieser Beschreibung wird eine Datenbank generiert, die folgende Struktur hat.

**Beispiel 24** Die generierte Initialdatenbank entspricht Beispiel 20

```
test(delete, 3, [empty,none], [], undefined).
test(delete, 3, [more,equal], [], undefined).
test(delete, 3, [more,greater], [], undefined).
test(delete, 3, [more,less], [], undefined).
test(delete, 3, [one,equal], [], undefined).
test(delete, 3, [one,greater], [], undefined).
test(delete, 3, [one,less], [], undefined).
```

Die Bedeutung ist die folgende (siehe [KHG97b], [KHG97a]): Jede Zeile beginnt mit dem Prädikatnamen `test`. Es folgt der Name und die Stelligkeit eines Prädikats (aus dem Programm). Dann folgt der Testrahmen (nur die selektierten Kombinationen). `[ ]` steht für die leere Liste der bisher aufgetretenen I/O-Paare. Wenn wir testen oder debuggen, dann werden hier nach und nach alle I/O-Paare, die in diesen Testrahmen fallen, eingefügt. Alle Testrahmen haben eine Bewertung: `correct` oder `incorrect` oder am Anfang `undefined`, weil noch nichts über sie bekannt ist. Noch einmal sei wiederholt: `correct` heißt, daß für alle I/O-Paare aus diesem Testrahmen die Ausgabe zu der dazugehörigen Eingabe korrekt ist. Die Bewertung ist `incorrect`, wenn die Ausgabe für die in den Testrahmen fallende Eingabe inkorrekt ist.

## 4.2 Testen eines PROLOG-Programms

Alle Programme müssen getestet werden. Myers definiert die Aufgabe des Testens in seinem Buch ([Mye79]) wie folgt: "Testen ist die Durchführung von Programmen mit der Absicht, Fehler zu finden". Software-Testen ist also ein Teil der Softwareentwicklung und logische Programmierung besonders geeignet, um Tests durchzuführen. Tester für andere Programmiersprachen sind zum Teil auch in PROLOG geschrieben ([SH89]).

Ein Testprozeß muß konsequent Testdaten liefern, muß den Anwender mit Informationen versorgen, nämlich welche Teile des Programms schon getestet wurden, welche Teile besonders wichtig zu testen sind und von welchem vermutet wird, daß sie Fehler enthalten.

Unser *IDTS* führt Tests mit Hilfe der Testdatenbank durch: Das System bietet dem Anwender die Möglichkeit vor dem eigentlichen Debugging, eine Klausel und dazu einen Testrahmen auszuwählen, wobei nur diejenigen Testrahmen getestet werden können, die noch keine Bewertung (*correct* oder *incorrect*) haben. Der Anwender muß ein Beispiel eingeben. Das System überprüft mit Hilfe der Suchfunktionen, ob dieses Beispiel zu dem Testrahmen gehört. Wenn das der Fall ist, wird das Beispiel und auch die vom Benutzer eingegebene Bewertung in die Datenbank eingefügt.

Um das Testverfahren durch ein Beispiel erläutern zu können, definieren wir eine umfangreichere Testbeschreibung des *compute*-Programms als die, die wir vorher verwendet haben. Diese Beschreibung und die generierten Testergebnisse werden später auch im Kapitel 4.3.2 verwendet.

Wir definieren die Testbeschreibung für die Prädikate *member*, *delete* und *complement*. Die für *delete* definierte Testbeschreibung ist anders als bisher.

**Beispiel 25** Testspezifikation: member

**Test specification:** member
**Category:** number_of_elements
choice: empty
choice: one
choice: more
**Category:** relation
choice: equal if not empty
choice: greater if not empty
choice: less if not empty
choice: none if empty

Die Kategorie `number_of_elements` gibt die Länge der Liste an, die als zweiter Parameter im Prädikat vorkommt und in der das im ersten Argument gegebene Element gesucht wird. Die Wahlmöglichkeiten sind *empty*, *one* und *more*. Es wurde eine zweite Kategorie definiert, die das Verhältnis zwischen dem ersten Argument und dem ersten Element der Liste zeigt, wie wir es schon bei der früheren Spezifikation von *delete* (siehe Beispiel 16) beschrieben haben. Die aus dieser Beschreibung generierbaren Testrahmen sind: *(empty, none), (one, equal), (one, greater), (one, less), (more,equal ), (more, greater), (more, less).*

Die folgende Testspezifikation für das Prädikat *delete* wurde gegenüber der früher verwendeten modifiziert. Dies ist notwendig, um später (in Kapitel 4.3.2) zeigen zu können, daß diese Variante inkonsistent ist.

**Beispiel 26** Testspezifikation: delete

**Test specification:** delete
**Category:** number_of_elements
choice: empty
choice: one
choice: more
**Category:** relationship
choice: somewhere if not empty
choice: nowhere if not empty
choice: none if empty

Die Kategorie *number_of_elements* hat drei verschiedene Wahlmöglichkeiten, nämlich die leere Liste (*empty*), die Liste mit einem Element (*one*) und die Liste mit mehreren Elementen (*more*).

Früher beschrieb die Kategorie *relationship* folgendes: Wenn wir ein Element aus der Liste $L$ löschen, überprüfen wir, ob es gleich (*equal*), größer als (*greater*) oder kleiner als (*less*) das erste Element von $L$ ist. Deswegen werden diese Fälle getrennt voneinander getestet.

Jetzt haben wir die Kategorie *relation* neu definiert: *somewhere* bedeutet, daß das erste Argument $e$ in der Liste $L$ vorkommt, aber wir wissen nicht, an welcher Stelle. Die Selektionsausdrücke (*selector expression*) aus dem Beispiel *if not empty* reduzieren die Testrahmen und selektieren ungültige Kombinationen aus. Eine solche liegt zum Beispiel vor, wenn $L$ gleich einer leeren Liste ist (*frame [empty,somewhere]*). Um die leere Liste testen zu können, wird eine zusätzliche Wahlmöglichkeit (*none*) eingeführt, die mit der Wahlmöglichkeit *empty* einen Testrahmen bilden kann.

Die generierten Testrahmen sind die gültigen Kombinationen aller Wahlmöglichkeiten aus den zwei Kategorien: *(empty, none), (one, nowhere), (one, somewhere), (more, nowhere), (more, somewhere).*

**Beispiel 27** Testspezifikation: complement

**Test specification:** complement
**Category:** number_of_elements_of_the_first_par
choice: empty_first if not empty_second
choice: one_first if not empty_second
choice: more_first if not empty_second
choice: none if empty second
**Category:** number_of_elements_of_the_second_par
choice: empty_second if not empty_first
choice: one_second if not empty_first
choice: more_second if not empty_first
choice: none if empty_first

Das Prädikat *complement* hat zwei Eingabeparameter. Beide sind Listen, und das Ergebnis im dritten Argument ist das Komplement dieser Listen. Deswegen haben wir die folgenden Kategorien definiert: `number_of_elements_of_the_first_par`, `number_of_elements_of_the_second_par`. Sie beziehen sich auf die Länge der Listen. Beide Kategorien werden in vier Wahlmöglichkeiten eingeteilt: `empty, one, more, none`. Die Wahlmöglichkeit `none` wird verwendet, wenn eine der beiden Listen kein Element enthält. Die generierten Testrahmen sind die gültigen Kombinationen aller Wahlmöglichkeiten aus den zwei Kategorien: *(empty_first, none),(none, empty_second),(more_first, more_second), (more_first, one_second),(one_first, more_second), (one_first, one_second).*
Aus der oben angegebenen Beschreibung werden dann die folgenden Testrahmen generiert.

**Beispiel 28** Die generierten Testrahmen

```
The screened frames for member/2 are:
  [empty,none]
  [one,equal]
  [one,greater]
```

```
  [one,less]
  [more,equal]
  [more,greater]
  [more,less]


The screened frames for delete/3 are:
  [empty,none]
  [one,nowhere]
  [one,somewhere]
  [more,nowhere]
  [more,somewhere]


The screened frames for complement/3 are:
  [empty_first,none]
  [none,empty_second]
  [one_first,more_second]
  [one_first,one_second]
  [more_first,more_second]
  [more_first,one_second]
```

Aus der Beschreibung wird die folgende Datenbank generiert:

**Beispiel 29** Die generierte Initialdatenbank

```
The list of current test database:
 test(complement, 3, [empty_first,none], [], undefined).
 test(complement, 3, [none,empty_second], [], undefined).
 test(complement, 3, [more_first,more_second], [], undefined).
 test(complement, 3, [more_first,one_second], [], undefined).
 test(complement, 3, [one_first,more_second], [], undefined).
 test(complement, 3, [one_first,one_second], [], undefined).
 test(delete, 3, [empty,none], [], undefined).
 test(delete, 3, [more,nowhere], [], undefined).
 test(delete, 3, [more,somewhere], [], undefined).
 test(delete, 3, [one,nowhere], [], undefined).
 test(delete, 3, [one,somewhere], [], undefined).
 test(member, 2, [empty,none], [], undefined).
 test(member, 2, [more,equal], [], undefined).
 test(member, 2, [more,greater], [], undefined).
 test(member, 2, [more,less], [], undefined).
 test(member, 2, [one,equal], [], undefined).
 test(member, 2, [one,greater], [], undefined).
 test(member, 2, [one,less], [], undefined).
```

Zunächst wählt man den Prädikatnamen aus, den man testen möchte. *IDTS* bietet hierzu eine Liste aller Möglichkeiten an (für die Prädikate, die eine *CPM*-Beschreibung haben):

**Beispiel 30** Liste aller Prädikate mit *CPM*-Beschreibung

```
The predicates that have test database descriptions
    1. Predicate: "complement/3".
    2. Predicate: "delete/3".
    3. Predicate: "member/2".
```

Hier wird das Prädikat `delete` ausgewählt. Als nächstes wird eine Liste zum ausgewählten Prädikat mit allen Testrahmen angezeigt, die in der Testdatenbank mit „undefined" gekennzeichnet sind:

**Beispiel 31** Die Testrahmen für das ausgewählte Prädikat `delete`

```
The frames for  delete/3  are:
    1. Frame: "[empty,none]"
    2. Frame: "[more,nowhere]"
    3. Frame: "[more,somewhere]"
    4. Frame: "[one,nowhere]"
    5. Frame: "[one,somewhere]"
```

Es wird ein Testrahmen ausgewählt. Dann ist eine Beispiel-Klausel einzugeben. Dabei wird automatisch mit Hilfe der Suchfunktionen aus der *CPM*-Beschreibung überprüft, ob das Beispiel auch zum ausgewählten Testrahmen gehört.

**Beispiel 32** Eine Beispiel-Klausel, die in den ausgewählten Testrahmen `[more,somewhere]` fällt:

```
    Enter clause: delete(4,[4,3],X).
```

Es wird das Ergebnis bestimmt. Dann wird gefragt, ob das Ergebnis richtig oder falsch ist:

**Beispiel 33** Die Frage an den Anwender

```
    Is the goal delete(4,[4,3],[3]) correct? (y/n) y
```

Im Beispiel wurde mit *yes* geantwortet. Natürlich kann das Testverfahren mit mehreren Klauseln durchgeführt werden. Wir haben weiterhin auf der Klausel *complement* die zwei Testrahmen `[more_first,more_second]`, `[more_first,one_second]` mit den folgenden Testfällen getestet.

**Beispiel 34** Eine Beispiel-Klausel, die für die Klausel *complement* in den Testrahmen `[more_first,more_second]` fällt:

```
    Is the goal complement([1,2],[1,2],[]) correct? (y/n) y
```

**Beispiel 35** Eine Beispiel-Klausel, die für die Klausel *complement* in den Testrahmen `[more_first,one_second]` fällt:

```
    Is the goal complement([4,3],[4],[3]) correct? (y/n) y
```

In beiden Fällen wird das von *compute* gelieferte Ergebnis als *korrekt* bewertet. Demnach wird die Testdatenbank wie folgt ergänzt:

**Beispiel 36** Änderung der Testdatenbank nach dem Test

```
test(complement,3,[more_first,more_second],[([[1,2],[1,2]],[[]])],correct).
test(complement,3,[more_first,one_second],[([[4,3],[4]],[[3]])],correct).
test(delete,3,[more,somewhere],[([4,[4,3]],[[3]])],correct).
```

Die restlichen Zeilen der Testdatenbank bleiben unverändert.

## 4.3 Die um CPM erweiterten Originalmethoden von Shapiro für Terminierung mit inkorrekter Ausgabe

In diesem Kapitel werden die drei verschiedenen Methoden für die Diagnose der falschen Klauseln im Fall inkorrekter Ausgabe vorgestellt. Bei diesen Methoden wird der Ableitungsbaum, dessen Wurzel $< C, x, y >$ ist, durchsucht, und bei allen Knoten wird entweder eine Zugehörigkeitsfrage gestellt, die vom Anwender beantwortet wird, oder eine Bewertung aus der Testdatenbank genommen. Die Anzahl der Fragen hängt von der Position der inkorrekten Klauselinstanz im Ableitungsbaum und von der Anzahl der Bewertungen in der Testdatenbank ab.

Nehmen wir an, daß eine *CPM*-Testkonfiguration für das Programm **P** existiert, für einige $C \in \mathbf{P}$ wurde eine Menge von Wahlmöglichkeiten definiert, und einige Testrahmen haben eine Bewertung.

Im Debugging-Prozeß wird vorausgesetzt, daß die gegebene *CPM*-Testbeschreibung konsistent ist. Aus dieser Voraussetzung folgt, daß, wenn ein Testrahmen schon bewertet wurde, diese Bewertung unabhängig von dem ausgewählten Testfall ist. Es ist nur eine Hypothese des Benutzers, die auch falsch sein kann. Deshalb wird auch der Fall betrachtet, daß es inkonsistente Testrahmen in der *CPM*-Testbeschreibung von **P** gibt (siehe Kapitel 4.3.2).

Wenn ein Testrahmen nicht bewertet ist (d.h. $\varphi$ ist *undefined* für diesen Testrahmen), wird eine Zugehörigkeitsfrage gestellt, wenn die Parametereinstellungen für das Debugging es erlauben. Man kann als Parameter einstellen:

- *Debugging Points*: Eine Liste von Prädikatnamen, bei denen der Debugger anhalten soll.
- *Debugging Point Action*: Entweder hält der Debugger nur bei Elementen der Liste *Debugging Points* (*skip*), oder er schaltet nach dem ersten gefundenen Element der Liste auf den Einzelschritt-Modus (*step*).

Im folgenden sei $C \in \mathbf{P}$ eine Klausel und wir nehmen an, daß $< C, x, y > \notin \mathcal{I}$ für beliebige $x$ und $y$ und der vollständige Ableitungsbaum mit der Wurzel $< C, x, y >$ endlich ist.

### 4.3.1 Die modifizierte Bottom-Up-Methode

Wie die Originalmethode von Shapiro besucht unser Algorithmus den Ableitungsbaum in Postorder-Weise. Nehmen wir an, daß wir weiterhin die Frage, ob $< C_j, x_j, y_j > \in \mathcal{I}$ ist oder nicht, beantworten müssen, wobei $< C_j, x_j, y_j >$ ein Knoten im Ableitungsbaum ist. Im Fall *CPM* überprüfen wir, ob der Testrahmen zu $C_j$ und $x_j$ schon existiert und ob er schon eine Bewertung hat: $\varphi_{C_j}(\chi_{C_j}(x_j))$ ist nicht *undefined*. Wenn dieser Testrahmen schon bewertet wurde, dann wird darauf verzichtet, den Anwender zu fragen, und stattdessen das Ergebnis der Bewertung als Antwort der Zugehörigkeitsfrage benutzt und das neue Paar $(x_j, y_j)$ in die Datenbank eingefügt. Wenn der Testrahmen existiert, aber $\varphi_{C_j}(\chi_{C_j}(x_j)) = \mathit{undefined}$ und die Parametereinstellung es erlaubt, dann wird die Frage gestellt. Die Antwort des Anwenders und $(x_j, y_j)$ werden in die Datenbank eingefügt. Wenn dieser Rahmen das nächstes Mal identifiziert wird, wird die jetzt gegebene Bewertung verwendet. Wenn das System den Testrahmen $f$ wegen fehlender Suchfunktionen nicht eindeutig identifizieren kann, also $\chi_{C_j}(x_j)$ nicht entscheidbar ist, dann wird der Anwender gebeten, diese Auswahl über ein Menü zu treffen.

**Algorithmus 15** Shapiro + *CPM* im *Bottom-Up*-Fall

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, wo der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei $modus = step$ in jedem Schritt eine Frage stellt und
$modus = skip$ nur bei Elementen der *debug_points* Liste eine Frage stellt
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel
ein Wahrheitswert für *found*,
wobei *found* =*true*, wenn $C_{buggy}$ gefunden wurde, sonst *found* =*false*,
eine modifizierte Datenbank $\mathbf{D}$,
modifizierter *modus*

**procedure** $CPM\_bottom\_up(< C, x, y >, \mathbf{D}, debug\_points, modus, C_{buggy}, found)$
**begin**
  let $\{< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, \ldots, < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$;
  $C_{buggy}$:=$\varepsilon$; $i$:=1; *found*:=false
  **while** $(i \leq n)$ **and** (not found) **do**
    $CPM\_bottom\_up(< C_i, x_i, y_i >, \mathbf{D}, debug\_points, modus, C_{buggy}, found)$;
    $i := i + 1$
  **endwhile**
  **if** not found **then**
    **if** $\neg\Sigma_C$ **then** answer := *Query* $(< C, x, y >)$
    **else** answer := *Query_database*$(< C, x, y >, \mathbf{D}$, debug_points, modus)
    **endif**
    **if** answer = false **then**
      found := true;
      $C_{buggy} := (< C, x, y > \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >)$
    **endif**
    **if** answer = cancel **then**
      found := true
    **endif**
  **endif**
**end**

Nehmen wir an, daß $< C, x, y > \notin \mathcal{I}$ für ein $C \in \mathbf{P}, x \in D^n$ und $y \in D^m$ ist. Der Algorithmus 15 nimmt das Tripel $< C, x, y >$ als Eingabe und gibt einen booleschen Wert *found*, eine Instanz der fehlerhaften Klausel $C_{buggy}$ und die modifizierte Datenbank $\mathbf{D}$ als Ausgabe zurück. Der Wert von *found* zeigt uns an, ob die Prozedur die fehlerhafte Klauselinstanz gefunden hat oder nicht. Wenn der Wert *true* ist, dann enthält $C_{buggy}$ den Fehler. Weiterhin nehmen wir an, daß $\{< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, \ldots, < C_n, x_n, y_n >\}$ die direkten Nachkommen von $< C, x, y >$ (top level trace) sind. Der Algorithmus ruft sich solange rekursiv selbst mit den Elementen des *top level trace* auf, bis er entweder die falsche Klauselinstanz findet $(found = true)$ oder ein Blatt des Baums erreicht $(i = n)$. Wenn $(found = false)$ ist und für den aktuellen Prädikatnamen der Klausel $C$ keine Testspezifikation $\Sigma_C$ eingegeben wurde, dann wird $Query(< C, x, y >)$ gefragt (siehe Algorithmus 4). Abhängig von der vom Anwender eingegebenen Antwort wird der Algorithmus weitergeführt ($answer = false$) oder die falsche Klauselinstanz wird gefunden ($answer = true$). Wenn aber die Testspezifikation $\Sigma_C$ existiert, dann wird eine erweiterte *Query*-Funktion, *Query_database* (siehe Algorithmus 17) aufgerufen, die die Testdatenbank verwendet. Die vom Algorithmus gestellten Fragen muß der Benutzer immer korrekt beantworten.

Eine nicht korrekt beantwortete Frage kann den Debuggingprozeß auf einen falschen Weg führen, weil der Algorithmus die Besuchsreihenfolge der Baumknoten anhand der vom Anwender gegebenen *yes/no* Antworten steuert. Der Algorithmus bietet die Möglichkeit, statt nur *yes/no* Antworten anzugeben, den Debuggingprozeß mit *cancel* zu beenden. Diese Eigenschaft ist sehr nützlich, wenn der Benutzer später bemerkt, daß eine früher gestellte Frage von ihm nicht richtig verifiziert wurde. In diesem Fall ist es sinnlos, die Suche nach dem Fehler weiterzuführen, und das Debugging muß neu gestartet werden. Natürlich bedeutet das auch, daß die bisherigen Einträge in $T(F(\sum_{\mathbf{P}}))$ gelöscht werden müssen. Dafür sorgt der Algorithmus *IDTS_Debug_CPM*, der *CPM_bottom_up* aufruft und danach den Wert von *found* überprüft. Wenn *found true* ist, aber $C_{buggy}$ den Wert $\varepsilon$ behalten hat, dann bedeutet es, daß als Antwort *cancel* gegeben wurde. In diesem Fall muß der Zustand von $T(F(\sum_{\mathbf{P}}))$ vor Beginn des Debuggings wieder hergestellt werden. Wenn der Wert von *found false* bleibt, bedeutet es, daß der *CPM_bottom_up*-Algorithmus keine falsche Klauselinstanz finden konnte. In diesem Fall ist die Testspezifikation inkorrekt und der Debuggingprozeß kann mit der Originalmethode von Shapiro wiederholt werden.

**Algorithmus 16** Der Hauptalgorithmus des *Bottom-Up*-Verfahrens, wenn die Testdatenbank verwendet wird

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei $modus = step$ in jedem Schritt eine Frage stellt und
$modus = skip$ nur bei Elementen der *debug_points* Liste eine Frage stellt

**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel

procedure IDTS_Debug_CPM($< C, x, y >, \mathbf{D}, C_{buggy}$, debug_points, modus, found)
**begin**
  $\mathbf{D} = T(F(\sum_{\mathbf{P}}))$;
  save($T(F(\sum_{\mathbf{P}}))$);
  *CPM_bottom_up*($< C, x, y >, \mathbf{D}, C_{buggy}$, debug_points, modus, found)
  **if** (found and $C_{buggy} = \varepsilon$) **then**
    restore($T(F(\sum_{\mathbf{P}}))$)
  **endif**
  **if** (not found) **then**
    bottom_up($< C, x, y >$, $C_{buggy}$, found)
  **endif**
**end**

**Algorithmus 17** Die Fragestellung an den Anwender, modifizierte Version des Algorithmus 4

**Eingabe:** $< C, x, y >$ ein Atom,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei *modus* = *step* in jedem Schritt eine Frage stellt
*modus* = *skip* nur bei Elementen der Liste *debug_points* eine Frage stellt und
**Ausgabe:** Die Antwort *oracle_answer*
*true* wenn $< C, x, y > \in \mathcal{I}$
*false* wenn $< C, x, y > \notin \mathcal{I}$,
die modifizierte Datenbank $\mathbf{D}$,
der eventuell veränderte *modus*

**function** $Query\_database(< C, x, y >, \mathbf{D}$, debug_points, modus): oracle_answer
**begin**
  p := predicate_name(C)
  **if** (p $\notin$ debug_points and modus $\neq$ *trace*) **then**
    oracle_answer := yes
  **else**
    **if** modus = *step* **then** modus = *trace*
    **endif**
    **if** $\exists!(p, a, f, i, o, e) \in \mathbf{D}$ **then**
      oracle_answer := $handle\_database((p, a, f, i, o, e), \mathbf{D})$
    **else**
      **if** $\neg\exists\chi_C(x) = f$ **then**
        frame := {}
        **for** $j := 1$ **to** $a$
          write $(\delta_{Cj}, c_{Cj1}, \ldots, c_{Cjn})$;
          read $(c_{Cjk})$ $1 \leq k \leq n$;
          frame := frame $\cup\{c_{Cjk}\}$
        **endfor**
        oracle_answer := $handle\_database((p, a, frame, i, o, e), \mathbf{D})$
      **endif**
    **endif**
    **return**(oracle_answer)
  **endif**
**end**

Die Funktion *Query_database* wird mit dem Tripel $< C, x, y >$ als Eingabe initialisiert und liefert als Ausgabe den Wert *yes, no* oder *cancel*. *Query_database* unterscheidet sich von dem Original darin, daß sie auf die *CPM*-Datenbank zugreift. In diesem Algorithmus wird ein dritter, interner Wert *trace* für die Variable *modus* eingeführt, damit der Algorithmus sich im *modus step* merken kann, ob bereits ein Debuggingpoint passiert wurde (*trace*) oder nicht (*step*).

Nehmen wir an, daß der Algorithmus einen Knoten erreicht, an dem ein Atom $< C, x, y >$

steht, und hier eine Frage gestellt werden muß. Dann können die folgenden Fälle eintreten:

- Der Prädikatname kommt nicht unter den verdächtigen und deswegen zu überprüfenden Prädikaten vor und *modus* ist nicht auf *trace* gestellt. In diesem Fall stellt der Algorithmus dem Anwender keine Frage, sondern gibt automatisch *yes* zurück.

- Die Funktion hat das erste Mal ein Element aus der Liste *debug_points* gefunden. Dann wird *modus* auf *trace* gestellt, wenn er bislang *step* war: Von diesem Knoten an werden in allen Folgeknoten Fragen gestellt, unabhängig davon, ob das Prädikat in der Liste steht oder nicht.

- Die Funktion hat das erste Mal ein Element aus der Liste *debug_points* gefunden. Wenn *modus skip* ist, dann behält *modus* den gleichen Wert, und auch weiterhin werden nur bei Elementen aus der Liste Fragen gestellt.

- Es wird versucht, einen Testrahmen in der Testdatenbank **D** zu identifizieren. Wenn dies gelingt, wird die Funktion *handle_database* aufgerufen, die abhängig von der Bewertung in der Datenbank weiterarbeitet.

- Wenn *Query_database* zum Beispiel wegen fehlenden Suchfunktionen ($\chi_C(x)$) keinen Testrahmen eindeutig bestimmen kann, dann ist es die Aufgabe des Anwenders, diesen zusammenzustellen. Die Vorgehensweise ist die folgende: Für alle Kategorien $\delta_{Cj}$ des Prädikats der Klausel $C$ werden alle Wahlmöglichkeiten $c_{Cj1}, \ldots, c_{Cjn}$ aufgelistet. Der Benutzer wählt eine Wahlmöglichkeit pro Kategorie aus. Das System führt eine zusätzliche Überprüfung durch, ob das Atom wirklich in den vom Benutzer zusammengestellten Testrahmen fällt. Wenn dies nicht der Fall ist, muß der Auswahlprozeß wiederholt werden.

- Auf der Basis des so ermittelten Testrahmens und seiner Bewertung wird weitergearbeitet mit dem Aufruf der Funktion *handle_database*.

**Algorithmus 18** Veränderung der Datenbank entsprechend der Bewertung des Testrahmens

**Eingabe:** $(p, a, f, i, o, e)$ ein Element aus $\mathbf{D}$,
eine Datenbank $\mathbf{D}$
**Ausgabe:** Die Antwort *oracle_answer*
*true*, wenn $< C, x, y > \in \mathcal{I}$,
*false*, wenn $< C, x, y > \notin \mathcal{I}$
die modifizierte Datenbank $\mathbf{D}$
**function** *handle_database*$((p, a, f, i, o, e), \mathbf{D})$: oracle_answer
**begin**
  **if** $e$ = undefined **then**
    oracle-answer := "Is the goal $< p, x, y >$ *correct*? (y/n)"
    new_i := $i \cup x$; new_o := $o \cup y$
    **if** oracle_answer = false **then**
      $\mathbf{D} = \mathbf{D} \setminus \{(p, a, f, i, o, e)\} \cup \{(p, a, f,$ new_i, new_o, incorrect $)\}$
    **endif**
    **if** oracle_answer = true **then**
      $\mathbf{D} = \mathbf{D} \setminus \{(p, a, f, i, o, e)\} \cup \{(p, a, f,$ new_i, new_o, correct $)\}$
    **endif**
  **else**
    **if** $e$ = correct **then** oracle_answer := yes; new_i := $i \cup x$; new_o := $o \cup y$
    **else** oracle_answer := false; new_i := $i \cup x$; new_o := $o \cup y$
    **endif**
    $\mathbf{D} = \mathbf{D} \setminus \{(p, a, f, i, o, e)\} \cup \{(p, a, f,$ new_i, new_o, e $)\}$
  **endif**
**end**

Wenn ein Testrahmen $(p, a, f, i, o, e)$ ermittelt wurde, dann überprüfen wir die Bewertung $e$. Wenn dieser Testrahmen bisher $\varphi_p(\chi_p(x)) = \mathit{undefined}$ ist, dann wird der Anwender gefragt. Ist $\varphi_p(\chi_p(x)) \neq \mathit{undefined}$, dann wird keine Zugehörigkeitsfrage gestellt, diese wird mit der Bewertung des Testrahmens beantwortet. Wenn $\varphi_p(\chi_p(x)) = \mathit{false}$ ist, dann hält die Funktion *handle_database* den Algorithmus an, und es wird die entsprechende Instanz der Klausel $C$ in der Variablen $C_{buggy}$ zurückgegeben. In allen Fällen werden die aktuellen Ein- und Ausgabeparameterbelegungen in der Datenbank gespeichert.

Nehmen wir an, daß $< C, x, y > \notin \mathcal{I}$ ist. Sei $N = | < C, x, y > |$ die Anzahl der Knoten des vollständigen Ableitungsbaums, dessen Wurzel $< C, x, y >$ ist. Dann ist $0 \leq n \leq N$, wobei $n$ die Anzahl der Fragen ist, die vom Benutzer beantwortet werden müssen.

Im besten Fall findet der Algorithmus die falsche Klausel genau so, daß überhaupt keine Frage gestellt wird, weil die Antworten für alle Fragen in der Testdatenbank vorkommen. Im schlimmsten Fall kann der Algorithmus entweder keine falsche Klausel finden, weil inkonsistente Testrahmen definiert wurden, oder die Testdatenbank kann überhaupt nicht verwendet werden. Im Fall inkonsistenter Testrahmen kann natürlich auch die Original-Shapiro-Methode aufgerufen werden.

**Beispiel 37** Die bei der *Bottom-Up*-Methode gestellten Fragen, wenn eine Datenbank benutzt wird

```
Is the goal [sum([],0)] correct? (y/n) y
```

```
 Is the goal [sum([4],4)] correct? (y/n) y
 Is the goal [sum([3,4],7)] correct? (y/n) y
 Is the goal [sum([2,3,4],9)] correct? (y/n) y
 Is the goal [sum([1,2,3,4],10)] correct? (y/n) y
 Is the goal [sum([2,4],6)] correct? (y/n) y
 Is the goal [sum([1,2,4],7)] correct? (y/n) y
 Is the goal [sigma([1,2,3,4],[1,2,4],17)] correct? (y/n) y
 Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n) y
 Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n) y
 Is the goal [member( 4 , [4] )] correct? (y/n) y
 Is the goal [member( 4 , [3,4] )] correct? (y/n) y
 Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
 Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n

The false clause:
delete(4 , [3,4] , [4]) :- delete(4 , [4] , [])
```

Dabei werden in der Datenbank die folgenden Modifikationen durchgeführt:

**Beispiel 38** Die Änderungen in der Testdatenbank im *Bottom-Up*-Fall

```
test(member,2,[more,equal],[([2,[2,3,4]],[2,[2,3,4]]),([1,[1,2,3,4]],
     [1,[1,2,3,4]])],correct).
test(delete,3,[more,equal],[([2,[2,3,4],[3,4]],[2,[2,3,4],[3,4]]),
    ([1,[1,2,3,4],[2,3,4]],[1,[1,2,3,4],[2,3,4]])],correct).
test(member,2,[one,equal],[([4,[4]],[4,[4]])],correct).
test(member,2,[more,greater],[([4,[3,4]],[4,[3,4]])],correct).
test(delete,3,[one,equal],[([4,[4],[]],[4,[4],[]])],correct).
test(delete,3,[more,greater],[([4,[3,4],[4]],[4,[3,4],[4]])],incorrect).
```

Die gestellten Fragen können auch grafisch dargestellt werden (siehe Abbildung 4.1).

### 4.3.2 Debugging mit inkonsistenter Testspezifikation

Weil unsere Hypothese über die Konsistenz der Testrahmen nicht bewiesen wird, müßte das von unserem Algorithmus ermittelte Ergebnis verifiziert werden. Einerseits ist es möglich, daß wir die falsche Klausel nicht finden. Andererseits, wenn die inkorrekte Klausel vom *IDTS*-Algorithmus identifiziert wurde, kann es vorkommen, daß der zum Kopf der Klausel gehörende Testrahmen *inkonsistent* ist oder daß es im Körper der Klauselinstanz ein Atom gibt, dessen Testrahmen inkonsistent ist. Der zweite Fall bedeutet, daß die gefundene Klausel nicht die tiefste inkorrekte Klausel im Baum ist. In beiden Fällen liegt es im Ermessen des Benutzers, den Debuggingvorgang mit den Algorithmen von Shapiro zu wiederholen. Nur dann kann die vermutete Inkonsistenz bestimmt und aufgelöst werden. Wir speichern alle Fragen und Antworten in der Testdatenbank. Deshalb ist die Anzahl der von dem Benutzer zu beantwortenden Fragen kleiner oder gleich (im schlimmsten Fall) als bei der Original-Methode. In dem Fall der konsistenten Testrahmen kann die Anzahl der Fragen erheblich reduziert werden. Wenn wir beim Debugging einen inkonsistenten Testrahmen finden, dann muß der Benutzer die *CPM*-Konfiguration verändern. Genauer gesagt, der Benutzer muß eine Verfeinerung der *CPM*-Konfiguration definieren, so daß die neu erstellte Konfiguration bezüglich der bekannten Testfälle konsistent wird.

Zuerst werden Verfeinerungen für Äquivalenzrelationen und dann, als Erweiterung, für *CPM*-Testkonfigurationen formal definiert.

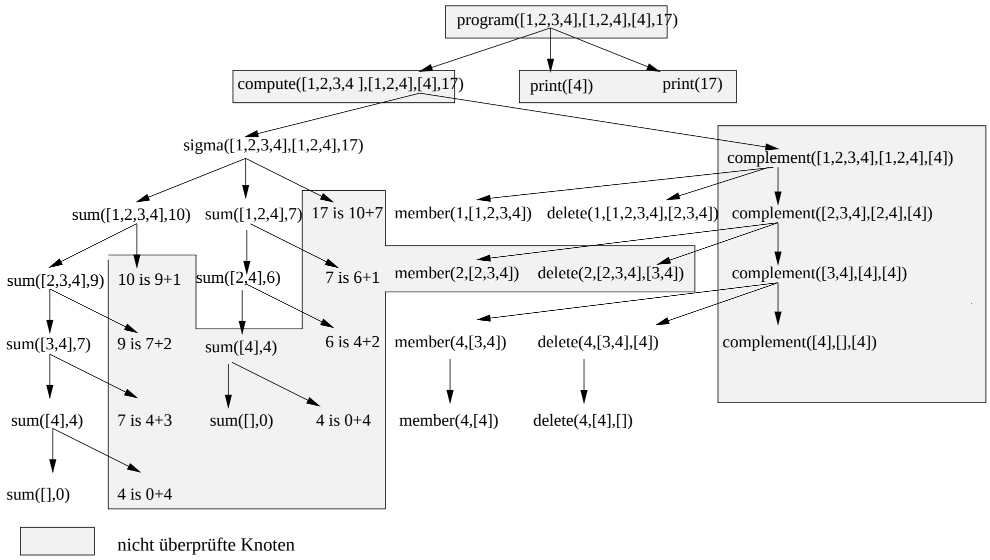



Abbildung 4.1: Die von der *CPM-Bottom-Up*-Methode gestellten Fragen für das *compute*-Beispiel

**Definition 64** Seien $\delta$ eine Äquivalenzrelation über $D^n$ und $x$ ein beliebiges Element aus dieser Menge und $\delta[x]$ bezeichnet die Klasse von $\delta$, die $x$ enthält. Seien $\delta_1$ und $\delta_2$ Äquivalenzrelationen über $D^n$. Wir sagen, daß $\delta_2 \geq \delta_1$ genau dann, wenn aus $\delta_2[a] = \delta_2[b]$ folgt, daß $\delta_1[a] = \delta_1[b]$ für beliebige $a, b \in D^n$.

**Definition 65** Sei $F(\Sigma_{C_i})$ und $F(\Sigma'_{C_i})$ zwei Mengen von Testrahmen einer beliebigen Klausel $C_i$ in **P**. Wir sagen, daß die *CPM*-Testkonfiguration $T(F(\Sigma'_{C_i}))$ der Klausel $C_i$ genau dann eine *Verfeinerung* (*refinement*) von $T(F(\Sigma_{C_i}))$ ist, wenn $F(\Sigma'_{C_i}) \geq F(\Sigma_{C_i})$ (also $\forall \pi \in F(\Sigma_{C_i})$ $\exists \varphi \in F(\Sigma'_{C_i})$, so daß $\varphi \geq \pi$) ist.

Die obigen zwei Definitionen können auf Programme **P** verallgemeinert werden.

**Definition 66** Für ein Programm **P** ist eine *CPM*-Testkonfiguration $T(F(\Sigma'_{\mathbf{P}}))$ von **P** genau dann eine Verfeinerung von $T(F(\Sigma_{\mathbf{P}}))$, wenn $T(F(\Sigma'_{C_i}))$ eine Verfeinerung von $T(F(\Sigma_{C_i}))$ für alle $C_i \in \mathbf{P}$ ist.

Wenn $T_1$ und $T_2$ zwei *CPM*-Testkonfigurationen desselben Programms sind und $T_2$ eine Verfeinerung von $T_1$ ist, dann ist die Anzahl der inkonsistenten Testrahmen von $T_2$ kleiner oder gleich der von $T_1$.

Jetzt demonstrieren wir die Verfeinerung der *CPM*-Konfiguration. Die Verfeinerung ist notwendig, wenn *IDTS* erkennt, daß ein inkonsistenter Testrahmen definiert wurde. Das bedeutet, daß ein Testrahmen sowohl einige Testfälle mit korrektem Verhalten als auch solche mit inkorrektem Verhalten enthält. Die Aufgabe ist, die inkonsistenten Testrahmen zu erkennen, aus der Testdatenbank zu entfernen und die Testbeschreibung so zu modifizieren, daß diese sich unterschiedlich verhaltenden Testfälle zu verschiedenen Testrahmen gehören. Nehmen wir wieder das *compute*-Beispiel mit dem gleichen Fehler, aber mit der in Kapitel 4.2 beschriebenen Testspezifikation.

**Beispiel 39** Das Aufrufen des *compute*-Beispiels mit dem Ziel:

```
complement([1,2,3,4],[1,2,4],X)
```

und die Suche nach dem Fehler, wenn die Testdatenbank verwendet wird.

```
Debugging points: complement.
Debugging point [step, skip] : step
Enter predicate: complement([1,2,3,4],[1,2,4],X).

 Is the goal  complement([1,2,3,4],[1,2,4],[4]) correct? (y/n) n

 Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n) y
 Is the goal [member( 4 , [4] )] correct? (y/n) y
 Is the goal [member( 4 , [3,4] )] correct? (y/n) y
 Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
 Is the goal [complement( [4] , [] , [4] )] correct? (y/n) y
No errors were found.
```

Das *IDTS*-System findet wegen der inkonsistenten Testbeschreibung keinen Fehler. Das fehlerhafte Prädikat `delete( 4 , [3,4] , [4] )` wurde mit *correct* bewertet, weil es in den gleichen Testrahmen fällt wie das Prädikat `delete(4,[4,3],[3])` und dieses hat der Anwender beim Testen mit *yes* bewertet. Um den inkonsistenten Testrahmen zu finden, wird das Original-*Top-Down*-Verfahren aufgerufen.

**Beispiel 40** Aufruf des Original-*Top-Down*-Verfahrens, um den inkonsistenten Testrahmen zu finden

```
Do you want to proceed with Shapiro? (y/n) y
 Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n) y
 Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n) y
 Is the goal [member( 2 , [2,3,4] )] correct? (y/n) y
 Is the goal [delete( 2 , [2,3,4] , [3,4] )] correct? (y/n) y
 Is the goal [member( 4 , [4] )] correct? (y/n) y
 Is the goal [member( 4 , [3,4] )] correct? (y/n) y
 Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
 Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
 Inconsistent test frame: [[more,somewhere]]
```

Wie das Beispiel 40 zeigt, ist der generierte Testrahmen `[more,somewhere]` inkonsistent. Daraufhin werden in der Testdatenbank die folgenden Änderungen durchgeführt.

**Beispiel 41** Während des Debuggens in der Testdatenbank durchgeführte Änderungen

```
test(member, 2, [one,equal_member], [([4,[4]],[4,[4]])], correct).
test(member, 2, [more,greater], [([4,[3,4]],[4,[3,4]])], correct).
test(delete, 3, [one,somewhere], [([4,[4],[]],[4,[4],[]])], correct).
test(delete, 3, [more,somewhere],[([4,[3,4],[4]],[4,[3,4],[4]]),([2,[2,3,4],[3,4]],
     [2,[2,3,4],[3,4]]),([1,[1,2,3,4],[2,3,4]],[1,[1,2,3,4],[2,3,4]]),([4,[4,3],[3]],
     [4,[4,3],[3]])], correct).
test(complement, 3, [one_first,empty_second], [([[4],[],[4]],[[4],[],[4]])], correct).
test(complement, 3, [more_first,one_second], [([[3,4],[4],[4]],[[3,4],[4],[4]]),
    ([[4,3],[4],[3]],[[4,3],[4],[3]])], correct).
```

```
test(complement, 3, [more_first,more_second], [([[1,2,3,4],[1,2,4],[4]],[[1,2,3,4],
    [1,2,4],[4]]), ([[2,3,4],[2,4],[4]],[[2,3,4],[2,4],[4]]),([[1,2],[1,2],[]],[[1,2],
    [1,2],[]])], correct).
```

### 4.3.3 Die modifizierte Top-Down-Methode

Die Vorgehensweise ist ähnlich wie wir sie bei der Kombination von *CPM* mit dem *Bottom-Up*-Fall beschrieben haben, nur wird hier der Ableitungsbaum *Top-Down* durchlaufen. Wir benutzen eine modifizierte Version des Algorithmus 5.

**Algorithmus 19** Shapiro + *CPM* im *Top-Down*-Fall

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei $modus = step$ in jedem Schritt eine Frage stellt und
$modus = skip$ nur bei Elementen der Liste *debug_points* eine Frage stellt

**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
ein Wahrheitswert $found$,
wobei $found$ =*true*, wenn $C_{buggy}$ gefunden wurde, sonst $found$ =*false*,
eine modifizierte Datenbank $\mathbf{D}$,
modifizierter *modus*

**procedure** $CPM\_top\_down(< C, x, y >, \mathbf{D}, debug\_points, modus, C_{buggy}, found)$
**begin**
  let $\{< C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$;
  $C_{buggy} := \varepsilon$; $i$:=1; found:=false
  **while** $(i \leq n)$ **and** (not found) **do**
    **if** $\neg\Sigma_C$ **then** answer := $Query(< C_i, x_i, y_i >)$
    **else** answer := $Query\_database(< C_i, x_i, y_i >, \mathbf{D}$, debug_points, modus)
    **endif**
    **if** answer = false **then**
    $CPM\_top\_down(< C_i, x_i, y_i >, \mathbf{D}, debug\_points, modus, C_{buggy}, found)$
    **else**
      **if** answer = true **then** $i := i + 1$
      **else** found := true
      **endif**
    **endif**
  **endwhile**
  **if** not found **then**
    $C_{buggy} := (< C, x, y > \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >)$;
    found := true
  **endif**
**end**

Der Algorithmus 19 arbeitet so, wie wir es schon im Kapitel 3.1.2 beschrieben haben: Es wird bei allen Knoten versucht, den falschen Kindknoten zu finden. Allerdings wird hier dem Benutzer nicht jedesmal eine Frage gestellt, sondern nur, wenn die Antwort auf die Frage noch nicht in der Testdatenbank enthalten ist oder es keine Testbeschreibung zum Prädikat gibt.

Der Hauptalgorithmus ist fast genauso wie Algorithmus 16 aufgebaut. Es wird lediglich statt *CPM_bottom_up* der Algorithmus *CPM_top_down* aufgerufen.

Kommen wir nun wieder zu unserem Standardbeispiel *compute*. Im folgenden zeigen wir die gestellten Fragen und die durchgeführten Änderungen in der Testdatenbank. Der Algorithmus stellt nur neun Fragen, das sind fünf weniger als bei der *Bottom-Up*-Methode. Wie bei der *Bottom-Up*-Methode hängt die Anzahl der Fragen stark von der Vollständigkeit der Testdatenbank ab.

**Beispiel 42** Die bei der *Top-Down*-Methode gestellten Fragen, wenn die Datenbank auch benutzt wird:

```
 Is the goal [compute([1,2,3,4],[1,2,4],[4],17)] correct? (y/n) n
 Is the goal [sigma([1,2,3,4],[1,2,4],17)] correct? (y/n) y
 Is the goal [complement( [1,2,3,4] , [1,2,4] , [4] )] correct? (y/n) n
 Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n) y
 Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n) y
 Is the goal [complement( [3,4] , [4] , [4] )] correct? (y/n) n
 Is the goal [member( 4 , [3,4] )] correct? (y/n) y
 Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
 Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y

The false clause:
delete(4 , [3,4] , [4]) :- delete(4 , [4] , [])
```

Die Vorgehensweise des Algorithmus kann man auch grafisch mit Hilfe der Abbildung 4.2 veranschaulichen.
Die in der Testdatenbank durchgeführten Änderungen zeigt das Beispiel 43.

**Beispiel 43** Während des Debuggens in der Testdatenbank durchgeführte Änderungen

```
test(complement,3,[more_first,more_second],[([[2,3,4],[2,4],[4]],[[2,3,4],[2,4],[4]]),
    ([[1,2,3,4],[1,2,4],[4]],[[1,2,3,4],[1,2,4],[4]])],incorrect).
test(member,2,[more,equal],[([2,[2,3,4]],[2,[2,3,4]]),([1,[1,2,3,4]],
     [1,[1,2,3,4]])],correct).
test(delete,3,[more,equal],[([2,[2,3,4],[3,4]],[2,[2,3,4],[3,4]]),([1,[1,2,3,4],
     [2,3,4]],[1,[1,2,3,4],[2,3,4]])],correct).
test(complement,3,[more_first,one_second],[([[3,4],[4],[4]],[[3,4],[4],[4]])],incorrect).
test(member,2,[more,greater],[([4,[3,4]],[4,[3,4]])],correct).
test(delete,3,[more,greater],[([4,[3,4],[4]],[4,[3,4],[4]])],incorrect).
test(delete,3,[one,equal],[([4,[4],[]],[4,[4],[]])],correct).
```

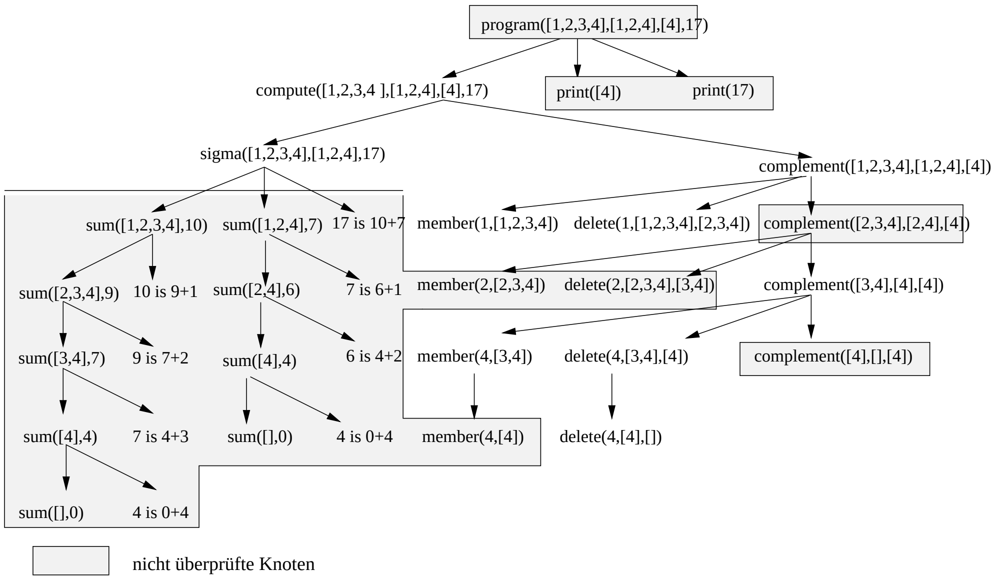



Abbildung 4.2: Die von der *CPM-Top-Down*-Methode gestellten Fragen für das *compute*-Beispiel

### 4.3.4 Die modifizierte Divide-and-Query-Methode

Die dritte Möglichkeit ist, die *CPM*-Methode mit dem Shapiros *Divide-and-Query*-Verfahren zu kombinieren. Wir verändern zuerst den beim *Divide-and-Query* verwendeten Algorithmus 9.

---

**Algorithmus 20** Shapiro + *CPM* im *Divide-and-Query*-Fall

---

**Eingabe:** eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
$\mathcal{I}' \subset \mathcal{I}$, wobei $\mathcal{I}'$ leer sein kann,
das Gewicht des zu $C$ gehörenden Ableitungsbaums *weight*,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei $modus = step$ in jedem Schritt eine Frage stellt und
$modus = skip$ nur bei Elementen der *debug_points* Liste eine Frage stellt
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
eine modifizierte Datenbank $\mathbf{D}$,
modifizierter *modus*

**procedure** $CPM\_divide\_query(< C, x, y >, \mathcal{I}'$, weight, $C_{buggy}$, $\mathbf{D}$, debug_points, modus)
**begin**
  middle_weight = 0; $C_{middle}{:=}\varepsilon$
  **while** weight $\neq 1$ **do**
    $middle\_point(< C, x, y >, C_{middle}$, middle_weight,weight)
    **if** $\neg\Sigma_C$ **then** answer := Query $(< C, x, y >)$
    **else** answer := $Query\_database(< C, x, y >, \mathbf{D}$, debug_points, modus)
    **endif**
    **if** answer = yes **then**
      $CPM\_divide\_query(< C, x, y >, \mathcal{I}' \cup C_{middle}$,
          weight−middle_weight,$C_{buggy}$, $\mathbf{D}$, debug_points, modus)
    **else**
      **if** answer = no **then**
        $CPM\_divide\_query(C_{middle}, \mathcal{I}'$, middle_weight,$C_{buggy}$, $\mathbf{D}$, debug_points, modus)
      **else**
      $C_{buggy}{:=}\varepsilon$ **return**
      **endif**
    **endif**
  **endwhile**
  $C_{buggy} := C_{middle} \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >,$
**end**

---

Der Algorithmus 20 arbeitet so, wie wir es schon beim Algorithmus 9 beschrieben haben: Er versucht, rekursiv die falsche Klausel zu finden, wobei er das schon vorher für den ganzen Baum berechnete Gewicht *weight* verwendet. Mit Hilfe dieses Werts wird das Gewicht des mittleren Knotens und der übrigen Knoten bestimmt. Der Unterschied zwischen den zwei

Algorithmen liegt darin, daß Algorithmus 9 für den mittleren Knoten dem Anwender immer eine Frage stellt, während Algorithmus 20 zuerst überprüft, ob eine Testbeschreibung des in diesem Knoten stehenden Prädikates eingegeben wurde. Wenn das der Fall ist, dann versucht er, die Antwort mit Hilfe der Prozedur *Query_database* in der Testdatenbank zu finden. Wenn die vom Benutzer eingegebene Antwort *yes* lautet oder in der Testdatenbank eine Bewertung *correct* gefunden wurde, dann liegt der Fehler nicht im Teilbaum dieses Knotens. Deswegen wird die Prozedur *CPM_divide_query* rekursiv mit der gleichen Wurzel, aber mit der Differenz des Original-Gewichts und des mittleren Gewichts als Gewicht *weight* aufgerufen. Andernfalls liegt der Fehler in dem Teilbaum, dessen Wurzel der gerade gefragte Knoten ist. Deswegen wird die Prozedur *CPM_divide_query* beim rekursiven Aufruf mit diesem Knoten als Wurzel und mit *middle_weight* als Gewicht aufgerufen.

Die Kombination der *CPM* mit der *Divide-and-Query*-Methode zeigt der Hauptalgorithmus 21. Der Algorithmus erledigt zwei wichtige Aufgaben:

- Ruft die Prozedur *compute_weight* auf, die das Gewicht des ursprünglichen Ableitungsbaums *weight* liefert. Damit arbeitet dann das Verfahren *CPM_divide_query*.
- Behandelt den Fall, wenn als Antwort *cancel* eingegeben wurde. In diesem Fall wird der Debuggingprozeß beendet, die einzelnen Einträge, die während des Debuggings eingetragen wurden, werden wieder gelöscht.

---

**Algorithmus 21** Der Hauptalgorithmus des *Divide-and-Query*-Verfahrens, wenn die Testdatenbank verwendet wird

---

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
$\mathcal{I}' \subset \mathcal{I}$,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei $modus = step$ in jedem Schritt eine Frage stellt und
$modus = skip$ nur bei Elementen der *debug_points* Liste eine Frage stellt

**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
eine Datenbank $\mathbf{D}$

**procedure** $IDTS\_Debug\_divide\_and\_query(< C, x, y >, C_{buggy}, \mathcal{I}', \mathbf{D}, debug\_points, modus)$
**begin**
  $compute\_weight(< C, x, y >, \mathcal{I}'$, weight);
  $\mathbf{D} = T(F(\sum_{\mathbf{P}}))$;
  save($T(F(\sum_{\mathbf{P}}))$);
  $CPM\_divide\_query(< C, x, y >, \mathcal{I}'$, weight, $C_{buggy}$, $\mathbf{D}$, debug_points, modus)
  **if** $C_{buggy} = \varepsilon$ **then**
    restore($T(F(\sum_{\mathbf{P}}))$)
  **endif**
**end**

---

Um den Vergleich zwischen den beiden Divide-and-Query-Methoden zu zeigen, betrachten wir wieder das Beispiel 8 und nehmen das gleiche Ziel:

```
program([1,2,3,4],[1,2,4],C,S).
```

Verwendet man die Original-Divide-and-Query-Methode, so sind fünf Fragen zu beantworten. Im Fall der Kombination mit *CPM* werden auch fünf Fragen gestellt. Dieses Ergebnis haben wir bekommen, weil es während des Debuggings nicht vorkommt, daß zwei Fragen in den gleichen Testrahmen fallen, also kann man keine vorherige Bewertung benutzen. Dies zeigt, daß die Definition der Testbeschreibung eine große Bedeutung hat. Im Kapitel 6.3 zeigen wir ein anderes Beispiel, bei dem durch Verwendung der *CPM*-Methode die Anzahl der Fragen auch im Divide-and-Query-Fall reduziert werden kann.

**Beispiel 44** Die bei der *Divide-and-Query*-Methode gestellten Fragen, wenn die Datenbank benutzt wird

```
 Is the goal [complement( [1,2,3,4] , [1,2,4] , [4] )] correct? (y/n) n
 Is the goal [complement( [3,4] , [4] , [4] )] correct? (y/n) n
 Is the goal [member( 4 , [3,4] )] correct? (y/n) y
 Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
 Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
The false clause:
delete(4 , [3,4] , [4]) :- delete(4 , [4] , [])
```

**Beispiel 45** Dabei werden in der Datenbank die folgenden Modifikationen durchgeführt:

```
test(complement,3,[more_first,more_second],[([[1,2,3,4],[1,2,4],[4]],
     [[1,2,3,4],[1,2,4],[4]])],incorrect).
test(complement,3,[more_first,one_second],[([[3,4],[4],[4]],[[3,4],[4],[4]])],
     incorrect).
test(member,2,[more,greater],[([4,[3,4]],[4,[3,4]])],correct).
test(delete,3,[more,greater],[([4,[3,4],[4]],[4,[3,4],[4]])],incorrect).
test(delete,3,[one,equal],[([4,[4],[]],[4,[4],[]])],correct).
```

Weil gleich viele Fragen gestellt werden, ist die Abbildung des Ableitungsbaums die gleiche wie in Abbildung 3.5.

## 4.4 Verbesserung des Algorithmus für Terminierung ohne Ausgabe

Wie wir schon im Kapitel 3.2 erwähnt haben, kann der Original-Algorithmus keinen Ableitungsbaum erzeugen, wenn es keine vollständige Ableitung gibt. Dies ist eine große Einschränkung, die wir bei der Entwicklung der grafischen Oberfläche korrigieren mußten, weil alle drei Fehlermöglichkeiten in gleicher Weise behandelt werden sollten. Also wurde die Erzeugung des Ableitungsbaums in zwei Richtungen weiterentwickelt:

- Eine weitere Erweiterung, daß Ableitungsbäume für solche Programme erzeugt werden können, die nicht nur *pure* PROLOG verwenden, sondern auch:
  - Deklarationen
  - Kontrollmöglichkeiten: if (`->`), or (;), repeat
  - Definite Clause Grammars

  - *Built-In*-Prädikate
  - Zusammengesetzte Ziele

- Wenn die Berechnung ohne Ausgabe terminiert, werden nicht berechnete Zweige mit *uncalculated* gekennzeichnet.

Die Abbildung 4.3 wurde zum gleichen Beispiel erzeugt wie im Kapitel 3.2: Wir haben wieder unser *compute*-Beispiel mit dem Ziel:

```
program([1,2,3,4],[3,2,4],C,S).
```

genommen. Das Programm liefert mit diesem Ziel *fail* zurück. Aber der generierte Ableitungsbaum ist nicht der gleiche wie im vorherigen Kapitel.

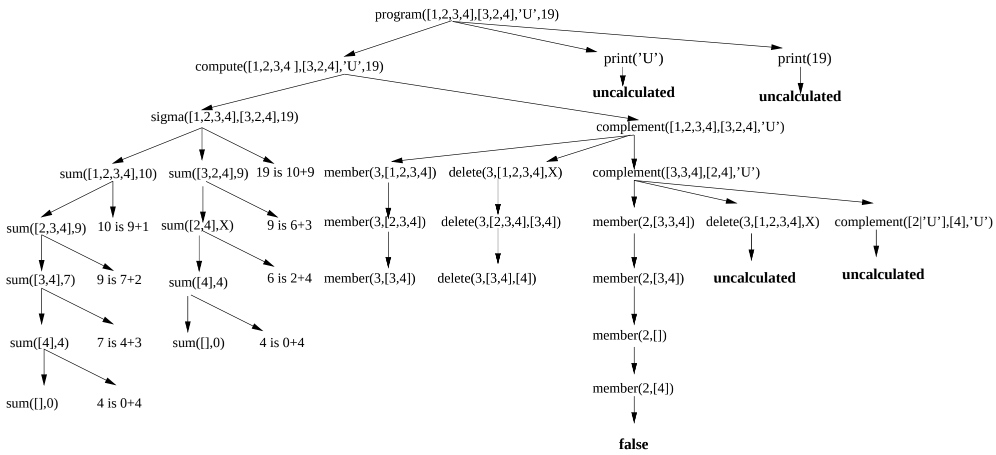



Abbildung 4.3: Der erweiterte Ableitungsbaum zu dem Ziel `program([1,2,3,4],[3,2,4],C,S)`

Der Unterschied liegt nicht nur darin, daß ein neues Symbol *uncalculated* eingeführt wurde. Wenn im Original-Verfahren das gefragte Atom Variablen enthält, sammelt das Verfahren die Variablen ein und erwartet vom Anwender, daß er diese Variablen mit einem Wert bindet. Einerseits kann der Original-Algorithmus nur eine einzige Variable pro Atom behandeln. Andererseits nimmt dieser Algorithmus die Variablennamen aus einer festen Liste, unabhängig davon, welche vom ursprünglichen Programm verwendet wurden. Das kann zu Mißverständnissen führen und erschwert die Beantwortung der Fragen. Im verbesserten können beliebig viele Variablen, gekennzeichnet mit den Originalnamen aus dem Programm, durch die dem Anwender gestellten Fragen behandelt und unifiziert werden.

**Beispiel 46** Der verbesserte Algorithmus stellt dann die folgenden Fragen:

```
This fact fail. Is it correct or not (y/n/c) n
error: missing solution program([1,2,3,4],[3,2,4],'U',19). diagnosing...
Query:program([1,2,3,4],[3,2,4],'U',19)? y
Which 'U' ?[1].

Query:compute([1,2,3,4],[3,2,4],[1],19)? y
```

```
Query:sigma([1,2,3,4],[3,2,4],19)? y

Query:complement([1,2,3,4],[3,2,4],[1])? y

Query:member(3,[1,2,3,4])? y

Query:delete(3,[1,2,3,4],'U')? y
Which 'U' ?[1,2,4].


[Error diagnosed:,delete(3,[1,2,3,4],[1,2,4]), is uncovered.]
Listing of delete(X,Z,U):
delete(X,[X|T],T):- true.
delete(X,[Y|Z],[X|U]) :- delete(X,Z,U).
```

Die grafische Oberfläche erzeugt den in Abbildung 4.4 gezeigten Ableitungsbaum zum Ziel `program([1,2,3,4],[3,2,4],C,S)`.

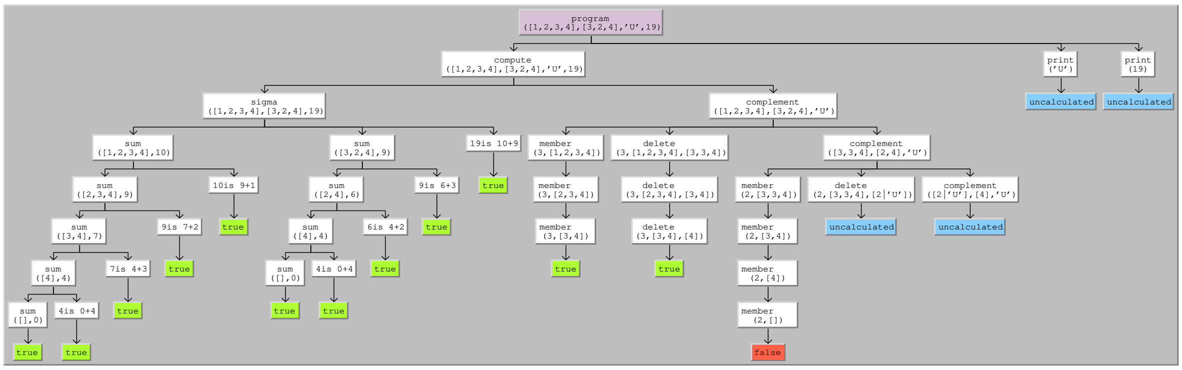


Abbildung 4.4: Der erzeugte Ableitungsbaum für das *compute*-Beispiel zum verbesserten Algorithmus für Terminierung ohne Ausgabe

## 4.5 Zusammenfassung

In diesem Kapitel haben wir zwei der von Shapiro erwähnten Fehlermöglichkeiten (Terminierung mit inkorrekter Ausgabe, Terminierung ohne Ausgabe) um die *CPM*-Methode (*Category Partition Methode*) erweitert.

Das Programm kann nicht mit allen möglichen Eingabeparameterbelegungen getestet werden. Deswegen ist das Ziel der *CPM*-Methode, eine Testdatenbank zu generieren und zu benutzen, um die Anzahl der dem Benutzer gestellten Fragen zu reduzieren. Also sind zunächst die *kritischen Eigenschaften* (*Categories*) der Parameter der Prädikate zu definieren. Nur die kritischen Eigenschaften werden beim Testen überprüft. Diese Kategorien können weiterhin in *Wahlmöglichkeiten* (*choices*) aufgeteilt werden. Aus Kategorien und Wahlmöglichkeiten erzeugt *IDTS* alle möglichen *Testrahmen* (*test frames*), die eigentlich Testdatenbank bilden. Alle Testrahmen enthalten genau eine Wahlmöglichkeit aus jeder Kategorie. Dabei wird vorausgesetzt, daß das Verhalten der Elemente innerhalb eines Testrahmens identisch ist.

Am Anfang des Kapitels wurde der Formalismus dieser Methode beschrieben. Danach haben wir im Fall der Terminierung mit inkorrekter Ausgabe alle drei verschiedenen Durchlaufarten des Ableitungsbaums (Bottom-Up, Top-Down, Divide-and-Query) um *CPM* ergänzt und am gleichen Beispiel (*compute*), das schon bei der Einführung der Original-Algorithmen verwendet wurde, gezeigt, welche Ergebnisse mit diesem ersten Verbesserungsvorschlag erreichbar sind. Wir haben festgestellt, welche wesentlichen Mängel der Algorithmus zur Diagnostizierung der fehlenden Ausgabe hat und haben ihn verbessert. Hier wurde wieder das *compute*-Beispiel verwendet.

# Kapitel 5

# Kombination des Debugging mit Slicing

## 5.1 Formale Beschreibung der Slicing-Technik

In diesem Kapitel werden die theoretischen Grundlagen des Slicing auf Grund [PGH94], [GP95] und [PGH95] erläutert. Das Slicing-Verfahren ist eine allgemeine Methode, die bei algorithmischer Fehlersuche verwendet wird. Mit ihm wird versucht, die Elemente zu finden, die im Verdacht stehen sind, den Fehler zu verursachen. Intuitiv enthält eine Programmslice auf einer bestimmten Variablen die Teile des Programms, die eine Wirkung auf diesen Punkt haben können (*Backward-Slicing*), oder die Punkte, die von dieser Variablen abhängig sind ( *Forward-Slicing*). Die Slicing-Techniken können *statisch* oder *dynamisch* sein. Im ersten Fall wird die Slice ohne Durchführung des Programms festgestellt. Das bedeutet, daß diese Slice bei der Durchführung des Programms gültig ist, aber es werden jedesmal auch Abhängigkeiten betrachtet, die nicht relevant sind. Im dynamischen Fall sieht die Slice abhängig von dem gegebenen Ziel unterschiedlich aus. In unserer hier vorgeführten Methode werden beide Vorgehensweisen kombiniert. Die Grundlagen des Slicing können in [Wei84], [HRB90], [Tip94] und [Kam93] gefunden werden.

Es gibt mehrere Verwendungsmöglichkeiten. Wir beschränken uns hier auf eine Backward-Slicingtechnik, die bei der Fehlersuche in PROLOG behilflich sein kann. Das Forward-Slicing ist eine neuere Methode, die zum Beispiel beim Testen sehr hilfreich sein könnte. Diese Methode kann helfen, die Wirkung der Modifizierung eines Programms festzustellen.

Das Wesentliche des hier erläuterten Slicing-Verfahrens ist, daß die Programmteile bestimmt werden, die auf einen von uns definierten Punkt Wirkung haben könnten. Das gelingt mit Hilfe einer Datenflußanalyse des Programms. Es wird angenommen, daß es zwischen zwei Punkten eine Datenabhängigkeit gibt, wenn beide mit einer Datenflußkante verbunden sind. Um das Slicing durchführen zu können, müssen zwei Vorbereitungschritte gemacht werden. Zuerst wird durch die Analyse des Quellprogramms der *statische Abhängigkeitsgraph* aufgebaut. Dieser enthält alle Datenflußkanten. Weil in PROLOG die Richtung der Daten in den Argumentpositionen nicht direkt bestimmt werden kann, können die Daten, abhängig von dem Aufruf, in beide Richtungen fließen. Wenn im statischen Abhängigkeitsgraphen eine Kante zwischen zwei Positionen existiert, dann ist sie doppelt gerichtet. Als zweites muß zu dem gegebenen Ziel und dem vorgegebenen Programm der *Ableitungsbaum* konstruiert werden.

Nach den Vorbereitungsschritten kann das eigentliche Slicing durchgeführt werden. Das

Verfahren ist rekursiv. Es startet an der vom Benutzer festgelegten fehlerhaften Position und prüft anhand des Programmabhängigkeitsgraphen, von welchen Punkten Kanten zu diesem Punkt führen können. Danach werden aus diesen diejenigen Positionen ausgesucht, die bei dem Aufruf wirklich involviert sind. Die so bestimmten Punkte werden in einer Liste gespeichert und weiter verarbeitet. Der Algorithmus läuft so lange, wie die Liste, die die noch nicht verarbeiteten Punkte enthält, noch nicht leer ist. Die in der Liste auftretenden Punkte bilden den *Backward-Slice.*

Die Aufgabe der statische Analyse ist, die möglichen Datenflußkanten durch statische Überprüfung eines PROLOG-Programms festzustellen. Diese Kanten sind doppelt gerichtet und können in zwei Gruppen eingeteilt werden. Eine Gruppe bilden die *lokalen Kanten*, die den Datenfluß zwischen Argumentpositionen innerhalb einer Klausel bestimmen. In die zweite Gruppe gehören die Kanten, die zwischen zwei Klauseln (eine Argumentposition in einem Atom auf der rechten Seite und eine Argumentposition in einem Atom des Kopfes) stehen.

Im zweiten Schritt wird diese Kantenmenge eingeschränkt, und die Kanten bekommen eine eindeutige Richtung. Um diesen Schritt durchführen zu können, brauchen wir das Programm und ein vorgegebenes Ziel. Aus diesen wird der Ableitungsbaum (siehe Definition 40) aufgebaut. In diesem Baum kommen alle Atome, mit Ausnahme des Zieles und der Systemprädikate, so vor, wie sie auf der rechten Seite einer Klausel stehen. Um die Ableitung fortführen zu können, müssen diese Atome aber einmal oder mehrmals als Kopfelement in einer Klausel stehen. Bei der Generierung des Baums kann man schon die Richtung des Datenflusses feststellen und die sogenannten *Ein-* und *Ausgabepositionen* bestimmen. Diesen Vorgang nennt man auch *Annotieren.* Eine Position, die eigentlich einem Argument in einem Atom entspricht, wird als Eingabeposition gekennzeichnet:

- Wenn das Atom, das die Position enthält, im Kopf steht und an dieser Position keine freie Variable steht, oder
- Wenn das Atom, das diese Position enthält, im Rumpf vorkommt und bei der Durchführung noch nicht gebunden ist.

Eine Position heißt Ausgabeposition:

- Wenn das Atom mit dieser Position im Klauselkopf steht und die vorkommenden Variablen frei sind, oder
- Wenn das Atom, das diese Position enthält, im Rumpf der Klausel steht und bei der Durchführung alle vorkommenden Variablen in dieser Position gebunden sind.

Eine Position kann dual sein, die Daten können also in beide Richtungen fließen, wenn die vorgegebene Position einen zusammengesetzten Term enthält und einige Variablen schon einen Wert haben, während andere noch frei sind. Natürlich müssen nicht alle Positionen annotiert werden. Eine Position kann unannotiert bleiben, wenn wir die Richtung des Datenflusses nicht wissen oder nicht definieren wollen. Der zweite Schritt braucht als Eingabe das Quellprogramm und ein Ziel, und als Ausgabe liefert es den Ableitungsbaum (*parse tree*, *PT*) zusammen mit den Ein- und Ausgabeinformationen in den Positionen.

Eine Annotation heißt unvollständig, wenn es nicht annotierte Positionen gibt, und vollständig, wenn bei allen Positionen festgestellt wurde, ob sie Ein- oder Ausgabepositionen sind. Mehr darüber, wie (unvollständige oder vollständige) Annotationen bestimmt werden können, findet man in [BPM93]. Ein Teil der Annotationen kann automatisch, ohne Programmausführung mit Hilfe der Positionen berechnet werden, deren Annotation unabhängig vom Kontext vorgegeben ist. Diese Methode heißt *statische Annotation.*

Das Slicing selbst braucht den Ableitungsbaum ($PT$) und einen Anfangspunkt (*starting point*, $SP$), an dem man den Fehler festgestellt hat. Das System startet bei diesem Punkt und sucht rekursiv nach den zur Slice gehörenden Punkten. Informell können wir den Algorithmus wie folgt beschreiben:

- Es wird eine Liste, die Positionen aus dem Ableitungsbaum enthält, generiert (*saved found points*, $SFP$). In dieser Liste werden die Positionen, die schon gefunden wurden, aber noch nicht verarbeitet wurden, gespeichert. Das Anfangselement des $SFP$ ist $SP$.

- Die erste Position (*current point*, $CP$) dieser Liste wird genommen, und es wird überprüft, von welchen Positionen diese Position abhängig ist. Diese Positionen werden in $SFP$ gespeichert. Die Abhängigkeit kann lokal oder global sein, und es wird überprüft, ob sie schon einmal vorgekommen ist oder nicht. Wenn sie schon vorhanden ist, dann wird sie nicht zu $SFP$ hinzugefügt.

- So lange $SFP$ nicht leer ist, wird der zweite Schritt wiederholt.

Wie wir schon erwähnt haben, ist der Programmabhängigkeitsgraph zuerst zu konstruieren. Dafür haben wir das Programm *GAPLOG* ([Ekl94]) verwendet. Dieses Programm wird um ein *Slicing-Verfahren* und um die dort eingeführten Techniken erweitert. Die Sätze und Definitionen dieses Kapitels werden mit Hilfe des Beispielprogramms *compute* erläutert.

### 5.1.1 Statische Analyse

Wie man in [DM93] detailliert lesen kann, gibt es eine große Ähnlichkeit zwischen attributierten Grammatiken ([Knu68], [DJL88]) und logischen Programmen ([AM91]). Zuerst zeigen wir, wie die bei den Grammatiken verwendeten Techniken zur Attributberechnung auch hier sinnvoll eingesetzt werden können. Wie wir in Kapitel 2 bereits definiert haben, sei $\mathbf{P}$ ein Programm und $\mathbf{T}$ der dazugehörige Ableitungsbaum, dessen Wurzel das vorgegebene Ziel ist. Einer Klausel $C_i : A_0 \leftarrow A_1, \ldots A_n$ kann ein Teilbaum mit der Wurzel $A_0$ und den Nachkommen $A_i$, $1 \leq i \leq n$ zugeordnet werden. Ein Programm ist dann eine Menge von Klauseln, $\mathbf{P} = \cup_{C_i \in \mathbf{P}} C_i$. $A_0$ heißt Kopfelement, und $A_1$ ist das erste Element im Körper. $\Theta$ sei eine Substitution und $\delta_0, \delta_1$ Umbenennungssubstitutionen.

Die meisten PROLOG-Systeme erlauben, externe, nicht in einer logischen Programmiersprache geschriebene Teile zu verwenden, wie zum Beispiel die in C implementierte *print*-Anweisung. Solche Programme heißen *funktionale logische Programme.* Die in solchen Elemente enthaltenen Positionen können automatisch annotiert werden, weil klar ist, was Eingaben und was Ausgaben sind. Außerdem gibt es eine Menge von Systemprädikaten in PROLOG, deren Annotierung nicht beliebig definierbar, sondern vorgegeben ist. Denken wir an das im *compute*-Beispiel benutzte *is*-Prädikat, dessen zweites Argument immer gebunden werden muß. Eine Berechnung der Datenabhängigkeitsgraphen kann mit diesen Positionen gestartet werden. Formal können wir die hier beschriebenen Ideen wie folgt definieren.

**Definition 67** Sei $\mathbf{P}$ ein Programm. Die Funktionssymbole sind aus zwei disjunkten Mengen zusammengestellt: Eine ist die Menge der Konstruktoren ($\mathcal{C}$), die andere ist die Menge der vordefinierten Symbole ($\mathcal{F}$). Aus den vordefinierten Symbolen erzeugte Terme werden als *Funktionalterme* bezeichnet.

Erläutern wir diese Definition durch das Beispiel *compute*.

**Beispiel 47** Die Klausel $C_1$: `program(L1,L2,C,S) :- compute(L1,L2,C,S), print(C), print(S)` enthält das Atom *print* zweimal. Einmal mit $C$ als Argument und einmal mit $S$. *print* ist eine vordefinierte Symbole. In dem Moment des Aufrufs muß das Argument des *print*-Prädikats schon einen Wert haben. Die dort vorkommende Variable muß schon gebunden sein, sonst kann nichts auf den Bildschirm geschrieben werden.

Ein anderes Vorkommen einer vordefinierten Symbole findet man in der Klausel $C_9$:`sigma(L1,L2,S) :- sum(L1,S1), sum(L2,S2), S is S1+S2`: das *is*-Prädikat. Dieses Prädikat hat zwei Argumente, wobei das zweite Argument beim Aufruf immer gebunden sein muß. In diesem Fall kann $S$ berechnet werden, wenn $S1$ und $S2$ schon einen Wert haben.

**Definition 68** Sei $q$ ein Prädikat in $\mathbf{P}$. $q$ hat $m$ Argumente, das Argument an Position $k$ bezeichnen wir mit $q_k$. Seien *Argpos*$(q)$ die *Argumentpositionen* in $q$, die Menge der Argumente von $q$ und *Argpos(*$\mathbf{P}$*)* $= \cup_{q \in \mathbf{P}}$ Argpos $(q)$.

**Definition 69** Sei $C_i$:$A_0 \leftarrow A_1, \ldots, A_n$ eine Klausel in $\mathbf{P}$. Wir können das $k$-te Argument in $A_j$ eindeutig mit dem Quadrupel $(C_i, j, q, k)$ identifizieren, wobei $q$ das in $A_j$ verwendete Prädikatsymbol ist. POS($\mathbf{P}$) ist als $\{$ POS($C_i$) $|C_i$ ist eine Klausel in $\mathbf{P}\}$ und POS($C_i$) als $\{(C_i, j, q, k) | j = 0, \ldots, n_i, k = 1, \ldots, m_j\}$ definiert, wobei $n_i$ die Anzahl von Atomen in der Klausel $C_i$ und $m_j$ die Anzahl von Argumenten im Atom $A_j$ ist. Für eine Position kann eine Unifikation definiert werden.

Nehmen wir die Klausel $C_2$ aus unserem *compute*-Beispiel, um die Bedeutung der obigen zwei Definitionen zu illustrieren.

**Beispiel 48** Die Klausel $C_2$ ist die folgende:

```
compute(L1,L2,C,S) :- sigma(L1,L2,S), complement(L1,L2,C).
```

Das Prädikat *compute* im Kopf hat vier Argumente: `L1`, `L2`, `C` und `S`. Diese bilden die Menge *Argpos*(*compute*). Nehmen wir das Prädikat *complement* aus dem Rumpf von $C_2$. Das zweite Argument $L2$ kann mit dem Quadrupel $(C_2, 2, complement, 2)$ eindeutig identifiziert werden. Dieses Argument kommt in der zweiten Klausel ($C_2$), im zweiten Atom (2, complement) und in der zweiten Position vor, wobei das im Kopf stehende Atom die Nummer 0 hat.

**Definition 70** $C_1$ und $C_2$ sind Klauseln. Wir schreiben: $C_1 \rightarrow C_2$, wenn es ein Atom $A_i$ im Körper von $C_1$ gibt, so daß mgu($A_i\delta_0$, head($C_2$)$\delta_1$) $\neq fail$. Die Relation $\rightarrow$ heißt *statischer Graphaufruf*.

**Definition 71** Ein Programm heißt genau dann *funktional vollständig*, wenn bei der erfolgreichen Durchführung des Programms alle *verzögerten Aufrufe* korrekt sind. Die funktionale Vollständigkeit hängt dabei vom Anfangsziel ab. Die Gebundenheit einiger Argumente des Anfangszieles kann ausreichende Voraussetzung für die funktionale Vollständigkeit sein. *Verzögerungsfreiheit* bedeutet, daß alle Aufrufe eines Programms mit der Ausführungsstrategie von PROLOG gebunden sind, so daß die Verwendung von *dynamischen Verzögerungen* nicht notwendig ist.

**Definition 72** Ein *Argument* in einem Atom des PROLOG-Programms kann *geerbt* ($\downarrow$), *synthetisiert* ($\uparrow$), *dual* ($\updownarrow$) oder *nicht annotiert* ($\Box$) sein. So kann eine Funktion $\mu$ definiert werden:

$\mu$: Argpos $(\mathbf{P}) \rightarrow \{\downarrow, \uparrow, \updownarrow, \Box\}$

Die *Annotation* ist *unvollständig*, wenn sie duale oder nicht annotierte Positionen enthält.

**Beispiel 49** Nehmen wir wieder das schon oft verwendete Beispiel *compute* und davon die erste Klausel. Abbildung 5.1 zeigt, wie die Argumente in dieser Klausel annotiert werden, wenn wir sie mit dem Ziel `compute([1,2,3,4], [1,2,4], C, S)` aufrufen.

$$\mathrm{compute}(\overset{\downarrow}{\mathbf{L1}}, \overset{\downarrow}{\mathbf{L2}}, \overset{\uparrow}{C}, \overset{\uparrow}{S}) \text{ :- } \mathrm{sigma}(\overset{\downarrow}{L1}, \overset{\downarrow}{L2}, \overset{\uparrow}{\mathbf{S}}), \mathrm{complement}(\overset{\downarrow}{L1}, \overset{\downarrow}{L2}, \overset{\uparrow}{\mathbf{C}}).$$

Abbildung 5.1: Die Annotierung für die erste Klausel im Programm *compute*

Bei einem Prädikataufruf sind die geerbten Argumente gebunden. Nach der erfolgreichen Ausführung werden die synthetisierten Argumente gebunden. Wenn wir bei einem Argument die Richtung des Datenflusses nicht feststellen können, dann sprechen wir von dualen Argumenten.

**Definition 73** Eine *Annotation* $\mu$ für ein Prädikat heißt genau dann *korrekt*, wenn die Eigenschaft, daß die geerbten Argumente bei dem Aufruf schon gebunden sind und die synthetisierten Argumente nach der Ausführung gebunden werden, gültig ist. Eine Annotation in **P** ist genau dann korrekt, wenn die Annotation für alle Prädikate $C_i$ im Programm korrekt ist.

Nun haben wir alle notwendigen Definitionen, um die Annotation auf Klauseln erweitern zu können. Zur Veranschaulichung der folgenden Definitionen betrachten wir Abbildung 5.2. Hierbei sei die Klausel $C_i$ von der Form

$A_0 : -A_1, \ldots, A_k$.

Außerdem bezeichne $q_{lj}$ das Argument $l$ des Prädikats $A_j$ (mit $j = 0, \ldots, k$), wobei $A_j$ $m_j$ viele Argumente besitzt. Wenn $\mu(q_{l0}) =\downarrow$, oder wenn $\mu(q_{lj}) =\uparrow$, wobei $j = 1, \ldots, k$, dann sprechen wir bei $q_{l0}$ bzw. $q_{lj}$ von einer *Eingabeposition.* Wenn $\mu(q_{l0}) =\uparrow$ oder wenn $\mu(q_{lj}) \downarrow$, wobei $j = 1, \ldots, k$, dann ist die Position $q_{l0}$ bzw. $q_{lj}$ eine *Ausgabeposition.*

**Beispiel 50** Wenn wir die in Abbildung 5.1 gekennzeichneten Positionen betrachten, können wir die Ein- und Ausgabepositionen wie folgt angeben:
Eingabepositionen: ($C_1$,0,compute,1), ($C_1$,0,compute,2), ($C_1$,1,sigma,3), ($C_1$,2,complement,3)
Ausgabepositionen: ($C_1$,0,compute,3), ($C_1$,0,compute,4), ($C_1$,1,sigma,1), ($C_1$,1,sigma,2), ($C_1$,2,complement,1), ($C_1$,2,complement,2)
In der Abbildung 5.3 haben wir die Eingabepositionen mit Fettdruck und die Ausgabepositionen mit Kursivschrift voneinander getrennt.

**Definition 74** Sei

$$\begin{aligned} I(C_i) &:= \{\text{die Eingabepositionen von } C_i\} \\ O(C_i) &:= \{\text{die Ausgabepositionen von } C_i\} \\ I(\mathbf{P}) &:= \cup_{C_i \in \mathbf{P}} I(C_i) \\ O(\mathbf{P}) &:= \cup_{C_i \in \mathbf{P}} O(C_i) \end{aligned}$$

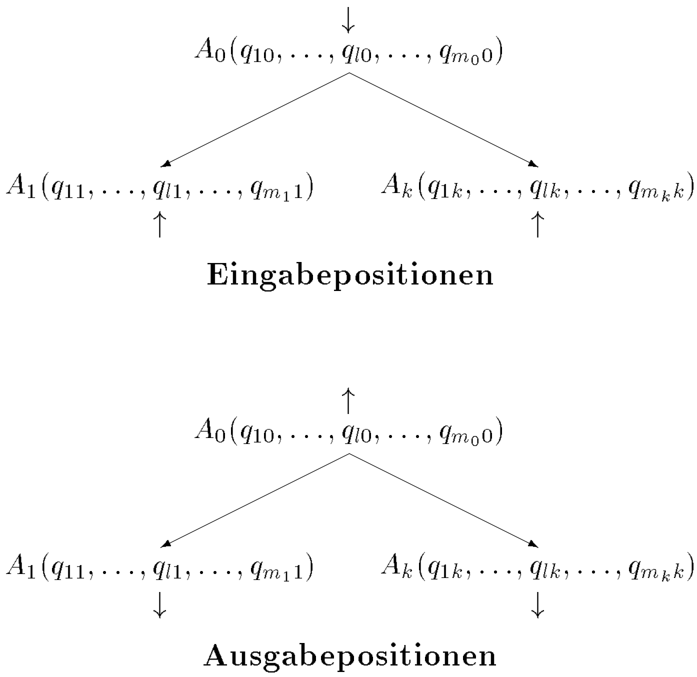



Abbildung 5.2: Ein- und Ausgabepositionen einer Klausel in Baumdarstellung

**Definition 75** Für alle Klauseln $C \in \mathbf{P}$ gilt: $\mathrm{Dep}_C \subseteq I(C) \times O(C)$ ist der *lokale Abhängigkeitsgraph*, wenn $\gamma \in I(C)$ und $\beta \in O(C)$ beide wenigstens eine gemeinsame Variable besitzen. Wir schreiben kurz $\gamma Dep_C \beta$.

**Beispiel 51** Nehmen wir wieder das *compute*-Beispiel (siehe Beispiel 8). Es wird angenommen, daß die eingebauten (built-in) Prädikate eine normale Annotierung (also *is/2* $\uparrow$, $\downarrow$, *print/1* $\downarrow$) haben. Der Term im zweiten Argument des Prädikats *is/2* und im Argument *print/1* müssen schon beim Aufruf der sie enthaltenden Klausel bewertet sein. Deswegen können diese Argumente als funktionaler Ausdruck definiert werden. Aufgrund dieser Feststellung kann die kleinste vollständige Programm-Annotation wie folgt berechnet werden:

Dieses Programm verwendet zweimal funktionale Ausdrücke mit *is*, in der Klausel $C_9$ mit dem zweiten Argument $S1 + S2$ und in der Klausel $C_{11}$ mit dem zweiten Argument $S1 + X$. Weil die Variablen auf der rechten Seite von *is* in den jeweiligen Klauseln (*sigma, sum*) in funktionalen Ausdrücken vorkommen, können sie als Ausgabepositionen annotiert werden. Die Annotierung des Prädikats *sum* ergibt sich als ($\downarrow\uparrow$), weil $S$ bei dem *is*-Prädikat in Klausel $C_{11}$ als synthetisierte Eingabeposition und $X$ als geerbte Ausgabeposition auftritt, also *sum* eine synthetisierte Ausgabeposition und eine geerbte Eingabeposition besitzt. Jedes Auftreten des *sum*-Prädikats muß die gleiche Annotierung besitzen. In Klausel $C_{11}$ besitzt also das *sum*-Prädikat im Körper ebenfalls die Annotierung ($\downarrow\uparrow$). Das bedeutet, daß $T$ eine geerbte Ausgabeposition und $S1$ eine synthetisierte Eingabeposition ist. Damit ergeben sich

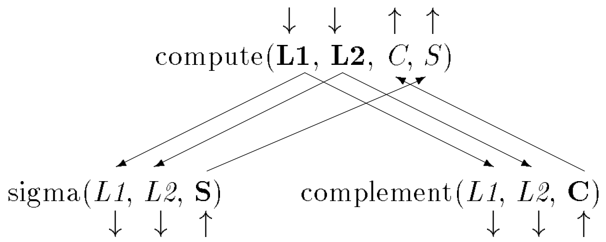



Abbildung 5.3: Die Ein- und Ausgabepositionen für die erste Klausel im Programm *compute*

die folgenden lokalen Abhängigkeiten für Klausel 11:

$$\begin{aligned}
(C_{11}, 0, sum, 1) &\leadsto_{C_{11}} (C_{11}, 1, sum, 1)\ (T)\\
(C_{11}, 0, sum, 1) &\leadsto_{C_{11}} (C_{11}, 2, is, 2)\ (X)\\
(C_{11}, 1, sum, 2) &\leadsto_{C_{11}} (C_{11}, 2, is, 2)\ (S1)\\
(C_{11}, 2, is, 1) &\leadsto_{C_{11}} (C_{11}, 0, sum, 2)\ (S)
\end{aligned}$$

$\leadsto_{C_i}$ kennzeichnet die Kanten in $Dep_{C_i}$.
Graphisch kann man diese Abhängigkeiten wie in Abbildung 5.4 veranschaulichen.

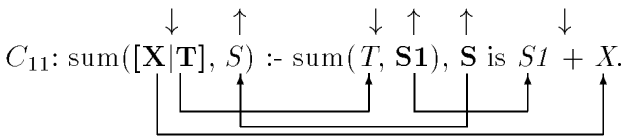



Abbildung 5.4: Lokale Abhängigkeiten in der Klausel $C_{11}$

Auf ähnliche Weise erhält man in Klausel $C_9$ für das Prädikat *sigma* die Annotierung (↓↓↑) und folgende lokale Abhängigkeiten (Abbildung 5.5):

$$\begin{aligned}
(C_9, 0, sigma, 1) &\leadsto_{C_9} (C_9, 1, sum, 1)\ (L1)\\
(C_9, 0, sigma, 2) &\leadsto_{C_9} (C_9, 2, sum, 1)\ (L2)\\
(C_9, 1, sum, 2) &\leadsto_{C_9} (C_9, 3, is, 2)\ (S1)\\
(C_9, 2, sum, 2) &\leadsto_{C_9} (C_9, 3, is, 2)\ (S2)\\
(C_9, 3, is, 1) &\leadsto_{C_9} (C_9, 0, sigma, 3)\ (S)
\end{aligned}$$

Selbstverständlich kann der Benutzer auch selbst eine Annotation angeben. Durch den Aufruf des Programms kann aber auch eine bestimmte Annotierung festgelegt werden: Wenn das Programm mit *program*$([1,2,3,4],[1,2,4],S,C)$ aufgerufen wird, wobei $S$ die berechnete Summe und $C$ die symmetrische Differenz der ersten beiden Argumente ist, also im Moment des Aufrufs $S$ und $C$ noch nicht gebunden sind, dann ergibt sich die Annotierung: *compute* (↓↓↑↑).

Im folgenden betrachten wir Abhängigkeiten zwischen verschieden Klauseln.

$$C_9\text{: sigma}(\overset{\downarrow}{\mathbf{L1}},\ \overset{\downarrow}{\mathbf{L2}},\ \overset{\uparrow}{S})\text{ :- sum}(\overset{\downarrow}{L1},\ \overset{\uparrow}{\mathbf{S1}}),\ \text{sum}(\overset{\downarrow}{L2},\ \overset{\uparrow}{\mathbf{S2}}),\ \overset{\uparrow}{\mathbf{S}}\text{ is } S1 \overset{\downarrow}{+} S2.$$

Abbildung 5.5: Lokale Abhängigkeiten in der Klausel $C_9$

**Definition 76** Seien $C$ und $D$ zwei Klauseln. $D_0$ ist der Kopf von $D$, und $C_j$ ist ein Atom im Rumpf von $C$, so daß $\text{mgu}(C_j\delta_0, D_0\delta_1) \neq$fail gilt. Der *globale Abhängigkeitsgraph* $Dep_{C,D}$ über $O(C \cup D) \times I(C \cup D)$ wird wie folgt definiert:

$$\gamma Dep_{C,D}\beta \leftrightarrow \begin{cases} \gamma = (C, j, q, k) \\ \beta = (D, 0, q, k) \\ \mu(q_k) = \downarrow \end{cases} \quad oder \begin{cases} \gamma = (D, 0, q, k) \\ \beta = (C, j, q, k) \\ \mu(q_k) = \uparrow \end{cases}$$

**Definition 77** *Der Programmabhängigkeitsgraph* kann wie folgt definiert werden:

$$Dep_{\mathbf{P}} = \bigcup_{C \in \mathbf{P}} Dep_C \cup \bigcup_{C,D \in \mathbf{P}} Dep_{C,D}$$

Die oben definierten Begriffe können wie folgt interpretiert werden: In dem Programm kann sich die Information auf zwei verschiedene Weisen verbreiten: Entweder innerhalb einer Klausel (zwischen gleichen Variablen an zwei oder mehreren Positionen). In diesem Fall sprechen wir von einem *lokalen Abhängigkeitsgraphen*. Oder aber die Information fließt zwischen zwei verschiedenen Klauseln. Wenn wir die Annotation $\mu$ kennen, dann können wir entscheiden, in welche Richtung sich die Information verbreitet, nämlich:

- von der Eingabeposition zur Ausgabeposition in einer Klausel oder
- von der Ausgabe zur Eingabe durch die Unifizierung (siehe Definition 25)

**Beispiel 52** Verfolgen wir unser *compute*-Beispiel weiter. Bei der Konstruktion der lokalen Abhängigkeiten (Beispiel 51) haben wir gesagt, daß jedes Auftreten eines Prädikats auf die gleiche Weise annotiert werden muß. Das ist eine Folge der Unifikation. Dies führt wiederum zu globalen Abhängigkeitskanten von Ausgabepositionen zu den Eingabepositionen bei der Unifikation. Wenn wir die Unifikation der *sum*-Prädikate in Klausel $C_9$ mit den Klauseln $C_{10}$ und $C_{11}$ betrachten, erhalten wir folgende globale Abhängigkeiten (Abbildung 5.6):

$$\begin{array}{rll}
(C_9, 1, sum, 1) & \rightsquigarrow_{C_9,C_{10}} & (C_{10}, 0, sum, 1)(L1) \\
(C_9, 1, sum, 1) & \rightsquigarrow_{C_9,C_{11}} & (C_{11}, 0, sum, 1)(L1) \\
(C_9, 2, sum, 1) & \rightsquigarrow_{C_9,C_{10}} & (C_{10}, 0, sum, 1)(L2) \\
(C_9, 2, sum, 1) & \rightsquigarrow_{C_9,C_{11}} & (C_{11}, 0, sum, 1)(L2) \\
(C_{10}, 0, sum, 2) & \rightsquigarrow_{C_{10},C_9} & (C_9, 1, sum, 2)(S1) \\
(C_{10}, 0, sum, 2) & \rightsquigarrow_{C_{10},C_9} & (C_9, 2, sum, 2)(S2) \\
(C_{11}, 0, sum, 2) & \rightsquigarrow_{C_{11},C_9} & (C_9, 1, sum, 2)(S1) \\
(C_{11}, 0, sum, 2) & \rightsquigarrow_{C_{11},C_9} & (C_9, 2, sum, 2)(S2)
\end{array}$$

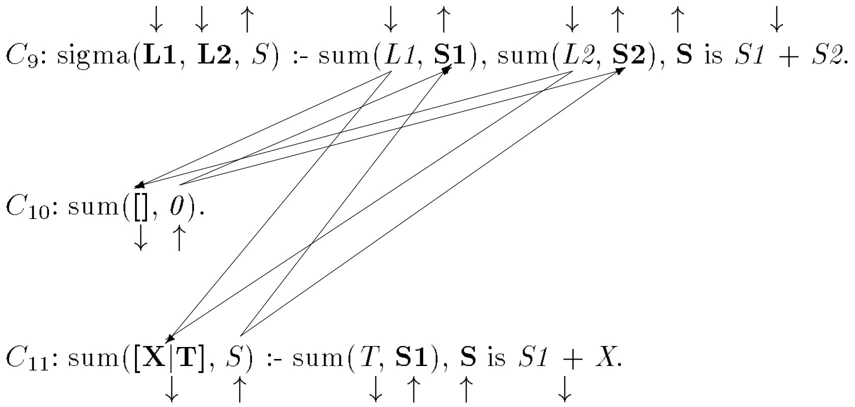



Abbildung 5.6: Die globalen Abhängigkeiten bei der Unifikation von *sum* aus Klausel $C_9$ mit den Klauseln $C_{10}$ und $C_{11}$

Die Knoten eines Graphen (die Positionen in Klauseln), zu denen keine Kante hinführt, bezeichnen wir als minimal.

Sei $\gamma$ eine Ausgabeposition. Wenn alle minimalen Knoten des globalen Abhängigkeitsgraphen, die einen Pfad zu $\gamma$ besitzen, gebunden sind und wenn alle Abhängigkeitskanten, die zu $\gamma$ führen, entlang der Ausführungsreihenfolge (*depth-first*, *left-right*) liegen, dann ist $\gamma$ sicher gebunden.

Die erste Bedingung heißt *einfache Annotation*, während die zweite der Definition von $L$-attributierten Grammatiken entspricht ([LRS74], [Boc76]) und als $L$-Annotation bezeichnet wird.

Um die für die Slice von **P** notwendigen Informationen zu berechnen, wird der Programmabhängigkeitsgraph $Dep_{\mathbf{P}}$ benutzt.

Für alle funktionalen Positionen $\gamma$ suchen wir die Bedingungen, für die alle Vorkommen der Variablen sicher gebunden werden. Um diese Bedingungen näher beschreiben zu können, benötigen wir einige weitere Definitionen.

**Definition 78** $\gamma \in O(\mathbf{P})$ ist eine *einfache Position*, wenn für alle $C_i \in \mathbf{P}$ und für jede Ausgabeposition $\beta$ gilt: $\beta Dep^*_{\mathbf{P}}\gamma$, und alle Variablen in $\beta$ kommen in $C_i$ in einer Eingabeposition vor.

**Definition 79** Eine *Annotation* für ein Programm **P** ist genau dann *einfach*, wenn alle $\gamma \in O(\mathbf{P})$ einfach sind. Dabei wird angenommen, daß die Abhängigkeit nicht zyklisch ist, also $X = f(X)$ nie bewertet wird.

**Definition 80** Seien $D$ und $C$ zwei Klauseln, so daß $C$ in der Ableitung von $D$ auftritt, also $D \stackrel{*}{\rightarrow} C$, wobei $\stackrel{*}{\rightarrow}$ die transitive Hülle von $\rightarrow$ ist, und sei $\gamma \in O(C)$. $Dep_\gamma$ kann dann wie folgt definiert werden:

$$\beta_1 \mathrm{Dep}_\gamma \beta_2 \leftrightarrow (\beta_1 \mathrm{Dep}_D \beta_2 \wedge \beta_2\ \mathrm{Dep}_{\mathbf{P}} \gamma)$$

**Definition 81** Die Position $\gamma \in O(\mathbf{P})$ ist eine *L-annotierte Position*, wenn gilt:

$$(D, i, p, k)\ \mathrm{Dep}_\gamma(D, j, g, l) \Rightarrow (i < j \text{ oder } i = j = 0)$$

Um diese Definitionen zu erklären, nehmen wir ein anderes Beispiel als vorher. Das neue Beispiel wurde im Buch von [DM93] vorgestellt.

**Beispiel 53** Die Eingabe des Programms ist eine Liste von Listen:

```
[[This, is, some, text], [Another, line, of, text]]
```

und die erzeugte Ausgabe kann als eine Steuerung für einen Drucker interpretiert werden:

```
[[put(1,1,This), put(1,8,is), put(1,12,some), put(1,16,text)],
[put(2,1,Another), put(2,8,line), put(2,12,of), put(2,16,text)]]
```

wobei die ersten zwei Argumente von `put` die Zeilen- und Spaltenkoordinaten bedeuten. In diesem Fall kann die Ausgabe der durchgeführten Anweisungen folgenden Text liefern:

```
This    is   some text
Another line of   text
```

Alle Spalten sind so breit wie das breiteste Wort in der Spalte. Das Programm beginnt mit dem Hauptprädikat `tt`:

```
C1: tt(_, _, [], X, X, []).
C2: tt(Line,Ind,[Row|Rows],MaxWidths,MaxSoFar,[InstRow|Insts]):-
            tr(Line, Ind, Row, MaxWidths, Widths, InstRow),
            compute_max(Widths, MaxSoFar, NewMaxSoFar),
            tt(Line+1, Ind, Rows, MaxWidths, NewMaxSoFar, Insts).
```

Die Argumente haben die folgende Bedeutung:

`Line`: Die aktuelle Zeilennummer
`Ind`: Die aktuelle Position in der Zeile
`[Row|Rows]`: Die ganze Eingabe
`MaxWidths`: Die Breite des breitesten Elements in jeder Spalte
`MaxSoFar`: Die Breite des breitesten Elements für jede Spalte in den bisher verarbeiteten Zeilen
`[InstRow|Insts]`: Die Liste von Druckeranweisungen

Das Atom `compute_max` berechnet aus zwei Listen, die natürliche Zahlen als Elemente enthalten, paarweise das Maximum, also aus $[i_1, \ldots, i_n]$ und $[j_1, \ldots, j_n]$ $[max(i_1, j_1), \ldots, max(i_n, j_n)]$.

```
C3: tr(_, _, [], [], [], []).
C4: tr(Line,Ind,[Text|Ts],[MaxWid|Ms],[size(Text)|Ss],
      [put(Line, Ind, Text)|Insts])
          :- tr(Line, Ind+MaxWid, Ts, Ms, Ss, Insts).
```

Hier sind + und `size(Text)` Funktionalterme. `size(Text)` liefert die Anzahl der im `Text` vorkommenden Zeichen. Die Argumente haben die folgende Bedeutung:

`Line`: Die aktuelle Zeilennummer

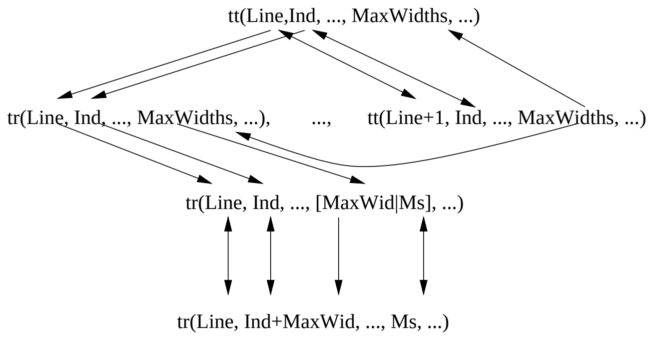



Abbildung 5.7: Ein Teil des Abhängigkeitsgraphen für das Programm `tt`

`Ind`: Die aktuelle Position in der Zeile
`[Text|Ts]`: Ein Element in der Liste, zum Beispiel: [This, is, some, text]
`[MaxWid|Ms]`: Die Breite des breitesten Elements in jeder Spalte
`[size(Text)|Ss]`: Die Anzahl der Zeichen jedes Elements in einer Zeile
`[put(Line, Ind, Text)|Insts]`: Die Liste von Druckeranweisungen

Wir können feststellen, daß die maximale Breite der Spalten so lange nicht berechnet wird, bis die ganze Eingabe verarbeitet ist. Das bedeutet, daß `MaxWid` im Ausdruck `Ind+MaxWid` in Klausel $C_4$ nicht sofort berechnet wird, sondern erst, wenn der rekursive Aufruf des `tr`-Prädikats beendet ist.

Bauen wir zuerst $Dep_{\mathbf{P}}$ für dieses Programm auf: Abbildung 5.7 zeigt einen Teil davon ($C_2$ und $C_4$), wobei angenommen wird, daß die Annotierung von `tr` $\downarrow\downarrow\downarrow\downarrow\uparrow\uparrow$ und von `tt` $\downarrow\downarrow\downarrow\uparrow\downarrow\uparrow$ ist.

Eine einfache, und $L$-annotierte Position im Programm `tt` ist $(C_4, 1, tr, 1)$ (`Line`). Eine einfache aber nicht $L$-annotierte Position ist die Position $\gamma = (C_4, 1, tr, 2)$ (`Ind+MaxWid`), weil eine Abhängigkeit von rechts nach links, von $(C_2, 3, tt, 4)$ zu $(C_2, 1, tr, 4)$, vorhanden ist und $(C_2, 3, tt, 4)$ $Dep_{\mathbf{P}}$ $\gamma$.

**Satz 13** Seien $C : A_0 \leftarrow A_1, \ldots, A_n$ eine Klausel und $\gamma$ eine einfache $L$-annotierte Position in $A_i$, wobei $A_i$ einen funktionalen Ausdruck $e$ enthält. Wenn $A_i$ das aktuelle Atom und $\theta$ die aktuelle Substitution ist, dann ist $e\theta$ gebunden und kann unmittelbar ausgewertet werden.

**Definition 82** Die Position $\gamma \in O(\mathbf{P})$ ist eine *$R$-annotierte Position*, wenn sie nicht $L$-annotiert ist. $\gamma$ ist *lokal $R$-annotiert*, wenn es $R$-annotiert ist und für alle $L$-annotierten $\beta \in O(\mathbf{P})$ $\beta Dep^*_{\mathbf{P}}\gamma$, $D \xrightarrow{*} C$ und $D \neq C$ gilt.

**Beispiel 54** Eine lokal $R$-annotierte Position in `tt` ist: $(C_4, 1, tr, 4)$, weil diese von rechts nach links abhängig ist. Daraus folgt, daß $(C_2, 1, tr, 4)$ keine lokal $R$-annotierte Position (siehe Abbildung 5.7) ist.

**Satz 14** Seien $C : A_0 \leftarrow A_1, \ldots, A_n$ eine Klausel, $\gamma$ eine einfache und lokal $R$-annotierte Position in $A_i$ und ferner $A_j$ das am weitesten rechts stehende Atom in $C$, so daß $A_j$ eine

Eingabeposition $\gamma'$ enthält, wobei $\gamma' Dep_C \gamma$. Nehmen wir an, daß $C$ aufgerufen wird und daß $A_k$, $1 \leq k \leq n$, durch die Substitution $\theta$ berechnet werden kann, dann ist $\gamma\theta$ gebunden.

Die wichtigste Voraussetzung, unter der Annotationen in *IDTS* durchgeführt werden, kommt von der Grundeigenschaft von PROLOG-Programmen, daß funktionale Ausdrücke sofort berechnet werden.

Andererseits müssen in $L$-attributierten Grammatiken alle lokalen Abhängigkeiten, die eine $L$-annotierte Position bestimmen, von links nach rechts oder direkt aus dem Kopf abgeleitet werden. Das Slice-Programm versucht, eine gültige Annotation, die auf [BPM93] basiert, zu berechnen. In der Berechnung kann man auch aus einer minimalen Annotation starten. Am Anfang haben alle Prädikatpositionen $\Box$ als Grundannotation. Dabei sind die Systemprädikate ausgenommen. Sie haben eine vordefinierte Annotation. Der Benutzer kann selbst Anfangsannotationen eingeben. Der wichtigste Punkt bei der Berechnung ist aber das vorgegebene Ziel. Das *IDTS*-System kann aus den im Ziel gebundenen und nicht gebundenen Positionen eine Initialannotation berechnen. Diese wird dann schrittweise konsistent vergrößert, bis eine einfache $L$-Annotation für das Programm gefunden ist. Der Algorithmus ordnet nur den noch nicht annotierten Positionen $\downarrow$ und $\uparrow$ zu, die die Berechnung von funktionalen Ausdrücken beeinflussen. Daraus folgt, daß die Annotation unvollständig sein kann.

Wie wir früher erwähnt haben, wird angenommen, daß die Annotierung azyklisch ist. Die Azyklizität kann einfach überprüft werden: Aus dem Programmabhängigkeitsgraphen $Dep_{\mathbf{P}}$ wird für alle $C \in \mathbf{P}$ der Klauselabhängigkeitsgraph $Dep_{C^*}$ generiert und die Zyklenfreiheit überprüft.

**Definition 83** Sei $Dep_{\mathbf{P}}$ ein Programmabhängigkeitsgraph und $C \in \mathbf{P}$ eine Klausel. Der *Klauselabhängigkeitsgraph* $Dep_{C^*}$ ist der folgende:

- Die Knoten von $Dep_{C^*}$ sind alle Positionen von $C$, die Knoten von $Dep_{\mathbf{P}}$ entsprechen.
- Eine Kante $\beta Dep_{C^*} \gamma \leftrightarrow \beta Dep_C \gamma$ ist eine lokale Abhängigkeit in $Dep_{\mathbf{P}}$.
- Sei $\gamma = (C, j, q, k), \beta = (C, j, q, l)$ mit $j > 0$, $\mu(q_k) =\downarrow$ und $\mu(q_l) =\uparrow$. $\gamma Dep_{C^*} \beta$ ist genau dann eine Kante, wenn $\gamma Dep^+_{\mathbf{P}} \beta$ eine transitive Abhängigkeit in $Dep_{\mathbf{P}}$ ist.

Wenn $Dep_{C^*}$ $\forall C \in P$ keinen Zyklus enthält, dann ist auch die Annotation von $\mathbf{P}$ azyklisch.

**Satz 15** Wenn alle Ausgabepositionen des Anfangszieles gebunden sind und die Annotierung von $\mathbf{P}$ azyklisch ist, dann werden alle funktionalen Terme in allen Ableitungsbäumen von $\mathbf{P}$ durch dieses Anfangsziel gebunden.

### 5.1.2 Die Slicing-Technik

Beim Debugging möchten wir die zu dem inkorrekten Ergebnis führenden Klauseln finden. Das sogenannte Programm-Slicing ist eine Standardtechnik, bei der durch Transformation eines Programms der Suchraum beschränkt wird. Informell ist Slicing ein Verfahren, das diejenigen Klauseln des Programms ermittelt, die den Wert einer Variablen über ihre Parameter beeinflussen oder vom Wert dieser Variablen beeinflußt werden können. Den ersten Fall nennt man *Backward-Slicing*, den zweiten *Forward-Slicing*, entsprechend der Richtung des Datenflusses. Ferner kann man *statisches* und *dynamisches* Slicing unterscheiden. Im ersten Fall werden bei der Generierung einer Slice nur statische Informationen (zum Beispiel $Dep_{\mathbf{P}}$) verwendet. Bei der dynamischen Vorgehensweise werden die Informationen aus dem Ablauf des Programms gewonnen.

Das Konzept des Slicing stammt ursprünglich von Weiser [Wei84], und es existiert eine große Zahl verschiedener statischer und dynamischer Slicing-Techniken ([HRB90], [KL90] und [Kam93] speziell empfohlen sei [Tip94]).

Beide Slicing-Methoden haben Vor- und Nachteile:

- Statisches Slicing ist meistens nicht präzise, aber gültig für alle Programmabläufe.
- Dynamisches Slicing ist präzise, muß aber bei allen Aufrufen des Programms neu generiert werden.

Der (statische) Slicingteil von *IDTS* basiert auf den für attributierte Grammatiken ausgearbeiteten Techniken. Wir möchten hier nicht auf die Details attributierter Grammatiken eingehen. Eine ausgezeichnete Beschreibung kann unter anderem bei Knuth ([Knu68]), Deransart ([DJL88]) und Alblas ([AM91]) gefunden werden. Die Semantik für attributierte Grammatiken und PROLOG-Programme kann durch markierte Bäume definiert werden. Für attributierte Grammatiken wird ein attributierter *Ableitungsbaum (parse tree)* definiert. Bei der logischen Programmierung spricht man von einem für ein gegebenes Ziel generierten *Beweisbaum (proof tree)*. Hierbei entspricht die Markierung im ersten Fall den Instanzen von Attributen und im zweiten den Argumentwerten. Die genaue Erklärung des Formalismus ist in den Arbeiten von Deransart und Maluszynski ([DM93]) zu finden.

Um die hier eingeführten Definitionen zu veranschaulichen, nehmen wir ein einfaches PROLOG-Programm, das wir aus [Har96] entnommen haben.

**Beispiel 55** Ein einfaches PROLOG-Programm

```
a(X) :- b(X), c(X).
b(X) :- d(X), e(X), f(X).
c(X) :- g(X), b(X).
g(a).
d(a).
e(a).
f(a).
```

Wenn wir das im Beispiel 55 angegebene Programm mit dem Ziel `a(a)` aufrufen, bekommen wir den Ableitungsbaum in Abbildung 5.8.

Die von uns verwendete Backward-Slicing-Methode, die auf dem Programmabhängigkeitsgraphen aufbaut, kann wie folgt formuliert werden: Der Datenfluß realisiert sich in einem logischen Programm durch den lokalen Abhängigkeitsgraphen über die Terme, durch den globalen Abhängigkeitsgraphen zwischen verschiedenen Termen und durch den Verlängerungsalgorithmus.

Die *Annotierung* und die Generierung des *Abhängigkeitsgraphen* kann gleichzeitig durchgeführt werden. Der Programmabhängigkeitsgraph $Dep_{\mathbf{P}}$ ist ein Teil des globalen Programmabhängigkeitsgraphen, der für alle $\updownarrow$ und nicht annotierten Positionen eine doppelt gerichtete Kante enthält.

**Definition 84** Sei $C$ eine Klausel und bezeichnen wir mit $U(C)$ die doppelt gerichteten ($\updownarrow$) und nicht annotierten ($\square$) Positionen über $C$. Sei $\beta \in U(C)$ und $\gamma \in POS(C)$, dann kann der

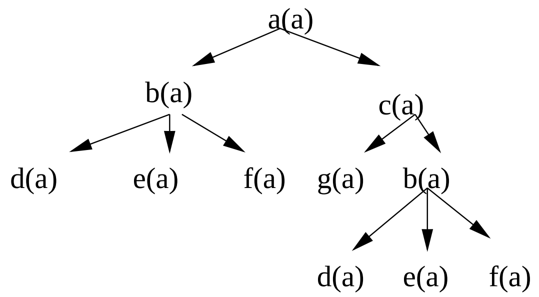



Abbildung 5.8: Der Ableitungsbaum zum Beispiel 55 mit dem Ziel `a(a)`

Abhängigkeitsgraph ($Dep_{\mathbf{P}}$) mittels der Relation $\leftrightarrow_C \subseteq POS(C) \times POS(C)$ wie folgt definiert werden:

$\beta \leftrightarrow_C \gamma$ und $\gamma \leftrightarrow_C \beta$ gilt genau dann, wenn $\beta$ und $\gamma$ gemeinsame Variablen haben.

Seien $C$ und $D$ zwei Klauseln und $B_0$ ein Atom im Kopf von $D$ und $A_j$ ein Atom im Körper von $C$, so daß $B_0$ und $A_j$ unifizierbar sind.
Seien $\beta = (C, j, q, k) \in U(C)$ und $\gamma = (D, 0, q, k) \in U(D)$. Dann hat $\leftrightarrow_{C,D} \subseteq U(C \times D) \times U(C \times D)$ die folgende Definition:

$\beta \leftrightarrow_{C,D} \gamma$ und $\gamma \leftrightarrow_{C,D} \beta$ gilt genau dann, wenn $\beta$ und $\gamma$ unifizierbar sind.

Der *globale Programmabhängigkeitsgraph* $Dep_{G\mathbf{P}}$ kann dann so definiert werden:

$$Dep_{G\mathbf{P}} = Dep_{\mathbf{P}} \cup \left( \bigcup_{C \in \mathbf{P}} \leftrightarrow_C \right) \cup \left( \bigcup_{C,D \in \mathbf{P}} \leftrightarrow_{C,D} \right)$$

Wenn wir einen Fehler im Programm suchen, interessieren wir uns nur für die Bestimmung der Klauseln, die das falsche Ergebnis verursachen. Dabei ist der Anfangspunkt dort, wo wir den falschen Wert identifizieren. Nach der Ausführung eines Backward-Slice bekommen wir einen reduzierten Baum, dessen Knoten aus dem ursprünglichen Ableitungsbaum stammen und eine Wirkung auf die Berechnung des falschen Wertes haben. Die Knoten, die keinen Einfluß auf die Berechnung haben, können weggelassen werden. Der Algorithmus kann sich so bei der Fehlersuche auf den reduzierten Baum beschränken.

**Definition 85** Sei $\mathbf{P}$ ein logisches Programm, $Dep_{G\mathbf{P}}$ sein globaler Programmabhängigkeitsgraph und $\mathbf{T}$ ein Ableitungsbaum mit einer Position $\gamma$. Eine Slice des Baums $\mathbf{T}$ auf $\gamma$ ist ein Baum, in dem gilt:

- Alle Knoten in der Slice sind Knoten in $\mathbf{T}$.
- Ein Knoten ist genau dann in der Slice, wenn er eine Position $\beta$ hat, so daß $\beta Dep^*_{G\mathbf{P}} \gamma$.

Es gibt zwei verschiedene Arten von Slicing:

- Slicing eines Programms und
- Slicing eines Ableitungsbaums.

Das Slicing eines Programms kann durch den globalen Abhängigkeitsgraphen des logischen Programms definiert werden. Als nächstes werden die Definitionen für *p*-, *pa*-, *PTree*-, *ATree*- und *NTree*-Slice eingeführt. Eine *pa*-Slice bestimmt einer Programmslice aus der Slice des Ableitungsbaums. Die *PTree*-Slice eines Programms wird mit Hilfe der *p*-Slice definiert. Diese Slice basiert auf einem annotierten Programm im Gegensatz zu der *ATree*-Slice, die einen annotierten Baum braucht, um die Slice berechnen zu können. Der Vorteil dieser Berechnung ist, daß die Slice genauer sein kann. Ein Nachteil ist aber, daß die benötigte Speicherkapazität groß sein kann, weil der Baum auch gespeichert werden muß. Eine Erweiterung ist die *NTree*-Slice, die statt eines annotierten Ableitungsbaums $Dep_{G\mathbf{P}}$ verwendet.

**Definition 86** Sei $\mathbf{P}$ ein logisches Programm, $Dep_{G\mathbf{P}}$ der globale Programmabhängigkeitsgraph, der zu dem annotierten Programm gehört. Mit $q$ und $r$ bezeichnen wir verschiedene Positionen in $Dep_{G\mathbf{P}}$. *p*-Slice des $Dep_{G\mathbf{P}}$ auf $r$ ist ein Teilgraph von $Dep_{G\mathbf{P}}$, so daß alle $q \in Dep_{G\mathbf{P}}$ in der *p*-Slice sind, für die ein Weg von $q$ nach $r$ existiert.

**Definition 87** Sei $s$ eine *p*-Slice von $\mathbf{P}$ auf Position $r$. Aus $s$ wird eine Menge $P'(s)$, eine Teilmenge der Atome in $\mathbf{P}$, bestimmt, so daß ein Atom $l$ genau dann in $P'(s)$ ist, wenn $l$ eine Position in $s$ hat.

**Definition 88** Seien $s_1$ und $s_2$ je eine *p*-Slice aus $\mathbf{P}$. Außerdem seien $P'(s_1)$ und $P'(s_2)$ die aus $s_1$ und $s_2$ gebildeten Mengen von Atomen $\{l$ Atom $|l$ besitzt eine Position aus der Slice$\}$. Die *p*-Slices $s_1$ und $s_2$ sind genau dann isomorph, wenn $P'(s_1) = P'(s_2)$. Weiterhin definieren wir $s_1 \leq s_2$ genau dann, wenn $P'(s_1) \subseteq P'(s_2)$. Insbesondere bedeutet dies, daß, wenn ein Teil des Programm verändert wurde, die davon abhängigen Teile nicht neu getestet werden müssen.

**Definition 89** Eine *PTree*-Slice zu der Position $\gamma$ in einem Knoten des Ableitungsbaums $\mathbf{T}$ ist ein Baum, in dem gilt:

- Alle Knoten der *PTree*-Slice sind Knoten in $\mathbf{T}$.
- Ein Knoten $n$ in $\mathbf{T}$ gehört genau dann zu der *PTree*-Slice, wenn $n$ eine Position $\beta$ enthält, die in der *p*-Slice zu $\gamma$ enthalten ist.

**Definition 90** Sei $\mathbf{T}$ der Ableitungsbaum von $\mathbf{P}$. Der annotierte Baum von $\mathbf{P}$ $\mathbf{T}_a$ kann wie folgt definiert werden:

- Die Knoten von $\mathbf{T}_a$ sind die gleichen wie die Knoten von $\mathbf{T}$.
- Sei $A_0$ ein Knoten in $\mathbf{T}$ mit Nachfolgern $A_1, \ldots, A_n$ $(n > 0)$ und $C : A'_0 :- A'_1, \ldots, A'_n$ eine entsprechende Klausel in $\mathbf{P}$. Außerdem sei $\beta$ eine Position in $A_i$ und $\gamma$ eine andere in $A_j$ $(0 \leq i, j \leq n)$. Die dazu gehörenden Positionen in $C$ bezeichnen wir mit $\beta' \in POS(C)$ und $\gamma' \in POS(C)$. Dann existiert genau dann eine direkte Kante von $\beta$ zu $\gamma$, $\beta \rightarrow \gamma$, in $\mathbf{T}_a$, wenn $\beta' \leftrightarrow^*_C \gamma'$. Die transitive Hülle von $\rightarrow$ wird mit $\rightarrow^*$ bezeichnet.

**Definition 91** Sei $\mathbf{T}_a$ der Ableitungsbaum des Programms $\mathbf{P}$ mit einer gekennzeichneten Position $\gamma$. Die *ATree*-Slice bezüglich $\gamma$ wird wie folgt definiert:

- Alle Knoten in *ATree* sind auch Knoten in $\mathbf{T}_a$.

- $n$ ist ein Knoten von $\mathbf{T}_a$. $n \in ATree$-Slice bezüglich $\gamma$ genau dann, wenn $n$ eine Position $\beta$ hat, so daß $\beta \leftrightarrow^* \gamma$.

**Definition 92** Sei $Dep_{G\mathbf{P}}$ der globale Programmabhängigkeitsgraph von $\mathbf{P}$. $\mathbf{T}_a$ ist ein annotierter Ableitungsbaum in $\mathbf{P}$ und $\gamma$ eine Position in $\mathbf{T}_a$. Eine *pa*-Slice über $Dep_{G\mathbf{P}}$ bezüglich $\gamma$ ist ein Teilgraph von $Dep_{G\mathbf{P}}$, so daß $q \in Dep_{G\mathbf{P}}$ genau dann in diesem Teilgraph ist, wenn es eine $q$-entsprechende Position $\beta$ im Baum gibt, so daß $\beta \leftrightarrow^* \gamma$.

Wenn wir die $p$-Slice mit der $pa$-Slice und die $ATree$-Slice mit der $PTree$-Slice vergleichen, können wir folgendes sagen.

**Satz 16** Sei $\mathbf{P}$ ein Programm mit $Dep_{G\mathbf{P}}$. $\mathbf{T}_a$ ist ein annotierter Baum und $\gamma$ eine Position in $\mathbf{T}_a$. Sei $pa(\gamma)$ die $pa$-Slice, $AT(\gamma)$ die $ATree$-Slice, $PT(\gamma)$ die $PTree$-Slice und $p(r)$ eine $p$-Slice, wobei $r$ eine Position im Programm ist, dann gilt:

- $pa(\gamma) \subseteq p(r_1) \cup p(r_2)$, wobei $r_1$ und $r_2$ die entsprechenden Programmpositionen für $\gamma$ sind.

- $AT(\gamma) \subseteq PT(\gamma)$

Wie wir schon erwähnt haben, braucht der bei der Berechnung der $ATree$-Slice verwendete Ableitungsbaum viel Speicherplatz. Deswegen wurde die $NTree$-Slice entwickelt, die den globalen Programmabhängigkeitsgraphen benutzt. Dieser Graph ist viel kleiner als der Baum aber die Berechnung der $ATree$-Slice ist schneller, da es hierbei einfacher ist, die Abhängigkeiten zu bestimmen. Um die $NTree$-Slice definieren zu können, brauchen wir die folgenden Hilfsdefinitionen.

**Definition 93** Seien $n$ und $m$ zwei Knoten in einem Ableitungsbaum $\mathbf{T}$. Der Weg von $n$ bis $m$ ist $m = n_1, n_2 \dots n_k = n$. Die $NR$-Reihe von $m$ bis $n$ ist $n_1, C_1, n_2, C_2, \dots C_{k-1}, n_k$, wobei $C_j (1 \leq j \leq k-1)$ Klauseln aus $\mathbf{P}$ bezeichnen, so daß gilt:

- Wenn $n_j$ ein Nachfolger von $n_{j+1}$ ist, dann ist $C_j$ eine auf $n_{j+1}$ angewandte Klausel.

- Wenn umgekehrt $n_{j+1}$ dem Knoten $n_j$ folgt, dann ist $C_j$ eine auf $n_j$ angewandte Klausel.

- Wenn $n_j$ und $n_{j+1}$ einen gemeinsamen Vorfahren $n_l$ haben, dann kennzeichnet $C_j$ die auf $n_l$ angewendete Klausel.

**Definition 94** Sei $\mathbf{P}$ ein Programm, $Dep_{G\mathbf{P}}$ der globale Abhängigkeitsgraph und $\mathbf{T}$ der Ableitungsbaum von $\mathbf{P}$. Seien $m$ und $n$ zwei Knoten in $\mathbf{T}$ und $p_{n_j}$ bezeichnet das der Position $n_j$ entsprechende Atom. Sei $n_1, C_1, \dots, C_{k-1}, n_k$ die von $m = n_1$ ausgehende $NR$-Reihe. Von $m$ bis $n$ ist eine *P-Reihe* definiert als eine Kantenreihe $a_1, b_1, a_2, b_2 \dots b_{k-2}, a_{k-1}$, so daß gilt:

- $a_1$ startet in der Programmposition $p_m$ und $a_1 \in \leftrightarrow^*_{C_1}$.

- $a_{k-1}$ endet in der Programmposition $p_n$ und $a_{k-1} \in \leftrightarrow^*_{C_{k-1}}$.

- Alle Kanten $a_j$ $(1 \leq j \leq k-1)$ starten bei der Position $r$ und enden in der Position $q$, wobei gilt:

  - $r \in POS(p_{n_j})$ und $q \in POS(p_{n_{j+1}})$

- Der Endpunkt $b_{j-1}$ ist $r$.
- $a_j \leftrightarrow^*_{C_j}$
- $p_{n_j}$ und $p_{n_{j+1}}$ sind Atome in der Klausel $C_j$

- Alle Kanten $b_j$ $(1 \leq j \leq k-2)$ starten bei der Position $r$ und enden in der Position $q$:
  - $r \in POS(P_{n_j})$ ist in Klausel $C_j$.
  - $r$ ist der Endpunkt von $a_j$.
  - $b_j \in\leftrightarrow^*_{u,v}$ wobei $u = C_j, v = C_{j+1}$, wenn $p_{n_j}$ das Kopfelement von $C_{j+1}$ ist, oder $u = C_{j+1}$ und $v = C_j$, wenn $p_{n_j}$ im Kopf von $C_j$ ein Atom ist

**Definition 95** Sei **T** ein annotierter Baum von **P** mit einer gekennzeichneten Position $\gamma$. Seien $n$ die Knoten, die $\gamma$ enthalten. Eine $Ntree$-Slice über **T** auf $\gamma$ ist ein Baum, in dem gilt:

- Alle Knoten von $Ntree$ sind Knoten in **T**.
- Ein Knoten $m$ aus **T** ist genau dann in $Ntree$, wenn eine $P$-Reihe aus $m$ in $n$ existiert, deren Endpunkt $\gamma$ ist.

**Satz 17** Sei **T** ein annotierter Ableitungsbaum mit einer gekennzeichneten Position $\gamma$ in **P**. Dann $NT(\gamma) = AT(\gamma)$, wobei $NT(\gamma)$ die $Ntree$-Slice und $AT(\gamma)$ eine ATree-Slice auf $\gamma$ ist.

### 5.1.3 Die Realisierung der Slicing-Technik

In diesem Kapitel werden die Algorithmen, die für die Generierung des globalen Programmabhängigkeitsgraphen $Dep_{G\mathbf{P}}$ notwendig sind, beschrieben. Der Graph wird dann später bei der Erzeugung der $p$-Slice und der $ATree$-Slice benutzt.

---

**Algorithmus 22** Die einfache Slicing-Methode

---

**Eingabe**: das Programm **P**, das Ziel **G**,
der Programmabhängigkeitsgraph $Dep_{\mathbf{P}}$,
die Liste der Anfangspunkte des Slicingverfahrens $\alpha$
**Ausgabe**: $p$-Slice
**procedure** $einfache\_slicing\_methode(Dep_{\mathbf{P}}, \alpha, \text{p}-Slice)$
**begin**
  **if** $\exists Dep_{\mathbf{P}}$ **then** read$(Dep_{\mathbf{P}}, File)$
  **else** $generate\_dependency\_graph(Dep_{\mathbf{P}}, \mathbf{P})$ **write**$(Dep_{\mathbf{P}}, File)$
  **endif**
  read$(\alpha)$;
  $p$-slice := $compute\_p\_slice(Dep_{\mathbf{P}}, \alpha)$;
  **write**($p$-Slice)
**end**

---

Der Algorithmus überprüft, ob der globale Programmabhängigkeitsgraph $Dep_{\mathbf{P}}$ schon existiert. Wenn notwendig, wird der Graph neu erzeugt (siehe $generate\_dependency\_graph(Dep_{\mathbf{P}}, \mathbf{P})$). Aus diesem Graphen und den Anfangspunkten des

Slicing-Verfahrens wird dann die $p$-Slice mit Hilfe des Algorithmus $compute\_p\_slice(Dep_{\mathbf{P}}, \alpha)$ berechnet.

Die Generierung des Programmabhängigkeitsgraphen ist eine verbesserte Version des Programms *GAPLOG* ([Ekl94]). Das Original-Programm kann aber die dualen Kanten nicht feststellen. Eine doppelt gerichtete Kante wird zwischen gleiche Variablen enthaltende, nicht annotierte lokale Positionen in einer Klausel oder vom statischen Programmabhängigkeitsgraphen bestimmten globalen Positionen, die noch nicht annotiert sind, eingefügt.

Die Erzeugung des globalen Programmabhängigkeitsgraphen wird in folgenden Schritten durchgeführt:

- Suche nach den funktionalen Positionen und den Variablenvorkommen.
- Konstruiere den statischen Aufrufgraphen.
- Erzeuge eine Annotation und den dazugehörenden *reduzierten* Abhängigkeitsgraphen.

Die Algorithmen in *GAPLOG* konstruieren die Annotation und den reduzierten Programmabhängigkeitsgraphen so, daß für alle Variablen in den Klauseln nur eine Eingabeposition ausgewählt wird.

**Definition 96** Der *reduzierte lokale Abhängigkeitsgraph* $Dep_{redC}$ ist ein Teilgraph des lokalen Abhängigkeitsgraphen $Dep_C$, wobei gilt:

- $\alpha Dep_{redC} \beta$ dann, wenn $\beta \in Funkpos(\mathbf{P})$ und $\alpha$ und $\beta$ wenigstens eine gemeinsame Variable besitzen und für alle gemeinsamen Variablen es nur die Position $\alpha$ gibt, so daß $\alpha Dep_{redC} \beta$.
- Zu jeder Variablen einer beliebigen Position $\beta$ gibt es eine Kante $\alpha Dep_{redC} \beta$ oder $\beta Dep_{redC} \alpha$, so daß die Variable auch in $\alpha$ auftritt.

**Definition 97** Der *reduzierte globale Abhängigkeitsgraph* $Dep_{redC,D}$ ist ein Teilgraph des globalen Abhängigkeitsgraphen $Dep_{C,D}$ (siehe Definition 76), wobei gilt: wenn gilt:

- Zwischen zwei Klauseln existiert höchstens eine Kante in $Dep_{redC,D}$.
- Jede Klausel ist Anfangs- oder Endpunkt mindestens einer Kante in $Dep_{redC,D}$.

**Definition 98** *Der reduzierte Programmabhängigkeitsgraph* kann wie folgt definiert werden:

$$Dep_{red\mathbf{P}} = \bigcup_{C \in \mathbf{P}} Dep_{redC} \cup \bigcup_{C,D \in \mathbf{P}} Dep_{redC,D}$$

Der folgende Algorithmus, *generate_dependency_graph* berechnet für alle funktionalen Ausdrücke $\delta \in Funkpos$ den reduzierten lokalen Abhängigkeitsgraphen.

**Algorithmus 23** Die Berechnung des reduzierten Programmabhängigkeitsgraphen $Dep_{redG\mathbf{P}}$

**Eingabe**: das Programm **P**
**Ausgabe**: der reduzierte Programmabhängigkeitsgraph $Dep_{red\mathbf{P}}$
**procedure** $generate\_dependency\_graph(Dep_{red\mathbf{P}},\mathbf{P})$
**begin**
  $Dep_{red\mathbf{P}} := \{\}$;
  find the list $Funkpos$ of functional expressions
  **while** $Funkpos \neq []$ or there is a not annotated position $\delta'$, **do**
    **if** $Funkpos \neq []$ **then** $\delta \in Funkpos$
    **else** $\delta := \delta'$
    **endif**
    $Dep\_Graph := \emptyset$;
    $compute\_local\_dep\_graph(\delta, Dep\_Graph)$;
    $Funkpos := Funkpos \backslash \delta$;
    $Dep_{red\mathbf{P}} = Dep_{red\mathbf{P}} \cup Dep\_Graph$
  **endwhile**
  **return**$(Dep_{red\mathbf{P}})$
**end**

Wenn $\beta = (C, i, p/n, k)$ eine Position im Kopf ist, dann wird $p/n =\uparrow$. Wenn $\alpha$ im Rumpf der Klausel steht, dann wird $p/n =\downarrow$.

Der Algorithmus *compute_local_dep_graph* sucht nach der Position $\alpha \in POS(C)$, die eine gemeinsame Variable mit $\beta$ hat und keine Ausgabeposition ist (es kann mehrere solche Positionen in einer Klausel geben, in solchen Situationen wird die am weitesten links stehende ausgewählt). Dann wird eine Kante ($\in Dep_{red\mathbf{P}}$) zwischen $\alpha$ und $\beta$ erzeugt und der Algorithmus *reduce_trans_graph* aufgerufen, um einen globalen Abhängigkeitsgraphen mit Parameter $\alpha$ zu berechnen.

**Algorithmus 24** Die Berechnung des lokalen Abhängigkeitsgraphen

**Eingabe:** $\beta = (C, i, p/n, k)$ eine Ausgabeposition, zu der wir Eingabepositionen suchen
*Dep_Graph* der schon berechnete Abhängigkeitsgraph
**Ausgabe:** *Dep_Graph* der vergrößerte Graph
**procedure** *compute_local_dep_graph*($\beta$, *Dep_Graph*)
**begin**
  Inputs := {}
  **if** $(i = 0)$ **then**
    **if** (not contradictory to the current annotation) **then** $\mu(p/n_k) :=\uparrow$
    **else** backtracking
    **endif**
  **else**
    **if** (not contradictory to the current annotation) **then** $\mu(p/n_k) :=\downarrow$
    **else** backtracking
    **endif**
  **endif**
  **for** $\forall v \in Var(\beta)$ **do**
    find leftmost $\alpha \in POS(C)$ where $v \in Var(\alpha)$ and $\alpha$ is no output position
    **if** $\alpha \mathrm{Dep}_{redC}\beta \notin$ *Dep_Graph* **then**
      Inputs := Inputs $\cup\{\alpha\}$;
      *Dep_Graph* := *Dep_Graph* $\cup\{\alpha \mathrm{Dep}_{redC}\beta\}$
    **endif**
  **endfor**
  **if** Inputs = {} and there is no edge $Dep_{redC,D}$ to $\beta$ **then**
    $\mu(p/n_k) := \Box$;
    *reduced_trans_graph*($\beta$, *Dep_Graph*)
  **endif**
  **while** Inputs $\neq$ {} **do**
    $\alpha \in$ Inputs;
    *reduced_trans_graph*($\alpha$, *Dep_Graph*);
    Inputs := Inputs $\setminus \alpha$
  **endwhile**
  **return**(*Dep_Graph*)
**end**

Der Algorithmus *reduced_trans_graph* kennzeichnet in einer Position $\beta = (C, i, p/n, k)$ den $k$-ten Parameter vom Prädikat $p/n$ mit $\uparrow$, wenn $\beta$ im Rumpf ist, sonst mit $\downarrow$. Wenn $\beta$ eine Position im Kopf ist, dann sucht der Algorithmus nach allen $\alpha = (D, j, p/n, l)$, wobei $D \to C$. Wenn $\beta$ im Rumpf ist, dann versucht der Algorithmus, alle Positionen $\alpha$ in den Klauseln $D$ zu finden, wobei $C \to D$ und $\alpha = (D, 0, p/n, l)$. Eine Kante zwischen $\alpha$ und $\beta$ wird erzeugt und zu $Dep_{\mathbf{P}}$ hinzugefügt. Danach wird der Algorithmus für den Aufbau des lokalen Abhängigkeitsgraphen *compute_local_dep_graph* aufgerufen.

**Algorithmus 25** Aufbau des globalen Abhängigkeitsgraphen

**Eingabe:** $\beta = (C, i, p/n, k)$ eine Eingabeposition, zu der wir Ausgabepositionen suchen
*Dep_Graph* der schon berechnete Abhängigkeitsgraph
**Ausgabe:** *Dep_Graph*, der vergrößerte Graph
**procedure** *reduced_trans_graph*($\beta$,*Dep_Graph* )
**begin**
  Outputs := {} ,
  **if** $(i = 0)$ **then**
    **if** (not contradictory to the current annotation) **then** $\mu(p/n_k) :=\downarrow$
    **else** backtracking
    **endif**
  **else**
    **if** (not contradictory to the current annotation) **then** $\mu(p/n_k) :=\uparrow$
    **else** backtracking
    **endif**
  **endif**
  **if** $i = 0$ **then**
    **for** $\forall D \mapsto C$ **do** $\alpha := (D, j, p/n, k)$
      Outputs := Outputs $\cup\alpha$
      **if** $\alpha Dep_{redCD}\beta \notin$ *Dep_Graph* **then** *Dep_Graph* := *Dep_Graph* $\cup(\alpha Dep_{redCD}\beta)$
      **endif**
    **endfor**
  **else**
    **for** $\forall C \mapsto D$ **do** $\alpha := (D, 0, p/n, k)$
      Outputs := Outputs $\cup\alpha$
      **if** $\alpha Dep_{redCD}\beta \notin$ *Dep_Graph* **then** *Dep_Graph* := *Dep_Graph* $\cup(\alpha Dep_{redCD}\beta)$
      **endif**
    **endfor**
  **endif**
  **while** Outputs $\neq$ {} **do**
    $\alpha \in$ Outputs;
    *compute_local_dep_graph*($\alpha$, *Dep_Graph*);
    Outputs := Outputs $\backslash\alpha$
  **endwhile**
  **return**(*Dep_Graph*)
  **end**

Der Algorithmus *compute_p_slice* verwendet als Anfangspunkte des Slicing-Verfahrens die durch den Benutzer gegebenen Anfangspositionen $\alpha$. Zu diesen Punkten sucht er im globalen Abhängigkeitsgraphen Positionen. Die Kanten, deren Endpunkte diesen Positionen entsprechen, werden in die *backward*-Liste eingetragen. Diejenigen Kanten aber, die als Anfangsknoten diese Punkte haben, werden in der *forward*-Liste ergänzt. Natürlich wird überprüft, ob die neu berechneten Kanten schon in der Liste sind oder nicht. Das Programm nimmt nur solche Kanten neu hinzu, die noch nicht in die Liste eingetragen wurden.

**Beispiel 56** Den berechneten reduzierten Programmabhängigkeitsgraphen des Beispiels 55

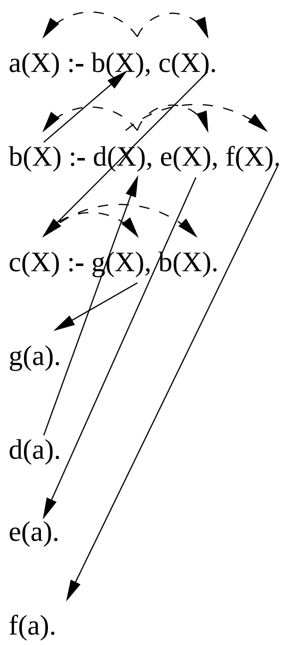



Abbildung 5.9: Der reduzierte Programmabhängigkeitsgraph des Beispiels 55

zeigt die Abbildung 5.9. Die globalen Abhängigkeiten sind mit durchgezogenen und die lokalen Abhängigkeiten mit gestrichelten Linien gekennzeichnet. Der Algorithmus beginnt mit dem Ziel `a(X)`. Dies führt zu einem unvollständigen Programmabhängigkeitsgraphen. Die Vervollständigung wird schrittweise durch den Hauptalgorithmus 23 erreicht, indem zusätzliche Teilgraphen für noch nicht annotierte Positionen konstruiert werden.

**Algorithmus 26** Die Berechnung des $p$-Slice

**Eingabe**: $\alpha$ die Anfangspunkte des Slicing-Verfahrens
der Abhängigkeitsgraph des Programms $Dep_{red\mathbf{P}} = Dep_{CD} \cup Dep_C$,
wobei $(C, D \in \mathbf{P})$
**Ausgabe**: $p$-Slice
**funktion** $compute\_p\_slice(Dep_{red\mathbf{P}}, \alpha)$: $p$-slice
**begin**
  forward := {}; backward := {}
  base_forward := $\{\alpha\}$; base_backward := $\{\alpha\}$
  **while** (base_forward $\neq$ {} **or** base_backward $\neq$ {}) **do**
    **if** base_backward $\neq$ {} **then**
      endpoint := find nodes with an edge to a node in base_backward using $Dep_{\mathbf{P}}$;
      where the node in base_backward is the endpoint
    **endif**
    **if** base_forward $\neq$ {} **then**
      beginpoint := find nodes with an edge from a node in base_forward using $Dep_{\mathbf{P}}$;
      where the node in base_forward is the beginpoint
    **endif**
    newendpoint:= endpoint $\setminus$ (backward $\cap$ endpoint);
    newbeginpoint := beginpoint $\setminus$ (forward $\cap$ beginpoint);
    forward := forward $\cup$ newbeginpoint;
    backward := backward $\cup$ newendpoint;
    base_forward := newbeginpoint;
    base_backward := newendpoint
  **endwhile**
  $p$-slice := (backward $\cup$ forward)
**end**

Das verbesserte Slicing-Verfahren, *improved_slicing_method* beruht auf dem $ATree - slice$. Bei dieser Methode wird nur die Backward-Slice berechnet. Es werden also diejenigen Positionen gesucht, von denen die vom Benutzer gegebenen Positionen abhängen.

**Algorithmus 27** Das verbesserte Slicing-Verfahren

**Eingabe**: das Programm **P**, das Ziel **G**
der Programmabhängigkeitsgraph $Dep_{\mathbf{P}}$
die Liste der Anfangspunkte des Slicing-Verfahrens $\alpha$
**Ausgabe**: $ATree$-Slice
**function** $improved\_slicing\_method(Dep_{red\mathbf{P}}, \alpha)$: $ATree$-slice
**begin**
  **if** $\exists Dep_{red\mathbf{P}}$ **then** read($Dep_{red\mathbf{P}}$,File)
  **else** $generate(Dep_{red\mathbf{P}})$ write($Dep_{red\mathbf{P}}$,File)
  **endif**
  $Dep_{redG\mathbf{P}} = Dep_{red\mathbf{P}} \cup$ build_dual(**P**);
  read(Goal) read($\alpha$);
  generate **T** proof tree;
  $ATree$-slice := $compute\_ATree\_slice(Dep_{redG\mathbf{P}}, \mathbf{T}, \alpha)$;
  write($ATree$-slice)
  **end**

Die Knoten des Slice-Baums sind die Knoten aus dem globalen reduzierten Programmabhängigkeitsgraphen, die im Ableitungsbaum auch vorkommen. So werden die Kanten weggelassen, die zwischen Klauseln stehen, die in der Ausführung des Zieles keine Rolle spielen.

**Algorithmus 28** Die Berechnung der $ATree$-Slice

**Eingabe**: $Dep_{redG\mathbf{P}}$ der globale reduzierte Programmabhängigkeitsgraph
**T** Ableitungsbaum
$\alpha$ die Liste der Anfangspunkte der Slice
**Ausgabe:** $ATree$-Slice, Knoten des Ableitungsbaums
**function** $compute\_ATree\_slice(Dep_{redG\mathbf{P}}, \mathbf{T}, \alpha)$: $ATree$-slice
**begin**
  backward := {}; base-backward := $\alpha$
  **while** base-backward $\neq$ {} **do**
    endpoint := find edge from base-backward using $Dep_{redG\mathbf{P}}$
    **for** $\forall e \in$ endpoint **do**
      **if** ($b \in$ base-backward **and** edge$(e, b) \notin \mathbf{T}$) **then** endpoint := endpoint $\setminus e$
      **endif**
    **endfor**
    newendpoint := endpoint - (backward $\cap$ endpoint);
    backward := backward $\cup$ newendpoint;
    base-backward := newendpoint
  **endwhile**
  $ATree$-Slice := backward
**end**

**Beispiel 57** Vergleichen wir die herausgeschnittenen Teilstücke der zwei verschiedenen Slice-Methoden: $pTree$-Slice (5.10) und $ATree$_slice .

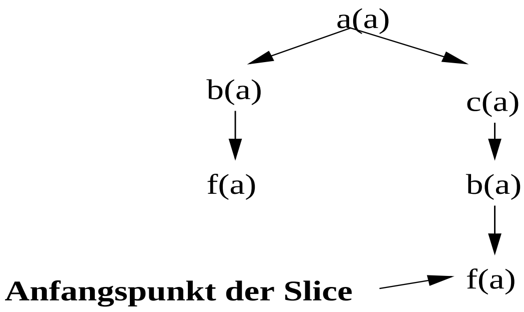



Abbildung 5.10: Eine *pTree*-Slice des Programms 55

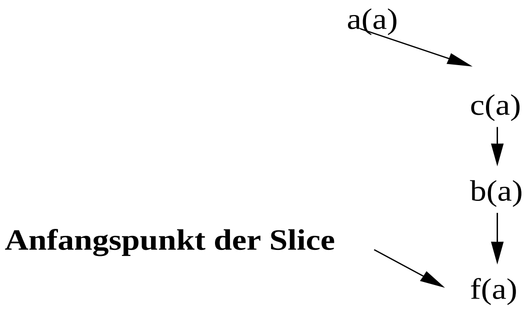



Abbildung 5.11: Ein *ATree*-Slice des Programms 55

Jetzt sind wir so weit, daß wir die Kombination der Original-Algorithmen von Shapiro im Fall inkorrekter Ausgabe mit Slicing beschreiben können. Die Vorgehensweise ist die gleiche, der wir auch bei der Beschreibung der Vereinigung von Shapiros Algorithmen und *CPM* gefolgt sind: Wir gehen bei allen drei Fällen der Traversierung des Baums (Bottom-Up, Top-Down, Divide-and-Query) auf die Algorithmen ein, die die Vorgehensweise beschreiben. Dann demonstrieren wir mit Hilfe des *compute*-Beispiels 8 die gestellten Fragen und zeigen den jeweiligen Ableitungsbaum.

Bei diesen erweiterten Methoden wird der Ableitungsbaum, dessen Wurzel $< C, x, y >$ ist, durchsucht. Bei allen Knoten wird eine Zugehörigkeitsfrage gestellt, wenn dieser Knoten in der Slice ist. Wenn ein Knoten in einer Slice ist, bedeutet das, daß dieser Knoten an der Berechnung der inkorrekten Ausgabe beteiligt war. Wenn eine Frage zu einem Knoten gestellt wird, der nicht in der *Backward-Slice* vorkommt, wird vom System automatisch mit *yes* geantwortet, weil dieser Knoten keine Wirkung auf die falsche Ausgabe hat.

Die Aufgabe des Anwenders ist die Eingabe eines Punktes im Programm, an dem er einen Fehler erkannt hat. Am Anfang des Slicing wird zuerst der Abhängigkeitsgraph statisch berechnet und beginnend mit dem vom Anwender gegebenen Punkt wird dann die Slice konstruiert. Zu der Berechnung des Abhängigkeitgraphen und der Slice werden die vorher beschriebenen Algorithmen `generate_dependency_graph` und `compute_ATree_slice` verwendet.

Im folgenden nehmen wir an, daß $C \in \mathbf{P}$ eine Klausel ist, wobei $< C, x, y > \notin \mathcal{I}$ für beliebige $x$ und $y$ ist und der vollständige Ableitungsbaum mit der Wurzel $< C, x, y >$ endlich ist.

## 5.2 Die modifizierte Bottom-Up-Methode

Wie die Originalemethode von Shapiro besucht unser Algorithmus den Ableitungsbaum in Postorder-Weise. Nehmen wir an, daß wir weiterhin die Frage, ob $< C_j, x_j, y_j > \in \mathcal{I}$ ist oder nicht, beantworten müssen, wobei $< C_j, x_j, y_j >$ ein Knoten im Ableitungsbaum ist. Im Fall Slicing überprüfen wir, ob $C_j$ in der Slice ist oder nicht. Wenn dies nicht der Fall ist, dann wird die Frage automatisch mit *yes* beantwortet und dementsprechend der Algorithmus weitergeführt.

**Algorithmus 29** Shapiro + Slice im *Bottom-Up*-Fall

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \bot$ und inkorrekt in $\mathcal{I}$ und
die Liste von Prädikaten, die in der Slice sind $ATree$_slice
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
ein Wahrheitswert für $found$,
wobei $found$ =*true*, wenn $C_{buggy}$ gefunden wurde, sonst $found$ =*false*

**procedure** *slice_bottom_up*($< C, x, y >$, $ATree$_slice, $C_{buggy}$, found)
**begin**
  let $\{< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, \ldots, < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$;
  $C_{buggy}$:= $\varepsilon$; $i$ := 1; found := false
  **while** ($i \leq n$) **and** (not found) **do**
    *slice_bottom_up*($< C_i, x_i, y_i >$, $ATree\_slice$, $C_{buggy}$, found);
    $i := i + 1$
  **endwhile**
  **if** not found **then**
    **if** $C \in ATree$_slice **then** answer := Query ($< C, x, y >$)
    **else** answer := true
    **endif**
    **if** answer = false **then**
      found := true;
      $C_{buggy} := (< C, x, y > \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >)$
    **endif**
    **if** answer = cancel **then**
      found := false
    **endif**
  **endif**
**end**

Nehmen wir an, daß $< C, x, y > \notin \mathcal{I}$ für ein $C \in \mathbf{P}, x \in D^n$ und $y \in D^m$. Der Algorithmus 29 nimmt das Tripel $< C, x, y >$ und das Ergebnis des Slicing-Verfahrens als Eingabe und gibt einen booleschen Wert $found$ und eine Instanz der fehlerhaften Klausel $C_{buggy}$ als Ausgabe zurück. $found$ =*true* bedeutet, daß $C_{buggy}$ den Fehler enthält. Der Algorithmus ruft sich selbst rekursiv mit den Elementen des *top level trace* auf, bis er entweder die falsche Klauselinstanz findet oder ein Blatt des Baums erreicht. Wenn ($found = false$) ist und der aktuelle Prädikatname der Klausel $C$ in der Slice ($C \in ATree\_slice$) ist, dann wird $Query(< C, x, y >)$ gefragt (siehe Algorithmus 4) sonst wird als Antwort *true* gegeben. Hier gilt auch, daß der Benutzer die vom Algorithmus gestellten Fragen immer korrekt beantworten muß.

**Algorithmus 30** Der Hauptalgorithmus des *Bottom-Up*-Verfahrens, wenn Slicing verwendet wird

**Eingabe**: das Programm $\mathbf{P}$, das Ziel $\mathbf{G}$, wobei $\mathbf{G}$ aus einer Klausel $C \in \mathbf{P}$ besteht
mit eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$, in der Form $\alpha = (c, a, p)$,
die Liste der Anfangspunkte des Slicing-Verfahrens $\alpha$,
wobei $c$ für die Klauselnummer steht,
$a$ gibt die Position des Atom an
(das im Kopf stehende Atom kennzeichnen wir mit 0) und
$p$ ist die Parameterposition in dem von $c$ und $a$ bestimmten Atom
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
procedure *IDTS_Debug_Slice*($< C, x, y >$, $\mathbf{P}$, $(c, a, p)$, $C_{buggy}$, found, $\alpha$)
**begin**
  *ATree*_slice = *improved_slicing_method*($Dep_{red\mathbf{P}}, \alpha$);
  *slice_bottom_up*($< C, x, y >$, *ATree*_slice, $C_{buggy}$, found)
  **if** (not found) **then**
    bottom_up($< C, x, y >$, $C_{buggy}$, found)
  **endif**
**end**

Der Hauptalgorithmus des *Bottom-Up*-Verfahrens, wenn Slicing verwendet wird, muß zuerst dafür sorgen, daß die Slice berechnet wird. Deswegen ruft er den Algorithmus *improved_slicing_method* auf. Mit dem ermittelten reduzierten Ableitungsbaum (dieser Baum enthält nur einen Teil des Originals, genau den, der in der Slice auch vorkommt) kann der Algorithmus *slice_bottom_up* aufgerufen werden. Wenn die falsche Klausel nicht gefunden wurde (*notfound*), dann kann der Fehler immer noch mit dem Aufruf der Originalmethode von Shapiro ermittelt werden.

Nehmen wir wie früher im Kapitel 4.3 an, daß $< C, x, y > \notin \mathcal{I}$. Sei $N = | < C, x, y > |$ die Anzahl der Knoten des vollständigen Ableitungsbaums, dessen Wurzel $< C, x, y >$ ist. Dann ist $0 \leq n \leq N$, wobei $n$ die Anzahl der Fragen ist, die vom Benutzer beantwortet werden müssen.

Im schlimmsten Fall kann der Algorithmus überhaupt nicht verbessert werden, weil die Slice den Ableitungsbaum nicht verkleinert hat, deswegen werden alle Fragen aus der Originalmethode von Shapiro auch hier gestellt.

Im besten Fall findet der Algorithmus die falsche Klausel genau so, daß eine Frage gestellt wird, weil sofort bei der ersten Frage, die aus der Slice kommt, die falsche Klausel gefunden wird.
In dem nächsten Beispiel 58 verwenden wir unser *compute*-Beispiel, um zu zeigen, welche Fragen, die durch Slicing verbesserte *Bottom-Up*-Methode stellt.

**Beispiel 58** Die bei der *Bottom-Up*-Methode gestellten Fragen, wenn Slicing benutzt wird

```
Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n)  y
Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n)  y
Is the goal [member( 2 , [2,3,4] )] correct? (y/n)  y
Is the goal [delete( 2 , [2,3,4] , [3,4] )] correct? (y/n)  y
Is the goal [member( 4 , [4] )] correct? (y/n)  y
```

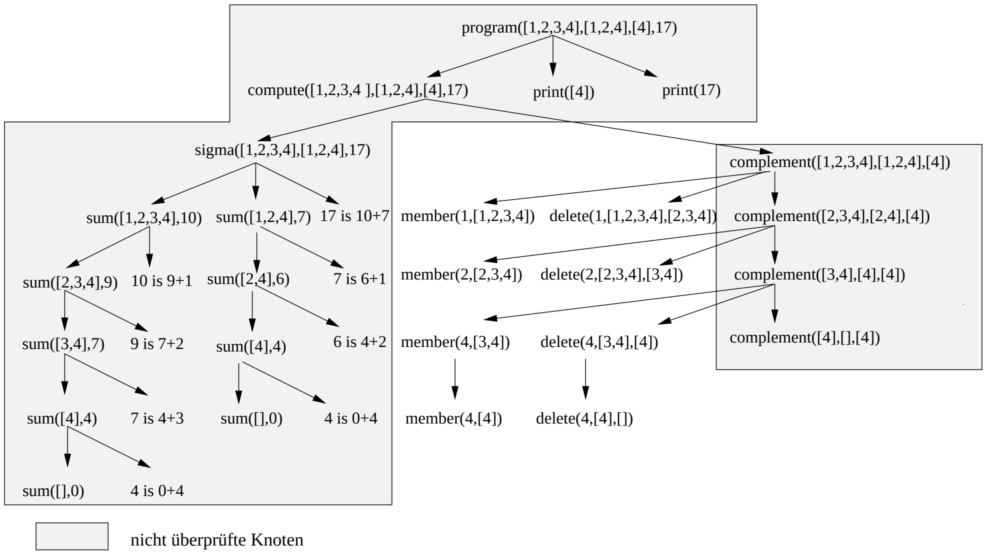



Abbildung 5.12: Die bei der mit Slicing kombinierten von Shapiro's *Bottom-Up*-Methode gestellten Fragen für das *compute*-Beispiel

```
Is the goal [member( 4 , [3,4] )] correct? (y/n)  y
Is the goal [delete( 4 , [4] , [] )] correct? (y/n)  y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n)  n
```

Die gestellten Fragen können auch graphisch dargestellt werden (siehe Abbildung 5.12).

## 5.3 Die modifizierte Top-Down-Methode

Die Vorgehensweise ist ähnlich wie bei der Kombination von Slicing mit dem *Bottom-Up*-Fall, nur wird hier der Ableitungsbaum *Top-Down* durchlaufen. Wir benutzen wieder eine modifizierte Version des Algorithmus 5.

**Algorithmus 31** Shapiro + Slice im *Top-Down*-Fall

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$ und
die Liste von Prädikaten, die in der Slice sind $ATree$_slice
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
ein Wahrheitswert $found$,
wobei $found$ =*true*, wenn $C_{buggy}$ gefunden wurde sonst $found$ =*false*
**procedure** $slice\_top\_down(< C, x, y >$,ATree_$slice, C_{buggy}, found)$
**begin**
  let $\{< C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$;
  $C_{buggy}$:= $\varepsilon$; $i$ :=1; found := false
  **while** $(i \leq n)$ **and** (not found) **do**
    **if** $C \in ATree$_slice **then** answer := $Query$ $(< C, x, y >)$
    **else** answer := true
    **endif**
    **if** answer = false **then**
    $slice\_top\_down(< C_i, x_i, y_i >$,ATree_$slice, C_{buggy}, found)$
    **else**
      **if** answer = true **then** i := i+1
      **else** found := true
      **endif**
    **endif**
  **endwhile**
  **if** not found **then**
    $C_{buggy}$ := $(< C, x, y > \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >)$;
    found := true
**end**

Der Algorithmus 31 arbeitet so, wie wir es schon im Kapitel 3.1.2 beschrieben haben: Es wird bei allen Knoten versucht, den falschen Kindknoten zu finden. Allerdings wird hier dem Benutzer nicht jedesmal eine Frage gestellt, sondern nur, wenn das im Knoten vorkommende Prädikat auch in der Slice vorkommt. Sonst wird die Frage nicht gestellt, sondern automatisch mit *yes* beantwortet.

Der Hauptalgorithmus ist fast genauso wie Algorithmus 30 aufgebaut. Es wird lediglich statt *slice_bottom_up* der Algorithmus *slice_top_down* aufgerufen.

Kommen wir nun wieder zu unserem Standardbeispiel *compute*. Im folgenden zeigen wir die gestellten Fragen. Der Algorithmus stellt elf Fragen, drei mehr als bei der *Bottom-Up*-Methode, wenn Slicing verwendet wird, und auch zwei Fragen mehr als bei der *Top-Down*-Methode, wenn das Original-Verfahren mit *CPM* kombiniert wird. Es zeigt sich, daß die Anzahl der Fragen auch stark davon abhängt, in welchem Teil des verkleinerten Baums der Fehler liegt.

**Beispiel 59** Die bei der *Top-Down*-Methode gestellten Fragen, wenn Slicing benutzt wird

```
Is the goal [compute( [1,2,3,4] , [1,2,4] , [4] , 17 )] correct? (y/n)  n
```

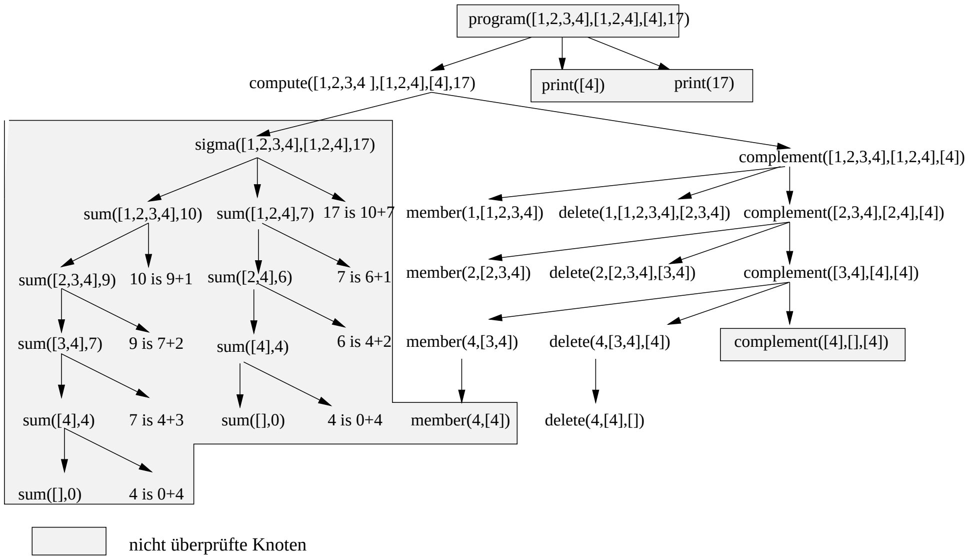



Abbildung 5.13: Die bei der mit Slicing kombinierten von Shapiro's *Top-Down*-Methode gestellten Fragen für das *compute*-Beispiel

```
Is the goal [complement( [1,2,3,4] , [1,2,4] , [4] )] correct? (y/n)  n
Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n)  y
Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n)  y
Is the goal [complement( [2,3,4] , [2,4] , [4] )] correct? (y/n)  n
Is the goal [member( 2 , [2,3,4] )] correct? (y/n)  y
Is the goal [delete( 2 , [2,3,4] , [3,4] )] correct? (y/n)  y
Is the goal [complement( [3,4] , [4] , [4] )] correct? (y/n)  n
Is the goal [member( 4 , [3,4] )] correct? (y/n)  y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n)  n
Is the goal [delete( 4 , [4] , [] )] correct? (y/n)  y
```

Die Vorgehensweise des Algorithmus kann man auch graphisch mit Hilfe der Abbildung 5.13 veranschaulichen.

## 5.4 Die modifizierte Divide-and-Query-Methode

Die dritte Möglichkeit ist, wenn das *Divide-and-Query*-Verfahren von Shapiro mit Slicing kombiniert wird. Wir nehmen den beim *Divide-and-Query* verwendeten Algorithmus 9 als Anfangspunkt.

**Algorithmus 32** Shapiro + Slice im *Divide-and-Query*-Fall

**Eingabe:** eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ $y$ zurückgibt,
wobei $y \neq \bot$ und inkorrekt in $\mathcal{I}$,
$\mathcal{I}' \subset \mathcal{I}$, wobei $\mathcal{I}'$ leer sein kann,
das Gewicht des zu $C$ gehörenden Ableitungsbaums *weight* und
die Liste von Prädikaten, die in der Slice sind *ATree*_slice
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel
**procedure** $slice\_divide\_query(<C, x, y>, \mathcal{I}'$, weight, $C_{buggy}$, $ATree$_slice)
**begin**
  middle_weight:=0; $C_{middle}$:=$\varepsilon$
  **while** weight $\neq 1$ **do**
    $middle\_point(<C, x, y>, C_{middle}$, middle_weight,weight)
    **if** $C \in ATree$_slice **then** answer := Query $(<C, x, y>)$
    **else** answer := true
    **if** answer = yes **then**
      $slice\_divide\_query(<C, x, y>, \mathcal{I}' \cup C_{middle}$,weight−middle_weight,
        $C_{buggy}$, $ATree$_slice)
    **else**
      **if** answer = no **then**
        $slice\_divide\_query(C_{middle}, \mathcal{I}'$, middle_weight,$C_{buggy}$, $ATree$_slice)
      **else**
      $C_{buggy}$:=$\varepsilon$ **return**
      **endif**
    **endif**
  **endwhile**
  $C_{buggy} := C_{middle} : - <C_1, x_1, y_1>, \ldots, <C_n, x_n, y_n>,$
**end**

Der Algorithmus 32 arbeitet so, wie wir es schon beim Algorithmus 9 beschrieben haben: Er versucht rekursiv die falsche Klausel zu finden, wobei er das schon vorher für den ganzen Baum berechnete Gewicht *weight* verwendet. Mit Hilfe dieses Werts wird das Gewicht des mittleren Knotens und der übrigen Knoten bestimmt. Der Unterschied zwischen den zwei Algorithmen liegt darin, daß Algorithmus 9 immer für den mittleren Knoten dem Anwender eine Frage stellt. Der Algorithmus 32 prüft dagegen zuerst, ob er das im Knoten stehenden Prädikat in der Slice-Liste (*ATree*_slice) gefunden hat, sonst wird die Frage automatisch mit *correct* bewertet. Entsprechend der Antwort geht der Algorithmus weiter:

- *answer = yes*: Der Fehler liegt in dem Teilbaum, dessen Wurzel der gerade gefragte Knoten ist, `slice_divide_query` wird mit diesem Knoten als Wurzel und mit `middle_weight` als Gewicht aufgerufen.

- *answer = no*: `slice_divide_query` wird rekursiv mit der gleichen Wurzel aber mit der Differenz des Original-Gewichts und des mittleren Gewichts als Gewicht *weight* aufgerufen.

Die Kombination des Slicing-Verfahrens mit der *Divide-and-Query*-Methode zeigt der

Hauptalgorithmus 33. Der Algorithmus erledigt zwei wichtige Aufgaben:

- Er berechnet die Slice: *ATree*-slice.
- Er ruft die Prozedur *compute_weight* auf, die das Gewicht des ursprünglichen Ableitungsbaums *weight* liefert. Damit arbeitet dann das Verfahren *slice_divide_query*.

---

**Algorithmus 33** Der Hauptalgorithmus des *Divide-and-Query*-Verfahrens, wenn Slicing verwendet wird

---

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
$\mathcal{I}' \subset \mathcal{I}$
die Liste der Anfangspunkte des Slicing Verfahrens $\alpha$, in der Form $\alpha = (c, a, p)$,
wobei $c$ für die Klauselnummer steht,
$a$ gibt die Position des Atom an
(das im Kopf stehende Atom kennzeichnen wir mit 0) und
$p$ ist die Parameterposition in dem von $c$ und $a$ bestimmten Atom
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel
**procedure** $IDTS\_Debug\_divide\_and\_query\_Slice(< C, x, y >, \mathbf{P}, C_{buggy}, \mathcal{I}', \alpha)$
**begin**
  $ATree$_slice $= improved\_slicing\_method(Dep_{red\mathbf{P}}, \alpha)$;
  $compute\_weight(< C, x, y >, \mathcal{I}', weight)$;
  $slice\_divide\_query(< C, x, y >, \mathcal{I}'$, weight, $C_{buggy}$, $ATree$_slice)
**end**

---

Um den Vergleich zwischen den beiden Methoden *Original-Divide-and-Query* und der Kombination mit *Slice* zu zeigen, betrachten wir wieder das Beispiel 8 und nehmen das gleiche Ziel:

```
program([1,2,3,4],[1,2,4],C,S).
```

Verwendet man die Original-Divide-and-Query-Methode, so sind fünf Fragen zu beantworten. Im Fall der Kombination mit *CPM* werden auch fünf Fragen gestellt. Slicing bringt auch keine Verbesserung, es müssen wieder fünf Fragen beantwortet werden. Dies kommt vor, weil schon der Original-Algorithmus nur zu dem Teil des Ableitungsbaums Fragen stellt, der auch im Slicing-Baum ist. Weil gleich viele Fragen gestellt werden, ist die Abbildung des Ableitungsbaums die gleiche wie in Abbildung 3.5.

**Beispiel 60** Die bei der *Divide-and-Query*-Methode gestellten Fragen, wenn Slicing benutzt wird

```
Is the goal [complement( [1,2,3,4] , [1,2,4] , [4] )] correct? (y/n) n
Is the goal [complement( [3,4] , [4] , [4] )] correct? (y/n) n
Is the goal [member( 4 , [3,4] )] correct? (y/n) y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
```

## 5.5 Zusammenfassung

In diesem Kapitel haben wir uns mit dem *Slicing*-Verfahren beschäftigt. Das Slicing-Verfahren ist eine allgemeine Methode, mit der versucht wird, die Elemente zu finden, die vermutlich den Fehler enthalten. Unser System verwendet *Backward-Slicing*. Das bedeutet, daß die Programmslice mit einer bestimmten Variablen die Teile des Programms enthält, die eine Wirkung auf diese Variable haben können.

Um die Slice berechnen zu können, müssen wir den Programmabhängigkeitsgraphen ermitteln, der die möglichen Datenflußkanten enthält,. Dazu muß zuerst eine Analyse des Quellprogramms durchgeführt werden. Dies geschieht statisch. Weiterhin ist der zweite Vorbereitungsschritt die Konstruktion des Ableitungsbaums anhand eines gegebenen Ziels. Schon bei der Generierung dieses Baums kann man die Richtung des Datenflusses feststellen und die sogenannten *Ein-* und *Ausgabepositionen* bestimmen.

Diesen Vorbereitungsschritten folgt das eigentliche Slicing. Das Verfahren ist rekursiv, startet aus der vom Benutzer festgestellten fehlerhaften Position und prüft mit Hilfe des Programmabhängigkeitsgraphen, von welchen Argumenten aus diese Position erreicht werden kann.

Zuerst wurden die theoretischen Grundlagen des Slicing erläutert. Bei der Beschreibung der statischen Analyse haben wir die Ähnlichkeiten, die zwischen attributierten Grammatiken und logischer Programmierung existieren, ausgenutzt, um die schon bei attributierten Grammatiken bekannten Techniken an die logische Programmierung anzupassen.

Im zweiten Teil des Kapitels haben wir die Kombination des Original-Algorithmus von Shapiro im Fall inkorrekter Ausgabe mit Slicing beschrieben und durch das *compute*-Beispiel demonstriert. Wir haben alle drei Traversierungsarten des Baums (Bottom-Up, Top-Down, Divide-and-Query) um Slicing erweitert.

Im nächsten Kapitel versuchen wir, das in diesem Kapitel erklärte Slicing und die im vorherigen Kapitel eingeführte *CPM*-Methode mit den Original-Algorithmen zu vereinigen.

# Kapitel 6

# Interactive Debugging, Testing and Slicing System

In den vorherigen Kapiteln 4 und 5 haben wir die Category Partition Methode und das Slicing beschrieben und angegeben, wie sich diese Techniken getrennt voneinander mit den Diagnosealgorithmen von Shapiro kombinieren lassen. In diesem Kapitel werden wir alle drei Methoden vereinigen und in das *IDTS*-System einbauen. Zuerst geben wir die Kombination von *CPM*, dem Slicing und den Original-Algorithmen für die drei Fälle inkorrekter Ausgabe an. Es folgt als Zusammenfassung eine Beschreibung des Gesamtsystems. Danach demonstrieren wir mit Hilfe von drei Beispielen (*compute*, *isort* and *convert*), wie sich die Anzahl der dem Anwender gestellten Fragen ändert, wenn die verschiedenen Techniken verwendet werden. Im Rest des Kapitels findet sich eine Einführung in die graphische Bedienoberfläche *GIDTS*, die zu *IDTS* entwickelt wird, um das System benutzerfreundlich und leistungsfähiger zu machen.

## 6.1 Kombination aller drei Methoden im Fall inkorrekter Ausgabe

In diesem Kapitel wird die Kombination der drei verschiedenen Methoden für die Diagnose der falschen Klauseln im Fall inkorrekter Ausgabe mit der *CPM*-Methode und Slicing vorgestellt.

Wie wir wissen, wird bei diesen Methoden der Ableitungsbaum, dessen Wurzel $< C, x, y >$ ist, durchsucht, und bei allen Knoten wird eine Zugehörigkeitsfrage gestellt. Für diese Fragen wird zuerst überprüft, ob sie in der Slice sind oder nicht. Wenn das nicht der Fall ist, werden sie automatisch mit *true* beantwortet. Wenn die Knoten dagegen bei der Berechnung eines falschen Ergebnisses eine Rolle spielen, muß entweder der Anwender die gestellte Frage beantworten, oder die Bewertung wird aus der Testdatenbank genommen.

Um *CPM* verwenden zu können, nehmen wir an, daß eine *CPM*-Testkonfiguration für das Programm **P** existiert, daß für einige $C \in \mathbf{P}$ eine Menge von Wahlmöglichkeiten definiert wurde, daß einige Testrahmen getestet wurden und daß deswegen eine Bewertung vorliegt. Außerdem haben wir die folgenden Voraussetzungen: $< C, x, y > \notin \mathcal{I}$ für beliebige $x$ und $y$, und der vollständige Ableitungsbaum mit der Wurzel $< C, x, y >$ ist endlich. In diesem Kapitel werden die vorher definierten Algorithmen für die Abfrage der Datenbank (`Query_database`, Algorithmus 17), die Veränderung der Datenbank (`handle_database`, Algorithmus 18), Berechnung des Abhängigkeitsgraphen (`generate_dependency_graph`, Algorithmus 23) und die Erzeugung einer *ATree*-Slice (`compute_ATree_slice`, Algorithmus 28) verwendet.

### 6.1.1 Kombination aller drei Methoden im Bottom-Up-Fall

Wie die Original-Methode von Shapiro durchläuft unser Algorithmus den Ableitungsbaum in Postorder-Weise. Nehmen wir an, daß wir weiterhin die Frage, ob $< C_j, x_j, y_j > \in \mathcal{I}$ ist oder nicht, beantworten müssen, wobei $< C_j, x_j, y_j >$ ein Knoten im Ableitungsbaum ist. Im Fall *CPM&Slicing* überprüfen wir, ob das Prädikat im Knoten zu der Slice gehört ($C \in ATree$_slice). Wenn nicht, dann ist die Antwort *true*. Wenn doch, dann wird als nächstes ermittelt, ob der Testrahmen zu $C_j$ und $x_j$ schon existiert und ob er schon eine Bewertung hat. Wenn dieser Testrahmen schon bewertet wurde, dann wird das Ergebnis der Bewertung bei der Antwort der Zugehörigkeitsfrage benutzt und das neue Paar $(x_j, y_j)$ eingefügt. Wenn der Testrahmen existiert, aber keine Bewertung hat und die Parametereinstellung es erlaubt, dann wird dem Benutzer die Frage gestellt, welche Bewertung dieser Testrahmen bekommt. Die Antwort des Anwenders und das Paar $(x_j, y_j)$ werden in die Datenbank eingefügt. Wenn der Testrahmen nicht existiert, wird nur der Anwender gefragt.

**Algorithmus 34** Shapiro + *CPM* + Slice im *Bottom-Up*-Fall

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, wo der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei *modus* = *step* in jedem Schritt eine Frage stellt und
*modus* = *skip* nur bei Elementen der *debug_points* Liste eine Frage stellt,
die Liste von Prädikaten, die in der Slice sind *ATree*_slice

**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
ein Wahrheitswert *found*,
wobei *found* =*true*, wenn $C_{buggy}$ gefunden wurde, sonst *found* =*false*,
eine modifizierte Datenbank $\mathbf{D}$,
modifizierter *modus*

**procedure** *CPM_slice_bottom_up*($< C, x, y >, \mathbf{D}$, debug_points,
modus, *ATree*_slice, $C_{buggy}$, found)
**begin**
  let $\{< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, \ldots, < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$;
  $C_{buggy}$:=$\varepsilon$; $i$:=1; found:=false
  **while** ($i \leq n$) **and** (not found)
    *CPM_slice_bottom_up*($< C_i, x_i, y_i >, \mathbf{D}$, debug_points,
    modus, *ATree*_slice, $C_{buggy}$ , found);
    $i := i + 1$
  **endwhile**
  **if** not found **then**
    **if** $C \in$ *ATree*_slice **then**
      **if** $\neg\Sigma_C$ **then** answer := *Query* ($< C, x, y >$)
      **else** answer := *Query_database*($< C, x, y >, \mathbf{D}$, debug_points, modus)
      **endif**
    **else** answer := true
    **endif**
    **if** answer = false **then**
      found := true;
      $C_{buggy} := (< C, x, y > \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >)$
    **endif**
    **if** answer = cancel **then**
      found := true
    **endif**
  **endif**
**end**

Der Algorithmus 34 nimmt das Tripel $< C, x, y >$ als Eingabe, wobei $< C, x, y > \notin \mathcal{I}$ für ein $C \in \mathbf{P}, x \in D^n$ und $y \in D^m$ ist, und gibt als Ausgabe einen booleschen Wert *found*, eine Instanz $C_{buggy}$ der fehlerhaften Klausel und die modifizierte Datenbank $\mathbf{D}$ zurück. Seien

$\{< C_1, x_1, y_1 >, < C_2, x_2, y_2 >, \ldots, < C_n, x_n, y_n >\}$ die direkten Nachkommen von $< C, x, y >$ (*top level trace*). Der Algorithmus ruft sich solange rekursiv selbst mit den Elementen des *top level trace* auf, bis er entweder die falsche Klauselinstanz findet ($found = true$) oder ein Blatt des Baums erreicht ($i = n$).

Wenn ($found = false$) ist und der aktuellen Prädikatname der Klausel $C$ in der Slice ist, außerdem keine Testspezifikation $\Sigma_C$ eingegeben wurde oder der Testrahmen keine Bewertung hat, dann wird $Query(< C, x, y >)$ gefragt (siehe Algorithmus 4). Sonst wird die Bewertung aus der Datenbank genommen oder, es wird, wenn $C \notin ATree$-Slice, die Frage mit *true* beantwortet. Abhängig von *answer* wird der Algorithmus weitergeführt.

**Algorithmus 35** Der Hauptalgorithmus des *Bottom-Up*-Verfahrens, wenn die Testdatenbank und Slicing verwendet werden

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei $modus = step$ in jedem Schritt eine Frage stellt und
$modus = skip$ nur bei Elementen der *debug_points* Liste eine Frage stellt,
die Liste der Anfangspunkte des Slicing-Verfahrens $\alpha$
in der Form $\alpha = (c, a, p)$, wobei $c$ für die Klauselnummer steht,
$a$ gibt die Position des Atom an
(das im Kopf stehende Atom kennzeichnen wir mit 0),
$p$ ist die Parameterposition in dem von $c$ und $a$ bestimmten Atom
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel
**procedure** IDTS_Debug_CPM_Slice($< C, x, y >$, $\mathbf{D}$,(c, a, p), $C_{buggy}$, debug_points,
  modus, found, $\alpha$)
**begin**
  $\mathbf{D} = T(F(\sum_{\mathbf{P}}))$;
  save($T(F(\sum_{\mathbf{P}}))$);
  $ATree$_slice = *improved_slicing_method*($Dep_{red\mathbf{P}}, \alpha$)
  *CPM_slice_bottom_up*($< C, x, y >$, $\mathbf{D}$, $C_{buggy}$, debug_points, modus, found);
  **if** (found and $C_{buggy} = \varepsilon$) **then**
    restore($T(F(\sum_{\mathbf{P}}))$);
  **endif**
  **if** (not found) **then**
    bottom_up($< C, x, y >$, $C_{buggy}$, found)
  **endif**
**end**

Der Hauptalgorithmus muß dafür sorgen, daß die Slice berechnet wird. Außerdem wird der Fall behandelt, daß der Anwender, statt *yes/no*-Antworten zu geben, mit *cancel* geantwortet hat. Das bedeutet, daß der Debuggingprozeß beendet wird und auch die bisherigen Einträge in der Datenbank ($T(F(\sum_{\mathbf{P}}))$) gelöscht werden müssen. Dafür sorgt der Algorithmus

*IDTS_Debug_CPM_Slice*, der *CPM_slice_bottom_up* aufruft. Danach überprüft der Algorithmus den Wert von *found*:

- $found = true$ und $C_{buggy} = \varepsilon$: Der Zustand der Datenbank vor Beginn des Debuggings muß wiederhergestellt werden.
- $found = false$: *CPM_slice_bottom_up* konnte keine falsche Klauselinstanz finden, die Original-Methode von Shapiro wird aufgerufen.

In dem Beispiel 61 verwenden wir unser *compute*-Beispiel, um zu zeigen, welche Fragen die mit Slicing und *CPM* verbesserte *Bottom-Up*-Methode stellt.

**Beispiel 61** Die bei der *Bottom-Up*-Methode gestellten Fragen, wenn Slicing und *CPM* benutzt werden

```
Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n) y
Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n) y
Is the goal [member( 4 , [4] )] correct? (y/n) y
Is the goal [member( 4 , [3,4] )] correct? (y/n) y
Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
```

**Beispiel 62** Vergleich der Anzahl der gestellten Fragen durch graphische Darstellung des Ableitungsbaums

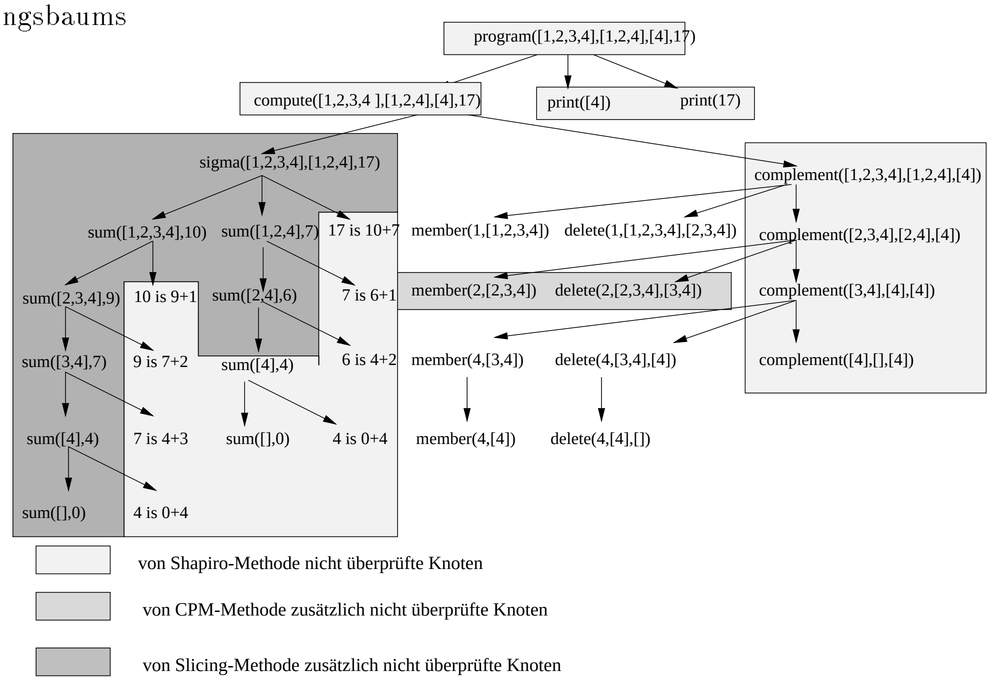



Abbildung 6.1: Die von der Kombination aus *CPM*, Slice und von Shapiro's *Bottom-Up*-Methode gestellten Fragen für das *compute*-Beispiel

Weil wir auch die Testdatenbank verwendet haben, werden in dieser genau die Änderungen durchgeführt, die wir in der Abbildung 38 gezeigt haben, deswegen wiederholen wir sie hier nicht.

Die Abbildung 6.1 veranschaulicht graphisch, wie die verschiedenen Methoden die Anzahl der Fragen verringert haben. Mit der Kombination der drei Methoden haben wir im Vergleich zu den von der Original-Methode gestellten vierzehn Fragen erreicht, daß wir den Fehler mit sechs Fragen finden können. Dieses Ergebnis ist eindeutig besser, als wenn wir nur die *CPM*-Methode (vierzehn Fragen) oder nur Slicing (acht Fragen) verwenden.

### 6.1.2 Kombination aller drei Methoden im Top-Down-Fall

Die Vorgehensweise ist ähnlich der, die wir bei der Kombination von *CPM* und Slice mit dem *Bottom-Up*-Fall beschrieben haben, nur wird hier der Ableitungsbaum *Top-Down* durchlaufen. Wir geben eine modifizierte Version des Original-Algorithmus 5 an.

**Algorithmus 36** Shapiro + *CPM* + Slice im *Top-Down*-Fall

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei *modus* = *step* in jedem Schritt eine Frage stellt
*modus* = *skip* nur bei Elementen der *debug_points* Liste eine Frage stellt und
die Liste von Prädikaten, die in der Slice sind *ATree*_slice
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
ein Wahrheitswert *found*,
wobei *found* =*true*, wenn $C_{buggy}$ gefunden wurde, sonst *found* =*false*,
eine modifizierte Datenbank $\mathbf{D}$,
modifizierter *modus*

**procedure** *CPM_slice_top_down*($< C, x, y >$, $\mathbf{D}$, debug_points, modus, $C_{buggy}$,
found, *ATree*_slice)
**begin**
let $\{< C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$;
$C_{buggy}$:=$\varepsilon$; $i$:=1; found:=false
**while** ($i \leq n$) **and** (not found) **do**
**if** $C \in$ *ATree*_slice **then**
**if** $\neg\Sigma_C$ **then** answer := *Query* ($< C, x, y >$)
**else** answer := *Query_database*($< C, x, y >$, $\mathbf{D}$, debug_points, modus)
**endif**
**else** answer := true
**endif**
**if** answer = false **then**
*CPM_slice_top_down*($< C_i, x_i, y_i >$, $\mathbf{D}$, debug_points, modus, $C_{buggy}$, found, *ATree*_slice )
**else**
**if** answer = true **then** $i := i + 1$
**else** found := true
**endif**
**endif**
**endwhile**
**if** not found **then**
$C_{buggy} := (< C, x, y > \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >)$;
found := true
**endif**
**end**

Der Algorithmus 36 versucht rekursiv von den Elternknoten ausgehend den falschen Kindknoten zu finden. Dem Benutzer wird jedesmal eine Frage gestellt, wenn das Prädikat im aktuellen Knoten in der Datenbank ist, aber die Antwort auf die Frage noch nicht in der Testdatenbank enthalten ist oder es keine Testbeschreibung zum Prädikat gibt.

Der Hauptalgorithmus ist fast genauso wie Algorithmus 35. Es wird aber statt *CPM_slice_bottom_up* der Algorithmus *CPM_slice_top_down* aufgerufen.

Überprüfen wir, wie wir mit der Kombination der drei Techniken in unserem *compute*-Beispiel den Fehler finden können. Die ursprüngliche *Top-Down*-Methode stellt zwölf Fragen, die Verbesserung mit *CPM* braucht neu Fragen, Slicing kommt mit elf Fragen aus, aber das beste Ergebnis erzielen wir auch hier, wenn wir alle drei Methoden kombinieren: Der Anwender muß acht Fragen beantworten. Im folgenden zeigen wir die gestellten Fragen (siehe Beispiel 63).

**Beispiel 63** Die bei der *Top-Down*-Methode gestellten Fragen, wenn Slicing und *CPM* benutzt werden

```
Is the goal [compute( [1,2,3,4] , [1,2,4] , [4] , 17 )] correct? (y/n) n
Is the goal [complement( [1,2,3,4] , [1,2,4] , [4] )] correct? (y/n) n
Is the goal [member( 1 , [1,2,3,4] )] correct? (y/n) y
Is the goal [delete( 1 , [1,2,3,4] , [2,3,4] )] correct? (y/n) y
Is the goal [complement( [3,4] , [4] , [4] )] correct? (y/n) n
Is the goal [member( 4 , [3,4] )] correct? (y/n) y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
```

Die durchgeführten Änderungen in der Testdatenbank im *Top-Down*-Fall, wenn Slicing und *CPM* benutzt werden, sind die gleichen wie für den Fall, daß nur *CPM* verwendet wurde (siehe Beispiel 43).
Die Abbildung 6.2 zeigt, wie die einzelnen Methoden die Anzahl der Fragen beeinflussen.

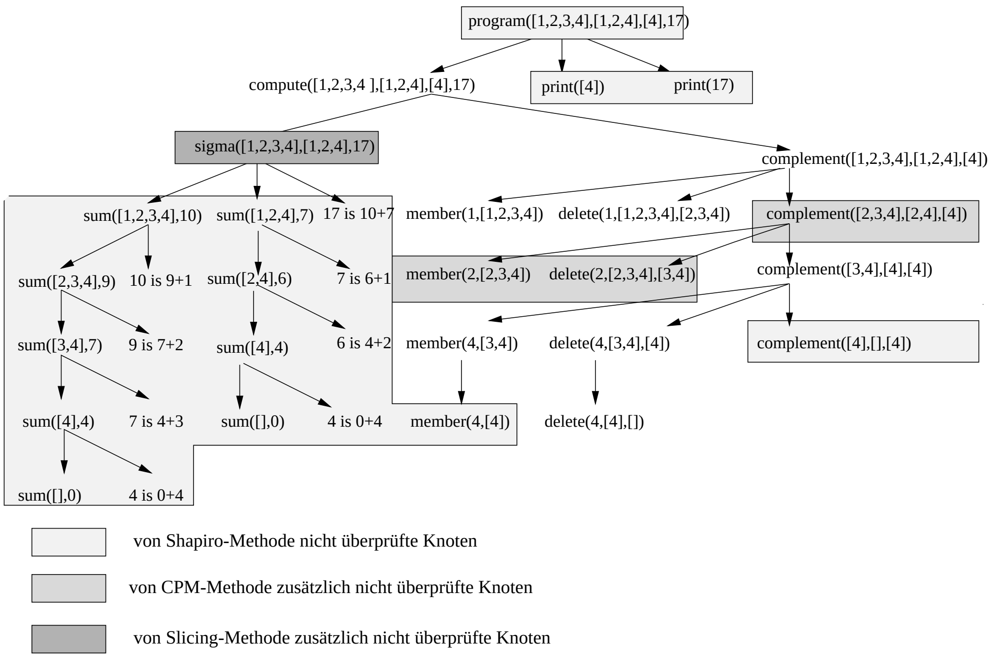



Abbildung 6.2: Die von der Kombination aus *CPM*, Slice und *Top-Down*-Methode gestellten Fragen für das *compute*-Beispiel

### 6.1.3 Kombination aller drei Methoden im Divide-and-Query-Fall

Die dritte Möglichkeit ist, wenn im Fall *Divide-and-Query* aller drei Methoden kombiniert werden. Wieder nehmen wir den beim *Divide-and-Query* verwendeten Algorithmus 9 als Anfangspunkt.

**Algorithmus 37** Shapiro + *CPM* + Slice im *Divide-and-Query*-Fall

**Eingabe:** eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ $y$ zurückgibt, wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
$\mathcal{I}' \subset \mathcal{I}$, wobei $\mathcal{I}'$ leer sein kann,
das Gewicht des zu $C$ gehörenden Ableitungsbaums *weight*,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei *modus* = *step* in jedem Schritt eine Frage stellt und
*modus* = *skip* nur bei Elementen der *debug_points* Liste eine Frage stellt,
die Liste von Prädikaten, die in der Slice sind *ATree*_slice
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
eine modifizierte Datenbank $\mathbf{D}$,
modifizierter *modus*
**procedure** *CPM_slice_divide_query*($< C, x, y >, \mathcal{I}'$, weight, $C_{buggy}$, $\mathbf{D}$, debug_points,
modus,*ATree*_slice)
**begin**
middle_weight = 0; $C_{middle}$:=$\varepsilon$
**while** weight $\neq 1$ **do**
*middle_point*($< C, x, y >$, $C_{middle}$, middle_weight,weight)
**if** $C \in ATree$_slice **then**
**if** $\neg\Sigma_C$ **then** answer := Query ($< C, x, y >$)
**else** answer := *Query_database*($< C, x, y >$, $\mathbf{D}$, debug_points, modus)
**endif**
**else** answer := true
**endif**
**if** answer = yes **then**
*CPM_slice_divide_query*($< C, x, y >, \mathcal{I}' \cup < C_{middle}, u, v >$, weight−middle_weight,
$C_{buggy}$, $\mathbf{D}$, debug_points, modus, *ATree*_slice)
**else**
**if** answer = no **then**
*CPM_slice_divide_query*($C_{middle}, \mathcal{I}'$, middle_weight,
$C_{buggy}$,$\mathbf{D}$, debug_points, modus, *ATree*_slice)
**else**
$C_{buggy}$:=$\varepsilon$; **return**($C_{buggy}$)
**endif**
**endif**
**endwhile**
$C_{buggy} := C_{middle} \leftarrow < C_1, x_1, y_1 >, \ldots, < C_n, x_n, y_n >$
**end**

Die Kombination der *CPM* mit der *Divide-and-Query*-Methode zeigt der Hauptalgorithmus 38. Der Algorithmus erledigt zwei wichtige Aufgaben:

- Er ruft die Prozedur *compute_weight* auf, die das Gewicht des ursprünglichen Ableitungsbaums *weight* liefert. Damit arbeitet dann das Verfahren *CPM_slice_divide_query*.
- Er behandelt den Fall, wenn als Antwort *cancel* eingegeben wurde. In diesem Fall wird der Debuggingprozeß beendet, die einzelnen Einträge, die während des Debuggings eingetragen wurden, werden wieder gelöscht.

**Algorithmus 38** Der Hauptalgorithmus des *Divide-and-Query*-Verfahrens, wenn die Testdatenbank und Slice verwendet werden

**Eingabe**: eine Klausel $C \in \mathbf{P}$,
eine Eingabe $x$, so daß $C$ die Antwort $y$ zurückgibt,
wobei $y \neq \perp$ und inkorrekt in $\mathcal{I}$,
$\mathcal{I}' \subset \mathcal{I}$,
eine Datenbank $\mathbf{D}$,
die Liste von Prädikatnamen, bei denen der Debugger anhalten muß *debug_points*,
die durchzuführende Aktion *modus*,
wobei $modus = step$ in jedem Schritt eine Frage stellt und
$modus = skip$ nur bei Elementen der *debug_points* Liste eine Frage stellt,
die Liste der Anfangspunkte des Slicing Verfahrens $\alpha$
in der Form $\alpha = (c, a, p)$, wobei $c$ für die Klauselnummer steht,
$a$ gibt die Position des Atom an
(das im Kopf stehende Atom kennzeichnen wir mit 0),
$p$ ist die Parameterposition in dem von $c$ und $a$ bestimmten Atom
**Ausgabe:** eine Instanz $C_{buggy}$ der fehlerhaften Klausel,
eine Datenbank $\mathbf{D}$

**procedure** $IDTS\_Debug\_divide\_and\_query\_CPM\_slice(< C, x, y >, C_{buggy}, \mathcal{I}',$
$\mathbf{D}, debug\_points, modus, \alpha)$
**begin**
  $compute\_weight(< C, x, y >, \mathcal{I}', weight)$;
  $ATree\_\text{slice} = improved\_slicing\_method(Dep_{red\mathbf{P}}, \alpha)$;
  $\mathbf{D} = T(F(\sum_\mathbf{P}))$;
  save$(T(F(\sum_\mathbf{P})))$;
  $CPM\_slice\_divide\_query(< C, x, y >, \mathcal{I}', weight, C_{buggy}, \mathbf{D},$ debug_points,
  modus,$ATree_$slice)
  **if** $C_{buggy} = \varepsilon$ **then**
    restore$(T(F(\sum_\mathbf{P})))$;
  **endif**
**end**

Das ist der letzte Fall, bei dem wir auf das Beispiel *compute* zurückgreifen. Wieder nehmen wir das Ziel:

```
program([1,2,3,4],[1,2,4],C,S).
```

Wie wir es schon bei den Fällen *CPM* und *Slice* getrennt festgestellt haben, erzielt die Kombination dieser Techniken mit der *Divide-and-Query*-Methode keine Verbesserung in der Anzahl der Fragen. Das gilt auch, wenn wir alle drei miteinander verbinden: Die Anzahl der Fragen bleibt immer fünf. Wenn man nur den Aspekt der Anzahl der Fragen betrachtet, dann kann man behaupten, daß diese Methode besser ist als alle anderen. Sogar die einfachste Version stellt weniger Fragen als bei allen anderen Fällen die Kombination. Aber man darf dabei nicht vergessen, daß der Baum immer neu berechnet wird, so daß sich die Laufzeit erhöht.

**Beispiel 64** Die bei der *Divide-and-Query*-Methode gestellten Fragen, wenn Slicing und *CPM* benutzt werden

```
Is the goal [complement( [1,2,3,4] , [1,2,4] , [4] )] correct? (y/n) n
Is the goal [complement( [3,4] , [4] , [4] )] correct? (y/n) n
Is the goal [member( 4 , [3,4] )] correct? (y/n) y
Is the goal [delete( 4 , [3,4] , [4] )] correct? (y/n) n
Is the goal [delete( 4 , [4] , [] )] correct? (y/n) y
```

Die Änderungen in der Testdatenbank im *Divide-and-Query*-Fall, wenn Slicing und *CPM* benutzt werden, sind die gleichen wie für den Fall *CPM* (siehe Beispiel 45).

Abbildung 6.3 zeigt, wie die einzelne Methoden die Anzahl der Fragen beeinflussen. Wie man leicht feststellen kann, ist mit den zwei Techniken *CPM* und Slice keinerlei Verbesserung zu erreichen. Die Abbildung ist identisch mit den Abbildungen 3.5.

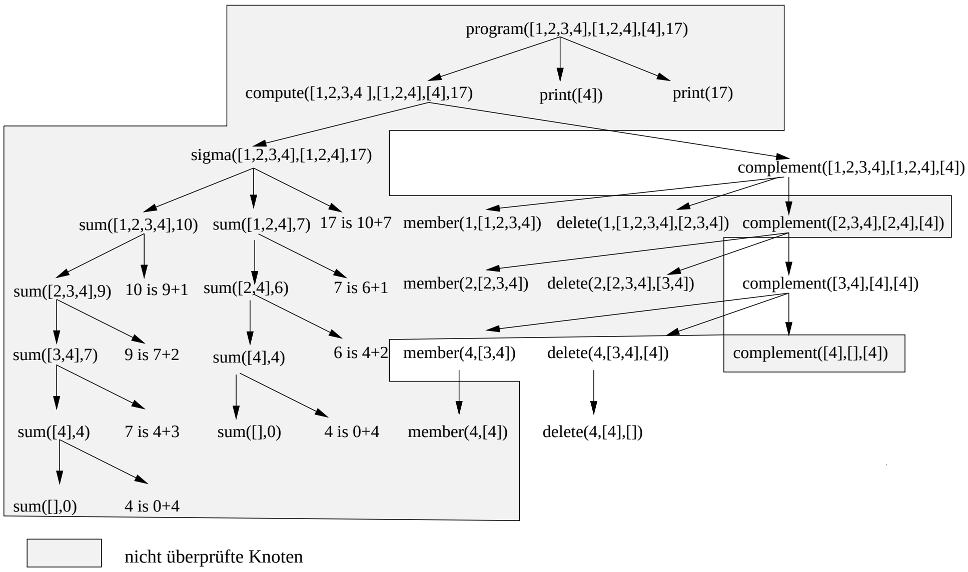



Abbildung 6.3: Die bei der Kombination aus *CPM*, Slice und *Divide-and-Query*-Methode gestellten Fragen für das *compute*-Beispiel

In diesem Kapitel haben wir durch unser Standardbeispiel *compute* erklärt, wie alle drei Methoden miteinander kombiniert werden können. Im nächsten Kapitel nehmen wir andere Beispiele, um die Vorteile der Kombination zu demonstrieren.

## 6.2 Überblick über das Gesamtsystem

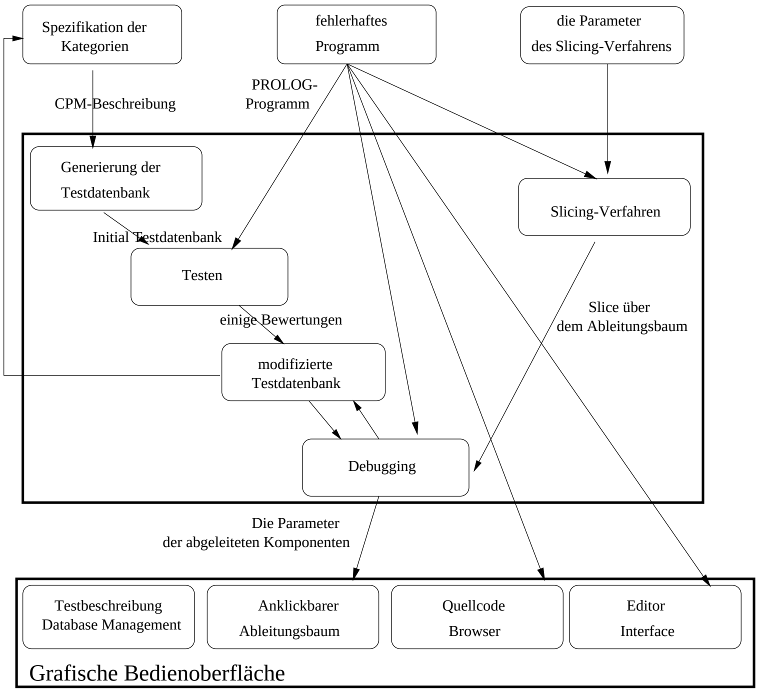



Abbildung 6.4: Die Struktur des *IDTS*-Systems

Die vier Hauptteile des Systems sind die folgenden:

- *Der Diagnosealgorithmus von Shapiro*: Die Methode von Shapiro liefert eine fehlerhafte Klausel, wenn das Programm und das Ziel gegeben sind. Die Algorithmen durchlaufen den Ableitungsbaum des Programms in unterschiedlicher Weise und stellen dem Benutzer Fragen. Diese Fragen beziehen sich auf das vermutete Verhalten der jeweiligen Klauseln.
- *Die Category Partition Methode*: Das Ziel besteht darin, eine Testdatenbank zu generieren und zu benutzen, um die Anzahl der dem Benutzer gestellten Fragen zu reduzieren.
- *Das Slicing-Verfahren*: Mit dem Slicing-Verfahren wird versucht, die Programmteile zu finden, die vermutlich den Fehler enthalten. Unser System verwendet *Backward-Slicing*.
- *Die graphische Bedienoberfläche*: Um das System einfach bedienen zu können, wurde eine graphische Bedienoberfläche entwickelt.

Vom Anwender erwartet das System die folgenden Eingaben:

- *Spezifikation der Kategorien*: Der Benutzer kann für einige Prädikate aus dem Programm die kritischen Eigenschaften (die *Kategorien*) und die *Wahlmöglichkeiten* (*choices*) eingeben. Außerdem müssen die *Suchfunktionen* definiert werden (siehe Kapitel 4).
- *Das fehlerhafte Programm*: Das Programm enthält einen oder mehrere Fehler, die nacheinander mit Hilfe des Debugging gefunden und ausgebessert werden können.

- *Die Parameter des Slicing-Verfahrens*: Der Anwender muß einen Punkt (eine Parameterposition in einem Atom in einer beliebigen Klausel aus dem Programm) angeben, wo er den Fehler in der Form einer falschen Ausgabe erkannt hat.

Durch Integration der drei Methoden (Original-Methode von Shapiro, *CPM* und Slicing) kann ein hoher Automatisierungsgrad erreicht werden, weil die von Shapiros Diagnosealgorithmen gestellten Fragen nur dann vom Anwender beantwortet werden müssen, wenn sie in der Slice sind (also bei der Berechnung des falschen Ergebnisses eine Rolle spielen) und keine Antwort in der Datenbank vorhanden ist.

## 6.3 Vergleich der Ergebnisse

In diesem Unterkapitel werden einige Ergebnisse der Debuggingmethoden, die in *IDTS* integriert sind, dargelegt. Drei kleine Beispiele (*convert*, *isort* und *compute*) werden ausgewählt, um die Anzahl der Fragen zu untersuchen, die die verschiedenen Methoden brauchen, um den Fehler zu finden.

Das erste Beispiel ist *convert*, das aus einer Liste $[x_1, \ldots, x_n]$ von ganzen Dezimalzahlen aus dem Intervall $\{-9, -8, \ldots, 0, 1, \ldots, 9\}$ eine gleichlange Liste von Binärzahlen generiert. Das Programm wandelt jede Dezimalzahl in eine acht-bit binäre Darstellung um, so daß alle konvertierten binären Zahlen genau vier *1*-Bits enthalten. Außerdem berechnet das Programm die Summe über alle Anzahlen von *1*-Bits. Die Transformation der Ziffern wurde in Form von einfachen PROLOG-Fakten $t_1(\ldots)$ und $t_2(\ldots)$ eingegeben. Bei Überprüfung des Programms können wir feststellen, daß der Fehler bei dem Fakt $t_1(0, [0, 0, 0, 1, 1, 1, 1, 1])$ liegt, weil es statt vier fünf Einsen enthält.

**Beispiel 65** Das Programm *convert*

```
q([],0).
q([N|T],S)          :- q2(N,NS),q(T,TS),S is NS+TS.
q2(N,NS)            :- name(N,CharList),convert(CharList,NBL),sum_list(NBL,NS),
                       open('bin',append,S),print(S,NBL),close(S),print(NS),nl.
convert([45,NC],NB) :- name(N,[NC]),t2(N,NB). % 45 is the '-'
convert([NC],NB)    :- name(N,[NC]),t1(N,NB).
sum_list([],0).
sum_list([H|T],N)   :- sum_list(T,N1), N is H+N1.
%coding positive digits     %coding negative digits
t1(0,[0,0,0,1,1,1,1,1]).
t1(1,[0,0,0,1,1,1,1,0]).    t2(1,[1,1,1,1,0,0,0,0]).
t1(2,[0,0,0,1,1,1,0,1]).    t2(2,[1,1,1,0,0,0,0,1]).
t1(3,[0,0,1,1,1,1,0,0]).    t2(3,[1,1,0,0,0,0,1,1]).
t1(4,[0,0,0,1,1,0,1,1]).    t2(4,[1,1,1,0,0,1,0,0]).
t1(5,[0,0,1,1,1,0,1,0]).    t2(5,[1,1,0,0,0,1,0,1]).
t1(6,[0,0,1,1,1,0,0,1]).    t2(6,[1,1,0,0,0,1,1,0]).
t1(7,[0,1,1,1,1,0,0,0]).    t2(7,[1,0,0,0,0,1,1,1]).
t1(8,[0,0,0,1,0,1,1,1]).    t2(8,[1,1,1,0,1,0,0,0]).
t1(9,[0,0,1,1,0,1,1,0]).    t2(9,[1,1,0,0,1,0,0,1]).
```

Das zweite Beispiel ist ein Programm aus der Arbeit von Shapiro, das *isort*-Beispiel. Wir haben zwei Fehler in das Programm eingebaut, so daß die Klauseln $C3$ und $C4$ verändert wurden. Die korrekten Klauseln sind:

```
(C3) insert(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
(C4) insert(X,[Y|Ys],[Y|Zs])   :- X > Y, insert(X,Ys,Zs).
```

Die veränderten falschen Klauseln sind:

```
(C3) insert(X,[Y|Ys],[Y,X|Ys]) :- X >= Y.
(C4) insert(X,Y,[X|Y]).
```

Dieses veränderte Programm liefert in vielen Fällen ein falsches Ergebnis, wie etwa, wenn als Eingabe die Liste `[-7, -5, 8, 9, 4, -3, 0, 7]` gegeben wurde.

**Beispiel 66** Das Programm *isort*

```
isort([X|Xs],Ys)          :- isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).
insert(X,[Y|Ys],[Y,X|Ys]) :- X >= Y.
insert(X,Y,[X|Y]).
insert(X,[],[X]).
```

Wir haben die Anzahl der Fragen für eine kleine Menge von Eingaben untersucht. Die Ergebnisse zeigen die Tabellen 1, 2 und 3. Die Spalten verweisen auf die gerade verwendeten Debuggingmethoden (Original-Shapiro, Shapiro & *CPM*, Shapiro & slice, Shapiro & *CPM* & slice), und die Zeilen zeigen die verschiedenen Durchführungsmechanismen (Bottom-Up, Top-Down, Divide-query).

**Tabelle 1** Die Anzahl der gestellten Fragen für das Ziel `program([1,2,3,4],[1,2,4],C,S)` im Programm *compute*. Der ganze Ableitungsbaum enthält 34 Knoten.

| compute | Shapiro | Shapiro + *CPM* | Shapiro + slice | Shapiro + *CPM*+slice |
|---|---|---|---|---|
| Bottom-Up | 16 | 14 | 8 | 6 |
| Top-Down | 12 | 9 | 11 | 8 |
| Divide-and-Query | 5 | 5 | 5 | 5 |

**Tabelle 2** Die Anzahl der gestellten Fragen für das Ziel `q([-7,-5,8,9,4,-3,0,7],X)` im Programm *convert*. Der ganze Ableitungsbaum enthält 195 Knoten.

| convert | Shapiro | Shapiro + *CPM* | Shapiro + slice | Shapiro + *CPM*+slice |
|---|---|---|---|---|
| Bottom-Up | 68 | 24 | 11 | 11 |
| Top-Down | 16 | 11 | 2 | 2 |
| Divide-and-Query | 6 | 6 | 3 | 3 |

**Tabelle 3** Die Anzahl der gestellten Fragen für das Ziel `isort([1,...,10,13,11,12,14,15,0],X)` im Programm *isort*. Der ganze Ableitungsbaum enthält 96 Knoten.

| isort | Shapiro | Shapiro + *CPM* | Shapiro + slice | Shapiro + *CPM*+slice |
|---|---|---|---|---|
| Bottom-Up | 10 | 7 | 10 | 7 |
| Top-Down | 13 | 3 | 10 | 3 |
| Divide-and-Query | 5 | 2 | 5 | 2 |

Tabelle 3 zeigt, daß die Anzahl der Fragen deutlich reduziert wurde kann, wenn *CPM*-Test und die Slicing-Technik angewandt wird. Die größte Verbesserungsrate kann man im Fall des *Bottom-Up*-Verfahrens erreichen, wo der Original-Algorithmus 68 Fragen braucht. Natürlich darf nicht vergessen werden, daß der Testanteil auch einige Antworten vom Anwender braucht.

## 6.4 Die graphische Bedienoberfläche des IDTS-Systems

Bislang hatte das in PROLOG implementierte *IDTS* eine textuelle Ein- und Ausgabeschnittstelle. Bei der Entwicklung ist jedoch klar geworden: Um das System benutzerfreundlich und leistungsfähiger zu machen, muß eine grafische Bedienoberfläche entwickelt werden, die alle von *IDTS* angebotenen Funktionen unterstützt und diese in interaktiver und übersichtlicher Form am Bildschirm darstellt. In der um eine solche grafische Komponente erweiterten Version *GIDTS* (*Graphical Interactive Debugging, Testing and Slicing System*) kann das System dem Anwender eine Umgebung anbieten, die den kompletten Ablauf des Debuggingverfahrens, wie beispielweise die Visualisierung des Ableitungsbaums, unterstützt und damit den Kontext der dem Benutzer gestellten Fragen erläutert.

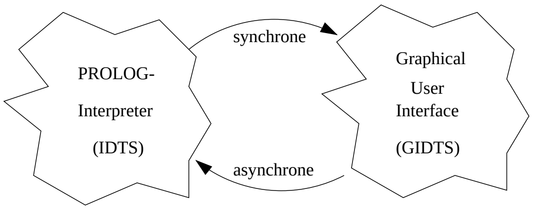



Abbildung 6.5: Die zwei in *GIDTS* zusammenarbeitenden Prozesse

Grundsätzlich läuft die *GUI* (*Graphical User Interface*) parallel zum *IDTS*-System, wobei der PROLOG-Interpreter von der grafischen Oberfläche aus gesteuert wird. Deswegen bestand die erste Aufgabe darin, eine Prozeßkommunikation zwischen diesen beiden Teilen zu entwickeln, so daß der langsamere PROLOG-Interpreter die Aktionen der GUI nicht blockiert. Die Lösung wurde darin gefunden, daß die Kommunikation von der GUI zum PROLOG-System synchron, in der umgekehrten Richtung jedoch asynchron erfolgt (siehe Abbildung 6.5).

Beim Design der Benutzeroberfläche haben wir als Ausgangspunkt das *GADT* (*Generalized Algorithmic Debugging Tool*, [WF93]), das für die Programmiersprache PASCAL entwickelt wurde, betrachtet. Das von uns entwickelte *GIDTS* erfüllt vollständig die Aufgaben von *IDTS*. Die Struktur von *GIDTS* zeigt Abbildung 6.6.

Im Rest dieses Kapitels werden wir die einzelnen Module von *GIDTS* näher beschreiben. Weil die Implementierung von *GIDTS* auf $\mathcal{N}$-GUIDE basiert, möchten wir zuerst kurz $\mathcal{N}$-GUIDE einführen.

### 6.4.1 Das New Graphical User Interface Developer's Environment - $\mathcal{N}$-GUIDE

$\mathcal{N}$-GUIDE ist eine Entwicklungsumgebung für grafische Benutzeroberflächen. Es unterscheidet sich von vergleichbaren Softwaresystemen durch seine hohe Effizienz, Portabilität und

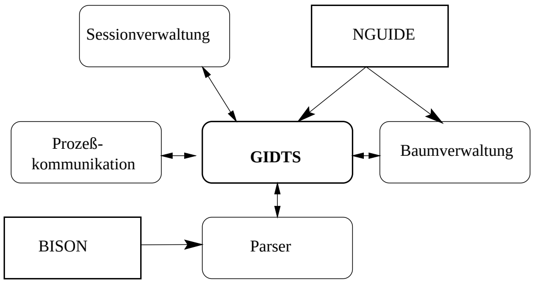



Abbildung 6.6: Die Struktur von *GIDTS*

Konfigurierbarkeit. $\mathcal{N}$-GUIDE wurde objektorientiert unter Verwendung der Programmiersprache C++ entwickelt. Das Kernsystem besteht aus einer Klassenbibliothek. Diese enthält bereits alle grundlegenden Funktionalitäten wie:

- Layoutberechnung: Die Layoutberechnung von $\mathcal{N}$-GUIDE basiert zwar auf einem ähnlichen Ansatz wie die von Motif, aber $\mathcal{N}$-GUIDE bietet erweiterte Möglichkeiten und eine effizientere Berechnung. Darüberhinaus bietet $\mathcal{N}$-GUIDE ein zweistufiges Layout, dessen abstraktere Stufe auch dem mit Layout unerfahrenen Programmierer schnell die Erzeugung übersichtlich gestalteter Dialoge ermöglicht.

- Ereignisbehandlung: Die Ereignisbehandlung ist an der Hierarchie der Dialogstruktur ausgerichtet und außerdem portabel ausgelegt. Selbst die Integration neuer Ereignistypen ist auf einfache Weise möglich.

- Dialogelemente: Dialogelemente sind die Grundlage jeder grafischen Benutzeroberfläche. Die wichtigsten Grundtypen befinden sich bereits in der Klassenbibliothek von $\mathcal{N}$-GUIDE (Texteingabefelder, Beschriftungen, Rollbalken, Menüs, Register, usw.).

- Möglichkeiten zur Verknüpfung von Objekten mit Eigenschaften: $\mathcal{N}$-GUIDE bietet sehr vielfältige Möglichkeiten, Objekte mit Eigenschaften von Dialogelementen zu verknüpfen. So können neben Prozeduren und Variablen auch andere Eigenschaften oder sogar ganze Ausdrücke von Funktionen, Operatoren, Variablen und Eigenschaften verknüpft werden. $\mathcal{N}$-GUIDE sorgt dabei stets für die Konsistenz aller beteiligten Objekte.

Die reichhaltige Funktionalität der Klassenbibliothek wird von zwei dazugehörigen Werkzeugen unterstützt: dem Konfigurationsmanager und dem Dialogeditor. Der Konfigurationsmanager vereinfacht die Veränderung der individuellen Konfiguration. Der Dialogeditor ist dagegen ein Werkzeug, das nur für den Programmierer gedacht ist. Er dient der Erstellung beliebiger statischer Dialoge. Statische Dialoge sind Dialoge, deren Struktur bereits zum Zeitpunkt der Übersetzung festliegt. Der Dialogeditor ist in hohem Maße erweiterbar angelegt, so daß er ohne Funktionalitätseinbuße auf einfache Weise um neue Dialogelementklassen erweitert werden kann.

Als eines der ersten Softwaresysteme seiner Art versucht $\mathcal{N}$-GUIDE, zweidimensionale Benutzeroberflächen mit dreidimensionalen zu verbinden. Dies gelingt einerseits durch Projektionen der zweidimensionalen Dialogelemente und andererseits durch neu implementierte dreidimensionale Dialogelemente.

*GIDTS* ist die erste Anwendung von realistischer Größe und Komplexität, die mit $\mathcal{N}$-GUIDE implementiert wurde. Der grafische Anteil von *GIDTS* konnte mit erwartungsgemäß geringem Aufwand programmiert werden.

### 6.4.2 Die Bestandteile der grafischen Benutzeroberfläche

In diesem Kapitel werden wir die Hauptmodule des Systems, die Abbildung 6.6 zeigt, näher erläutern:

- Prozeßkommunikation
- Parser für PROLOG-Ableitungsbäume
- Baumverwaltung
- Sessionverwaltung

Außerdem geben wir bei allen Modulen eine kurze Einführung in die Anwendung des Systems.

#### Prozeßkommunikation

Wie wir schon erwähnt haben, läuft der PROLOG-Interpreter parallel zur grafischen Oberfläche und wird von der GUI ferngesteuert. Dies wird dadurch erreicht, daß er sich genauso verhält, als wenn der Interpreter in einer Shell laufen würde. Daß der PROLOG-Interpreter im Hintergrund läuft, kann der Benutzer erkennen, wenn er im Logfenster folgen kann, wie das System arbeitet.

Die ursprüngliche Idee war, daß die GUI die vom Anwender erzeugten Eingaben simuliert. Beispielsweise könnte die Eingabe eine Ziffer sein, die einen Menüpunkt kennzeichnet, wie es bei der ursprünglichen textuellen Version des *IDTS*-Systems vorgesehen war. Diese Ziffer würde an *IDTS* weitergegeben. In der verbesserten Version werden die entsprechenden PROLOG-Prädikate im *IDTS*-Modul direkt aufgerufen, weil dies viel effektiver ist.

Die Prozeßkommunikation wird von einer $\mathcal{N}$-GUIDE Klasse *CGTerminal* gesteuert, wie Abbildung 6.7 zeigt:

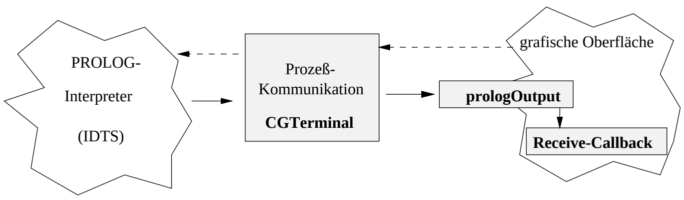



Abbildung 6.7: Die Prozeßkommunikation

Die Aufgaben dieser Klasse sind die folgenden:

- Ein Terminal, in dem eine Shell gestartet wird, wird geöffnet. Danach kann der PROLOG-Interpreteraufruf für *IDTS* gestartet werden.

- Die Übertragung der Ausgaben von *IDTS* zur grafischen Oberfläche findet asynchron statt, so daß *IDTS* nicht die GUI blockiert. Wie Abbildung 6.7 zeigt, wird die Stringvariable (`prologOutput`) einerseits mit dem Readoperator des *CGTerminal* und andererseits mit einer Prozedur (`ReceiveCallback`) verknüpft. Diese Prozedur wird aufgerufen, wenn `prologOutput` geändert wird. Der `ReceiveCallback` sorgt dafür, daß die grafische Oberfläche auf die *IDTS*-Ausgaben reagiert. Eine nähere Beschreibung der Prozeßkommunikation kann in [Nis98] nachgelesen werden.

- Die Kommunikation wird beendet.

Wenn der Anwender die Prozeßkommunikation selbst in die Hand nehmen möchte, kann er über ein *Logfenster* (siehe Abbildung 6.8) direkt Befehle an den Interpreter schicken. Dieses Fenster zeigt auch die PROLOG-Ausgaben.

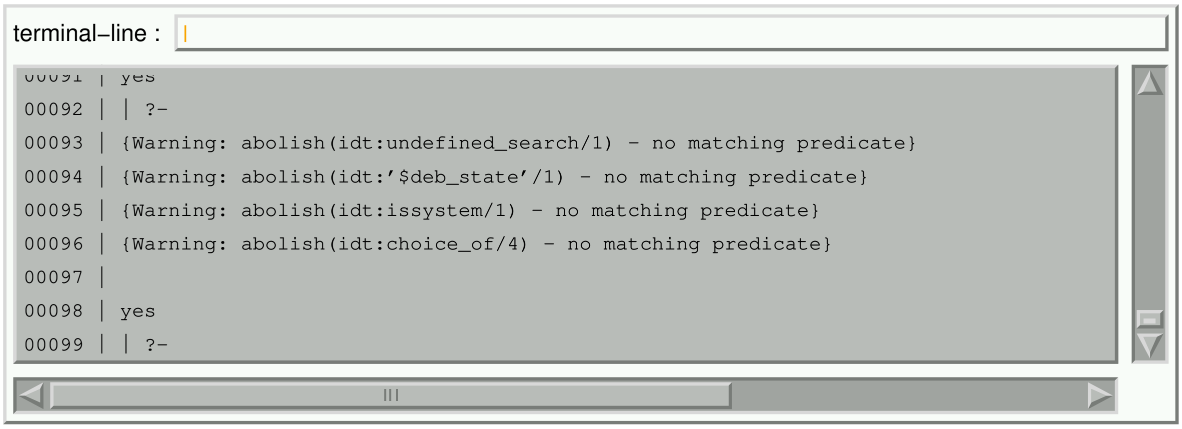



Abbildung 6.8: Das Logfenster

### Parser für PROLOG-Ableitungsbäume

Ein Parser muß verwendet werden, um aus der von *IDTS* kommenden Baumbeschreibung einen grafischen Ableitungsbaum generieren zu können. *IDTS* liefert eine geklammerte Baumstruktur, wie beispielweise die folgende für das von uns oft verwendete *compute*-Beispiel (siehe Beispiel 8):

```
IDT-Tree-Start
  1:program([1,2,3,4],[1,2,4],[4],17):-(2:compute([1,2,3,4],[1,2,4],[4],17)
  :-(10:sigma([1,2,3,4],[1,2,4],17):-(12:sum([1,2,3,4],10):-(12:sum([2,3,4],9)
  :-(12:sum([3,4],7):-(12:sum([4],4):-(11:sum([],0):-true),(0:(4 is 0+4)
  :-true)),(0:(7 is 4+3):-true)),(0:(9 is 7+2):-true)),(0:(10 is 9+1):-true)),
  (12:sum([1,2,4],7):-(12:sum([2,4],6):-(12:sum([4],4):-(11:sum([],0):-true),
  (0:(4 is 0+4):-true)),(0:(6 is 4+2):-true)),(0:(7 is 6+1):-true)),(0:(17 is 10+7)
  :-true)),(4:complement([1,2,3,4],[1,2,4],[4]):-(5:member(1,[1,2,3,4])
  :-true),(7:delete(1,[1,2,3,4],[2,3,4]):-true),(4:complement([2,3,4],[2,4],[4])
  :-(5:member(2,[2,3,4]):-true),(7:delete(2,[2,3,4],[3,4]):-true),
  (4:complement([3,4],[4],[4]):-(6:member(4,[3,4]):-(5:member(4,[4])
```

```
  :-true)),(9:delete(4,[3,4],[4]):-(7:delete(4,[4],[]):-true)),
  (3:complement([4],[],[4]):-true))))),(0:print([4]):-true),(0:print(17):-true)
IDT-Tree-End
```

Zum Parsen dieser Struktur wird die folgende Grammatik verwendet, die in BNF-Form gemäß dem Bison-Parser ([Han95]) angegeben ist:

```
Baum              ::= Baumliste | RPraedikat ':-' Baumliste.
Baum2             ::= '(' RPraedikat ':-' Baumliste ')' | 'true' | 'false' |
                      'uncalculated'.
Baumliste         ::= Baum2 | Baumliste ',' Baum2 | Baumliste ',' '(' Baumliste ')'
                      | '(' Baumliste ')'.
Struktur          ::= Atom '(' Termliste ')' | Zeichenkette '(' Termliste ')'.
Termliste         ::= Term | Termliste ',' Term.
Term              ::= Variable | Struktur | Liste | IntReal | Atom | Zeichenkette
                      | '(' Termliste ')'.
Liste             ::= '[' Termliste ']' | '[' Termliste '|' Term ']' | '[' ']'.
RPraedikat        ::= IntReal ':' RPraedikat1 | IntReal ':' '(' RPraedikat1 ')'
                      | RPraedikat1.
RPraedikat1       ::= Struktur | Atom | Vergleich | Wertzuweisung | Univ-Operation
                      | '!' | 'true' | 'false' | Zeichenkette.
Vergleich         ::= VarInt Vergleichsoperator Atom
                      | VarInt Vergleichsoperator Arithmet.Ausdruck.
Wertzuweisung     ::= VarInt 'is' Atom | VarInt 'is' Arithmet.Ausdruck.
Arithmet.Ausdruck ::= '+' Arithmet.Termliste | '-' Arithmet.Termliste
                      | Arithmet.Termliste.
Arithmet.Termliste ::= Arithmet.Term | Arithmet.Termliste '+' Arithmet.Term
                      | Arithmet.Termliste '-' Arithmet.Term.
Arithmet.Term     ::= Faktorliste | Liste.
Faktorliste       ::= Faktor | Faktor '*' Faktorliste | Faktor '/' Faktorliste.
Faktor            ::= Variable | IntReal | '(' Arithmet.Ausdruck ')' | Zeichenkette
                      | Struktur.
Univ-Operation    ::= Variable '=..' Variable | Variable '=..' Liste
                      | Struktur '=..' Variable | Struktur '=..' Liste
                      | Liste '=..' Variable | Liste '=..' Liste.
VarInt            ::= Variable | IntReal | Zeichenkette | Struktur | Liste | Atom.
Variable          ::= ( '_' | Großbuchstabe ) { Ziffer | Buchstabe | '_' }.
Atom              ::= ( Kleinbuchstabe { Ziffer | Buchstabe | '_' } ) | {Sonderzeichen}
                      | ( '(' Zeichenkette ')' ) | 'C' | '\+'.
Vergleichsoperator::= '/=' | '=' | '=<' | '<' | '>' | '>=' | '==' | '=|=' | '\=='.
IntReal           ::= ['-'] Ziffer { Ziffer } [ '.' { Ziffer } ].
Zeichenkette      ::= ''' { Zeichen } ''' | { Zeichen }.
Zeichen           ::= Sonderzeichen | Buchstabe | Ziffer.
Buchstabe         ::= Kleinbuchstabe | Großbuchstabe.
Großbuchstabe     ::= 'A'..'Z'.
Kleinbuchstabe    ::= 'a'..'z'.
Sonderzeichen     ::= '%' | '&' | '$' | '+' | '-' | '*' | '/' | '<' | usw..
Ziffer            ::= '0'..'9'.
```

Dieser Parser erzeugt Quellcode in der Programmiersprache C++, der eine Parseroberklasse und mehrere Unterklassen enthält. Die Vorgehensweise zeigt die Abbildung 6.9.

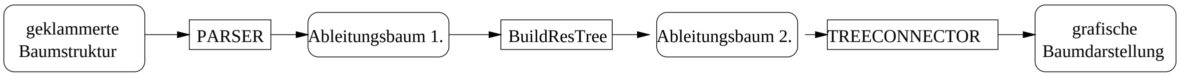



Abbildung 6.9: Die Erzeugung von grafischen Ableitungsbäumen

Die grafische Darstellung der erzeugten Baumstruktur wird von $\mathcal{N}$-GUIDE-Klassen ausgeführt.

**Baumverwaltung**

Bei der Entwicklung von *GIDTS* war eines der Hauptziele, die Visualisierung der Umgebung der gestellten Fragen zu erreichen. Wie wir wissen, repräsentiert jede Frage einen Knoten im Ableitungsbaum. Die Antwort des Anwenders könnte enorm erleichtert werden, wenn er das Voranschreiten oder den vorherigen Ablauf des Debuggings sehen könnte. Deswegen sollten die Ableitungsbäume grafisch dargestellt werden, und die Visualisierung komplexer Parameterwerte in den Knoten eines Ableitungsbaums sollte zufriedenstellend gelöst werden. Eine weitere erwünschte Eigenschaft war ein kontextsensitiver Wechsel vom Ableitungsbaum zum Quelltext. Klickt beispielsweise der Benutzer auf einen Knoten des Baumes, so wird die dazugehörige Stelle im Programmtext in einem Browserfenster angezeigt.

Wir wissen schon, daß *IDTS* während des Debuggingvorgangs den Anwender über die Korrektheit der berechneten Prädikate befragt. Das Problem der Identifizierung eines Knotens im Ableitungsbaum mußte dabei gelöst werden. Es kann nämlich vorkommen, daß das gleiche Prädikat mit der gleichen Parameterbelegung im Baum mehrmals vorkommt. *IDTS* wurde geändert und liefert außer der Frage auch einen Pfad zu dem jeweiligen Knoten:

```
[2,3,2,1]
Is the goal delete(2,[2,3,4],[3,4]) correct? (y/n)
```

Die leere Liste steht für die Wurzel und die hier im Beispiel vorkommende `[2,3,2,1]` Liste bedeutet: Wurzel, 1. Nachfolger, 2. Nachfolger, 3. Nachfolger und 2. Nachfolger.

In der bisherigen Version des Systems gibt es vier Möglichkeiten der Visualisierung der Parameter in den Knoten im Ableitungsbaum (siehe Abbildung 6.10).

Für den kontextsensitiven Wechsel vom Knoten zur jeweiligen Stelle im Programm könnten die folgenden Möglichkeiten in Frage kommen:

- eine textuelle Suche nach dem Prädikatnamen mit der jeweiligen Stelligkeit im Quellcode,
- die Suche in einem Syntaxbaum oder
- die Anzeige der entsprechenden Zeile im Quellcode zu jedem Prädikat im Ableitungsbaum (siehe Abbildung 6.11).

Die ersten zwei Methoden scheitern an den Eigenschaften von PROLOG, deswegen wurde die dritte Lösung gewählt und implementiert.

Die Verwaltung der Ableitungsbäume geschieht durch das *GIDTS*-Hauptfenster, das eine Menüleiste, einen scrollbaren Zeichenbereich, der die Ableitungsbäume zeigt, und eine Knopfleiste mit den beim Debuggingvorgang verwendbaren Knöpfen enthält (siehe Abbildung 6.12).

Die Menüleiste unterstützt die folgenden Funktionen:

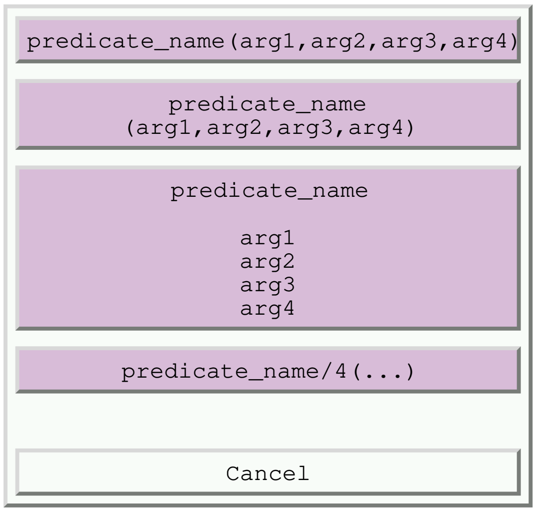



Abbildung 6.10: Die vier Möglichkeiten, um die Parameter im Ableitungsbaum zu zeigen

- Im *System*-Menü kann *IDTS* neu gestartet (*Reset System*) oder *GIDTS* beendet werden (*Quit*).
- Das *Session*-Menü wird in Kapitel 6.4.2 beschrieben.
- Das *Database*-Menü erlaubt die Generierung (*Generate*), das Entfernen (*Clear*), das Abspeichern (*Save*) und das Laden (*Load*) der Datenbank.
- Das *Debug&Test*-Menü wird in Kapitel 6.4.2 beschrieben.
- Das *Windows*-Menü stellt verschiedene wichtige Funktionen zur Verfügung:
  - *Show Source*: In eim neuen Fenster wird der Quellcode des fehlerhaften PROLOG-Programms angezeigt.
  - *Show Terminal log*: Das Logfenster (siehe Abbildung 6.8) wird geöffnet.
  - *Show CPM Description*: Für die *CPM*-Beschreibung wird ein Fenster geöffnet.
  - *Show Test Database*: Ein Fenster für die generierte Testdatenbank wird dargestellt.

Die Knoten im Baum, die keine Beschriftung *true*, *false* oder *uncalculated* haben, sind mit der Maus anwählbar. Klickt der Benutzer auf einen Knoten, so erscheint ein Popupmenü. In diesem Menü kann ausgewählt werden:

- *Zoom*: Der Knoten zeigt nur den Prädikatnamen und die Stelligkeit.
- *View source*: Es kann zum Quellcode gewechselt werden.
- *Copy*: Das Prädikat kann kopiert werden.

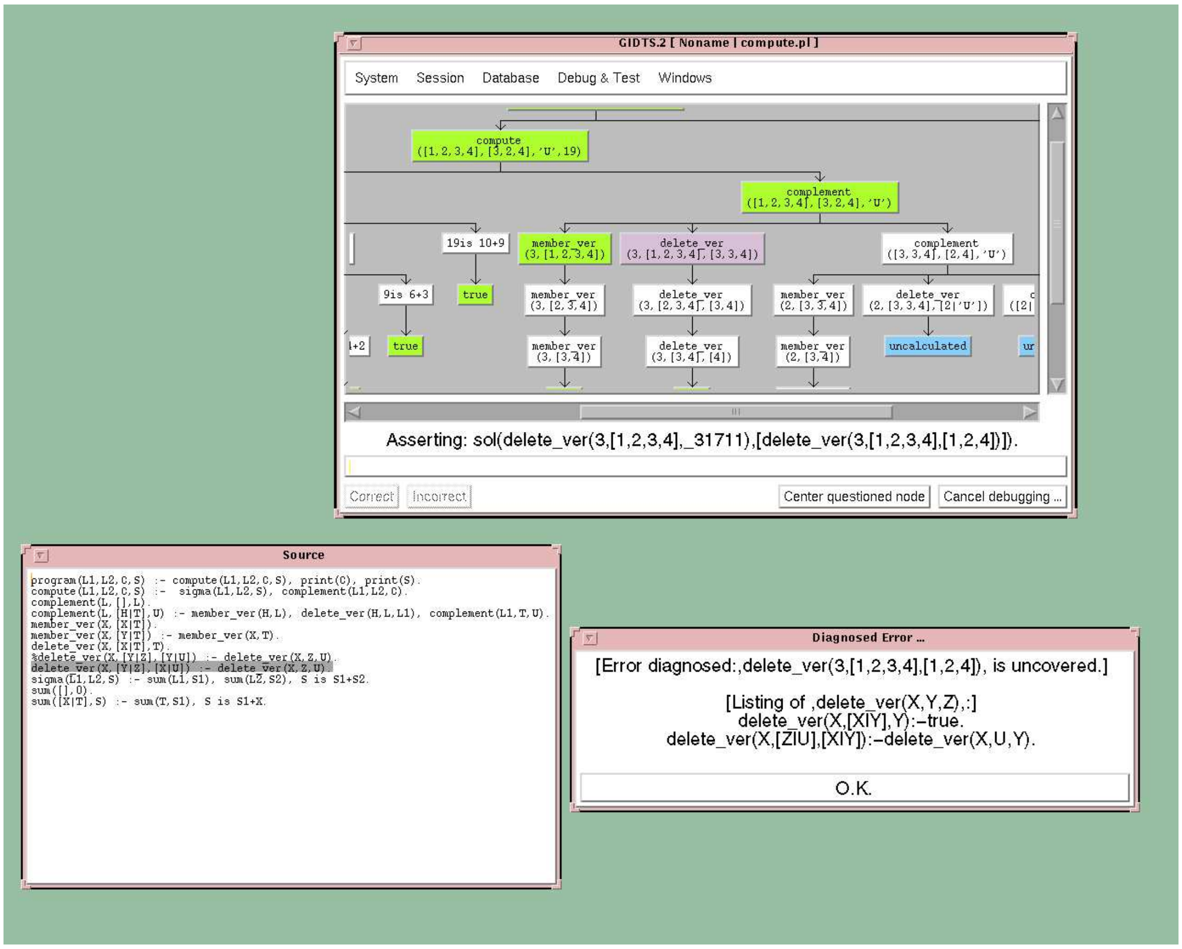



Abbildung 6.11: *IDTS* gibt zu jedem Prädikat im Ableitungsbaum die entsprechende Zeile im Quellcode aus

### Sessionverwaltung

Der Debugkontext (Programmname, Debuggingparameter usw.) muß geladen und gespeichert werden. Dadurch kann die Arbeit mit dem Debugger für einige Zeit unterbrochen werden, so daß bei einem späteren Neustart des Systems dort weitergearbeitet werden kann, wo man aufgehört hat. Die Informationen, die abgespeichert werden müssen, sind die folgenden:

- Dateiname der PROLOG- und der *CPM*-Datei
- Dateiname der Testdatenbank bzw. *none* oder *generated*
- das getestete Prädikat mit seinen Debuggingparametern
- die Debuggingmethode (Shapiro, *CPM*, Slicing oder die Kombination aller drei Verfahren)
- die Traversierungsart (Bottom-Up, Top-Down, Divide-and-Query)

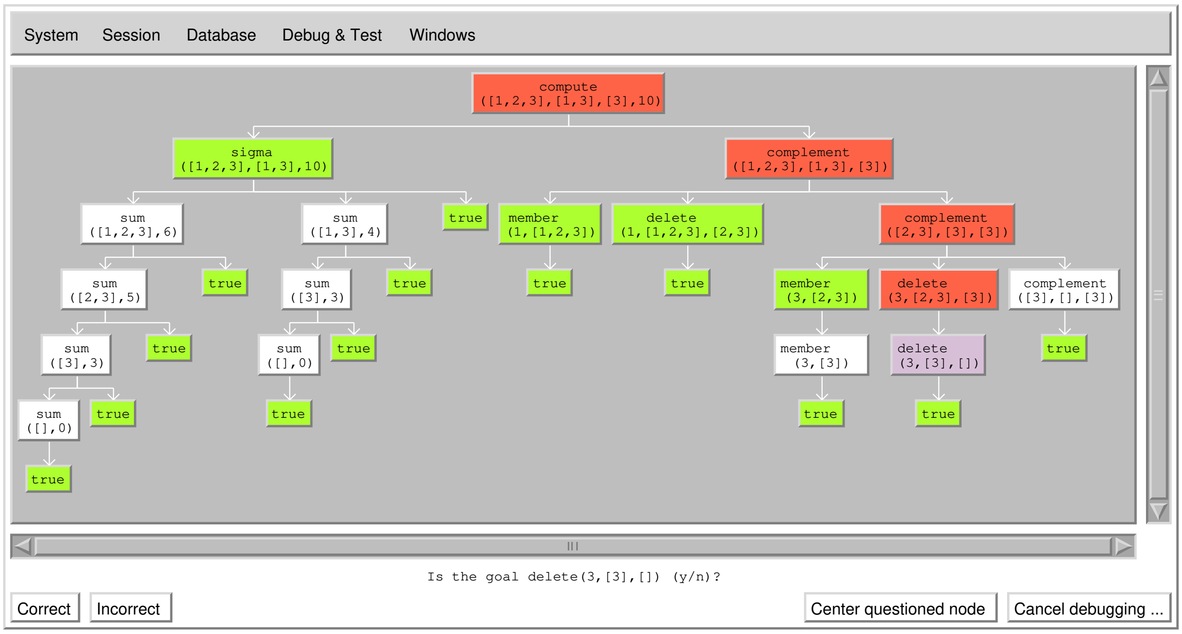



Abbildung 6.12: Das Hauptfenster

- die Benutzerinteraktionen

Wenn eine Sitzung geladen wird, kann der Anwender dort fortfahren, wo die Protokollierung aufgehört hat.

Beim Beginn der Arbeit mit *GIDTS* muß eine Sitzung (Session) gestartet werden, was über das *Session*-Menü möglich ist. Es gibt eine Möglichkeit, mit *New Session* eine neue Sitzung zu erzeugen, außerdem kann eine schon existierende Session geladen (*Load Session*) oder gespeichert (*Save Session*) werden. Das erscheinende Fenster zeigt Abbildung 6.13. Der Anwender muß den Namen des Programms (*Buggy PROLOG file*) und der Testbeschreibung (*CPM description*) eingeben. Wenn als Testbeschreibung das Zeichen "–" eingeben wird, dann bedeutet es, daß keine Testbeschreibung vorhanden ist. Der Name der Sitzung (*Session Name*) kann auch vom Benutzer eingegeben werden. Dieser dient später für die eindeutige Identifizierung der bisher gespeicherten Ergebnisse der Debuggingsitzung.

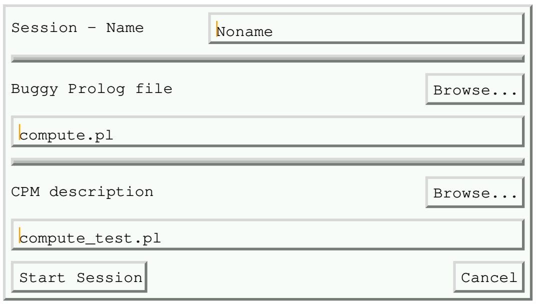



Abbildung 6.13: Sessionverwaltungsfenster

### Ablaufen eines Debuggingvorgangs

Ein Debuggingvorgang beginnt, indem eine Sitzung neu gestartet oder eingeladen wird (siehe Kapitel 6.4.2). Wenn eine neue Sitzung angefangen wird, kann im *Database*-Menü eine Testdatenbank eingeladen oder aus einer *CPM*-Beschreibung eine Anfangsdatenbank generiert werden, wie wir dies im Kapitel 4.1 beschrieben haben. Der nächste Schritt ist das Testen, das durch das Menü *Debug&Test* aufrufbar ist, wo man den Menüpunkt *Test* auswählt. Dann wird ein Fenster geöffnet, in dessen oberer Hälfte diejenigen Prädikate stehen, die eine *CPM*-Beschreibung besitzen, und darunter erscheinen nach der Auswahl eines Prädikats die zu diesem Prädikat gehörenden, aber noch nicht getesteten Testrahmen (vergleiche mit Kapitel 4.2). Das Fenster zeigt die Abbildung 6.14.

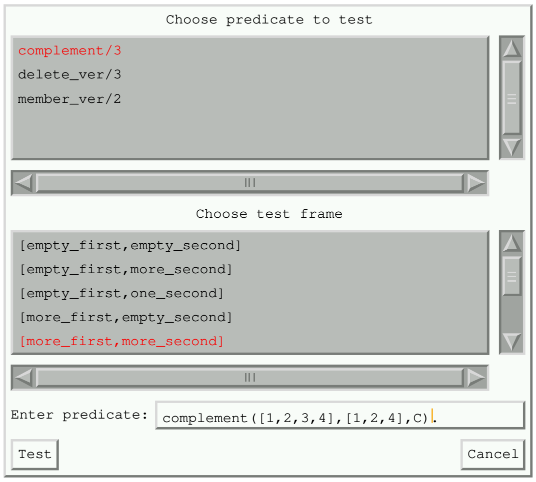



Abbildung 6.14: Testfenster

Die Testergebnisse werden in der Datenbank eingetragen. Nach dem Test kann der Debuggingprozeß gestartet werden, wobei man zuerst eine Debuggingmethode (Shapiro, *CPM*, Slicing oder die Kombination der drei Verfahren) auswählen muß und dann in einem Fenster (Abbildung 6.15) die notwendigen Debuggingparameter einträgt:

- *Debug Clause*: das Zielprädikat
- *Traversal Method*: die Traversierungsart (Bottom-Up, Top-Down, Divide-and-Query)
- *Debug Points*: die Liste von Prädikatnamen, bei denen der Debugger anhalten muß
- *Debugging Point Action*: Entweder hält der Debugger nur bei den Elementen der Liste *Debug Points* (*skip*), oder er schaltet nach dem ersten gefundenen Element aus dieser Liste auf den Einzelschritt-Modus ( *step*)

In dem scrollbaren Zeichenbereich erscheint der Ableitungsbaum, und der Benutzer kann mit Hilfe der Knöpfe *correct*, *incorrect* und *cancel debugging* interagieren. Die Knoten, bei denen der Anwender die Frage mit *incorrect* beantwortet hat (und natürlich alle, die zu der gleichen Klasse gehören) sind rot gefärbt, die korrekten Knoten sind grün gekennzeichnet.

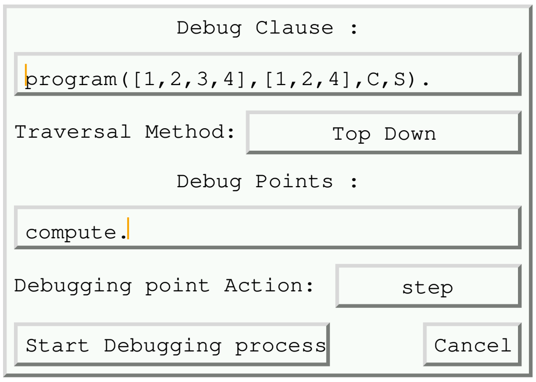



Abbildung 6.15: Das Fenster für die Einstellung der Debuggingparameter

Mit einer dritten Farbe wird der aktuelle Knoten sichtbar gemacht. Wenn durch den Debuggingprozeß die falsche Klausel gefunden wurde, dann kann das Browserfenster gestartet werden. Dieses enthält den PROLOG-Quellcode. Nach der Änderung des Programms kann der nächste Fehler gesucht werden.

### Gesamtsystem

In diesem Kapitel haben wir *GIDTS* vorgestellt.

Das ursprüngliche *IDTS*-System bietet nur eine textuelle Benutzeroberfläche. Die grafische Oberfläche *GIDTS* macht *IDTS* benutzerfreundlicher und leistungsfähiger. Dieses System liefert zusätzliche Informationen, die es für den Anwender einfacher machen, gestellte Fragen zu beantworten und den Debuggingprozeß zu kontrollieren. Abbildung 6.16 zeigt einige der *GIDTS*-Funktionen:

- Einstellung der Debuggingparameter,
- Anzeige des Quellcodes der Testbeschreibung,
- Anzeige des Quellcodes des fehlerhaften PROLOG-Programms,
- grafische Darstellung des Ableitungsbaums,
- das Logfenster,
- Anzeige des Fehlers.

### Erweiterungsmöglichkeiten

*GIDTS* kann noch weiterentwickelt werden, um das System noch effizienter zu machen:

- Ein integrierter Editor für die Darstellung und Veränderung des Quelltextes sollte vorhanden sein.
- Eine flexible Modulschnittstelle kann genutzt werden, um die Ausgaben des getesteten Programms zu visualisieren.

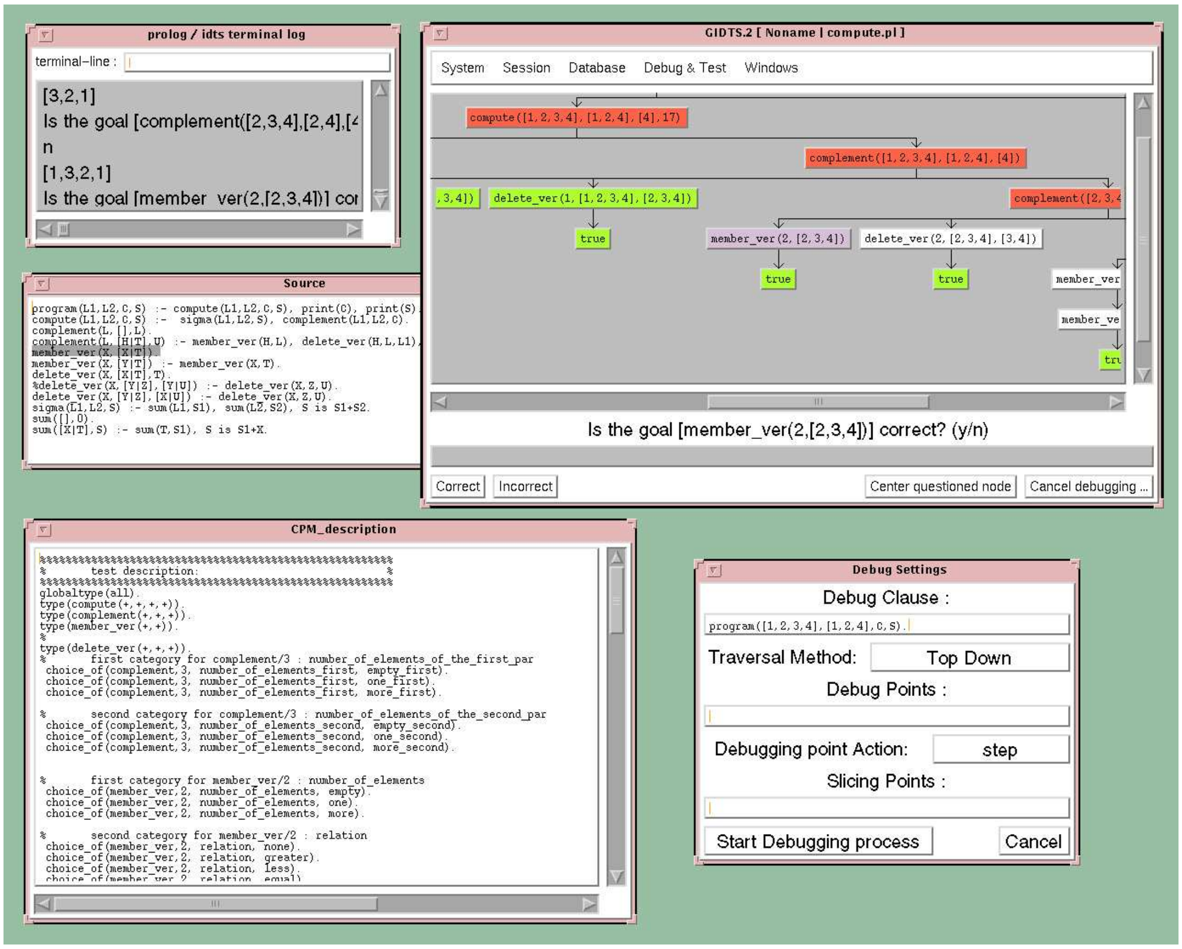


Abbildung 6.16: Das vollständige *GIDTS* mit Baum und Editoren

- Ableitungsbäume sollten durch Übersichtsgrafiken und grafische Folding-Techniken übersichtlicher gemacht werden können.
- Teile des PROLOG-Programms können in Form eines Cross-Referencing-Diagramms dargestellt werden.
- Mit Hilfe der Sessionverwaltung könnte man ein sogenanntes *Undo* anbieten: Einige Schritte können zurückgenommen werden.

## 6.5 Zusammenfassung

In diesem Kapitel haben wir zuerst das Gesamtsystem *IDTS* beschrieben (frühere Version des System sehe [HGAK93]). *IDTS* kombiniert die Original-Methoden von Shapiro mit der *CPM*-Methode und der Slicing-Technik. Deswegen haben wir im zweiten Teil des Kapitels die bei der Kombination verwendeten Algorithmen genau beschrieben und die Vorgehensweise durch das *compute*-Beispiel illustriert. Das Kapitel enthält einen Vergleich der Ergebnisse, wo wir bei drei verschiedenen Programmen die Anzahl der gestellten Fragen untersucht haben, wenn die verschiedenen Techniken verwendet werden. Zum System wurde eine grafische Oberfläche entwickelt, dessen kurze Beschreibung im letzten Teil des Kapitels enthalten ist.

Im nächsten Kapitel werden wir andere zum Testen und Debugging entwickelte Systeme vorführen, nach verschiedenen Kriterien beurteilen und einen Vergleich mit *IDTS* ziehen.

# Teil II

# IDTS im Vergleich

# Kapitel 7

# Debuggingsysteme

In allen Programmierumgebungen bildet die Test- und Debuggingmöglichkeit einen wichtigen Teil. Manche Programmierumgebungen bieten eine Testunterstützung, um feststellen zu können, ob ein gegebenes Programm Fehler enthält oder nicht. Debugging ist ein Verfahren, bei dem mit Hilfe einer Analyse des Programms die Fehler lokalisiert und entdeckt werden sollen. Hierbei ist die Idealvorstellung die folgende: Es gibt Fehler im Programm, von denen der Programmierer selbst nicht weiß, wo sie liegen könnten. Am Ende des Debuggingvorgangs wurden alle Fehler gefunden. In Wirklichkeit sieht die Situation anders aus: Es gibt keine Garantie, daß alle Fehler gefunden werden. Wie Ducasse ([DN94]) schreibt, sieht die Situation in der Mitte eines Debuggingprozesses folgendermaßen aus:

- Einige Fehler sind gefunden und behoben.
- Von einem Teil des Programms kann behauptet werden, daß er korrekt ist.
- Einige Teile sind noch nicht überprüft.
- Es gibt noch Fehler im Programm.

Es gibt noch andere Probleme, die man betrachten muß:

- Die Bedeutung des Begriffes Fehler ist nicht selbstverständlich. Ein und dasselbe Verhalten eines Programms kann manchmal als Fehler und manchmal als Eigenschaft betrachtet werden.
- Alle Programmierer wissen, daß man von einigen Teilen des erstellten Programms vermutet, daß sie sicher korrekt sind (zum Beispiel Standardbibliotheken).

Die Grenze zwischen den zwei Begriffen *Testen* (siehe Kapitel 4.2) und *Debuggen* ist nicht eindeutig. Sogar einige der hier vorgestellten Systeme vermischen beide Aufgaben. Deswegen werden wir im folgenden die zwei Techniken nicht streng voneinander trennen.

In diesem Kapitel versuchen wir, einen Überblick über die Systeme zu geben, die mit der Zielsetzung Testen und Debuggen entwickelt worden sind. Hierbei konzentrieren wir uns auf Systeme für logische Programme, wobei eindeutig zwischen *Lehrsystemen* und eigentlichen *Diagnosesystemen* getrennt werden kann.

Lehrsysteme sind für unerfahrene Programmierer geeignet. Das Programm, das dem Debugging unterzogen wird, ist klein. Die auftretenden Probleme sind stereotyp. Eine kurze

Spezifikation wird auch oft vom Anwender angegeben. Ein solches System ist zum Beispiel *Apropos* (siehe 7.5.1).

Diagnosesysteme sind für die Fehlersuche in von erfahrenen Programmierern geschriebenen Systemen gedacht. Das Programm ist meistens größer und deswegen nicht so einfach zu überblicken. Spezifikationen sind nur sehr selten angegeben. Als Folge davon haben Lehrsysteme und Diagnosesysteme verschiedene Zielsetzungen und Anforderungen.

Es ist sehr schwer, die schon existierenden Test- und Debuggingsysteme zu beschreiben. Die erste Schwierigkeit ergibt sich von selbst: Wie können die verschiedenen Systeme überhaupt verglichen werden? Sie sind für verschiedene Ziele entwickelt worden, deswegen brauchen sie verschiedene Eingaben und liefern verschiedene Ausgaben. Auch die verwendeten Techniken sind unterschiedlich. Wir haben versucht, Kriterien zu finden, durch deren Verwendung die Systeme verglichen werden können. Wir haben vier Kriterien gefunden, nach denen sich die Systeme beurteilen und klassifizieren lassen:

- aus Anwendersicht wichtige Kriterien
- die benötigten Kenntnisse
- die Aufgabe des Debuggingsystems
- die verwendeten Techniken

Weiterhin können die Systeme interaktiv oder statisch sein. Einige Systeme, wie zum Beispiel unser *IDTS*, haben eine „Orakeldatenbank", in der Informationen über das erwartete Verhalten eingegeben werden können. Diese Informationen können zum Beispiel in Form einer formalen Spezifikation vorliegen. Aber die meisten Debuggingprozesse sind um so komplizierter, je mehr Fälle in der Spezifikation vorgesehen und definiert werden. Eine mögliche Lösung dieses Problems ist, daß der Anwender selbst auch in den Debuggingprozeß einbezogen wird. Er kann das fehlende Wissen über Programmteile liefern. In diesem Sinn springt der Benutzer als Ersatz für die fehlende Spezifikation ein. Diese Systeme werden interaktiv genannt, sollten aber besser als halbautomatisch ([Sev87]) bezeichnet werden. Das Gegenteil bilden die statischen Systeme, die während des Debuggings keinerlei Informationen vom Anwender erfragen. Zu dieser Gruppe gehören Systeme, die Programme verifizieren, und die Lehrsysteme. Erfahrungsgemäß sind die Systeme, die die interaktive Strategie benutzen, für das Debugging besser geeignet und haben realistischere Anwendungen.

In diesem Kapitel werden wir zuerst die oben aufgeführten Kriterien näher beschreiben. Anschließend versuchen wir, einige Systeme nach diesen Kriterien zu beurteilen. Bei der Ausarbeitung der Kriterien haben wir uns auf die Arbeiten von Ducasse ([Duc93], [DE88],[DN94]) und Shahmehri ([Sha91],[SKF93]) gestützt. Andere Kriterien wurden zum Beispiel von Seviora ([Sev87]) ausgearbeitet.

## 7.1 Aus Anwendersicht wichtige Kriterien

Wie können die verschiedenen Systeme aus der Sicht des Anwenders beurteilt werden? Allgemein kann ein Debuggingsystem wie in Abbildung 7.1 beschrieben werden. Eine Gruppierung nach vier Kriterien schlägt Shahmehri vor.

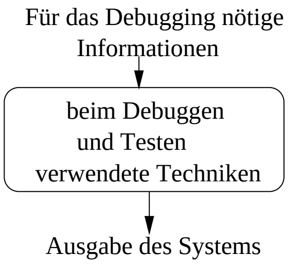



Abbildung 7.1: Eine allgemeine Beschreibung eines Debuggingsystems

- *Generalität*: Ein System wird als generell bezeichnet, wenn es das Programm ohne vorherige Kenntnisse über den verwendeten Algorithmus oder die Implementation verstehen kann. Zusätzlich bedeutet das, daß das System flexibel genug sind, mehrere Fehlerarten (inkorrekte Ausgabe oder Terminierung ohne Ausgabe, usw.) zu erkennen.

- *Erkennende Plausibilität* (*cognitive plausibility*): Was versteht man unter erkennender Plausibilität? Ein System ist gut, wenn es der Denkweise des Anwenders folgen oder sie simulieren kann. Genau das drückt dieser Begriff aus. Um dieses Ziel zu erreichen, gibt es verschiedene Ansätze:

  - Verschiedene Debuggingstufen: abstrakt oder ausführlich. Das Ziel beim abstrakten Debugging ist die Lokalisierung des Fehlers in einem kleinen Teil des Quellcodes (*Debuggingeinheit*, zum Beispiel als *loop*-Prozedur etc.), während auf Implementationsdetails verzichtet wird. Der Benutzer interessiert sich nur für das Gesamtverhalten der Debuggingeinheit. In der ausführlichen Stufe werden die Einzelheiten in der Debuggingeinheit mit Hilfe der Kontroll- und Datenabhängigkeiten ermittelt.
  - Die Richtung des Debuggings: Top-Down, Bottom-Up, vorwärts oder rückwärts. Der Durchführungsreihenfolge eines Programms kann auf abstrakter Ebene vom Debuggingprozeß gefolgt werden, zum Beispiel Top-Down, depth-first. Auf der ausführlichen Ebene des Debuggings gibt es zwei Möglichkeiten: Durch die Vorwärts-Strategie kann die Durchführung des Codes geprüft werden. Bei der Rückwärts-Methode wird versucht, die Ursache des Fehlers in den vorher durchgeführten Schritten zu finden. Der Rückwärts-Strategie folgt unser *IDTS*-System, wenn wir den Slicing-Teil aufrufen. Näheres siehe Kapitel 5.

- *Automatisierungsgrad*: Das Ziel besteht darin, dem Anwender so viele Aufgaben wie möglich abzunehmen.

- *Beurteilung der Benutzererfahrung*: Shahmehri et al. weisen darauf hin, daß ein System für verschiedene Gruppen von Anwendern geeignet sein soll. Es muß sowohl für den Programmierer, der mit der Sprache oder dem Programm noch nicht vertraut ist, als auch für den, der mehr Erfahrung hat, ein nützliches Werkzeug sein.

## 7.2 Die benötigten Kenntnisse

Wenn wir in einem Programm Fehler finden wollen, brauchen wir bestimmte Informationen über das Programm und verschiedene zusätzliche Kenntnisse. Beide können aus der Sicht des

Menschen und des Systems betrachtet werden. Die benötigten Informationen aus der Sicht des Systems werden wir bei der Beschreibung der einzelnen Systeme vorführen.

Die benötigten Kenntnissen, wie sie Abbildung 7.2 zeigt, können folgendermaßen gruppiert werden:

1. Kenntnisse über das *beabsichtigte* Programm:
   - *Ein- und Ausgabe*: Wenn keine Übereinstimmung zwischen der Ausgabe des *beabsichtigten* Programms und der Ausgabe des *tatsächlichen* Programms besteht, dann ist Debugging notwendig. Wenn das Programm von einem anderen Programmierer geschrieben wurde, kann der Anwender auf den ersten Blick nur dies erkennen.
   - *Implementation*: Die Implementation ist die erwartete, wenn sie das beabsichtigte Ergebnis liefert und die Spezifikation erfüllt. Auch hier kann ein Vergleich zwischen der Implementation des beabsichtigten Programms und der Implementation des tatsächlichen Programms angestellt werden. Es gibt einen starken Zusammenhang mit der erwarteten Ausgabe.
   - *Verhalten*: Der Anwender muß eine Vorstellung über das erwartete Verhalten haben, damit er dem Debuggingprozeß folgen kann.
2. Kenntnisse über das *tatsächliche* Programm:
   - *Ein- und Ausgabe*: Was das Programm bei der Durchführung liefert.
   - *Implementation*: Um die Verständlichkeit zu erhöhen, werden einige Programmierkonventionen, die die Implementation betreffen, festgelegt.
   - *Verhalten*: Die Erscheinungsform dieser Information ist meistens ein Kontroll- oder Datenfluß.
3. *Verstehen der Programmiersprache*: Dieser Punkt ist stark mit der tatsächlichen Implementation verbunden, weil man diese Kenntnisse braucht, um im Programm verwendete Daten- und Kontrollstrukturen zu verstehen.
4. *Allgemeine Kenntnisse über die Programmierung*: Es ist nicht einfach, Kenntnisse über die Programmiersprache von Kenntnissen über die Programmierung zu trennen. Es ist notwendig, weil einige Systeme nicht nur für eine einzige Programmiersprache geeignet sind. Viele haben untersucht (siehe [Wer87] und [SSP85]), welches Wissen die Programmierer verwenden:
   - *Programmschema*: Abstrakte Programmvorstellung.
   - *Ziel der Programmierung*: Mit welchem Ziel wird das Programm geschrieben?
   - *Wechselwirkung zwischen Ziel und Schema*: Ob das Schema genau das ausführt, was das Ziel war.
5. *Kenntnisse über das Anwendungsgebiet*: Diese braucht man, um überprüfen zu können, ob die oben angegebenen Daten (Ergebnis, Verhalten, Implementation) mit dem Anwendungsgebiet konsistent sind.

6. *Kenntnisse über die Fehler*: Viele Arbeiten befassen sich mit den verschiedenen vorkommenden Fehlertypen. Sie sind sich darin einig, daß es Unterschiede zwischen Fehlern gibt, die von unerfahrenen und von geübten Programmierern begangen werden. Die Aufgabe der Lehrsysteme besteht darin, möglichst viele Fälle von Fehlern zu erkennen und zu wissen, wie und warum Anfänger Fehler machen. Die von Anfänger gemachten Fehler heißen *stereotype Fehler*.

7. *Kenntnisse über Debuggingmethoden*: Die benutzten Methoden werden vom zur Verfügung stehenden Werkzeug bestimmt. Der Anwender muß wissen, wie er diese benutzen, und was er damit erreichen kann.

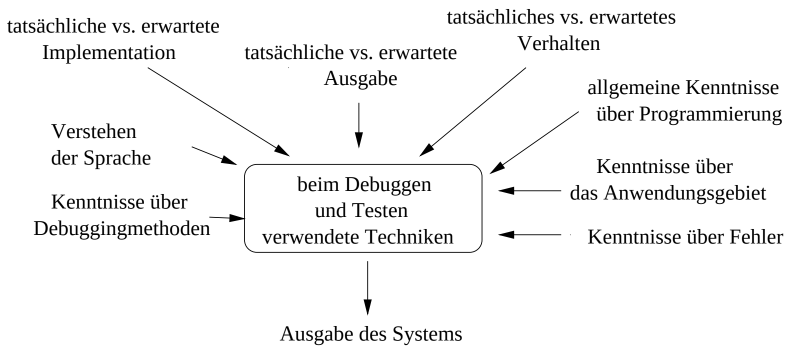



Abbildung 7.2: Die verwendeten Informationen

Die Kombinationen erwartete/tatsächliche Ein- und Ausgabe und erwartete/tatsächliche Implementation sind für die Verifikation geeignet. Die Systeme benutzen die Spezifikation des beabsichtigten Programms, um die Korrektheit des tatsächlichen Programms zu überprüfen. Bislang ist es nur gelungen, sehr kleine Programme mit Hilfe dieser Vorgehensweise zu überprüfen. Daher wird diese Technik vorwiegend in Lehrsystemen benutzt. Diese Systeme dienen in erster Linie dem Erlernen der Sprache, und wenn in einem von einem Studenten geschriebenen Programm ein Fehler vorhanden ist, dann soll das Programm den Fehler erklären.

Die Möglichkeit des Vergleichs des erwarteten/tatsächlichen Programmverhaltens verwenden die Systeme, die für das Debugging entwickelt worden sind. Beim Debugging wird zuerst der Fehler lokalisiert. Einige Systeme versuchen dann, den Fehler zu erklären und dem Anwender Verbesserungsvorschläge zu geben ([TS85], [Eis85a]).

Die Segmentierung des verwendeten Wissens im Debuggingprozeß kann auch aus der Sicht des Programmierers als Anwender betrachtet werden. Vessey ([Ves85]) hat die Hierarchie der für den Programmierer wichtigen Kenntnisse folgendermaßen definiert:

- *Bestimmen des Problems, Vergleich der korrekten und inkorrekten Ausgabe*: Man braucht Kenntnisse über die Eingabe und die tatsächliche und erwartete Ausgabe.

- *Verstehen der Struktur des Programms und der Einzelteile*: Hier werden Kenntnisse über das Programm und über die Programmiersprache, außerdem allgemeine Kenntnisse über die Programmierung verwendet.

- *Untersuchen der Programm-Durchführung und der AND/OR-Kontrolle*: Dies bedeutet die Analyse des Verhaltens des tatsächlichen Programms.

- *Korrigieren des Fehlers.*

## 7.3 Die Aufgabe des Debuggingsystems

Was ist eigentlich die Aufgabe eines Test- und Debuggingsystems? Shahmehri, Kamker und Fritzson haben fünf verschiedene Aufgaben als die wichtigsten bezeichnet (siehe Abbildung 7.3):

- Testfallgenerierung
- Erkennung des Fehlers
- Lokalisieren des Fehlers
- Fehlererklärung
- Fehlerkorrektur

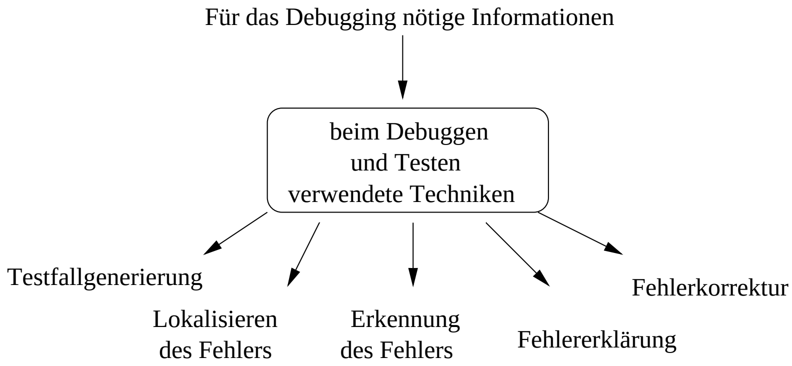



Abbildung 7.3: Die verschiedenen von Debuggingsystemen erfüllten Aufgaben

Die Systeme, die Testfälle generieren, sind für das Testen entwickelt worden. Die heutzutage existierenden Systeme bieten meistens die Möglichkeit der Fehlerlokalisierung und Fehlererkennung. Einige Systeme versuchen, auch die Aufgabe der Erklärung (warum der Fehler aufgetreten ist) und der Korrektur des Fehlers dem Anwender abzunehmen, aber dieses Gebiet scheint sehr komplex und mit viel Aufwand verbunden zu sein. Die Systeme, die versuchen, diese Aufgabe zu erfüllen, verwenden Ideen aus der Programmverifikation (es wird versucht, die totale Korrektheit des Programms unabhängig von Testfällen zu beweisen).

## 7.4 Die verwendeten Techniken

In diesem Kapitel versuchen wir zuerst, die drei beim Debugging verwendeten Hauptstrategien zu beschreiben. Diese Strategien werden ausschließlich oder miteinander kombiniert in den Debuggingsystemen verwendet. Es folgt die Auflistung der einzelnen Techniken, die bei den Strategien verwendet werden können.
Die drei Hauptstrategien, wie sie Abbildung 7.4 zeigt, sind die folgenden:

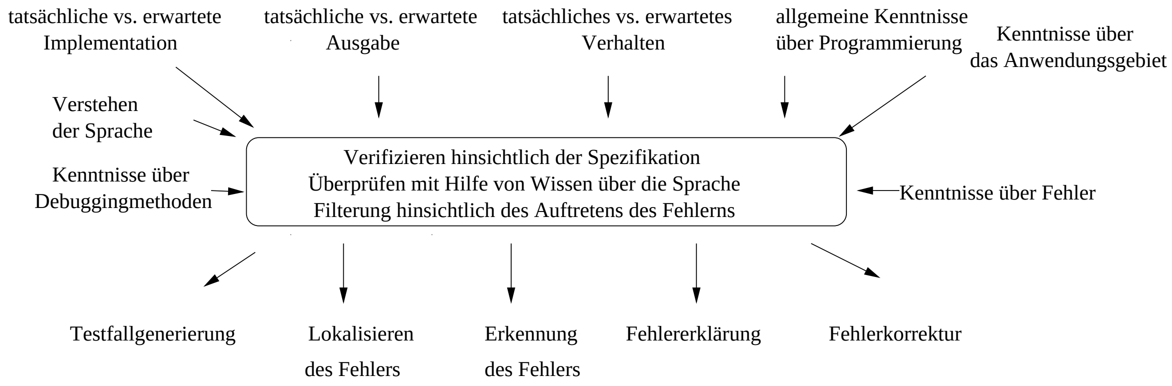



Abbildung 7.4: Die verwendeten Strategien

- *Verifizieren hinsichtlich der Spezifikation*: Das tatsächliche Programm wird mit der Spezifikation des erwarteten Programms verglichen. Die berühmteste Methode, die man hier erwähnen kann, ist das algorithmische Programmdebugging von Shapiro. Diese Technik vergleicht die Durchführung des tatsächlichen Programms mit einer kompletten Spezifikation. Weil es im allgemeinen unmöglich ist, eine solche Spezifikation einzugeben, wird angenommen, daß der Benutzer sie in Form eines *Orakels* ergänzen kann. Die einfachste Vorgehensweise ist die folgende: Die berechneten Zusicherungen werden in allen Ableitungsschritten dem Anwender gezeigt, und er kann entscheiden, ob sie gültig sind oder nicht. Wenn das Programm inkorrekt ist und der Benutzer alle Fragen korrekt beantworten kann, dann muß die fehlerhafte Komponente ein Teil des Programms sein, dessen Verhalten inkorrekt ist, aber dessen Teilkomponenten alle ein korrektes Verhalten zeigen.

  Nach dem Vorbild von Shapiros Arbeit wurde eine Vielzahl von Systemen erstellt ([Fer87] und [Llo87a]).

  Verifizieren hinsichtlich der Spezifikation wird zum großen Teil in Lehrsystemen verwendet (siehe *APROPOS*). Im Fall der Lehrsysteme wird versucht, das erwartete Modell zu finden, das dem tatsächlichen Programm am nächsten kommt. Die Eingabe einer Spezifikation ist nur für sehr kleine Programme möglich.

  Der Nachteil dieser Methode ist, daß die Anzahl der Fragen ziemlich groß werden kann. Eine Möglichkeit, dies zu verbessern, ist, zuerst die wichtigen Fragen herauszufinden und zu stellen ([PC91], [PC88], [HH93], und [PDA93]). Eine andere Verbesserung wäre die Verwendung von partiellen Spezifikationen ([DNTM88b]).

  Algorithmisches Programmdebugging hat auch eine Anwendung bei paralleler logischer Programmierung (*concurrent logic programming*, [LS]), deduktiven Datenbanken ([ST90]) und in der Programmiersprache Gödel ([Bin93]).

  Außerdem gibt es Arbeiten ([LD93b]), die versuchen, den Fehler automatisch zu erkennen, wenn eine komplette Spezifikation vorhanden ist. Dieser Weg ist für Lehrsysteme nützlich, aber wir glauben kaum, daß für realistische Programme eine wirklich vollständige Spezifikation definierbar ist. Algorithmisches Debugging wurde auch für imperative ([Sha91]) und lazy-funktionale Sprachen ([NF92]) angepaßt.

Ein weiterer Nachteil dieser Methode neben der Notwendigkeit einer Spezifikation ist, daß der Benutzer selbst Fehler machen kann und damit den Debuggingprozeß irreführen kann.

- *Überprüfung mit Hilfe von Wissen über die Sprache*: Das Debuggingsystem sucht nach verdächtigen Teilen, die mit bestimmtem explizitem Wissen über die Programmiersprache nicht im Einklang steht. Ein Teil des Programm ist verdächtig, wenn es mit eingebauten Konsistenzregeln nicht übereinstimmt, oder wenn es den im System eingebauten Regeln, die stereotype Fehler beschreiben, entspricht. Ein Vorteil dieser Strategie ist, daß die bei der Fehlersuche benutzten Regeln einen Hinweis darauf geben, was falsch sein kann. Wenn vermutet werden kann, daß der Fehler in einem bestimmten Teil des Programms enthalten ist, ist es nützlich, die Überprüfung hier durchlaufen zu lassen. Diese Strategie kann zum Beispiel auch bei Optimierungsprozessen erfolgreich benutzt werden. Stereotype Fehlererkennung steht auch im Mittelpunkt der Forschung ([Eis85b], [ED90], [Loo88], und [TS85]).

- *Filterung hinsichtlich des Auftretens des Fehlers*: Es werden solche Teile herausgefiltert, die sicher für das fehlerhafte Verhalten nicht verantwortlich sind. Diese Strategie benötigt keinen Quelltext, sondern das Ziel ist die Beschränkung des Codes, der vom Anwender überprüft werden muß. Die wichtigste Methode ist hier die symptomgesteuerte Abhängigkeitsanalyse. Es können zwei Symptome betrachtet werden: falscher Ausgabewert und falsche Kontrollsequenz. Slicing läßt die Teile des Programms weg, von denen der falsche Wert oder der fehlerhafte Teil der Kontrolle unabhängig sind. Der Vorteil dieser Strategie ist, daß sich die Filterung darauf konzentriert, Fehler zu lokalisieren. Deswegen ist sie besser für die Diagnose geeignet als die anderen Strategien. Mit ihr kann das ganze Programm analysiert werden, es müssen nur wenige Informationen eingegeben werden, und als Folge davon ist sie billiger. Filterung eignet sich als vorbereitender Schritt im Debugging.

Je nach Strategie werden verschiedene Techniken verwendet, die wir hier kurz beschreiben möchten.

1. *Beim Verifizieren hinsichtlich der Spezifikation verwendete Techniken*: Alle bei der Verifizierung verwendeten Techniken brauchen entweder eine vollständige oder eine partielle Spezifikation:

   - *Symbolische Durchführung*: Das erwartete Programm ist durch ein Modellprogramm beschrieben. Es wird mit dem *general theorem prover* eine formale Verifikation durchgeführt. Das Ergebnis ist nicht der Fehler selbst, sondern die Angabe, ob beide Programme übereinstimmen oder nicht. Diese Technik ist sehr teuer.
   - *Programmtransformation*: Das als Modellprogramm beschriebene Ergebnisprogramm und das vom Studenten geschriebene Programm werden transformiert. Es wird festgestellt, ob sie übereinstimmen oder verschieden sind. Die unterschiedlichen Teile werden verdächtigt, Fehler zu enthalten. Die hier auftretenden Probleme sind die folgenden:
     - Wie wird die Transformation ausgewählt?
     - Wann wird die Transformation beendet?
     - Die ganze Geschichte der Transformationen muß gespeichert werden.

- *Algorithmische Erkennung*: Das erwartete Programm ist eine Menge möglicher alternativer Algorithmen, die als Lösung des Problems von einem Lehrer dem System eingegeben werden. Die Aufgabe ist, die größte Übereinstimmung zwischen einem Algorithmus aus dieser Menge und einem Algorithmus, der aus dem tatsächlichen Programm abgeleitet wird, zu finden. Dieser wird ausgewählt, und die Unterschiede werden festgestellt. Die hier auftretenden Probleme sind die folgenden:
  - Eine gute Wahl der Repräsentation der Algorithmen.
  - Die Frage ist, ob die Menge der Algorithmen, die eine Lösung zu einem Problem liefern, vollständig ist (ob alle Algorithmen, mit denen man das Problem lösen kann, eingegeben sind).
  - Nur ein Algorithmus kann zu einem kompletten großen Programm ausgewählt werden.
- *Symbolische Ableitung von Zusicherungen*: Das erwartetes Verhalten wird durch vom Anwender erstellte Zusicherungen beschrieben, die das Verhalten der Programmkomponenten spezifizieren. Dann wird für alle Programmkomponenten des tatsächlichen Programms eine Zusicherung symbolisch abgeleitet, oder symbolische Zusicherungen werden zum Programmschema hinzugefügt. Die Zusicherungen aus dem erwarteten Verhalten und die abgeleiteten Zusicherungen werden verglichen. Hier tritt das Problem auf, wie man größere Programme behandeln kann.
- *Konkrete Auswertung der Zusicherungen*: Das erwartete Programm wird wie oben durch vom Anwender erstellte Zusicherungen definiert. Diese Zusicherungen können entweder symbolisch oder konkret sein. Konkret bedeutet, daß sie die erwartete Ausgabe für eine diskrete Menge von Eingaben beschreiben. Die Vorgehensweise ist die gleiche, nur wird hier das spezifizierte Verhalten mit der konkreten Durchführung verglichen. Die bei der Durchführung der Programmteile abgeleiteten Zusicherungen werden mit der Spezifikation verglichen. Diese Technik ist praktischer als die andere, aber weniger allgemein.
- *Konsistenzüberprüfung mit Hilfe von Zusicherungen*: Die Eigenschaften (zum Beispiel Typ, Modus, Kontroll- und Datenabhängigkeiten), die durch Verwendung des Quellcodes automatisch überprüfbar sind, können durch Zusicherungen spezifiziert werden. Dann wird die Konsistenz zwischen der Spezifikation und dem tatsächlichen Programmtext überprüft. Wo Unterschiede gefunden werden, sind vermutlich Fehler.
- *Verifikation des algorithmischen Verhaltens*: Diese Technik heißt auch algorithmisches Debugging. Hier wird das konkrete Verhalten des ganzen Programms mit einer für das beabsichtigte Programm angegebenen Spezifikation verglichen. Es ist hierbei im allgemeinen unmöglich, eine vollständige formale Spezifikation anzugeben. Der Anwender wird in den Prozeß mit einbezogen, in der Annahme, daß er diese Spezifikation vervollständigen kann. Die einfachste Form ist, wenn die tatsächliche Berechnung läuft, in jedem Schritt, wo eine Zusicherung berechnet wird, dem Benutzer die Möglichkeit zu geben zu entscheiden, ob sie gültig ist oder nicht. Wenn er alle Fragen beantwortet hat, dann wird ein Fehler gefunden: Es ist der Teil des Programms, dessen Verhalten inkorrekt ist, aber dessen Komponenten alle ein korrektes Verhalten zeigen. Diese Technik wurde in *IDTS* als Grundtechnik beschrieben, (siehe Kapitel 3) und in späteren Kapiteln erweitert.

2. *Beim Überprüfen mit Hilfe von Sprachwissen angewandte Techniken*: Diese Techniken arbeiten am Programmquelltext und benutzen das Verhalten des Programms.

   - *Schemaerkennung*: Bei dieser Technik enthalten die Schemata das Wissen über die Programmiersprache. Wie Johnson und Soloway für ihr *PROUST*-System ([JS85]) definieren, ist ein Schema eine stereotype Methode für die Implementierung eines Zieles (funktionale Spezifikation), wo die wichtigsten Elemente ausgesucht und explizit definiert werden. Die Technik versucht, die Schemata im Programm zu finden. Der Unterschied zwischen dem vom Anwender erstellten Schema und dem aus dem Programm abgeleiteten Schema wird dann als fehlerverdächtig definiert. Diese Technik ist ähnlich der algorithmischen Erkennung. Die Frage ist nur, wie man komplette Schemata eingeben kann. Für komplexe Programme kann die Anwendung schwierig sein.
   - *Sprachkonsistenz-Überprüfung*: Das Wissen ist in Form von Konsistenzregeln schon im System enthalten. Die Regeln enthalten Informationen, die nicht in einen Übersetzer eingegeben werden können. Diese Technik überprüft den Quelltext und in selteneren Fällen das Verhalten des Programms, ob dieses die Regeln erfüllt. Wenn dies nicht der Fall ist, dann wird dieser Teil als fehlerverdächtig eingestuft. Die Regeln können sogar Erklärungen für die Fehler liefern. Nachteile dieser Strategie sind:
     - Einige Regeln setzen voraus, daß Programmierkonventionen beachtet werden.
     - Je größer das Programm ist, desto wahrscheinlicher ist es, daß es den Programmkonventionen nicht folgt. Wie Ducasse erwähnt, ist deswegen die Auswahl der Empfindlichkeit für die Feststellung des Abweichungsgrades zwischen dem Programm und den Regeln wichtig.
     - Im allgemeinen können nicht alle Fehler erkannt werden, wenn nur Informationen über die Sprache benutzt werden.
     - Es gibt kein Kriterium, um die Vollständigkeit der Regeln zu bewerten.
   - *Stereotype Fehlererkennung*: In den Regeln sind die stereotypen Fehler kodiert. Diese Technik überprüft den Programmquellcode und/oder das Verhalten. Ein Programmteil, der diesen Regeln entspricht, könnte Fehler enthalten. Die Regeln spüren nicht nur den Fehler auf, sondern sie liefern auch eine Erklärung.

3. *Filterung*: Hier wird vorausgesetzt, daß Fehlersymptome erkannt wurden:

   - *Slicing*: Kann als symptomgesteuerte Abhängigkeitsanalyse betrachtet werden: Ein falscher Ausgabewert oder eine falsche Kontrollsequenz zerlegen das Programm und beschneiden den Teil, der diese zwei Fehlerarten (falscher Ausgabewert und falsche Kontrollsequenz) nicht beeinflußt. Was übrig bleibt, wird als die Slice bezeichnet. Die Slice kann mit Hilfe des Programmquelltextes statisch oder dynamisch unter Einbeziehung des Verhaltens berechnet werden. Wir haben diese Technik auf logische Programmierung adaptiert und in *IDTS* integriert, deswegen kann man über Slicing in Kapitel 5 mehr erfahren.
   - *Mutation von Programmen (constraint suspension)*: Die Idee ist, daß kleine Teile des Programms ausgetauscht werden. Dann wird überprüft, ob sich das Programm immer noch inkorrekt verhält. Wenn die Mutation keine Wirkung hatte, also das

Symptom nicht beseitigt wurde, dann wird der Teil, der geändert wurde, um das neue Programm zu bekommen, als korrekt angenommen. Diese Technik kann bei allen Symptomen benutzt werden. Der Nachteil der Technik ist, daß es schwierig ist, die Mutationstransformation geschickt genug auszuwählen, um eine kombinatorische Explosion zu vermeiden. Außerdem kann es vorkommen, daß durch Mutation neue Fehler hinzukommen.

- *Heuristische Filterung*: Diese Technik vermutet nach Verwendung gewisser Kriterien von einigen Programmteilen, daß sie den Fehler nicht enthalten, und beschränkt damit den Suchraum. Wegen der Möglichkeit, daß die ursprüngliche Voraussetzung über die Korrektheit falsch sein könnte, müssen Backtracking-Mechanismen ermöglicht werden.

## 7.5 Die entwickelten Systeme für das Debuggen und Testen von PROLOG-Programmen

In diesem Kapitel werden wir einige Debuggingsysteme vorführen. Wir konnten keine Vollständigkeit erreichen, aber die von uns ausgewählten und hier beschriebenen Systeme wenden die wichtigsten Techniken an, die beim Debugging von PROLOG-Programmen vorkommen. Die meisten Systeme behandeln die zwei üblichen Fehlertypen: inkorrekte und fehlende Lösung. Inkorrekt bedeutet, daß es keine Instanz einer Klausel in **P** gibt, wo der Kopf (conclusion) falsch ist, aber die Prämissen (die Atome auf der rechten Seite) korrekt sind. Der andere Fehlertyp beschreibt, daß es ein Atom gibt, das zu der erwarteten Semantik gehört, aber dieses kommt nicht als Konklusion einer Klauselinstanz der Klauseln vor, deren Prämissen *true* liefern.

### 7.5.1 APROPOS

Das Ziel bei der Entwicklung des Systems *APROPOS* (Analyser of Prolog Programs of Students, [Loo88]) ist, in von Studenten entwickelten Programmen Fehler zu finden und zu korrigieren. Die Autoren selbst nennen das System *PROLOG Intelligent Teaching System* (*PITS*) und bieten damit ein Werkzeug für noch unerfahrene PROLOG-Programmierer, um die Sprache besser lernen zu können. Die Haupteigenschaften des Systems zeigt Abbildung 7.5.

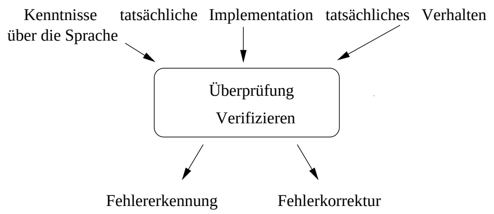



Abbildung 7.5: Das APROPOS-System

Dieses System ist interaktiv: Dem Studenten wird eine Aufgabegestellt, und es wird von ihm wird erwartet, daß er die Prädikate der Lösung eingibt. Danach sucht das System nach den Fehlern in dem vom Studenten eingegebenen Programm. *APROPOS* überprüft nur syntaktisch korrekte Programme in zwei Schritten:

- Das Programm wird zuerst auf allgemeine Abweichungen überprüft. Das bedeutet, daß Fehler gesucht werden, die ein erfahrener Programmierer sofort erkennt:
  - *Erkennung von unendlichen Berechnungen oder Schleifen*: *APROPOS* verwendet hier Datenflußanalyse und überprüft, ob der rekursive Aufruf, der ein Grund für eine unendliche Berechnung sein kann, einer wohldefinierten Ordnung genügt.
  - *Typinformationen*: *APROPOS* leitet die Parametertypen (Ein- und/oder Ausgabe) eines Prädikates aus dem Kopf der Klauseln ab. Wenn dieses Prädikat im Rumpf anderer Klauseln aufgerufen wird, wird überprüft, ob die Argumente dieses Teilzieles bekannt oder abgeleitet sind.
  - *Schreibfehler*: Bei PROLOG kommt es öfter vor, daß sich jemand vertippt, aber der Interpreter den Fehler nicht erkennt (zum Beispiel nur einmal vorkommende (singleton) Variablen).
  - *Unerwünschte Verwendung von Datenstrukturen*: Zum Beispiel im Fall, wenn ein Term $[H|T]$ in einer Klausel auftritt, aber keine anderen Informationen zeigen, daß $T$ eine Liste ist.
- Durch Verwendung von Kenntnissen über die Aufgabe wird versucht, die Fehler zu finden und zu korrigieren. Ein nicht syntaktischer Fehler kann auf drei verschiedenen Ebenen auftreten:
  - *Algorithmik*: Die Lösungsstrategie ist inkorrekt.
  - *Definition der Prädikate*: Die drei Möglichkeiten sind, daß die Lösung des Studenten ein Prädikat nicht enthält, das zur Lösung des Problems notwendig ist, daß er überflüssige Prädikate eingegeben hat, oder daß ein definiertes Prädikat einen Fehler enthält.
  - *Implementation*: Hier konzentriert man sich auf die am häufigsten auftretenden Fehler im Programmtext: fehlende, überflüssige oder falsch eingegebene oder in falscher Reihenfolge definierte Klauseln oder Literale.

*APROPOS* bekommt das vom Studenten geschriebene Programm als Eingabe. Aus diesem gewinnt es Informationen über Modus, Typ, Prädikatnamen, Rekursion, benutzte Programmtechnik, Anzahl der für die Prädikate definierten Klauseln, sowie die Anordnung der Klauseln. Dann wird nach dem am besten übereinstimmenden Algorithmus für alle vom Studenten geschriebenen Prädikate gesucht. Die Algorithmen werden vorher von einem Lehrer eingegeben. Im nächsten Schritt werden die Unterschiede zwischen dem Quellcode des Studenten und der Implementation des Algorithmus festgestellt. Die gefundenen Unterschiede werden mit Hilfe des Fehlersuchprogramms von Shapiro überprüft.

Zusammenfassend läßt sich sagen, daß *APROPOS* die folgenden Techniken verwendet:

- algorithmische Erkennung
- Konsistenzüberprüfung
- Verifizierung des algorithmischen Verhaltens

- Schemaerkennung
- Überprüfung der Sprachkonsistenz

### 7.5.2 PRESET

Die von Takahashi und Shibayama entwickelte Debuggingumgebung für PROLOG-Programme *PRESET* ([TS85], siehe Abbildung 7.6) hat zwei Bausteine:

- Ein Diagnostizierer für Prädikate: Dieser Teil findet den Fehler im Programm.
- Ein *Warum/Warum nicht*-Erklärungssystem: Dieser Teil versucht, mit Hilfe der *Top-Down*-Diagnose den gefundenen Fehler zu erklären, also die Korrektur zu erleichtern.

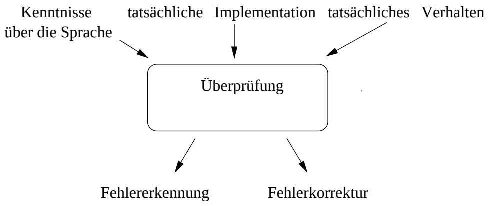



Abbildung 7.6: Das PRESET-System

Die Hauptaufgabe des Diagnostizierers ist die Suche nach statisch erkennbaren Fehlern, also Fehlern, die bereits vor der Durchführung des Programms erkennbar sind. Dies sind die folgenden:

- nicht definierte Klauseln
- ungeeignete Stelligkeit
- nicht reduzierbare Ziele.

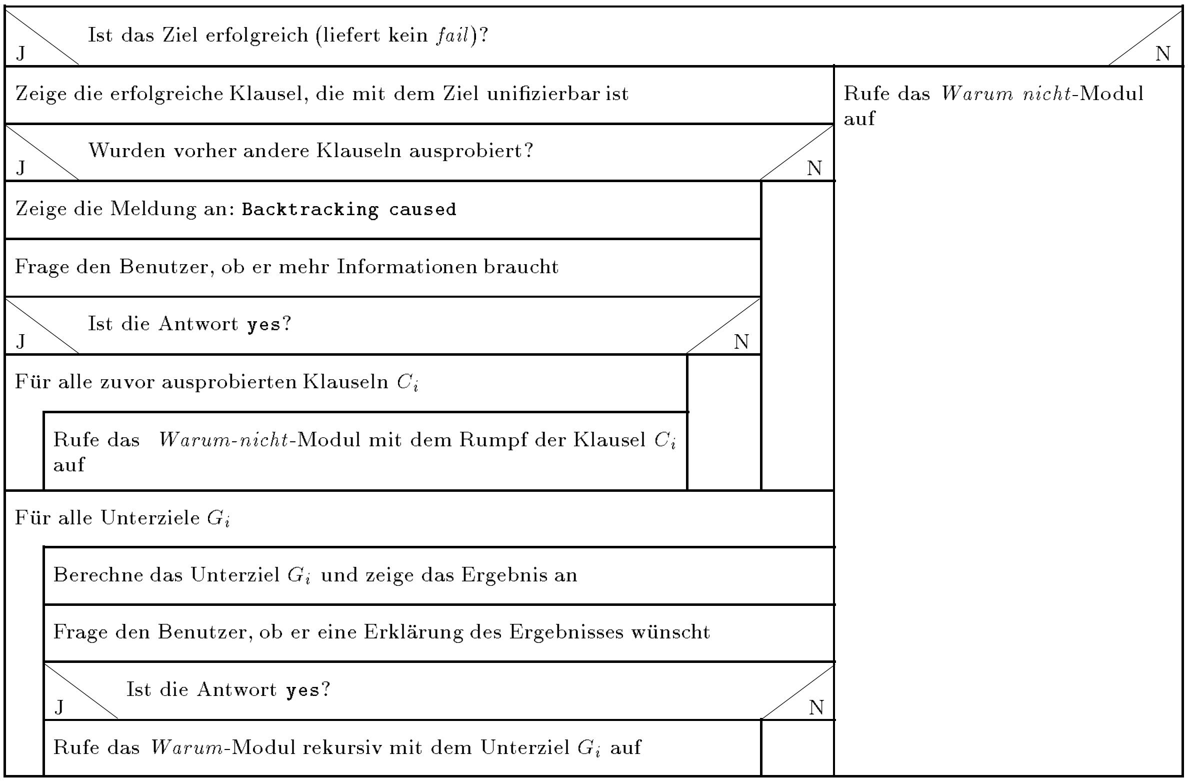
*PRESET* überprüft alle Teilziele in allen Klauselrümpfen. Wenn ein Literal auf der rechten Seite einer Klausel nicht mit einer linken Seite einer Klausel im Programm unifizierbar ist, dann wird das Programm *fail* liefern, wenn es die Klausel benutzt. Alle erkannten Fehler in den Prädikaten werden dem Anwender mit einer Meldung gezeigt, der danach eine von fünf Aktionen durchführen kann:

- Laß das Literal so, wie es ist
- Bearbeite die Prädikate
- Liste die Prädikate auf
- Abbruch (*Abort*)
- Unterbrechung (*Break*)

Das System kann zwei Ausnahmen zulassen: Vom Hauptziel und von während des Programmlaufs mit *assert* hinzugefügten Klauseln wird vermutet, daß sie grundsätzlich korrekt sind. Deswegen erlaubt *PRESET*, diese zu kennzeichnen. Es wird keine Fehlermeldung liefern, selbst wenn sie nach der Systemüberprüfung Fehler enthalten.

Das Erklärungsmodul ist eine attraktive Eigenschaft des Systems. Es verwendet die *Top-Down*-Diagnosemethode, um die Durchführung zu zeigen und die dabei auftretenden Probleme zu erklären: wo und warum das Programm nicht korrekt ist. Wie der Name *Warum/Warum nicht* sagt, handelt es sich eigentlich um zwei Module. Das erste, das *Warum*-Modul (siehe Algorithmus 39), erklärt die Ausführung des Zieles, das *fail* liefert:

**Algorithmus 39** das *Warum*-Modul




Das *Warum nicht*-Modul (siehe Algorithmus 40) erklärt den Durchführungsprozeß, der zum Fehlschlag führt, folgendermaßen:

**Algorithmus 40** Das *Warum-nicht*-Modul

- Ist das Ziel nicht erfolgreich (liefert *fail*)?
  - J: Für alle Klauseln $C_i$, deren Kopf mit dem Ziel unifizierbar ist
    - Zeige die Klausel $C_i$ an
    - Für alle Unterziele $G_j$ der Klausel $C_i$
      - Ist die Ausführung des Unterziels $G_j$ erfolgreich?
        - J:
          - Zeige das Ergebnis an
          - Frage den Benutzer, ob er eine Erklärung des Ergebnisses wünscht
          - Ist die Antwort `yes`?
            - J: Rufe das *Warum*-Modul rekursiv mit dem Unterziel $G_j$ auf
            - N:
        - N:
          - Informiere den Benutzer, daß das Unterziel $G_j$ fehlgeschlagen ist
          - Frage den Benutzer, ob er eine Erklärung des Ergebnisses wünscht
          - Ist die Antwort `yes`?
            - J: Rufe das *Warum-nicht*-Modul rekursiv mit dem Unterziel $G_j$ auf
            - N:
          - Backtracking: Suche eine andere Lösung des jüngsten Unterziels $G_k$, ($k \leq j$) das immer noch Alternativen besitzt
          - Existiert ein solches $G_k$?
            - J: Führe die innerste Schleife ab $G_k$ fort
            - N:
  - N: Rufe das *Warum*-Modul auf

Der Nachteil dieser Vorgehensweise ist, daß immer wieder eine Neuberechnung durchgeführt werden muß, deswegen arbeitet das System sehr langsam. Außerdem bietet dieser Algorithmus keine Behandlung für Programme mit Endlosrekursion.

### 7.5.3 APD

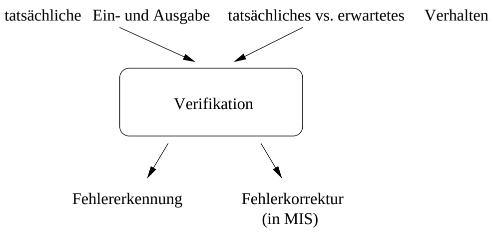



Abbildung 7.7: Das APD-System

Auf diesem System basiert das von uns entwickelte Interaktive Debugging und Test System. Das *APD*-System ( Algorithmic Program Debugging, [Sha83], siehe Abbildung 7.7) beschreiben wir detailliert im Kapitel 3. Deswegen geben wir hier nur einen kurzen Überblick. Es benutzt die Kenntnisse der tatsächlichen Ein- und Ausgabe und Kenntnisse über das tatsächliche Verhalten. Es erfüllt die Aufgabe der Lokalisierung des Fehlers und in der

Weiterentwicklung (Model Inference System, *MIS*) seiner Korrektur. Der Anwender ruft den Debugger mit dem vermutlich fehlerhaften Programm mit der tatsächlichen Ein- und Ausgabe auf. Dieses Programm wird durchgeführt, und das System liefert den Trace mit dem tatsächlichen Verhalten. Der Trace traversiert den Ableitungsbaum. Das System befragt den Anwender und vergleicht das erwartete und das gelieferte Verhalten. Auf diese Weise kann es den Fehler feststellen. Dieses System kann drei verschiedene Arten von Fehlern erkennen: falsche oder fehlende Lösung (siehe Kapitel 3.1 und 3.2) und unendliche Schleifen (siehe Kapitel 3.3). Der Traversierungsmechanismus des Ableitungsbaums kann Bottom-Up (Kapitel 3.1.1), Top-Down (Kapitel 3.1.2) oder Divide-and-Query (Kapitel 3.1.3) sein.

*APD* ist ein Debugger für PROLOG-Programme, eine deklarative Debuggingmethode. Die Aufgabe des Anwenders besteht darin, die vom Debugger über das abstrakte Verhalten und über die Ein- und Ausgabe gestellten Fragen zu beantworten. Es wird vorausgesetzt, daß der Benutzer diese Fragen korrekt beantworten kann. Mit anderen Worten, das System verwendet als einzige Strategie die Verifizierung des algorithmischen Verhaltens. Dies ist ein Nachteil des Systems, weil einerseits die Anzahl der Fragen groß sein kann und andererseits die Korrektheit der vom Anwender gegebenen Antworten nicht sicher ist. Diese Nachteile abzuschwächen oder aufzuheben, dafür sorgt unser *IDTS*-System (siehe Kapitel 6).

### 7.5.4 RD

Das von Pereira entwickelte System *RD* (Rational Debugging, [Per86], siehe Abbildung 7.8) unterstützt die Fehlersuche in logischen Programmen. Das System ist für die Aufgaben, Fehler zu lokalisieren und zu erklären, gedacht. *RD* benutzt bei der Fehlersuche Kenntnisse über die tatsächliche Ein- und Ausgabe und das Verhalten und verwendet dabei zusätzlich die deklarative und operationelle Semantik des logischen Programms. Der Debugger basiert auf Abhängigkeitsrelationen (siehe Kapitel 5.1.1).

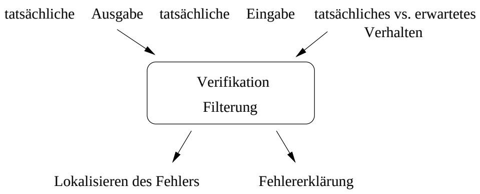



Abbildung 7.8: Das RD-System

Dieses System kann zwei verschiedene Fehlertypen erkennen: widersprüchliche (inkonsistente) und unvollständige Klauseln. Der Anwender hilft, das erwartete Verhalten festzustellen, indem er die vom System gestellten Fragen beantwortet. Die gestellten Fragen können, wenn das Ziel zulässig ist ( *admissible*), über die Instanz der Klausel gestellt werden, oder ob das gelöste Ziel korrekt oder falsch ist, und als letztes, ob die Lösung vollständig ist. Ein Ziel ist zulässig, wenn das Ziel textuell korrekt ist und die Argumente den korrekten Typ haben. Ein lösbares Ziel kann bei erwartetem Verhalten eine Lösung liefern.

Dieses System kann, wie Pereira und Calejo behaupten ([PC88]), auch Seiteneffekte in einem PROLOG-Programm behandeln. Rationalisierung bedeutet, wie die Autoren definieren, daß die gestellten Fragen nicht nur hinreichend, sondern auch notwendig sein müssen. Pereira behauptet, daß die dynamisch erzeugten Abhängigkeitsrelationen eine Basis für eine benutzbare Debuggingmethode sein können. Diese können bei beiden Fehlertypen verwendet werden: Bei falscher Lösung kann man damit den Fehler finden, und bei fehlender Lösung kann der Suchraum für die Identifizierung begrenzt werden. Also gibt es zwei voneinander gut abgrenzbare Aufgaben:

- Erzeugen der Abhängigkeitsinformationen.
- Erzwingen von direktem Backtracking zu einem bestimmten Knoten, der dann auf Fehler untersucht wird.

Pereira hat den Unifikationsalgorithmus modifiziert, so daß dieser die Abhängigkeiten behält. Dieser Algorithmus erzeugt einen skelettartigen Ableitungsbaum, bei dem alle Substitutionen ignoriert werden. Dieser Baum wird bei der Ableitung vergrößert. Formal kann er wie folgt definiert werden, wobei $t - T$ ein Term $t$ mit dem angeschlossen Ableitungsbaum $T$ ist. Unter der Vereinigung zweier Ableitungsbäume kann man sich hier das Nebeneinanderstellen der Bäume vorstellen.

**Definition 99** skelettartiger Ableitungsbaum

- Die Unifikation von $t - T_1$ und $t - T_2$ generiert eine leere (*empty*) Substitution mit dem Ableitungsbaum $T_1 \cup T_2$. Falls $T_1$ und $T_2$ verschieden sind, könnte der Fehler darin liegen, daß die Terme fälschlicherweise gleich sind.
- Die Unifikation von $f(t_1, \ldots, t_m) - T_1$ und $g(t_1, \ldots, t_n) - T_2$ erzeugt *fail* mit dem Ableitungsbaum $T_1 \cup T_2$ als einem inkonsistenten Ableitungsbaum.
- Die Unifikation von $f(t_1, \ldots, t_n) - T_1$ mit $f(r_1, \ldots, r_n) - T_2$ ($n > 0$ und $t_i \neq r_i, 1 \leq i \leq n$) vereinigt alle $t_i - T_1$ mit $r_i - T_2$.
- Die Unifikation $X_1 - T_1$ und $t_2 - T_2$, wobei $X_1$ keine freie Variable ist und eine Substitution $X_1/t$ mit dem Ableitungsbaum $T$ existiert, unifiziert $t - T_1 \cup T$ mit $t_2 - T_2$.
- Die Unifikation von $X_1 - T_1$ und $t_2 - T_2$, wobei $X_1$ eine freie Variable ist, generiert die Substitution $X_1/t_2$ und den Ableitungsbaum $T_1 \cup T_2$.

Wie wir schon erwähnt haben, können zwei Fehlerarten aufgespürt werden. Dementsprechend wurden zwei Hauptalgorithmen entwickelt: Einer davon findet die falsche Lösung auf der andere berechnet die fehlende Lösung.

Die Grundidee bei der Aufklärung der falschen Lösung ist die Verwendung einer Art Vorwärtsstrategie, die die existierenden Abhängigkeiten im Ableitungsbaum betrachtet. Außer diesen Abhängigkeiten und dem Durchführungsweg muß sie die vom Anwender gegebenen Antworten, die auf die das tatsächliche Verhalten betreffenden Fragen gegeben werden, speichern. Diese auf der operationellen Semantik basierende Methode braucht den Quellcode und Abhängigkeiten, wobei letztere vom System selbst erzeugt werden. *RD* durchsucht den Ableitungsbaum rückwärts ab dem Erkennen des Fehlers. Das System erwartet *yes*- und *no*-Antworten auf die dem Anwender gestellten Fragen. Außerdem fragt es nach, ob das gerade untersuchte Ziel zulässig (*admissible*) oder lösbar (*solvable*) ist.

### 7.5.5 OPIUM

Das von Ducasse entwickelte *OPIUM*-System [Duc87] ist ein Trace-Analysator für PROLOG-Programme und kann den Quellcode eines Programms und den Trace (das tatsächliche Verhalten des Programms bei der Durchführung) während der Durchführung des Programms analysieren. Ducasse hat eine Traceabfragesprache in PROLOG entwickelt. Damit kann der Benutzer genau eingeben, welche Zeilen er sehen möchte. Außerdem unterstützt eine Datenbank das Backtracking. Das System berechnet eine Darstellung der Durchführung des PROLOG-Programms. Damit wird erreicht, daß der Anwender einerseits das Verhalten des Programms verstehen kann und andererseits sogar verstehen kann, warum die Fehler aufgetreten sind. Dazu braucht das System nur das tatsächliche Verhalten des Programms als Eingabe (siehe Abbildung 7.9).

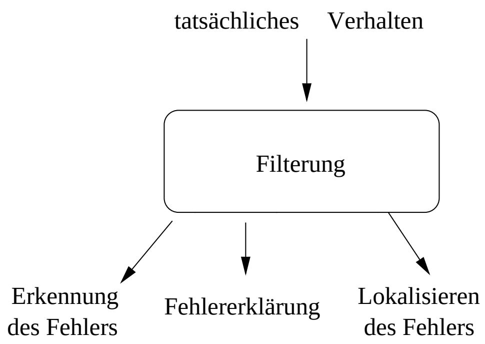



Abbildung 7.9: Das OPIUM-System

Wir beschreiben hier die interessanteste Eigenschaft des Systems, die Traceabfragesprache, ein bißchen näher. Ducasse definiert die Durchführung eines Programms als Strom (*stream*) von Ereignissen (*events*), die ähnlich repräsentiert und analysiert werden können. Nach ihrer Ansicht kann mit Hilfe von zwei Primitiven die Traceinformation einfach beschrieben werden:

- durch ein Primitiv, das Informationen liefert, die zum aktuellen Ereignis gehören
- durch ein Primitiv, das Informationen liefert, die zum nächsten Ereignis gehören

Dann ist der Trace nichts anderes als die Reihenfolge der Durchführung dieser Ereignisse. Die zwei Primitive bilden mit Hilfe von PROLOG eine ausdrucksvolle Traceabfragesprache.

Das wichtigste dabei ist, daß Haltepunkte (*breakpoints*) noch während des Debuggingvorgangs angelegt werden können. Betrachten wir das folgende Beispiel:

```
?- next(Event), CONDITION(Event), DISPLAY(Event).
```

Laut Ducasses Arbeit bedeutet diese Zeile: Wiederhole solange das nächste Ereignis, bis die Bedingung `CONDITION` gilt, dann zeige (`DISPLAY`) dieses Ereignis an.

Der Nachteil dieser Methode ist, wie Ducasse auch selbst erkannt hat, daß diese Vorgehensweise nicht sehr effizient ist, wenn die Durchführung lang ist oder die Datenstrukturen groß sind. Deswegen hat Ducasse die Sprache um mehrere Anweisungen erweitert, die für die Optimierung des Prozesses sorgen.

Ducasse hat auch die Notwendigkeit einer Tracedatenbank erkannt. Damit kann man zu schon durchgeführten Teilen des Programms zurückkehren. *OPIUM* speichert die Ereignisse in chronologischer Reihenfolge. Hier tritt das Problem der Datenbankgröße auf. Darin sieht sie kein Problem, weil sie behauptet, daß die Datenbanken sowieso nicht permanent sind. Wir sind anderer Meinung: Damit die Ereignisse einer PROLOG-Debuggingsitzung zu einem späteren Zeitpunkt fortgesetzt werden können, sollten die bisherigen Ereignisse gespeichert werden. Das kann *OPIUM* im Gegensatz zu *IDTS* nicht bieten. Was *OPIUM* kann, ist, daß es dem Anwender erlaubt, die Größe der Datenbank zu kontrollieren.

Wegen des Abarbeitungsmechanismus von PROLOG ist es kompliziert, den Zusammenhang zwischen der Durchführung des Programms und dem Programm selbst nachzuvollziehen. Um das Ergebnis eines Aufrufs eines Zieles zu erfahren, muß man große Teile der Durchführung mit dem Tracer analysieren. *OPIUM* ermöglicht es, die irrelevanten Informationen wegzulassen und die übriggebliebenen in einer kompakten Form anzuzeigen.

Es gibt verschiedene Stellen, an denen Abstraktionen anwendbar sind:

- Kontrollflußinformationen können ausgewählt werden.
- Die Daten im Programm können in verschiedenen Formen auf den Bildschirm geschrieben werden.
- Der Datenfluß einiger Variablen kann angezeigt werden.
- Konzentriert auf einige Erscheinungen des Fehlers können Traceinformationen ausgewählt werden.

### 7.5.6 PTP

Die Arbeit von Eisenstadt *PTP* (Prolog Trace Package, [Eis85b], [Eis85a], siehe Abbildung 7.10) ist ein System, das beim Debugging das Trace verwendet (Tracesystem). Es wurde für logische Programme entwickelt und basiert auf einer selektiven rückwirkenden Analyse des Laufzeittrace des Programms.

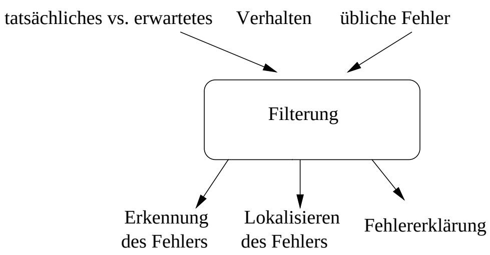



Abbildung 7.10: Das PTP-System

Wie Eisenstadt beschreibt, wurden schon früher verschiedene Systeme entwickelt. Die meisten davon eignen sich für Sprachen, die zur PROLOG-10-Familie gehören (siehe 2.2.5). Aber diese haben einige Nachteile:

- Sie folgen dem sogenannten Boxen-Modell (siehe [Byr80]). Die dazu gehörenden Schlüsselwörter sind *call*, *fail*, *exit* und *redo*. Sie erschweren den Überblick des Ablaufs, weil sie alle aus vier Buchstaben bestehen.

- Sie zeigen nicht die verschiedenen Unifikationsmöglichkeiten, wenn ein Ziel mit mehreren Klauselköpfen unifizierbar ist.
- Der Wert der unifizierten Variablen wird nur bei *exit* angezeigt.
- Es gibt keine Ausrichtung der Ausgabe der Teilzeilen zwischen den verschiedenen Ebenen.
- Wir bekommen keine Informationen über Systemprädikate, wie zum Beispiel *open*.
- Es gibt mehrere Möglichkeiten, daß ein PROLOG-System *fail* liefert, aber die Trace-Systeme liefern keinerlei Information darüber, aus welchem Grund an einer bestimmten Stelle *fail* zurückgegeben wurde.

Wegen der oben erwähnten Schwierigkeiten des Boxen-Modells hat Eisenstadt die Ableitung eines PROLOG-Zieles als Suche in einer Datenbank definiert: die Aufgabe ist, einen Klauselkopf zu finden, der mit dem Ziel unifizierbar ist. Wenn wir eine Ableitung so betrachten, dann liefert ein Ziel *fail*:

- Wenn ein Teilziel *fail* zurückgegeben hat und es keine anderen Resolventen gibt.
- Wenn ein Teilziel mit *fail* zum *Cut*-Symbol zurückgekehrt ist.
- Wenn überhaupt keine linke Seite existiert, die mit dem Teilziel unifizierbar wäre.
- Wenn die definierten Klauselnamen den gleichen Namen, aber nicht die gleiche Stelligkeit haben.
- Wenn die Variablen nicht unifizierbar sind.

Die letzten drei können zusammengefaßt werden, indem man sagt, daß es keinen Resolventen mit dem Klauselkopf gibt. Das hier beschriebene System bietet die folgenden Möglichkeiten:

- Getrennte Behandlung von sechs verschiedenen Fehlern:
    - Teilziel liefert *fail*
    - Keine andere Backtrackinglösung
    - Backtracking begegnete *cut*
    - Keine Definition des Prädikats
    - Falsche Stelligkeit
- Grafische Markierung des *Cut*-Symbols
- Ausgabe der Klauselnummer, die mit dem Ziel unifizierbar ist
- Darstellung der Variablen-Instanz, nicht nur bei *call* und *exit*
- Automatisches Kennzeichnen und Analyse der beim Debugging ermittelten verdächtigen Stellen
- Rückwirkendes Zoomen der Einzelheiten des Traces

- Anzeige der aktuell überprüften Klausel in dem vom Anwender geschriebenen Quellcode (beschränkt, nur für *VAX*-Maschinen implementiert)

Dieser Trace kann auch bestimmte Details bei der Durchführung des Programms auslassen (*skip*: innerer Teil wird ausgelassen, *leap*: geht zum nächsten Haltepunkt, *creep*: schaltet auf Einzelschritt-Ausführung um, *retry*: geht zum letzten ausgegebenen Ziel zurück). *PTP* speichert alle Traceinformationen, der Anwender kann das Ergebnis vorab sehen, und er kann mit Hilfe der verschiedenen Befehle verschiedene Einzelheiten über die interessantesten Teilziele erfahren. Im letzten Punkt haben wir erwähnt, daß das System auch Quellcode anzeigen kann. Diese Eigenschaft ist allerdings maschinenabhängig und wurde nur für *VAX*-Maschinen implementiert.

Zusammenfassend läßt sich sagen, daß das *PTP*-System viele wichtige Funktionen enthält, aber die Bedienung des Systems ist sehr schwerfällig: *PTP* benutzt neunzehn verschiedene Symbole, um den Trace zu steuern.

### 7.5.7 DED

J. W. Lloyd hat einen Debugger *DED* (Declarative Error Debugging) entwickelt, der *deklarativ* ist: Der Anwender braucht nur die Ein- und Ausgabe und das erwartete Verhalten des inkorrekten Programms zu kennen (siehe Abbildung 7.11). Sein System kann nicht nur in einem *puren* PROLOG-Programm, sondern auch in einem solchen, das die erweiterte Syntax und die erweiterten Kontrolleigenschaften enthält, nach Fehlern suchen. Lloyd konzentriert sich allerdings auf falsche oder fehlende Lösungen und beschäftigt sich nicht mit unendlichen Schleifen.

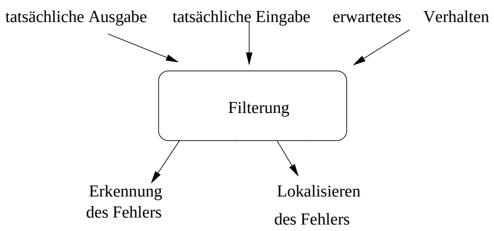



Abbildung 7.11: Das DED-System

Er hat auch erkannt, daß der Idealfall, daß alle Programmierer nur *pure* PROLOG-Programme schreiben, nicht ausreichend ist. Er erwähnt einige Ergebnisse der Forschung wie Einführung von Kontrollhilfen ([Nai85a]), Einführung der Behandlung von Negation ([Nai85b]) oder Verwendung von Programmtransformationen mit dem Ziel, daß der Programmierer ein Programm in eine Form bringen kann, die näher zu der Spezifikation steht ([TS84]).

Zu seiner Zeit hat er es sich so vorgestellt, daß die Entwicklung der logischen Programmierung in diese Richtung weitergehen wird, daß nämlich die Übersetzer alle diese Techniken integrieren würden und die Aufgabe der Programmierer darin bestehen würde, Formeln erster Ordnung einzugeben. Aus diesen generiert der Rechner ein *Assembler*-Programm, wobei Kontrollinformationen automatisch hinzugefügt werden. Außerdem werden Transformationen angewandt. Im zweiten Schritt wird dieses *Assembler*-Programm in ein *Maschinen*-Programm konvertiert, das dann zum Beispiel auf Warrens abstrakter PROLOG-Maschine laufen kann ([War83]).

Er sah eindeutig die Zukunft, daß die Debuggingsysteme *deklarativ* werden, und er versuchte, einen deklarativen Debugger zu entwickeln. In seiner Arbeit hat er ([Llo87a]) mit Hilfe von Definitionen, die wir im Kapitel 2 eingeführt haben, theoretisch die zwei Fehlermöglichkeiten (falsche beziehungsweise fehlende Lösung) beschrieben. Damit hat er erreicht, daß er die Gültigkeit (*soundness*) und die Vollständigkeit des Diagnostizierens mathematisch beweisen konnte.

Er hat das System in MU-PROLOG als einen *Top-Down*-Diagnostizierer implementiert. Dieser *Top-Down*-Diagnostizierer ist gültig (*sound*), aber die Vollständigkeit ist fraglich. Dieses Programm kann nicht-deklarative Eigenschaften wie *cut* und *assert* jedoch nicht behandeln.

### 7.5.8 EDS

Das *EDS*-System (Error Diagnosis System) ist eine theoretische Weiterentwicklung des Systems von Shapiro. In seiner Arbeit ([Fer87]) hat Ferrand mit der Methode von Shapiro angefangen und versuchte im theoretischen Gerüst, die Fehler und die Fehlersuche zu beschreiben. Deswegen hat er sich nicht mit PROLOG selbst, sondern allgemein mit logischen Programmiersprachen beschäftigt. Aber die theoretischen Ergebnisse wurden nicht in ein praktisches System umgewandelt.

Er hat die Fehlerdiagnose von Shapiro theoretisch im Rahmen von Horn-Klauseln überprüft. Er hat dabei die Semantik von Clark ([Cla79]) verwendet, die erlaubt, daß das Herbrand-Universum Variablen enthält. Er braucht als Eingabe das fehlerhafte Programm, meistens eine vom Anwender (Orakel) eingegebene erwartete Semantik und natürlich einige Informationen über die Symptome des Fehlers (siehe Abbildung 7.12). Letzteres kann zum Beispiel irgendetwas sein, das falsch läuft, wenn das Programm durchgeführt wird.

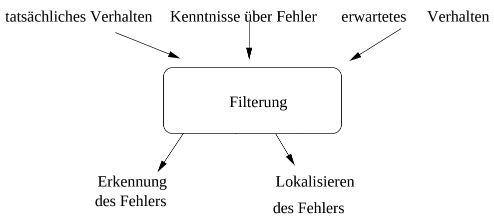



Abbildung 7.12: Das EDS-System

In seiner Arbeit hat Ferrand den Anwendungsbereich der Fixpunkt- und deklarativen Semantik in der logischen Programmierung (siehe im Kapitel 2.1.3) so erweitert, daß die Semantik auch Variablen enthält. Damit wollte er erreichen, daß der Fehler klar und genau formuliert werden kann. Einige Ergebnisse sind die gleichen wie bei Shapiro, aber die Rechtfertigung ist anders. Er behauptet, daß er den Zusammenhang zwischen der Fehlerstelle und dem Fehler selbst beschreiben kann. Im klassischen Fall sprechen wir bei der Semantik für logische Programme von Termen ohne Variablen, vom Herbrand-Universum (siehe im Kapitel 2.1.3), hier handelt sich um eine Semantik mit Variablen. Aus der anderen hier verwendeten Definition der Semantik folgt, daß diese Arbeit als eine Erweiterung der Shapiro-Methode betrachtet werden kann.

Ferrand behandelt theoretisch die zwei üblichen Fehlertypen: inkorrekte und fehlende Lösung. Er hat die Bedeutung des logischen Programms definiert; diese Bedeutung muß *re-*

*lativ* zum fehlerhaften Programm sein. Er hat also die erwartete Bedeutung definiert. Der Grund der Erweiterung der Semantik der logischen Programmierung ist, daß die Semantik eines festgelegten Diagnoseprogramms *relativ* zu allen fehlerhaften Programmen und zu der erwarteten Semantik sein muß.

Diese Arbeit ist eine umfangreiche Beschreibung der Theorie auf dem Gebiet des Debuggings von logischen Programmen, aber weil sie sich mit logischer Programmierung statt PROLOG beschäftigt und die Übertragung der theoretischen Ergebnisse in ein praktisches System nicht durchgeführt wurde, ist es mit wirklich anwendbaren Systemen kaum zu vergleichen.

### 7.5.9 ADAss

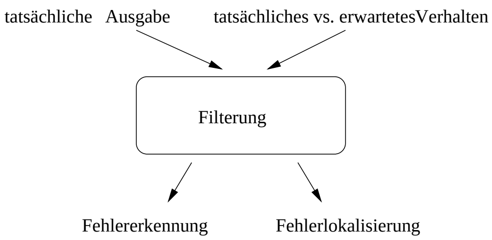



Abbildung 7.13: Das ADAss-System

Auch *ADAss* (Automatic Debugging with Assertions) ist eine Weiterentwicklung von Shapiro. Drabent, Tehrani und Małuszyński sind zu der gleichen Folgerung gekommen wie wir im *IDTS*-System: Die Anzahl der dem Anwender gestellten Fragen muß reduziert werden, wenn die algorithmische Debuggingmethode verwendet wird. Sie haben eine Vorgehensweise entwickelt, bei der sie *Zusicherungen* (*assertions*, formale Beschreibungen der Eigenschaften des erwarteten Programms) eingeben (siehe Abbildung 7.13). Selbst Shapiro hat angedeutet, daß die Einführung von Beschränkungen (*constraints*) und partiellen Spezifikationen die Anzahl der Anwenderinteraktionen vermindert.

Das System von Shapiro braucht bestimmte Informationen über das erwartete Modell (eine Teilmenge von $\mathcal{I}_{\mathbf{P}}$ oder *Herbrand-Basis*), diese definieren Drabent et al. folgendermaßen:

- *Endliche Menge der erwarteten Modelle*: Die Elemente sind die vom Anwender gegebenen *yes*-Antworten (*Grundorakel*), und dazu gehören auch die vom Anwender gelieferten Lösungen, die Elemente des erwarteten Modells und eine Instanz des gegebenen Ziels sind (*Existenzorakel*).

- *Endliche Menge von Kopfelementen des erwarteten Modells*: Die Elemente sind die *no*-Antworten des Grundorakels.

- *Endliche Menge der das erwartete Modell ausführenden Atome*: Die Elemente sind die *yes*-Antworten des Existenzorakels.

- *Endliche Menge der Atome, die im erwarteten Modell unerfüllbar sind*: Die Elemente sind die *no*-Antworten des Existenzorakels.

Gegenüber $\mathcal{I}_{\mathbf{P}}$ ist das Modell des Programms $\mathbf{P}$ mit $\mathcal{M}_{\mathbf{P}}$ gekennzeichnet und kann, wie schon im Kapitel 2.1.3 erklärt wurde, mit dem kleinsten Herbrand-Modell definiert werden. Dann sagen wir, daß ein Programm *inkorrekt* ist, wenn $\mathcal{M}_{\mathbf{P}} - \mathcal{I}_{\mathbf{P}} \neq \emptyset$, und es ist *unzureichend* (*insufficient*), wenn $\mathcal{I}_{\mathbf{P}} - \mathcal{M}_{\mathbf{P}} \neq \emptyset$. Mit diesen zwei Fällen haben Drabent et al. sich beschäftigt, und entsprechend den oben beschriebenen Fragearten haben sie vier verschiedene Zusicherungen ausgearbeitet. Diese formalen Spezifikationen werden während des Debuggings interaktiv eingegeben und vom Diagnosesystem verwendet, um die Fehler im erwarteten Modell zu lokalisieren. Immer, wenn sie nicht hinreichend sind, wird weiterhin der Anwender gefragt. Mit der Antwort werden die Systemkenntnisse vergrößert. Für die neue Spezifikationsmethode haben sie einen neuen Algorithmus entwickelt, in dem der Anwender nicht die Instanz des gegebenen Atoms liefern muß (ob sie im erwarteten Modell ist), sondern es wird vom Anwender erwartet, die Lösungen des Zieles zu erkennen und die Fälle zu identifizieren, in denen die Antworten fehlen.

Aber bleiben wir bei den Zusicherungen und sehen wir, wie diese beschrieben werden können und welche Arten es überhaupt gibt. Die Zusicherungen enthalten außer *yes/no*-Antworten eine Beschreibung der Eigenschaften des erwarteten Modells. Zusicherungen bilden eine nicht unbedingt endliche Menge aus nicht nur Grundatomen der Sprache. Als Sprache für Zusicherungen eignet sich die logische Programmiersprache, weil zum Beispiel bei der Beschreibung der Zusicherungen alle Standardbibliotheken verwendet werden können. Natürlich ist die Eingabe einer vollständigen Spezifikation unmöglich. Einige Autoren schlagen vor, daß bei der Ausarbeitung einer neuen Version des Programms das alte Programm als Spezifikation verwendet wird, aber auch hierbei treten Schwierigkeiten auf.

Der Anfangspunkt der Autoren war, die Zusicherungen so einfach wie möglich zu halten. Sie haben vier Prädikate entwickelt, um die Eigenschaften des erwarteten Modells beschreiben zu können. Sie werden mit der Bezeichnung $\mathcal{A}_s(\mathcal{I}_{\mathbf{P}})$ zusammengefaßt. $\mathcal{A}_s(\mathcal{I}_{\mathbf{P}})$ ist ein Metaprogramm, das die Menge der Atome der Sprache des Programms $\mathbf{P}$ spezifiziert. Deswegen sind die Terme $\mathcal{A}_s(\mathcal{I}_{\mathbf{P}})$ eine passende Repräsentation für die Atome des Programms $\mathbf{P}$. Wenn $A$ ein Atom ist, das die Variablen $X_1, \ldots, X_n$ enthält, dann ist unter dem Repräsentationsschema sein Bild $A'$, bei dem $X_i$ mit *VAR(i)* $(1 \leq i \leq n)$ substituiert wurde.

Die vier verschiedenen Arten von Zusicherungen sind die folgenden:

- *Positive Zusicherungen*: Sie werden benutzt, um die Atome, die im erwarteten Modell gültig sind, zu definieren. Dies geschieht mit dem Prädikatsymbol *true*. Wenn $\mathcal{A}_s(\mathcal{I}_{\mathbf{P}}) \models true(A')$ und $A'$ ein Bild von $A$ ist, dann gilt für alle Substitutionen $\theta$ $A\theta \in \mathcal{I}_{\mathbf{P}}$.

- *Negative Zusicherungen*: Sie spezifizieren die Menge von Atomen, die nicht gültig sind. Hier wird das Prädikat *false* verwendet: $\mathcal{A}_s(\mathcal{I}_{\mathbf{P}}) \models false(A')$ und $A'$ ist ein Bild von $A$, dann gibt es keine Substitution $\theta$, so daß $A\theta \in \mathcal{I}_{\mathbf{P}}$ und $A\theta$ gebunden ist.

- *Positive existentielle Zusicherungen*: Sie beschreiben die Atome, die im erwarteten Modell erfüllbar sind. Mit Hilfe des Prädikates *posex* kann die Beschreibung folgendermaßen eingegeben werden: $\mathcal{A}_s(\mathcal{I}_{\mathbf{P}}) \models posex(A')$ und $A'$ ist ein Bild von $A$, dann existiert eine Substitution $\theta$, so daß $A\theta \in \mathcal{I}_{\mathbf{P}}$ ist.

- *Negative existentielle Zusicherungen*: Sie dienen für die Atome, die in dem erwarteten Modell unerfüllbar sind. Hier nehmen wir das Prädikat *negex*. Wenn $\mathcal{A}_s(\mathcal{I}_{\mathbf{P}}) \models negex(A')$ und $A'$ ein Bild von $A$ ist, dann gilt für alle Substitutionen $\theta$, daß $A\theta \notin \mathcal{I}_{\mathbf{P}}$ ist.

Es ist wichtig, daß bei jedem Entwicklungsstand das Programm $\mathcal{A}_s(\mathcal{I}_\mathbf{P})$ ein erwartetes Modell beschreiben muß. Die Konsistenz der Beschreibung wird hier nicht überprüft, das bleibt die Aufgabe des Anwenders.

Als nächstes erklären wir kurz, wie der Algorithmus funktioniert. Fangen wir an mit den Fragen. Der Algorithmus stellt drei verschiedene Fragen:

- *universelle Fragen*: Ist die atomare Formel im erwarteten Modell gültig?
- *existentielle Fragen*: Ist $A$ im erwarteten Modell erfüllbar?
- *Unvollständigkeitsfragen*: Gibt es für das Atom $A$ eine Instanz $A\theta \in \mathcal{I}_\mathbf{P}$, so daß $A\theta$ keine Instanz der Menge $\{A\theta_1, \ldots, A\theta_n\}$ der gefundenen Lösungen ist?

Das System versucht zuerst, diese Fragen mit Hilfe von Zusicherungen zu beantworten. Bei universellen Fragen werden positive, negative und negativ existenzielle Zusicherungen benutzt. Auf ähnliche Weise können positiv existenzielle, negativ existentielle und positive Zusicherungen bei existenziellen Fragen verwendet werden. Darüber kann mehr in [DNTM88a] nachgelesen werden.

Die Entwickler des *ADAss*-Systems haben sich auf zwei verschiedene Fehler konzentriert: Erkennen der Inkorrektheit und Diagnose des unzureichenden (insufficient) Verhaltens. Sie haben sich mit dem Fehler der unendlichen Schleife nicht beschäftigt. Im ersten Fall wird der Ableitungsbaum mit der Wurzel, die nicht in $\mathcal{I}_\mathbf{P}$ ist, konstruiert, wobei nicht alle Knoten Grundatome sind. Das bedeutet, daß hier universelle Fragen in *Top-Down*-Reihenfolge gestellt werden müssen. Hier wird als Ausgabe eine nicht unbedingt gebundene Klauselinstanz geliefert. Der zweite Algorithmus braucht als Eingabe ein Atom, für das das Programm nicht alle Ausgaben liefert. Als Ergebnis erkennt es ein Atom $A$, für das eine Instanz $A\theta \in \mathcal{I}_\mathbf{P}$ existiert, so daß alle Atome in einer Klausel in $\mathcal{I}_\mathbf{P}$ sind und der Kopf $A\theta$ ist. Für ein gegebenes Ziel wird der *top level trace* überprüft. Es werden existentielle Fragen gestellt, wenn ein Teilziel *fail* liefert, und Unvollständigkeitsfragen, wenn das Teilziel erfolgreich durchgeführt wurde. Dann wird der Algorithmus rekursiv für ein Atom aufgerufen, für das die Antwort *yes* war. Wenn alle angegebenen Antworten *no* sind, wurde das unvollständige Atom gefunden. Diese Algorithmen wurden implementiert und in das System *ADAss* integriert.

Ein Nachteil des Systems ist, daß es auch Fragen zu Systemprädikaten stellt. Weiter kann es vorkommen, daß die gleiche Frage zweimal gestellt wird. Außerdem meinen wir, daß die Eingabe der Zusicherungen nicht einfach ist und tiefe Kenntnisse über PROLOG erfordert. Darbent et al. haben nur reine logische Programme betrachtet und damit das Anwendungsgebiet des Systems beschränkt.

## 7.6 Zusammenfassung

In diesem Kapitel haben wir zunächst versucht, Kriterien zu finden, mit deren Hilfe sich die verschiedenen Debuggingsysteme klassifizieren lassen. Die vier wichtigsten sind:

- aus Anwendersicht wichtige Kriterien
- die benötigten Kenntnisse
- die Aufgaben des Debuggingsystems
- die verwendeten Techniken

Bei der Bestimmung dieser Kriterien haben wir uns auf die Arbeit von Ducasse und Shahmehri gestützt. Zusammenfassend läßt sich sagen, daß die bedeutendste Arbeit Shapiro mit dem *APD*-System geleistet hat. Eine Menge anderer Systeme verwenden teilweise seine Vorgehensweise oder versuchen diese zu verbessern. Selbst Shapiro hat nicht nur einen Diagnostizierer, sondern auch einen Mechanismus für die Fehlerkorrektur entwickelt (*MIS*). Aber er hat sich nur mit definiten Programmen und mit den Standard-Berechnungsregeln beschäftigt. Das von Pereira entwickelte *RD* kann in beliebigen PROLOG-Programmen, die zum Beispiel nicht deklarative Eigenschaften wie *cut* enthalten, Fehler suchen. Ferrand gab eine umfangreiche mathematische Analyse der Fehlerdiagnoseprogramme für definite Programme. Das System *APROPOS* von Looi ist ein Lehrsystem, aber es versucht, die entdeckten Fehler zu korrigieren. In *Preset* von Takahashi wird versucht, nicht nur die Fehler durch Debugging zu finden, sondern eine Erklärung zu liefern, wie die Fehler entstanden sind. Das *OPIUM*-System analysiert die Durchführung des Programms. Das gleiche Ziel verfolgte Eisenstadt bei der Entwicklung seines Systems *PTP*. *DED* ist ein Debuggingsystem, das auch erweiterte Syntax und Kontrollmöglichkeiten behandeln kann. In *ADAss* haben Drabent et al. daran gearbeitet, die von *APD* gestellten Fragen zu minimieren.

In *IDTS* haben wir versucht, alle von uns für wichtig gehaltenen bisherigen Ergebnisse zu integrieren und zu erweitern:

- Wenn wir das aus Anwendersicht wichtige *Kriterium des Automatisierungsgrades* betrachten, können wir sagen, daß in dieser Hinsicht unser System besonders gut ist, weil es während des Debuggings gesammelte Informationen benutzt, statt sie jedesmal erneut vom Anwender zu erfragen.

- Unser System erfüllt das Kriterium der *Beurteilung der Benutzererfahrung*, weil *IDTS* für verschiedene Gruppen von Anwendern geeignet ist: Durch die Vorbereitung der Testspezifikation kann jeder seine Kenntnisse dem Debugger angeben. Außerdem bietet unser System einen umfangreichen Satz an Möglichkeiten, die von ihm gelieferten Meldungen graphich zu interpretieren.

- Einige Systeme brauchen in der Fehlersuche eine zusätzliche Spezifikation. Unser System verwendet zum Beispiel die *CPM*-Beschreibung. Während bei anderen Systemen der Benutzer bei der Angabe der Spezifikation Fehler einbauen kann, ohne es später zu erkennen, findet *IDTS* die Inkonsistenzen in der Testbeschreibung und liefert sogar den fehlerhaften Testrahmen.

- Beim Debugging muß der Anwender Fragen beantworten. Falsche Antworten können den Debuggingprozeß irreführen. In *IDTS* haben wir enorme Verbesserungen im Vergleich zu anderen Systeme erreicht, weil wir in einigen Fällen erkennen können, daß der Anwender Fehler begangen hat, und wir die Möglichkeit bieten, abzubrechen.

- *IDTS* kann in Programmen mit erweiterter Syntax und zusätzlichen Kontrollmöglichkeiten nach Fehlern suchen.

- Das System benutzt die Debuggingvorgehensweise von Shapiro (Bottom-Up, Top-Down, Divide-and-Query) und kann alle drei Fehlertypen erkennen: *IDTS* findet nicht nur die falsche und fehlende Ausgabe in einem Programm, sondern behandelt auch den Fehler der unendlichen Schleife.

- Mit Hilfe des Slicing und der Testdatenbank wird die Anzahl der Fragen reduziert.

- Die zusätzlich entwickelte graphische Oberfläche, die noch kein bisheriges PROLOG-Debuggingsystem besitzt, bietet eine enorme Hilfe bei der Durchführung des Debuggingverfahrens.

# Kapitel 8

# Beispiel der Klassifizierung von Elektrokardiogrammen

In diesem Kapitel wird das System *SPECG* (Specialized Prolog-based Electrocardiogram) vorgestellt, das *EKG*-Wellenformen klassifizieren kann. Im nächsten Kapitel wird dieses System um ein Lernverfahren erweitert, und damit kann das System sowohl Syntax- als auch Semantikregeln von *EKG*-Grammatiken lernen. Die hier verwendeten Regeln braucht der Anwender, um die sogenannten *QRS*-Komplexe von den anderen Mustern trennen zu können. Dieses System hilft dem Benutzer, das Klassifikatorprogramm zu verbessern, wenn es die vorgegebene Eingabe nicht klassifizieren kann. Das System zeigt, wie die *Theory-Revision*-Methode für das Lernen großer PROLOG-Programme benutzt werden kann.

Das *SPECG*-System kombiniert einen Elektrokardiogramm-Wellenklassifikator (*EKG*-Wellenklassifikator) mit dem Debuggingsystem *IDTS* und später im Kapitel 9 mit dem Lernsystem *IMPUT* (Interactive Multiple Predicate Learner Using Unfolding Transformation). Bei der Klassifizierung werden wir anhand eines Beispiels zeigen, wie der Fehler in der Grammatik, die die Klassifizierung beschreibt, gefunden werden kann. Beim Lernprozeß ist die Eingabe eine unterbestimmte Grammatik, die später verbessert wird. Aber diese Grammatik enthält zusätzlich vom Anwender definierte Schwellenwerte und Entscheidungsregeln, mit deren Hilfe die Klassifizierung von *QRS*-Komplexen und die Entscheidung zwischen *QRS* und anderen Musterformen durchgeführt werden kann. Unser Ziel ist, ein System zu entwickeln, das durch Verwendung positiver und negativer Beispiele und durch ein Lernsystem die Aufgabe des Lernens von *EKG*-Wellenformen erfüllen kann. In diesem Kapitel geben wir zuerst eine sehr kurze Einführung in die Anatomie des Herzens und die *EKG*-Diagnostik. Dann erklären wir, wie die *EKG*-Wellenformen beschrieben werden können. Es folgt die Beschreibung des Systems, und am Ende zeigen wir durch ein Beispiel, wie das von uns entwickelte System die Wellenformen klassifizieren kann.

## 8.1 Die Wellenformen im Elektrokardiogramm

### 8.1.1 Die Anatomie des Herzens

Das Herz (siehe Abbildung 8.1) kann in vier Kammern aufgeteilt werden: den rechten und linken *Vorhof* (*Atrien*) und die linke und rechte *Herzkammer* (*Ventrikel*) (siehe Abbildung 8.2).

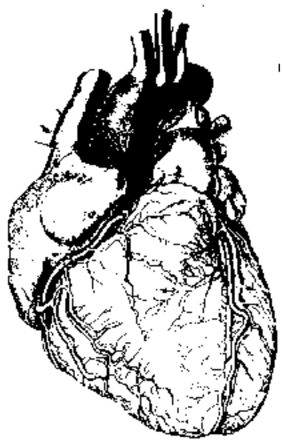


Abbildung 8.1: Das Herz

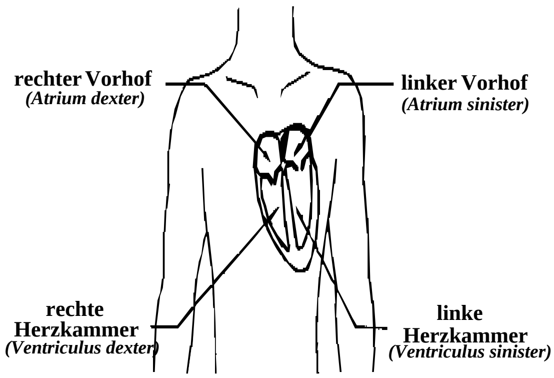



Abbildung 8.2: Die Anatomie des Herzens

Das *Kreislaufsystem* wird von den *Schlagadern* (*Arterien*) gebildet. Sie leiten das Blut vom Herzen in die Peripherie. Die große *Körperschlagader* (*Aorta*) kommt aus der linken Herzkammer und bedient den *großen* oder *Körperkreislauf.* Die *Lungenschlagader* (*A. pulmonalis*) kommt aus der rechten Herzkammer und führt das verbrauchte Blut dem *kleinen* oder *Lungenkreislauf* zu.

Ein normaler Herzschlag beginnt mit dem Zusammenziehen der beiden Vorkammern. Es folgt die Kontraktion der beiden Herzkammern. Dieser geordnete Prozeß des Zusammenziehens wird von einem elektrischen Nervenimpuls ausgelöst und gesteuert. Er kann im Elektrokardiogramm erkannt werden. Das Herz bildet die für seine Muskelkontraktion notwendigen Erregungen selbst (*Eigengesetzlichkeit* (*Autonomie*) des Herzrhythmus). Das elektrische Reizleitungssystem des Herzens kann man in Abbildung 8.3 sehen.

Der elektrische Impuls startet normalerweise aus dem Sinusknoten (*SA-Knoten*, Keith-Flack-Knoten). Er ist der primäre Schrittmacher des Herzens und befindet sich im oberen Teil der rechten Vorkammer. Der Impuls kann drei verschiedene interne Wege nehmen, um den *Atrioventrikularknoten* (*AV-Knoten*, Aschoff-Tawara-Knoten) zu erreichen. Diese Wege

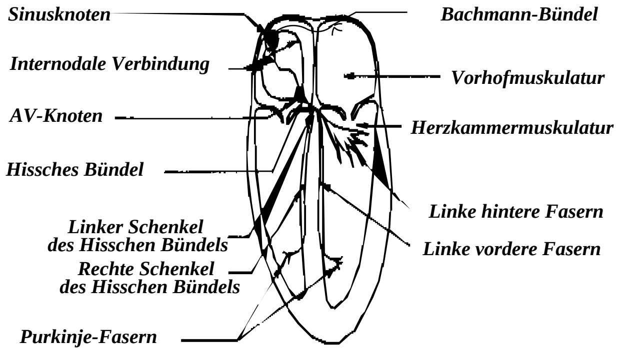



Abbildung 8.3: Reizleitungssystem des Herzens

führen rund um die rechte Vorkammer (*hintere, mittlere und vordere internodale Verbindung*) und um die linke Vorkammer (*Bachmann-Bündel*). Die Stimulierung der langsamleitenden Muskelzellen in beiden Vorkammern erzeugt die sogenannte $P$-Welle im $EKG$ (siehe Abbildung 8.5). Diese Wellenform repräsentiert die elektrische Erregung der Vorkammermuskelzellen. Der elektrischen Stimulation folgt das mechanische Zusammenziehen der Vorkammern.

Die Geschwindigkeit der Weiterleitung der elektrischen Impulse in der rechten Vorkammer ist dort langsamer, wo die Impulse in den *AV-Knoten* enden. Der AV-Knoten fungiert als eine Wartestation, ein Gebiet, wo die Impulse aus den beiden Vorkammern langsamer werden. Diese Verzögerung ist notwendig, um die Vorkammern zusammenzuziehen und ihren Inhalt in die jeweiligen Herzkammern zu pumpen.

Nach der kurzen Verzögerung im AV-Knoten geht der Impuls durch eine spezialisierte Leitungsbahn (*Hissches Bündel*) hinunter. Das Hissche Bündel spaltet sich in zwei Leitungsbahnen auf: den *rechten Schenkel des Hisschen Bündels (RBB)* und den *linken Schenkel des Hisschen Bündels (LBB)*, die die linke Herzkammer durchqueren.

Die kurze Haupt-LBB zerlegt sich später in die *hinteren und vorderen Fasern*, die die vordere-obere beziehungsweise hintere-untere Region der linken Herzkammer versorgen. Sowohl RBB als auch LBB verteilen sich in kleinen Zweigen und schließlich in terminalen Leitungsbahnen in den Herzkammern, die sogenannten *Purkinje-Fasern*. Aus den Purkinje-Fasern werden die Muskelzellen in beiden Herzkammern stimuliert, die den sogenannten $QRS$-Komplex der $EKG$-Wellenform erzeugen (siehe Abbildung 8.5). Der $QRS$-Komplex repräsentiert tatsächlich die elektrische Erregung der Muskelzellen in der Herzkammer. Den ganzen Ablauf der elektrischen Aktivierung des Herzens zeigt Abbildung 8.4.

Eine ausführlichere Beschreibung in deutscher Sprache kann in [Fal95] oder [GvHWW94] nachgelesen werden. Außerdem ist die Arbeit von Lipman und Lipman [LL93] zu empfehlen, in der die Grundbegriffe und die Verwendung von $EKG$ beschrieben werden.

### 8.1.2 Das Prinzip der EKG-Analyse

Der Depolarisations- und Repolarisationsprozeß äußert sich in Form von Ausschlägen in der $EKG$-Wellenform für beide Vorkammern und die Herzkammern. Die $P$-Welle repräsentiert die Depolarisation der Vorkammern, während der $QRS$-Komplex die Depolarisation der Herzkammern zeigt. Der $QRS$-Komplex ist viel größer als die $P$-Welle, weil die Herzkammern größer sind und mehr Muskelmasse als die Vorkammern haben.

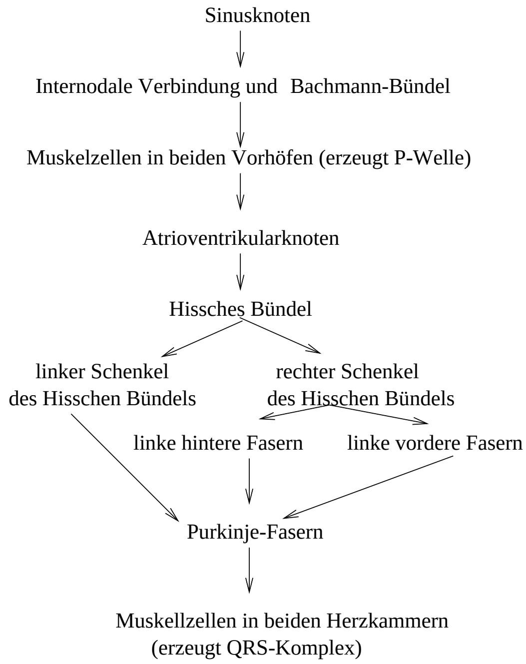



Abbildung 8.4: Die normale Sequenz der elektrischen Aktivierung

Die Depolarisation der Herzkammern wird durch das $ST$-Segment und die $T$-Wellen repräsentiert. Die Repolarisation der Vorkammern tritt ungefähr zur gleichen Zeit wie die Depolarisation der Herzkammern auf. Die Ausschläge, die von der Repolarisation der Vorkammern erzeugt werden, sind so klein, daß sie meistens vom $QRS$-Komplex überlagert werden.

Die normale elektrische Signalerregung des Herzens, die vom $EKG$ aufgezeichnet wird, enthält die $P$-Welle (Atrien-Schlag), gefolgt vom $QRS$-Komplex, dem $ST$-Segment und der $T$-Welle (ventrikulärer Schlag) (siehe Abbildung 8.5).

Die Fachterminologie der Ausschläge und Wellen im normalen $EKG$ schließt die folgenden Wellen und Komplexe ein:

- $P$-Welle: Depolarisation der linken und rechten Vorkammer.
- $QRS$-Komplex: Depolarisation der linken und rechten Herzkammer.
- $ST$-Segment: Der Beginn der Repolarisation der Herzkammern wird vom Ende des $QRS$-Komplexes bis zum Anfang der $T$-Welle gemessen. Der Anfang des $ST$-Segments wird als $J$-Punkt bezeichnet.
- $T$-Welle: Repolarisation der linken und rechten Herzkammer.
- $U$-Welle: nicht immer erkennbar, aber wenn sie vorhanden ist, folgt sofort die $T$-Welle und repräsentiert wahrscheinlich eine späte Repolarisation.

Wir erwähnen hier noch zwei wichtige Punkte:

(1) Die Herzkammern enthalten die meisten Muskeln und die größte Masse des Herzens. Das ist der Grund, warum der $QRS$-Komplex viel größer ist als die $P$-Welle.

(2) Die Ausschläge, die die Repolarisation der Vorkammern repräsentieren, sind klein und können im viel größeren $QRS$-Komplex verschwinden.

Die ventrikuläre Depolarisation erzeugt den $QRS$-Komplex, der verschiedene Konfigurationen haben kann. Die Terminologie der Ausschläge im $QRS$-Komplex ist die folgende:

- $Q$-Welle: anfängliche Abwärtsabweichung
- $R$-Welle: erste Vorwärtsabweichung
- $S$-Welle: anfängliche Abwärtsabweichung, die $R$-Welle folgt
- $R'$-Welle: zweite Vorwärtsabweichung, die $S$-Welle folgt
- $S'$-Welle: zweite Abwärtsabweichung, die $R$-Welle folgt
- $QS$-Welle: vollständiger $QRS$-Komplex, enthält negative Ausschläge

Die grundlegende $EKG$-Auswertung schließt die folgenden Parameter ein:

(1) $PR$-Intervall: Das Intervall wird vom Anfang der $P$-Welle bis zum Beginn des $QRS$-Komplexes gemessen. Dieses Intervall repräsentiert die Zeit, die die Impulse brauchen, um vom SA-Knoten durch die spezialisierte Leitungsbahn in den Vorkammern die Muskeln in beiden Herzkammern zu erreichen. Dieser Weg führt durch den AV-Knoten, das Hissche Bündel und die Schenkel des Hisschen Bündels in den Herzkammern und zu den Purkinje-Zellen. Das $PR$-Intervall enthält die Depolarisation der Muskelzellen in den Vorkammern ($P$-Welle) und den Weg der Impulse zum Punkt der Stimulation der Muskeln in den Herzkammern.

(2) $QRS$-Dauer: Ein Intervall vom Beginn bis zum Ende des $QRS$-Komplexes. Dieses Intervall repräsentiert die Depolarisation der beiden Herzkammern.

(3) $QT$-Intervall: Das Intervall wird vom Beginn des $QRS$-Komplexes bis zum Ende der $T$-Welle gemessen. Dies variiert invers mit der Pulszahl.

(4) $P$-Wellen-Morphologie: Diese kann normal, anormal, wechselnd oder fehlend sein.

(5) $P$-Wellen-Rate: Die Anzahl der $P$-Wellen pro Minute.

(6) $QRS$-Rhythmus: Regelmäßigkeit des $QRS$.

(7) $P$-$QRS$-Relation: Relation zwischen $P$-Welle und $QRS$-Komplex; es kann sein, daß $P$ immer oder manchmal von $QRS$ begleitet wird, oder daß $P$ und $QRS$ unabhängig sind.

Die normalen Zeitintervalle einer erwachsenen Person sind die folgenden:

- $PR$-Intervall: 0.12-0.20 Sekunden
- $QRS$-Dauer: 0.06-0.1 Sekunden
- $QT$-Intervall: variiert invers mit der Pulszahl

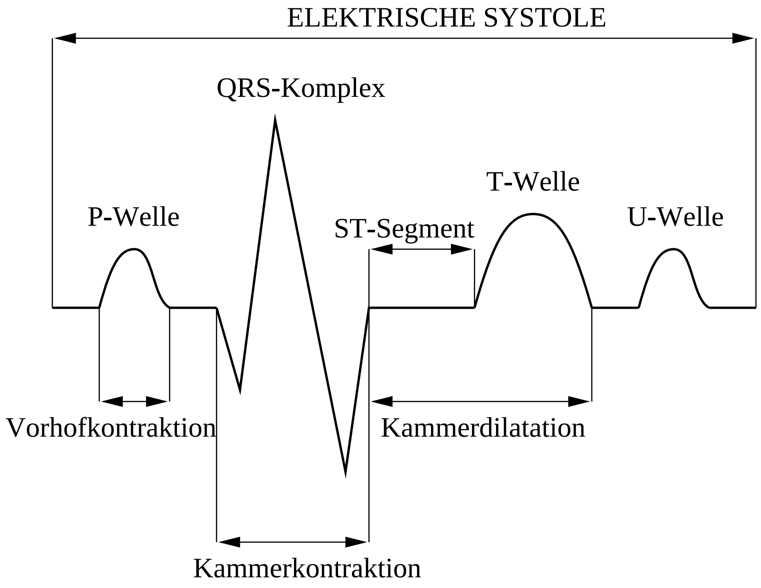



Abbildung 8.5: Normale *EKG*-Wellenform

## 8.2 Das syntaktische Erkennen der EKG-Wellenformen

In diesem Teil der Arbeit wird ein kurzer Überblick über das syntaktische Erkennen der *EKG*-Wellenformen gegeben. Die Methodologie und die Notation wurde von Skordalakis ausgearbeitet (siehe [Sko90]). Viele verschiedene Methoden können verwendet werden, um das *EKG*-Signal zu erkennen und zu analysieren, aber in allen diesen Prozessen gibt es zwei verschiedene Phasen, nämlich:

(1) die *Mustererkennungs- und Mustermessungsphase*, in der bestimmte Muster des *EKG* erkannt und verschiedene Parameter (Amplitude und Dauer) gemessen werden, um die Wellen charakterisieren zu können, und

(2) die *Interpretationsphase*, in der mit Hilfe des im ersten Schritt erkannten Musters und mit Attributen, die zu verschiedenen primitiven Mustern gehören, eine Diagnose gestellt wird. Die in Phase (1) noch nicht beseitigten Signalstörungen (Geräuschspitzen) dürfen dabei nicht berücksichtigt werden. In dieser Phase können heuristische diagnostische Kriterien, die bei der Analyse von Millionen von *EKG*s festgestellt worden sind, benutzt werden.

Bei der syntaktischen Erkennung des *EKG*-Signals gibt es vier verschiedene Teilprobleme:

(1) Die Segmentierung von primitiven Mustern

(2) Die Kodierung der primitiven Muster

(3) Das Parsen der kodierten Muster

(4) Die Klassifizierung der *EKG*-Wellenformen

Für die ersten drei Probleme haben wir ein Programm entwickelt, das die *EKG*-Wellenformen anzeigt. Der Anwender kann die primitiven Muster selektieren, und danach wird die Eingabe der Grammatik generiert. Genau wird dies im Kapitel 9.3.1 beschrieben. Unsere Primitive sind die *Spitze* (*peak*), die *Linie* (*line*) und die *Parabel* (*parabola*).

Eine typische Spitze zeigt die Abbildung 8.6. Diese ist der wichtigste Teil und kann durch die *linke Grenze*, den *Extremwert*, der der höchste Punkt der Spitze ist, wenn diese positiv ist, und der niedrigste, wenn die Spitze negativ ist, und die *rechte Grenze* charakterisiert werden.

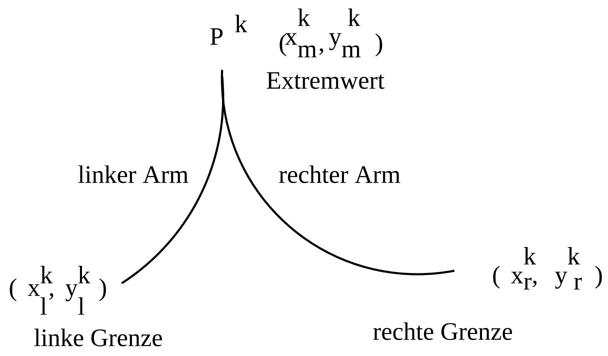



Abbildung 8.6: Eine Spitze

Um eine Spitze $P^k$ beschreiben zu können, braucht man elf Attribute: Die Attribute $\{x_l^k, y_l^k, x_m^k, y_m^k, x_r^k, y_r^k, x_{la}^k, v_{la}^k, x_{ra}^k, v_{ra}^k, A^k, line^k\}$, die benutzt werden, um das primitive Muster Spitze $P^k$ zu definieren, sind im einzelnen:

- $(x_l^k, y_l^k)$: die Koordinaten der linken Grenze der Spitze $P_k$,
- $(x_m^k, y_m^k)$: die Koordinaten des Extremwerts der Spitze $P_k$,
- $(x_r^k, y_r^k)$: die Koordinaten der rechten Grenze der Spitze,
- $x_{la}^k$: die $x$-Koordinate des Punktes am linken Arm der Spitze $P_k$, bei der die maximale (absolute) Steigung vorkommt,
- $v_{la}^k$: der Wert der Steigung bei $x_{la}^k$,
- $x_{ra}^k$: die $x$-Koordinate des Punktes am rechten Arm der Spitze $P_k$, bei der die maximale (absolute) Steigung vorkommt,
- $v_{ra}^k$: der Wert der Steigung bei $x_{ra}^k$,
- $A^k$: der Winkel am Extremwert der Spitze,
- $line^k$: die Energie der Spitze $k$ mit dem Wert:
  $\sum_{i=l+1}^{r}(y_i - y_{i-1})^2$.

Die sechs Attribute $\{x_b^k, y_b^k, x_e^k, y_e^k, B^k, C^k\}$, die benutzt wird, um das primitive Muster Linie (straight line) $L^k$ zu definieren, sind im einzelnen:

- $(x_b^k, y_b^k)$: der Anfangspunkt des Liniensegments,
- $(x_e^k, y_e^k)$: der Endpunkt des Liniensegments,
- $B^k$ and $C^k$: die Koeffizienten in der Gleichung $y = B^k x + C^k$.

Die sieben Attribute $\{x_b^k, y_b^k, x_e^k, y_e^k, B^k, C^k, D^k\}$, um das primitive Muster Parabel (parabola) $K^k$ zu definieren, sind im einzelnen:

- $(x_b^k, y_b^k)$: der Anfangspunkt des Parabelsegments,
- $(x_e^k, y_e^k)$: der Endpunkt des Parabelsegments,
- $B^k, C^k, D^k$: die Koeffizienten in der Gleichung $y = B^k x^2 + C^k x + D^k$.

Die Hauptaufgaben der Grammatik sind die folgenden:

- *$QRS$-Entdeckung und -Erkennung*: Es gibt dreizehn verschiedene Regeln für die Erkennung der hintereinanderstehenden Spitzen als $QRS$-Komplex. Ein $QRS$-Komplex kann aus ein bis sieben Spitzen aufgebaut sein. Die folgenden Bedingungen müssen erfüllt sein:
  (1) Der Winkel zwischen dem rechten Arm der Spitze $i$ und dem linken Arm der Spitze $i+1$ ist kleiner als ein vorgegebenes $\varepsilon_2$.
  (2) Der Winkel der Spitze ist kleiner als ein vorgegebenes $\varepsilon_5$.
  (3) Die Energie wenigstens einer Spitze ist größer als ein vorgegebenes $\varepsilon_1$.
- *$P$-, $T$-Entdeckung und -Erkennung*: Der $P$- oder $T$-Komplex kann nur zwischen zwei $QRS$-Komplexen gefunden werden. Ein $P$- oder $T$-Komplex enthält entweder eine oder zwei hintereinander stehende Spitzen. Diese Spitzen können von den anderen mit Hilfe von semantischen Regeln unterschieden werden, wobei die Energie der Spitzen ausgewertet wird. Eine Spitze wird als $P$- oder $T$-Komplex erkannt, wenn die Größe und die Breite aller Arme für alle Spitzen größer sind als vorgegebene $\varepsilon_3$ und $\varepsilon_4$.
- *$QRS$-Klassifikation*: Die Aufgabe ist die Unterscheidung zwischen einem $QRS$-Komplex und einer Klasse von $QRS$-Komplexen oder die Entfernung zwischen zwei Klassen, die $QRS$-Komplexe enthalten, zu berechnen. Im ersten Fall wird die Unterscheidung für den $QRS$-Komplex und für alle $QRS$-Komplexe in der Klasse berechnet. Im zweiten Fall kann die *city block*-Metrik behilflich sein, das Intervall zwischen zwei $QRS$-Komplexen zu berechnen. Die normalisierte Dauer und Größe für alle Komponenten der Welle des $QRS$-Komplexes werden verwendet, um die Entfernung festzustellen.

## 8.3 Ein logischer Ansatz für die Repräsentation von EKGs

Im Artikel [Sko90] wurde eine attributierte Grammatik für die Erkennung von $EKG$-Wellenformen benutzt. Die $EKG$-Wellenform wurde mit Symbolen aus dem Alphabet $\sum = \{n\_peak, p\_peak, line, par\}$ dargestellt, wobei *p_peak* und *n_peak* die positive und negative Spitze, *line* das gerade Liniensegment und *par* das Parabelsegment sind.

Bei der Umformulierung der attributierten Grammatik in PROLOG mußten wir mehrere Probleme lösen. Das erste war die Implementierung der semantischen Funktionen. Es war problematisch, weil sie für interaktive Sprachen geschrieben wurden und nicht für logische Programmiersprachen. Wir mußten eine große Menge von Parametern behandeln. Obwohl unser Verfahren auf der Methode von Skordalakis basiert, haben wir einen logischen Ansatz für die Erkennung von Kategorien benutzt. Die attributierte Grammatik wurde als Anfangspunkt verwendet, und wir haben sie in PROLOG umgesetzt und modifiziert, weil wir festgestellt haben, daß die von Skordalakis verwendete fehlerhaft ist. Der Unterschied zwischen der attributierten Grammatik und dem PROLOG-Programm hat diesen Umsetzungsprozeß zusätzlich erschwert. Wir haben die definite Klausel-Grammatik-Repräsentation (DCG, siehe [PBB+78]) benutzt, weil diese Notation ein gutes Werkzeug für die Umsetzung attributierter Grammatiken bietet (siehe [DM85]). Der Umsetzungsprozeß enthält die folgenden Schritte:

(1) Die nichtterminalen Symbole der Grammatik entsprechen den Prädikaten des PROLOG-Programms.

(2) Sowohl die vererbten als auch die synthetisierten Attribute werden auf die Argumente der PROLOG-Prädikate abgebildet. Die zwanzig am häufigsten verwendeten Argumente und ihre Bedeutung sind die folgenden:

| Attribut | Bedeutung | Typ |
|---|---|---|
| $iw$ | Anzahl der Wellen | vererbt |
| $sw$ | | synth. |
| $ic$ | Anzahl der Klassen | vererbt |
| $sc$ | | synth. |
| $iqrs[i]$ | Anzahl von $QRS$'s in der Klasse $i$ | vererbt |
| $sqrs[i]$ | | synth. |
| $type$ | $QRS$ Morphologie | synth. |
| $beg$ | Anfangspunkt eines Musters | synth. |
| $end$ | Endpunkt eines Musters | synth. |
| $xm$ | $x$-Koordinate des Extremwerts einer Spitze, | synth. |
| $ybeg$ | $y$-Koordinate des Extremwerts, wo die $x$-Koordinate $beg$ ist | synth. |
| $yend$ | $y$-Koordinate des Punktes, wo die $x$-Koordinate $end$ ist | synth. |
| $yxm$ | $y$-Koordinate des Extremwerts einer Spitze | synth. |
| $te$ | Energie einer Spitze | synth. |
| $ang$ | Winkel einer Spitze | synth. |
| $ldur$ | Dauer des linken Armes einer Spitze | synth. |
| $rdur$ | Dauer des rechten Armes einer Spitze | synth. |
| $lh$ | Extremwert des linken Armes einer Spitze | synth. |
| $rh$ | Extremwert des rechten Armes einer Spitze | synth. |
| $tp - flag$ | Kandidat einer Spitze als ein $P$ oder $T$ (Unter-)Muster | vererbt und synth. |

(3) Die semantische Routine und viele Hilfsfunktionen werden in zusätzliche Bedingungen in Form von PROLOG-Prozeduraufrufen konvertiert. Diese müssen auf der rechten Seite der Grammatikregeln stehen.

Als Beispiel sind hier zwei PROLOG-Klauseln angegeben, die zwei verschiedene Arten von $QRS$-Mustern beschreiben:

```
qrs(Iw1,Sw1,Ic1,Sc1,Iqrs1,Sqrs1,Beg1,End1,Type1)  -->
   {insert_last(0,[],B0),insert_last(0,[],E0),
   insert_last(0,[],Yx0),
   retract((build_draw_qrs(_))), assert((build_draw_qrs([])))},
   r(Beg2,End2,Xm2,Ybeg2,Yend2,Yxm2,Te2,Ang2),
   {insert_last(Te2,[],X1),insert_last(Ang2,[],Y1),
   insert_last(Beg2,B0,B1),insert_last(End2,E0,E1),
   insert_last(Yxm2,Yx0,Yx1),
   build_qrs(Beg2,Ybeg2,Xm2,Yxm2,End2,Yend2)},
   s(Beg3,End3,Xm3,Ybeg3,Yend3,Yxm3,Te3,Ang3),
   {insert_last(Te3,X1,Te_list),insert_last(Ang3,Y1,Ang_list),
   insert_last(Beg3,B1,Beg_list),insert_last(End3,E1,End_list),
   insert_last(Yxm3,Yx1,Yxm_list),
   angle(Xm2,Yxm2,End2,Yend2,Beg3,Ybeg3,Xm3,Yxm3,Angle2),
   insert_last(Angle2,[],Angle_list),
   build_qrs(Beg3,Ybeg3,Xm3,Yxm3,End3,Yend3),
   Type1 = 11,
   sem_rou1(Ybeg2,Iqrs1,Ic1,Iw1,Sw1,Beg1,Beg2,Sc1,Sqrs1,End1,End3,2,1,
            Te_list,Ang_list,Angle_list,Beg_list,End_list,Yxm_list)}.

qrs(Iw1,Sw1,Ic1,Sc1,Iqrs1,Sqrs1,Beg1,End1,Type1)  -->
   {insert_last(0,[],B0),insert_last(0,[],E0),
   insert_last(0,[],Yx0),
   retract((build_draw_qrs(_))), assert((build_draw_qrs([])))},
   q(Beg2,End2,Xm2,Ybeg2,Yend2,Yxm2,Te2,Ang2),
   {insert_last(Te2,[],X1),insert_last(Ang2,[],Y1),
   insert_last(Beg2,B0,B1),insert_last(End2,E0,E1),
   insert_last(Yxm2,Yx0,Yx1),
   build_qrs(Beg2,Ybeg2,Xm2,Yxm2,End2,Yend2)},
   r(Beg3,End3,Xm3,Ybeg3,Yend3,Yxm3,Te3,Ang3),
   {insert_last(Te3,X1,X2),insert_last(Ang3,Y1,Y2),
   insert_last(Beg3,B1,B2),insert_last(End3,E1,E2),
   insert_last(Yxm3,Yx1,Yx2),
   angle(Xm2,Yxm2,End2,Yend2,Beg3,Ybeg3,Xm3,Yxm3,Angle2),
   insert_last(Angle2,[],Z1),
   build_qrs(Beg3,Ybeg3,Xm3,Yxm3,End3,Yend3)},
   s(Beg4,End4,Xm4,Ybeg4,Yend4,Yxm4,Te4,Ang4),
   {insert_last(Te4,X2,Te_list),insert_last(Ang4,Y2,Ang_list),
   insert_last(Beg4,B2,Beg_list),insert_last(End4,E2,End_list),
   insert_last(Yxm4,Yx2,Yxm_list),
   angle(Xm3,Yxm3,End3,Yend3,Beg4,Ybeg4,Xm4,Yxm4,Angle3),
   insert_last(Angle3,Z1,Angle_list),
   Type1 = 9,
   build_qrs(Beg4,Ybeg4,Xm4,Yxm4,End4,Yend4),
   sem_rou1(Ybeg2,Iqrs1,Ic1,Iw1,Sw1,Beg1,Beg2,Sc1,Sqrs1,End1,End4,3,0,
            Te_list,Ang_list,Angle_list,Beg_list,End_list,Yxm_list)}.
```

Diese beiden Klauseln beschreiben zwei verschiedene Arten des $QRS-Komplexes$ in DCG-Form. Die verschiedenen numerischen Werte werden mit dem Prädikat `insert_last` getrennt in Listen eingetragen. Dann wird der Anruf der semantischen Prozedur `sem_rou1` entscheiden, ob der gerade analysierte $QRS$-Komplex ein $RS-Komplex$ oder $QRS$-Komplex ist (insgesamt gibt es dreizehn Möglichkeiten für die syntaktische Beschreibung des $QRS$-Komplexes). Die nächsten zwei Abbildungen 8.7 und 8.8 zeigen graphisch die verschiedenen $QRS$-Komplexe, wenn sie als $RS$ oder als $QRS$ klassifiziert wurden.

**Beispiel 67** Die graphische Darstellung des $QRS$-Komplexes $RS$

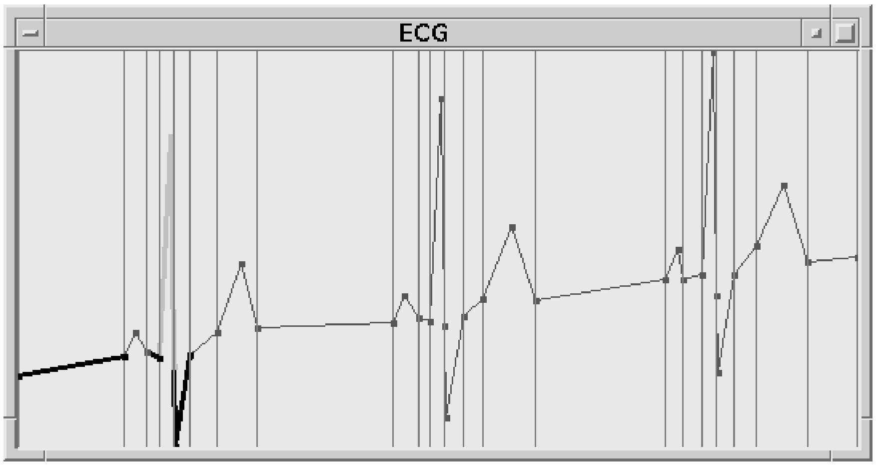



Abbildung 8.7: Die graphische Repräsentation des erkannten $QRS$-Teils als $RS$

**Beispiel 68** Die graphische Darstellung des $QRS$-Komplexes $QRS$

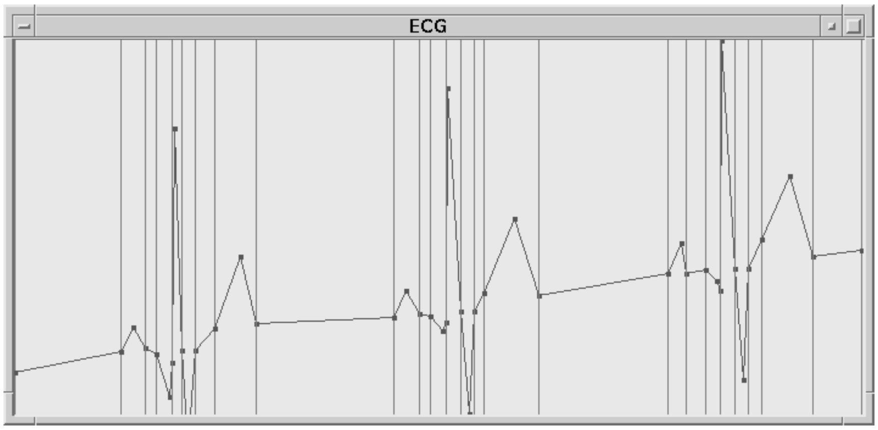



Abbildung 8.8: Die graphische Repräsentation des erkannten $QRS$-Teils als $QRS$

## 8.4 Die Struktur des SPECG-Systems

In diesem Teil wird ein Überblick über das *SPECG*-System gegeben. Das *SPECG*-System ist im wesentlichen eine Integration des *EKG*-Erkennungsmoduls in das *IDTS*-Modul. Wie schon früher erwähnt wurde, kann mit Hilfe dieses Werkzeugs ein Fehler in einer Klassifikation gefunden werden. Das System wird später weiterentwickelt. Die Erweiterung beschreiben

wir im Kapitel 9.3. Ein interaktives Vorverarbeitungsmodul für *EKG*, eine interaktive Lernmethode und eine graphische Benutzeroberfläche (*SPECG-GUI*) werden integriert. Das so erweiterte System kann nicht nur den genauen Platz des Fehlers erkennen, sondern es kann in vielen Fällen mit Hilfe von Unfolding-Transformation und *IDTS* das korrekte Programm gelernt werden, wenn wir mit einem fehlerhaften (*buggy*) Programm anfangen.
Die Struktur des *SPECG*-Systems zeigt uns die Abbildung 8.9.

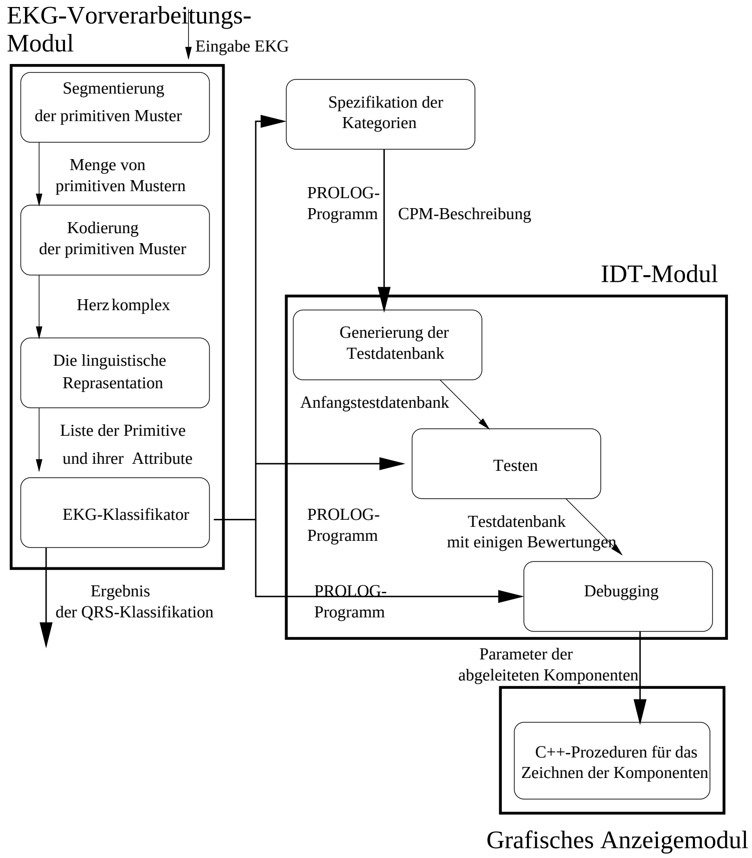



Abbildung 8.9: Die Struktur des *SPECG*-Systems

Die Hauptmodule sind die folgenden:

- *EKG-Vorverarbeitungsmodul*: Dieser Teil des Systems benutzt das syntaktische Verfahren, um ein Elektrokardiogramm-Muster zu erkennen.
- *IDTS-Modul*: Dieser Teil ist der am Anfang der Arbeit erklärte interaktive Tester und Debugger, der die Methoden von Shapiro mit der *CPM*-Methode (*Category Partition Testing Method*) kombiniert.
- *Grafisches Anzeigemodul*: Dieser Teil macht den aktuellen Zustand des Debuggingprozesses sichtbar.

Das *EKG*-Vorverarbeitungsmodul wird aus den folgenden Teilmodulen aufgebaut:

- *Segmentierung der primitiven Muster*

- *Kodierung der primitiven Muster*
- *Linguistische Repräsentation*
- *EKG-Klassifikator*

Die Aufgabe der oben aufgelisteten Teilmodule ist die folgende:

- *Segmentierung der primitiven Muster*: Eine *EKG*-Wellenform kann als ein zusammengesetztes Muster betrachtet werden. Das bedeutet, daß sie so lange zerlegt werden kann, bis wir primitive Muster (Linie, Spitze, Parabel) bekommen.
- *Kodierung der primitiven Muster*: Die wichtigste Aufgabe dieses Teils ist die Auswahl der Spitzen. Dieses Untermodul beruht auf der Erkennung von Geräuschspitzen. Nur die Spitzen, die keine Geräuschspitzen sind, werden weiterverarbeitet.
- *Die linguistische Repräsentation*: Das Ergebnis der letzten zwei Schritte kann als eine Kette von Symbolen aus einem Alphabet repräsentiert werden.
- *Der EKG-Klassifikator*: Dieses Untermodul erkennt die *EKG*-Wellenform aus ihrer linguistischen Repräsentation.

Das *IDTS*-Modul kann in drei Teilmodule aufgeteilt werden. Hier erwähnen wir nur kurz die Bausteine, weil diese schon in Kapitel 4 detailliert erklärt wurden.

- *Generierung der Testdatenbank*: Aus der CPM-Beschreibung des Programms generiert dieses Teilmodul die Anfangsdatenbank.
- *Testen:* Ein Prädikat kann ausgewählt werden. Der Anwender kann ein repräsentatives Element zu einem Testrahmen, das zu diesem Prädikat generiert wurde und eine *undefined* Bewertung hat, eingeben. Dann entscheidet der Anwender, ob für die vorgegebene Eingabe die Ausgabe korrekt oder inkorrekt ist.
- *Debuggingprozeß*: Wenn das Programm schon getestet wurde, dann kann das Ergebnis des Testverfahrens direkt, ohne den Anwender zu fragen, verwendet werden.

Das graphische Anzeigemodul hilft dem Benutzer zu entscheiden, ob sich das Programm mit der gegebenen Eingabe korrekt oder inkorrekt verhält, indem es die Schritte der Debuggerphase illustriert. Am Anfang des Debuggingverfahrens wird die ganze *EKG*-Wellenform (siehe Abbildung 8.10) erscheinen. In allen folgenden Schritten wird dann die aktuelle Komponente der Eingabe markiert. Dieses Modul enthält Prozeduren, die in der Programmiersprache *C++* geschrieben wurden.

## 8.5 Analysieren von *EKG*-Wellenformen mit Hilfe von IDTS — Ein Beispiel

In diesem Kapitel erklären wir, wie das *IDTS* System als Debugger für eine Beschreibung der *EKG*-Wellenformen benutzt werden kann. Der Debugger-Prozeß wird mit einer graphischen Repräsentation unterstützt. Diese zeigt den Werdegang der Traversierung des Ableitungsbaums des *EKG*-Programms.

Um die Arbeit des Debuggers zu demonstrieren, nehmen wir an, daß ein Fehler in der logischen Repräsentation des $QRS$-Komplexes vorliegt. Aus diesem Grund wird eine korrekte Klausel aus dem Programm entfernt. *IDTS* wird benutzt, um diesen Fehler zu identifizieren, nämlich herauszufinden, daß eine $QRS$-Klasse in der Klassifizierung fehlt.

In der Entwicklungsphase wurde festgestellt, daß es für den Benutzer sehr kompliziert sein kann, von dem System gestellte Fragen nur auf Grund einer textuellen Repräsentation dieses Teils der $EKG$-Wellenform zu beantworten, weil die Beschreibung eine große Menge von Parametern enthält. Aus diesem Grund wurde eine graphische Anzeige entwickelt. In einem Fenster erscheint eine graphische Repräsentation der $EKG$-Wellenform, und der Anwender kann verfolgen, welcher Teil der $EKG$-Wellenform gerade analysiert wird.

Im Rest des Kapitels wird ein Beispiel gegeben, das zeigt, wie das Programm arbeitet. Eine $EKG$-Wellenform ist gegeben, und diese erkennt das System nicht korrekt. Deswegen muß der $EKG$-Klassifikator geändert werden.

Nehmen wir an, daß wir eine Anfangs-$EKG$-Wellenform mit folgender Repräsentation haben:

```
[line p_peak par p_peak n_peak line p_peak par p_peak line p_peak n_peak par
p_peak par p_peak par p_peak n_peak line p_peak par]
```

wobei die folgende Notation benutzt wird:

- *n_peak:* eine Spitze mit negativem Extremwert und elf Parametern.
- *p_peak:* eine Spitze mit positivem Extremwert und elf Parametern.
- *line:* eine gerades Linien-Segment mit sechs Parametern.
- *par:* die vierte Art von primitiven Mustern, die Parabel, die sieben Parameter hat.

Die Bedeutung der Parameter wurde in 8.3 näher beschrieben.

In diesem Beispiel wird mit Hilfe des Debuggers festgestellt, daß die einzige Möglichkeit, den Fehler zu beheben, eine Modifikation in der $EKG$-Syntax ist. Eine neue Ableitungsregel muß zusätzlich definiert werden. Die Anfangsgrammatik ist die folgende:

```
  ecg_lead       --> initial_part, cardiac_cycles, final_part.
  initial_part   --> segment, initial_part | "" | peak, initial_part
  final_part     --> qrs, rest_part | rest_part
  rest_part      --> segment, rest_part | peak, rest_part | ""
  cardiac_cycles --> cardiac_cycle, cardiac_cycles | qrs, non_qrs
  qrs            --> q , r , s , r_ver, s_ver, r_ver2, s_ver2 |
                     q , r , s , r_ver, s_ver, r_ver2 |
                     r , s , r_ver, s_ver, r_ver2, s_ver2|
                     r , s , r_ver, s_ver, r_ver2|
                     q , r , s , r_ver | q , r , s , r_ver |
                     r , s , r_ver , s_ver | r , s , r_ver | q, r , s |
                     q , r | r  | qs |
% Die Variation qrs --> r , s. fehlt
  non_qrs       --> sr | st, t, tr | sp, p, pr | st, t, p, pr | tp
  st            --> interwave_segment.
  tp            --> interwave_segment.
  pr            --> interwave_segment.
  tr            --> interwave_segment.
```

```
sp                --> interwave_segment.
sr                --> segment, interwave_segment.
sr                --> peak, interwave_segment.
interwave_segment --> segment, interwave_segment  | ""  |  peak, interwave_segment
t                 --> t_or_p.
sp                 --> t_or_p.
t_or_p            --> "p_peak", "n_peak" | "n_peak", "p_peak" | "p_peak" |  "n_peak"
r                 --> "p_peak".
r_ver             --> "p_peak".
r_ver2            --> "p_peak".
q                 --> "n_peak".
qs                --> "n_peak".
s                 --> "n_peak".
s_ver             --> "n_peak".
s_ver2            --> "n_peak".
peak              --> "p_peak" | "n_peak"
segment           --> "line" | "par"
```

Als erster Schritt wird das System eingelesen und gestartet. Der Benutzer kann den Namen der Dateien eingeben, die einerseits die Anfangsbeschreibung der *EKG*-Grammatik und andererseits die *CPM*-Beschreibung zu dieser Grammatik enthalten.

**Beispiel 69** `Enter file name to load program: ecg.pl`

```
Enter file name to load test description: ecg_testdesc.pl

--> Load program from file: ecg.pl
--> Load CPM description from file: ecg_testdesc.pl.
```

Das System generiert die Anfangsdatenbank aus der CPM-Beschreibung. Hier zeigen wir nur eine einfache CPM-Spezifikation für das Prädikat `qrs`. Natürlich kann genauso eine CPM-Testbeschreibung für andere Prädikate, wie `cardiac_cycles` oder `ecg_lead`, vorbereitet werden.

Die Beschreibung beginnt mit der Zeile `type(qrs())`, die definiert, ob die Parameter von `qrs` ausgeschrieben werden oder nicht (im Detail siehe Beispiel 21).

Die *CPM*-Beschreibung dieses Prädikats enthält drei Kategorien: `number_of_the_waves`, `number_of_the_classes` und `number_of_the_qrs`. Alle Kategorien können in zwei Wahlmöglichkeiten verteilt werden. Zum Beispiel sagt die Kategorie `number_of_the_qrs`, daß kein *QRS* (choice `zero_qrs`, property `p_zero`) oder eine beliebige Zahl von *QRS* erkannt wurde (choice `non_zero_qrs`). Die Suchfunktionen sind einfache PROLOG-Fakten. Für die Wahlmöglichkeit `non_zero_qrs` wurde keine Suchfunktion definiert. In diesem Fall ist es die Aufgabe des Benutzers, die richtige Wahlmöglichkeit zu identifizieren. Er kann dabei das vom System angebotene Menü (siehe Kapitel 4.3.1) benutzen. Die Propertynamen und die Selektionsausdrücke bestimmen den Testraum, so daß sie die redundanten Kombinationen der Eigenschaften wie `[non_zero_classes, zero_qrs, non_zero_waves]` ausschließen.

**Beispiel 70** Die CPM-Beschreibung des Prädikats `qrs`

```
%  test description:
type(qrs(+,+,+,+,+,+,+,+,+,-,-)).

% first category: number_of_the_waves
  choice_of(qrs,11,number_of_the_waves,zero_waves).
```

```
    choice_of(qrs,11,number_of_the_waves,non_zero_waves).

  % second category: number_of_the_classes
    choice_of(qrs,11,number_of_the_classes,zero_classes).
    choice_of(qrs,11,number_of_the_classes,non_zero_classes).

  % third category: number_of_the_qrs
    choice_of(qrs,11, number_of_the_qrs,zero_qrs).
    choice_of(qrs,11, number_of_the_qrs,non_zero_qrs).

  % searching functions:
    zero_waves(qrs,11,_,0,_,_,_,_,_,_,_,_,_).
    non_zero_waves(qrs,11,_,Sw1,_,_,_,_,_,_,_,_,_) :- Sw1 \== 0.
    zero_classes(qrs,11,_,_,_,0,_,_,_,_,_,_,_).
    non_zero_classes(qrs,11,_,_,_,Sc0,_,_,_,_,_,_,_) :- Sc0 \== 0.
    zero_qrs(qrs,11,_,_,_,_,_,[0],_,_,_,_,_).
    undefined_search(non_zero_qrs).

  % properties:
    p_zero(qrs,11,zero_qrs_qrs,X)  :- member(zero_qrs_qrs,X).

  % conditions:
    cond(qrs,11,zero_classes_qrs,X):- !, p_zero(qrs,11,_,X).
    cond(qrs,11,zero_waves_qrs,X)  :- !, p_zero(qrs,11,_,X).
```

Die generierten Testrahmen für alle zulässigen Kombinationen von Wahlmöglichkeiten der drei Kategorien sind die folgenden:

**Beispiel 71** Die generierten und selektierten Testrahmen für das Prädikat `qrs`

```
 The generated frames for qrs/11 are:
  [non_zero_classes,non_zero_qrs,non_zero_waves]
  [non_zero_classes,non_zero_qrs,zero_waves]
  [non_zero_classes,zero_qrs,non_zero_waves]
  [non_zero_classes,zero_qrs,zero_waves]
  [zero_classes,non_zero_qrs,non_zero_waves]
  [zero_classes,non_zero_qrs,zero_waves]
  [zero_classes,zero_qrs,non_zero_waves]
  [zero_classes,zero_qrs,zero_waves]

 The screened frames for qrs/11 are:
  [non_zero_classes,non_zero_qrs,non_zero_waves]
  [non_zero_classes,non_zero_qrs,zero_waves]
  [zero_classes,non_zero_qrs,non_zero_waves]
  [zero_classes,non_zero_qrs,zero_waves]
  [zero_classes,zero_qrs,zero_waves]
```

Die vom System generierte Testdatenbank, wenn wir annehmen, daß eine CPM-Beschreibung für die Prädikate `cardiac_cycles` und `ecg_lead` vorhanden ist, ist die folgende:

**Beispiel 72** Die gesamte generierte Anfangstestdatenbank

```
The list of current test database:
  test(cardiac_cycles, 8, [non_zero1,non_zero2,non_zero], [], undefined).
```

```
test(cardiac_cycles, 8, [non_zero1,non_zero2,zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,zero2,non_zero], [], undefined).
test(cardiac_cycles, 8, [non_zero1,zero2,zero], [], undefined).
test(cardiac_cycles, 8, [zero1,zero2,zero], [], undefined).
test(ecg_lead, 5, [non_zero1,non_zero2,non_zero], [], undefined).
test(ecg_lead, 5, [non_zero1,non_zero2,zero], [], undefined).
test(ecg_lead, 5, [non_zero1,zero2,non_zero], [], undefined).
test(ecg_lead, 5, [non_zero1,zero2,zero], [], undefined).
test(ecg_lead, 5, [zero1,zero2,zero], [], undefined).
test(qrs, 11, [non_zero_classes,non_zero_qrs,non_zero_waves], [], undefined).
test(qrs, 11, [non_zero_classes,non_zero_qrs,zero_waves], [], undefined).
test(qrs, 11, [zero_classes,non_zero_qrs,non_zero_waves], [], undefined).
test(qrs, 11, [zero_classes,non_zero_qrs,zero_waves], [], undefined).
test(qrs, 11, [zero_classes,zero_qrs,zero_waves], [], undefined).
```

Jetzt kann diese Datenbank getestet werden. Wir führen einen Testprozeß durch.

**Beispiel 73** Selektion eines Prädikats, das getestet wird

```
    1. Frame: "cardiac_cycles,8".
    2. Frame: "ecg_lead,5".
    3. Frame: "qrs,11".
    0. Quit.

    Enter selection: 1
```

Wie wir schon erklärt haben (siehe Kapitel 4.2), zählt der Debugger diejenigen Prädikate auf, die eine Testbeschreibung haben. Dann baut das Programm eine Liste auf und wählt diejenige Zeile aus der Datenbank aus, die zu dem gewählten Prädikat gehören und keine Bewertung haben (Bewertung ist *undefined*).

**Beispiel 74** Die selektierten Zeilen zum Prädikat `cardiac_cycles`

```
The frames for cardiac_cycles/9 are:

    1. Frame: "[non_zero1,non_zero2,non_zero]".
    2. Frame: "[non_zero1,non_zero2,zero]".
    3. Frame: "[non_zero1,zero2,non_zero]".
    4. Frame: "[non_zero1,zero2,zero]".
    5. Frame: "[zero1,zero2,zero]".
    0. Quit.

    Enter selection: 1
```

Auf der Basis der vom Anwender selektierten Wahlmöglichkeit ist im nächsten Schritt ein Beispiel einzugeben, das in diese Wahlmöglichkeit fällt.

**Beispiel 75** Ein Beispiel für die Wahlmöglichkeit `[non_zero1, non_zero2, non_zero]` des Prädikats `cardiac_cycles`

```
Enter clause: cardiac_cycles(0,0,[0],0,0,[0],
              p_peak,par,line,p_peak,n_peak,line,p_peak,par,p_peak,line,p_peak,n_peak,
              par,p_peak,par,p_peak,par,p_peak,n_peak,line,p_peak,par], [p_peak,par,
              line,p_peak,par,p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,
              p_peak,n_peak,line,p_peak,par])
```

Der komplizierteste Teil des Systems ist der algorithmische Debugger-Teil. Wir demonstrieren jetzt ein Debugging-Verfahren, bei dem wir die Datenbank benutzen. Der Anwender hat zwei Möglichkeiten: Entweder es wird auf den

**Beispiel 76** Starten des Debugging-Prozesses

```
--> Debugging.

Debugging points: segment, negative_peak, cardiac_cycles.

Debugging point [step, skip]: skip

Enter predicate: main(e).
```

Das System führt das vorher eingegebene Programm durch und schreibt das Ergebnis auf den Bildschirm. Der Anwender kann entscheiden, ob dieses Ergebnis korrekt ist oder nicht. In unserem Beispiel sagt er, daß das Ergebnis inkorrekt ist. Zur gleichen Zeit zeigt das graphische Werkzeug die ganze *EKG*-Wellenform an.

**Beispiel 77** Die Ausgabe des Programms

```
NUMBER OF CLASSES =  1
CLASS [1]: NUMBER OF QRS  = 3
QRS 1  BEGIN = 253   END = 280     MORPHOLOGY: r
QRS 2  BEGIN = 737   END = 764     MORPHOLOGY: r
QRS 3  BEGIN = 1224  END = 1250    MORPHOLOGY: r

 Is it correct main(e) (y/n)n
```

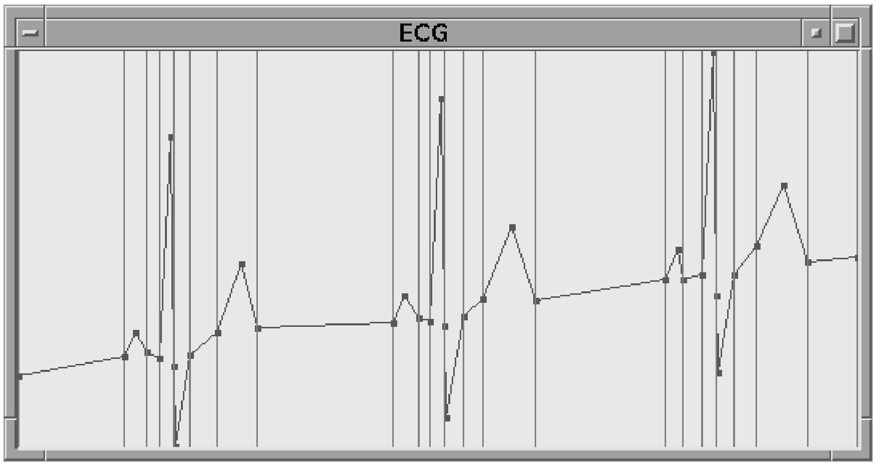



Abbildung 8.10: Die Repräsentation des Eingabe-*EKG*s

Weil die Antwort falsch war, beginnt die Debugging-Phase. Wie früher kann die vom System gestellte Frage entweder mit *yes* oder mit *no* beantwortet werden. Weil wir die *Bottom-Up*-Methode verwenden, bedeutet die erste *no*-Antwort, daß die falsche Klausel gefunden wurde.

In diesem Fall wird der Debugging-Prozeß beendet. Die falsche Klausel erscheint auf dem Bildschirm. Im anderen Fall geht der Debugger weiter und stellt die nächste Frage bei dem Knoten, der in der Postorder-Reihenfolge des Ableitungsbaumes vorkommt.

In diesem Beispiel erscheinen die Parameter für das Prädikat `segment` nicht, weil in der Testbeschreibung die Zeile `type(segment(-,-))` steht. Das graphische Werkzeug zeigt dem Anwender immer das gerade aktuelle Teilmuster. Um dieses von den übrigen Teilmustern unterscheiden zu können, wird eine andere Zeichenfarbe und Linienbreite verwendet (siehe Abbildung 8.11, 8.12 und 8.13).

**Beispiel 78** Die vom System gestellte Frage

```
Is it ok [segment( ? , ? )] (y/n)y
```

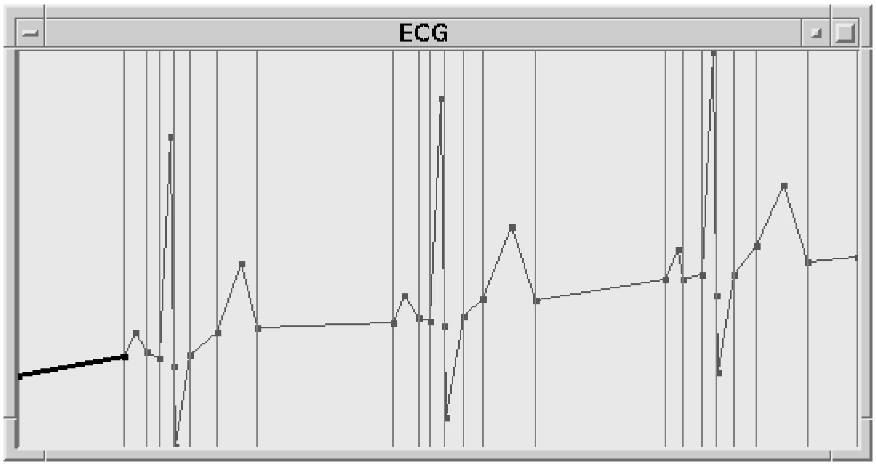



Abbildung 8.11: Die graphische Darstellung zu dem aktuell gefragten Primitiv (Teil 1)

Der Debugger fragt den Benutzer nicht, ob das Prädikat `cardiac_cycles` korrekt oder inkorrekt ist, weil der Debugger den Testrahmen `[zero1, zero2, zero]` identifizieren kann. Dieser Rahmen wurde schon getestet, und beim Testen wurde ihm die Bewertung *korrekt* zugewiesen.

**Beispiel 79** Die zweite Frage im Debugging-Prozeß

```
Is it ok [segment( ? , ? )] (y/n) y
```

Die Testbeschreibung enthält keine Zeile `type(negative_peak())` für das Prädikat `negative_peak`, deswegen ist in diesem Fall `globaltype(all)` gültig. Es bedeutet, daß die ganze Parameterliste des Prädikats `negative_peak` ausgeschrieben wird.

**Beispiel 80** Die Fragestellung zum Prädikat `negative_peak`

```
Is it ok[negative_peak([n_peak,line,p_peak,par,p_peak,line,
p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,line,
p_peak,par],[line,p_peak,par,p_peak,line,p_peak,n_peak,par,
p_peak,par,p_peak,par,p_peak,n_peak,line,p_peak,pa])] (y/n)
```

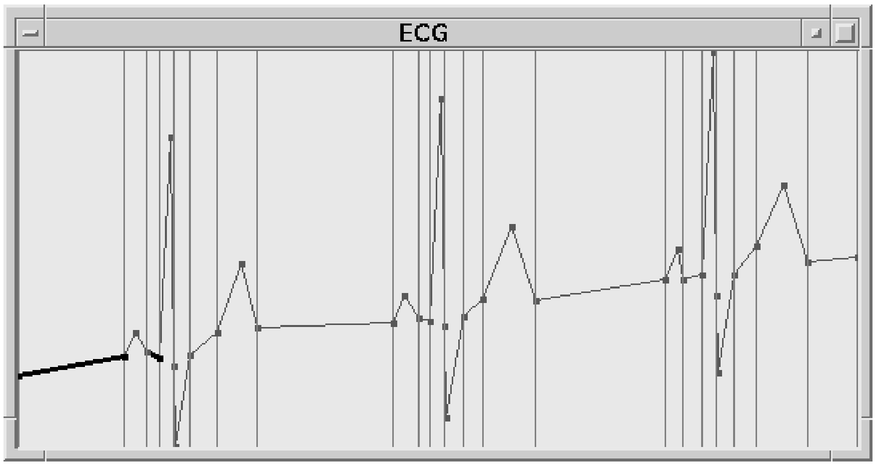



Abbildung 8.12: Die graphische Darstellung zu dem aktuell gefragten Primitiv (Teil 2)

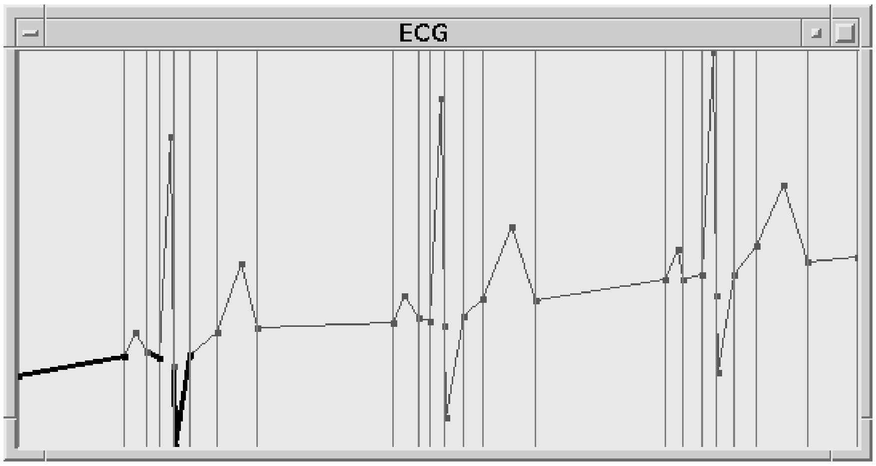



Abbildung 8.13: Die graphische Darstellung zu dem aktuell gefragten Primitiv (Teil 3)

Wenn keine Suchfunktionen definiert wurden (es gibt in der Beschreibung die Zeile *undefined_search(choice_name)*), dann muß der Anwender den entsprechenden Testrahmen selbst zusammenstellen und auswählen. Der Debugger listet die möglichen Wahlmöglichkeiten auf, und der Benutzer kann aus diesen eine auswählen. Danach führt das System eine zusätzliche Prüfung über die Auswahl der Wahlmöglichkeiten durch, ob der Testrahmen wirklich das gerade bewertete Prädikat enthält.

In Abbildung 8.13 kann man erkennen, daß das graphische Werkzeug die gesamte analysierte $QRS$-Wellenform in einem anderen Linienstil zeichnet.

**Beispiel 81** Selektierung der Wahlmöglichkeiten duch den Anwender, um einen Testrahmen bilden zu können

```
Clause : qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )
```

```
     Category :  number_of_the_classes
     1 Choice: "non_zero_classes".
     2 Choice: "zero_classes".
     0. Quit

     Enter selection:   1

  Clause : qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )

     Category :  number_of_the_qrs
     1 Choice: "non_zero_qrs".
     2 Choice: "zero_qrs".
     0. Quit

     Enter selection:  1

 WARNING: Not existing searching function for non_zero_qrs_qrs
 Press <Enter> to continue ...

 Clause : qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )

    Category :  number_of_the_waves
    1 Choice: "non_zero_waves".
    2 Choice: "zero_waves".
    0. Quit

    Enter selection: 1

Is it ok [qrs( 0 , 1 , 0 , 1 , [0] , [1] , 253 , 280 , 12 , ? , ? )] (y/n) n
```

Die Antwort des Benutzers auf die oben gestellte Frage war *no*. Das bedeutet, daß das System die falsche Klausel gefunden hat, und diese wird mit der ganzen Parameterliste auf dem Bildschirm angezeigt.

**Beispiel 82** Die falsche Klausel im Programm *ecg.pl*

```
The false clause:
qrs(0,1,0,1,[0],[1],253,280,12,[p_peak,n_peak,line,p_peak,par,
    p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
    n_peak,line,p_peak,par],[n_peak,line,p_peak,par,p_peak,
    line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,
    line,p_peak,par]):-
insert_last(0,[],[0]),
insert_last(0,[],[0]),
insert_last(0,[],[0]),
retract(user:build_draw_qrs([])),
assert(user:build_draw_qrs([]))),
r(253,280,275,-220,-246,461,158466,11.1,[p_peak,n_peak,
  line,p_peak,par,p_peak,line,p_peak,n_peak,par,p_peak,
  par,p_peak,par,p_peak,n_peak,line,p_peak,par],[n_peak,
  line,p_peak,par,p_peak,line,p_peak,n_peak,par,p_peak,
  par,p_peak,par,p_peak,n_peak,line,p_peak,par]),
insert_last(158466,[],[158466]),
```

```
insert_last(11.1,[],[11.1]),
insert_last(253,[0],[0,253]),
insert_last(280,[0],[0,280]),
insert_last(461,[0],[0,461]),
12=12,
build_qrs(253,-220,275,461,280,-246),
sem_rou1(-220,[0],0,0,1,253,253,1,[1],280,280,1,1,
 [158466],[11.1],[],[0,253],[0,280],[0,461]).

Press <Enter> to continue ...
```

Der Debugging-Prozeß ist beendet. In der Datenbank wurden die folgenden Änderungen durchgeführt: Die aufgetretenen Eingabewerte und die berechneten Ausgabewerte werden in den passenden Klassen gespeichert. Der nicht aufgelistete Teil bleibt unverändert.

**Beispiel 83** Die modifizierte Testdatenbank

```
The list of current test database:
  test(cardiac_cycles, 8, [zero1,zero2,zero],
  [([0,0,[0],0,0,[0],[p_peak,par,line,p_peak,n_peak,line,p_peak,
  par,p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
  n_peak,line,p_peak,par],[p_peak,par,line,p_peak,par,p_peak,
  line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,
  line,p_peak,par]])], correct).
  test(qrs, 11, [non_zero_classes,non_zero_qrs,non_zero_waves],
  [[0,1,0,1,[0],[1],253,280,12,[p_peak,n_peak,line,p_peak,par,
  p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
  n_peak,line,p_peak,par],[n_peak,line,p_peak,par,p_peak,line,
  p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,n_peak,line,
  p_peak,par]],[0,1,0,1,[0],[1],253,280,12,[p_peak,n_peak,line,
  p_peak,par,p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,
  par,p_peak,n_peak,line,p_peak,par],[n_peak,line,p_peak,par,
  p_peak,line,p_peak,n_peak,par,p_peak,par,p_peak,par,p_peak,
  n_peak,line,p_peak,par]])], incorrect).
```

In diesem Kapitel wurde das *SPECG*-System vorgestellt, Wir haben gezeigt, wie dieses System *EKG*-Wellenformen klassifizieren kann. Das Ziel war, Regeln zu definieren, die die sogenannten *QRS*-Komplexe von den anderen Mustern trennen. Dieses System hilft dem Benutzer, das Klassifikatorprogramm zu verbessern, wenn es die vorgegebene Eingabe nicht klassifizieren kann.

Das *SPECG*-System kombiniert einen Elektrokardiogramm-Wellenklassifikator mit dem Debuggingsystem *IDTS*. Nach einer sehr kurzen Einführung in die Anatomie des Herzens und die *EKG*-Diagnostik wurde am Ende des Kapitels durch ein Beispiel gezeigt, wie der Fehler in der Grammatik gefunden werden kann.

# Teil III

# Integration in komplexe Systeme

# Kapitel 9

# Erweiterung des Systems um ein Lernverfahren

In diesem Kapitel werden wir eine Erweiterung von *SPECG* vorstellen. Wir haben in das System die *ILP*-Lernmethode *IMPUT* (Interactive Multiple Predicate Learner Using Unfolding Transformation) integriert. Damit kann das System nicht nur die *EKG*-Wellenformen klassifizieren, sondern kann mit Hilfe von positiven und negativen Beispielen und Hintergrundwissen eine korrekte Grammatik erzeugen. Das bedeutet, daß das von uns entwickelte System die syntaktischen und semantischen Regeln einer *EKG*-Grammatik lernen kann.

Die Hauptidee war, daß wir das schon in Kapitel 8.5 und in [KAK94] oder [AGK95] beschriebene *SPECG*-System weiterentwickeln, indem wir ein *EKG*-Vorverarbeitungs-, ein Lernmodul und eine graphische Benutzeroberfläche in das System integrieren. Das Lernmodul basiert auf der *induktiven Lernmethode* (*ILP*, siehe [BG95], [MR94]) und heißt *IMPUT* ([AGB97]). Das *IMPUT*-Verfahren kombiniert *IDTS* mit dem Spezialisierungsalgorithmus SPECTRE ([BIA94]).

Wie unsere Ergebnisse zeigen, können mit Hilfe dieser Methode nicht nur die Fehler im Programm gefunden werden, sondern wir können ein korrektes Programm gewinnen, wenn wir *Unfolding-Transformationen* und den *IDTS*-Debugger benutzen. Wenn *IMPUT* benutzt wird, kann das System dem Benutzer eine korrekte Lösung vorschlagen, weil die fehlerhafte Klausel durch eine neue ersetzt werden kann.

Dieses System hilft nicht nur, eine Grammatik, die *EKG*-Wellenformen erkennt und klassifiziert, zu verbessern, sondern demonstriert, wie große Programme (die *EKG*-Klassifizierung ist länger als 2000 Zeilen) durch die *theory revision learning* Methode gelernt werden können.

Bei der Entwicklung des Systems haben wir festgestellt, daß eine graphische Oberfläche sehr nützlich wäre. Wir haben eine leicht modifizierte Version der **GUI** von *IDTS* (siehe Kapitel6.4) verwendet. Damit fällt dem Benutzer leicht, dem Debuggingprozeß zu folgen, weil zum Beispiel der Ableitungsbaum angezeigt wird. Wir wissen schon, daß *IDTS* während des Debuggings dem Anwender Fragen stellt. Diese Fragen können sehr schwierig zu beantworten sein, weil sie viele Parameter enthalten können. Deswegen haben wir zusätzliche Funktionen implementiert, so daß der Benutzer in jedem Moment des Prozesses die gerade analysierte Teilwellenform sieht.

Um unser System testen zu können, brauchten wir eine Menge von *EKG*-Wellenformen, die wir von der Klinik für Innere Medizin der medizinischen Universität Szentgyörgyi Albert bekommen haben. Um diese Dateien, die von *EKG*-Geräten stammen, verarbeiten zu können, haben wir sie zuerst auf einen Rechner übertragen (siehe Abbildung 9.1). Die kodierten Wellenformen haben wir dekodiert und mit Hilfe des *EKG*-Vorverarbeitungsprogramms auf dem Bildschirm angezeigt. Das Programm erlaubt die interaktive Segmentierung einer *EKG*-Aufname. Aus den Segmenten werden die numerischen Parameter berechnet. Ferner generiert dieses Modul eine Datei in einem Format, das von den anderen Modulen weiterverarbeitet werden kann.

**Beispiel 84** Die Übertragung die *EKG*-Daten

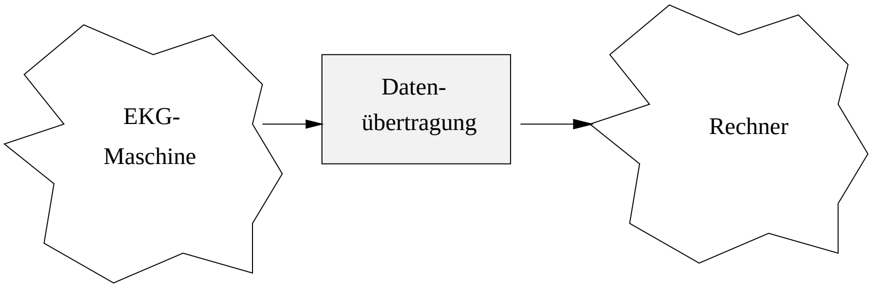



Abbildung 9.1: Die Importierung der Daten auf einen Rechner

In diesem Kapitel geben wir zuerst eine Einführung in die induktive logische Programmierung *ILP*, weil das Lernsystem *IMPUT* die Techniken von *ILP* verwendet. Dann beschreiben wir das *IMPUT*-Verfahren selbst. Im Rest des Kapitels erläutern wir die Erweiterung des *SPECG*-Systems, und durch ein Beispiel zeigen wir, wie die Syntax und die Semantik der Grammatik gelernt wird.

## 9.1 Induktive Logische Programmierung (ILP) als maschinelle Lernstrategie

Ein Teilgebiet der künstlichen Intelligenz, das sich mit Lernen beschäftigt, heißt *maschinelles Lernen* (*machine learning*). Es gibt mehrere Ansätze, wie Lernen mit Hilfe von *neuronalen Netzen*, *genetischen Algorithmen*, *Entscheidungsbäumen* und *Logik*. *Induktive Logische Programmierung* (*ILP*) ist eine neue Technik, die die Grundprinzipien von induktivem maschinellen Lernen mit der logischen Programmierung kombiniert. Der Name stammt von Stephen Muggleton [Mug92]. Mehr über die Geschichte dieses Gebiets kann in [NCdW96], S. 174ff nachgelesen werden. Das Ziel dieser neuen Lernmethode ist die Induktion von allgemeinen Regeln aus *speziellen Beobachtungen* (*positiven und negativen Beispielen*) und aus *Hintergrundwissen* (*background knowledge*). Kurz: *ILP* = I $\cap$ LP, was bedeutsam, daß *ILP* nichts anderes ist als die Anwendung von Techniken, die auf Induktion und logischer Programmierung beruhen. Diese Verfahrensweise macht *ILP* viel leistungsfähiger im Vergleich zu anderen traditionellen Techniken:

- *ILP* benutzt die ausdrucksvollere *Logik erster Ordnung* (first-order logic) im Gegensatz zu früher verwendeten, auf Attributwerten basierenden Lernverfahren. Die allgemeine Logik und besonders die Logik erster Ordnung ist mathematisch sehr gut begründet

[Llo87b] und *ILP* kann eine Menge von Ergebnissen, Techniken und Konzepten aus der Logik benutzen.

- Logik gewährt eine sehr ausdrucksvolle Repräsentationsmöglichkeit: Die Beispiele, das Hintergrundwissen (wobei auch ein Vorteil ist, daß *ILP* überhaupt Hintergrundwissen berücksichtigen kann) und die Hypothese selbst können als Formeln in der Klauselsprache eingegeben werden. Besonders bedeutend ist, daß die Hypothese und das Hintergrundwissen dieselbe Repräsentation haben, obwohl sie aus verschiedenen Quellen kommen: Die Hypothese wird vom System selbst erzeugt, Hintergrundwissen muß vom Anwender definiert werden. Außerdem existieren Anwendungsgebiete, die nicht nur mit herkömmlichen Attributwerten beschreibbar sind, wie zum Beispiel Lernen natürlicher Sprache.

- Die Repräsentation des Hintergrundwissens in Form von Regeln und Fakten steht der natürlichen Sprache näher als alle anderen Methoden des maschinellen Lernens. Eine Menge von Klauseln, die uns *ILP* liefert, ist leichter zu verstehen als zum Beispiel ein neuronales Netz.

### 9.1.1 Das Hauptproblem von ILP

Das Hauptproblem bei der induktiven Logikprogrammierung (*ILP*) ist, logische Programme aus Beispielen und Hintergrundwissen zu lernen. Formal kann es so definiert werden:

**Definition 100** Voraussetzungen:

- Eine Beschreibungssprache $\mathcal{L}$, die die syntaktischen Beschränkungen über dem Prädikat $p$ spezifiziert.
- Eine Menge von Beobachtungen $\mathcal{E}$ bestehend aus positiven Beispielen $\mathcal{E}^+$ und negativen Beispielen $\mathcal{E}^-$, die Grundatome sind, und es gilt:
  $(\mathcal{E} \subset \mathcal{L}) \wedge (\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-) \wedge (\mathcal{E}^+ \cap \mathcal{E}^- = \emptyset)$.
- Eine Menge von Hintergrundwissen $\mathcal{B}$, bestehend aus Prädikaten $q_i$, mit deren Hilfe zusätzliche Informationen über die ursprünglich vorhandenen Prädikate $p$ angegeben werden können.

Dann bezeichnet *ILP* das Finden einer Hypothese $\mathcal{H}$ für $p$, die der Sprachvorgabe $\mathcal{L}$ genügt, so daß gilt:

- $\forall e \in \mathcal{E}^+$: $\mathcal{B} \cup \mathcal{H} \models e$ und
- $\forall e \in \mathcal{E}^-$: $\mathcal{B} \cup \mathcal{H} \not\models e$

Eine Verallgemeinerung der oben angegebenen Definition ist die *ILP-Theorierevision*, die besagt, daß nicht nur Hintergrundwissen und Beispiele, sondern auch schon eine Hypothese als Anfangspunkt definiert wird. Unser Lernverfahren *IMPUT* (siehe Kapitel 9.2) erfüllt diese Aufgabe. Eine ausführliche Beschreibung dieses Gebietes kann unter [LW96] nachgelesen werden.

**Definition 101** Eine Hypothese $\mathcal{H}$ heißt *vollständig*, wenn $\mathcal{B} \wedge \mathcal{H} \models \mathcal{E}^+$ und *konsistent*, wenn $\mathcal{B} \wedge \mathcal{H} \wedge \mathcal{E}^- \not\models \square$

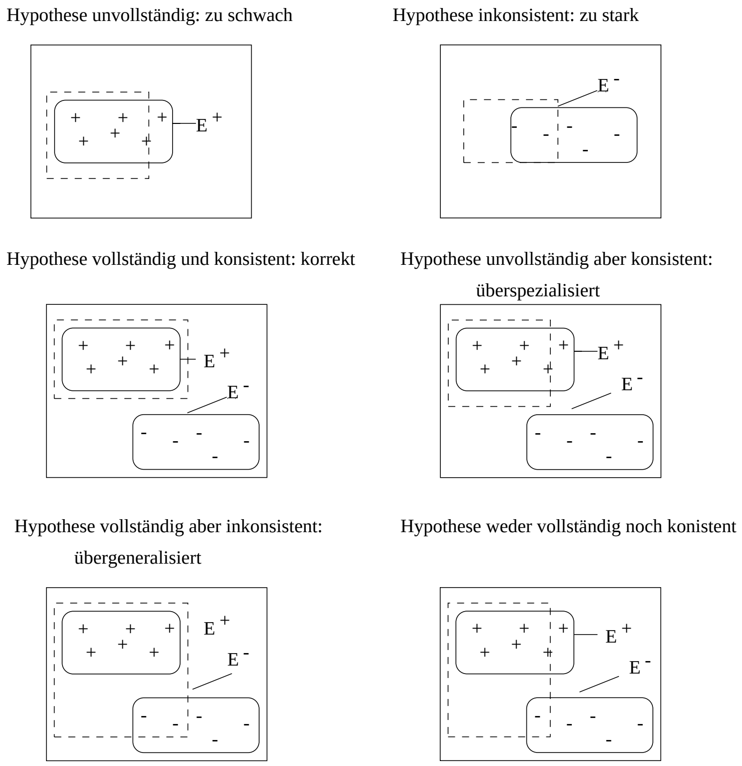



Abbildung 9.2: Die Vollständigkeit und Konsistenz der Hypothese

Die vorkommenden sechs Fälle zeigt die Abbildung 9.2 (aus [LD93a]). Veranschaulichen wir die Definitionen durch das schon im Kapitel 2.1.4 verwendete *daughter* Beispiel. Die Aufgabe ist, das Prädikat `daughter(X,Y)` zu lernen. Es kann so interpretiert werden, daß die Person `X` eine Tochter von `Y` ist. Laut Definition von *ILP* brauchen wir Hintergrundwissen, das zwei Klauseln `parent` und `female` enthält. Außerdem müssen positive und negative Beispiele eingegeben werden.

**Beispiel 85** Lernen von Verwandtschaftsbeziehungen
Hintergrundwissen:

```
parent(ann,mary).
parent(ann,tom).
parent(tom,eve).
parent(tom,ian).
female(ann).
female(mary).
female(eve).
```

Positive Beispiele $\mathcal{E}^+$:

```
daughter(mary,ann).
daughter(eve,tom).
```

Negative Beispiele $\mathcal{E}^-$:

```
daughter(tom,ann).
daughter(eve,ann).
```

Die folgende Klausel ist eine korrekte Lösung:

```
daughter(X,Y) :- female(X), parent(X,Y).
```

Diese Definition ist korrekt und vollständig in Hinblick auf Hintergrundwissen und Beispiele.

In Wirklichkeit kann die Definition des Zielprädikates abhängig von $\mathcal{B}$ und $\mathcal{L}$ und der Komplexität des Zielbegriffes mehrere Klauseln enthalten. Nehmen wir an, daß das Prädikat `parent` durch zwei weitere Prädikate `mother` und `father` definiert wurde. Dann könnte die Lösung zum Beispiel folgende sein:

```
daughter(X,Y) :- female(X), mother(X,Y).
daughter(X,Y) :- female(X), father(X,Y).
```

Wenn die vollständige Prädikatenlogik als $\mathcal{L}$ verwendet wird und der Ableitungsmechanismus auf der allgemeinen logischen Folgerbarkeit basiert, dann tritt das Problem der Unentscheidbarkeit auf, nämlich der Unentscheidbarkeit der Prädikatenlogik. Es bedeutet, daß es innerhalb des System nicht überprüfbar ist, ob eine Hypothese tatsächlich eine Lösung des Problems ist oder nicht. Es gibt zwei Möglichkeiten, Entscheidbarkeit zu erreichen:

- Beschränkung der Sprache auf eine echte Untermenge der Prädikatenlogik erster Ordnung
- Verwendung der Relation *$\theta$-Subsumption* als logische Folgerbarkeit ($\models$)

Was die Sprache betrifft, werden in den meisten *ILP*-Systemen die Hornklauseln (siehe Definition 7) als Hypothesensprache verwendet. Diese sind ebenfalls nicht entscheidbar, wie Kietz und Džeroski in ihrer Arbeit ([KD94]) durch ein Programm gezeigt haben, das aus einem Grundatom, einer Grundfrage und zwei Hornklauseln mit nur zwei Literalen besteht. Deswegen sind weitere Einschränkungen nötig. Mehr kann hierzu in [LW96] nachgelesen werden.

Außerdem benutzen die *ILP*-Systeme statt der Folgerung ($\models$) die von Robinson definierte (siehe [Rob65]) und für das Lernen von Plotkin (siehe [Plo70]) verwendete Relation $\theta$-Subsumption ($\geq_\theta$), die wir im Kapitel 2.1.4 in der Definition 28 schon angegeben haben. Dort haben wir auch ein Beispiel gezeigt. Hier wollen wir nur kurz zwei Eigenschaften der $\theta$-Subsumption erwähnen:

- Wenn $C_i \geq_\theta C_j$, wobei $C_i$ und $C_j$ Klauseln sind, dann gilt $C_i \models C_j$. Die umgekehrte Richtung ($C_i \models C_j \Rightarrow C_i \geq_\theta C_j$) gilt nicht, wie Flach im Artikel [Fla94] gezeigt hat.
- Die Relation $\theta$-Subsumption führt eine *Verbandsstruktur* (*lattice*) ein. Das bedeutet, daß zwei Klauseln eine *kleinste obere Schranke* (*least upper bound, lub*) und eine *größte untere Schranke* (*greatest lower bound, glb*) haben. Beide Schranken sind, abgesehen von Umbenennungen der Variablen, eindeutig.

Was sind die kleinste obere und die größte untere Schranke? Um sie erklären zu können, müssen wir noch zwei andere Begriffe einführen.

**Definition 102** Eine Klausel $C_i$ ist mindestens so allgemein (*at least general*) wie die Klausel $C_j$ ( $C_i \leq C_j$), wenn $C_i$ die Klausel $C_j$ $\theta$-subsumiert.

**Definition 103** Eine Klausel $C_i$ ist *allgemeiner* (*more general*) als eine Klausel $C_j$ ( $C_i < C_j$), wenn $C_i \leq C_j$ gilt, aber $C_j \not\leq C_i$.

Wenn $\mathcal{B}$ das Hintergrundwissen ist, dann bedeutet dies, daß $C_i \wedge \mathcal{B} \models C_j$ gilt. In diesem Fall sagen wir, daß $C_j$ eine *Spezialisierung* (*refinement*) von $C_i$ und $C_i$ eine *Generalisierung* von $C_j$ ist.

Im weiteren spielt der Begriff *minimale Verallgemeinerung* (*least general generalization, lgg*) eine wesentliche Rolle, der mit Hilfe der $\theta$-Subsumption definiert werden kann.

**Definition 104** Die *minimale Verallgemeinerung* von zwei Klauseln $C_i$ und $C_j$ (lgg($C_i, C_j$)) ist die kleinste untere Schranke von $C_i$ und $C_j$ in der $\theta$-Subsumptionsverbandsstruktur.

Was bedeutet das in unserem *daughter*-Beispiel?

**Beispiel 86** Minimale Verallgemeinerung mit $\theta$-Subsumption
Seien zwei Klauseln gegeben:

```
daughter(mary,ann) :- female(mary), parent(ann,mary).
daughter(eve,tom)  :- female(eve),  parent(tom,eve).
```

Für diese zwei Klauseln ist die minimale Verallgemeinerung

```
daughter(X,Y) :- female(X), parent(Y,X).
```

Diese Relation $\theta$-Subsumption bietet keine Lösung für das Problem, wenn das Prädikat, das wir lernen wollen, selbst rekursiv ist. Außerdem versucht man in der Praxis immer nur ein Zielprädikat zu lernen. Das Lernen von mehreren Zielprädikaten wird auf das sukzessive Lernen einer Prädikatfolge zurückgeführt ([RLD93]).

Die Bedeutung der $\theta$-Subsumption ist folgende:

- Sie bietet eine Möglichkeit, die Hypothesen zu ordnen, weil sie den Hypothesenraum strukturiert.
- Mit Hilfe der $\theta$-Subsumption kann der Suchraum verkleinert werden:
    - Wenn $C_j$ aus $C_i$ generalisiert wird ($C_j < C_i$), werden alle von $C_i$ abgedeckten Beispiele auch mit $C_j$ ableitbar, denn, wenn $\mathcal{B} \cup \{C_i\} \models e$ wahr ist, dann ist $\mathcal{B} \cup \{C_j\} \models e$ auch wahr. Wenn $e \in \mathcal{E}^-$, dann bedeutet das, daß alle Verallgemeinerungen von $C_i$ inkonsistent sind, also aus diesen Klauseln die negativen Beispielen auch ableitbar sind; deswegen können sie weggelassen werden.
    - Wenn $C_j$ aus $C_i$ spezialisiert wird ($C_i < C_j$) und ein Beispiel $e$ nicht mit Hilfe von $C_i$ ableitbar ist, dann kann keine Spezialisierung von $C_i$ verwendet werden, um $e$ abzuleiten, weil $\mathcal{B} \cup \{C_i\} \not\models e$ wahr ist und dann $\mathcal{B} \cup \{C_j\} \not\models e$ auch wahr ist. Wenn $e \in \mathcal{E}^+$, dann bedeutet es, daß alle Spezialisierungen von $C_i$ ebenfalls weggelassen werden können.

Im nächsten Teil werden wir kurz die in *ILP* verwendeten Grundtechniken beschreiben, die auf Generalisierung und Spezialisierung beruhen.

### 9.1.2 Grundtechniken in ILP

Wie wir schon im letzten Kapitel kurz erwähnt haben, gibt es zwei Möglichkeiten, den Hypothesenraum zu durchsuchen. Deswegen kann *ILP*, wie andere induktive Lernmethoden ([Mit82]), als ein Suchprozeß betrachtet werden. *Bottom-Up-Methoden* sind *Generalisierungstechniken*: Sie starten aus der Beispielmenge $\mathcal{E}$ und durchsuchen den Hypothesenraum, wobei sie Generalisierungsoperatoren verwenden. Diese Technik kann gut beim interaktiven und inkrementellen Lernen verwendet werden, wo die Anzahl der Elemente von $\mathcal{E}$ klein ist. Die andere Vorgehensweise ist, die Suche *Top-Down* durchzuführten. In diesem Fall sprechen wir von *Spezialisierungstechniken*. Die von uns in Kapitel 9.2 vorgestellte Methode *IMPUT* (Interactive Multiple Predicate Learner Using Unfolding Transformation) gehört auch dazu. Wie der Name schon sagt, gehen wir in diesem Fall von der allgemeinsten Hypothese in die Richtung der spezialisierten Hypothese; natürlich werden dabei Spezialisierungsoperatoren verwendet. Diese Technik ist gut anwendbar beim empirischen Lernen, wenn auch *Rauschen* (*noise*) enthalten ist.

Es gibt noch einige andere Klassifizierungen von *ILP*-Methoden:

- *Induktive Inferenz:* Es gibt Positivbeispiele, für die das Programm *nicht terminiert.* Häufig ist die Anfangshypothese das leere Programm bzw. das fehlerbehaftete Programm, in welchem die Fehler gefunden werden sollen. Dieses fehlerhafte Programm kann nun beispielsweise durch Hinzufügen von neuen Klauseln korrigiert werden. Solche Klauseln kann man durch Generalisierungsalgorithmen erhalten.

- *Abduktive Inferenz:* Es gibt Negativbeispiele, für die das Programm *terminiert.* Das Anfangsprogramm kann durch Entfernen oder Verändern von Klauseln korrigiert werden. Hierfür kann man zum Beispiel *Unfolding*- und/oder *Subsumptions*-Operatoren verwenden.

Programmspezialisierung ist eine abduktive Lernmethode. Von der Anfangshypothese verlangt man, daß sie für alle Positivbeispiele terminiert und für einige Negativbeispiele ebenfalls terminiert. Ausgabe des Algorithmus ist eine Modifikation der Anfangshypothese. Sie wird so lange spezialisiert, bis für sie keine Negativbeispiele mehr ableitbar sind.

In diesem Teil der Arbeit werden wir die zwei Methoden Generalisierung und Spezialisierung kurz einführen. Eine theoretische Einführung kann in [MR94] nachgelesen werden.

#### Generalisierungstechnik in ILP

Generalisierungstechniken beginnen mit dem Durchsuchen des Hypothesenraums, um als Anfangspunkt eine spezialisierte Klausel zu finden. Diese Klausel deckt ein positives Beispiel ab. Dann versuchen sie durch die Top-Down-Methode diese Klausel so lange zu verallgemeinern, wie dies möglich ist. Dabei wird beachtet, daß die so erzeugte Klausel kein negatives Beispiel abdeckt. Der in dieser Technik verwendete Operator kann wie folgt definiert werden.

**Definition 105** Sei $\mathcal{L}$ die Sprache. Der *Generalisierungsoperator* ist eine Funktion $\wp$, die $C_i \in \mathcal{L}$ auf $C_{i1}, \ldots, C_{in} \in \mathcal{L}$ abbildet, wobei $C_{ij}$ $(1 \leq j \leq n)$ eine Generalisierung von $C_i$ ist:

$$\wp(C_i) \subseteq \{C_{ij} | C_{ij} \in \mathcal{L}, C_{ij} \geq C_i\}$$

Die Generalisierung kann einfach als die entgegengesetzte Richtung der Spezialisierung betrachtet werden. Daraus folgt, daß die dort verwendeten Operatoren durch Umkehrung als

Generalisierungsoperatoren definiert werden können. Eine der im Kapitel 2.2.1 eingeführten Regeln ist die Resolution, die hier als *inverse Resolution* angewendet werden kann ([MB88]). Der andere Generalisierungsoperator kann auf die Substitutionsregel zurückgeführt werden. Wegen der hier auftretenden Schwierigkeiten (nichtdeterministische Auswahl) geht die Forschung jedoch stärker in die Richtung, neue Operatoren, wie zum Beispiel eine *inverse Implikation* [ALLM94], zu definieren. Eine ausführliche Beschreibung hierzu kann man in [MR94] finden.

Muggleton hat die zwei wichtigsten Generalisierungsoperatoren, die *relativ kleinste Generalisierung* (*relative least generalization*, *rlgg*) und die *inverse Resolution*, vereinigt und nennt sie *spezifischster inverser Resolvent* (*most specific inverse resolvent*). Um $rlgg$ beschreiben zu können, müssen wir zuerst $lgg$ (die minimale Verallgemeinerung von zwei Klauseln, siehe Definition 104) berechnen können. $lgg$ kann mit Hilfe der $\theta$-Substitution definiert werden.

**Definition 106** $lgg$ für Terme: $lgg(t_i, t_j)$

- $lgg(t,t) = t$.
- $lgg(f(s_1,\dots,s_m), f(t_1,\dots,t_m)) = f(lgg(s_1,t_1),\dots,lgg(s_m,t_m))$.
- $lgg(f(s_1,\dots,s_m), g(t_1,\dots,t_n)) = V$, wo $f \neq g$ und $V$ ist eine Variable, die $lgg(f(s_1,\dots,s_m), g(t_1,\dots,t_n))$ repräsentiert.
- $lgg(s,t) = V$, wo $s$ und $t$ Terme sind ($s \neq t$) und entweder $s$ oder $t$ oder beide Variable sind, dann ist $V$ eine Variable, die $lgg(s,t)$ repräsentiert.

**Beispiel 87** $lgg$ für Terme

$$lgg([a,b,c],[a,c,d]) = [a,X,Y]$$
$$lgg(f(a,a),f(b,b)) = f(lgg(a,b),lgg(a,b)) = f(V,V)$$

**Definition 107** $lgg$ für Atome: $lgg(A_i, A_j)$

- $lgg(p(s_1,\dots,s_n), p(t_1,\dots,t_n)) = p(lgg(s_1,t_1),\dots,lgg(s_n,t_n))$, wenn beide Atome den gleichen Prädikatnamen haben.
- $lgg(p(s_1,\dots,s_n), q(t_1,\dots,t_m))$ ist nicht definiert, wenn $p \neq q$.

**Definition 108** $lgg$ für Literale: $lgg(L_i, L_j)$

- Wenn $L_i$ und $L_j$ atomar sind, dann wird $lgg$ gemäß Definition 107 gebildet.
- Wenn $L_i$ und $L_j$ negative Literale sind: $L_i = \overline{A_i}$ und $L_j = \overline{A_j}$, dann $lgg(L_i, L_j) = lgg(\overline{A_i}, \overline{A_j}) = \overline{lgg(A_i, A_j)}$.
- Wenn $L_i$ ein positives und $L_j$ ein negatives Literal (oder umgekehrt) ist, dann ist $lgg(L_i, L_j)$ undefiniert.

**Beispiel 88** $lgg$ für Literale, die das Prädikat *daughter* beschreiben:

$$lgg(parent(ann,mary), parent(ann,tom)) = parent(ann,X)$$

$$lgg(parent(ann,mary), \overline{parent(ann,tom)}) = undefined$$

**Definition 109** *lgg* für Klauseln: $lgg(C_i, C_j)$

- Seien $C_i = \{L_{i1}, \ldots, L_{in}\}$ und $C_j = \{L_{j1}, \ldots, L_{jm}\}$. Dann $C_q = lgg(C_i, C_j) = \{L_{qkl} = lgg(L_{ik}, L_{jl}) \mid 1 \leq k \leq n,\ 1 \leq l \leq m,\ L_{ik} \in C_i,\ L_{jl} \in C_j$ und $lgg(L_{ik}, L_{jl})$ ist definiert.$\}$

**Beispiel 89** *lgg* für Klauseln daughter
Wenn $C_i$ ist:

```
daughter(mary,ann) :- female(mary), parent(ann,mary).
```

und $C_j$ ist:

```
daughter(eve,tom) :- female(eve), parent(tom,eve).
```

dann ist $lgg(C_i, C_j)$

```
daughter(X,Y) :- female(X), parent(Y,X).
```

wobei $X = lgg(mary, eve)$ und $Y = lgg(ann, tom)$ sind.

Nach Plotkin [Plo70] und mit Algorithmus 1, den wir im Kapitel 2.2.1 definiert haben, wo wir die Berechnung des allgemeinsten Unifikators gezeigt haben, kann der folgende Algorithmus definiert werden.

**Algorithmus 41** Suche nach *lgg*($L_1$, $L_2$)

**Eingabe**: $L_1$ und $L_2$ sind Atome oder Literale
**Ausgabe**: $lgg(L_1, L_2) = L_3$
**function** $lgg(L_1, L_2) : L_3$
**begin**
  **if** head($L_1$) $\neq$ head($L_2$) **then**
    $L_3 := \varepsilon$; **return**($L_3$)
  **else**
    $V_1$:=$L_1$; $V_2$:=$L_2$
    **while** $V_1 \neq V_2$ **do**
      find_position($V_1$, $V_2$, $t_1$, $t_2$);
      generate_variable $V_{t_1,t_2}$;
      substitute($V_1$, $V_2$, $t_1$, $t_2$, $V_{t_1,t_2}$)
    **endwhile**
    $L_3 := V_1$; **return**($L_3$)
  **endif**
**end**

Dieser Algorithmus berechnet die minimale Verallgemeinerung von zwei Atomen oder Literalen. Zuerst wird überprüft, ob $L_1$ und $L_2$ das gleiche Prädikat oder Atom haben. Wenn dies nicht der Fall ist, gibt es keine *lgg*. Danach wird nach der ersten Position gesucht, an der sich die zwei Eingabeatome oder Literale unterscheiden bzw. eine von beiden eine Variable enthält (*find_position*) In der nächsten Prozedur wird das gefundene Unterscheidungspaar ($t_1$,$t_2$) überall, wo sie in der gleichen Position auftreten, durch eine Variable ersetzt (*substitute*), deren Name aus dem Unterscheidungspaar erzeugt wird (es gilt: $V_{i,j} = V_{j,i}$). Dieses Verfahren wiederholt sich so lange, bis die beiden Atome bzw. Literale $V_1$ und $V_2$ gleich sind.

**Algorithmus 42** Suche nach verschiedenen Teilen in zwei Termen oder Literalen

**Eingabe**: $V_1$ und $V_2$ sind Terme oder Literale
**Ausgabe**: $t_1 \in V_1$ und $t_2 \in V_2$ Terme
**procedure** find_position($V_1$, $V_2$, $t_1$, $t_2$)
**begin**
  **if** $V_1 = V_2$ **then return**$(\varepsilon, \varepsilon)$
  **endif**
  **if** ($V_1$ is atomic **or** $V_2$ is atomic) **then return**$(V_1, V_2)$
  **endif**
  $V_1 = f_1(t_{11}, \ldots, t_{1n})$;
  $V_2 = f_2(t_{21}, \ldots, t_{2m})$
  **if** $f_1 \neq f_2$ **then return**$(V_1, V_2)$
  **endif**
  i:=1
  **while** i $\leq$ n **do**
    find_position($t_{1i}$, $t_{2i}$, $t_1$, $t_2$)
    **if** $t_1 \neq \varepsilon$ **then return**$(t_1, t_2)$
    **endif**
    i:= i+1
  **endwhile**
**end**

Die Prozedur *find_position* bekommt als Eingabeparameter zwei Terme oder Literale $V_1$ und $V_2$. Sie sucht nach einem Unterscheidungspaar und gibt die Teilterme $t_1$ und $t_2$ zurück, die an der Position stehen, wo $V_1$ und $V_2$ nicht mehr gleich sind. Wo können sie verschieden sein? Wenn $V_1 = V_2$, dann gibt es natürlich kein Unterscheidungspaar, so daß $t_1$ und $t_2$ die leeren Terme sind ($\varepsilon$). Wenn $V_1$ oder $V_2$ atomar ist, dann haben wir das gesuchte Unterscheidungspaar gefunden. Ansonsten können beide nur zusammengesetzte Terme sein in der Form $V_i = f_i(t_{i1}, \ldots, t_{in})$, wobei $i = 1, 2$ ist. Sind $f_1$ und $f_2$ verschieden, hat man ein Unterscheidungspaar gefunden. Sonst sind $n$ und $m$ gleich und wir müssen rekursiv vorgehen: Solange die Teilterme der gerade überprüften Terme gleich sind, müssen wir die Prozedur *find_position* rekursiv mit den aktuellen Teiltermen aufrufen.

**Algorithmus 43** Ersetze überall $t_1$ und $t_2$ durch eine Variable in $V_1$ und $V_2$

**Eingabe**: $V_1$ und $V_2$ Terme oder Literale, $t_1$ und $t_2$ Terme, $X$ Variable
**Ausgabe**: $V_1$ und $V_2$ mit Substitution $X$ für $t_1$ und $t_2$
**procedure** substitute($V_1$, $V_2$, $t_1$, $t_2$, $X$)
**begin**
  **if** $V_1 = t_1$ **and** $V_2 = t_2$ **then return**$(X, X)$
  **endif**
  **if** $V_1 = V_2$ **then return**$(V_1, V_2)$
  **endif**
  **if** ($V_1$ is atomic **or** $V_2$ is atomic) **then return**$(V_1, V_2)$
  **endif**
  $V_1 = f_1(t_{11}, \ldots, t_{1n})$;
  $V_2 = f_2(t_{21}, \ldots, t_{2m})$
  **if** $f_1 \neq f_2$ **then return**$(V_1, V_2)$
  **endif**
  i:=1
  **while** i $\leq$ n **do**
    substitute($t_{1i}$, $t_{2i}$, $t_1$, $t_2$, $X$);
    i:= i+1
  **endwhile**
**end**

Wenn der Unterschied zwischen zwei Termen/Literalen mit Hilfe der Prozedur $find\_position$ gefunden wurde, müssen in den beiden Termen/Literalen $V_1$ und $V_2$ die entsprechenden Teilterme $t_1$ und $t_2$ durch eine neu generierte Variable $X$ ersetzt werden. Aber nicht nur in der Position, wo sie gefunden werden, sondern in allen Positionen, wo das gleiche Unterscheidungspaar auftritt. Das geschieht folgendermaßen: Es gibt nur im Fall zusammengesetzter Terme mit $f_1 \neq f_2$ etwas zu tun, also für die Form $V_i = f_i(t_{i1}, \ldots, t_{in})$, wobei $i = 1, 2$ gilt. In diesem Fall muß nach der ersten Ersetzung rekursiv vorgegangen werden, und mit Hilfe des rekursiven Aufrufs von *substitute* ist zu überprüfen, ob $t_1$ und $t_2$ an weiteren gleichen Positionen auftreten. Wenn dies der Fall ist, dann muß auch dort eine Substitution durchgeführt werden. Das Ergebnis sind die geänderten $V_1$ und $V_2$.

Jetzt sind wir so weit, daß wir die *minimale Verallgemeinerung bezüglich des Hintergrundwissens* (*rlgg*) definieren können.

**Definition 110** Die *minimale Verallgemeinerung bezüglich des Hintergrundwissens* (*relative least general generalization*) von zwei Klauseln $C_i$ und $C_j$ ist die minimale Verallgemeinerung lgg($C_i$,$C_j$) relativ zum Hintergrundwissen $\mathcal{B}$.

Hier müssen wir also auch das vordefinierte Hintergrundwissen betrachten. Die Beispiele in *ILP* sind als Grundatome definiert $(A_1, A_2, \ldots)$. Das Hintergrundwissen $C_{\mathcal{B}}$ enthält Grundfakten, die mit Konjunktion zusammengesetzt werden können:

$$rlgg(A_1, A_2) = rlgg((A_1 \leftarrow C_{\mathcal{B}}), (A_2 \leftarrow C_{\mathcal{B}}))$$

Die Berechnung von rlgg ist einfach die Interpretation von Plotkins Algorithmus auf Klauseln.

**Algorithmus 44** Suche nach *rlgg($C_i, C_j$)*

**Eingabe**: $C_i$ und $C_j$ sind zwei Klauseln
in der Form $C_i = \{L_{i1}, \ldots, L_{in}\}$ und $C_j = \{L_{j1}, \ldots, L_{jm}\}$
**Ausgabe**: $rlgg(C_i, C_j) = C_k$
**function** $rlgg(C_i, C_j) : C_k$
**begin**
  k := 1; l := 1; $C_l$ := {}
  **while** $k < n$ **do**
    **while** $l < m$ **do**
      lgg($L_{ik}, L_{jl}, L$);
      $C_l := C_l \cup \{L\}$;
      l := l+1
    **endwhile**
    k := k+1
  **endwhile**
  $C_k := C_l$;
  **return**($C_k$)
**end**

Sehen wir für ein Beispiel, wie dieser Algorithmus arbeitet.

**Beispiel 90** Berechnung des *rlgg* im *daughter*-Beispiel
Wir haben zwei positive Beispiele $e_1$ und $e_2$ in $\mathcal{E}^+$:

```
daughter(mary,ann).
daughter(eve,tom).
```

Bilden wir die Konjunktion der Prädikate mit dem Hintergrundwissen:

$C_{\mathcal{B}} =$

```
parent(ann,mary), parent(ann,tom), parent(tom,eve), parent(tom,ian), female(ann),
female(mary), female(eve)
```

Das *rlgg* kann folgendermaßen berechnet werden:

$$rlgg(e_1, e_2) = lgg((e_1 \leftarrow C_{\mathcal{B}}), (e_2 \leftarrow C_{\mathcal{B}}))$$

Dann wird die folgende Klausel erzeugt, die aus der Konjunktion der *lgg* aller möglichen Kombinationen von Hintergrundfakten besteht:

daughter($V_{mary,eve}, V_{ann,tom}$) :- parent(ann,mary), parent(ann,tom),
  parent(tom,eve), parent(tom,ian), female(ann), female(mary),
  female(eve), parent(ann,$V_{mary,tom}$), parent($V_{ann,tom}, V_{mary,eve}$),
  parent($V_{ann,tom}, V_{mary,ian}$), parent($V_{ann,tom}, V_{tom,eve}$),
  parent($V_{ann,tom}, V_{tom,ian}$), parent(tom,$V_{eve,ian}$), female($V_{ann,mary}$),
  female($V_{ann,eve}$), female($V_{mary,eve}$).

wobei $\forall x, \forall y V_{x,y}$ für $lgg(x, y)$ steht.

Wenn die für die Konjunktion überflüssigen Literale (Fakten; Literale, die außer Singleton-Variablen keine anderen Variablen enthalten) entfernt werden, bekommen wir die folgende Klausel:

daughter($V_{mary,eve}$, $V_{ann,tom}$) :- parent($V_{ann,tom}$, $V_{mary,eve}$),
female($V_{mary,eve}$).

Beziehungsweise mit berechneten *lgg*

```
daughter(X,Y) :- parent(Y,X), female(X).
```

wobei $X = V_{mary,eve}$ und $Y = V_{ann,tom}$.

Die andere oft verwendete, Methode ist die *inverse Resolution* (*ires*), die einfach die SLD-Ableitung invertiert (siehe Kapitel 2.2.1). Nehmen wir das Beispiel, das wir bei der Einführung der SLD-Ableitung angegeben haben und ergänzen dieses um Hintergrundwissen und Beispiele. Aber um die inverse Resolution berechnen zu können, müssen wir die Substitution auch invertieren können. Wie dies zu machen ist, haben Muggleton und Buntine ([MB88]) definiert.

**Definition 111** Sei $F$ eine wohldefinierte Formel (siehe Definition 3). Die *inverse Substitution* $\theta^{-1}$ der Substitution $\theta$ ist eine Funktion, die die Terme aus $F\theta$ in Variablen abbildet, so daß $F\theta\theta^{-1} = F$.

**Beispiel 91** Eine inverse Substitution für das Beispiel `daughter`
Sei $C_i$: `daughter(X,Y) :- female(X), parent(Y,X)` eine Klausel und $\theta = \{X/mary, Y/ann\}$ eine Substitution. Wenn wir diese Substitution auf die Klausel anwenden, dann gilt:

$C_i\theta$ = `daughter(mary,ann) :- female(mary), parent(ann,mary)`

Wenn wir jetzt die inverse Substitution $\theta^{-1} = \{mary/X, ann/Y\}$ anwenden, dann bekommen wir die ursprüngliche Klausel $C_i$.

Hier tritt die richtige Rückwärtsersetzung der Variablen auf. Im allgemeinen benötigt man zusätzlich den neuen Begriff *Platz*. Der sagt, an welcher Stelle welche Variable zurückersetzt werden muß.

**Beispiel 92** Inverse Resolution für das Beispiel `daughter`
Das Hintergrundwissen enthält zwei Fakten:
$b_1$: `female(mary).`
$b_2$: `parent(ann, mary).`

Die aktuelle Hypothese ist leer, $\mathcal{H} = \emptyset$, und wir nehmen an, daß wir das positive Beispiel $e_1$: `daughter(mary, ann)` haben. Um die inverse Resolution berechnen zu können, gehen wir wie folgt vor:

- Zuerst wird versucht, die Klausel $C_1$ zu finden, die mit $b_2$ das Beispiel $e_1$ ergibt. Diese Klausel wird dann zu $\mathcal{H}$ hinzugefügt. Dies geschieht in diesem Beispiel[1] mit Hilfe der inversen Substitution $\theta_2^{-1} = \{ann/Y\}$. Also ist die generierte Klausel: $C_1 = \text{ires}(b_2, e_1)$ = `daughter(mary, Y) :- parent(Y,mary)`. Die neue Hypothese ist jetzt $C_1$, wobei $\mathcal{H} \cup \{b_2\} \models e_1$.

[1]Man könnte genauso gut auch $\theta_1^{-1} = \{mary/X\}$ verwenden.

- Wir nehmen den ersten Fakt $b_1$ aus dem Hintergrundwissen und die im ersten Schritt ermittelte Hypothese $\mathcal{H} = \{C_1\} = \{$ `daughter (mary, Y) :- parent(Y,mary)` $\}$. Die nächste Klausel $C_2$ wird durch Anwendung der inversen Substitution $\theta_1^{-1} = \{mary/X\}$ ermittelt. Das Ergebnis ist dann $C_2 = \mathrm{ires}(b_1, C_1) =$ `daughter(X,Y) :- female(X), parent(Y,X)`. In der Hypothese wird die Klausel $C_1$ dann durch die allgemeinere Klauseln $C_2$ ersetzt, so daß $\mathcal{H} = \{$`daughter(X,Y) :- female(X), parent(Y,X)` $\}$.

Den Resolutionsbaum zeigt die Abbildung 9.3.

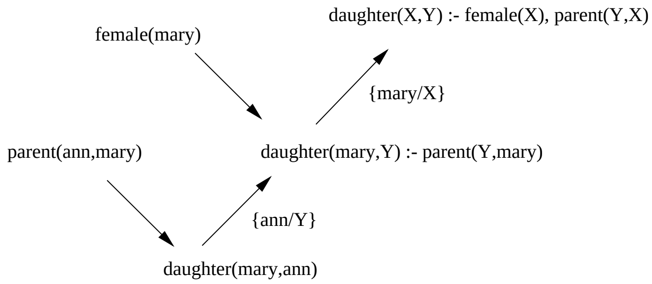



Abbildung 9.3: Eine inverse Resolution für das Familienbeispiel

Wir beginnen unten. Der Anfangspunkt ist `daughter(mary, ann)`. Dies kann mit dem Fakt aus dem Hintergrundwissen `parent(ann, mary)` mit der inversen Substitution $\theta_2^{-1} = \{ann/Y\}$ unifiziert werden. Das bedeutet, daß die Klausel `daughter(mary, Y) :- parent(Y,mary)` erzeugt wird, wobei $ann$ durch $Y$ ersetzt wird. Dann kann festgestellt werden, daß die erzeugte Klausel auch mit dem ersten Fakt aus dem Hintergrundwissen kombinierbar ist, wenn wir hier die inverse Substitution $\theta_1^{-1} = \{mary/X\}$ benutzen. Das Ergebnis ist eine neue Klausel `daughter(X,Y) :- female(X), parent(Y,X)`. An diesem Beispiel kann man gut erkennen, daß die inverse Resolution nichtdeterministisch ist.

Um das Problem des Nichtdeterminismus zu lösen, hat Muggleton *rllg* und inverse Resolution kombiniert. Der von ihm entwickelte Operator heißt *spezifischste inverse Resolution* (*most specific inverse resolution*) ($ires_{ms}$). Dieses Kapitel schließen wir mit einem kurzen Beispiel, das zeigt, wie dieser Operator funktioniert.

**Beispiel 93** Die spezifischste inverse Resolution für das `daughter`-Beispiel

- Wir haben, wie im Beispiel zuvor, das gleiche Hintergrundwissen und das gleiche Beispiel, aber jetzt nehmen wir zuerst den Fakt $b_2$: `parent(ann, mary)` und versuchen ihn mit dem Beispiel $e_1$: `daughter(mary,ann)` unter der leeren inversen Substitution zu unifizieren. Das Ergebnis $C_1 = ires_{ms}(b_2, e_1)$ ist die Klausel `daughter(mary,ann) :- parent(ann, mary)`.

- Im nächsten Schritt kann auf $C_1$ und $b_1$ wieder die leere inverse Substitution angewendet werden. Jetzt bekommen wir $C_2 = ires_{ms}(C_1, b_1) =$ `daughter(mary,ann) :- female(mary), parent(ann, mary)` als spezifischste inverse Resolution.

Dieser Operator hat sich als der erfolgreichste erwiesen und wird in verschiedenen Systemen, wie zum Beispiel GOLEM ([MF]), verwendet.

### Spezialisierungstechnik in ILP

Das Ziel bei dieser Strategie ist die Suche im Hypothesenraum durch Top-Down mit Hilfe von Spezialisierungsoperatoren, die meistens als *Verfeinerungsoperatoren* (*refinement*) bezeichnet werden. Diese Methoden fangen mit einer sehr allgemeinen Klausel (vom Benutzer definiert oder die leere Klausel) an und versuchen diese zu spezialisieren, wobei natürlich immer überprüft wird, ob negative Beispiele ableitbar sind. Weil wir im nächsten Kapitel ein System, das einen Spezialisierungsalgorithmus verwendet, ausführlich schildern werden, beschreiben wir hier nur kurz die wichtigsten Eigenschaften.

**Definition 112** Sei $\mathcal{L}$ die Sprache. Der *Spezialisierungsoperator* ist eine Funktion $\wp$, die $C_i \in \mathcal{L}$ auf $C_{i1}, \ldots, C_{in} \in \mathcal{L}$ abbildet, wobei $C_{ij}$ $(1 \leq j \leq n)$ eine Spezialisierung von $C_i$ ist:

$$\wp(C_i) \subseteq \{C_{ij} | C_{ij} \in \mathcal{L}, C_{ij} < C_i\}$$

Typische Verfeinerungsoperatoren sind die Durchführung einer Substitution oder das Hinzufügen eines Literals zum Rumpf der Klausel. Dieser Operator wurde als erster im *Model Inference System* (*MIS*) von Shapiro ([Sha83]) verwendet. Für das *daughter*-Beispiel kann *MIS* den Verfeinerungsgraph in Abbildung 9.4 generieren. In diesem Graph entsprechen die Knoten den Klauseln und die Kanten der Verfeinerung.

**Beispiel 94** Der zum *daughter*-Beispiel generierte Verfeinerungsgraph

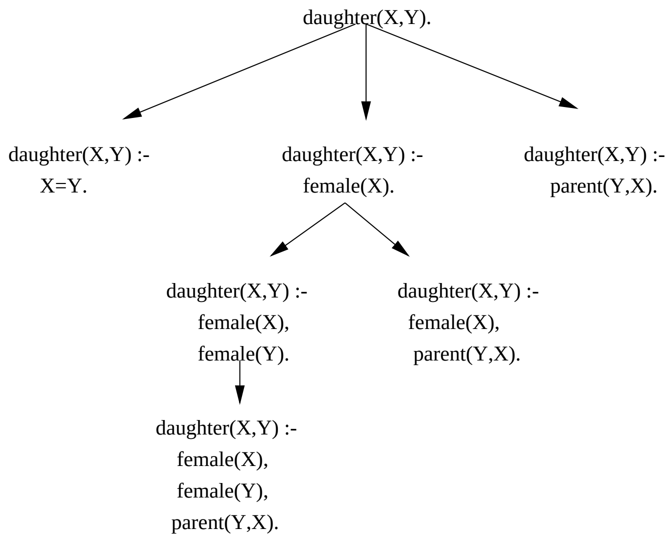



Abbildung 9.4: Ein Verfeinerungsgraph im *MIS*-System

Die Hauptschritte einer Spezialisierungsmethode zeigt der folgende Algorithmus.

**Algorithmus 45** Allgemeiner *Top-Down*-Algorithmus

**Eingabe**: $\mathcal{E}$ ist die Menge der positiven Beispiele
$\mathcal{B}$ ist das Hintergrundwissen
**Ausgabe**: Hypothese $\mathcal{H}$
**procedure** $top\_down\_ILP(\mathcal{E}, \mathcal{B}, \mathcal{H})$
**begin**
  $\mathcal{E}_{cur} := \mathcal{E}$; $\mathcal{H} := \emptyset$
  **repeat**
    choose T$\in \mathcal{E}_{cur}$
    $C$ := T$\leftarrow$.
    **repeat**
      find the best refinement $C_{best} \in \wp(C)$;
      $C := C_{best}$
    **until** $C$ is acceptable
    $\mathcal{H} := \mathcal{H} \cup C$;
    $\mathcal{E}_{cur} := \mathcal{E}_{cur}$ - positive examples covered by $C$
  **until** hypothesis $\mathcal{H}$ satisfies stopping criterion
  **return**($\mathcal{H}$)
**end**

Natürlich läßt die Beschreibung einige Fragen offen, wie zum Beispiel:

- Wie definiert man die beste Verfeinerung?
- Was bedeutet, daß eine Klausel $C_i$ annehmbar ist?
- Welche Haltekriterien können definiert werden?

Es gibt eine Reihe von anderen Methoden, die in *ILP*-Systemen verwendet werden. Wir werden hier nur zwei kurz erwähnen:

- Wie wir im vorherigen Kapitel beschrieben haben, kann *ILP* als Suchproblem im Hypothesenraum betrachtet werden, wobei eine lokale heuristische Suche durchgeführt wird. Hier wird das Problem des Verlassens lokaler Minima auftreten. In den *stochastischen ILP-Methoden* wird versucht, dieses Problem zu lösen:
  - Eine Hilfe könnte zum Beispiel die Integration von stochastischen Suchverfahren schaffen. Im System MILP ([Kov94]) werden die Hypothesen bewertet und die Übergänge mit Wahrscheinlichkeiten versehen. Die Wahrscheinlichkeit eines Übergangs wird dann proportional zur Qualität der Hypothese bestimmt.
  - Ein anderes System GA+Smart ([BG93]) versucht, die Vorgehensweise *der genetischen Programmierung* zu verwenden. Die einzelnen Hypothesen werden als Individuen betrachtet, an denen Mutation und Kreuzung durchgeführt werden können. Dann werden aus der so entstehenden Population die besten Individuen ausgewählt ((m,n): die $m$ Eltern überleben nicht). Hier tritt aber das Problem auf, wie die einzelnen Hypothesen als Individuen im Sinne von *genetischen Algorithmen* repräsentiert werden können. Die Lösung von Kovacic ist nicht so elegant:

Er nimmt eine Menge von Klauseln als Hypothesenraum und definiert die Hypothese als eine Teilmenge davon. Im GILP-System ([Var93]) wird dagegen versucht, die genetischen Algorithmen auf eine Menge von PROLOG-Klauseln anzuwenden.

- Das *ILP*-Problem wird auf eine aussagenlogische Form transformiert. Das hier auftretende Problem ist, daß die Aussagenlogik eine Teilmenge der Prädikatenlogik bildet, deswegen können nicht alle Probleme transformiert werden. Der Vorteil dieser Strategie ist, daß das Hintergrundwissen weiterhin verwendbar bleibt. Das System LINUS ([LD93a]) zum Beispiel übersetzt zuerst die Aufgabe in eine funktionsfreie typisierte Sprache, wo alle Variablen im Rumpf einer Klausel auch im Kopf auftreten müssen. Das Lernen wird durch Anwendung dieser Sprache stattfinden, und eigentlich ist LINUS ein Attributwerte-Lerner. Das Ergebnis wird dann zurücktransformiert.

### 9.1.3 Die verschiedenen ILP-Systeme

Im Lauf der Zeit sind mehrere *ILP*-Systeme entwickelt worden, die nach verschiedenen Eigenschaften charakterisiert werden können:

- *Inkrementell/empirisch:* Beschreibt die Art und Weise, wie die Menge von Beispielen dem System mitgegeben werden. Im Fall des empirischen Lernens werden beim Beginn des Lernprozesses alle Beispiele definiert und bis zum Ende nicht geändert. Im zweiten Fall, beim inkrementellen Lernen, werden die Beispiele dem System nach und nach eingegeben.

- *Interaktiv/ nicht interaktiv:* Interaktives Lernen braucht die Mitwirkung eines Anwenders, der Zwischenergebnisse verifizieren kann (zum Beispiel durch Beantworten von Fragen) oder der generalisierte Beispiele klassifizieren kann. Wenn keine Interaktion nötig ist, dann sprechen wir von nicht-interaktivem Lernen.

- *Einzelner/mehrfacher Prädikatlerner:* Nehmen wir an, daß $\mathcal{P}(F)$ das Prädikatsymbol in der Formel $F$ ist. Beim einzelnen Prädikatlernen sind die Beispiele nur für ein Prädikat gegeben: $\mathcal{P}(\mathcal{E})$ ist eine einelementige Menge. In dem Fall, wenn mehrere Prädikate gelernt werden sollen, ist $\mathcal{P}(\mathcal{E})$ nicht auf ein Element beschränkt. Dies bedeutet, daß mehrere Prädikate lernbar sind.

- *Erfindung von Prädikaten:* Es wird meistens vermutet, daß $\mathcal{P}(\mathcal{H}) \subseteq \mathcal{P}(\mathcal{B}) \wedge \mathcal{P}(\mathcal{E}^+)$ ist. Wenn $\mathcal{E}^+$ eine abzählbar unendliche Menge ist, stellt sich die Frage, ob $\mathcal{P}(\mathcal{B})$ für eine endliche Axiomatisierung von $\mathcal{E}^+$ ausreichend ist. Rice hat gezeigt, daß dieses Problem im allgemeinem unentscheidbar ist. Kleenes Ergebnis sagt aber, daß es immer möglich ist, die Menge $\mathcal{E}^+$ mit einer endlichen Vergrößerung von $\mathcal{P}(\mathcal{B})$ zu axiomatisieren.

- *Deklaratives Beeinflussen (bias):* Der Hypothesenraum kann ziemlich groß sein, deswegen ist es notwendig, die Aufmerksamkeit des Systems auf die wichtigsten Klauseln zu lenken. Die aus diesem Grund verwendete Vorgehensweise in einem Lernsystem heißt *Beeinflussen* (*bias*, [Mit80]). Bias können verschiedene sein:
  - Schemata [Wro88]
  - Parametrisierte Sprache [RB90]
  - Menge von Prädikaten [BGP89]

  - $ij$-Determinierung [MF]

- *Unvollständige Daten und Rauschen:* Wenn wir versuchen, Daten aus Anwendungen zu verwenden, können diese Daten oft unvollständig sein oder Rauschen enthalten. Es gibt Systeme, die erlauben, daß die Menge von Beispielen und Hintergrundwissen Fehler enthält. Diese verwenden Schätzmethoden, um kalkulieren zu können, wieviele Fehler als akzeptabel betrachtet werden.

Die entwickelten Systeme sind meistens in zwei Kategorien unterteilt:

- Der Lerner braucht schon am Anfang alle Beispiele, ist nicht interaktiv und kann nur ein einziges Prädikat lernen.

- Der Lerner arbeitet interaktiv und kann mehrere Prädikate lernen, aber er braucht eine Anfangshypothese.

In die erste Klasse fallen Systeme wie *FOIL* ([Qui88]), *mFOIL* ([Dze91]), *GOLEM* ([Mug92]), *LINUS* ([LD93a]) und *Markus* ([Gro92]). Die zweite Klasse enthält unter anderen *MIS* ([Sha83]), *MARVIN* ([SB86]), *CLIENT* ([Rae92]) und *CIGOL* ([MB88]).

Ein ausführlicher Vergleich der Systeme kann in [LD93a] nachgelesen werden.

### 9.1.4 Die Anwendungsgebiete von ILP

Ohne Anspruch auf Vollständigkeit zu erheben, möchten wir hier kurz einige bedeutende Anwendungsgebiete nennen. Sowohl Lavrač ([LD93a]) als auch de Raedt ([MR94]) erwähnen, daß die wichtigsten Gebiete die folgenden sind:

- Datenanalyse und Wissensextraktion

- Erzeugung korrekter Programme aus Beispielen und Programmteilen

Die Aufgabe der Datenanalyse ist die Erkennung von Informationen in Datenbanken. Hier tritt das Problem des Rauschens besonders stark auf. Die *ILP*-Systeme wie *FOIL* , *GOLEM* oder*LINUS* können dieses Problem beherrschen. Wenn wir Abhängigkeiten zwischen verschiedenen Relationen feststellen wollen, brauchen wir Lernsysteme wie *MOBAL* ([Mor]) oder *CLAUDIN* ([RB95]). Diese Systeme können mehrere Prädikate lernen. Die Wissensextraktion hat das Ziel, einen Satz, der bislang unbekannt war und als eine Formalisierung der Daten betrachtet werden kann, zu erkennen. In [KMLS92] findet sich eine Beschreibung, wie *ILP*-Techniken anwendbar sind, um etwas über die Wirkung eines neuen Medikaments vorhersagen zu können. Die Beispiele enthalten Medikamente und deren Wirkungen. Die Genauigkeit dieser Methode ist besser als die herkömmlicher statistischer Verfahren. Ein anderes Anwendungsbeispiel ist die Bestimmung der dreidimensionalen Form von Proteinen, wenn die Aminosäuren-Sequenzen bekannt sind. Es ist ein ziemlich schwieriges Problem, und die Präzision lag bislang zwischen fünfzig und sechzig Prozent. Mit Hilfe eines *ILP*-Verfahrens ([MKS92]) kann sie auf achtzig Prozent erhöht werden. Eine kurze Aufzählung der behandelten Probleme enthält den Entwurf integrierter Schaltungen ([HAPR94]), finite element mesh design ([DM92]), Telekommunikation ([SMAU94]), Robotik ([MR93]), Lernen von Endspielstrategien im Schach ([Moe92]), Lernen einer Grammatik ([WO92]) und Spracherkennung ([ACG+97]). Ein besonders wichtiges Anwendungsgebiet ist das Gebiet der medizinischen

Diagnose ([MPMC93], [BML89]). In diesem Bereich haben auch wir gearbeitet und ein System entwickelt, das *EKG*-Wellenformen nicht nur klassifizieren, sondern auch lernen kann. Damit haben wir einen *EKG*-Klassifikator entwickelt, der für alle Patienten eine individuelle Klassifizierung liefern kann. Nähere Informationen hierzu empfehlen wir in den Proceedings der jährlich stattfindenden *ILP*-Konferenzen nachzulesen.

Im Bereich Programmerzeugung erwähnt De Raedt ([MR94]) vier weitere Teilaufgaben:

- *Synthese von logischen Programmen:* Ein Programm kann durch Synthese (erzeuge Programm aus Spezifikation) oder durch Transformation (verbessere ein schon existierendes Programm) abgeleitet werden (siehe [Cle93]). Diese Aufgaben gehören zum Gebiet Software Engineering, wie es Bergadano und Gunetti beschreiben ([BG95]). Das eigentliche Ziel besteht darin, Programme nur aus Beispielen zu erzeugen. Obwohl mehrere Bestrebungen in diese Richtung gehen ([MF], [Rae92]), zeigen die Ergebnisse, daß dies zur Zeit nicht möglich ist (zum Beispiel muß die Beispielmenge ziemlich groß sein, und die Beispiele müssen gut ausgewählt werden).

- *Induktion der Eigenschaften von Programmen, Wissensbasen:* Das Ziel ist hier die Erkennung von Regelmäßigkeiten im Programm oder in der Wissensbasis. Bei Programmen kann dies dann als Spezifikation betrachtet werden. In der Datenbank bedeutet dies, Integritätsbedingungen (*integrity constraints*) zu finden.

- *Programmtesten und Debugging:* Die Arbeit von Shapiro, *MIS*, zeigt den Zusammenhang zwischen induktiver Inferenz und algorithmischem Debugging: Wenn ein Fehler gefunden und lokalisiert wird, kann inkrementelle induktive Inferenz benutzt werden, um diesen zu beheben. Eine ganz andere Aufgabe hat das Testen. Hier besteht das Ziel darin, herauszufinden, ob das Programm einen Fehler enthält. Ist dies der Fall, so kann versucht werden, mit dem Debugger diesen zu finden. Im Fall Testen wäre es sehr nützlich, eine Menge von Beispielen zu generieren, die das Verhalten des Programms simulieren. *ILP*-Techniken können benutzt werden, um eine Testmenge, die bestimmte Eigenschaften enthält, aus dem Programm zu generieren ([GBBT93] und [dRSB91]).

- *Verbesserung der Wissensbasen:* Die Aufgabe kann wie folgt definiert werden: Seien eine deduktive Datenbank $\mathcal{D}$, die eine Menge von Integritätsbedingungen $\mathcal{I}_\mathcal{C}$ erfüllt und eine Formel $f$, die von $\mathcal{D}$ nicht erfüllt ist, gegeben. Das Ziel ist, die Datenbank so zu erweitern, daß $f$ von der neuen Datenbank erfüllt wird, wobei alle Elemente aus $\mathcal{I}_\mathcal{C}$ auch wahr bleiben. Die hierzu verwendeten Methoden sind verschieden, aber meistens werden Fakten in die Datenbank integriert oder aus dieser entfernt. Wenn wir $\mathcal{B} \wedge \mathcal{H} \models \mathcal{D}$ und $\mathcal{E}^+ = \mathcal{I}_\mathcal{C}$ betrachten, dann ist das Problem dem inkrementellen Lernproblem ähnlich.

### 9.1.5 Beschränkungen in ILP

Um die Leistungsfähigkeit der *ILP* noch verbessern zu können, werden in den meisten *ILP*-Systemen *extra-logische Beschränkungen* eingeführt. Dies sind entweder statische Zusicherungen (eine graded preference surface wird dem Hypothesenraum zugeordnet) oder *Beschränkungen der Sprache* (*bias*, Verkleinerung des Hypothesenraums).

Die Zusicherungsfunktion bildet die Elemente des Hypothesenraums in die Teilmenge der reellen Zahlen ab. Sie ist eine totale Funktion, die in *ILP* als MDL-Theorie, minimale Beschreibungslängentheorie ( *minimal description length theory*, [Qui88], [MSB92]) entwickelt wurde. Džeroski hat in seiner Arbeit ([Dze91]) gezeigt, daß Bayes-statistische Analyse auch als Zusicherungsfunktion verwendet werden kann, die besonders nützlich bei Anwendungen ist, wo die Daten Rauschen enthalten.

Eine andere Beschränkung betrifft die Größe von $\mathcal{L}$. Die neuesten Ergebnisse des *PAC-Lernens* (Probably Approximately Correct Learning [NCdW96]) beschäftigen sich damit.

Muggleton gibt in seinem Artikel [Mug93] eine gute Zusammenfassung von realen Anwendungen, von *ILP*-Systemen und deren Beschränkungen. Er hat acht Anwendungsbeispiele ausgewählt: Das Lernen von Schachendspielstrategien [Moe92], die Vorhersage der Aktivität von Strukturen [KMLS92], numerische Anwendungen [MF], finite element mesh design [DM92], Lernen der Grammatik [WO92], Bestimmung der dreidimensionalen Form von Proteinen [MKS92], Lernen von qualitativen Modellen [BMV91] und die Diagnose von Fehlern [Fen92]. Von Muggleton verwendete Systeme sind: *FOIL*, *GOLEM*, *ITAU* ([Rou92], *CLIENT*, *RDT* ([KW92]), *SIERES* ([WO92]), *LINUS*, *FOCL* ([PBS92]). Die Einschränkungen, die verhindern, daß die Systeme auf wirkliche Probleme anwendbar werden, sind die folgenden:

- *Hintergrundwissen*: Das Hintergrundwissen kann in tabellarischer Form eingegeben werden (*FOIL*, *GOLEM*). Aber wenn die Menge von Grundinstanzen, wie zum Beispiel in den Anwendungen natürliche Sprache, Schachendspiel oder qualitative Modelle, ziemlich groß ist, dann ist diese Angabeform nicht anwendbar. Andere Systeme (*CLIENT* oder *LINUS*) sind in der Lage, mit Hilfe eines PROLOG-Interpreters Grundatome aus einer Spezifikation von Hintergrundprädikaten zu generieren.

- *Nicht numerische Daten*: Im Beispiel des Gitterdesigns von finiten Elementen müssen genaue numerische Werte ermittelt werden, wobei die Anzahl der Abschnitte eine Rolle spielt, so daß eine Kante eines CAD-Objekts zerlegt werden muß. Wenn die Beispielmenge nicht hinreichend groß ist, kann es schwierig sein, diese Werte genau zu ermitteln. Es wäre schon eine Annährung, wenn Intervalle definierbar wären, wo dieser Wert liegen könnte, aber auch diese Konstruktion können *ILP*-Systeme nicht behandeln. Die Lösung des in der Statistik als Regression bezeichneten genannten Problems muß in die *ILP*-Systeme eingebaut werden.

- *Determinierung*: Die Beschränkung der $ij$-Determinanten ist ein sehr verbreiteter Mechanismus, der aber nicht für alle Probleme eine gute Lösung liefert. Dies sind genau die Anwendungen, zum Beispiel Schachendspiel, wo *ILP* besser ist als herkömmliche auf Attributwerten beruhende Lerner. Ein Ergebnis von Kietz ([Kie93] ist, daß es keine allgemeine *PAC*-Lösung (*probably approximately c orrect learning*, *wahrscheinlich annähernd korrektes Lernen*) gibt, um ein nichtdeterministisches logisches Programm zu lernen.

- *Kurzsichtigkeit der Suche*: Das hier auftretende Problem zeigt *FOIL* oder *FOCL* und es ist das folgende: Wir haben für rekursive Prädikate teilweise entwickelte Lösungen, mit denen aber kein Unterschied zwischen positiven und negativen Beispielen festgestellt werden kann. Allein die ganze vollständige Klausel liefert die Entscheidung.

- *Effizienz des Lernens*: Eine wichtige Frage ist, wie effizient das System arbeitet. Die *ILP*-Systeme müssen eine Beispielmenge von realen Anwendungen behandeln, wo die Anzahl der Beispiele ziemlich groß sein kann, wie zum Beispiel bei der Bestimmung der dreidimensionalen Form von Proteinen.

Das Ergebnis zeigt die folgende Tabelle:

| Beschränkung | System | Problembereich |
|---|---|---|
| Hintergrundwissen | *FOIL, GOLEM* | Endspiel, Lernen der Grammatik, Lernen von qualitativen Modellen |
| nicht numerische Daten | *ITAU, FOIL, GOLEM, CLIENT, RDT, SIERES* | finite element mesh design, die Vorhersage der Struktur |
| Determinierung | *GOLEM, FOIL, LINUS* | Lernen von qualitativen Modellen, Endspiel, finite element mesh design |
| Kurzsichtigkeit der Suche | *FOIL, FOCL* | numerische Anwendungen |
| Effizienz des Lernens | *ITAU, CLIENT* | Endspiel, Diagnose von Fehlern, Bestimmung der dreidim. Form von Proteinen |

In diesem Kapitel haben wir kurz die *induktive logische Programmierung* eingeführt. Induktion (Lernen aus Beispielen) war früher ein Thema der Philosophie, aber heutzutage beschäftigt sich damit auch das maschinelle Lernen, ein Teilgebiet der künstlichen Intelligenz. Induktive logische Programmierung ist eine Kombination von maschinellem Lernen und logischer Programmierung. Dementsprechend ist die Aufgabe des Lernens, aus Beispielen im Rahmen der Logik die Beschreibung des Problems zu ermitteln.

Die Komponenten der Lernaufgabe sind eine endliche Menge von Klauseln ($\mathcal{B}$, Hintergrundwissen), eine Menge von Beispielen $\mathcal{E}$, die positive ($\mathcal{E}^+$) und negative Beispiele ($\mathcal{E}^-$) sein können, eine möglichst leere Anfangsmenge von Klauseln ($\mathcal{H}$, die Hypothese). Die Lösung des Problems ist eine Hypothese, aus der alle positiven und keine negativen Beispiele ableitbar sind. Es wird nach der richtigen Klausel im Hypothesenraum gesucht. Die dabei verwendeten zwei wichtigsten Methoden sind die Generalisierung (Bottom-Up-Methode) und Spezialisierung (Top-Down-Methode).

Im folgenden Kapitel wird das System *IMPUT*, das auf dem Spezialisierungsalgorithmus basiert, näher erklärt. Für Interessierte auf dem Gebiet *ILP* ist das Buch von Nienhuys-Cheng und de Wolf ([NCdW96]) zu empfehlen. Dieses Werk enthält eine ausführliche Beschreibung des aktuellen Stands des Gebietes. Weitere interessante Bücher sind [BGS94] und [Mug92].

## 9.2 Das Lernverfahren IMPUT

In diesem Kapitel wird ein interaktiver Lerner für mehrere Prädikate eines logischen Programms, das System *IMPUT* (*Interactive Multiple Predicate Learner Using Unfolding Transformation*) beschrieben. Das *IMPUT*-System besteht aus zwei Hauptteilen: Der Spezialisierungsalgorithmus stammt aus dem *SPECTRE*-System ([BIA94]), während der interaktive Debuggerteil das *IDTS*-System ist. So kann *IMPUT* als eine Erweiterung von *IDTS* betrachtet werden. Außerdem ist dieses System ein Teil des von uns entwickelten *SPECG*-Systems, das *EKG*-Wellenformen klassifizieren und lernen kann.

### 9.2.1 Vorbereitung

Wie wir erwähnt haben, verwendet das System einen *Spezialisierungsalgorithmus*. Dieser wird weiterentwickelt, um die vielfältigen rekursiven Prädikate gleichzeitig überarbeiten und *DCG*-Regeln (*Definite Clause Grammar*) behandeln zu können. Der Algorithmus *SPECTRE* spezialisiert logische Programme, indem er positive und negative Beispiele betrachtet und die Transformation *Unfolding* (siehe [TS84]) zusammen mit dem *Entfernen* von Klauseln verwendet. Die Auswahl des Literals, auf das Unfolding angewendet wird, wird von einer Kontrollregel, die dem System als Eingabe mitgegeben wird, gesteuert.

Versuchen wir, die Vorgehensweise formal zu beschreiben. Das Spezialisierungsproblem kann wie folgt formuliert werden:

**Definition 113** Das Spezialisierungsproblem

**Gegeben:** Ein definites Programm **P**
Zwei disjunkte Mengen von Grundatomen:
die positiven Beispiele $\mathcal{E}^+$ und die negativen Beispiele $\mathcal{E}^-$
**Suche:** Ein definites Programm **P'**, als Spezialisierung von **P**

Die Frage ist, wie wir **P'** bekommen können. Wie früher verwenden wir auch hier die folgende Schreibweise:

- $G_i :\leftarrow A_1, \ldots, A_k$ ist ein Ziel
- $C_j : A \leftarrow B_1, \ldots, B_q$ ist eine Klausel

Nehmen wir eine Berechnungsregel und nennen wir sie $R$. Wie wir schon früher gezeigt haben (siehe Kapitel 2.2.1), kann $G_{i+1}$ aus $G_i$ und $C_{i+1}$ mit Hilfe von $R$ abgeleitet werden, wenn

- $A_m$ das durch $R$ selektierte Atom ist,
- $A_m\theta_{i+1} = A\theta_{i+1}$ und
- $G_{i+1} :\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_{i+1}$

Wenn diese Voraussetzungen erfüllt sind, dann sagen wir, daß $G_{i+1}$ die Resolventen von $G_i$ und $C_{i+1}$ sind.

**Definition 114** Seien C: A $\leftarrow L_1, \ldots, L_m, \ldots, L_n$ eine Klausel im Programm **P** und **C** $= \{C_{m1}, \ldots, C_{mq}\}$ eine Menge von Klauseln in **P**, so daß der Kopf von allen $C_{mj} \in$ **C** $(1 \leq j \leq q)$ mit dem Literal $L_m$ durch den Unifikator $\theta_j$ unifizierbar ist. $L_m$ wird mit der Berechnungsregel $R$ ausgewählt. Das Programm **P'** nach der Unfolding-Transformation ist das folgende:

$$\mathcal{U}(\mathbf{P}) = \mathbf{P'} = \mathbf{P}\ \setminus \{C\} \cup \left( \bigcup_{j=1}^{q} A \leftarrow L_1, \ldots, L_{m-1}, body(C_{mj})\theta_j, L_{m+1}, \ldots, L_n \right)$$

Das bedeutet, daß das Literal $L_m$ durch ihre Resolventen ersetzt wird. $\mathcal{U}$ ist der Unfoldingoperator. Wenn wir diesen verwenden, dann bekommen wir eine Variation des Anfangsprogramms. Der Unfoldingoperator wird stets auf die Klausel angewendet, die als falsch identifiziert wurde. Wir suchen also nach der falschen Klausel, die folgendermaßen definiert werden kann:

**Definition 115** Seien **P** ein Programm (eine Menge von Klauseln), $e$ ein Grundatom und $\mathcal{M}$ ein gedachtes Modell des Programms. Dieses kann sich von der Interpretation $\mathcal{I}$ unterscheiden. $e \in \mathcal{E}^-$ sei ableitbar aus **P**. Dann ist $C$ die falsche Klausel in **P**, wenn folgendes gilt:

- Eine Instanz von $C$ kommt in der Ableitung von $e$ vor.
- Die Instanz von $C$ ist die Klausel $A \leftarrow L_1, \ldots, L_m, \ldots, L_n$ und $(\forall L_i)\ (L_i \in \mathcal{M})(1 \leq i \leq n)$, aber $A \notin \mathcal{M}$.

Wie gesagt, das *IMPUT*-System integriert den Algorithmus *SPECTRE* (Specialization by Transformation and Elimination) mit dem Debugger *IDTS*. Letzteren haben wir schon in vorherigen Kapiteln dieser Arbeit detailliert erklärt. Wir beschreiben jetzt den von Boström [BIA94] entwickelten *SPECTRE*-Algorithmus und demonstrieren durch ein Beispiel, wie dieser funktioniert.

### 9.2.2 Der Algorithmus SPECTRE

Der Algorithmus *SPECTRE spezialisiert* ein logisches Programm. Dabei verwendet er positive und negative Beispiele und benutzt die *Unfolding*-Transformation. Informell kann er folgendermaßen erklärt werden: Solange es eine Klausel im Programm gibt, die außer positiven Beispielen auch negative abdeckt, wird diese Klausel der Unfolding-Transformation unterzogen. Wenn die Klausel nur in der Ableitung von negativen Beispielen eine Rolle spielt, kann sie einfach aus dem Programm entfernt werden. Bei der Unfolding-Transformation wird die Berechnungsregel $R$ verwendet, um das Literal, das die Grundlage der Transformation bildet, auswählen zu können. Daraus ist schon erkennbar, daß die Wahl von $R$ eine große Bedeutung hat. Die Experimente von Boström (siehe [BIA94]) zeigen auch, daß man die Anzahl der Unfolding-Transformationen so niedrig wie möglich halten muß, weil sich die Anzahl der Klauseln durch die Anwendung dieser Transformation vergrößert.

Boström und Idestam-Almquist haben in ihrem Artikel [BIA94] gezeigt, daß die Anzahl der Klauseln, die notwendig sind, um zwischen positiven und negativen Beispielen zu unterscheiden, minimiert werden kann. Sie haben die in IDT3 ([Qui88]) verwendete *minimale Rest-Unreinheit* (*minimal residual impurity*) benutzt:

$$-c\sum_{i=1}^{n} cov(C_i, \mathcal{E}^+) log_2\left(\frac{cov(C_i, \mathcal{E}^+)}{cov(C_i, \mathcal{E}^+ \cup \mathcal{E}^-)}\right) + cov(C_i, \mathcal{E}^-) log_2\left(\frac{cov(C_i, \mathcal{E}^-)}{cov(C_i, \mathcal{E}^+ \cup \mathcal{E}^-)}\right)$$

wobei $C_1, \ldots, C_n$ die Resolventen über einem Literal aus der Klausel $C$ sind und $cov(C_i, \mathcal{E})$ die Anzahl von Elementen aus $\mathcal{E}$, die mit $C_i$ ableitbar sind, $c$ bezeichnet eine Konstante $1/cov(C_i, \mathcal{E}^+ \cup \mathcal{E}^-)$.
Der Algorithmus *SPECTRE* ist dann der folgende:

**Algorithmus 46** Der Algorithmus SPECTRE

**Eingabe:** Ein definites Programm **P**,
Zwei Mengen $\mathcal{E}^+$ und $\mathcal{E}^-$ von Grundatomen,
die Menge von positiven und negativen Beispielen,
Eine Berechnungsregel $R$
**Ausgabe:** Eine Spezialisierung **P'** des Programms **P**,
wobei alle $e \in \mathcal{E}^+$, aber keine $e \in \mathcal{E}^-$ ableitbar sind.
**procedure** SPECTRE(**P**, $\mathcal{E}^+$, $\mathcal{E}^-$, $R$, **P'**)
**begin**
  **while** $(\exists C \in \mathbf{P})(C$ covers $e \in \mathcal{E}^-)$
    **if** $\neg(\exists e \in \mathcal{E}^+)$ ($C$ covers $e$) **then** remove $C$ from **P'**
    **else** select a literal with $R$ make unfolding
    **endif**
  **endwhile**
  **return**(**P'**)
**end**

### 9.2.3 Ein Beispiel zu SPECTRE

Um die Vorgehenweise von *SPECTRE* und später *IMPUT* demonstrieren zu können, nehmen wir ein einfaches Beispiel *rectangle*. Dieses wurde ursprünglich in [AGB95] eingeführt. Das Programm selbst ist ganz einfach und nicht sehr realistisch, aber es eignet sich gut, um die Algorithmen zu demonstrieren. In den Artikeln [BIA94] und [AGB95] kann man die Ergebnisse des Lernprozesses aus der *UCI Maschinellen Lerndatenbank* nachlesen.

Das Programm versucht, horizontal liegende Rechtecke[2] mit vertikalen linken und rechten Seiten und mit horizontalen anderen Seiten zu erkennen. Die vier Parameter des Prädikates `rectangle` repräsentieren die Steigungen der Liniensegmente (links, unten, rechts, oben), wobei das Atom `inf` für eine unendliche Steigung steht. Nehmen wir an, daß wir das folgende Anfangsprogramm haben:

**Beispiel 95** Das Anfangsprogramm für `rectangle`

```
rectangle(X, Y, Z, U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- segment(X).
top(X)       :- segment(X).
leftside(X)  :- segment(X).
rightside(X) :- segment(X).
segment(X)   :- horiz(X).
segment(X)   :- vert(X).
horiz(line(X)) :- Z is abs(X), Z < 0.01.
vert(line(X))  :- Z is abs(X), Z > 100.
```

Dieses Programm kann auch Rechtecke wie zum Beispiel:

```
rectangle (line (0.005),line(0.0),line(0.0),line(0.005))
```

[2] Es geht hier nicht um mathematisch exakte Rechtecke, sondern um eine eher umgangssprachlich geprägte Vorstellung von Rechtecken.

akzeptieren, die vier horizontale Seiten haben, also falsch sind. Um das System benutzen zu können, sind positive und negative Beispiele einzugeben. Diese sind die folgenden:

**Beispiel 96** Die positiven und negativen Beispiele für `rectangle`
$\mathcal{E}^+ =$

```
{rectangle(line(150), line(0.005), line(inf), line(0.0)),
rectangle(line(-160), line(0.0), line(inf), line(0.005)),
rectangle(line(150), line(-0.004), line(150), line(-0.004)),
rectangle(line(inf), line(0.005), line(150), line(0.0))},
```

$\mathcal{E}^- =$

```
{rectangle(line(150), line(-160), line(0.005), line(-160)),
rectangle(line(-170), line(160), line(0.0), line(180)),
rectangle(line(200), line(0.001), line(300), line(400)),
rectangle(line(-160), line(0.005), line(-160), line(-160)),
rectangle(line(0.005), line(0.0), line(0.0), line(0.005)),
rectangle(line(0.005), line(-160), line(0.0), line(0.0)),
rectangle(line(150), line(0.0), line(0.0), line(-0.004))}.
```

Eine vorstellbare korrekte Lösung wäre die folgende:

**Beispiel 97** Ein korrektes Programm für `rectangle`

```
rectangle(X, Y, Z, U) :- leftside(x), base(Y), rightside(Z), top(U).
base(X)         :- horiz(X).
top(X)          :- horiz(X).
leftside(X)     :- vert(X).
rightside(X)    :- vert(X).
segment(X)      :- horiz(X).
segment(X)      :- vert(X).
horiz(line(X)) :- Z is abs(X) , Z < 0.01.
vert(line(X))  :- Z is abs(X), Z > 100.
```

*SPECTRE* arbeitet wie folgt: Das System kann mit der Eingabe des Befehls `start` starten. Dann erwartet es als Eingabe eine Theoriedatei, die das zu spezialisierende Programm enthält, und eine Datei mit den Beispielen, in der die positiven mit dem Prädikat `pos` und die negativen mit `neg` gekennzeichnet sind. Beide werden nach der Eingabe aufgelistet.

**Beispiel 98** Eingabe der Dateinamen für das Spectre-System

```
Welcome to SPECTRE 0.1 (with trace)
Enter file with theory:rect2th.

The following theory is to be specialized:

rectangle(A, B, C, D) :-
        leftside(A),
        base(B),
        rightside(C),
        top(D).
...
```

```
Enter file with examples:rect2ex.

The following are the positive examples:
rectangle(line(150), line(0.005), line(inf), line(0.0)).
...

The following are the negative examples:
rectangle(line(150), line(-160), line(0.005), line(-160)).
```

Das System wählt eine Klausel aus, die ein negatives Beispiel abdeckt, und überprüft, ob diese Klausel auch in der Ableitung von positiven Beispielen verwendet wird. Wenn das der Fall ist, dann kann sie nicht entfernt werden, und es muß eine Unfolding-Transformation durchgeführt werden.

**Beispiel 99** Spezialisierungsprozeß in *SPECTRE*

```
The following clause covers the negative example
rectangle(line(150),line(-160),line(0.005),line(-160)):
rectangle(A, B, C, D) :-
        leftside(A),
        base(B),
        rightside(C),
        top(D).
It also covers the positive example
rectangle(line(150),line(0.005),line(inf),line(0.0)) and cannot be removed.
```

Dann berechnet das System die Rest-Unreinheit und wählt die Klausel aus, bei der dieser Wert minimal ist. In diesem Fall sind alle Werte gleich. Deswegen nimmt das System die erste Möglichkeit.

**Beispiel 100** Die Wahl zwischen den verschiedenen Klauseln auf der Basis der minimalen Rest-Unreinheit

```
The following unfoldings are possible:

Alternative 1 with impurity measure 10.4:
rectangle(A, B, C, D) :-
        segment(A),
        base(B),
        rightside(C),
        top(D).

Alternative 2 with impurity measure 10.4:
rectangle(A, B, C, D) :-
        leftside(A),
        segment(B),
        rightside(C),
        top(D).

Alternative 3 with impurity measure 10.4:
rectangle(A, B, C, D) :-
        leftside(A),
        base(B),
```

```
        segment(C),
        top(D).

Alternative 4 with impurity measure 10.4:
rectangle(A, B, C, D) :-
        leftside(A),
        base(B),
        rightside(C),
        segment(D).


The selected alternative is:
rectangle(A, B, C, D) :-
        segment(A),
        base(B),
        rightside(C),
        top(D).
```

Der Prozeß geht weiter mit der Überprüfung der Beispiele, Berechnung der minimalen Rest-Unreinheit und Unfolding-Transformationen.

**Beispiel 101** Der zweite Unfolding-Schritt in *SPECTRE*

```
The following clause covers the negative example
rectangle(line(150),line(-160),line(0.005),line(-160)):
rectangle(A, B, C, D) :-
        segment(A),
        base(B),
        rightside(C),
        top(D).
It also covers the positive example
rectangle(line(150),line(0.005),line(inf),line(0.0)) and cannot be removed.

The following unfoldings are possible:

Alternative 1 with impurity measure 8.92:
rectangle(A, B, C, D) :-
        horiz(A),
        base(B),
        rightside(C),
        top(D).
rectangle(A, B, C, D) :-
        vert(A),
        base(B),
        rightside(C),
        top(D).

Alternative 2 with impurity measure 10.4:
rectangle(A, B, C, D) :-
        segment(A),
        segment(B),
        rightside(C),
```

```
        top(D).

Alternative 3 with impurity measure 10.4:
rectangle(A, B, C, D) :-
        segment(A),
        base(B),
        segment(C),
        top(D).

Alternative 4 with impurity measure 10.4:
rectangle(A, B, C, D) :-
        segment(A),
        base(B),
        rightside(C),
        segment(D).


The selected alternative is:
rectangle(A, B, C, D) :-
        horiz(A),
        base(B),
        rightside(C),
        top(D).
rectangle(A, B, C, D) :-
        vert(A),
        base(B),
        rightside(C),
        top(D).
```

*SPECTRE* arbeitet weiter. Am Ende ist das vom System gefundene Programm, das alle positiven und keine negativen Beispiele ableiten kann, das folgende:

**Beispiel 102** Die von *SPECTRE* gefundene Lösung

```
rectangle(line(150), line(0.005, A, B)    :- segment(A), horiz(B).
rectangle(line(150), line(-0.004), A, B)  :- rightside(A), top(B).
rectangle(line(-160), line(0.0), A, B)    :- rightside(A), top(B).
rectangle(line(-160), line(-0.004), A, B) :- rightside(A), top(B).
rectangle(line(inf), A, B, C)  :- rightside(B), top(C).
base(A)      :- segment(A).
top(A) : - segment(A).
leftside(A)  :- segment(A).
rightside(A) :- segment(A).
segment(A)   :- horiz(A).
segment(A)   :- vert(A).
horiz(line(X)) :- Z is abs(X), Z < 0.01.
vert(line(X))  :- Z is abs(X), Z > 100.
```

Diese Ausgabe zeigt einen der Nachteile von *SPECTRE*, nämlich daß es immer an der Klausel, die das Ziel ist, Unfolding durchführt, obwohl es meistens sinnvoller wäre, andere Klauseln zu verbessern. In solchen Fällen kann *IMPUT* behilflich sein, weil mit dem eingebauten *IDTS*-Debugger die Klausel, die für die Ableitung von falschen Beispielen verantwortlich ist, gefunden werden kann. Im folgenden beschreiben wir das *IMPUT*-System selbst und zeigen, wie das gleiche Beispiel mit Hilfe von *IMPUT* gelernt werden kann. Außerdem hatte der Original-*SPECTRE*-Algorithmus einige Schwierigkeiten, das korrekte Programm zu lernen, wenn das Zielprädikat rekursiv war oder wenn die gleiche Sequenz von Eingabeklauseln in der Ableitung sowohl von positiven als auch von negativen Beispielen benutzt wurde. Eine erste Verbesserung wurde von Boström selbst im Artikel [Bos95] angegeben.

### 9.2.4 Das IMPUT-Verfahren

*IMPUT* ist eine Integration von *IDTS* und *SPECTRE*. Die Idee in *IMPUT* ist, daß die Identifizierung der Klausel, die als nächste mit der Unfoldingtransformation zerlegt wird, eine entscheidende Bedeutung für den Erfolg des Spezialisierungsprozesses hat. Der Spezialisierungsschritt wird von den negativen Beispielen aufgerufen, die das Zielprädikat abdecken. Dabei wird angenommen, daß, wenn ein negatives Beispiel aus dem aktuellen Programm ableitbar ist, es eine Klausel gibt, die für das inkorrekte Verhalten verantwortlich ist. Um diese *falsche Klausel* zu finden, benutzt *IMPUT* als Teilmodul *IDTS*. Die vom Debugger gefundene Klausel wird im nächsten Schritt dem Unfoldingprozeß unterzogen.

Der grundlegende Algorithmus des *IMPUT*-Systems ist der folgende, eine detaillierte Beschreibung kann in [AGB96] nachgelesen werden.

**Algorithmus 47** Das *IMPUT*-Verfahren

**Eingabe:** Ein Anfangsprogramm **P**
Zwei Mengen $\mathcal{E}^+$ und $\mathcal{E}^-$ von Grundatomen,
die Menge von positiven und negativen Beispielen
Eine Berechnungsregel $R$
Hintergrundwissen B
**Ausgabe:** Eine Spezialisierung **P'** des Programms **P**,
wobei alle $e \in \mathcal{E}^+$, aber keine $e \in \mathcal{E}^-$ ableitbar sind.
Dabei wird eine Folge von Programmen $\mathbf{P'}^{(0)} = \mathbf{P}$, $\mathbf{P'}^{(1)}$, ..., $\mathbf{P'}^{(n)}$ berechnet,
wo $\mathbf{P'}^{(i+1)} = \tilde{\mathcal{U}}(\mathbf{P'}^{(i)})$ $(0 \leq i < n)$,
$\tilde{\mathcal{U}}$ bedeutet die Unfolding-Transformation und das Löschen von Klauseln
**procedure** IMPUT(**P**, $\mathcal{E}^+$, $\mathcal{E}^-$, $R$, **B**, **P'**)
**begin**
  $i := 0$
  **if P** does not terminate on all $e \in \mathcal{E}^+$ **then**
    **write**(Initial program should cover all positive examples.) **halt**
  **endif**
  **while** $(\exists e \in \mathcal{E}^-)($ $\mathbf{P'}^{(i)}$ does not fail on $e)$ **do**
    Find a buggy clause $C \in \mathbf{P'}^{(i)}$ using the *IDTS* debugger;
    Perform $\tilde{\mathcal{U}}(c)$ using $R$;
    **let** $\mathbf{C} = \{C_1, \ldots, C_q\}$ be the resolvents of $C$;
    Remove from **C** all those clauses that do not occur in refutations $e \in \mathcal{E}^+$;
    **let** $\mathbf{P'}^{(i+1)} = \mathbf{P'}^{(i)} \setminus \ \{C\} \cup \mathbf{C}$;
    $i := i + 1$
  **endwhile**
  **P'**= $\mathbf{P'}^{(i)}$
  **return**(**P'**)
**end**

### 9.2.5 Ein Beispiel zu IMPUT

In diesem Abschnitt beschreiben wir, wie das Beispiel `rectangle` (siehe Beispiel 95) mit *IMPUT* gelernt werden kann. Wir können auch Hintergrundwissen definieren. Bei diesem Beispiel nehmen wir an, daß die Klauseln, die die Prädikate `horiz` und `vert` definieren, korrekt sind. Sie können also als Hintergrundklauseln ausgewählt werden.

**Beispiel 103** Testbeschreibung für das Prädikat `line`

| | |
|---|---|
| **Category** Measure of the slope | |
| **Choice** small | (abs(slope) $<$ 0.01) |
| **Choice** medium | (0.01 $<$ abs(slope) $<$ 100) |
| **Choice** big | (abs(slope) $>$ 100 ) |
| **Choice** zero | (slope = 0) |
| **Choice** inf | (slope = inf) |
| **Category** Sign of the slope | |
| **Choice** negative | **if** not zero and not inf and slope $<$ 0 |

**Choice** positive **else**

Dies ist eine Spezifikation für das PROLOG-Prädikat `line`, wobei `line` als Prädikat das Linien-Segment mit der Neigung `X` bedeutet. Wenn wir `line(inf)` eingeben, dann verstehen wir darunter eine senkrechte Linie mit unendlich großer Neigung. Die Testbeschreibung enthält zwei Kategorien: `Measure of the slope` und `Sign of the slope`. Die erste Kategorie enthält fünf Wahlmöglichkeiten: `small`, `medium`, `big`, `zero` und `inf`. Zu der zweiten haben wir zwei definiert: `negative` und `positive`. Wenn wir auch die vorgegebenen Eigenschaften (`zero` und `infinite`) verwenden, dann werden die folgenden Testrahmen definiert:
*(small, negative), (small, positive), (medium, negative), (medium, positive), (big, negative), (big, positive),(zero, positive), (inf, positive)*

Durch das System wird die Testdatenbank generiert, und nach bestimmten Tests (nähere Erklärung kann im Kapitel 4.2 gefunden werden) ergibt sich die Datenbank:

**Beispiel 104** Die generierte Testdatenbank nach dem Testen

```
test(segment,1,[small,negative],[line(-0.001)],true,correct).
test(segment,1,[small,positive],[line(0.002)],true,correct).
test(segment,1,[medium,negative],-,-,undefined).
test(segment,1,[medium,positive],-,-,undefined).
test(segment,1,[big,negative],-,-,undefined).
test(segment,1,[big,positive],-,-,undefined).
test(segment,1,[zero,positive],[line(0)],true,correct).
test(segment,1,[inf,positive],-,-,undefined).
```

Zum Beispiel der PROLOG-Fakt:

```
test(segment,1,[small,negative],[line(-0.001)], true, correct)
```

entspricht dem Testrahmen *[small,negative]*. Dieser Testrahmen wurde mit dem Testfall `line(-0.001)` getestet. Für diesen ist das Prädikat korrekt, und die vom Anwender gegebene Bewertung ist auch korrekt. Außerdem sind schon die Testrahmen für das horizontale Segment mit positiver, negativer und zero Neigung getestet worden. In allen Fällen sind die von dem Benutzer angegebenen Bewertungen *correct*. Die anderen, noch nicht getesteten Testrahmen haben als sechstes Argument *undefined*.

Nehmen wir die gleichen Daten, die wir im Beispiel 96 benutzt haben und beginnen wir mit dem *IMPUT*-Verfahren auf dem Programm 95:

- i = 0 : Alle positiven Beispiele sind ableitbar, aber auch das erste Beispiel aus der Menge $\mathcal{E}^-$: `rectangle(line(150), line(-160), line(0.005), line(-160))` ist vom Programm abgedeckt. Durch Aufruf des *IDTS*-Debuggers werden die folgenden Fragen gestellt:

  **Beispiel 105** Die gestellten Fragen:

  ```
  Is it segment(line(150)) correct? (y/n) y
  Is it leftside(line(150)) correct? (y/n) y
  Is it segment(line(-160)) correct? (y/n) y
  Is it base(line(-160)) correct? (y/n) n
  ```

wobei die Fakten aus dem Hintergrundwissen und diejenigen, die eine korrekte Bewertung in der Datenbank haben, nicht gefragt werden. Der Körper der Klausel `base(X) :- segment(X)`, die für den Fehler verantwortlich ist, enthält nur ein Literal. Deswegen findet keine Auswahl des Literals für die Unfolding-Transformation statt. Die Resolventen der Klausel sind:

$$\mathbf{C} = \{ \texttt{base(X) :- horiz(X).}, \texttt{base(X) :- vert(X).} \}$$

Die Klausel `base(X) :- vert(X)` kann aus dem Programm entfernt werden, weil keine Ableitung von positiven Beispielen diese enthält. Also ist die erste Annäherung $\mathbf{P}'^{(1)}$ die folgende:

**Beispiel 106** Die erste Spezialisierung des Programms $\mathbf{P} = \mathbf{P}'^{(1)}$

```
rectangle(X,Y,Z,U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- horiz(X).
top(X)       :- segment(X).
leftside(X)  :- segment(X).
rightside(X) :- segment(X).
segment(X) :- horiz(X).
segment(X) :- vert(X)
```

Dabei wurden die folgenden Änderungen in der Testdatenbank durchgeführt:

**Beispiel 107** Änderungen in der Testdatenbank

```
test(segment,1,[big,negative],[line(-160)],true,correct).
test(segment,1,[big,positive],[line(150)],true,correct).
```

Die hier nicht aufgelisteten Fakten bleiben unverändert. Die Bewertung der Testrahmen `[big,negative]` und `[big,positive]` werden in den folgenden Schritten des Verfahrens schon benutzt, um die dem Anwender gestellten Fragen zu verringern.

- i = 1: Das oben abgeleitete Programm $\mathbf{P}'^{(1)}$ schlägt fehl auf dem ersten, zweiten und sechsten negativen Beispiel. Aber das dritte Beispiel:

```
rectangle(line(200), line(0.001), line(300), line(400))
```

ist immer noch ableitbar. Wieder wird *IDTS* aufgerufen, das die folgenden Fragen stellt:

**Beispiel 108** Die von *IDTS* gestellten Fragen

```
Is it base(line(0.001)) correct? (y/n) y
Is it rightside(line(300)) correct? (y/n) y
Is it top(line(400)) correct? (y/n) n
```

Die falsche Klausel ist: `top(X) :- segment(X)`. Diese Klausel enthält wieder nur ein Literal auf der linken Seite, und das bildet die Grundlage des Unfoldings. Die Resolventen sind:

$$\mathbf{C} = \{\texttt{top(X)} : -\texttt{horiz(X)}, \texttt{top(X)} : -\texttt{vert(X)}\}$$

Die zweite Klausel spielt keine Rolle in der Ableitung von positiven Beispielen, deswegen kann sie weggelassen werden. Das Ergebnis des zweiten Schritts ist das folgende:

**Beispiel 109** Die zweite Spezialisierung des Programms $\mathbf{P}'^{(2)}$

```
rectangle(X,Y,Z,U) :- leftside(X), base(Y), rightside(Z), top(U).
base(X)      :- horiz(X).
top(X)       :- horiz(X).
leftside(X)  :- segment(X).
rightside(X) :- segment(X).
segment(X) :- horiz(X).
segment(X) :- vert(X).
```

- i = 2: Das nächste negative Beispiel:

  ```
  rectangle(line(0.005), line(0.0), line(0.0), line(0.005))
  ```

  Die dafür verantwortliche Klausel ist `leftside(X) :- segment(X)`. Wird diese dem Unfolding unterzogen, so bekommt man $\mathbf{P}'^{(3)}$.

- i = 3: Das einzige übriggebliebene negative Beispiel, das immer noch ableitbar ist, ist:

  ```
  rectangle(line(150), line(0.0), line(0.0), line(-0.004)).
  ```

  Mit dem Debugger kann festgestellt werden, daß die Klausel `rightside(X) :- segment(X)` die nächste und letzte ist, die zerlegt werden muß. Nach der Durchführung bekommen wir das Programm $\mathbf{P}'^{(4)}$, das keine negativen, aber alle positiven Beispiele ableiten kann und das gleiche ist, das wir erwartet haben (siehe Beispiel 97).

## 9.3 Das erweiterte SPECG-System

In diesem Kapitel beschreiben wir die Erweiterung des *SPECG*-Systems. Das sind im einzelnen:

- *ein interaktives Vorverarbeitungsmodul für EKG*
- *das schon eingeführte Lernsystem* IMPUT
- *eine graphische Benutzeroberfläche SPECG-GUI*

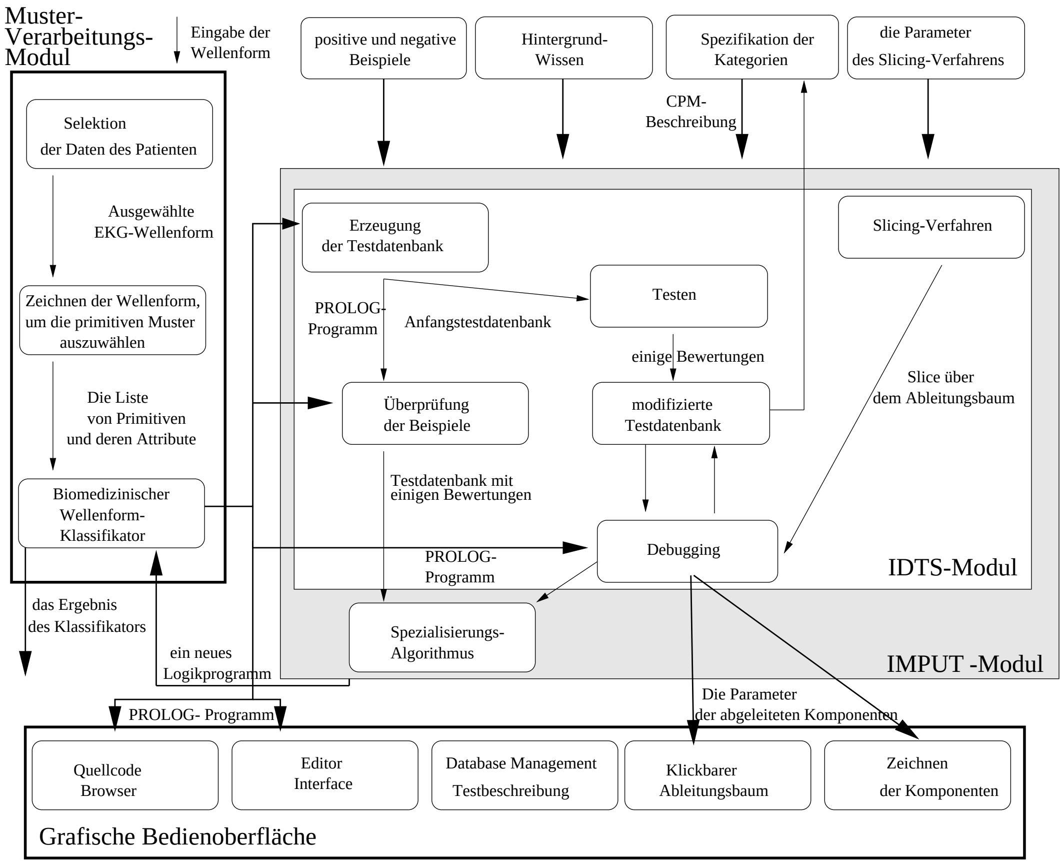



Abbildung 9.5: Das erweiterte *SPECG*-System

Die Gesamtstruktur des Systems zeigt die Abbildung 9.5.

Wie wir schon in Kapitel 8.4 erwähnt haben, ist mit Hilfe dieses Systems nicht nur die genaue Stelle des Fehlers erkennbar, sondern in vielen Fällen kann aus einem fehlerhaften Programm ein korrektes generiert werden. Wenn wir die *IMPUT*-Methode anwenden, schlägt *SPECG* dem Anwender eine korrekte Lösung vor und kann später die falsche Klausel, die bei dem Debuggingprozeß erkannt wurde, durch ihre Resolventen ersetzen.

### 9.3.1 Das interaktive Vorverarbeitungsmodul

Als Anfangspunkt dient eine *EKG*-Aufnahme, wie sie zum Beispiel Abbildung 9.6 zeigt.

Diese Aufnahme wird auf einen Rechner transportiert und dekodiert. Es folgt der Vorverarbeitungteil. Die Aufgabe dieses Teils ist die Erzeugung einer Eingabe für das Lernsystem. Diese Eingabe muß die Primitive mit bestimmten, aus *EKG*-Wellen ableitbaren, numerischen Werten enthalten. Dieses Modul arbeitet interaktiv und braucht eine Person, die die Primitive bestimmen kann. Die Erzeugung der numerischen Werte erfolgt dann automatisch. Die Vorgehensweise ist die folgende: Der erste Schritt besteht darin, daß die Person, die dieses Programm bedient, aus den *EKG*-Aufnahmen eine auswählt, die er/sie verarbeiten möchte. Dies geschieht mit einem Fenster, in dem alle Dateien aufgelistet werden (siehe Abbildung 9.7).

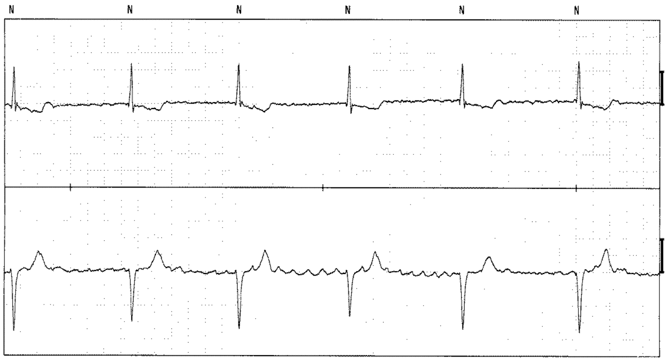



Abbildung 9.6: Die von einem *EKG*-Gerät erzeugten Elektrokardiogrammwellenformen

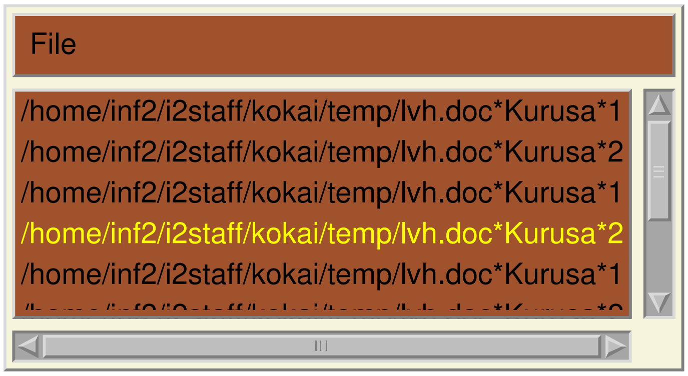



Abbildung 9.7: Die transportierten *EKG*-Daten

Im nächsten Schritt erscheint ein neues Fenster, und der Arzt sieht die ganze Wellenform. Seine Aufgabe ist jetzt, mit Hilfe der Maus die Primitive (Spitze, Parabel, Linie) auszuwählen. Dazu berechnet das System die numerischen Parameter der Primitive und zeichnet sie auf den Bildschirm. Wenn die beiden Linien zur Deckung kommen, dann ist die Selektion richtig. Natürlich kann diese Funktion ausgeschaltet werden. Wenn die Wellenform komplizierter ist, kann diese mit der *Zoom*-Funktion vergrößert werden. Ist der Selektierungsprozeß beendet, speichert das Programm alle erzeugten Parameter in einer vom Anwender angegebenen Datei ab.

### 9.3.2 Die graphische Benutzeroberfläche für SPECG: SPECG-GUI

Das Thema dieses Kapitels ist die kurze Beschreibung der graphische Benutzeroberfläche *SPECG-GUI*. Wie früher wurde NGUIDE ([Nil97]) benutzt, um *SPECG-GUI* zu programmieren. Das vollständige System mit allen Bäumen, Editoren und Browsern zeigt die Abbildung 9.9. Eine detailierte Beschreibung kann in [NK97] gefunden werden.
Die GUI unterstützt die folgenden Funktionen:

- *Klickbare Ableitungsbäume*: In jedem Schritt des Debuggingprozesses kann die GUI einen Knoten im Ableitungsbaum zeichnen, dessen beabsichtigtes Verhalten der Debugger

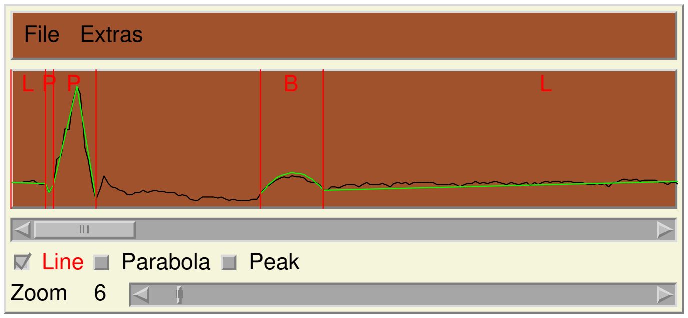



Abbildung 9.8: Die Vorverarbeitungsphase

kennen muß, um mit dem Debuggingprozeß fortschreiten zu können. Der Anwender behält die Kontrolle über die Suche nach dem Fehler. In dem Fall, in dem die *CPM*-Methode verwendet wird, können natürlich die Debuggingpunkte ausgewählt werden. Weiterhin kann die Benutzeroberfläche die Teile des Baums markieren, die schon korrekt oder uninteressant für das Problem sind. Ähnlich ermöglicht die Benutzeroberfläche die Auswahl der Anfangspunkte des Slicing-Verfahrens. Knoten können ein- oder ausgeblendet werden. Diese Eigenschaft ist sehr nützlich für den *EKG*-Klassifikator, weil er eine enorme Menge von Parametern hat. Der Anwender kann durch Drücken der Knöpfe *correct*, *incorrect* und *cancel* antworten. Wie wir schon früher erklärt haben (siehe Kapitel 8.5), wird der gerade erkannte Teil der *EKG*-Wellenform (siehe Abbildung 8.11) auch gezeigt, um die Entscheidung des Benutzers zu unterstützen.

- *Quellcode-Browser*: Der statische Aufrufgraph zeigt, welches Prädikat von der gegebenen Klausel aufgerufen werden kann. Mit Hilfe dieses Fensters kann die statische Programmstruktur als Komplement der dynamischen gezeichnet werden. Das *Datenanalyse-Fenster* (*information retrieval display*) gibt die Zeilennummern und die Klauseln an, die zu den vom Anwender ausgewählten Mustern passen.

- *Editor-Interface:* Durch Drücken der Maustaste über einem Knoten des Ableitungsbaums öffnet sich ein Editorfenster und in diesem Fenster kann man den Teil des PROLOG-Programms, der dem Knoten entspricht, sehen. Nach dem Speichern des Programms kann es weiter durch *SPECG* überprüft werden.

- *Testbeschreibung und Datenbankmanagement:* Alle Daten, die in *SPECG* verwendet werden, können mit wenigen Mausklicks geladen und gespeichert werden. Die Datenbank wird zusammen mit der ganzen Debuggersitzung gespeichert. Alle Daten, die mit der Debuggersitzung in Zusammenhang stehen, können gespeichert und geladen werden. Die Sammlung von Daten enthält unter anderem den Namen der Beschreibung und der Testdatenbank, die laufende Traversierungsmethode (Bottom-Up, Top-Down, Divide-and-Query) und die aktuelle Debuggermethode (Original Shapiro, CPM, Slicing oder die Kombination aller drei Methoden).

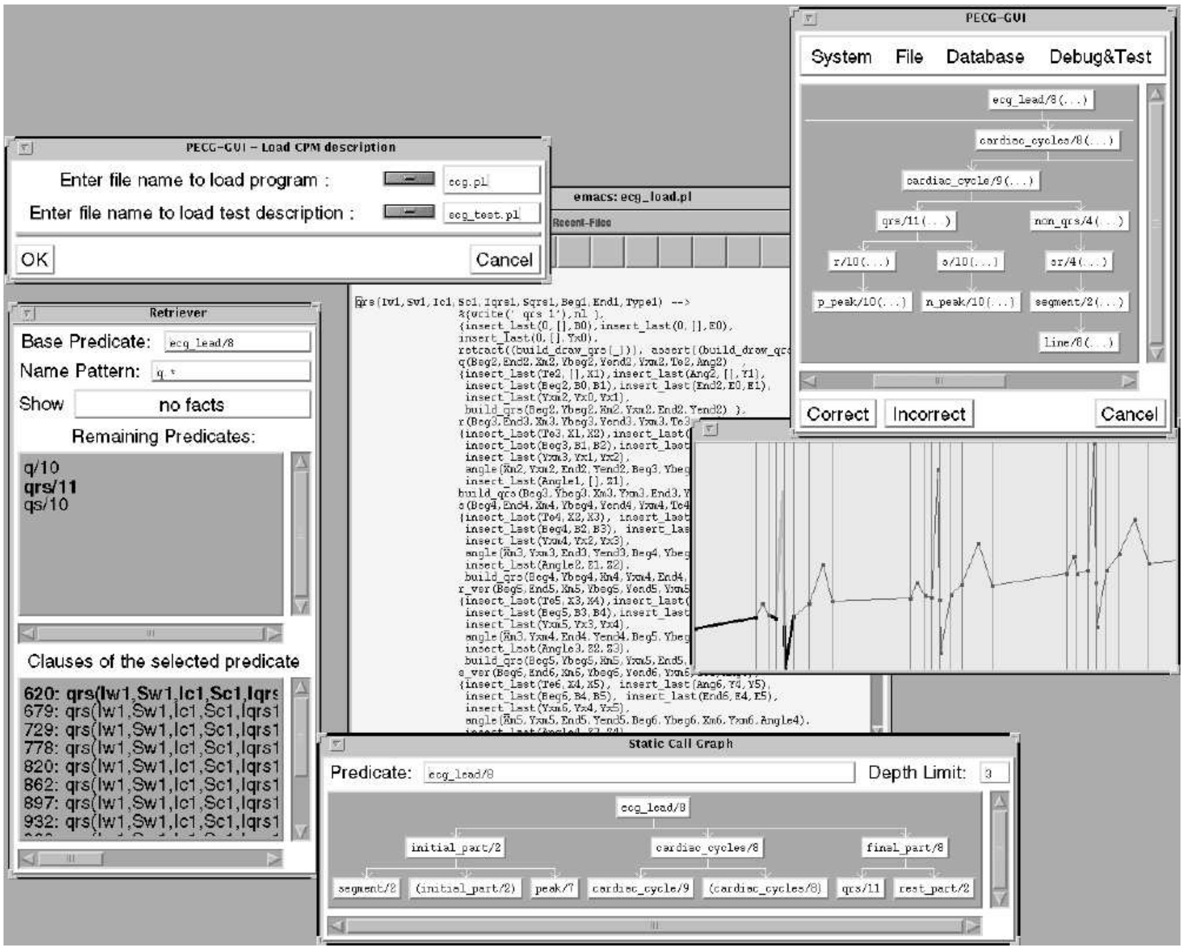


Abbildung 9.9: Die vollständige *SPECG-GUI* mit allen Bäumen, Browsern und Editoren

## 9.4 Lernen der Syntax der Grammatik

Im nächsten Kapitel werden wir erklären, wie das integrierte System die Anfangs-*DCG*-Repräsentation der Wellenformen verfeinern kann. Hierbei beschränken wir uns als Beispiel auf `cardiac_cycles` aus der Grammatik. Alle `cardiac_cycles` enthalten eine oder mehrere `cardiac_cycle`, aber der wichtigste Teil des Herzschlagzyklus ist der *QRS*-Komplex. Deswegen wurde dieser Teil ausgewählt, um den Lernprozeß zu demonstrieren. Die Beschreibung des *QRS*-Komplexes kann aus ein bis sieben Spitzen aufgebaut werden und, wie wir in Kapitel 8.5 gezeigt haben, gibt es dreizehn Regeln, die die *QRS*-Muster erkennen. In diesem Kapitel werden wir versuchen, eine Anfangs-*DCG*-Grammatik zu verfeinern, die dann die korrekte Reihenfolge von Spitzen erkennen kann.

Die Eingabe enthält in diesem Fall eine Grammatik, die `cardiac_cycles` beschreibt, die positiven und negativen Beispiele und das Hintergrundwissen. Die Aufgabe des Systems ist, das korrekte Programm zu definieren, das nur die positiven Beispiele erkennen kann.

Die Eingabe des Systems ist eine übergeneralisierte Grammatik der *EKG*-Wellenformen. Außerdem enthält sie Hintergrundwissen, das nötig ist, um positive und negative Beispiele abzuleiten. Das System verwendet das *IMPUT*-System, um diese übergeneralisierte Grammatik zu verbessern, so daß die modifizierte Grammatik alle positiven, aber keine negativen Beispiele akzeptiert. Das Hintergrundwissen wird durch eine Menge von Klauseln definiert, die während des Lernprozesses unverändert bleiben. Alle Beispiele sind Grundatome und werden so definiert:

**Beispiel 110** Die vorgegebenen positiven und negativen Beispiele

```
positive cardiac_cycles([p_peak,n_peak,line,p_peak,par,...],[]).
...
negative cardiac_cycles([p_peak,p_peak,n_peak,line,p_peak,...],[]).
```

Wir werden hier weiterhin die in 8.3 eingeführte Notation verwenden.

**Beispiel 111** Die Eingabe für *SPECG*: die Anfangsgrammatik, die Beispiele und das Hintergrundwissen

```
 0: cardiac_cycles --> cardiac_cycle, cardiac_cycles.
 1: cardiac_cycles --> {true}.
 2: cardiac_cycle  --> qrs, non_qrs.

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 %% the over-generalized part of the grammar %%
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 3: qrs     --> peak,peaks.
 4: peaks   --> peak,peaks.
 5: peaks   --> {true}.
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 6: non_qrs --> sr.
 7: non_qrs --> interwave_segment, t, interwave_segment.
 8: non_qrs --> interwave_segment, p, interwave_segment.
 9: non_qrs --> interwave_segment, t, interwave_segment, p,
                interwave_segment.

 10: sr  --> segment, interwave_segment.
 11: sr  --> peak, interwave_segment.

 12: interwave_segment --> segment, interwave_segment.
 13: interwave_segment --> {true}.
 14: interwave_segment --> peak, interwave_segment.

 15: t  --> t_or_p.
 16: p  --> t_or_p.

 17: background (t_or_p  --> [p_peak], [n_peak]).
 18: background (t_or_p  --> [n_peak], [p_peak]).
 19: background (t_or_p  --> [p_peak]).
 20: background (t_or_p  --> [n_peak]).
 21: background (peak    --> [p_peak]).
 22: background (peak    --> [n_peak]).
 23: background (segment --> [line]).
```

```
 24: background (segment --> [par]).

 25: positive cardiac_cycles([p_peak,n_peak,line,...],[]).
 26: positive cardiac_cycles([n_peak,p_peak,line,...],[]).
 27: positive cardiac_cycles([n_peak,p_peak,n_peak,...],[]).
 28: positive cardiac_cycles([p_peak,n_peak,p_peak,...],[]).
 ...
 37: negative cardiac_cycles([p_peak,p_peak,line,...],[]).
 38: negative cardiac_cycles([n_peak,n_peak,line,p_peak,...],[]).
 39: negative cardiac_cycles([p_peak,p_peak,n_peak,...],[]).
 40: negative cardiac_cycles([n_peak,n_peak,p_peak,...],[]).
 ...
```

Die Hauptschritte in diesem Kapitel sind die folgenden: Der Anwender ruft das Programm auf und gibt den Namen der Eingabedatei ein. Alle eingelesenen Klauseln werden durchnumeriert. Die bei der Unfoldingtransformation entstehenden Klauseln werden durch ein Paar von Zahlen *i-j* gekennzeichnet, wobei die erste Zahl die Elternklausel und die zweite die substituierte Klausel angibt.

**Beispiel 112** Anfang des Lernprozesses

```
Welcome to  IMPUT learning system.
Please enter the filename to be processed: cardiac_cycle_description.

The background knowledge is:
 17: t_or_p(A,B) :- 'C'(A,p_peak,C), 'C'(C,n_peak,B)
 ...
 24: segment(A,B):- 'C'(A,par,B)

The theory that needs to be specialized is:
 0: cardiac_cycles --> cardiac_cycle(A,C),cardiac_cycles(C,B)
 1: cardiac_cycles --> B=A
 2: cardiac_cycle  --> qrs(A,C),non_qrs(C,B)
 3: qrs            --> peak(A,C),peaks(C,B)
 ...
 16: p(A,B)        :-  t_or_p(A,B)

The positive examples are:
 25: cardiac_cycles([p_peak,n_peak,line,...])
 ...
 36: cardiac_cycles([n_peak,p_peak,n_peak,...])

The negative examples are:
 37: cardiac_cycles([p_peak,p_peak,line,...])
 ...
 76: cardiac_cycles([n_peak,n_peak,p_peak,...])
```

Die positiven und negativen Beispiele werden automatisch überprüft. Am Anfang sind sowohl die positiven als auch die negativen Beispiele aus dem Eingabeprogramm ableitbar. Als erstes untersucht das System, ob die positiven und negativen Beispiele verschieden sind. Als zweites sucht es nach Klauseln, die nicht nötig sind, um positive Beispiele ableiten zu können. Wenn solche Klauseln gefunden werden, dann werden sie aus dem Programm entfernt und die übriggebliebenen Klauseln werden angezeigt. In diesem Beispiel werden die Klauseln 7, 8, 9, 11, 12, 14, 15 und 16 gelöscht.

Als nächstes nimmt *SPECG* die negativen Beispiele. Wenn es ein negatives Beispiel findet, das im Programm ableitbar ist, dann wird die Lernphase aufgerufen, ansonsten meint das System, daß das korrekte Programm gefunden wurde. Hier werden nur die ableitbaren negativen Beispiele auf dem Bildschirm erscheinen.

Der erste Schritt im Lernen ist die Suche nach einer Klausel, die man dem Unfolding unterziehen kann. Hier werden dem Anwender vom Debugger Fragen gestellt, die entweder mit *yes* oder mit *no* beantwortet werden müssen. Die Antwort *no* bedeutet, daß der Debugger die falsche Klausel gefunden hat. In diesem Fall wird der Debuggingprozeß beendet, und der Unfoldingprozeß beginnt. Sonst geht der Debugger weiter zum nächsten Prädikat.

**Beispiel 113** Die letzte vom Debugger gestellte Frage

```
Is it ok [qrs([p_peak,p_peak])] (y/n) n
```

Die Antwort ist *no*, weil in der korrekten Klassifikation eine positive Spitze nur nach einer negativen Spitze stehen kann. Betrachten wir unser Beispiel, wie die korrekte Grammatik gelernt wird.

**Beispiel 114** Der erste Schritt im Lernen: Suche nach der fehlerhaften Klausel

```
37: cardiac_cycles([p_peak,p_peak,line,...[]) covered.
...
76: cardiac_cycles([n_peak,n_peak,p_peak,],[]) covered.

The fact cardiac_cycles([p_peak,p_peak,line,...],[]) is covered by the theory.
Starting the false proc. algorithm to determine the basis of the unfolding.
Is it ok [peaks([])] (y/n) y
Is it ok [peaks([p_peak])] (y/n) y
Is it ok [qrs([p_peak,p_peak])] (y/n) n
```

Der komplizierteste Teil des Systems ist der Unfolding-Teil. Wie wir schon früher erwähnt haben, wird angenommen, daß, wenn ein negatives Beispiel von der aktuellen Version des Programms ableitbar ist, es mindestens eine Klausel gibt, die für die inkorrekte Abdeckung verantwortlich ist. Es wurde ermittelt, daß die Klausel `peak --> peak(A,C), peaks(C,B)` inkorrekt ist. Diese Klausel kann nicht einfach aus dem Anfangsprogramm entfernt werden, weil die Ableitung der positiven Beispiele diese Klausel enthält.

**Beispiel 115** Unfolding-Prozeß für die Klausel `peak --> peak(A,C), peaks(C,B)`

```
 Unfolding at the clause instance:
    3: qrs-->peak([p_peak,p_peak,line,...],[p_peak,line,p_peak,...]),
             peaks([p_peak,line,p_peak,...],[line,p_peak,par,...])

  - trying resolvent(s): [3-1] actual minimum is: 12.5101.
  - trying resolvent(s): [3-2] actual minimum is: 0.

The result of the unfolding is:
 0: cardiac_cycles-->cardiac_cycle(A,C),cardiac_cycles(C,B)
 1: cardiac_cycles-->B=A
 2: cardiac_cycle-->qrs(A,C),non_qrs(C,B)
```

```
  3-4: qrs(A,B):-peak(A,C),peak(C,D),peaks(D,B)
  3-5: qrs(A,B):-peak(A,C),B=C
  ...
```

In dem neuen Programm kann das erste negative Beispiel nicht abgeleitet werden, aber dieses Programm deckt andere negative Beispiele ab. Natürlich kann der Debugger mehrmals verwendet werden, um die nächste falsche Klausel zu finden und zu zerlegen. Der letzte Schritt des Lernens ist der folgende:

**Beispiel 116** `Checking positive examples:`

```
Checking negative examples:
  66: cardiac_cycles([n_peak,p_peak,n_peak,...],[]) covered.

The above theory:
  covers   12 positive samples from 12 (100.00%) and
  fails on 39 negative samples from 40 (97.50%).

The fact cardiac_cycles([n_peak,p_peak,n_peak,...],[]) is covered by the theory.
Starting the false proc. algorithm to determine the basis of the unfolding.

Unfolding at the clause instance:
 3-4-4-22-22-4-21-4-21-4-22-21-4:
 qrs([n_peak,p_peak,n_peak,...],[line,p_peak,par,...]):-
   C([n_peak,p_peak,n_peak,...],n_peak,[p_peak,n_peak,p_peak,...]),
   C([p_peak,n_peak,p_peak,...],p_peak,[n_peak,p_peak,n_peak,...]),
   C([n_peak,p_peak,n_peak,...],n_peak,[p_peak,n_peak,p_peak,...]),
   C([p_peak,n_peak,p_peak,...],p_peak,[n_peak,p_peak,p_peak,...]),
   C([n_peak,p_peak,p_peak,...],n_peak,[p_peak,p_peak,line,]),
   C([p_peak,p_peak,line,...],p_peak,[p_peak,line,p_peak,...]),
   peak([p_peak,line,p_peak,...],[line,p_peak,par,...]),
   peaks([line,p_peak,par,...][line,p_peak,par,...])

- trying resolvent(s): [3-4-4-22-22-4-21-4-21-4-22-21-4-7] actual minimum is: 3.44203.
- trying resolvent(s): [3-4-4-22-22-4-21-4-21-4-22-21-4-8] actual minimum is: 3.52546.

The result of the unfolding is:
 0: cardiac_cycles-->cardiac_cycle(A,C),cardiac_cycles(C,B)
 1: cardiac_cycles-->B=A
 2: cardiac_cycle-->qrs(A,C),non_qrs(C,B)
 3-4-4-21-21-4-4-22-4-22-4-21: qrs(A,B):-
   C(A,p_peak,C),C(C,n_peak,D),C(D,p_peak,E),C(E,n_peak,F),
   C(F,p_peak,G),C(G,n_peak,H),C(H,p_peak,I),peaks(I,B)
 ...

Checking positive examples:

Checking negative examples:

The above theory:
  covers   12 positive samples from 12 (100.00%) and
  fails on 40 negative samples from 40 (100.00%).
```

Der Lernprozeß ist beendet, weil mit Hilfe des generierten Programms alle positiven und keine negativen Beispiele ableitbar sind. Das vollständige Ergebnis, also das gelernte Programm, ist das folgende:

**Beispiel 117** Das Ergebnis des Lernprozesses

```
0: cardiac_cycles --> cardiac_cycle,cardiac_cycles.
1: cardiac_cycles --> {true}.
2: cardiac_cycle  --> qrs,non_qrs.
3-4-4-21-21-4-4-22-4-22-4-21: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak],[p_peak],[n_peak],[p_peak],peaks.

3-4-4-21-21-4-4-22-4-22-5: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak],[p_peak],[n_peak].

3-4-4-21-21-4-4-22-5-21: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak],[p_peak].

3-4-4-21-21-4-5-22: qrs -->
  [p_peak],[n_peak],[p_peak],[n_peak].

3-4-4-21-21-5: qrs --> [p_peak],[n_peak],[p_peak].

3-4-4-22-22-4-21-4-21-4-22-21-4-22: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak],[n_peak],[p_peak],[n_peak],peaks.

3-4-4-22-22-4-21-4-21-4-22-21-5: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak],[n_peak],[p_peak].

3-4-4-22-22-4-21-4-21-5-22: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak],[n_peak].

3-4-4-22-22-4-21-5: qrs -->
  [n_peak],[p_peak],[n_peak],[p_peak].

3-4-4-22-22-5: qrs --> [n_peak],[p_peak],[n_peak].
3-4-5-21-22: qrs --> [n_peak],[p_peak].
3-4-5-22-21: qrs --> [p_peak],[n_peak].
3-5: qrs --> peak.
5: peaks --> {true}.
6: non_qrs --> sr.
7: non_qrs --> interwave_segment, t, interwave_segment.
8: non_qrs --> interwave_segment, p, interwave_segment.
9: non_qrs --> interwave_segment, t, interwave_segment, p,
             interwave_segment.
10: sr  --> segment, interwave_segment.
11: sr  --> peak, interwave_segment.
12: interwave_segment --> segment, interwave_segment.
13: interwave_segment --> {true}.
14: interwave_segment --> peak, interwave_segment.
15: t  --> t_or_p.
16: p  --> t_or_p.
```

## 9.5 Lernen der Schwellenregeln einer *EKG*-Grammatik

Im vorigen Kapitel haben wir gezeigt, daß das *SPECG*-System dem Anwender helfen kann, eine korrekte Syntax der *EKG*-Grammatik anzufertigen. Diese Grammatik kann weiterhin verwendet werden, um die Morphologie der *QRS*-Komplexe zu determinieren. Um entscheiden zu können, ob eine gegebene Folge von Spitzen zu einer Klasse von *QRS*-Komplexen gehört oder nicht, müssen wir die numerischen Eigenschaften dieser Spitzen benutzen. Das bedeutet, daß die *EKG*-Grammatik Regeln enthalten muß, die diese numerischen Werte (Energie, Winkel der Spitze) und manche Schwellenwerte für Entscheidungen überprüfen. Diese Regeln werden als *Schwellenregeln* bezeichnet.

In diesem Kapitel zeigen wir, wie *SPECG* verwendet werden kann, um diese Schwellenregeln und -werte abzuleiten. Dieses Problem könnte mit Hilfe eines Lernsystems, das Attributwerte lernen kann, gelöst werden. Wir haben aber eine konsistente *ILP*-Lernmethode gewählt, um sowohl den Syntax- als auch den Semantikteil der *EKG*-Grammatik zu lernen.

In dem oben beschriebenen Lernverfahren wurde vermutet, daß der Anwender die genauen Schwellenwerte und -regeln nicht kennt. Er definiert nur allgemeine Regeln in den Grenzintervallen für diese Schwellenwerte. Nehmen wir an, daß wir jetzt dasjenige *QRS*-Muster untersuchen, das nur zwei Spitzen enthält. Die anderen Arten von *QRS* können genauso definiert werden, so daß diese auf eine ähnliche Weise gelernt werden.

**Beispiel 118** Eine Schwellenregel zum *QRS*-Komplex mit zwei Spitzen

```
decide_flag(TeR,TeS,AngR,AngS,Angle) :-
     possible_threshold(Eps1,Eps2,Eps5),
     define_flag1(TeR,TeS,Eps1),
     define_flag2(AngR,AngS,Eps5),
     define_flag1(Angle,Eps2).
```

wobei die Werte der Argumente `TeR` und `TeS` die Energie der Spitze bedeuten, so lange `AngR` und `AngS` den Winkel zwischen den linken beziehungsweise rechten Armen der Spitzen definieren. `Angle` definiert den Winkel zwischen dem rechten Arm der Spitze $i$ und dem linken Arm der Spitze $i+1$. Es wird angenommen, daß diese Werte schon vorhanden sind.
Es gibt zwei verschiedene Aufgaben beim Lernen. Diese sind:

- Lernen der Beziehung zwischen den verschiedenen Komponenten
- Lernen der Werte von `Eps1`, `Eps2` und `Eps5`

Wir beginnen mit einem sehr allgemeinen Programm. Wir geben hier nur einen Teil davon an.

**Beispiel 119** Ein allgemeines Programm für das Lernen von Schwellenregeln

```
possible_threshold(Eps1,Eps2,Eps5) :-
  threshold1(Eps1),
  threshold2(Eps2),
  threshold5(Eps5).

threshold1(Eps1) :- Eps1 is 10000.
threshold1(Eps1) :- Eps1 is 20000.
...
```

```
threshold1(Eps1) :- Eps1 is 180000.

threshold2(Eps2) :- Eps2 is 20.
...
threshold2(Eps2) :- Eps2 is 50.

threshold5(Eps5) :- Eps5 is 10.
...
threshold5(Eps5) :- Eps5 is 120.

flag1(TeR,TeS,Eps1) :- TeR > Eps1 ; TeS > Eps1.
flag1(TeR,TeS,Eps1) :- TeR < Eps1 ; TeS < Eps1.
flag1(TeR,TeS,Eps1) :- TeR > Eps1 , TeS > Eps1.
flag1(TeR,TeS,Eps1) :- TeR < Eps1 , TeS < Eps1.

flag2(AngR,AngS,Eps5) :- AngR > Eps5 ; AngS > Eps5.
flag2(AngR,AngS,Eps5) :- AngR < Eps5 ; AngS < Eps5.
flag2(AngR,AngS,Eps5) :- AngR > Eps5 , AngS > Eps5.
flag2(AngR,AngS,Eps5) :- AngR < Eps5 , AngS < Eps5.

flag3(Angle,Eps2) :- Angle >Eps2.
flag3(Angle,Eps2) :- Angle <Eps2.

positive decide_flag(158466,34405,11.1,27.7,36.8).
positive decide_flag(162693,44257,11,29.4,34.3).
positive decide_flag(177737,35542,10.4,28.2,35.1).
positive decide_flag(35672,23211,11,29.2,36.8).
...
negative decide_flag(1939,0,113.8,0,0).
negative decide_flag(5860,42508,79.9,29.2,31.2).
negative decide_flag(21600,32811,10.2,11,23.2).
negative decide_flag(6280,5413,32.5,45.3,28.2).
...
```

Dieses Programm akzeptiert viele Beziehungen und Schwellenwerte. Der Effekt dieser Übergeneralisierung ist, daß zahlreiche negative Beispiele als $QRS$-Komplex erkannt werden. Durch Anwendung der *ILP*-Methode *IMPUT* kann dieses Programm jedoch in ein korrektes Programm transformiert werden. Wir wollen diese Transformation hier nicht detailliert durchführen, weil der Prozeß ganz ähnlich dem ist, den wir bei der Beschreibung des Lernens der syntaktischen Struktur zu `cardiac_cycles` erklärt haben. Hier möchten wir nur bemerken, daß, wenn die Unfolding-Transformation nicht durchgeführt werden konnte, die Klausel entfernt wird und am Ende des Lernens alle redundanten Schwellenwerte gelöscht werden. Das transformierte Programm ist das folgende:

**Beispiel 120** Das Ergebnis des Lernens der Schwellenregeln

```
decide_flag(TeR,TeS,AngR,AngS,Angle) :-
  possible_threshold(30000,30,30),
  define_flag1(TeR,TeS,30000),
  define_flag2(AngR,AngS,30),
  define_flag1(Angle,30).
```

```
possible_threshold(30000,30,30) :-
  threshold1(30000),
  threshold2(3
  threshold5(30).

threshold1(30000).
threshold2(30).
threshold5(30).

flag1(TeR,Tes,Eps1)   :- TeR > Eps1 ; TeS > Eps1.
flag2(AngR,Angs,Eps5) :- AngR < Eps5 , AngS < Eps5.
flag3(Angle,Eps2)     :- Angle <Eps2.
```

In diesem Kapitel haben wir gezeigt, wie das *SPECG*-System weiterentwickelt werden kann. Die Erweiterung ermöglicht, daß das System nicht nur *EKG*-Wellenformen klassifizieren, sondern auch lernen kann. Das *SPECG* Klassifikatorprogramm wurde mit dem induktiven Lernverfahren *IMPUT* kombiniert. Das von uns entwickelte Klassifikatorprogramm *SPECG* kann sowohl die Syntax- als auch die Semantikregeln einer *EKG*-Grammatik lernen. Diese Regeln werden benutzt, um *QRS*-Komplexe zu klassifizieren und um zwischen *QRS*− und nicht *QRS*-Mustern zu unterscheiden. Das System wird durch eine graphische Oberfläche gesteuert. Dieses System zeigt, wie die *theory-revision* Methode für das Lernen großer PROLOG-Programme benutzt werden kann.

# Kapitel 10

# Schlußbemerkungen

Zum Abschluß werden die wichtigsten Ergebnisse zusammengefaßt, und es wird in einem Ausblick auf weiterführende Fragestellungen eingegangen, die in dieser Arbeit nicht mehr behandelt werden konnten.

## 10.1 Zusammenfassung der Ergebnisse

Ziel der vorliegenden Arbeit war die Erstellung einer integrierten Methode zum Debuggen und funktionalen Testen logischer Programme.

Dazu wurden zuerst die verschiedenen Diagnosealgorithmen von Shapiro eingeführt, die die Grundvorgehensweise der im *IDTS*-System implementierten Algorithmen lieferten. Das algorithmische Debugging von Shapiro kann eine fehlerhafte Klausel identifizieren, aber der Nachteil dieses Debuggingverfahrens ist, daß der Benutzer eine große Anzahl von Fragen beantworten muß. Deswegen wurden Shapiros Methoden in zwei Richtungen erweitert:

- *Integration der* CPM-*Methode (*Category Partition Testing Method*)*: Bei dieser Methode werden Äquivalenzklassen über den Kombinationen der Eingabeparameter gebildet. Dabei wird vorausgesetzt, daß das Verhalten der Elemente innerhalb einer Klasse (Testrahmen) identisch ist. Wenn ein Element zu einer Klasse gehört, die schon getestet wurde, dann kann das Testergebnis unmittelbar verwendet werden.

- *Integration des Slicing-Verfahrens*: Bei der Berechnung der Slice wird der Ableitungsbaum verkleinert. Nach dem Slicing werden nur diejenigen Teile des Ableitungsbaums behalten, die auf einem bestimmten vom Anwender ausgewählten Knoten im ursprünglichen Ableitungsbaum, Wirkung haben. Wenn der ausgewählte Punkt ein falsches Ergebnis enthält, dann wird in einem der vorherigen Knoten, von denen dieser Punkt abhängig ist, der Fehler gemacht. Die übriggebliebenen Knoten müssen nicht überprüft werden, weil sie keinerlei Wirkung auf das falsche Ergebnis haben.

Es wurde noch im ersten Teil der Arbeit gezeigt, daß die Kombination der drei Techniken (Shapiro + *CPM* + Slicing) das beste Ergebnis hinsichtlich der Anzahl der Fragen liefert.

Im zweiten Teil der Arbeit wurde ein Vergleich mit anderen existierenden Systemen (*APROPOS*, *PRESET*, *APD*, *RD*, *OPIUM*, *PTP*, *DED*, *EDS*, *ADAss*) gezogen. Dabei stellte sich heraus, daß *IDTS* viele gute Eigenschaften von verschiedenen Systemen vereinigt, unter anderem die folgenden:

- Das System besitzt einen hohen Automatisierungsgrad: Es verwendet während des Debuggings gesammelte Informationen, um unnötige Fragen an den Anwender zu vermeiden.
- Das System ist ein nützliches Werkzeug für verschiedene Gruppen von Anwendern: Jeder kann seine bisherige Kenntnisse in den Debugger einbauen.
- Wenn das verwendete Wissen, das in Form einer Testspezifikation eingegeben wurde, Fehler enthält, hilft *IDTS* sogar, diese zu finden. Ebenso ist es, wenn der Anwender beim Beantworten der Fragen eine falsche Antwort eingegeben hat.
- Die Arbeit mit dem System wird durch die graphische Benutzeroberfläche *GIDTS* wesentlich erleichtert.

Im dritten Teil der Arbeit wurden zwei neue Systeme *SPECG* und *IMPUT* gezeigt, wobei das erste eine Anwendung von *IDTS* ist, um Elektrokardiogrammwellenformen zu klassifizieren, und das zweite ein Lernsystem für induktive logische Programmierung ist. Am Ende der Arbeit wurde gezeigt, daß durch die Integration von *SPECG* und *IMPUT* ein System entstand, das mit Hilfe von positiven und negativen Beispielen und Hintergrundwissen eine korrekte Grammatik erzeugen kann. Das ist ein Beweis dafür, daß größere Programme durch die *theory revision learning*-Methode gelernt werden können, weil das so entwickelte System die syntaktischen und semantischen Regeln einer *EKG*-Grammatik lernen kann.

## 10.2 Ausblick

Das *IDTS*-System erhöht die Effizienz des Debugging-Verfahrens dadurch, daß es eine Testdatenbank verwendet. Diese Datenbank wird aus einer vom Anwender eingegebenen Testbeschreibung durch das System generiert. Die Schwierigkeit liegt in der Korrektheit und der Vollständigkeit dieser Spezifikation.

Es kann untersucht werden, inwieweit Testspezifikationen automatisch erzeugbar sind. Für die Erzeugung der Testspezifikation sollte auch ein Werkzeug implementiert werden, das dann die folgenden Anforderungen erfüllen sollte:

- Das Werkzeug soll als Erweiterung der bereits existierenden graphischen Benutzeroberfläche *GIDTS* implementiert werden.
- Das Werkzeug soll die Eingabe der Testspezifikation in Form eines komfortablen Editors unterstützen.
- Eine Bibliothek von typischen Testbeschreibungen für einzelne Prädikate soll über den Editor erweiterbar sein und die Erstellung neuer Testbeschreibungen vereinfachen.

Eine andere Erweiterung wäre, eine statische Codeanalyse in das *IDTS*-System einzubauen. Mit Hilfe der Codeanalyse (die zum Beispiel die Erreichbarkeit von Code untersucht oder fehlende Prädikate identifiziert usw.) können bestimmte Fehlerklassen in Programmen ausfindig gemacht und korrigiert werden. Auch eine umfangreiche Fehleranalyse nach erfolgreichem Debugging erleichtert die Fehlerkorrektur.

Das vorhandene graphische Werkzeug *GIDTS* wäre ein guter Anfangspunkt dazu. Dann wäre die Aufgabe einerseits, dieses um die Codeanalyse zu erweitern. Dies bedeutet die Integration eines PROLOG-Parsers in *GIDTS*. Dieser Parser könnte die Datenstruktur, die für die

Codeanalyse unerläßlich ist, konstruieren. Dann könnten verschiedene Codeanalysevarianten, mindestens jedoch die folgenden, implementiert werden:

- Erreichbarkeit von Prädikaten
- Fehlende Prädikate

Andererseits könnte die bisherige Fehleranalyse verfeinert werden.

Außerdem könnte *IDTS* in Zukunft durch Einbauen von anderen Werkzeugen zu einer vollständigen Programmierumgebung erweitert werden. Dazu gehört die Wiederverwendung der Testdatenbank in einem wiederholten Testverfahren ebenso wie die Implementierung von Erweiterungen, die die Programmierung von PROLOG-Programmen erleichtern (wie zum Beispiel graphische Repräsentation des statischen Aufrufsgraphen).

Was das *SPECG*-System betrifft, ist unsere Idee, daß wir ein komplettes Diagnostikprogramm entwickeln, das eine individuelle Diagnose für jeden Kranken liefern kann. Was bedeutet das? Schon früher sind *EKG*-Diagnostikprogramme ausgearbeitet worden, sogar die heutzutage verwendeten *EKG*-Maschinen können eine Diagnose liefern, aber sie berücksichtigen die Krankengeschichte des Patienten nicht, sondern basieren auf allgemeinen Kriterien.

Unser System ist fähig, ein Klassifikatorprogramm zu lernen, das die *EKG*-Aufnahmen von einzelnen Patienten korrekt klassifiziert. Es berücksichtigt zum Beispiel, wenn ein *EKG* wegen Herz- oder anderen früheren Krankheiten modifiziert ist. Dieser veränderte Klassifikator wird bei späteren Untersuchungen verwendet. Die Integrierung eines Diagnostikmoduls in *SPECG* ist auch im Gang.

Das weiterzuentwickelnde System könnte die folgenden Funktionen erfüllen:

- Erkennung eines *EKG* mit einer *EKG*-Maschine
- Transport der Daten auf einen PC und Dekodierung
- Automatische Vorverarbeitung, bei der die Primitive ausgewählt werden
- Generierung der Eingabedaten für die Klassifikatorgrammatik, die die Beschreibung der Primitive (Namen der Primitive und vom System berechnete numerische Werte) enthält
- Klassifizierung mit Hilfe der Grammatik: Diese Grammatik kann sowohl die Syntax als auch die Semantik der Regeln lernen
- Sammeln numerischer Werte, die für die Diagnose notwendig sind
- Starten des Diagnosemoduls

# Abbildungsverzeichnis

# Algorithmenverzeichnis

# Literaturverzeichnis


[ACG+97] Z. Alexin, J. Csirik, T. Gyimóthy, M. Jelasity und L. Tóth. Learning Phonetic Rules in a Speech Recognition System. In *Proc. of the Seventh International Workshop on Inductive Logic Programming (ILP97)*, Seiten 37–44, Prague, Czech Republic, 1997. LNAI series Vol 129 Springer Verlag.

[AdBR94] K.R. Apt, J. W. de Bakker und J.J.M.M. Rutten. *Logic Programming languages Constraints, Functions and Objects.* MIT Press, 1994.

[AGB95] Z. Alexin, T. Gyimóthy und H. Boström. Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner. In *Proceedings of the Fifth International Workshop on Inductive Logic Programming*, Seiten 437–452, Leuven, Belgien, 1995.

[AGB96] Z. Alexin, T. Gyimóthy und H. Boström. Algorithmic Debugging and Unfolding Transformations in an Interactive Learner. In *Proceedings of the 12th European Conference on Artificial Intelligence ECAI-96*, Seiten 403–407, Budapest, Hungary, 1996. John Wiley & Son's Ltd.

[AGB97] Z. Alexin, T. Gyimóthy und H. Boström. IMPUT: An Interactive Learning Tool Based on Program Specialization. In *Intelligent Data Analysis Vol 1 4*. Elsevier Holland, 1997.

[AGK95] Z. Alexin, T. Gyimóthy und G. Kókai. IDT: Integrated System for Debugging and Testing Prolog Programs. In László Varga, Hrsg., *Proc. of the Fourth Symposium on Programming Languages and Software Tools*, Seiten 312–323, 1995.

[Agr91] Hiralal Agrawal. Towards Automatic Debugging of Computer Programs. Ph.d thesis, Software Engineering Research Center of Purdue University, 1991.

[ALLM94] D. W. Aha, S. Lapointe, C. X. Ling und S. Matwin. Inverting implication with small training sets. In *Proc International Conference on Machine Learning, ECML-94*, Seiten 31–48. Springer Verlag, 1994.

[AM91] H. Alblas und B. Melichar. Attribute grammars, applications and systems. In *International Summer School on Attribute Grammars, Application and Systems (SAGA).* Lecture notes in computer science ; 545, June 4-13, 1991.

[AvE82] K.R. Apt und M.H. van Emden. Contrabution to the Theory of Logic Programming. In *Journal of ACM 29*, Seiten 841–862, 1982.

[BBLM86] R. Barbuti, H. Bellia, G. Levi und M. Martelli. LEAF: A Language which Integrates Logic, Equations and Functions. In DeGroot-Lindstrom, Hrsg., *Logic Programming Functions Relations and Equations*, Seiten 201–238. Prentice-Hall, 1986.

[BG93] M. Botta und A. Giordana. Smart+: A MultiStrategy Learning Tool. In *Proc. of the IJCAI-93*, Seiten 937–943, 1993.

[BG95] F. Bergadano und D. Gunetti. *Inductive logic Programming from machine learning to software engineering.* MIT Press, 1995.

[BGL+87] P. Bosco, E. Giovanetti, G. Levi, C. Moiso und C. Palamidessi. A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions. In Lassez, Hrsg., *Proc of the Fourth International Conference on Logic Programming, Series in Logic Programming*, Seiten 318–327. The MIT-Press, 1987.

[BGP89] F. Bergadano, A Giordana und S. Ponsero. Deduction in top-down inductive learning. In *Proc. of the Sixth International Workshop on Machine Learning*, Los Altos, CA, 1989. Kaufman.

[BGS94] F. Bergadano, A. Giordana und L. Saitta. *Machine Learning An Integrated Framework and its Application.* ELLIS HORWORD, 1994.

[BIA94] H. Boström und P. Idestam-Almquist. Specialization of Logic Programs by Pruning SLD-trees. In *Proc. of the Fourth International Workshop on Inductive Logic Programming (ILP-94)*, Seiten 31–47, Bad Honnef/Bonn, Germany, September 12-14. 1994.

[Bib82] W. Bibel. *Automated Theorem Proving.* Vieweg Braunschweig, 1982.

[Bin93] D. Binks. Declarative debugging of abstract data types in Gödel. In *Proc AADEBUG'93*, Linköping, Sweden, 1993. LNCS 749.

[BML89] I. Bratko, I. Mozetic und N. Lavrac. *KARDIO: a Study in Deep and qualitative Knowledge for Expert Systems.* MIT Press, 1989.

[BMV91] I. Bratko, S. Muggleton und A. Varsek. Learning qualitative models of dynamic systems. In *Proc of the Eight International Machine Learning Workshop*, San Mateo Ca, 1991. Morgan Kaufmann.

[Bö88] B. W. Böhm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Boc76] G. V. Bochmann. Semantic Evaluation from Left to Right. In *Communications of the ACM 19,2*, Seiten 55–62, 1976.

[Bos95] H. Boström. Specialization of Recursive Predicates. In *Proc. of the Eighth European Conference on Machine Learning*, Seiten 92–100. Springer-Verlag, 1995.

[BPM93] J. Boye, J. Paakki und J. Maluszynski. Dependency-Based Groundness Analysis of Functional Logic Programs. In *Research Report LiTH-IDA-R-93-20, Department of Computer and Information Science, Linköping University*, 1993.

[Bra86] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1986.

[Bun79] A. Bundy. Solving Mechanics Problems using Meta-Level Inference in Expert Systems. In *Micro Electronic Age*. Edinburgh University Press, Edinburg, 1979.

[Byr80] L. Byrd. Understanding the control flow of Prolog programs. In *Proc of the Logic Programming Workshop*, Debrecen, Hungary, 1980.

[CL73] C.L. Chang und R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[Cla79] K.L Clark. Predicate Logic as a Computational Formalism. In *Technical Report 79/59 Department of Computing, Imperial College, London*, 1979.

[Cla82] K.L Clark. An Introduction to Logic Programming. In D. Michie, Hrsg., *Introductions Reading in Expert System*, Seiten 93–112, New York, 1982.

[Cle93] K. K. Lau T. Clement. *Logic Program Synthesis and Transformation*. Springer Verlag, 1993.

[CM79] K.L. Clark und F.G. McCabe. The Control Facilities of IC-PROLOG. In *Expert Systems in Micro Electronic Age Michie D*, Seiten 122–149. Edinburgh University Press, 1979.

[CM81] W.F. Clocksin und C.S. Mellish. *Programming in Prolog*. Springer Verlag, 1981.

[Coh85] J. Cohen. Describing Prolog by its Interpretation and Compilation. In *Communications of the ACM*, Seiten 1311–1314, 1985.

[Coh86] S. Cohen. The APPLOG language. In De Groot Lindstöm, Hrsg., *Logic Programming, functions relations and equations*, Seiten 239–276. Prentice Hall, 1986.

[Cov85a] M. A. Covington. Eliminating Unwanted Loops in Prolog. In *ACM SIGPLAN Notices Vol 20, No. 1.*, 1985.

[Cov85b] M. A. Covington. A further Note on Looping in Prolog. In *ACM SIGPLAN Notices Vol 20, No. 8.*, 1985.

[CR88] E. J. Chikofsky und B. L. Rubenstein. Case: Reliability engineering for information systems. In *IEEE Software*, Seiten 11–16, March 1988.

[CWA+] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz, H. Nilsson und T. Sjöland. *SICStus Prolog User's Manual*. SICS, PO Box 1263, S-164 28 KISTA, Sweden.

[DE88] M. Ducasse und A-M. Emde. A Review of Automated Debugging Systems: Knowledge, Strategy, and Techniques. In *Proceeding of 10th International Workshop on Software Engineering*, Seiten 162–171, Singapore, 1988.

[DJL88] P. Deransart, M. Jourdan und B. Lorho. *Attribute grammars.* Lecture Notes in Computer Science 323, 1988.

[DM85] P. Deransart und J. Maluszynski. Relating Logic Programs and Attribute Grammars. In *Journal of Logic Programming 2*, Seiten 119–156, 1985.

[DM92] B. Dolsak und S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In *Inductive Logic Programming*, London, 1992. Academic Press.

[DM93] P. Deransart und J. Maluszynski. *A Grammatical View of Logic Programming.* The MIT Press, 1993.

[DN94] M. Ducasse und J. Noye. Logic Programming Environments Dynamic Program Analysis and Debugging. In *Journal of Logic Programming, 19/20*, Seiten 351–384, 1994.

[DNTM88a] W. Drabent, S. Nadjm-Tehrani und J. Maluszynski. Algorithmic Debugging with Assertions. In *Proc Meta-Programming in Logic Programming*, Seiten 375–378, 1988.

[DNTM88b] W. Drabent, S. Nadjm-Tehrani und J. Maluszynski. The Use of Assertion in Algorithmic Debugging. In *Proc of the International Conference on Fifth Generation Computer Systems*, Seiten 573–581, 1988.

[dRSB91] L. de Raedt, G. Sablon und M. Bruynooghe. Using interactive concept-learning for knowledge base validation and verification. In *Validation Verification and Testing Knowledge based Systems*, Seiten 177–190. John Wiley and Sons, 1991.

[Duc87] M Ducasse. Opium - An Extensible Tracer for Prolog. In *Technical Report Lp14*, Seiten 203–210, January 1987.

[Duc93] M. Ducasse. A Pragmatic Survey of Automated Debugging. In *Proceeding of First International Workshop AADEBUG'93*, Seiten 1–16, Linköping, Sweden, 1993. LNCS 749, Springer.

[Dze91] S. Dzeroski. Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science University of Ljubljana Slovenia, 1991.

[ED90] A-M. Emde und M. Ducasse. Automated Debugging of non-terminating Prolog programms. In *Proc. of the ICLP'90 Workshop on Logic Programming Environment*, 1990.

[Eis85a] Marc Eisenstadt. A Powerful Prolog Trace Package,Advances in Artificial Intelligence. In *ECAI.* Elsevier Science Publishers B.V. (North-Holland), 1985.

[Eis85b] Marc Eisenstadt. Retrospective Zooming: A Knowledge Based Tracing and Debugging Methodology for Logic Programming. In *Proc of the 9.th IJCAI*, 1985.

[Ekl94] F. Ekleund. Synthesis of Directionality Information for GAPLog Programs. mastersthesis, University of Linköping Reg -nr LITH-IDA-Ex-94, 1994.

[Fal95] A. Faller. *Der Körper des Menschen, Einführung in Bau und Funktionen.* Thieme, 1995.

[Fen92] C. Feng. Inducing temporal fault diagnostic rules from a qualitative model. In *Inductive Logic Programming, chapter 24.* Academic Press, 1992.

[Fer87] Gérard Ferrand. Error Diagnosis in Logic Programming an Adaptation of E.Y. Shapiro's Method. In *Journal of Logic Programming*, Seiten 177–198, 1987.

[Fla94] P. A. Flach. Logical Approaches to Machine Learning - an Overview. In *THINK, 1(2)*, Seiten 25–36, 1994.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposium in Applied Mathematics*, Seiten 19–32. American Society, 1967.

[Fri85] L. Friburg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proc Symposium on Logic Programming*, Seiten 178–184. IEEE, 1985.

[GBBT93] D. Gunetti, F. Bergadano, S. Brusotti und U. Trinchero. Inductive test case generation. In *Proc. of 3rd International Workshop on Inductive Logic Programming*, Seiten 11–24, 1993.

[GM86] J. Goguen und J. Mesegues. EQLOG: Equality Types and Generic Modules for Logic Programming. In *Logic Programming, Functions, Relations and Equations*, Seiten 295–364. Prentice Hall, 1986.

[GP95] T. Gyimóthy und J. Paakki. Static Slicing of Logic Programs. In *Proc. of AADEBUG'95 Saint-Malo, France*, Seiten 57–69, 1995.

[Gro92] M. Grobelnik. Markus: an optimized Model Inference System. In *Proc. Workshop on Logical Approaches to Machine Learning, Tenth European Conference on Artificial Intelligence*, Vienna, Austria, 1992.

[GvHWW94] U. Gerlach, N. van Husen, H. Wagner und W. Wirth. *Innere Medizin für Krankenpflegeberufe.* Thieme, 1994.

[Han95] W. J. Hansen. Andrew Bison Version A. 2.6. In *CMU*, 1995.

[HAPR94] J. Herrmann, R. Ackerman, J. Peters und D. Reipa. A multistrategy learning and its integration into an interactive floorplanning tool. In *Proc. Seventh Europen Conference on Machine Learning ECML-94*, Seiten 138–153. Springer Verlag, 1994.

[Har96] L. Harmath. Logikai programok szeletelése. mastersthesis, Research Group on Artificial Intelligence, Hungarian Academy of Sciences, 1996.

[HGAK93] T. Horváth, T. Gyimóthy, Z. Alexin und F. Kocsis. Interactive Diagnosis and Testing Logic Programs. In *Proceedings of the Third Symposium on Programming Languages and Software Tools*, Seiten 34–46, Kääriku Estonia, August 23-24 1993.

[HH93] V. Hirunkitti und C. J. Hogger. A generalised query minimisation for program debugging. In *Proc AADEBUG'93*, Linköping, Sweden, 1993. LNCS 749.

[Hil74] R. Hill. LUSH-Resolution and its Completeness. In *DCL Memo 78 Department of Artificial Intelligence. University of Edinburgh*, 1974.

[HRB90] S. Horwitz, T. Reps und D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *ACM Transaction on Programming Languages and System 12, 1*, Seiten 26–61. ACM Press, 1990.

[JS85] W. L. Johnson und E. Soloway. Proust : Knowledge based program understanding. In *IEEE Transactions on Software Engineering SE-11(3)*, Seiten 267–275, March 1985.

[KAK94] G. Kókai, Z. Alexin und F. Kocsis. The IDT System and its Application for Learning Prolog Programs. In *Proc. of the Sixth International Conference on Artificial Intelligence and Information-Control Systems of Robots*, Seiten 315–320, Smolenice Castle, Slovakia, 12-16 September 1994. Worlds Scientific.

[Kam93] M. Kamkar. Interprocedural Dynamic Slicing with Applications to Debugging and Testing. Diplomarbeit, Linköping University, 1993.

[KD94] J.-U. Kietz und S. Dzeroski. Induktive Logic Programming and Learnability. In *SIGART Bulletin, 5(1)*, Seiten 22–32, 1994.

[KHG97a] G. Kókai, L. Harmath und T. Gyimóthy. Algorithmic Debugging and Testing Prolog Programs. In *ICLP '97 The Fourteenth International Conference on Logic Programming*, Leuven, Belgium, 1997.

[KHG97b] G. Kókai, L. Harmath und T. Gyimóthy. IDTS: a Tool for Debugging and Testing of Prolog Programs. In *LIRA'97, The 8th Conference on Logic and Computer Science*, Seiten 84–95, Novi Sad, Yugoslavia, 1997.

[Kie93] J.-U. Kietz. Some lower bounds for the computational complexity of inductive logic programming. In *Proc. of the European Conference on Machine Learning*, Berlin, 1993.

[KL90] B. Korel und J. Laski. Dynamic Slicing of Computer Programs. In *The Journal of Systems and Software 13, 3*, Seiten 187–195, 1990.

[KMLS92] R. King, S. Muggleton, R. Lewis und M. Stenberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationship of trimethoprim analogues binding to dihydrofolate reductase. In *Proceeding of National Academy of Science 89(23)*, 1992.

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. In *Mathematical Systems Theory 2*, Seiten 127–145, 1968.

[Kom82] J. Komorowski. QLOG The Programming Environment for Prolog in LISP. In *Logic Programming*, Seiten 315–322. Academic Press, 1982.

[Kov94] M. Kovacic. MILP a stochastic approach to inductive logic programming. In *Proc. Fourth International Workshop on Inductive Logic Programming (ILP-94)*, Seiten 123–138, 1994.

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing 74*, Seiten 569–574, 1974.

[Kow79] R. Kowalski. Algorithmic = Logic + Controll. In *CACM 22/7*, Seiten 424–435, 1979.

[KW92] J.-U. Kietz und S. Wrobel. Controlling of the complexity of learning in logic through syntactic and task oriented models. In *Inductive Logic Programming*. Academic Press, 1992.

[LD93a] N. Lavrac und S. Dzeroski. *Inductive Logic Programming Techniques and Application*. Ellis Horwood, 1993.

[LD93b] Y-J. Lee und N. Dershowitz. Debugging logic programming using specifications. In *Proc AADEBUG'93*, Linköping, Sweden, 1993. LNCS 749.

[Lev94] G. Levi. *Advances in Logic Programming*. MIT Press, 1994.

[LL93] B.C. Lipman und B.S. Lipman. *ECG Pocket Guide*. Year Book Medical Publishers, Inc. Chicago, London, 1993.

[Llo87a] J. W. Lloyd. Declarative Error Diagnosis. In *New Generation Computing*, Seiten 133–154. OHMSHA LTD and Springer Verlag, 1987.

[Llo87b] J. W. Lloyd. *Foundation of Logic Programming*. Springer Verlag, 1987.

[Loo88] Chee-Kit Looi. Analysing Novices' Programs in Prolog Intelligent Teaching System. In *Proc of the European Conference on Artificial Intelligence*, Seiten 314–319, Munich, 1988.

[Lov69] D.W. Loveland. A Simplified Format for the Model Elimination Procedure. In *Journal ACM 16, 3*, July 1969.

[LRS74] P. M. Lewis, D. J. Rosenkrantz und R. E. Stearns. Attributed Translation. In *Journal of Computer and System Sciences 9*, Seiten 279–307, 1974.

[LS] Y. Lichtenstein und E. Shapiro. Abstract Algorithmic Debugging. In *Proc. JICSLP'88*, Seiten 298–306.

[LW96] N. Lavrac und S. Wrobel. Induktive Logikprogrammierung-Grundlagen und Techniken. In *Künstliche Intelligenz 3/96*, Seiten 46–54, 1996.

[Man74] Z. Manna. *Mathematical Theory of Computation*. MCGraw-Hill, New York, 1974.

[MB88] S. Muggleton und W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc Fifth International Conference on Machine Learning*, Seiten 339–352, San Mateo, CA, 1988. Morgan Kaufmann.

[MF] S. Muggleton und C. Feng. Efficient induction of logic programs. In *Proc First Conference on Algorithmic Learning Theory*, Seiten 368–388, San Mateo, Ca. Morgan Kaufman.

[MH84] C. Mellish und S. Hardy. *Integrating Prolog in the Poplog Environment, Implementation of Prolog.* Compell Ellis Horwood, 1984.

[Mit80] T. M Mitchell. The need for bias in learning generalizations. In *Technical Report CBM-TR-117, Department of Computer science, Rutgers University*, 1980.

[Mit82] T. M Mitchell. Generalization as search. In *Artificial Intelligenze 18(2)*, Seiten 203–226, 1982.

[MKS92] S. Muggleton, R. King und M. Stenberg. Protein secondary structure prediction using logic-based machine learning. In *Protein Engineering, 5(7)*, Seiten 647–657, 1992.

[MNRA92] J. Moreno-Navaro und M. Radriguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. In *Journal of Logic Programming, 12*, Seiten 191–223, 1992.

[Moe92] E. Moeales. Learning Chees pattern. In *Inductive Logic Programming.* Academic Press, 1992.

[Mor] K. Morik. *Knowledge acquisition and machine learning: theory, methods, and applications.* Academic Press.

[MPMC93] K. Morik, G. Potamias, V. Moustakis und G. Charissis. Knowledgeable learning using Mobal - a case study on medical domain. In *Proc. of the Application Workshop at ECML-93*, Vienna, 1993.

[MR93] K. Morik und A. Rieger. Learning action-oriented perceptual features for robot navigation. In *Proc. of the Learning Robots Workshop at ECML-93*, Seiten 13–26, 1993.

[MR94] S. Muggleton und L. De Raedt. Inductive Logic Programming: Theory and methods. In *Journal of Logic Programming 19 (20)*, Seiten 629–679, 1994.

[MSB92] S. Muggleton, A. Srinivasan und M. Bain. Compression, significance and accuracy. In *Proc. of the Ninth International Machine Learning Conference*, San Mateo CA, 1992. Morgan Kaufmann.

[Mug92] S. Muggleton. *Inductive logic programming.* Academic Press, 1992.

[Mug93] S. Muggleton. Inductive logic programming: derivations, success and shortcomings. In *Proc ECML'93*, Seiten 21–39, 1993.

[Mye79] G. J. Myers. *The art of software testing business data processing.* John Wiley and Sons, 1979.

[Nai85a] L. Naish. Automatic Control for Logic Programs. In *Journal of Logic Programming 2, 3*, Seiten 167–183, 1985.

[Nai85b] L. Naish. *Negation and Control in PROLOG.* Lecture Notes in Computer Science 238, 1985.

[Nai86] L. Naish. Negation and Quantifiers in NU-PROLOG. In *Proc. Third International Conference on Logic Programming*, Seiten 624–634. Lecture Notes in Computer Science 225, 1986.

[NCdW96] S-W. Nienhuys-Cheng und Ronald de Wolf. *Foundations of Inductive Logic Programming.* Springer Verlag, 1996.

[NF92] H. Nilsson und P. Fritzson. Algorithmic Debugging of lazy functional languages. In *Proc PLILP'92*, 1992.

[Nil97] Jörg Nilson. *NGUIDE – Programmierhandbuch.* Institut für Mathematische Maschinen und Datenverarbeitung, Lehrstuhl für Programmier- und Dialogsprachen sowie ihre Compiler. Friedrich-Alexander-Universität Erlangen-Nürnberg, 1997.

[Nis98] C. Niss. Spezifikation und Implementierung einer grafischen Benutzeroberfläche für das Prolog-Debugging-System IDTS. Studienarbeit, Institut für Mathematische Maschinen und Datenverarbeitung, Lehrstuhl für Programmier- und Dialogsprachen sowie ihre Compiler. Friedrich-Alexander-Universität Erlangen-Nürnberg, 1998.

[NK97] J. Nilson und G. Kókai. PECG-GUI - A Graphical User Interface for an ECG Classifier System. In G. Brewka, C. Habel und B. Nebel, Hrsg., *KI-97: Advances in Artificial Intelligence, LNAI 1303*, Seiten 397–400, Freiburg, Germany, 1997. Springer-Verlag.

[NM90] U. Nilson und J. Maluszynski. *Logic, Programming and Prolog.* John Wiley & Sons Ltd, 1990.

[Nut85] D. Nute. A programming solution to certain problems with loops in Prolog. In *ACM SIGPLAN Notices Vol 20, No. 8.*, 1985.

[OB88] J. Thomas Ostrand und Marc J. Balker. The Category-Partition Method for Specifying and Generating Functional Tests. In *CACM 31:6*, Seiten 676–686, June 1988.

[Par85] D. L. Parnas. Software aspect of stategic defense systems. In *Communications of the ACM 28(12)*, Seiten 1326–1335, 1985.

[PBB+78] L. M. Pereira, D. L. Bowen, L. Byrd, F.C.N. Pereira und D.H.D Warren. *User's Guide to DEC-System 10 Prolog*, 1978.

[PBS92] M. Pazzani, C. Brunk und G. Silverstein. An information based approach to integrating empirical and explanation based learning. In *Proc. of Inductive Logic Programming*, London, 1992. Academic Press.

[PC88] L. M. Pereira und M. Calejo. A Framework for Prolog Debugging. In *Proc. JICSLP'88*, 1988.

[PC91] L. M. Pereira und M. Calejo. Declarative Source Debugging. In *In. Proc of the 5th Portuguese AI Conference*. Springer Verlag, 1991.

[PDA93] L. M. Pereira, C.V. Damasio und J.J Alferes. Debugging by diagnosing assumption. In *Proc AADEBUG'93*, Linköping, Sweden, 1993. LNCS 749.

[Per86] L. M. Pereira. Rational Debugging in Logic Programming. In *Proc of the Third Logic Programming Conference*, Seiten 203–210, London, England, July 1986.

[PGH94] J. Paakki, T. Gyimóthy und T. Horváth. Effective Algorithmic Debugging for Inductive Logic Programming. In *Proc. of the Fourth International Workshop on Inductive Logic Programming, ILP'94*, Seiten 175–195, Bonn, Germany, 1994.

[PGH95] J. Paakki, T. Gyimóthy und T. Horváth. Independent And-Parallelization of Logic Programs Using Static Slicing. In *Proc. of SPLST'95,*, Seiten 302–312, Visegrád, Hungary, 1995.

[Plo70] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence, 5*, Seiten 153–163. Edinburgh Univ. Press, 1970.

[Qui87] Quintus Computer System, Inc. *QUINTUS PROLOG User Guide and Reference Manual*, 1987.

[Qui88] J. Quinlan. Learning logical definitions from relations. In *Proc Machine Learning, 5(3)*, Seiten 239–266, 1988.

[Rae92] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.

[RB90] L. De Raedt und M. Bruynooghe. Indirect relevance and bias in inductive concept learning. In *Knowledge Acquisition 2*, Seiten 365–390, 1990.

[RB95] L. De Raedt und M. Bruynooghe. Towards friendly conceptlearner. In *Proc Eleventh International Joint Conference on Artificial Intelligence*, Seiten 849–854, San Mateo, CA, 1995. Morgan Kaufman.

[RGdA75] P. Roussel und University d' Axis Groupe d'Intelligence Artificielles, U.E.R. de Luminy. *Prolog, Manuel de Reference et d'utilisation*, 1975.

[RLD93] L. De Raedt, N. Lavrac und S. Dzeroski. Multiple predicate learning. In *Proc of the 13th International Joint Conference an Artificial Intelligence.* Morgan Kaufmann, 1993.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. In *Journal of the ACM 12*, Seiten 23–41, 1965.

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Book Company, 1967.

[Rou92] C. Rouveirol. Extension of inversion of resolution applied to theory completion. In *Proc. of Inductive Logic Programming*, London, 1992. Academic Press.

[RS82] J. Robinson und E. Sibert. LOGLISP: Motivation, Design and Implementation in Logic Programming. In Clark-Tärnlund, Hrsg., *Logic Programming*, Seiten 299–314. Academic Press, 1982.

[SB86] C. Sammut und R. Banerji. Learning concept by asking questions. In *Machine Learning: An Artificial Intelligence Approach, volume II*, San mateo, CA, 1986. Morgan Kaufmann.

[Sch71] Jacob T. Schwarz. An overview of bugs. In *Debugging Techniques in Large Systems*, Seiten 1–16. Prentice-Hall, 1971.

[Sev87] R. E. Seviora. Knowledge based program debugging system. In *IEEE Software 4(3)*, Seiten 20–32, 1987.

[SH89] P. Strooper und D. Hoffman. Prolog Testing of C modules. In *ILPS'91*, Seiten 596–610, 1989.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[Sha91] N. Shahmehri. Generalized Algorithmic Debugging. Dissertation, University of Linköping No 297, 1991.

[SKF93] N. Shahmehri, M. Kamkar und P. Fritzson. Usability Criteria for Automated Debugging Systems. In *Proceeding of International Workshop AADEBUG'93*, Seiten 16–24, Linköping, Sweden, 1993. LNCS 749, Springer.

[Sko90] E. Skordalakis. ECG Analysis in Syntactic and Structural Pattern Recognition. In *Theory and Applications*, Seiten 499–533. World Scientific, 1990.

[SMAU94] E. Sommer, K. Morik, J-M. Andre und M. Uszaynski. What online machine learning can do for knowledge acquisition a case study. In *Knowledge Aquisition 6*, Seiten 435–460, 1994.

[SS86] L. Sterling und E. Shapiro. *The Art of Prolog.* MIT-Press, 1986.

[SSP85] J. C. Spohrer, E. Soloway und E. Pope. A goal/plain analysis of buggy Pascal programs. In *Human-computer Interaction 1(2)*, Seiten 163–207, 1985.

[ST90] O. Shmueli und S. Tsur. Logical diagnosis of LDL programs. In *Proc ICLP'90*, Seiten 112–129, 1990.

[Tip94] F. Tip. A Survey of Program Slicing Techniques. In *Report Computer Science Department of Software Technology CS-R9438*, 1994.

[TS84] H. Tamaki und T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proceedings of the Second International Logic Programming Conference*, Seiten 127–138, Uppsala University, Uppsala, Sweden, 1984.

[TS85] H. Takahashi und E. Shibayama. PRESET - A Debugging Environment for Prolog. In *Proc of the International Conference on Logic Programming, Tokyo*, Seiten 90–99, 1985.

[TZ86] J. A. Thom und J. A. Zobel. *NU-Prolog 1.0 Reference Manual Machine Intelligence Project.* Technical Report 86/10 Department Computer Science, University of Melbourne, 1986.

[Var93] A. Varsek. Inductive logic Programming with Genetic Algorithms. Ph.d thesis, Faculty of Electrical Engineering and Computer Science University of Ljubljana, Slovenia, 1993.

[vEK76] M.H. van Emden und R. Kowalski. The semantic of Predicate Logic as a Programming Language. In *Jorunal of ACM 23*, Seiten 733–742, 1976.

[Ves85] I. Vessey. Expertise in Debugging Computer Programs: A Process Analysis. In *Journal of Man-Machine Studies Vol 23*, Seiten 459–494, 1985.

[War83] D.H.D Warren. *An Abstract PROLOG Instruction Set*, 1983.

[Wei84] M. Weiser. Program Slicing. In *IEEE Transactions on Software Engineering SE-10, 4*, Seiten 352–357, 1984.

[Wer87] H. Wertz. Automatic Correction and Improvement of Programs. In *Artificial Intelligence*. Ellis Horwood, 1987.

[WF93] R. Westman und P. Fritzson. Graphical User Interfaces for Algorithmic Debugging. In *Proc. of the First International Workshop, AADEBUG'93 LNCS 749*, Seiten 273–286, Linköping, Sweden, May 1993. Springer-Verlag.

[WO92] R. Wirth und P. O'Rorke. Constrain for Predicate invention. In *Proc. of Inductive Logic Programming*, London, 1992. Academic Press.

[Wro88] S. Wrobel. Automatic representation adjustment in an observational discovery system. In *EWSL-88*, Seiten 253–262, London, 1988. Pitman.

# Stichwortverzeichnis