Institut für Informatik
der Technischen Universität München

Konzepte für eine wissensbasierte visuelle Sprache zur Bildanalyse

Uwe Meyer–Gruhl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. E. W. Mayr
Prüfer der Dissertation:	
1.	Prof. Dr. B. Radig
2.	Prof. Dr. A. Endres

Die Dissertation wurde am 6.11.1995 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.2.96 angenommen.

Danksagung

Die Idee zu dieser Arbeit wurde 1992 an einem warmen Frühlingstag beim Mittagessen geboren: Professor Radig meinte, man müßte eigentlich die Möglichkeit haben, einen Schwellwert mit einem Schieberegler zu verändern, um damit die Auswirkungen auf den gesamten folgenden Analyseprozeß beobachten zu können. Dieser Gedanke war für mich der Anlaß, das Thema etwas eingehender zu betrachten. Wie sich herausstellte, steckte mehr dahinter, als sofort offensichtlich war.

Ich möchte mich an dieser Stelle zuallererst bei meinen Doktorvätern bedanken:

Prof. Dr. Bernd Radig der die eigentliche Idee lieferte, immer ein offenes Ohr hatte und mir größtmöglichen Spielraum ließ.

Prof. Dr. Albert Endres der bereit war, als zweiter Doktorvater zu fungieren und viele wertvolle Anregungen gab.

Weiterhin gilt mein Dank:

Dr. Wolfgang Eckstein ohne den es HORUS nicht gäbe und der bis morgens um vier mit mir diskutierte.

Dr. Olaf Munkelt der mich gerade zur rechten Zeit ermahnte und mit dem ich viel gelacht habe.

Boris Pasternak der *nicht* „Dr. Schiwago“ geschrieben hat, sondern wesentliche Gedanken beitrug und andere Blickwinkel eröffnete.

Andreas Wastl der bei der Implementierung des Systems wirklich großartige Arbeit geleistet hat.

Christiane Meyer–Gruhl die mir während der letzten viereinhalb Jahre vieles vom Leib hielt, obwohl sie selbst mitten im Berufsleben steht.

Ich bedanke mich ferner bei meinen Kolleginnen und Kollegen der Forschungsgruppe Bildverstehen, die in vielen Diskussionen durch Anregungen und Kritik zum Gelingen dieser Arbeit beigetragen haben.

München, im November 1995

Kurzfassung

Inhalt der vorliegenden Arbeit sind Dialog- und Sprachkonzepte zur Bildanalyse, mit denen Werkzeuge zur einfacheren Lösung von Bildanalyseproblemen im industriellen Bereich realisiert werden können. In diesem Rahmen wurde eine wissensbasierte visuelle Sprache mit objektorientierten Mitteln entwickelt und als Teil eines umfassenden Bildanalysesystems implementiert.

Ausgehend vom Gedanken des *Computer Aided Vision Engineering* besteht der prinzipielle Ansatz darin, die untere Ebene der Algorithmen zur Bildanalyse auszuklammern, um somit ohne Ballast und ausschließlich mit vorhandenen Operatoren Problemstellungen explorativ angehen zu können. Hierbei werden dem Benutzer Unterstützungsfunktionen auf verschiedenen Stufen angeboten. Diese umfassen u.a. die vollständige grafische und problembezogene Parametrierung von Operatoren. Grundlage hierfür ist die Verwendung von Informationen aus einer Wissensbasis.

Die visuelle Darstellung des Programms vereinfacht die Findung von Lösungen nach dem Prinzip der schrittweisen Verfeinerung. Gleichermaßen wird das Verstehen bestehender Programme durch den hohen Abstraktionsgrad einer visuellen Darstellung erleichtert. Anders als in herkömmlichen textuellen Programmiersprachen muß der Datenfluß nicht durch die Referenzierungen von Variablen im Programmtext analysiert, sondern kann direkt visuell erfaßt werden.

Die Wiederverwendbarkeit von erstellten Lösungen wird durch sogenannte *Moleküle* erzielt.

Das Zusammenwirken der vorgestellten Konzepte ermöglicht sowohl eine experimentelle Annäherung von Anfängern an das ansonsten eher durch fest vorgegebene Lösungen geprägte Gebiet Bildanalyse als auch eine wesentliche Beschleunigung bei der Erstellung von Bildanalyseapplikationen für Experten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Arbeit ist thematisch an der Schnittstelle zwischen objektorientierten Methoden, Bildanalyse und visuellen Sprachen angesiedelt (siehe Abbildung 1.1). Das erschwert eine vollständige Diskussion von vorhandenen Ansätzen in jedem Teilbereich. Deshalb verfolgt diese Arbeit die Hauptaspekte und verweist im übrigen auf die entsprechenden Literaturstellen. Bei einigen davon handelt es sich schon um Überblicksbeiträge, die der interessierte Leser am angegebenen Orte finden kann und die hier nur kurz zusammengefaßt sind. Ein Vergleich existierender Systeme findet sich in Kapitel 4.

1.1 Computer Aided Vision Engineering

In [REK+92] wird dargelegt, daß das Hauptproblem in der industriellen Bildanalyse heute nicht mehr darin besteht, neue Verfahren zu entwickeln, d.h. den Hunderten von Filteroperationen zum Auffinden von Kanten eine weitere hinzuzufügen, sondern darauf, die vielen aus der Forschung bekannten Verfahren sinnvoll einzusetzen.

Hier soll explizit nicht die Notwendigkeit einer Grundlagenforschung im Bereich Bildanalyse angezweifelt werden, lediglich die Erfordernis, bei *jedem* neu auftauchenden Problem in der Anwendung neue Verfahren erarbeiten zu müssen. Mehr und mehr setzt sich die Erkenntnis durch, daß die traditionelle Methode, Bildanalyseprobleme in der Praxis zu lösen, viel zu zeit- und kostenintensiv ist.

Prinzipiell sind für dieses Problem zwei mögliche Lösungswege aufgezeigt worden: zum einen die weitestgehend automatische Konfiguration von Bildanalysesystemen, zum anderen die bestmögliche Unterstützung des Benutzers durch grafische Oberflächen und große Bibliotheken (siehe auch Abschnitt 2.3).

An sich ist in diesem Zusammenhang der Gedanke von Software-Wiederverwendung sehr naheliegend, allerdings gibt es nicht viele Gebiete, bei denen — wie in der

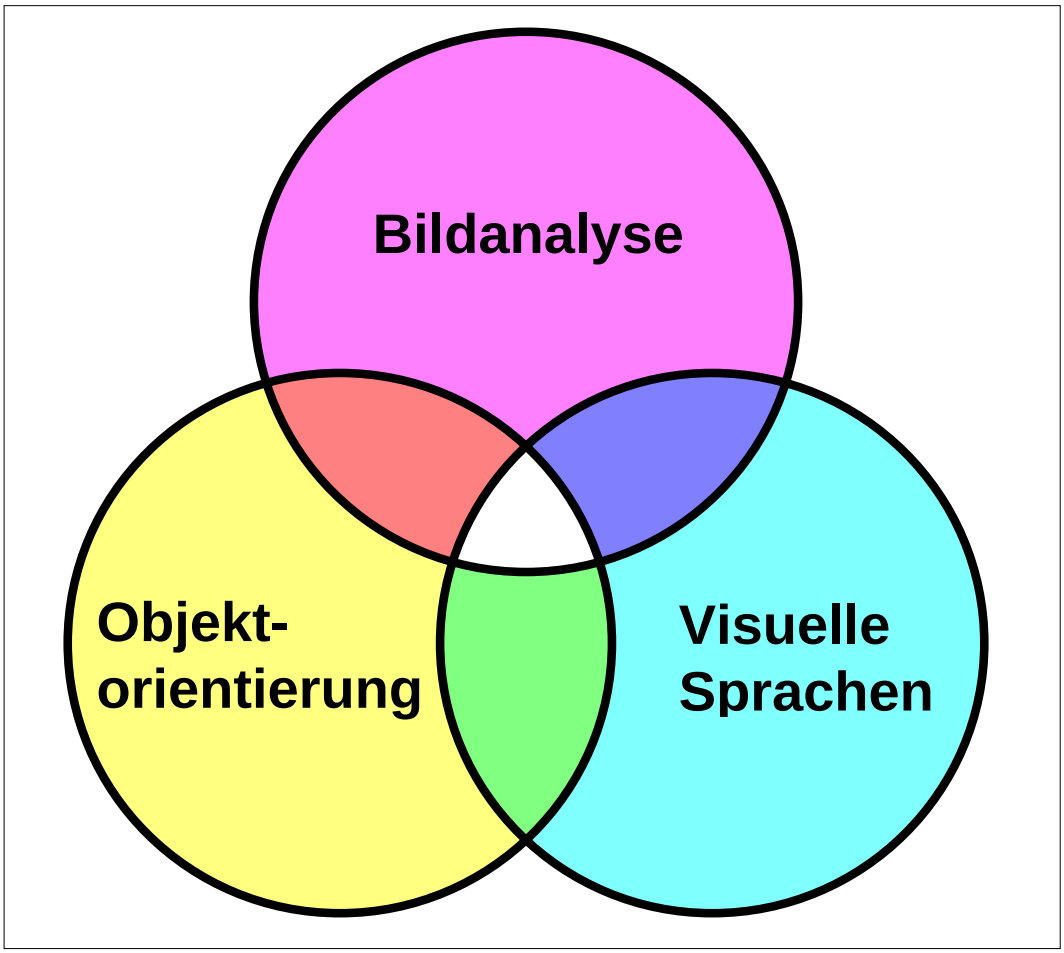


Abbildung 1.1: Themenkomplexe der Arbeit

Bildanalyse — derartig viele Funktionen vorhanden sind und bei denen eine systematische Behandlung dieser Vielzahl ebenso schwierig zu bewerkstelligen ist[1]. Ein Grund für die Schwierigkeiten liegt darin, daß praktisch kein Ansatz in der Vergangenheit objektorientiert war. Mittels objektorientierter Methoden läßt sich Wiederverwendung erheblich vereinfachen. Leider stellt die Umstellung von inhärent nicht objektorientiert geplanten Systemen einen großen Aufwand dar, wie in Abschnitt 3.9 noch gezeigt wird.

Die Erfahrung zeigt, daß sich die überwiegende Mehrzahl von Anwendungen mit bestehenden Verfahren realisieren ließen, wenn die nötige Transparenz vorhanden wäre. Gemeint ist hier das Wissen um die Anwendungsmöglichkeiten der vorhandenen Operationen, d.h. der Zugang zu ihren spezifischen Wirkungsweisen und ihre

[1] Andere Anwendungen, bei denen ähnliche Voraussetzungen gegeben sind, sind etwa Prozeßsteuerungsanlagen, Statistiksysteme und allgemein auch Mathematikpakete wie die in den letzten Jahren aufgekommenen Mathematica [Wol91] und Maple.

Anwendbarkeit auf gegebene Problemstellungen.

Dieser Gedanke des *Vision Engineering* wurde in [Jai88] entwickelt und fand mit *Computer Aided Vision Engineering* (CAVE) seine logische Fortsetzung [REK+92], indem die benötigten Komponenten mit Unterstützung des Computers realisiert werden, wie es allgemein bei C-Techniken üblich ist.

Einige Bildanalysesysteme verfügen zudem über einen zu kleinen Umfang an Operationen, sind also zu speziell auf bestimmte Problembereiche abgestimmt (z.B. Systeme zur Satellitenbildauswertung), oder stellen, wenn sie über einen ausreichenden Funktionsumfang verfügen, diesen nur in Form von riesigen, unübersichtlichen Bibliotheken zur Verfügung, so daß potentielle Anwender einen hohen Aufwand zur Einarbeitung haben.

Die Entwicklung in Richtung großer Software-Bibliotheken kam anfangs mit Systemen wie Spider [SPI83] in Gang und wurde durch Nachfahren wie Khoros [RA91], KBVision [Wil90] und andere verstärkt.

Viele Anwender waren der Ansicht, daß diese eingeschlagene Richtung nicht erfolgversprechend war, so daß danach eine Art „Gegenströmung“ entstand, nach dem Prinzip: ”Small is beautiful”. Das System Vista [SA90] ist ein gutes Beispiel dafür[2]. Der Benutzer muß bei diesem System auf großen Funktionsumfang verzichten, dagegen ist sein Einarbeitungsaufwand auch entsprechend niedrig. Diese Vorgehensweise führt natürlich dazu, daß Lösungen immer wieder neu gesucht werden müssen.

Ein weiterer wichtiger Grund für die Notwendigkeit von Werkzeugen für die Bildanalyse liegt darin, daß mit dem Verfügbarwerden von schnellen günstigen Universalrechnern heute die Rechenleistung für Bildanalysezwecke auch im nichtprofessionellen Bereich ausreichend ist. Darüber hinaus steht immer öfter auch die Zusatz-Hardware zur Verfügung, etwa, weil Multimedia-Hardware wie ein Framegrabber mit angeschlossener Kamera fast schon zur Grundausstattung eines heutigen Personalcomputers gehört.

So stehen viele potentielle Nutzer aus den verschiedensten Gebieten jetzt vor der Entscheidung, Bildanalysetechnologie in Ihrem Anwendungsbereich einzusetzen. Meist sind diese Nutzer aber reine Anwender, die keine oder wenig Informatikkenntnisse mitbringen und weder Erfahrung mit den üblichen Herangehensweisen noch mit dem Vokabular der Bildanalyse haben.

Während in der Vergangenheit durch spezialisierte Firmen oder Personen Bildanalyselösungen für industrielle Anwendungen erstellt wurden, die dem entsprechend teuer waren, geht der Trend heute eindeutig in Richtung von Standardwerkzeugen, mit denen der Anwender selbst seine Probleme lösen kann. Speziell im Bereich Bildanalyse ist aber neben dem bisherigen Fehlen solcher Werkzeuge auch ein Mangel

[2] Gedacht war Vista eigentlich als System mit einer sehr einfachen Schnittstelle, allerdings hatte es von Beginn an auch einen recht geringen Funktionsumfang, was von vielen Benutzern als Hauptvorteil angesehen wird. Zitat: „Da muß man nicht stundenlang Handbücher wälzen.“

an allgemeinverständlicher Umsetzung von Nomenklatur und Wissen über Verfahrensweisen zu beklagen. Auf der anderen Seite ist in Abwesenheit von wirklichen Durchbrüchen eine stärkere Mathematisierung des Gebiets zu verzeichnen. So erfolgt zur Zeit eine Aufspaltung der Disziplin Bildanalyse in Theorie und Praxis.

1.2 Rapid Prototyping

Von besonderer Wichtigkeit für die Akzeptanz eines Bildanalysesystems ist die Möglichkeit zur einfachen, intuitiven Bedienung mit dem Ziel, schnelle Entwicklung von prototypischen Bildanalyseapplikationen zu unterstützen. Ohne eine geeignete grafische Oberfläche ist es heute nicht mehr praktikabel, ein komplexes System zur Bildanalyse in dieser Weise zu benutzen [ELM+93].

Für das in dieser Arbeit eingesetzte Bildanalysesystem HORUS (siehe Absatz 5.1.1) gibt es seit einiger Zeit eine Oberfläche namens HDIALOG [Mül92], deren Nachfolger HDEVELOP [Man93] es erlaubt, Rapid Prototyping im Bereich der Bildanalyse durchzuführen. Der Benutzer kann nach Kapiteln gegliedert auf die HORUS-Operatoren zugreifen und bekommt Vorgabeparameter geliefert, die er natürlich verändern kann. Die bei den einzelnen Schritten entstehenden Zwischenergebnisse werden wie in Abbildung 1.2 dargestellt und das jeweils aktuelle wird im aktiven Visualisierungsfenster ausgegeben. Diverse Parameter, wie Darstellungsfarben oder Darstellungsform (konvexe Hülle, Regionenrand usw.) sind bequem einstellbar.

In dieser Arbeit wurde in `Smalltalk` eine neue Oberfläche mit dem Namen HCANVAS erstellt, die noch stärker grafisch orientiert ist und soweit sinnvoll von der textuellen Ebene der Programmierung wegführt. Vorläufer dieses Ansatzes waren diverse Arbeiten der Forschungsgruppe Bildverstehen am Institut für Informatik der Technischen Universität München, u.a. [Kne91, Lob91, Wäh91, Aue92, Mül92, Sud93, Man93, Eur94].

Die HORUS-Operatoren werden für `Smalltalk` *funktional* aufbereitet, d.h. möglichst ohne Seiteneffekte. Der Nebeneffekt der Vererbungsmechanismen besteht in der Möglichkeit zur Überdefinition der meisten Methoden, so daß Bildfolgen oder Farbbilder mit dem selben Operator gefiltert werden können wie Einzelbilder.

Die HORUS-Operatoren (*Atome*[3]) werden dann als grafische Elemente (Knoten) dargestellt, während die Kanten zwischen den Elementen den Datenfluß repräsentieren[4]. Zudem werden Datenquellen für die verschiedenen Eingabeobjekte (z.B. Ganzzahlen: Schieberegler, Listen: Optionsfelder, Bilder: Dateiselektor) und Datensenken zur Visualisierung (z.B. Grafikfenster) verwendet. Die Visualisierer für

[3] *Atom* bezeichnet eine in `C` implementierte HORUS-Basisoperation. Siehe auch Abschnitt 5.5.

[4] Ein aus Operatoren bestehendes Programm bildet dann einen Graphen, also ein Netz aus Knoten und Kanten.

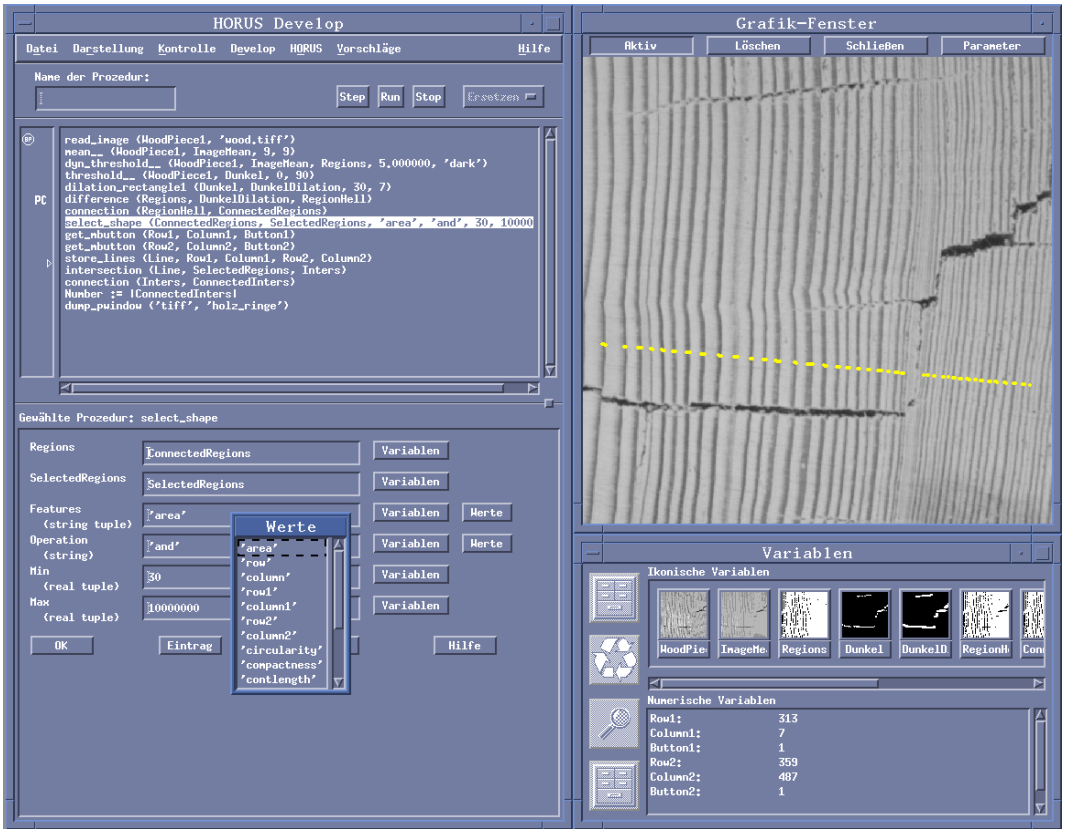


Abbildung 1.2: Oberfläche HDEVELOP

die verschiedenen Datentypen können an jede gewünschte Kante „angehängt“ werden. Während der Programmerstellung werden Veränderungen an Parametern sofort sichtbar gemacht, um eine interaktive Rückkopplung zu gewährleisten.

Der Benutzer wird auf verschiedenen Ebenen bei der Auswahl und Parametrierung unterstützt. Einige Kriterien lassen sich bereits lokal, andere aber nur global entscheiden. Wenn in einem längeren Segmentierungsprozeß Grauwertbilder auf Regionendaten abgebildet werden sollen, ist a priori nicht klar, an welcher Stelle konkret dieser Übergang stattfinden soll. Solche Probleme können mittels einer Planungskomponente gelöst werden, die sich modular in die Oberfläche einfügen läßt[5]. Lokale Restriktionen können statisch, wie Anzahl und Datentyp der Parameter oder dynamisch, wie Maximum und Minimum von Schwellwerten in Abhängigkeit vom Pixeltyp (long oder byte) eines Bildobjektes sein. Aus der HORUS-Operator-Datenbasis[6] werden auch Informationen über die Wirkungsweise von Parametern abgelei-

[5]Ein anderes System, das eine automatische Konfiguration und Parametrierung von Folgen von Bildanalyseoperatoren zum Ziel hat, ist in der Forschungsgruppe Bildverstehen entstanden [Mes92].

[6]Wissensbasis, in der semantische Informationen, z.B. über Anzahl, Namen, Art und Vorgabewerte von Parametern für Operatoren abgelegt sind.

tet, z.B. ob der Wirkungszusammenhang eher linear oder logarithmisch ist, um so zu „zielorientierten“ Parametern zu kommen.

Schließlich sollen mit diesem grafischen Editor erstellte Programm-Module nicht nur als ablauffähige Programme abgelegt werden können, sondern als *Moleküle*[7] wieder in die HORUS-Operator-Datenbasis abgelegt werden, so daß Operationen auf höherer Abstraktionsebene erzeugt und später auch wiederverwendet werden können. Das Wissen über die erzeugten Moleküle soll dabei, soweit möglich, aus dem Wissen über die verwendeten HORUS-Atome propagiert werden und wo dies nicht möglich ist, vom Programmierer spezifiziert werden.

Die diesbezüglichen Informationen werden der HORUS-Operator-Datenbasis entnommen, bzw. auch wieder eingestellt. Bei der Eigenerstellung von „Molekülen“ werden die Informationen über die Semantik der Parameter (Wirkungsweise, Restriktionen, Voreinstellungen) wieder in der Datenbasis abgelegt. Diese Wissensquelle sollte deshalb mehrstufig repräsentiert sein:

- *fest* für HORUS-Atome
- *quasi-fest* für Moleküle, die zum Standardsatz gehören
- *variabel* für Moleküle, die von einer Arbeitsgruppe definiert wurden
- *variabel* für Moleküle, die benutzerspezifisch sind
- *temporär* für Moleküle, die nur innerhalb einer Sitzung definiert werden

Bestimmte Daten, wie Restriktionen, können dabei propagiert werden, andere, wie der wahrscheinliche Wirkungszusammenhang mit gegebenen Parametern, müssen vom Benutzer vorgegeben werden.

Durch dies Konzept wird es ebenfalls möglich, Programme für andere Zielsysteme zu konfigurieren, indem auf Basis von HORUS-Atomen die Elementaroperationen des Zielsystems modelliert, mit Informationen wie Ausführungszeiten oder Beschränkungen versehen und in der HORUS-Wissensquelle nur diese neudefinierten Operatoren als gültig markiert werden. Die automatische Planung kann dann nur auf die Operationen des Zielsystems zurückgreifen, um einen Bildanalyseprozeß zu konfigurieren.

Ein weiterer Aspekt des Ansatzes ist in der wissensbasierten Benutzerunterstützung und -führung zu sehen (siehe Kapitel 6). Auch in diesem Bereich wurden Vorarbeiten in der Forschungsgruppe geleistet [Klo93].

[7] *Molekül* bezeichnet die Zusammenfassung von mehreren Atomen oder anderen Molekülen, d.h. das Äquivalent einer Prozedur oder eines Makros. Siehe auch Abschnitt 5.5.

Kapitel 2

Problemstellung

In diesem Kapitel wird die prinzipielle Vorgehensweise bei Bildanalyseaufgaben vorgestellt und die Problemklasse anhand einiger typischer Beispiele eingegrenzt. Die hier vorgestellten Anwendungen wurden mit HORUS programmiert, für eine kurze Einführung in HORUS siehe Abschnitt 5.1.1.

2.1 Vorgehensweise bei Bildanalyseaufgaben

Zum besseren Verständnis der grundsätzlichen Vorgehensweise bei Bildanalyseanwendungen wird in Abbildung 2.1 dargestellt, wie ein typischer Analysevorgang abläuft. Hier folgt das fragmentarische Programmlisting:

```
read_image(&Obj1,"bwlady.tiff");
open_window(0,0,256,256,0,"","");
disp_image(Obj1);
threshold(Obj1,&Obj2,0.0,55.0);
connection(Obj2,&Obj3);
closing_circle(Obj3,&Obj4,18.0);
select_shape(Obj4,&Obj5,"area",250.0,2000.0);
select_shape(Obj5,&Obj6,"anisometry",1.0,2.3);
disp_region(Obj6);
```

Es geht bei diesem bewußt einfach gewählten Beispiel darum, die Augen der Person auf dem Bild zu finden. Im ersten Schritt wird das Bild eingelesen. Anstelle einer Dateioperation (`read_image`) könnte hier auch der Zugriff auf einen Framegrabber erfolgen. Der nächste relevante Schritt ist es, die dunklen Teile des Bildes mittels einer Schwellwertoperation (`threshold`) zu extrahieren, wobei gleichzeitig von der ikonischen Ebene auf die Regionenebene übergegangen wird. Nun müssen Eigenschaften der resultierenden Regionen zur weiteren Unterscheidung herangezogen werden, was jedoch nur möglich ist, nachdem die sich ergebende Region in ihre

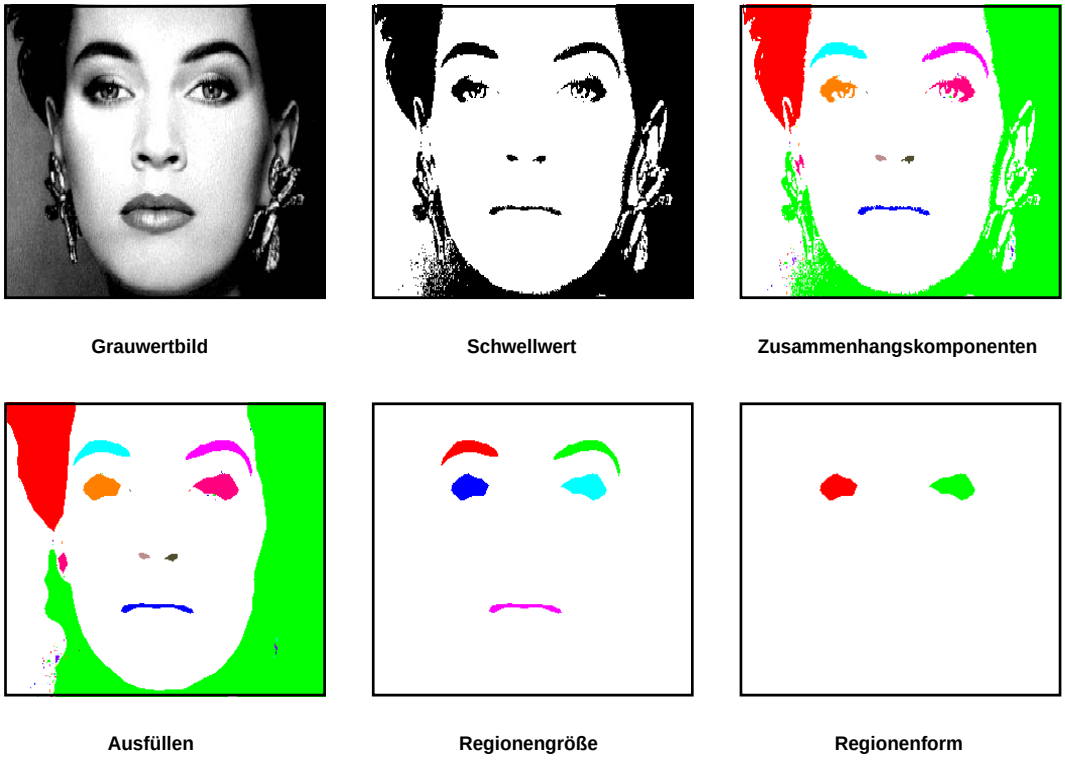


Abbildung 2.1: Beispiel eines typischen Analysevorgangs

Zusammenhangskomponenten zerlegt worden ist (`connection`). Da die Regionen sehr stark „ausgefranst“ sind, werden sie mittels einer morphologischen Operation (`closing_circle`) geglättet.

Zunächst werden dann zu kleine und zu große Regionen entfernt (mittels `select_shape`(”area”)), schließlich die verbleibenden Regionen nach der Form (`select_shape`(”anisometry”)) selektiert. Die übrigen im Listing aufgeführten Operationen dienen lediglich zur Visualisierung des Originalbildes und der segmentierten Regionen.

Obwohl das Beispiel sehr einfach ist, zeigt es doch bestimmte Elemente, die auf viele Aufgaben im Bildanalysebereich anwendbar sind (siehe auch [BB82, Jäh91, VS91, Hab89, LEM+93]). Typischerweise wird im Bildanalyseprozeß eine bestimmte Abfolge von Arbeitsschritten eingehalten, nämlich (siehe auch Abbildung 2.2):

1. Vorverarbeitung, beispielsweise Bildverbesserungsoperationen wie Anpassung der Bildhelligkeit oder Rauschfilter.

2. Kantenverstärkung, also Filterung mit Hochpaßfiltern.

3. Segmentierung, d.h. Übergang von der Pixelebene zu Regionen.

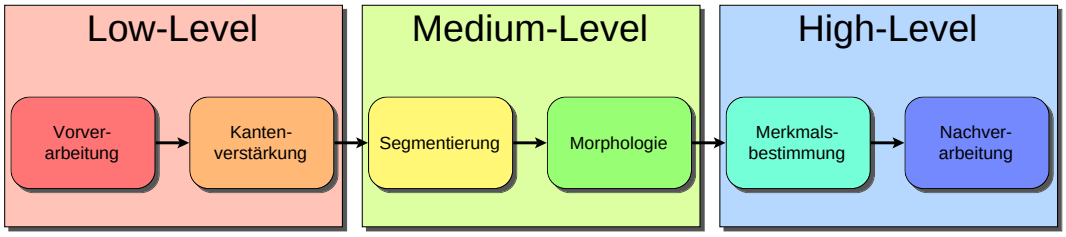


Abbildung 2.2: Gliederung des Bildanalyseprozesses

4. Morphologie, d.h. regionenorientierte Nachverarbeitung.
5. Merkmalsbestimmung von gefundenen Regionen.
6. Nachverarbeitung, d.h. Selektion von interessanten Regionen, Herstellung von Beziehungen zwischen Regionen.

Bei einer solchen Vorgehensweise wird nur bis zur Segmentierung mit Low-Level-Bilddaten gearbeitet und später mit ikonischen Bilddatenstrukturen höherer Ordnung, Regionen oder linienhaften Repräsentationen wie *Extended Line Descriptions*[1] (XLD). In realen Anwendungen kann jeder einzelne Schritt auch weggelassen werden (z.B. wurde beim Finden der Augen keine Vorverarbeitung benötigt), bzw. können einzelne Schritte mehrfach angewendet werden (im Beispiel die zweifache Anwendung der Selektion). Selbst eine Vertauschung der Reihenfolge ist möglich und mitunter sinnvoll.

2.2 Typische Bildanalyseanwendungen

Um die in dieser Arbeit betrachtete Klasse von Bildanalyseproblemen einzugrenzen, ist es nötig, einige typische Beispiele aus verschiedenen Anwendungsgebieten darzustellen. Es handelt sich zumeist um „real-world“-Anwendungen, die im Lauf der Jahre von verschiedenen Firmen und Organisationen an die Forschungsgruppe Bildverstehen herangetragen wurden. Aufgaben der im folgenden beschriebenen Art dienen im Rahmen eines Bildanalysepraktikums an der Technischen Universität München der Informatik-Ausbildung.

2.2.1 Beispiel 1: Industrielle Qualitätskontrolle

Bei dem hier gewählten Beispiel geht es darum, Bondierungspunkte zu finden. In einem weiteren Schritt könnte dann festgestellt werden, ob die Bondierung an den hier gefundenen Stellen (*Regions-Of-Interest*, ROIs) einwandfrei ist.

[1] HORUS-Datentyp, der Linien darstellt.

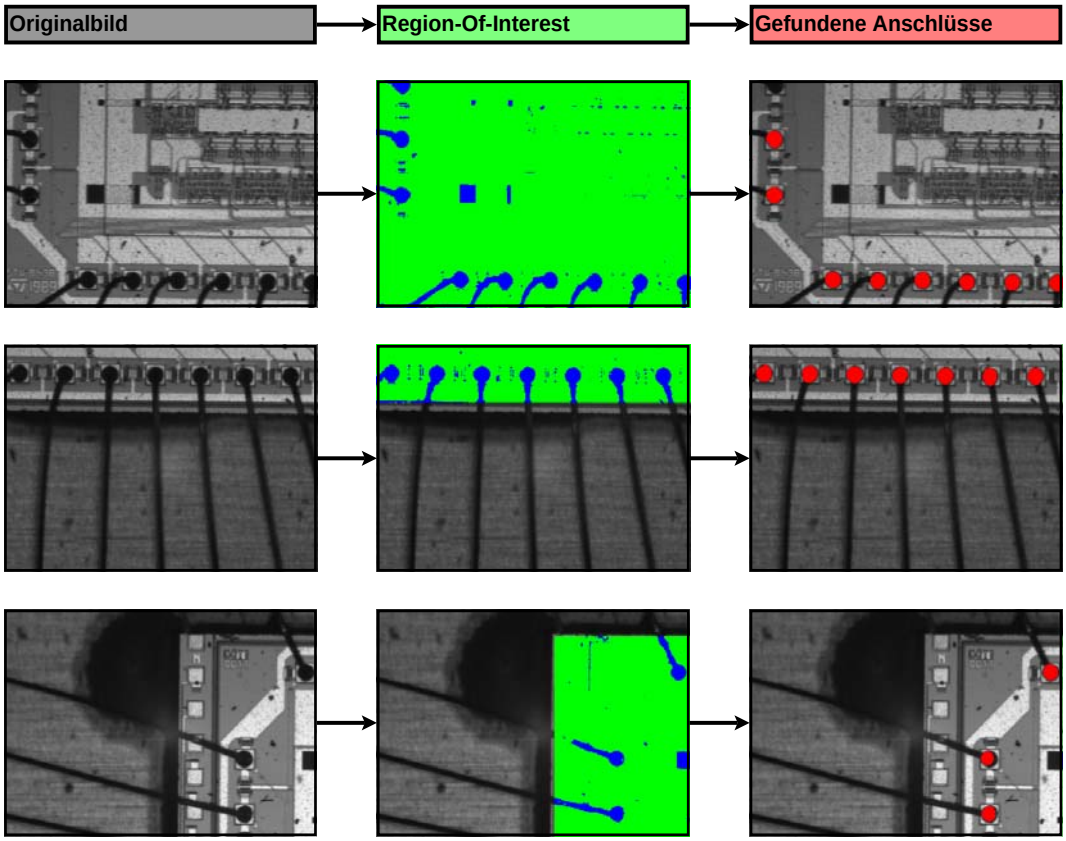


Abbildung 2.3: Prüfen von Bondierungsstellen

```
read_image(&Bild,"die7");
threshold(Bild,&Leitb,150.0,255.0);
position(Leitb,&Dummy,&Dummy,&Dummy,&L1,&C1,&L2,&C2);
rectangle1(&ROI,L1,C1,L2,C2);
reduce_domain(Bild,ROI,&Bereich);
threshold(Bereich,&Dunkel,0.0,40.0);
connection(Dunkel,&Einzel);
select_shape(Einzel,&Keinviereck,
             "compactness",2.0,99999.0);
opening_circle(Keinviereck,&Kreise,13.5);
connection(Kreise,&Anschluesse);
```

Wie im Listing zu sehen ist, läßt sich auch dies recht komplexe Beispiel in HORUS einfach ausdrücken. In Abbildung 2.3 ist der Ablauf an drei verschiedenen Beispielaufnahmen dargestellt.

Zunächst werden im eingelesenen Bild (`read_image`) die hellen Leiterplatten segmentiert (`threshold`), dann das umschließende Rechteck gebildet, um die *Region-Of-Interest* (ROI) zu finden, in der die gesuchten Anschlußpunkte nur liegen

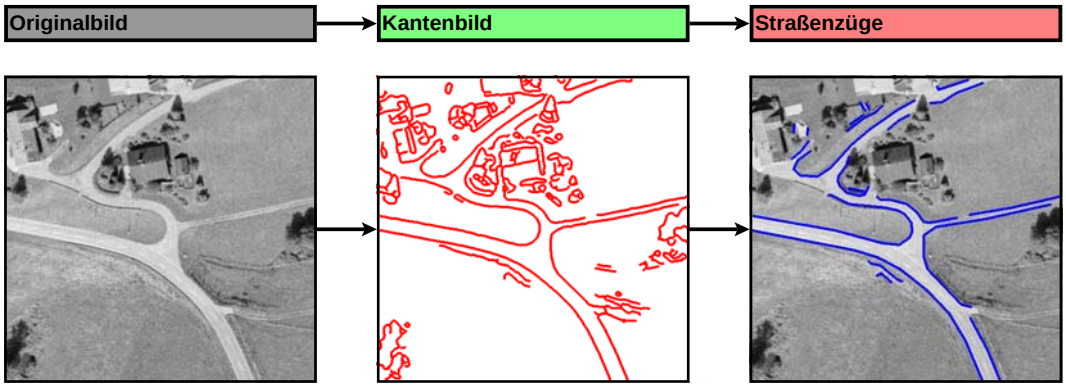


Abbildung 2.4: Finden von Straßenzügen

können (`position`, `rectangle1`, `reduce_domain`). In dieser Region werden dann alle dunklen Bereiche gesucht, die in Abbildung 2.3 in der mittleren Spalte blau dargestellt sind (`threshold`). Aus den separierten Objekten (`connection`) werden dann diejenigen ausgewählt, die nicht kompakt sind (`select_shape`), um die rechteckigen Markierungen auszuschließen. Schließlich wird mittels Morphologie (`opening_circle`) die Verdickung selektiert, die die Anschlußstelle markiert.

2.2.2 Beispiel 2: Luftbildauswertung

In diesem Beispiel geht es darum, Luftbilder für die Nachführung und Korrektur von Karten auszuwerten. Hierzu müssen zunächst Straßen und Gebäude identifiziert werden.

```
read_image (&Image, "mreut");
edges(Image, &Amplitude, &Direction,
      "mderiche2", 0.3, "nms", 20, 40);
threshold(Amplitude ,&EdgeRegion, 1, 255);
clip_region(EdgeRegion, &EdgeClipped, 1, 1, 510, 510);
skeleton (EdgeClipped, &EdgeSkel);
gen_contour2_xld(EdgeSkel, &RoadEdges, 1, "filter");
poly_xld(RoadEdges, &RamerEdges, "Ramer", 2.0);
parallels_xld(RamerEdges, &Para, 10.0, 30.0, 0.15);
mod_para_xld(Para, Image, &ModPara, &ExtPara,
             0.4, 160, 220, 10.0);
max_parallels_xld (ExtPara, &MaxPolygons);
```

Nach der Anwendung eines Kantenfilters (`edges`) und einer einfachen Segmentierung (`threshold`) wird das skelettierte (`skeleton`) Zwischenergebnis mit `gen_contour2_xld` in sogenannte *Extended Line Descriptions* (XLD) umge-

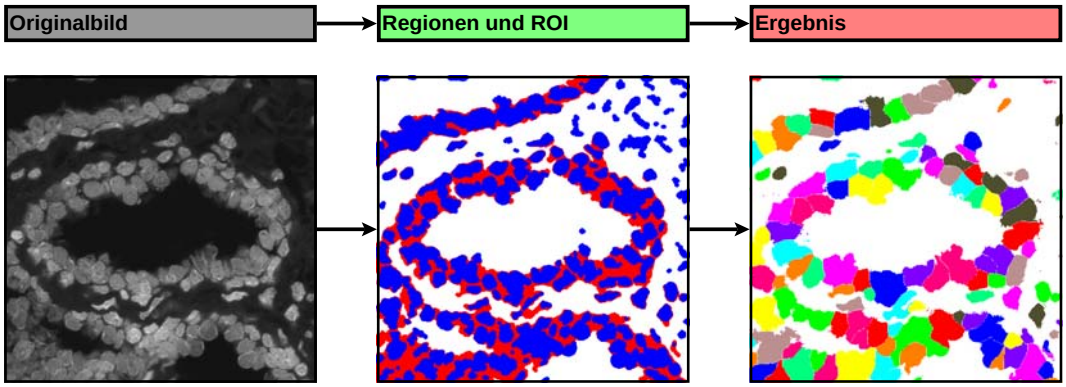


Abbildung 2.5: Finden von Zellen

wandelt. In diesen Linienbeschreibungen werden dann Parallelen gesucht, die bestimmten Kriterien bezüglich des Abstands genügen (`parallels_xld`).

2.2.3 Beispiel 3: Medizinische Bildanalyse

Neben der industriellen Qualitätskontrolle ist die Auswertung medizinischer Bilder (z.B. MR-, Ultraschall-, Tomographie-, Mikroskop- und Röntgenbilder) wohl die häufigste Anwendung für Bildanalyse. Deshalb ist die Eignung für Aufgaben in der medizinischen Diagnostik ein entscheidender Prüfstein für jedes Bildanalysesystem. Im hier gezeigten Beispiel sollen die einzelnen Zellen gefunden werden.

```
read_image(&Dru035, "dru035.tiff");
threshold(Dru035,&Region,50,255);
fill_up_shape(Region, &RegionFillUp, "area", 0.0, 100.0);
opening_circle(RegionFillUp, &RegionResultOp, 3.5);
mean(Dru035, &ImageMean, 39, 39);
dyn_threshold(Dru035, ImageMean, &DynRegs, 1.0, "light");
fill_up(DynRegs, &RegionFillUpDyn);
opening_circle(RegionFillUpDyn,&RegionFillUpDynOpen,3.5);
smooth(Dru035, &ImageMedian, "gauss", 5.0);
decompose3(ImageMedian, &Image1, &Image2, &Image3);
invert(Image1, &Invert);
watersheds(Invert, &Basins, &Watersheds);
get_domain(Basins, &Domain);
intersection(Domain, RegionFillUp, &RegionIntersection);
```

Zur Segmentation wird neben einem Gaußfilter (`smooth`) die Wasserscheidentransformation (`watersheds`) verwendet [ZM93].

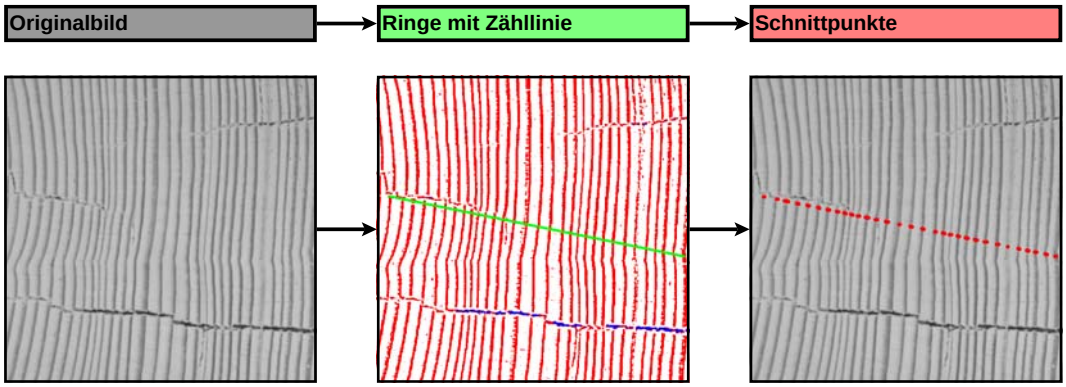


Abbildung 2.6: Zählen von Jahresringen

2.2.4 Beispiel 4: Holzwirtschaft

Die hier gezeigte Anwendung ist ein Paradebeispiel für die Mächtigkeit von HORUS. Um auf Holzschnitten das Alter von Bäumen erkennen zu können, sollen die Jahresringe automatisch gezählt werden.

```
read_image(&WoodPiece1, "wood_piece2.tiff");
mean(WoodPiece1, &ImageMean, 9, 9);
dyn_threshold(WoodPiece1, ImageMean,
              &Regions, 5.0, "dark");
threshold(WoodPiece1, &Dunkel, 0, 90);
dilation_rectangle1(Dunkel, &DunkelDilation, 30, 7);
difference(Regions, DunkelDilation, &RegionHell);
connection(RegionHell, &ConnectedRegions);
select_shape(ConnectedRegions, &SelectedRegions,
             "area", 30, 10000000);
get_mbutton(&Row1, &Column1, &Button1);
get_mbutton(&Row2, &Column2, &Button2);
store_lines(&Line, Row1, Column1, Row2, Column2);
intersection(Line, SelectedRegions, &Inters);
connection(Inters, &ConnectedInters);
count_obj(ConnectedInters, &Number);
```

Dazu wird vom Benutzer lediglich eine Schnittlinie spezifiziert (`get_mbutton`), obwohl auch diese automatisch bestimmt werden könnte, indem die Vorzugsrichtung der Jahresringe errechnet und die Schnittlinie dann senkrecht dazu gelegt wird. Diese Region wird dann mit den dunkel abgesetzten (`threshold`) Ringen geschnitten (`intersection`). Danach werden die Zusammenhangskomponenten (`connection`) gezählt (`count_obj`).

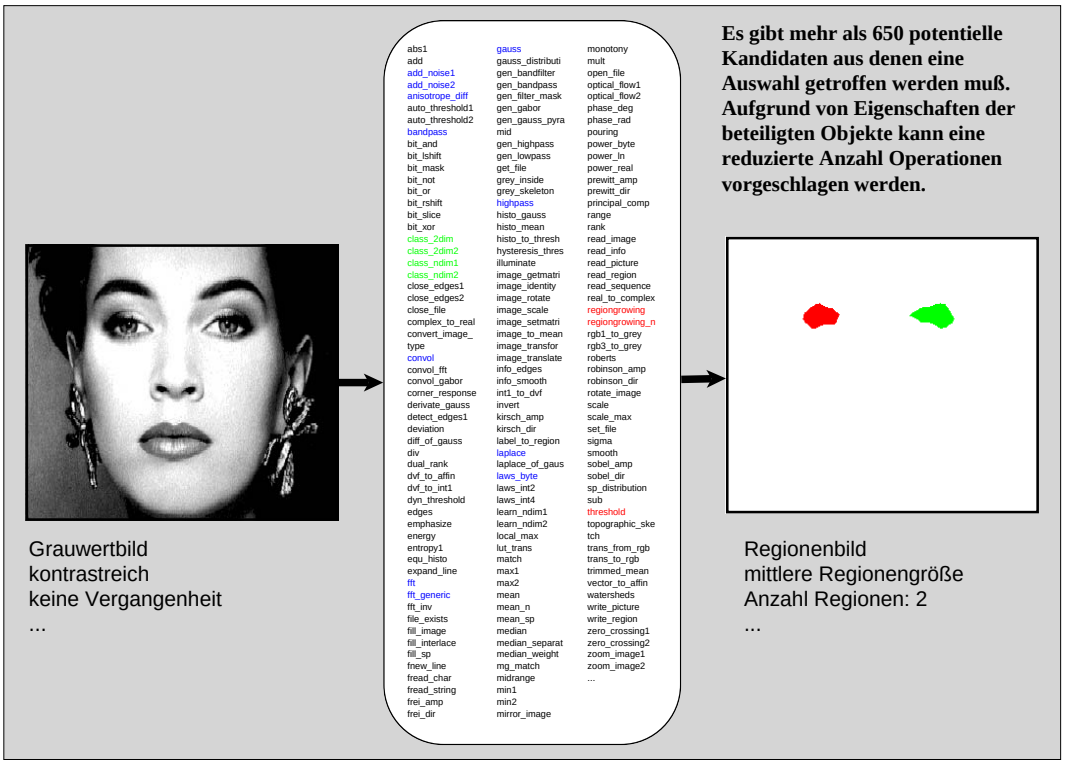


Abbildung 2.7: Schwierigkeiten bei der Auswahl des richtigen Operators

2.3 Lösungsansätze aus der KI

Wie die vorangehenden Beispielanwendungen zeigen, ergibt sich für den Benutzer in der Anwendung eines Systems der Komplexität von HORUS das Problem, aus der nahezu unüberschaubaren Vielfalt von Operationen die für seine Problemstellung geeigneten auszuwählen, zu parametrieren und zu verknüpfen. Diese Schwierigkeit wird in Abbildung 2.7 dargestellt. In jedem Schritt des Verfahrens hätte ein Benutzer aus einer Auswahl von (im Fall von HORUS) mehr als 650 Kandidaten zu wählen. Hierbei sind die verschiedenen Parametrierungen noch gar nicht berücksichtigt. Somit gäbe es damit potentiell 650^5, also mehr als 10^{14} Möglichkeiten. Selbst für ein System, das über KI-Ansätze den Suchraum einschränkt, ist diese Anzahl zumindest bei heutigen Rechenleistungen zu hoch, da pro Operation aufgrund der großen Datenmengen pro Bild (ca. 250 KByte) mindestens zehn bis zwanzig Millisekunden einkalkuliert werden müssen[2].

Ein weiteres, prinzipielleres Problem für KI-Ansätze ist das bisherige Fehlen von

[2]Hierbei wird vorausgesetzt, daß für die a-posteriori-Bewertung von Ergebnissen Bildanalyseoperatoren zum Einsatz kommen, die bei heutigen Rechengeschwindigkeiten etwa in diesem Bereich liegen.

brauchbaren Bewertungsfunktionen, die ein Maß für die Güte der errechneten Ergebnisse angeben könnten.

Trotzdem viele Versuche unternommen wurden, wobei bei bestimmten Anwendungsgebieten auch Erfolge erreicht wurden [Pau93, PBLR95], werden sie ohne Beschränkung auf spezifische Domänen vermutlich nicht weiterführen. In den entsprechenden Veröffentlichungen gibt es denn überwiegend auch nur zwei Lösungsformen: entweder die Einschränkung auf wenige anwendbare Operationen [LB95, Mes92], die als Vertreter einer Klasse gelten können, und damit eine Reduktion auf das in akzeptabler Geschwindigkeit Machbare, oder dem bloßen Zur-Verfügung-Stellen von Funktionen, d.h. der Aufstellung der Forderung, der Benutzer müsse genug Hintergrundwissen über die Domäne haben.

2.4 Benutzerunterstützung

Der hier verfolgte Ansatz besteht darin, dem Benutzer — soweit als irgend möglich — Unterstützung auf der taktischen Seite zu bieten[3], während das Finden der richtigen Strategie ihm überlassen bleibt. Auf diese Weise ist es möglich, sich die besten Fähigkeiten sowohl des Computers (Geschwindigkeit, Wissensbasis), als auch des menschlichen Benutzers (Vorstellungsvermögen, Deduktionsfähigkeit) zunutze zu machen und damit wesentliche Verbesserungen in dieser Domäne zu erzielen.

In Abbildung 2.7 könnte anhand der statischen und dynamischen Attribute (z.B. Parametertyp) der Quelle (hier: Grauwertbild) und des Ziels (hier: Regionen) die Operatorauswahl für den Benutzer vorselektiert werden, wie es im mittleren Teil der Grafik durch Hervorhebung anwendbarer Operatoren angedeutet ist.

2.5 Schrittweise Verfeinerung

Durch das Konzept der *schrittweisen Verfeinerung* können ganz allgemein die noch nicht vollständig spezifizierten Kanten im Graph mit Operationen versorgt werden. Durch eine entsprechende Spezifizierungskomponente ist es dabei möglich, ein gewünschtes (Teil-) Ergebnis am Ende der Kante anzugeben, um dann, mittels einer auf die am Anfang der Kante vorliegenden Attribute und die spezifizierten Zielattribute angesetzten wissensbasierten Komponente Vorschläge machen zu lassen, welche Operationen sinnvollerweise in Anwendung kommen könnten, um das gewünschte Ziel zu erreichen.

Dieser Ansatz unterscheidet sich von bisherigen, weil bislang entweder Vorwärts- oder Rückwärtsstrategien verfolgt wurden. In [GP92] werden diese Strategien dar-

[3] Dargestellt wird dies in Kapitel 6.

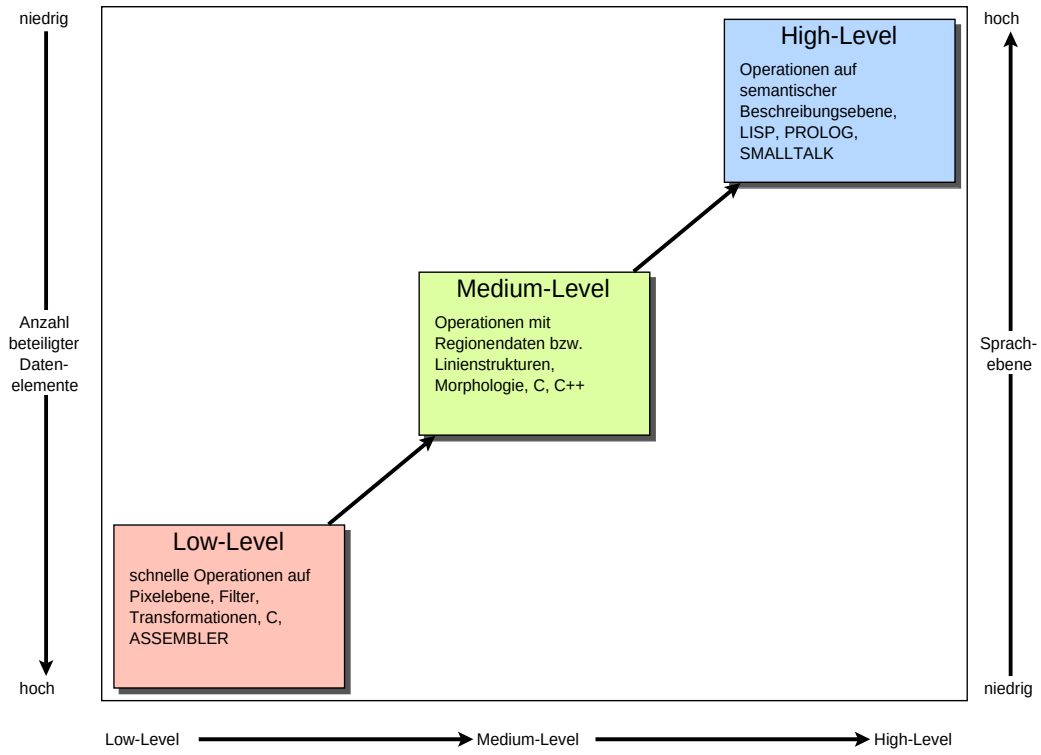


Abbildung 2.8: Sprachebenen im Bildanalyseprozeß

gestellt, die erste geht von gegebenen Daten aus und versucht, daraus mittels Anwendung von Operatoren ein bestimmtes Ergebnis zu erhalten. Die zweite versucht, von einem gegebenen Ziel die Schritte rückwärts zu erzeugen. Hier sollen dagegen nicht nur beide Strategien angewendet werden können, sondern auch der Weg dazwischen schrittweise verfeinert werden.

2.6 Sprachebene

Aufgrund der prinzipiell unterschiedlichen Erfordernisse in den verschiedenen Phasen des Bildanalyseprozesses sind die dort jeweils wünschenswerten Sprachfähigkeiten ebenfalls stark divergent, so daß sich eigentlich eine zweidimensionale Beschreibung wie in Abbildung 2.8 anbietet. Zum einen ist bei rechenzeitintensiven Anwendungen wie der Bildanalyse immer noch die Ausführungsgeschwindigkeit von Belang, die normalerweise mit der Ebene der verwendeten Sprache korreliert ist, zum anderen sind die verschiedenen Phasen des Bildanalyseprozesses in spezifischen Sprachebenen besser auszudrücken, bzw. zu behandeln.

Im folgenden wird lediglich die mittlere Ebene des Bildanalyseprozesses behandelt, d.h. zunächst wird die Low-Level-Bildanalyse als gegeben vorausgesetzt und wie ei-

ne *Black Box* behandelt. Wie in Abschnitt 1.1 schon dargestellt wurde, wird hier die Auffassung vertreten, daß die für Bildanalysezwecke nötigen Verfahren *im wesentlichen* nicht nur schon erforscht und bekannt, sondern in guten Bildanalysesystemen auch bereits enthalten sind.

Im Rahmen des schon angesprochenen Bildanalysepraktikums werden diese Low-Level-Algorithmen ebenfalls nur am Rande behandelt, jedenfalls nicht nachprogrammiert, denn die geschickte Implementierung solcher Verfahren ist reine Handarbeit, die besser in entsprechenden Programmierpraktika vermittelt werden kann. Mit der Problemlösung für die gestellte Aufgabe hat diese Art der Programmierung zumindest nur noch mittelbar zu tun.

Der letzte Teil des Bildanalyseprozesses, die High-Level-Analyse, verwendet normalerweise KI-Ansätze bzw. mindestens Programmierkonstrukte wie Schleifen und Fallunterscheidungen.

Wenn ein Bild in einige Regionen zerlegt worden ist, müssen diese üblicherweise nach bestimmten Merkmalen selektiert werden. Dazu wäre es normalerweise nötig, eine Schleife über alle Regionen zu realisieren und entsprechend den von der Bildanalysebibliothek berechneten Merkmalen eine Fallunterscheidung durchzuführen, um nicht relevante von den für die Anwendung wichtigen Regionen zu trennen. Insofern wird im verfolgten Ansatz auch eine bewußte Trennung zwischen der Low-Level-Bildverarbeitung und dem Kombinieren von Operatoren vorgenommen. Die im Bildanalyseprozeß nach der Segmentierung und Merkmalsbestimmung liegenden Schritte sind normalerweise ebenfalls sehr komplex und am ehesten mit typischen KI-Sprachen wie `LISP` oder `Prolog` zu lösen.

HORUS bietet solche Funktionalitäten in entsprechenden Operationen (beispielsweise `select_shape`) bereits an, so daß zumindest für dieses Problem keine *Programmierung* im eigentlichen Sinn erfolgen muß. Natürlich kann nicht jedes programmatisch darstellbare Konstrukt vorhergesehen und durch eine entsprechende Operation in HORUS ersetzt werden, aber hier geht es primär um denjenigen Bereich des Bildanalyseprozesses, der heute normalerweise experimentell durchgeführt werden muß, nämlich von der Vorverarbeitung über die Segmentierung bis zur Merkmalsbestimmung. Wenn nämlich (auch durch Experimentieren) der Weg bis dahin ermittelt ist, wird die Entwicklung einer entsprechenden Endanwendung für den gewünschten Spezialfall sehr einfach, wenn es möglich ist, das Gerüst des erzeugten Lösungswegs für die gewünschte Zielsprache[4] zu erzeugen.

Wenn eine Operation oder ein Konstrukt als Modul mit definierten Schnittstellen nach außen beschreibbar ist, dann kann es auch im weiteren wie ein Operator verwendet werden.

[4]Dies können im Fall von HORUS verschiedene sein, nämlich `C`, `C++`, `Smalltalk`, `Prolog` und `LISP`

Kapitel 3

Anforderungen

3.1 Geforderte Eigenschaften

Die in Abschnitt 2.2 aufgeführten Beispiele zeigen, daß für den Anwender bei so umfangreichen Bibliotheken wie HORUS nicht mehr die untere Ebene der Bildanalyse von Bedeutung ist. Er muß im Normalfall keine eigenen Verfahren auf der Pixelebene implementieren.

Dagegen wird ersichtlich, daß die Auswahl und Parametrierung von vorhandenen Operatoren mit zunehmendem Umfang ein größeres Problem darstellen. Ein wesentliches Ziel einer Oberfläche für Bildanalysesysteme muß daher sein, diese beiden Aufgaben zu unterstützen.

Ein derartiges System sollte also folgende Funktionen aufweisen:

Erstellen von Programmen Die Aneinanderreihung und logische Verknüpfung von (möglicherweise vorgefertigten) Operationen. Ferner das Erzeugen von eigenen Operatoren, die komplexere Operationen beinhalten (Makros).

Vorschlagen von Operatoren Eine Komponente, die Anwendbarkeit und andere Restriktionen benutzt, um Operatoren vorzuselektieren und vorzuschlagen.

Parametrierung von Operatoren Das System sollte in der Lage sein, „vernünftige“ Vorgabeparameter vorzuschlagen und ein interaktives Experimentieren zu ermöglichen.

Visualisierung von Ergebnissen Die beim Experimentieren entstehenden Zwischenergebnisse sollen in intuitiver Weise vom Benutzer erfaßt und bewertet werden können. Dazu sollten sie schnell und komfortabel dargestellt werden können.

Bezeichnung	Objekte	Konstrukte
'Languages that support visual interaction'	logische Objekte mit visueller Repräsentation	linear repräsentierte Konstrukte
'Visual programming languages'	logische Objekte mit visueller Repräsentation	visuell repräsentierte Konstrukte
'Visual information processing languages'	visuelle Objekte mit auferlegter logischer Repräsentation	linear repräsentierte Konstrukte
'Iconic visual information processing languages'	visuelle Objekte mit auferlegter logischer Repräsentation	visuell repräsentierte Konstrukte

Abbildung 3.1: Klassifikation von visuellen Sprachen (nach: [Cha87])

3.2 Grafische Oberfläche

Es liegt aufgrund dieser Anforderungen nahe, zur Realisierung eine Art grafischer Oberfläche zu verwenden, die die entsprechende Funktionalität bereitstellt. Solche *visuellen Sprachen*[1] (VL) existieren für die verschiedensten Anwendungsgebiete. Besonders in den Bereichen Prozeßvisualisierung und Parallelverarbeitung wurden hier schon früh gute Ergebnisse erzielt. Auch in aktuellen Arbeiten finden sich erste Ansätze, die neben visuellen Aspekten ebenfalls objektorientierte Methoden beinhalten [Sch95a].

Im Prinzip ist bei grafischen Ansätzen zunächst grob zu unterscheiden zwischen visuellen Sprachen, die ganz allgemein grafische Interaktion ermöglichen und *visuellen Programmiersprachen* (VPL), die echtes visuelles Programmieren mittels grafischer (also nicht-textueller) Elemente erlauben[2].

Zur genaueren Klassifizierung von Sprachen hat Chang in [Cha87] eine Einteilung in vier Grundtypen angegeben (siehe Abbildung 3.1). In der Spanne zwischen reinen grafischen Oberflächen, die nur bestimmte Aspekte eines Programmiersystems

[1] Eigentlich ist die Bezeichnung unglücklich, denn gemeint sind hierbei mit „Sprache“ keine Programmiersprache, sondern die Sprache der Interaktion mit einem System. Auch das Wort „visuell“ bedeutet für sich genommen noch keine Unterscheidung zu herkömmlichen Programmiersprachen, die ja auch visuell erfaßt werden. Eine bessere Bezeichnung wäre „grafische Interaktionssysteme“ (siehe auch [Gre95]).

[2] Oft wird in diesem Bereich versehentlich oder auch absichtlich mit zweideutigen Begriffen operiert. So wie vor einigen Jahren das Produkt dBase den Anspruch erhob, ein relationales Datenbanksystem zu sein, nur weil es (sinngemäß aus dem damaligen Handbuch) „Relationen zuläßt, wie $>, <$ u.ä.“, gibt es heute Produkte im Bereich der Programmiersprachen, die sich den falschen Anschein geben. Hierzu gehört Visual Basic, das aufgrund seiner mißverständlichen Bezeichnung von vielen fälschlicherweise für eine visuelle Programmiersprache gehalten wird. Auch integrierte grafische Oberflächen zu gängigen Programmiersprachen wie `PASCAL` oder `C` machen noch keine visuelle Programmiersprache aus.

grafisch aufbereiten und turing-vollständigen visuellen Programmiersprachen sind aber fast beliebig feine Abstufungen möglich.

In Kapitel 4 werden verschiedene Systeme für Bildanalyse dargestellt und unter anderem untersucht, welche Stufe von visueller Repräsentation sie erreichen.

3.3 Vereinfachung der Interaktion

Green schreibt in [Gre95]: *"Modern VPLs accordingly (i) reduce the number of concepts needed to program (e.g. no variables), (ii) allow data objects to be explored directly, (iii) explicitly depict relationships, and (iv) give immediate visual feedback of updated computations during editing. So it is claimed; non-VP fans might argue that modern textual languages have similar aims."*

Diese Vereinfachung durch Minimierung der Zahl der Konzepte ist eines der Grunderfordernisse für visuelle Sprachen. Verschiedene, von Green und Petre in [GP92] zitierte Studien haben gezeigt, daß trotz der viel herbeigeredeten (angeblichen) Vorteile von visuellen Sprachen ihr Verbreitungsgrad dennoch gering geblieben ist. Zitat: *"Large numbers of visual programmings languages (VPLs) have been invented. Their proponents claim that these languages are easier to understand than textual languages (TLs). Sometimes the claim is based on ill-digested pseudo-psychology about TLs 'not utilizing the full power of the brain' (Myers, 1990, p. 100); sometimes it rests on the unfortunately-named so-called 'software science', extended now into the visual domain by Glinert (1990) in an otherwise excellent paper."*

Citrin schreibt zu diesem Thema in [Cit93]: *"In short, although visual languages were proposed in order to simplify the programming process, they are encumbered by interfaces that make programming more difficult. If we expect users to accept visual languages, we will have to provide entry systems as simple and natural as pen and paper, or at least as simple as conventional screen-oriented text editors."*

In verschiedenen Arbeiten [Gre77, Gre89, GPB91] stellen Green et. al. schließlich fest, daß letztlich keine „optimale" Notation (textuell oder visuell) existiert, sondern daß abhängig von den Erfordernissen der jeweiligen Anwendung die eine oder andere spezifische Vorteile hat.

3.4 Kriterien für visuelle Programmiersprachen

Aus den bis hier dargelegten Anforderungen ergeben sich gegenüber klassischen Ansätzen etwas abgewandelte Kriterien für den Entwurf einer grafischen Oberfläche.

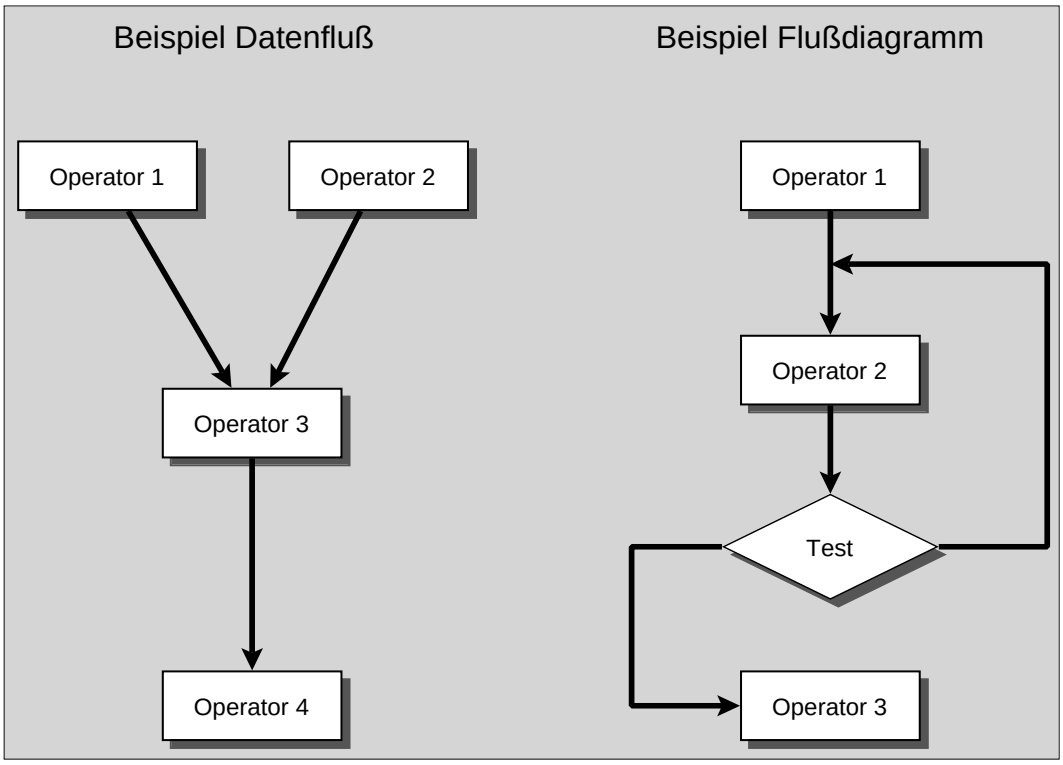


Abbildung 3.2: Darstellung von Programmen

Die normalerweise geforderten Eigenschaften, z.B. die einer turing-vollständigen Sprache, sind bei näherer Betrachtung mit entscheidenden Nachteilen behaftet. Bei der Realisierung einer solchen visuellen Sprache ergeben sich zwei Möglichkeiten:

1. Finden von grafischen Äquivalenten für die typischen syntaktischen Konstrukte von herkömmlichen Programmiersprachen, wie Verzweigungen und Schleifen. Dies hat den Nachteil, daß bestimmte Konstrukte (insbesondere Schleifen) in grafischen Repräsentationen — wie Flußdiagrammen — ähnlich zu Datenflußkanten sind. Tatsächlich stellen aber die Kanten in Flußdiagrammen nicht den Datenfluß, sondern nur die Reihenfolgebeziehungen dar (siehe Abbildung 3.2).

   Zum Teil ergeben sich semantische Bedeutungen lediglich aus der Topologie des Graphen, bei Schleifen z.B. aus der Tatsache, daß sie im Gegensatz zu reinen Datenflußverbindungen rückgekoppelt sind. Zudem haben derartige Lösungen Probleme bei der Formulierung von Ausdrücken (beispielsweise Bedingungen). Auch hier gibt es prinzipiell zwei Möglichkeiten:

   (a) Notation in textueller Form, dies hat jedoch einen *Medienbruch* zur Folge, d.h. den Wechsel von der grafischen zur textuellen Ebene. Sehr

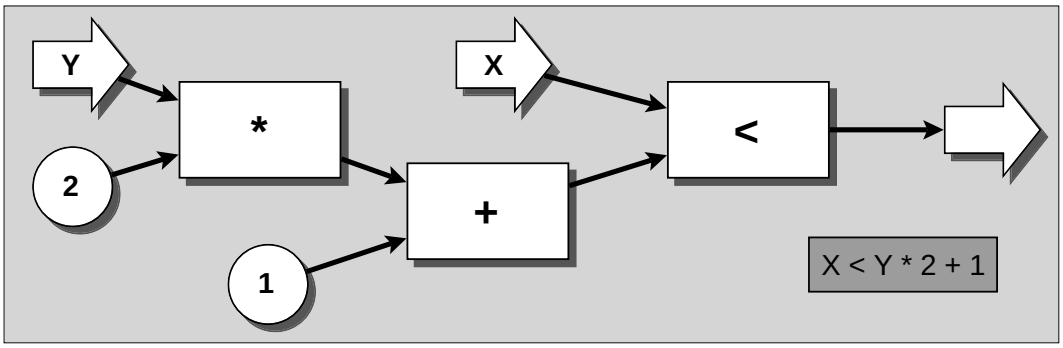


Abbildung 3.3: Grafische Repräsentation eines arithmetischen Ausdrucks

schwierig zu realisieren ist auch die Referenzierung von Daten: Der simple Ausdruck $A > 2$ hat bei datenflußbasierten grafischen Sprachen keine Bedeutung, weil es keine Variable *A* gibt. Andererseits muß es auch bei diesem Ansatz möglich sein, Daten zu referenzieren, die jedoch nur als grafische Kanten vorliegen.

(b) Rückführung auf n-stellige atomare Operationen, z.B. Ausdrücken einer Bedingung wie „$<$ (kleiner als)“ als Funktion *kleiner_als*(A,B,C) mit *A* und *B* als Eingabeparameter und *C* als Ausgabeparameter. Dies führt zu großen und damit unübersichtlichen Programmen, weil textuell einfach zu notierende Ausdrücke wie $X < Y * 2 + 1$ schon drei Operationen beinhalten, die in der entsprechenden grafischen Repräsentation sehr viel Platz beanspruchen (siehe Abbildung 3.3 und Abschnitt 3.6).

Die genannten Probleme berühren hierbei lediglich einen Teil der in der Realität auftretenden. Nicht betrachtet sind z.B.:

- Datentypdefinitionen (nicht wirklich notwendig für Turing-Vollständigkeit, aber wünschenswert aus Anwendersicht).
- Behandlung von Indizierung bei Feldern oder Selektion bei Strukturen. Konkret entsteht dieses Problem im Bereich der Bildanalyse bei der Bearbeitung von Bildfolgen, bzw. bei der Attributierung von Objekten mit spezifischen Merkmalen.

2. Anwendung von grafisch besser repräsentierbaren Turing-Äquivalenten, wie µ-Rekursion oder Petrinetzen. Für den Programmierer bedeutet dieser Ansatz jedoch ein radikales Umdenken bei der Art der Herangehensweise an ein Problem. Außerdem ist es auf diese Weise nicht leicht möglich, eine prototypische Lösung in eine herkömmliche, imperative Programmiersprache so zu transformieren, daß sie wartbar bleibt. Das transformierte Programm wird immer

eine Art „Laufzeitsystem“ beinhalten, während die eigentliche Programmlogik praktisch nur noch aus einer Tabelle besteht. In diesem Fall ist zwischen Eleganz und Praxistauglichkeit abzuwägen.

Gerade im Bereich der Bildanalyse sind bestimmte Anforderungen an die Programmierumgebung[3] verzichtbar. So kann auf bestimmte Sprachkonstrukte, wie Rekursion oder Iteration, weitgehend verzichtet werden, denn der Anspruch, eine turingvollständige Sprache zur Verfügung zu haben ist fast nur für die Low-Level-Programmierung von Operatoren notwendig. Hier allerdings verbietet sich die Anwendung von „echten“ objektorientierten Programmiersprachen von selbst, da diese zur Realisierung von Persistenz und dynamischer Bindung normalerweise interpretativ arbeiten und daher aus Gründen der Rechenzeit für die Bildanalyse nicht geeignet scheinen.

Insofern ist eine Trennung von Low-Level-Verarbeitung in einer maschinennahen Sprache (z.B. in `C`) und Steuerung in einer objektorientierten Sprache (wie z.B. `Smalltalk`) vernünftig. Ein gewisser Nachteil könnte in der Zweistufigkeit zwar gesehen werden, aber auch in visuellen Programmiersprachen, die turing-vollständig sind, ist mindestens der Medienbruch zwischen der reinen grafischen Manipulation der Grundelemente und der Erzeugung von komplexeren Ausdrücken wie Bedingungen vorhanden. Neben den schon besprochenen Performance-Nachteilen ist hier praktisch immer das Endergebnis unübersichtlich. Dies wird in den meisten Ansätzen dadurch zu lösen versucht, daß Modularisierung in Form von *Fokussierung* auf einzelne Bestandteile angewendet wird, indem Mengen von grafischen Elementen zusammengefaßt und durch ein anderes Primitiv symbolisiert werden (siehe [GQ94] und Abschnitt 7.7).

3.5 Gleiche Sprache für textuelle und visuelle Anwendung

Es existieren Ansätze, die versuchen, eine Sprache zu verwenden, für die sowohl eine textuelle als auch eine grafische Repräsentation möglich ist, z.B. *Signal* [BG90, GGBM91] und *Lustre* [HCRP91]. Prinzipiell ist diese Vorgehensweise sinnvoll, denn auf diese Weise können die mit der grafischen Komponente erstellten Programme textuell nacheditiert und das Resultat sogar wieder mit der grafischen Komponente weiterverarbeitet werden. Andererseits ist dem Benutzer damit die Sprache, in der er seine Applikation letztlich formulieren muß, praktisch schon vorgegeben. Aufgrund der spezifischen Vor- und Nachteile von visuellen Sprachen haben diese meist auch Probleme bei der Formulierung von Algorithmen für die unteren Ebenen der Bildanalyse (siehe Absatz 3.6).

[3] im Gegensatz zu Programmier*sprache*

Natürlich gilt hier mindestens die Einschränkung, daß so nur mit Objekten bzw. Datenstrukturen gearbeitet werden kann, die von den atomaren Operationen des zugrundeliegenden Systems beherrscht werden, falls nicht sämtliche Möglichkeiten einer turing-vollständigen Sprache in die grafische Form überführt werden sollen.

Neben der bloßen Verfügbarkeit von Äquivalenten von Konstrukten wie Fallunterscheidungen oder Schleifen ist hier auch an die Definition von neuen Datentypen zu denken. Fehlt diese Möglichkeit, muß schon das Basissystem für die anvisierte Domäne mächtig genug sein. HORUS bietet Datenstrukturen mit sehr hohem Abstraktionsgrad, neben Regionen auch Polygone, Punkte, Strecken und auch sogenannte *Extended Line Descriptions* (XLD), eignet sich also sehr gut als Basis für den hier besprochenen Einsatz.

3.6 Komplexität verschiedener Notationen

Nickerson entwickelt in [Nic94] Maße zur Komplexität der Darstellung in textueller sowie graphischer Notation. Der Autor kommt zu dem Schluß, daß aufgrund seiner Untersuchungen die Anwendung des visuellen Programmierparadigmas auf vollständige Programmiersprachen wegen der geringen Dichte vermutlich nicht optimal ist. Er schreibt: *"Application of these metrics to current visual programming languages does not paint an optimistic future for the use of fully general, fully diagrammatic visual programming languages due to their low density."* und später: *"Also, those problems lending themselves to graph rather than to tree representations will be best handled visually."* Diese offensichtliche Unterlegenheit bezüglich der Dichte von visuellen Sprachen gegenüber textuellen wird vielleicht am besten durch die sogenannte *Deutsch-Grenze* illustriert. Diese bezieht sich auf eine Aussage von Peter Deutsch und besagt, es sei ein wesentliches Problem von grafischen Programmiersprachen, daß nicht mehr als fünfzig Primitive gleichzeitig auf dem Schirm sichtbar sein können.

Nickerson schließt mit: *"Finally, hybrid languages in which graphic representations of program structure are combined with textual expressions can match textual densities."* Andere Verfechter visueller Sprachen plädieren allerdings gegen einen sogenannten *Medienbruch*, also den Wechsel des Programmierparadigmas von textueller zu grafischer Notation. So schreibt Citrin in [Cit93]: *"Text and graphics are conventionally entered through two different modes, the former through a keyboard and the latter through a pointing device. To enhance user satisfaction, front ends must either eliminate this mode change or somehow minimize the effort associated with making the change."*

Auch Green [GP92] kommt durch Experimente zu dem Schluß, daß: *"Our first conclusion obviously concerns the plausability of dataflow VPLs. Given these results, it can hardly be claimed that VPLs are consistently superior to TLs! The data show*

that the graphical notations are in fact consistently worse *than the textual notations."* In diesem Artikel werden verschiedene Beispiele untersucht, wobei festgestellt wird, daß bei Konstrukten wie *if-then-else* die grafische Notation unterlegen ist (siehe auch [Gre77, Gre89, GPB91]). Ferner scheint zwar die *Erstellung* von Programmen mittels datenflußbasierter Sprachen mitunter intuitiver zu sein, nicht jedoch die *Wartung*, weil grafische Notationen inhärent schwieriger zu *verstehen* sind als textuelle.

Obwohl also allgemein die Erkenntnis Fuß gefaßt hat, daß turing-vollständige visuelle Programmiersprachen allgemein nicht optimal sind, bleibt festzuhalten, daß diese Ergebnisse auf bestimmten Annahmen beruhen, die in der in dieser Arbeit anvisierten Domäne nicht in vollem Umfang zum Tragen kommen. Dabei sind Ergebnisse wie die von Brooke und Duncan [BD80] oder Curtis et. al. [CSK+89] ohnehin nicht zu berücksichtigen, weil sie sich lediglich auf Flußdiagramm-Sprachen beziehen. Aber auch die Arbeiten von Green [GP92, GP95] setzen implizit die Notwendigkeit für Turing-Vollständigkeit voraus. Hier soll eher von einer Art Experimentierumgebung mit Benutzerunterstützung ausgegangen werden.

Die wesentliche Existenzberechtigung haben grafische Programmierwerkzeuge im Bereich der Visualisierung und des Experimentierens mit Parametern, also genau im anvisierten Gebiet.

3.7 Keine Trennung zwischen Übersetzungs- und Laufzeit

Bei den heute gängigen visuellen Programmiersprachen ist eine Schwäche durchgängig zu erkennen: Scheinbar in Übertragung des Ansatzes, den Vorgang des Programmierens in drei klar abgrenzbare Phasen aufzuteilen (Editieren, Übersetzen und Ausführen), existieren in diesen Systemen normalerweise zwei separate Teile, nämlich eine Art „Editor" und eine Ausführungskomponente. Das bedeutet, daß der Benutzer, um sein Programm zu testen, in einen anderen Modus wechseln muß, indem er wie in Cantata[4] einen Run-Knopf drückt.

Es gibt in diesem Bereich fließende Übergänge, trotzdem haben hier oft gerade Systeme Probleme, die sowohl über eine textuelle als auch über eine grafische Notation verfügen. Meist ist die grafische Repräsentation nur eine Visualisierung der textuellen Programmbeschreibung.

Der eigentliche Vorteil bei visueller Programmierung liegt gerade in der Möglichkeit, schon zur Erstellungszeit der Applikation mit aktuellen Daten experimentieren zu können, wobei jede Veränderung an Parametern oder auch am Ablauf zwar

[4] Oberfläche zu Khoros, siehe Kapitel 4.

sofort sichtbar werden sollte, jedoch möglichst ohne eine vollständige Neuberechnung nach sich zu ziehen, was aus Geschwindigkeitsgründen nicht adäquat wäre. Dies ist sogar der *wesentliche* Vorteil, weil bestimmte Konstrukte auszudrücken in visuellen Programmiersprachen eher komplizierter ist als in textuell repräsentierten (siehe Abschnitt 3.6).

3.8 Konsistenz und Systemverhalten

Eine der essentiellen Anforderungen an das Systemverhalten der grafischen Oberfläche ist eine zu jedem Zeitpunkt gewährleistete Konsistenz[5]. Da die Oberfläche neben der Beschreibung der Ablaufstruktur (des Programms) auch einen Laufzeitzustand beinhaltet, ist diese Forderung besonders schwierig zu erfüllen, weil sowohl die Daten als auch die topologische Struktur des Netzes variabel sind.

Gerade die sofortige Reaktion auf Aktionen des Benutzers ist in einer Umgebung, die ein experimentelles Vorgehen ermöglichen und fördern soll, ein entscheidendes Gütekriterium. Der bis heute übliche Zyklus von Editieren, Kompilieren und Ausführung zum Austesten von Operatorfolgen bzw. verschiedenen Parametrierungen soll durch eine schnell erlernbare Oberfläche abgelöst werden, in der diese Ebenen nicht getrennt sind.

3.9 Funktionale Operationen

Eine der Grundvoraussetzungen für eine grafische Repräsentation ist eine komplett *funktional* aufgebaute Operatorbasis. Operatoren, die Seiteneffekte ausweisen, lassen sich grafisch nur sehr umständlich repräsentieren. In Abbildung 3.4 wird ersichtlich, daß aus dem Netz selbst die Reihenfolge der Anwendung der Operatoren 2 und 3 nicht vollständig bestimmt ist (ohne die gestrichelte Kante). Wenn aufgrund von Seiteneffekten eine bestimmte Reihenfolge von Operator 2 und 3 notwendig wäre, müßte diese zusätzlich neben den reinen Datenflußabhängigkeiten, durch zusätzliche Abhängigkeitskanten modelliert werden (in der Abbildung durch die gestrichelte Kante symbolisiert).

In einigen Fällen ist diese starke Forderung allerdings nicht haltbar. Gemeint sind spezielle Arten von Operatoren mit Seiteneffekten, zum einen Datenquellen, also Operatoren ohne Eingabeparameter, die Werte von außerhalb des Systems liefern, z.B. Benutzereingaben oder Bilddaten von einem Framegrabber. Zum anderen existieren Operatoren, die nur Eingabeparameter haben, also Datensenken, z.B. eine Operation, die Bilddaten auf eine Datei ablegt. Neben diesen Seiteneffektsroutinen,

[5]Gemeint ist die jederzeit gewährleistete exakte Beschreibarkeit und Reproduzierbarkeit des Systemverhaltens.

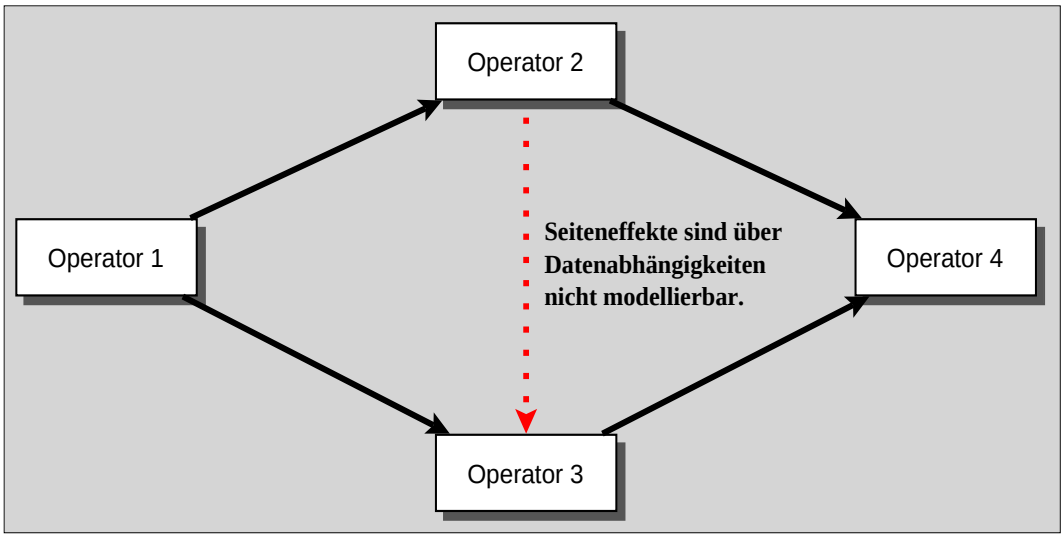


Abbildung 3.4: Modellierung von Seiteneffekten

die modelliert werden können und müssen, existieren auch in HORUS solche, die Seiteneffekte aufweisen, die nicht oder nur schwierig modellierbar sind. Beispiele hierfür sind Operatoren, die die Eigenschaften des Laufzeitsystems beeinflussen, z.B. ob es 4- oder 8-Punkt-Nachbarschaft berücksichtigt.

Als weiteres Beispiel sei hier eine allgemeine Bilddatentransformation genannt, bei der zunächst durch Hilfsprozeduren die Parameter für Translationen, Rotationen und Skalierungen eingestellt werden, die intern eine Transformationsmatrix modifizieren und bei der danach durch Anwendung der eigentlichen Transformationsoperation alle Operationen in einem Schritt ausgeführt werden können. Da die zu transformierenden Bilder nicht als Parameter in die Hilfsprozeduren zur Modifikation der Matrix eingehen, herrscht auch hier scheinbar keine Datenabhängigkeit.

Die Begründungen für die in Kauf genommenen Seiteneffekte sind sehr unterschiedlich, im ersten Fall sollte nicht in jede regionenorientierte Operation einen weiteren Parameter aufgenommen werden, im zweiten wäre durch die Hinzufügung des zu transformierenden Bildes die Effizienz reduziert worden, weil dann aufeinanderfolgende affine Abbildungen einzeln nacheinander ausgeführt werden müßten.

Zumindest in diesem Fall bot es sich jedoch an, die Operationen so zu ändern, daß sie vollständig funktional sind, daß sie also eine als Parameter spezifizierte Transformationsmatrix ändern, anstelle einer systemweiten. Hiermit wird weder die Effizienz noch die Modellierbarkeit eingeschränkt.

Kapitel 4

Bisherige Ansätze

4.1 Andere Arbeiten

In den Überblicksbeiträgen [AZJA94] und [Hil92] werden verschiedene aktuelle Ansätze verglichen. Nach einer anfänglichen Sichtung der nachfolgend aufgeführten Systeme wurden diejenigen, die am ehesten den in Kapitel 3 aufgestellten Kriterien genügen, für eine eingehendere Analyse ausgewählt und im Rahmen einer Diplomarbeit verglichen [Jak95].

4.1.1 AVS

Das *Application Visualization System* (AVS) ist eine kommerzielle Software für Visualisierungszwecke, die auf vielen UNIX-Plattformen verfügbar ist. Neben den Handbüchern der Herstellerfirma Advanced Visual Systems [AVS93, AVS92a, AVS92c, AVS92b] finden sich Zusammenfassungen in [UFK+89, Ber91, Ber92].

Abbildung 4.1 zeigt einen Datenflußgraphen, Abbildung 4.2 die Parametrierung eines Schwellwertoperators. Trotzdem die farbliche Unterscheidung von Parametertypen zunächst logisch erscheint, ist die daraus resultierende Darstellung insgesamt verwirrend. AVS wurde ebenfalls dem Systemvergleich unterzogen [Jak95].

4.1.2 Explorer

IRIS Explorer ist ein Visualisierungsprogramm von Silicon Graphics, Inc., eines Herstellers von visuellen Verarbeitungssystemen, der auf 3D-Visualisierung spezialisiert ist. Dies Software-Paket wird kostenlos mit jeder Workstation von SGI geliefert [Exp92a, Exp92b]. Die Reichweite der Bibliothek im Bereich Bildanalyse ist bei diesem System sehr beschränkt [Jak95]. Abbildung 4.3 zeigt den Datenflußgraph, die Parametrierung ist in Abbildung 4.4 dargestellt. Besonders interessant ist, daß

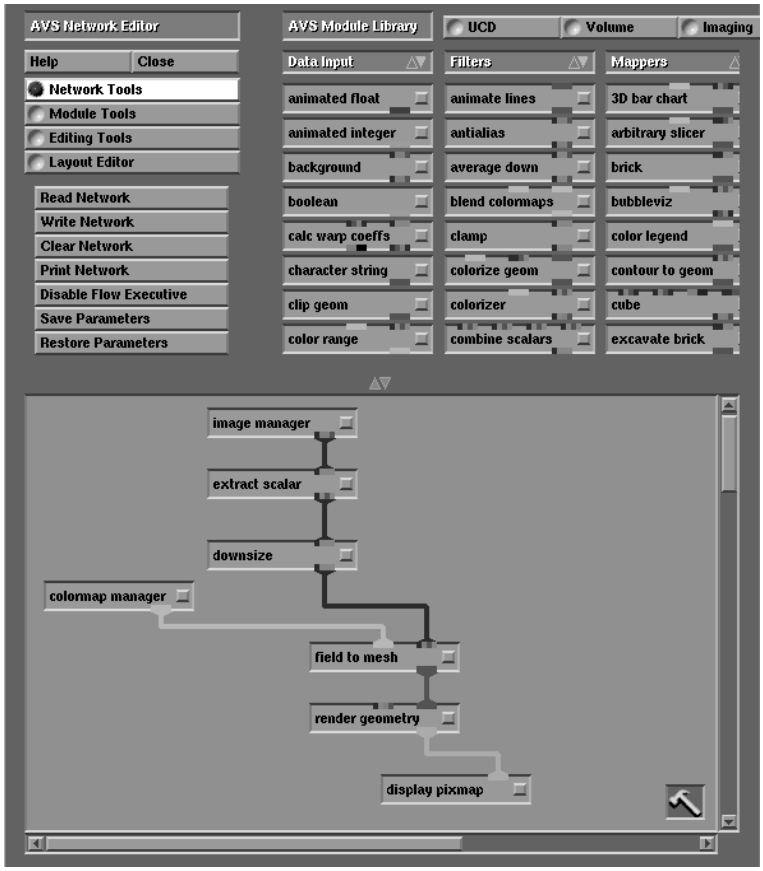


Abbildung 4.1: AVS: Datenflußgraph (aus: [Jak95])

hier semantische Informationen (z.B. Minimum und Maximum) vorhanden sind, die eine grafische Parameterspezifikation mittels eines Schiebereglers erlauben, eine Eigenschaft, die Explorer von anderen Werkzeugen (wie Khoros) positiv unterscheidet. Allerdings sind die grafischen Elemente im Operator selbst untergebracht, was bei vielen Parametern schnell unübersichtlich wird.

4.1.3 HDEVELOP/HINSPECTOR

Es handelt sich bei HDEVELOP um eine Oberfläche für HORUS (siehe Abschnitt 5.1.1), mit der, ebenso wie in dem in dieser Arbeit verfolgte Ansatz, auf einfache Weise Bildanalyseapplikationen erstellt werden sollen [Man93]. Eine Darstellung von HDEVELOP wurde schon in Abbildung 1.2 gezeigt, das Fenster zum Erstellen von Programmen wird in Abbildung 4.5 dargestellt. Im unteren Bereich ist die Parametrierung eines Operators herausgegriffen. Eigentlich erfüllt HDEVELOP nicht die Forderungen an eine visuelle Programmiersprache, denn die Konstrukte und die Darstellung sind inhärent textuell. Grafisch ist nur die Darstellung der Da-

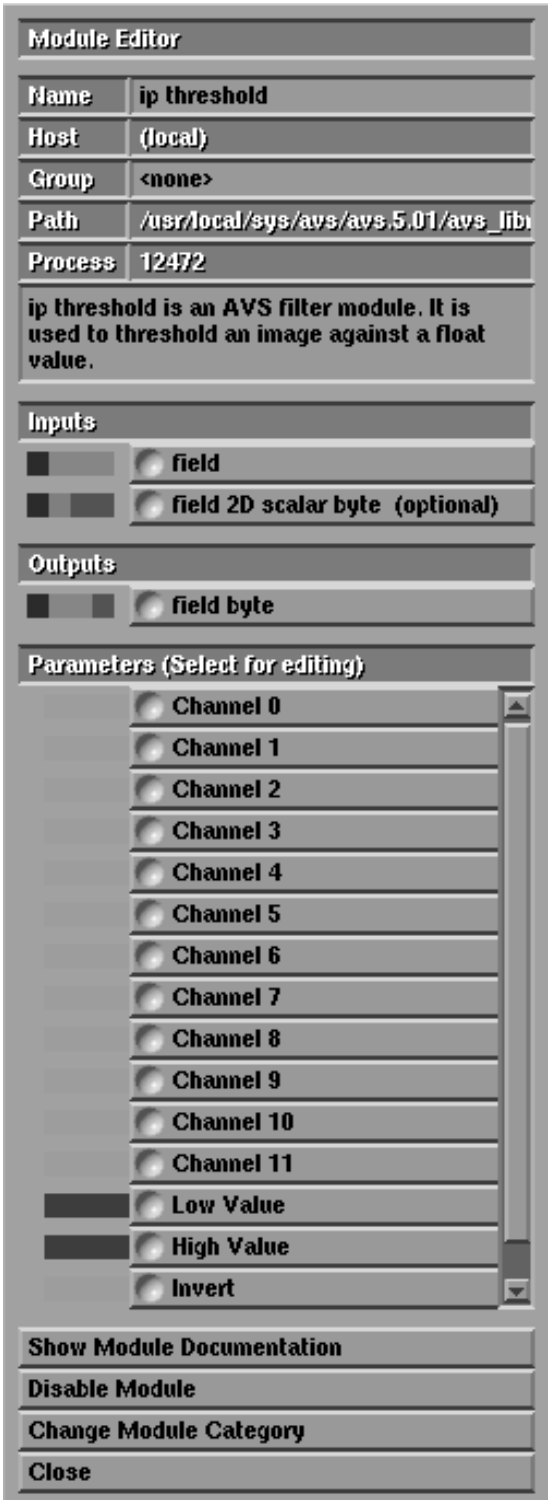


Abbildung 4.2: AVS: Parametrierung (aus: [Jak95])

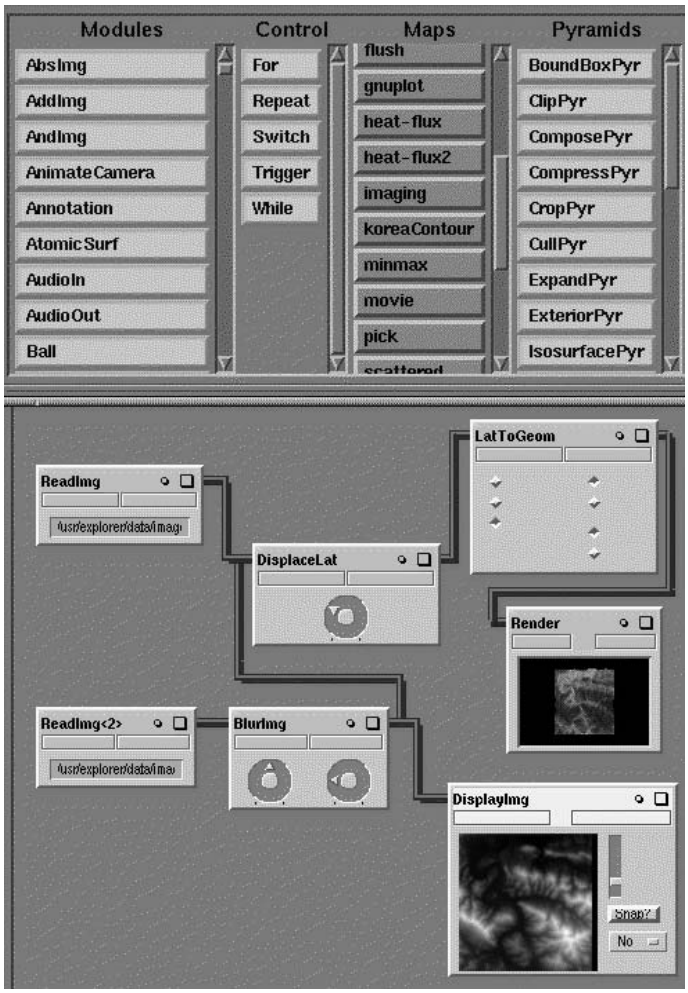


Abbildung 4.3: Explorer: Datenflußgraph (aus: [Jak95])

Abbildung 4.4: Explorer: Parametrierung (aus: [Jak95])

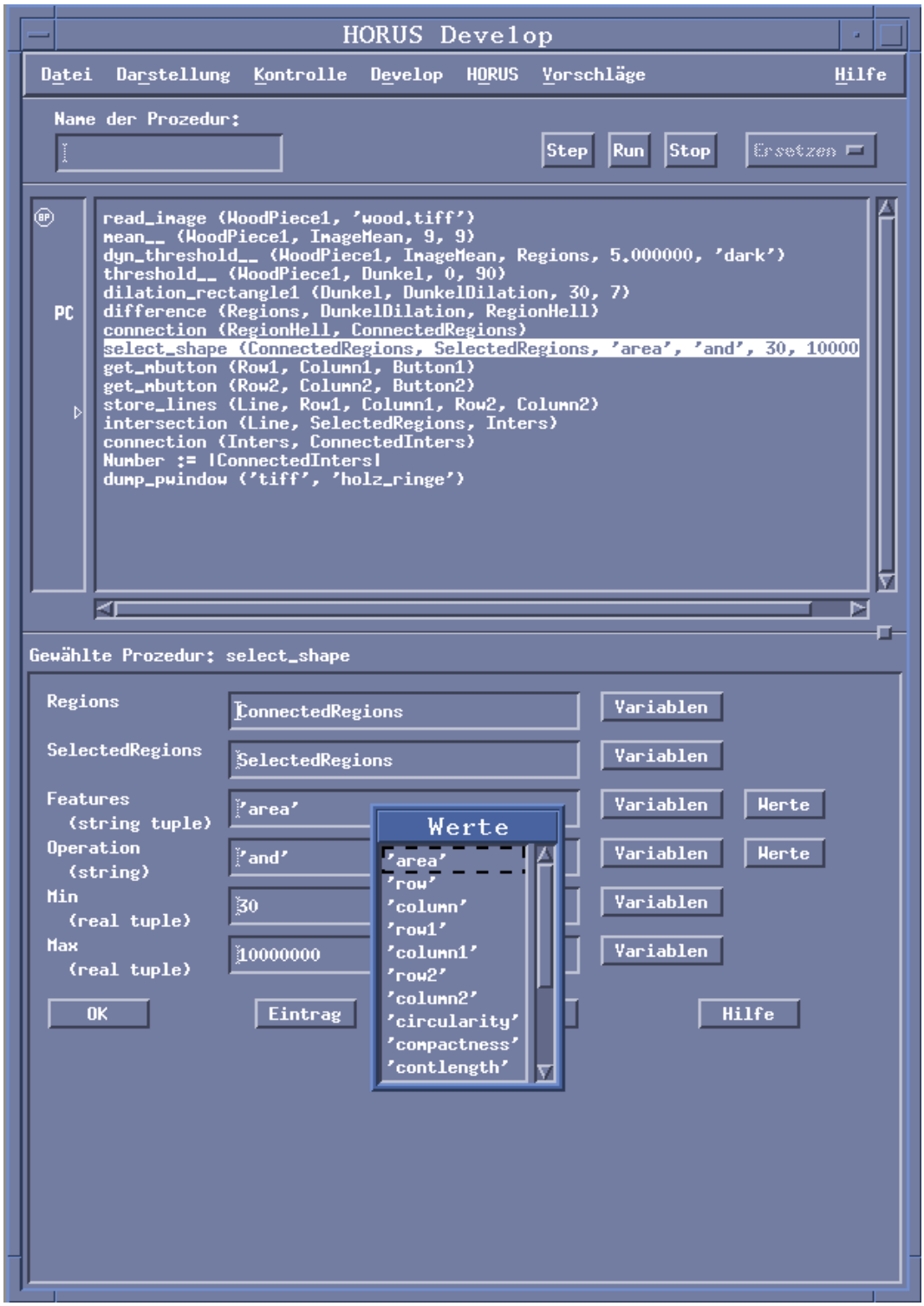


Abbildung 4.5: HDEVELOP: Darstellung des Programms (mit Parametrierung)

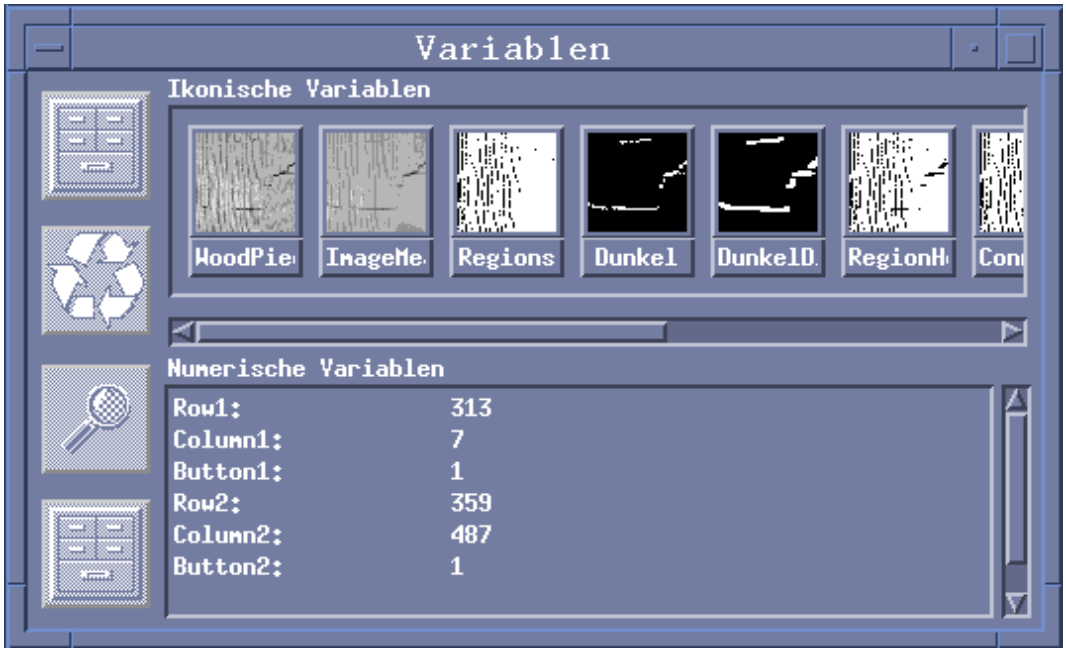


Abbildung 4.6: HDEVELOP: Darstellung der Objekte

tenobjekte (siehe Abbildung 4.6), deshalb gehört HDEVELOP dennoch zur Klasse der visuellen Sprachen. HDEVELOP ist mit einem Visualisierungswerkzeug namens HINSPECTOR gekoppelt, in der zunächst die Objekte in einem Visualisierungsfenster dargestellt werden (Abbildung 4.7). In diesem Fenster können dann Parameter eingestellt werden, z.B. Ausschnittvergrößerung, Darstellungsfarben und so weiter. Hier können auch Regionen selektiert und für diese verschiedene Visualisierungsmöglichkeiten gewählt werden (Abbildung 4.8), z.B. Grauwert- und Forminformationen oder Histogramme. Die gewählten Möglichkeiten werden dann in einem anderen Fenster dargestellt (Abbildung 4.9). HDEVELOP ist nach Tabelle 3.1 eindeutig der ersten Kategorie zuzuordnen, denn es handelt sich hierbei um eine Sprache, die lediglich grafische Interaktion, z.B. zum Einstellen von Parametern ermöglicht, jedoch keine eigentlich grafisch repräsentierten Konstrukte aufweist. Andererseits liegt der Vorteil von HDEVELOP in der Ähnlichkeit zu imperativen Sprachen wie C. Da dieses System eine Entwicklung des Lehrstuhls und keine echte visuelle Sprache ist, wurde auf eine Analyse im Rahmen des Systemvergleichs verzichtet.

4.1.4 IUE

Das *Image Understanding Environment* (IUE) ist der vom amerikanischen Verteidigungsministerium (ARPA) angeregte und unter Zusammenarbeit verschiedener Forschungsinstitute realisierte Plan, eine allgemein anerkannte Software-Umgebung

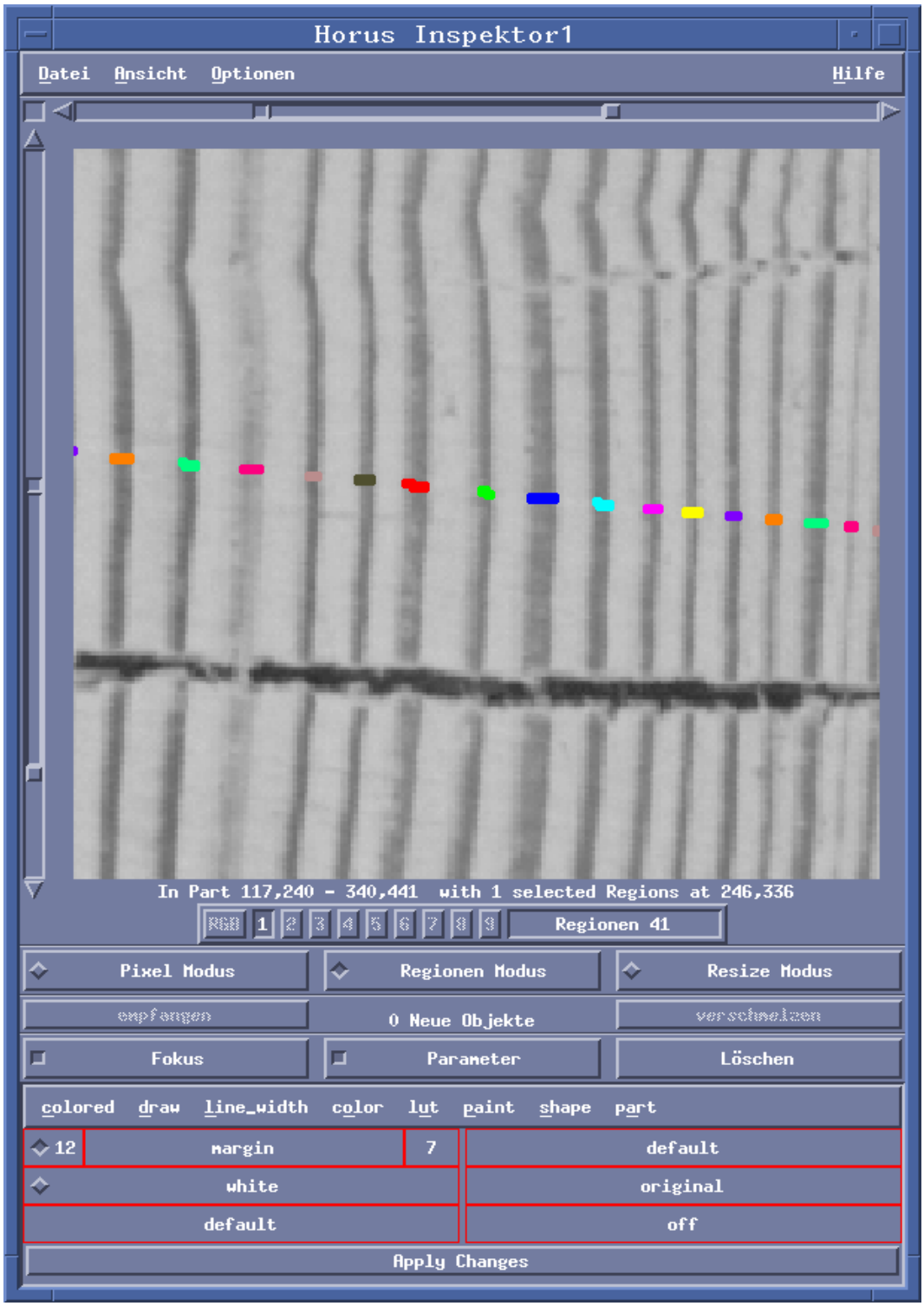


Abbildung 4.7: HINSPECTOR: Darstellungsfenster (mit Ausschnitt)

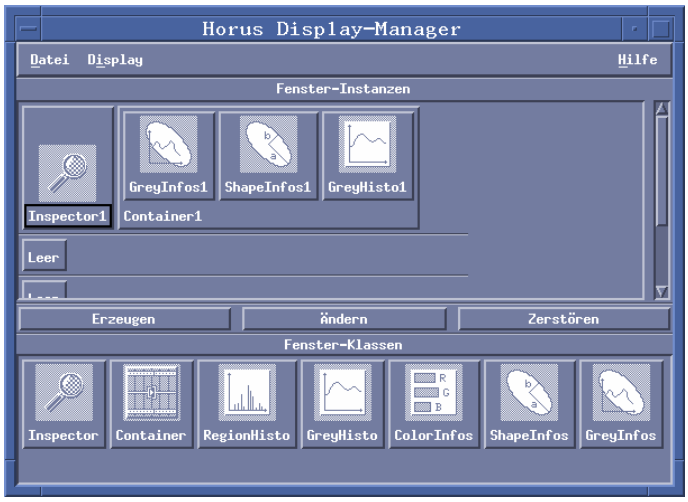


Abbildung 4.8: HINSPECTOR: Auswahl der Inspektoren

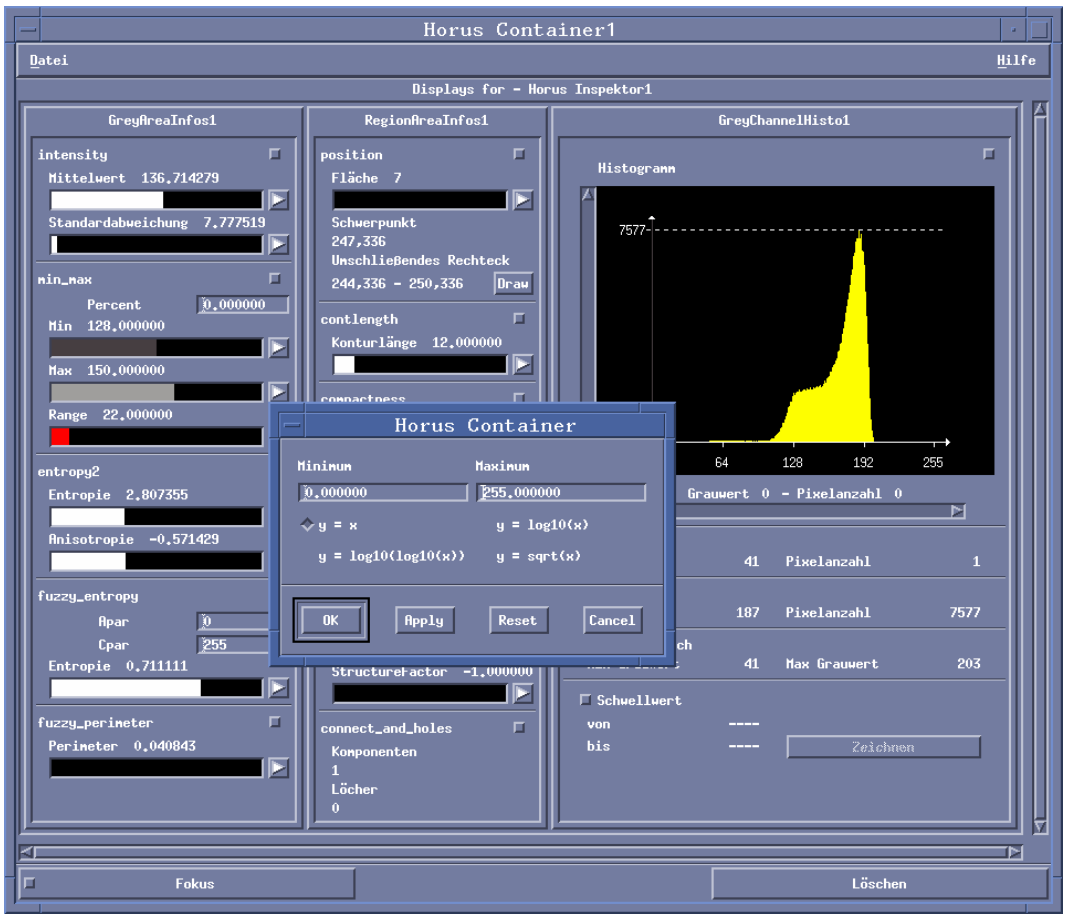


Abbildung 4.9: HINSPECTOR: Inspektoren

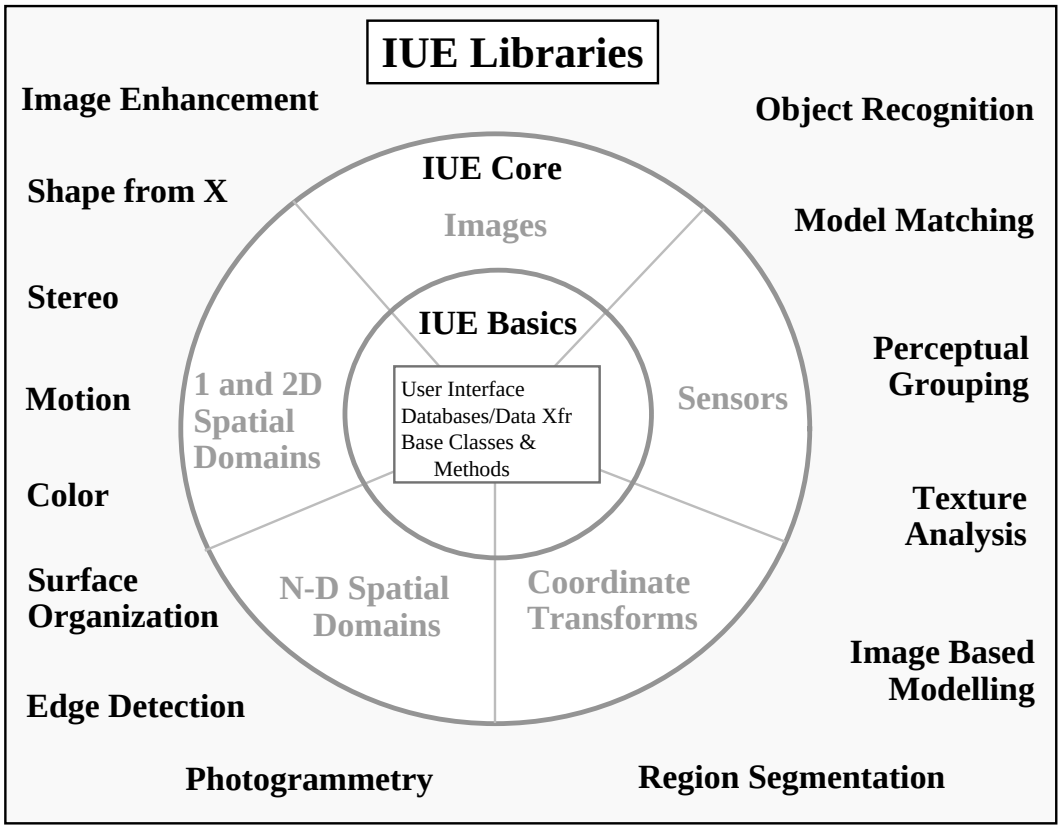


Abbildung 4.10: IUE: Architektur (aus: [IUE95])

für Bildverstehen zu etablieren. In den entsprechenden Veröffentlichungen werden die Zielgruppen klar genannt. Im wesentlichen zielt das IUE eher auf Ausbildungszwecke als auf praktische Anwendung, denn schon die minimalen Hardware-Erfordernisse (sehr aufwendig ausgestattete Workstation) sind nicht gerade gering. Daher ist das System für reale industrielle oder militärische Anwendungen mit möglicherweise vorhandenen Echtzeiterfordernissen nicht anwendbar, wohl aber, um neue Algorithmen zu entwickeln oder Verfahren auf Ihre Anwendbarkeit hin zu untersuchen. Obwohl das IUE public-domain verfügbar gemacht werden soll, wird die eigentliche Implementierung von der Firma Amerinex Artificial Intelligence, Inc., dem Hersteller von KBVision, vorgenommen.

Das Konzept des IUE ist der bislang am weitesten gehende Versuch, die Problematik des Bildverstehens mit objektorientierten Methoden anzugehen. Obwohl aus den Veröffentlichungen [IUE95] hervorgeht, daß für die Implementierung sowohl `LISP` als auch `C++` (mit transparentem Zugriff auf gegenseitige Datenstrukturen) verwendet werden sollte, ist davon vorerst Abstand genommen worden. Die aktuelle Version basiert komplett nur auf `C++`, was eine sprachunterstützte Persistenz von Objekten nicht ermöglicht.

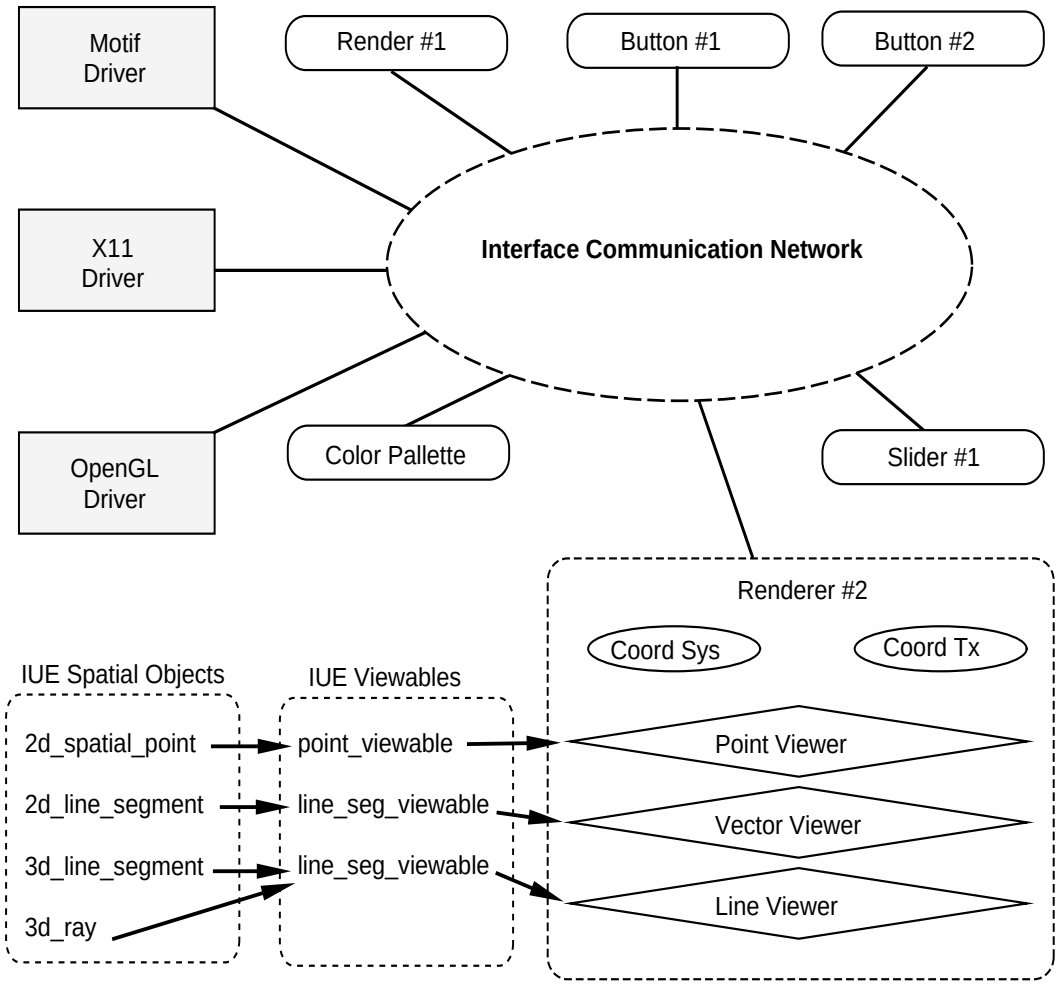


Abbildung 4.11: IUE: Interaktion (aus: [KM94])

Im IUE ist eine riesige Anzahl von Klassen definiert, aber die Objektorientierung bezieht sich nur auf die Datenobjekte selbst. Die Möglichkeit, Wissen über die Operatoren zu sammeln und für auf dem IUE aufsetzende Werkzeuge wie grafische Editoren oder wissensbasierte Unterstützungswerkzeugen verfügbar zu machen, fehlt bislang.

IUE ist in drei Schichten unterteilt, die Systemarchitektur ist in Abbildung 4.10 dargestellt (siehe [LM90, M+93, IUE95]). Im IUE-Paper [IUE95] wird ferner darauf hingewiesen, daß eine eigentliche Bildanalyse-Bibliothek nicht Bestandteil des IUE ist. Die ARPA-Planungen sehen vor, für die Pixelebene die Khoros zugrundeliegende Bibliothek einzubinden, Amerinex hat eine Zusammenführung von KBVision und IUE angekündigt.

Leider war zur Zeit des Systemvergleichs keine lauffähige Version des IUE verfügbar, wie schon dargestellt, ist weder die Bibliothek noch eine grafische Program-

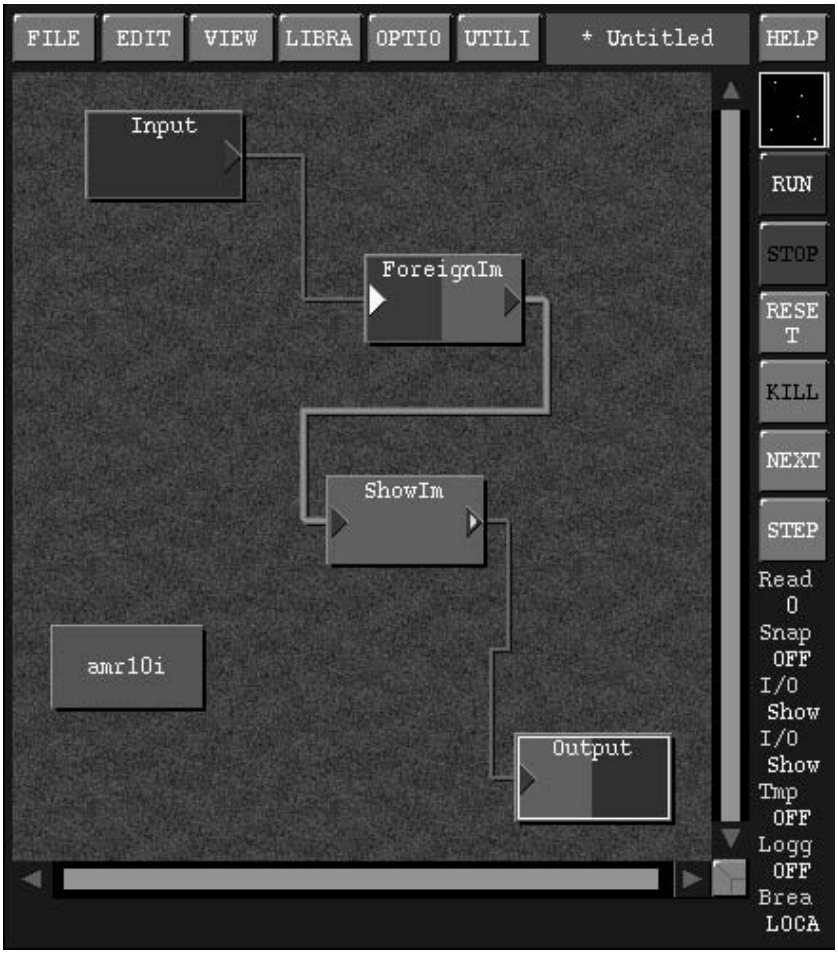


Abbildung 4.12: KBVision: Datenflußgraph (aus: [Jak95])

mieroberfläche integraler Bestandteil des Systems. Laut Kohl und Mundy [KM94] sind lediglich Visualisierungs- und Interaktionskomponenten vorgesehen (siehe Abbildung 4.11). So stellt das IUE nur eine Teilfunktionalität, nämlich den objektorientierten Rahmen für Bildanalysesysteme zur Verfügung. Aus diesen Gründen wurde auch hier auf einen Test verzichtet, trotzdem im Sinne der angestrebten Zielrichtung dieser Arbeit auf der Basis von IUE in der Zukunft realisierbare Systeme mit Sicherheit sehr interessant sein werden.

4.1.5 KBVision

Knowledge-Based Vision (KBVision) ist ein kommerzielles Produkt von Amerinex Artificial Intelligence, Inc., das eine Entwicklungsumgebung für das Gebiet des Bildverstehens darstellt. Diese Umgebung enthält grundlegende Komponenten für die Entwicklung eines wissensbasierten Systems zum Bildverstehen enthält

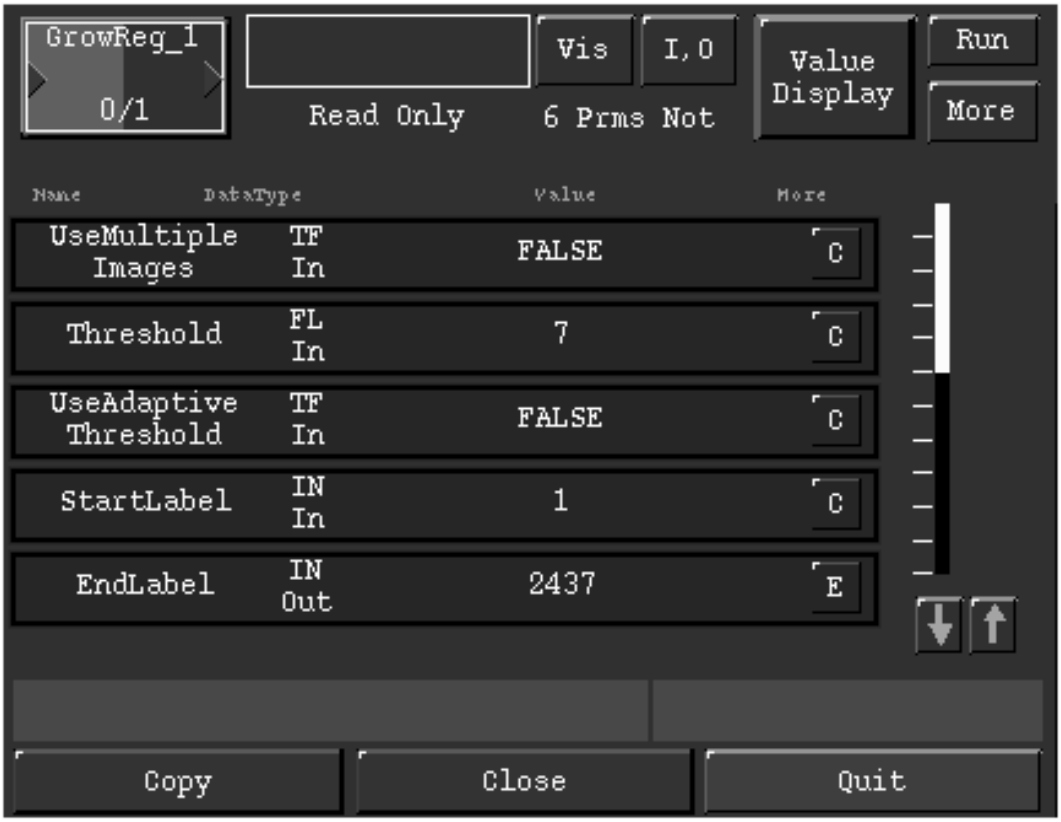


Abbildung 4.13: KBVision: Parametrierung (aus: [Jak95])

[Ame93, Wil90]. In Abbildung 4.12 ist wiederum zunächst ein Datenflußgraph zu sehen, danach in Abbildung 4.13 die Parametrierung. Auch hier ist, wie in Explorer die Möglichkeit vorhanden, grafische Eingabewerkzeuge zu verwenden (Schieberegler rechts im Bild). Auch dieses System wurde im Systemvergleich berücksichtigt [Jak95].

4.1.6 Khoros/Cantata

Khoros ist ein an der University of New Mexico entwickeltes System, auf dessen Basis mit Cantata eine grafische Oberfläche aufsetzt [RA91, RW91, RY92]. Khoros hat gerade im Bereich Bildanalyse relativ weite Verbreitung gefunden, ohne speziell für dieses Gebiet entworfen worden zu sein. Einige der Entwickler kamen jedoch aus dem Umfeld der Bildanalyseforschung, so daß die zunächst immer mitgelieferte Bibliothek von Anwendungsoperationen (das Anwendungsgebiet ist bei Khoros im Prinzip nicht festgelegt) eine Bildanalysebibliothek war. In Abbildung 4.14 ist ein Datenflußgraph dargestellt. Zu bemerken ist hier die zumindest noch in Khoros Version 1.0 fehlende Möglichkeit, automatische Neuberechnungen

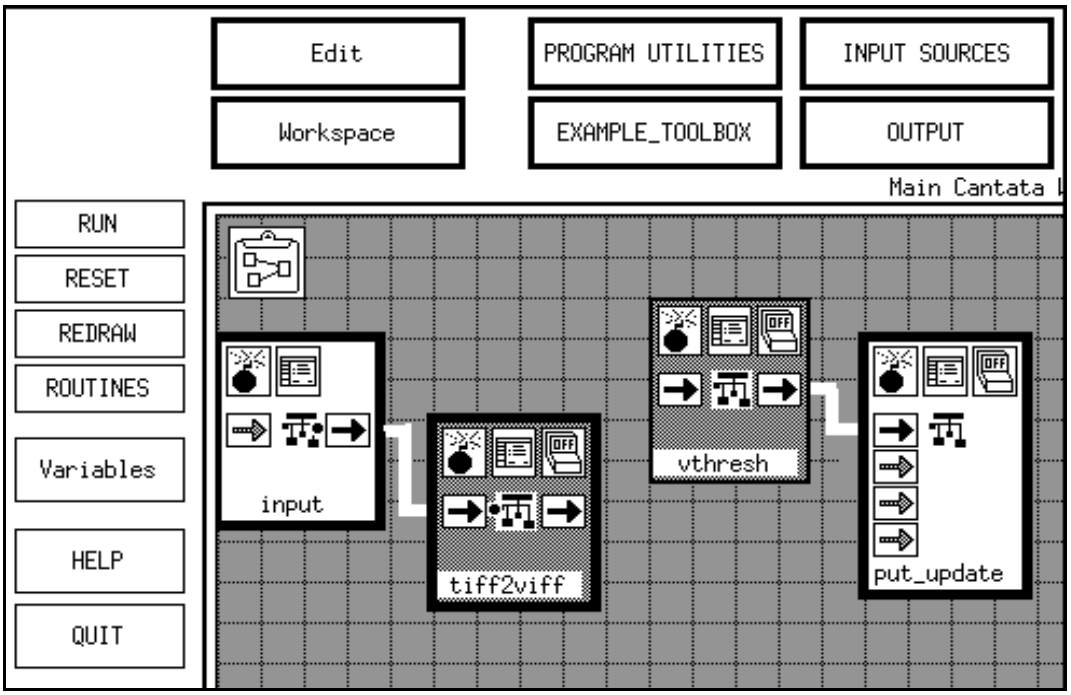


Abbildung 4.14: Khoros: Datenflußgraph (aus: [Jak95])

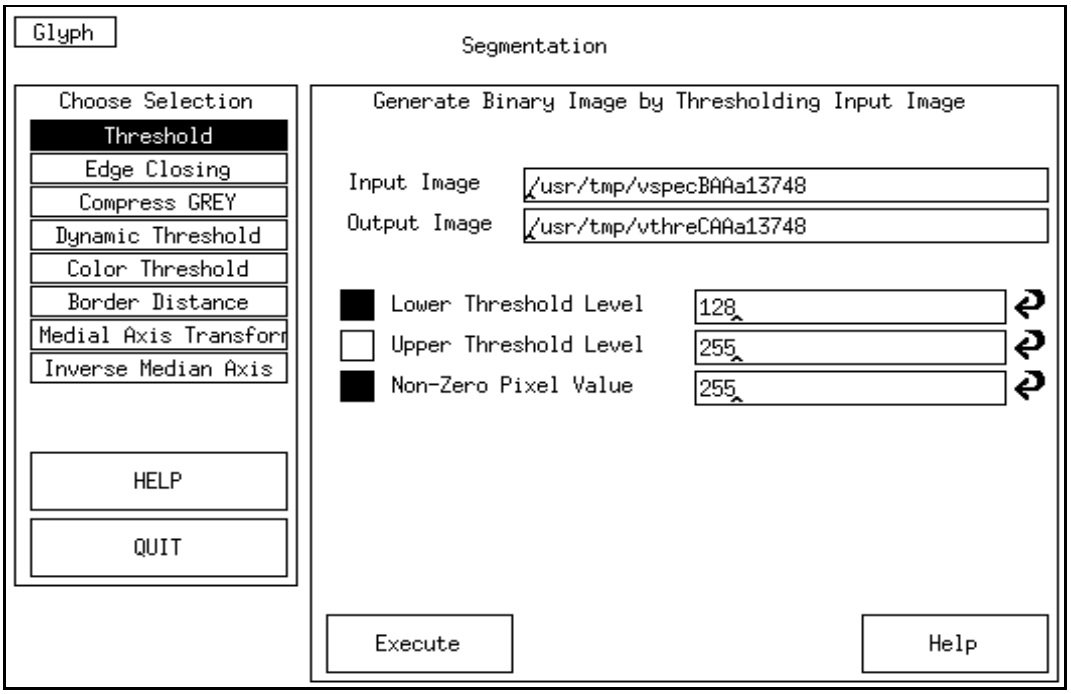


Abbildung 4.15: Khoros: Parametrierung (aus: [Jak95])

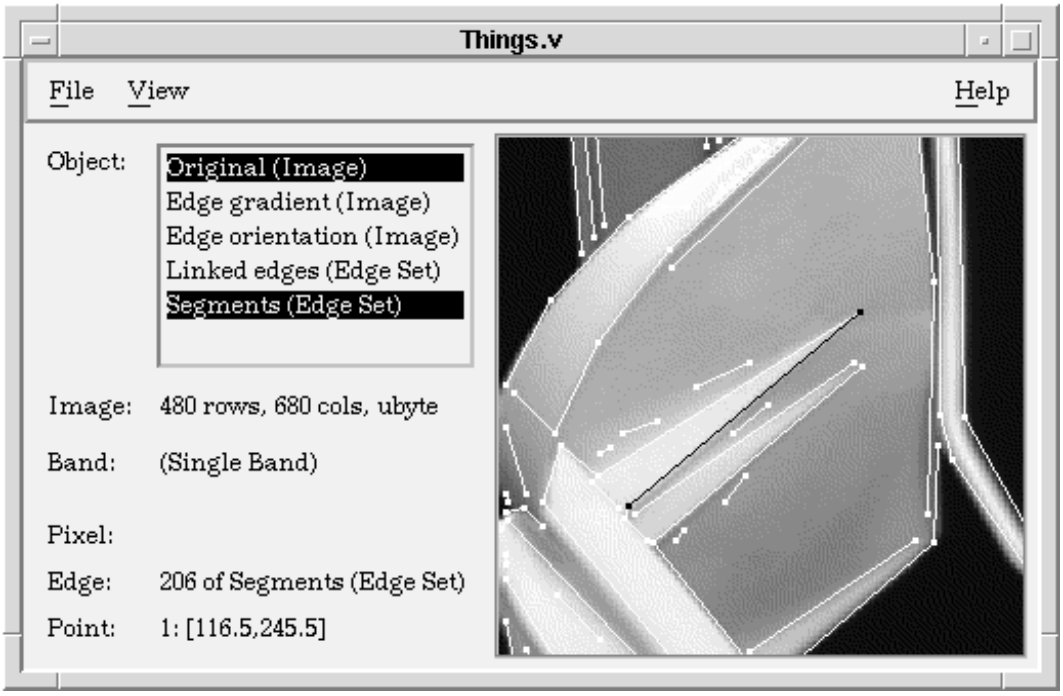


Abbildung 4.16: Vista: Visualisierer (aus: [PL94])

nur der invalidierten Teilgraphen zu machen (siehe Abschnitt 3.8). Die mittlerweile verfügbare Version 2 soll dieses Manko beheben. Bei der Parametrierung (Abbildung 4.15) müssen alle Parameter textuell vorgegeben werden, es existieren keine Vorgabewerte und keine Angaben über Minima oder Maxima.

4.1.7 LabVIEW

Die *Laboratory Virtual Instrument Engineering Workbench* (LabVIEW) ist ein System, das zur Simulation von Laboraufbauten und -geräten (beispielsweise Oszillographen) mit elektrotechnischer Zielrichtung entworfen wurde, das aber auch in anderen Bereichen Anhänger gefunden hat [Gre95, VW86]. Die Eignung seiner Bibliotheken für Bildverarbeitungs- und -analysezwecke ist aber sehr gering, weshalb es nicht in den Systemtest aufgenommen wurde.

4.1.8 Vista

Vista wurde an der University of British Columbia seit 1990 entwickelt [SA90, PL94]. Es verfügt über Komponenten zur Visualisierung (siehe Abbildung 4.16) und

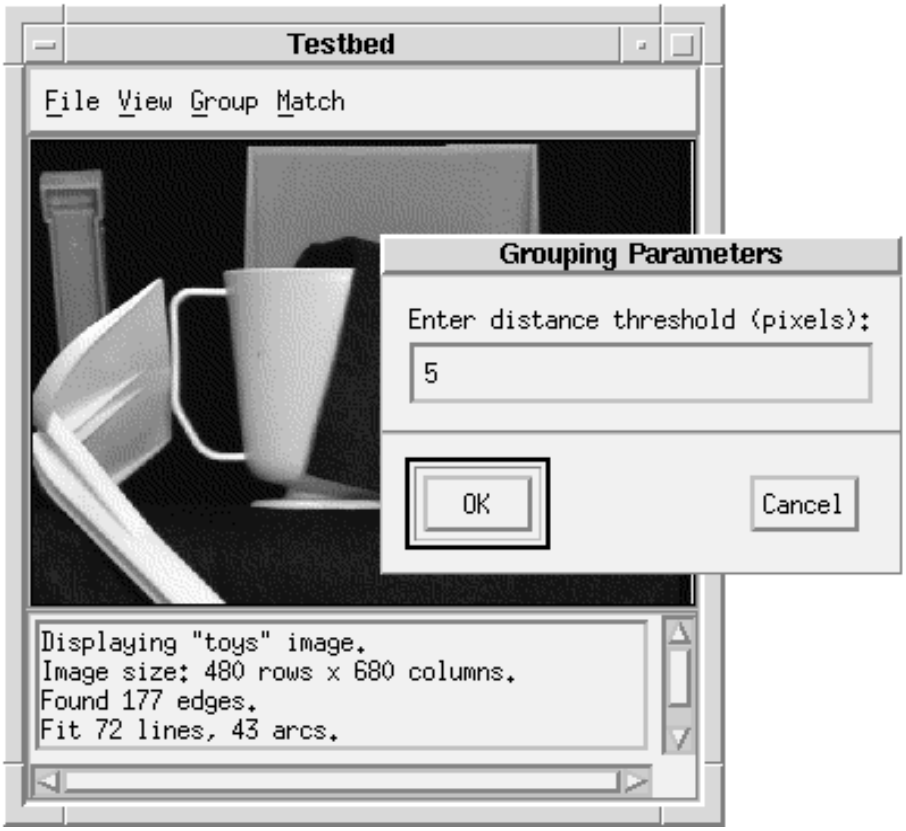


Abbildung 4.17: Vista: Parametrierung (aus: [PL94])

eine sogenannte *VX*-Erweiterung, die aber nur bestimmte *Primitive*[1] zur Verfügung stellt, um auch interaktive Parametrierungen vornehmen zu können (siehe Abbildung 4.17). Hierbei wird aber keine automatische Ableitung der Darstellungsparameter vorgenommen.

4.1.9 Sonstige Systeme

Einige andere Systeme, wie IDL [IDL90, IDL93], DIAS [VS91], Prograph [Gre95], OLIVE [FLD90], Signal [BG90, GGBM91], Spider [SPI83], Lustre [HCRP91] und MAVIS [OKHC92] wurden von vornherein ausgeklammert, entweder, weil ihre Eignung für Bildanalysezwecke nicht gegeben war (beispielsweise Prograph, OLIVE und MAVIS), oder weil sie keine visuellen Programmierumgebungen bieten (wie z.B. DIAS).

[1] Grafische Grundelemente, aus denen Sichten zusammengesetzt werden können, z.B. Schieberegler, Rollbalken, Knöpfe, Dateiselektoren

4.2 Diskussion der bisherigen Ansätze

Wie aus der Aufzählung ersichtlich ist, haben praktisch alle bisherigen Ansätze spezifische Nachteile.

Vergleiche wie [HBM+94] beziehen sich normalerweise auf die Umsetzbarkeit von typischen Programmfiguren wie etwa dem *Sieb des Eratosthenes*, sind jedoch für die Beurteilung von so spezifischen Problemen, wie sie typischerweise in der Bildanalyse auftreten, wenig geeignet. Unter anderem aus diesem Grund wurde eine Diplomarbeit [Jak95] mit dem Vergleich ausgeschrieben. In dieser Arbeit wurden vier grafische Systeme einer intensiveren Analyse bezüglich Ihrer Fähigkeiten im Bereich Bildanalyse unterzogen, nämlich AVS, Explorer, Khoros und KBVision. Hierzu wurde ein Bildanalyseproblem des in Abschnitt 2.2 beschriebenen Typs mit dem jeweiligen System implementiert. AVS hatte hierbei Schwierigkeiten mit der Mächtigkeit der Bibliothek (speziell im Bereich Morphologie), weil dieses System speziell zur Visualisierung und nicht zur Analyse geschaffen wurde. Explorer ist ebenfalls eher dem Bereich „wissenschaftliche Visualisierung“ zuzuordnen und hatte ähnliche Probleme wie AVS bei der Mächtigkeit der zugrundeliegenden Bildanalysebibliothek. Die beiden Systeme, die am besten den in Kapitel 3 gestellten Anforderungen gerecht werden konnten, waren Khoros und KBVision. In Khoros ließ sich aufgrund der relativ großen Bildanalysebibliothek die Aufgabe ohne Probleme lösen. KBVision war das beste System im Test, weil es anders als die übrigen Systeme sogar einfache Hilfsfunktionen zur Verfügung stellt.

Kapitel 5

Grundgedanken

Neben den in den vorangehenden Kapiteln angesprochenen Entwurfsprinzipien gibt es noch einige Konzepte, die nicht notwendigerweise aus den eingangs aufgestellten Anforderungen folgern, bzw. Arbeitsmittel und Werkzeuge, die Anwendung gefunden haben und nachfolgend zusammengetragen sind.

5.1 Das Bildanalysesystem HORUS

5.1.1 Überblick

HORUS ist ein Bildanalysesystem, das in der Forschungsgruppe Bildverstehen des Instituts für Informatik der Technischen Universität München seit 1985 entwickelt worden ist [Eck88].

Als Basis für die Implementierung eines grafischen Editors bot sich HORUS neben seiner Verfügbarkeit in Quellform auch aufgrund anderer Eigenschaften, etwa dem Bereitstellen von Wissen über Bildanalyseoperatoren (siehe Abschnitt 6.1), an.

Neben einer enormen Anzahl von Bildanalyseoperatoren liegen die besonderen Fähigkeiten von HORUS in der sehr effizienten Bearbeitung von Regionendaten mit morphologischen Operatoren, die aufgrund einer geschickten Wahl von zugrundeliegenden Datenstrukturen möglich ist.

In Abbildung 5.1 werden die von HORUS abgedeckten Bereiche dargestellt.

5.1.2 Bestandteile eines HORUS-Operators

Was in HORUS einen Operator ausmacht, ist im Gegensatz zu anderen Systemen nicht lediglich die syntaktische Repräsentation in einer Programmiersprache, sondern die Gesamtheit aus:

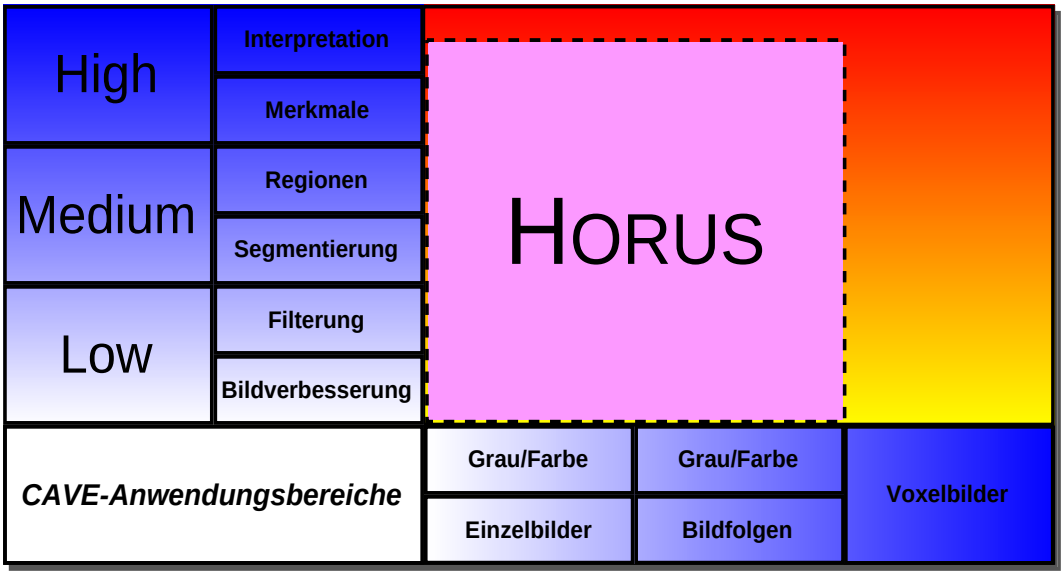


Abbildung 5.1: HORUS-Anwendungsgebiete innerhalb von CAVE

- Syntaktischer Repräsentation (Programmcode)
- Textueller Erklärung (Handbucheintrag)
- Wissen über:

  Anzahl und Art von Parametern: Diese werden benutzt, um die Anzahl der im nächsten Schritt anwendbaren Operatoren einzuschränken. Beispielsweise kann auf ein Regionenbild kein Filter anwendet werden.

  Wertebereich oder Liste gültiger Werte Für eine *echte* grafische Oberfläche ist diese Information absolut notwendig, um z.B. den Wertebereich von Schiebereglern vorzugeben.

  Wirkungszusammenhang: Einige Operatoren haben einen von einem Eingabeparameter linear abhängigen Effekt, andere einen eher logarithmischen oder exponentiellen. Diese Information kann sowohl für die grafische Darstellung als auch für eine wissensbasierte Unterstützung ausgenutzt werden.

  Gruppenzugehörigkeit: Es existiert eine mehrdimensionale Unterteilung der Operatoren in sogenannte *Gruppen*, die ähnlich wie eine Klassenhierarchie aufgebaut, jedoch in Ihrer Struktur nicht auf einen Baum beschränkt ist (siehe Abschnitt 5.5).

  Relationen zwischen Operatoren: Hier sind die verschiedensten Relationen gemeint:

Vorgänger und Nachfolger: Viele Operationen haben bestimmte Voraussetzungen für ihre Anwendbarkeit. Beispielsweise muß vor Anwendung einer dynamischen Schwelle ein Tiefpaßfilter angewendet worden sein. Ebenso gibt es Operatoren, die typischerweise erst *nach* anderen angewendet werden.

Ähnliche Operatoren und Querverweise: Für einige Operatoren existieren ähnliche Alternativen (z.B. `mean` und `gauss`), außerdem sind Verweise auf andere Operatoren vorhanden, die einen Bezug haben.

5.2 Objektorientierter Ansatz

Um einen grafischen Editor zu schaffen, ist es nicht mehr sinnvoll, in einer imperativen Sprache wie `C` zu programmieren [KR90]. Die Möglichkeiten, die unter `C` und X-Windows [NO92] geboten werden, sind selbst mit moderneren Werkzeugen wie Motif [OSF90] callback-orientiert und damit auf zu niedriger Ebene angesiedelt. Auch die Verwendung der verfügbaren Werkzeuge zur interaktiven Erstellung von Oberflächen ändert an diesem prinzipiellen Problem nichts, denn die Möglichkeit, grafische Elemente zu plazieren und zu modifizieren, wird lediglich dem Programmierer zur Zeit der Erstellung der Applikation, nicht aber dem Benutzer zur Laufzeit des mit dem Werkzeug erstellten Programms eingeräumt.

Er kann die Elemente weder verschieben noch verändern oder gar erzeugen und löschen.

Was alle diese Werkzeuge nicht beherrschen, ist die *Meta-Ebene*, ein Problem, das sich mit herkömmlichen (kompilierten) Sprachen auch nicht lösen läßt, weil die Ebene der Beschreibung nicht die der Ausführung ist.

Der große Unterschied von „angeblichen“ und echten grafischen Oberflächen wird in [VM94] herausgestellt, indem die Autoren [MR93] zitieren: *”What the market considers a GUI is little more than a glorified menu system, having no graphics. This leaves the graphical representation of the application domain as an exercise for the developer, …”*

Aus Gründen der Komplexität des angegangenen Problems war daher eine komfortablere Sprach- und Entwicklungsumgebung wünschenswert, die außerdem interpretativ arbeiten mußte.

Der zweite wichtige Grund für die Verwendung des objektorientierten Ansatzes war die Notwendigkeit von Persistenz, die z.B. in VisualWorks a priori gegeben ist.

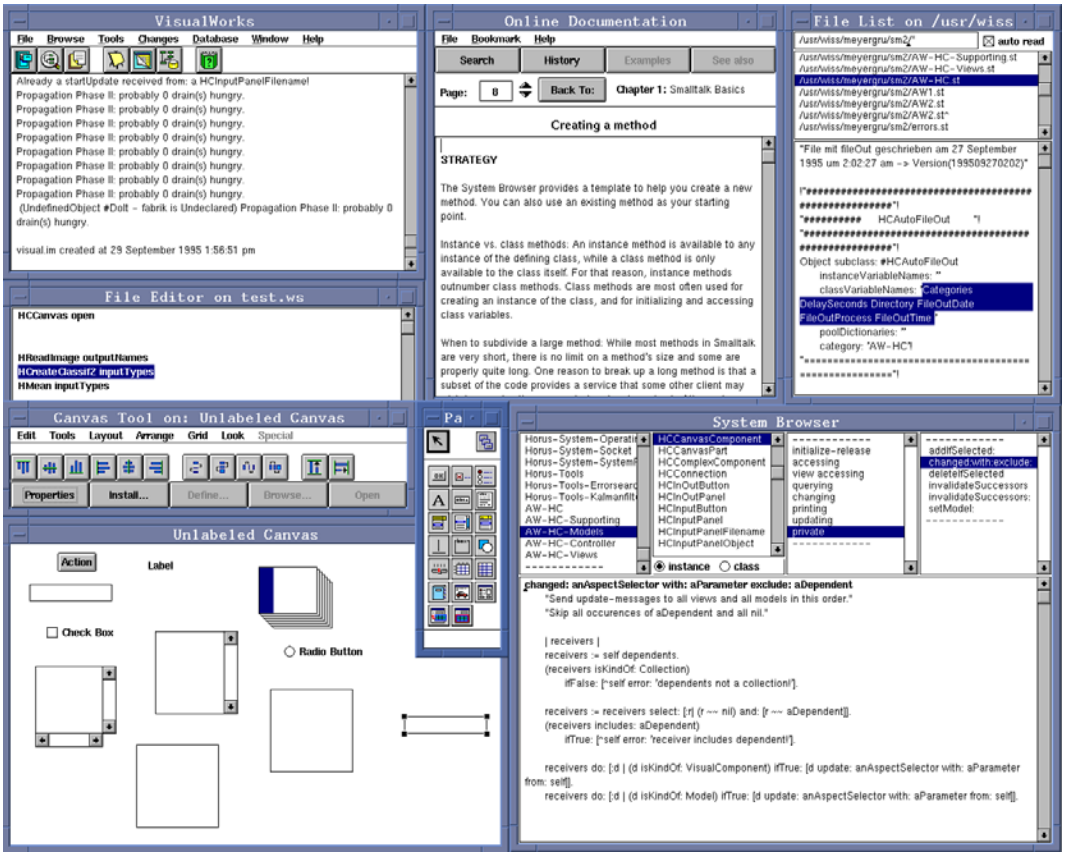


Abbildung 5.2: Programmierumgebung VisualWorks

5.3 Sprachumgebung

Die Sprache `Smalltalk` wurde 1980 von Adele Goldberg am Xerox PARC entwickelt. Für eine sehr gute Darstellung der Konzepte und Diskussion der Unterschiede zu anderen objektorientierten Sprachen siehe Gall et. al. in [GHK95]. `Smalltalk` kennt Objekte, die einander Nachrichten schicken, ein Ansatz, der auf sehr natürliche Weise das Verhalten eines interaktiven Systems widerspiegelt, in welchem ebenfalls Ereignisse verteilt werden müssen, so etwa ein Mausklick in einem klar abgegrenzten Teil eines Fensters, normalerweise als *Sicht* bezeichnet.

Ein weiterer Vorteil einer objektorientierten Sprache wie `Smalltalk` ist, daß durch Vererbung die Hierarchie von Bildobjekten spezifizierbar ist, z.B. Grauwertbild $\rightarrow$ Farbbild $\rightarrow$ Farbbildfolge. So läßt sich ein für Grauwertbilder definierter linearer Filter durch kanalweise Anwendung auch auf Farbbilder und durch Anwendung auf jedes Einzelfarbbild ebenso auf Farbbildfolgen anwenden.

Schon in der Urprungsversion war das Besondere an der Sprachbeschreibung von `Smalltalk`, daß sich diese in zwei Teile gliederte:

1. einen Band für die bei Programmiersprachen übliche syntaktische und semantische Definition [Gol85] und zusätzlich

2. einen Band für die Beschreibung des dazugehörigen Laufzeit- und Entwicklungssystems [Gol84], das im wesentlichen heute noch das selbe ist wie 1980.

In der verwendeten Version VisualWorks, die 1992 entwickelt wurde, ist der wesentliche Anteil nicht die virtuelle Maschine, sondern die riesigen Systembibliotheken, von denen sich ein großer Teil mit der Behandlung von Darstellung und Interaktion befaßt [VWOR94, VWUG94, VWCB94].
In VisualWorks wird ein sogenanntes *Image* auf einer virtuellen Maschine ausgeführt. In diesem werden alle Informationen über das Laufzeitsystem in Form eines binären Abbilds abgelegt und beim Beenden von VisualWorks auch wieder gespeichert, so daß Konzepte wie Persistenz von Objekten möglich werden. Da die kompletten Systembibliotheken selbst in `Smalltalk` geschrieben sind und mit ausgeliefert werden, kann der Benutzer das gesamte Systemverhalten analysieren und sogar ändern. Das geht im Extremfall so weit, daß die Bedeutung der Nachricht „+“ verändert werden könnte, natürlich mit fatalen Folgen, wenn es die falsche ist.
Der prinzipielle Nachteil der ständigen Interpretation jeden Teils des Laufzeitsystems wiegt aus zwei Gründen nicht so schwer, wie man bei einer so laufzeitkritischen Anwendung wie der Bildanalyse zunächst vermuten könnte:

1. Mit VisualWorks ist es möglich, `C`-Routinen einzubinden, was es ermöglichte, das dieser Arbeit zugrundeliegende Basissystem HORUS, das selbst in `C` implementiert ist, einzubinden [Eck93].

2. VisualWorks selbst kompiliert Methoden bei der ersten Ausführung und stellt das Kompilat in einen Cache. Auf diese Art werden Schleifen fast wie reines Kompilat ausgeführt.

Der einzige Punkt, der wirklich wesentlich langsamer ist, als bei einer maschinennahen Implementierung mit `C`, ist die eigentliche Visualisierung von Bildern, die normalerweise auch in (interpretiertem) `Smalltalk`-Code ausgeführt wird. Aus diesem Grund wurde auf die HORUS-Visualisierung in einem separaten Fenster zurückgegriffen, wenn die Bilder nicht miniaturisiert dargestellt werden.

5.4 Operatoren als Agenten

In `Smalltalk` ist es üblich, die Daten als Objekte zu behandeln, auf die Methoden anwendbar sind. Ein Bildverarbeitungsoperator wie ein Schwellwertoperator würde sich also zwangsläufig als Methode `threshold` mit mehreren Parametern, anwendbar auf eine Instanz der Klasse *Grauwertbild* präsentieren.

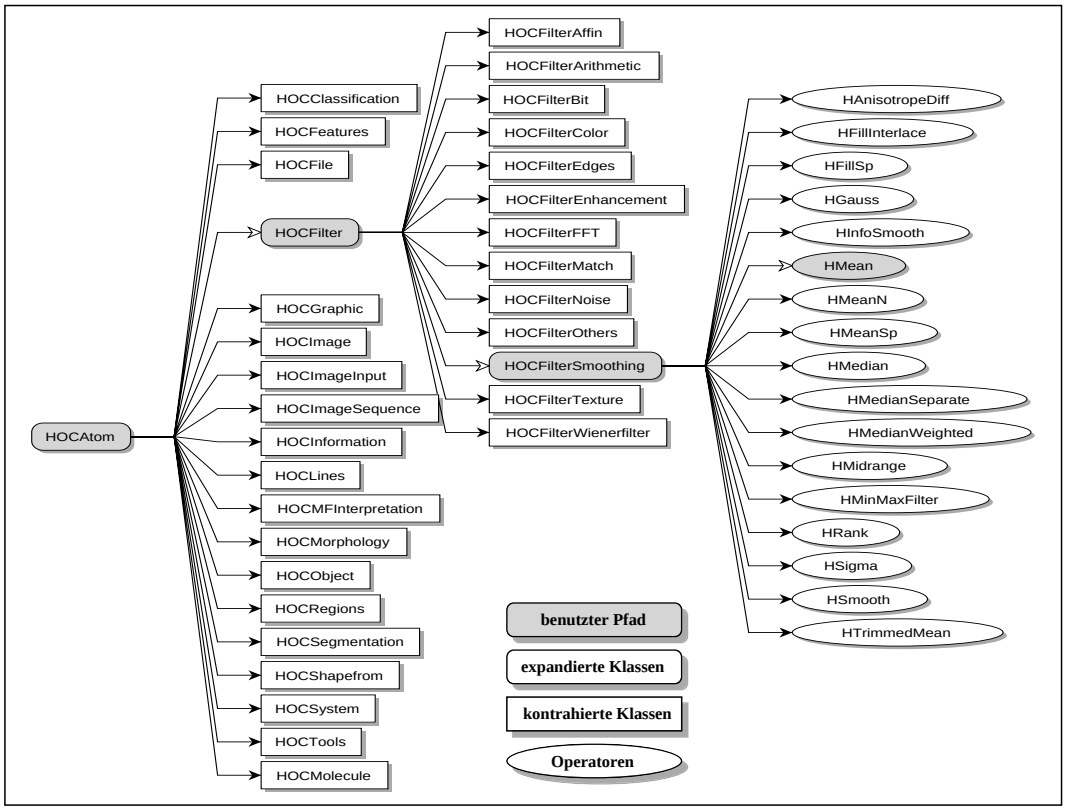


Abbildung 5.3: HORUS-Klassenhierarchie (größtenteils kontrahiert)

Eigentlich ist dabei die Objektorientierung noch nicht zu Ende gedacht, denn bei diversen Diskussionen über HORUS wurde klar, daß HORUS-Operatoren Agentencharakter haben. Wenn in einer Datenbasis Wissen über die Operatoren abgelegt werden, liegt es nahe, auch die Operatoren selbst als Objekte aufzufassen, die mit Datenobjekten (Bildern und Regionen) interagieren.

Jeder HORUS-Operator wurde also als eigene *Klasse* innerhalb von `Smalltalk` realisiert. Die (größtenteils kontrahierte) Hierarchie ist bis hinunter zum Operator `mean` in Abbildung 5.3 zu sehen.

Damit ist es möglich, jeden Operator mit einer Methode *do* zur Anwendung zu bringen, mit einer Methode *comment* nach seiner (textuellen) Dokumentation zu fragen oder mit anderen Methoden die verschiedensten Informationen zu erhalten, die für Planungskomponenten wichtig sein können, aber auch für den Editor selbst. So läßt sich u.a. die Anzahl der Parameter sowie deren Namen und Datentypen anfragen.

Die Frage, die hierbei sofort aufgeworfen wird, ist: „Wenn für jeden Operator eine Klasse geschaffen wird, gibt es dann auch Instanzen?“

Die Antwort darauf lautet ja, denn in einer Oberfläche kann es zur gleichen Zeit mehrere Instanzen von Schwellwertoperatoren geben, die „wissen“, wo auf der *Lein-*

wand[1] sie positioniert sind, oder wie ihre Eingangsparameter mit entsprechenden Werten belegt sind, mehr noch, sie können Informationen darüber haben, mit welchen anderen Operatoren sie verknüpft sind.

Interessanterweise ist die Idee, *Operatoren als Objekte* aufzufassen, bis jetzt noch nirgends aufgetaucht, obwohl es naheliegt, Objektorientierung auf das Gebiet der Bildanalyse anzuwenden [Pau92]. Auch im bis jetzt umfassendsten objektorientierten Ansatz im Bereich Bildanalyse, im IUE sind diese Gedanken noch nicht aufgenommen [IUE95], und können auch aufgrund von Implementierungsdetails nicht ohne weiteres umgesetzt werden. Entgegen der ursprünglich gewünschten Integration von `LISP` und `C++` ist in der zur Zeit aktuellen Version nur eine statische Klassenbibliothek in `C++` enthalten, was aufgrund fehlender Persistenz (zumindest auf der Sprachebene selbst) einigen Mehraufwand bei einer etwaigen Umstellung ergäbe.

Es gibt viele Implementierungsdetails, die sich durch die Anwendung dieser Technik „wie von selbst“ lösen, z.B. die von Atkins [AZJA94] und Poswig et. al. [PVM93] geforderte Möglichkeit, die Elemente auf der Leinwand zu animieren. Diese Fähigkeit kann auf sehr hoher Ebene allen Operator-Objekten vermittelt werden.

5.5 Nomenklatur für Operatoren

Nachdem der Agentencharakter von HORUS-Operatoren durch Darstellung als Klassen in `Smalltalk` klar war, mußten dem Gesamtsystem einige Fähigkeiten hinzugefügt werden, u.a. diejenige zur Wiedereinstellung von Operatoren in die Operator-Datenbasis (siehe Abschnitt 7.6). Zur Vereinfachung des Sprachgebrauchs auch innerhalb der Forschungsgruppe Bildverstehen wurde eine neue Nomenklatur eingeführt.

In Anlehnung an die Bezeichnungen aus der Chemie werden im weiteren für verschiedene Klassen von Operatoren folgende Begriffe verwendet:

Gruppe bezeichnet eine Klasse von Operatoren, z.B. existieren in HORUS die Gruppen „Kantenfilter“ und „Segmentierer“. Jeder konkrete Operator stellt insofern eine Instanz einer oder mehrerer Gruppen dar. Die Klassenhierarchie ist jedoch nicht notwendigerweise ein Baum, da durch die Zugehörigkeit zu einer Gruppe eine bestimmte Wirkungsweise eines konkreten Vertreters (Instanz) ausgedrückt werden soll und weil diese Vertreter eventuell mehrere Wirkungen haben, weshalb sie mehreren Gruppen zugeordnet sein können.

[1] Abgegrenzter Bereich, der es erlaubt, Objekte grafisch zu manipulieren.

Atom als kleinstes „unteilbares“ Element bezeichnet einen HORUS-Operator, der in der Basisbibliothek enthalten und aus Sicht des Anwenders und des grafischen Editors nicht mehr in Einzelschritte zerfällt. Neben den in `C` implementierten HORUS-Operatoren gehören hierzu auch in `Smalltalk` hinzugefügte Spezialoperatoren, z.B. für arithmetische Operationen oder interaktive Datenquellen und -senken.

Molekül bedeutet eine Abfolge von konkreten (d.h. voll instantiierten) Operatoren, die mittels bestimmter semantischer Konstrukte zu einem vom HORUS-Kern ausführbaren, zusammengesetzten Operator werden. Vereinfacht könnte man sich ein Molekül als Programm vorstellen, das nach außen aber dieselben Fähigkeiten wie ein Atom aufweist, d.h. Wissen über Parameter, Wirkungsweise und so weiter.

Meta-Molekül ist ein Molekül, bei dem mindestens eine beteiligte Operation nicht voll instantiiert ist und von dem beispielsweise nur bekannt ist, welcher Gruppe ein bestimmter, im Meta-Molekül vorkommender Operator angehören soll.

5.6 Verhalten von Objekten statt Algorithmen

Diese Trennung manifestiert sich in vielen Beiträgen im Bereich der visuellen Programmiersprachen, die sich mit völlig abgegrenzten Einzel*programmen*[2] beschäftigen. So werden Programme wie ein Visualisierer für eine visuelle Sprache entwickelt, der das semantische Netz, das in einer bestimmten Beschreibung vorliegt, mit einem Layout versieht und darstellt. Obwohl solche Funktionen natürlich in grafischen Programmiersystemen benötigt werden, zeigt sich hier das dahinterliegende Paradigma. Nur bei normalen, imperativen Programmiersprachen ließ sich das zugrundeliegende System modular durch Hinzufügen von Programmen für bestimmte Spezialzwecke[3] erweitern. In einem echten grafischen Programmiersystem sollte aber dynamisch ein ständiges Update des Layouts stattfinden und dies erfordert Erweiterungen der Funktionalität des Grundsystems selbst.

Solche Vorgehensweise setzt auch andere Algorithmen voraus. Um bei dem Beispiel der Layout-Komponente zu bleiben, ist es vernünftiger, wenn keine globale Routine existiert, die alle Objekte neu ordnet, sondern wenn jedes *einzelne* Objekt „weiß“, wo es hingehört. Bei objektorientierten Ansätzen wird eigentlich nur das Verhalten des Einzelobjekts beschrieben. Ein gutes Beispiel für diesen Ansatz ist der Propagationsmechanismus in Abschnitt 7.1.

[2] im Gegensatz zu Einzel*problemen*

[3] z.B. ein Pretty-Printer, der für herkömmliche Programmiersprachen das Äquivalent eines Layouters ist

5.7 Turing-Vollständigkeit

Aus den in Abschnitt 3.4 genannten Gründen ist für die weitere Vorgehensweise vorerst auf die konkrete Implementierung einer turing-vollständigen Sprache verzichtet[4], und eine dem Problemfeld angemessene, pragmatische Lösung angestrebt worden. Die damit eingehandelten Nachteile werden wieder wettgemacht, weil die prototypischen Lösungen in einer herkömmlichen Programmiersprache weiterbearbeitet werden und so bis zur Anwendungsreife gelangen können. Die für die Low-Level-Ebene selbstverständlich notwendige Turing-Vollständigkeit wird im weiteren zwar als vorhanden, aber gemäß eines Schichtenmodells in der Implementierungsschicht unterhalb der Oberfläche verborgen vorausgesetzt. Dies geschieht bewußt und trotz der Tatsache, daß einige Autoren es als sinnvoll erachten, eine Durchgängigkeit der verwendeten Sprache in allen Schichten zu gewährleisten (siehe hierzu auch Abschnitt 3.4, Seite 23).

[4] Obwohl die dafür notwendigen Konstrukte in Abschnitt 7.3 theoretisch erarbeitet werden.

Kapitel 6

Wissensverarbeitung

Der Frage der Wissensverarbeitung kommt aufgrund der in Kapitel 3 gesetzten Ziele eine besondere Bedeutung zu, weshalb ihr nachfolgend ein ganzes Kapitel gewidmet wird.

6.1 HORUS-Operator-Datenbasis

HORUS verfügt heute über eine enorme Vielfalt an Operationen (ca. 650). Neben der entsprechenden Menge an Quellcode (ca. 6 MByte) für die eigentliche Operatorbibliothek existieren nochmals etwa 2,5 MByte Daten in der sogenannten *Operator-Datenbasis*, in der verschiedenste Informationen enthalten sind, wie Handbucheintrag, Art und Anzahl der Parameter, Wirkungszusammenhänge, Minimal- und Maximalwerte, Querverweise, Nachfolger und Vorgängerfunktionen und vieles andere mehr. Es folgt als Beispiel der Eintrag für den `threshold`-Operator:

```
/************************  threshold__   *******************************/
threshold__ <- CSIPThreshold[Image:Region:MinGrey,MaxGrey:]

short.german
  Selektion aller Grauwerte innerhalb eines Intervalls.;

abstract.german
  \BefehlRef{threshold__} w"ahlt aus den Eingabebildern die Bildpunkte aus,
  deren Grauwerte $g$ der Schwellwertbedingung \\
  @a
              MinGrey <= g <=  MaxGrey
  @l
  \[
  \ParamRef{MinGrey}  \le  g  \le  \ParamRef{MaxGrey}
  \]
  @e
  gen"ugen. \\
  Alle Punkte eines Eingabebildes, die die Bedingung erf"ullen, werden
  gemeinsam als ein neues Bild abgespeichert.
  Wird mehr als ein Grauwertbereich "ubergeben (Tupel von Werten f"ur
```

```
  \ParamRef{MinGrey} und \ParamRef{MaxGrey}),
  dann wird f"ur jeden dieser Intervalle ein
  Bild erzeugt.;

chapter.german
  Segmentation;

chapter.english
  Segmentation;

functionality
  image;

keywords
  Schwellwert, Grauwertschwelle;

attention.german
  ;

complexity
  Sei $F$ die Fl"ache der Eingaberegion, dann ist die Laufzeitkomplexit"at:
  $O(F)$;

alternatives
  class_2dim__, hysteresis_threshold__, dyn_threshold__;

see_also
  zero_crossing1, zero_crossing2, background_seg, regiongrowing__;

predecessor
  histo_to_thresh, min_max__, sobel_amp, gauss__, reduce_domain,
  fill_interlace__
  ;

successor
  connection, dilation1, erosion1, opening, closing, count, shape_trans,
  skeleton;

example.trias
  read_image(:Image:'fabrik':) >
  sobel_dir(Image:EdgeAmp,EdgeDir:'sum_abs',3:) >
  threshold__(EdgeAmp:Seg:50,255,2) >
  skeleton(Seg:Rand::) >
  connection(Rand:Lines::) >
  select_shape(Lines:Edges:'area','and',10,1000000:).;

example.c
  read_image(&Image,"fabrik")\;
  sobel_amp(Image,&EdgeAmp,"sum_abs",3)\;
  threshold__(EdgeAmp,&Seg,50.0,255.0)\;
  skeleton(Seg,&Rand)\;
  connection(Rand,&Lines)\;
  select_shape(Lines,&Edges,"area","and",10.0,1000000.0)\;
  ;

example.c++
  #include  <iostream.h>
  #include  "HCPP.H"

  int main (int argc, char *argv[])
  {
    if (argc != 4)
```

```
    {
      cout << "Usage : " << argv[0] << " <image> MinGrey MaxGrey" << endl\;
      return (-1)\;
    }

    HImage   image (argv[1]),
             Sobel\;
    HWindow  win\;

    image.Display (win)\;

    int MinGrey = atoi (argv[2])\;
    int MaxGrey = atoi (argv[3])\;

    Sobel = image.SobelAmp ("sum_abs", 3)\;

    HRegionArray rand  = ((image >= MinGrey) & (image <= MaxGrey)).Skeleton()\;
    HRegionArray lines = rand.Connection ()\;
    HRegionArray edges = lines.SelectShape ("area", "and", 10.0, 10000000.0)\;

    edges.Display (win)\;
    win.Click ()\;

    return (0)\;
  }
  ;

result_state.german
  Sind die Parameterwerte korrekt, dann liefert \BefehlRef{threshold__}
  den Wert TRUE.
  F"ur das Verhalten bzgl.\ der Ein- und Ausgabebilder sind die
  Flags 'no_object_result', 'empty_region_result' und
  'store_empty_region' einstellbar (siehe \BefehlRef{set_system}).
  Gegebenenfalls wird eine Exception-Behandlung durchgef"uhrt.;

parameter
  Image:              input_object;
  description.german: Zu segmentierendes Bild.;
  sem_type:           image;
  multivalue:         optional;
  type_list:          byte, int2, int4, real;

parameter
  Region:             output_object;
  description.german: Region mit Punkten, die die Grauwertbedingung erf"ullen.;
  sem_type:           region;
  multivalue:         optional;

parameter
  MinGrey:            input_control;
  description.german: Untere Schwelle f"ur die Grauwerte.;
  sem_type:           number;
  type_list:          integer,real;
  default_type:       real;
  default_value:      128.0;
  values:             0.0,10.0,30.0,64.0,128.0,200.0,220.0,255.0;
  value_min:          0.0;
  value_max:          255.0;
  step_rec:           5.0;
  step_min:           0.01;
  value_function:     lin;
  mixed_type:         false;
```

```
  multivalue:         optional;

parameter
  MaxGrey:            input_control;
  description.german: Obere Schwelle f"ur die Grauwerte.;
  sem_type:           number;
  type_list:          integer,real;
  default_type:       real;
  default_value:      255.0;
  values:             0.0,10.0,30.0,64.0,128.0,200.0,220.0,255.0;
  value_min:          0.0;
  value_max:          255.0;
  step_rec:           5.0;
  step_min:           0.01;
  value_function:     lin;
  mixed_type:         false;
  multivalue:         optional;
```

Der aus diesem Eintrag generierte Handbucheintrag für `Smalltalk` sieht so aus:

HThreshold `image: minGrey: maxGrey:`
`^ region`

aHImage `hThreshold: maxGrey:`

aHImageArray `hThreshold: maxGrey:`

Selektion aller Grauwerte innerhalb eines Intervalls.

`HThreshold` wählt aus den Eingabebildern die Bildpunkte aus, deren Grauwerte g der Schwellwertbedingung

$$\texttt{MinGrey} \leq g \leq \texttt{MaxGrey}$$

genügen.
Alle Punkte eines Eingabebildes, die die Bedingung erfüllen, werden gemeinsam als ein neues Bild abgespeichert. Wird mehr als ein Grauwertbereich übergeben (Tupel von Werten für `MinGrey` und `MaxGrey`), dann wird für jeden dieser Intervalle ein Bild erzeugt.

Parameter

- **Image** (input_object) ..... HImage (Array) $\leftrightarrow$ *ObjType* : byte / int2 / int4 / real
  Zu segmentierendes Bild.
- **Region** (output_object) ..... HRegion (Array) $\leftrightarrow$ *ObjType*
  Region mit Punkten die die Grauwertbedingung erfüllen.
- **MinGrey** (input_control) ..... Number (Array) $\leftrightarrow$ *real* / integer
  Untere Schwelle für die Grauwerte.
  Defaultwert: 128.0
  Wertevorschläge: `MinGrey` $\in$ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0}
  Wertebereich: 0.0 $\leq$ `MinGrey` $\leq$ 255.0 (lin)
  Minimale Schrittweite: 0.01
  Empfohlene Schrittweite: 5.0

▷ **MaxGrey** (input_control) . . . . . Number (Array) ↔ *real* / integer
Obere Schwelle für die Grauwerte.
Defaultwert: 255.0
Wertevorschläge: `MaxGrey` ∈ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0}
Wertebereich: 0.0 ≤ `MaxGrey` ≤ 255.0 (lin)
Minimale Schrittweite: 0.01
Empfohlene Schrittweite: 5.0

Beispiel

```
#include  <iostream.h>
#include  "HCPP.H"

int main (int argc, char *argv[])
{
  if (argc != 4)
  {
    cout << "Usage : " << argv[0] << " <image> MinGrey MaxGrey" << endl;
    return (-1);
  }

  HImage   image (argv[1]),
           Sobel;
  HWindow  win;

  image.Display (win);

  int MinGrey = atoi (argv[2]);
  int MaxGrey = atoi (argv[3]);

  Sobel = image.SobelAmp ("sum_abs", 3);

  HRegionArray rand  = ((image >= MinGrey) & (image <= MaxGrey)).Skeleton ();
  HRegionArray lines = rand.Connection ();
  HRegionArray edges = lines.SelectShape ("area", "and", 10.0, 10000000.0);

  edges.Display (win);
  win.Click ();

  return (0);
}
```

Komplexität

Sei F die Fläche der Eingaberegion, dann ist die Laufzeitkomplexität: $O(F)$

Ergebnis

Sind die Parameterwerte korrekt, dann liefert `HThreshold` den Wert TRUE. Für das Verhalten bzgl. der Ein- und Ausgabebilder sind die Flags 'no_object_result', 'empty_region_result' und 'store_empty_region' einstellbar (siehe `HSetSystem`). Gegebenenfalls wird eine Exception-Behandlung durchgeführt.

Mögliche Vorgängerfunktionen

`HHistoToThresh`, `HMinMax`, `HSobelAmp`, `HGauss`, `HReduceDomain`, `HFillInterlace`

Mögliche Nachfolgerfunktionen

`HConnection`, `HDilation1`, `HErosion1`, `HOpening`, `HClosing`, `HCount`, `HShapeTrans`, `HSkeleton`

Alternativen

`HClass2dim`, `HHysteresisThreshold`, `HDynThreshold`

Siehe auch

`HZeroCrossing1`, `HZeroCrossing2`, `HBackgroundSeg`, `HRegiongrowing`

Im folgenden wird die Bedeutung der wichtigsten Einträge beschrieben:

short (Text): Kurzbeschreibung des Operators.

chapter (Zeichenkette): Kapitelzuordnung des Operators. Dies drückt keine Funktion aus, z.B. ist auch eine Operation `info_edges`, die nur Informationen zum Kantenoperator `edges` liefert, im HORUS-Kapitel „Filter-Kanten“ zu finden.

functionality (Klasse): Klasse der Instanzen, auf die dieser Operator angewendet werden kann.

keywords (Zeichenketten): Schlüsselworte, denen der Operator zugeordnet ist.

attention (Text): Warnungen, z.B. vor häufig vorkommenden Fehlbenutzungen.

complexity (Text): Textuelle Beschreibung der Zeitkomplexität.

alternatives (Operatoren): Verweis auf Operationen, die ähnliche Funktionalität aufweisen.

see_also (Operatoren): Querverweise auf Operatoren, die in Zusammenhang mit diesem Operator stehen. Beispiel: `select_shape`(”anisometry”) steht in Bezug zu `anisometry`.

predecessor (Operatoren): Verweis auf Vorgängeroperatoren, also Operatoren, die normalerweise vor diesem Operator angewendet werden müssen. Beispiel: Mittelwertfilter `mean` vor dynamischer Schwelle `dyn_threshold`.

successor (Operatoren): Verweis of Nachfolgeroperationen, d.h. Operatoren, die normalerweise nach diesem Operator angewendet werden können. Beispiel: Schwellwert `threshold` nach Sobelfilter `sobel`.

example (Programmtext): Anwendung des Operators als Programmierbeispiel.

result_state (Text): Textuelle Beschreibung der Wirkung des Operators.

parameter: Außer den Einträgen für Informationen, die sich auf den gesamten Operator beziehen sind für jeden Parameter einzeln zusätzliche Informationen abgelegt:

<**Name**>:<**Typ**>**:** Parametername und -typ.

description (Text): Beschreibung des Parameters.

sem_type (Typ): Semantischer Datentyp.

multivalue (Boolean): Angabe, ob mehrere Werte verarbeitet werden können, z.B. Listen von Regionen.

type_list (Typen): Liste erlaubter Datentypen, z.B. Real, Integer.

default_type (Typ): Bevorzugter Datentyp.

default_value (Wert): Vorgabewert.

values (Werte): „Gute“ Werte.

value_min (Zahl): Minimal zulässiger Wert.

value_max (Zahl): Maximal zulässiger Wert.

step_rec (Zahl): Empfohlene Schrittweite.

step_min (Zahl): Minimale Schrittweite.

value_function (Zeichenkette): Wirkungszusammenhang, z.B. linear, exponentiell oder logarithmisch.

mixed_type (Boolean): Angabe, ob gemischte Datentypen zugelassen sind.

Die beschriebenen Einträge können, soweit es sich um rein textuelle Informationen handelt, optional in verschiedenen Landessprachen (angezeigt durch das Suffix **.german** oder **.english**), im Fall von Programmierbeispielen für verschiedene Zielsprachen angegeben sein (Suffix **.lisp**, **.c**, **.c++**, **.trias**, **.prolog** oder **.smalltalk**).

Neben den kompletten Handbuchinformationen lassen sich die Querverweise (**see_also, alternatives**) und anderen Informationen natürlich neben der Visualisierung auch für intelligente Systeme zu Benutzerunterstützung und -führung ausnutzen (siehe dazu Abbildung 6.1).

6.2 Wissensbasierte Unterstützung

Um dem in der Bildanalyse weniger erfahrenen Benutzer Unterstützung zu geben, wurden verschiedene Ansätze für häufig auftretende Fragestellungen bzw. Situationen benutzt:

- **Welcher Operator ist an dieser Verbindung anwendbar?** Hierzu ist eine genauere Spezifikation notwendig. Im einfachsten Fall könnten Operatoren angeboten werden, die die nötigen Parameter haben, d.h. eine Übereinstimmung bezüglich der Parametertypen ergeben. Es können aber auch Informationen aus der Operator-Datenbasis bezüglich Vorgänger- und Nachfolgeroperatoren verwendet werden, da durch die Verbindung bzw. die selektierten Parameter auch Vorgänger bzw. Nachfolger bekannt sind. Es wäre auch denkbar, hier dynamische Informationen zur Auswahl einzusetzen, z.B.: Das Ausgangsbild ist verrauscht, also wird ein Glättungsfilter angeboten. Ferner lassen

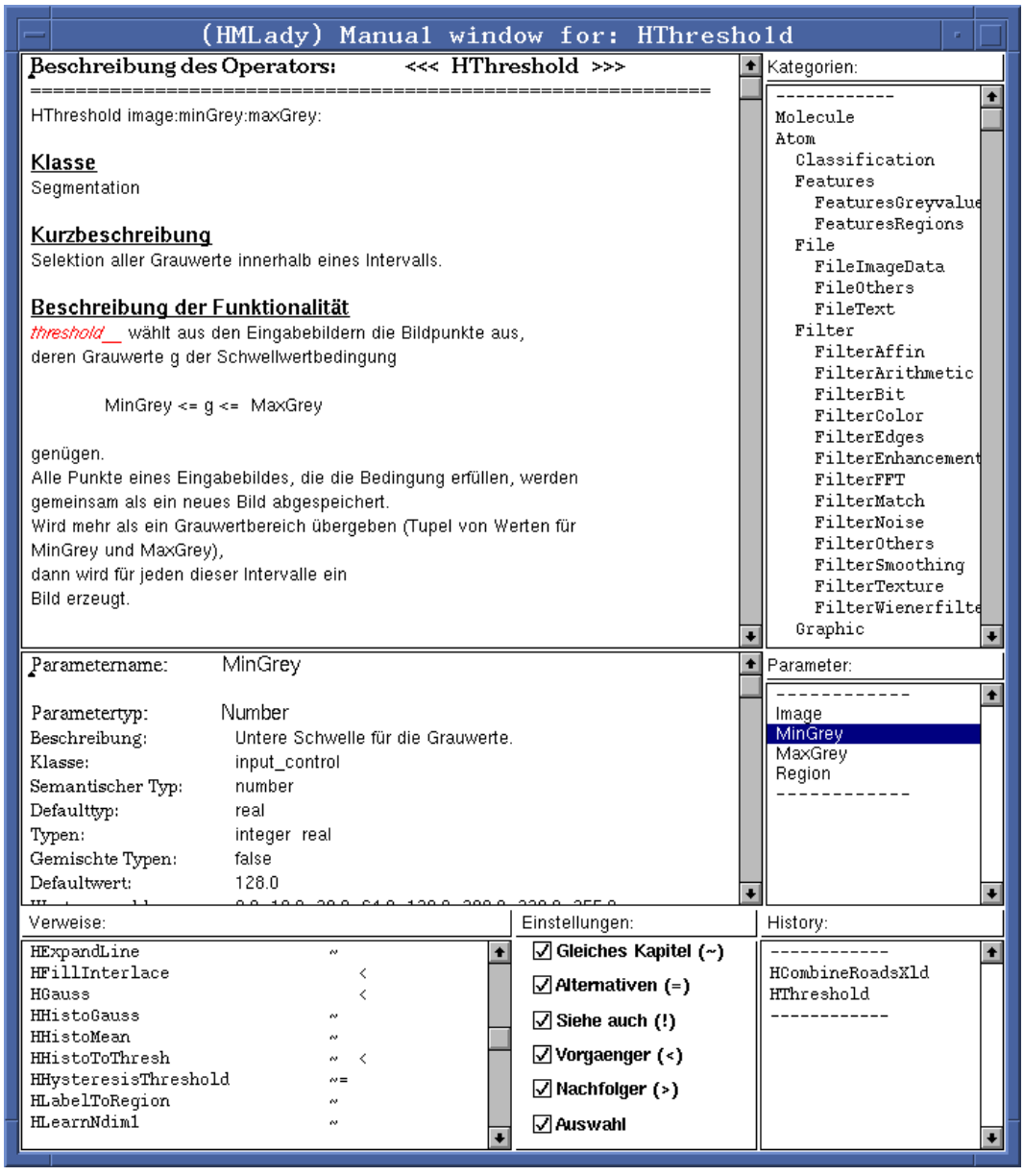


Abbildung 6.1: Hypermanual von HCANVAS

sich, bei geeigneter Erweiterung der in der Operator-Datenbasis vorhandenen Daten, Restriktionen einbauen, die z.B. die maximale Ausführungszeit oder spezifische Wirkungsweise („verkleinert“, „macht heller“) betreffen. Ebenfalls können Eigenschaften des Operators aufgenommen werden, z.B. ob er isotrop oder idempotent ist.

- **Welchen Operator kann man statt dieses Operators anwenden?** Gemeint ist die Situation, in der ein Operator, der schon im Graphen enthalten ist, nicht exakt das Gewünschte tut, oder zu langsam ist. Man kann dann z.B. einen Gaußfilter durch einen Mittelwertfilter ersetzen. Auch dies kann wieder auf verschiedene Arten geschehen. Erstens ist es möglich, die Einträge für Alternativen oder Querverweise heranzuziehen, zweitens kann über die Gruppenzugehörigkeiten auf „Geschwister“ in der Hierarchie verwiesen werden.

- **Welchen Operator kann man in dieser Phase des Bildanalyseprozesses verwenden?** Allgemeiner kann man sagen, daß hier eine bessere Gruppierung von Operatoren, z.B. nach den Phasen, oder auch bezüglich der Funktionalität vorgenommen wird. Problematisch ist allerdings, daß Benutzer aus verschiedenen Anwendungsgebieten oft nicht die selbe Nomenklatur verwenden, so daß einem Biologen die Angabe, ob ein bestimmter Operator beispielsweise *isotrop* ist, wahrscheinlich nichts sagt.

6.3 Präsentation des Wissens

Obwohl das in der Operator-Datenbasis abgelegte Wissen jeweils scheinbar lediglich den Operatoren zugeordnet ist, dient diese Information in manchen Teilen auch dazu, Wissen über Beziehungen, also Relationen zwischen Operatoren zu speichern. Beispiele hierfür sind Vorgänger- und Nachfolgerrelationen. Auch die Zugehörigkeit zu bestimmten Funktionsgruppen ist nicht ausschließlich eine Eigenschaft eines Operators, sondern kann auch strukturelles Wissen über eine bestimmte Vorgehensweise bei der Bildanalyse ausdrücken. Ein Beispiel wäre die Zugehörigkeit zu *morphologischen Operationen*, die eine bestimmte Phase im Bildanalyseprozeß repräsentieren.

Trotz der Tatsache, daß in HORUS all dieses Wissen in einer „flachen“ Wissensbasis abgelegt ist, erfordern die unterschiedlichen Interpretationen auch verschiedene Darstellungen gegenüber dem Benutzer. Um beispielsweise *ähnliche* Operatoren zu finden, reicht es aus, einen beliebigen Operator zu selektieren und dann diejenigen in der Operator-Datenbasis zu suchen, die im selben Kapitel zu finden sind oder die selbe Funktionalität haben. Natürlich ist diese Vorgehensweise auch für Relationen wie Vorgänger- oder Nachfolgerbeziehung anwendbar, aber im Sinn der schrittwei-

sen Verfeinerung kann hier auch die Frage nach dem „Dazwischen“ gestellt werden und hier bietet es sich an, nicht Operatoren zu selektieren, sondern Verbindungen.

Ein anderer, häufig auftretender Fall ist das Suchen nach einer bestimmten Funktionalität, die überhaupt nicht von schon vorhandenen Elementen ausgehen muß. Bei noch völligem Fehlen von Operationen könnte der Benutzer eine Frage wie „Was macht man zuerst?“ oder „Das Bild ist verrauscht, was kann man tun?“ stellen.

Bei derartigen Fragestellungen ist eines der Hauptprobleme die Verschiedenheit der Nomenklaturen. Selbst innerhalb der Forschungsgemeinschaft der Bildverarbeiter haben verschiedene Gruppen unterschiedliche Begriffe für ähnliche Sachverhalte (Beispiel: Rauschfilter, Glättungsfilter, Tiefpaß). Umso schwieriger ist es, einem Benutzer aus einem völlig fremden Feld, z.B. der Forstwirtschaft oder Medizin, Wissen darüber zu vermitteln, daß einer der ersten Schritte, speziell bei stark verrauschtem Bildmaterial, normalerweise die Anwendung eines Tiefpaßfilters ist, wie auch immer dieser dann möglichst allgemeinverständlich bezeichnet wird.

Kapitel 7

Realisierung der Konzepte

An einigen prototypischen Beispielen soll in diesem Kapitel aufgezeigt werden, welche Schwierigkeiten bei der Realisierung einer wissensbasierten, visuellen Oberfläche auftreten.

7.1 Propagation

Ein sehr gutes Beispiel für die in Abschnitt 5.6 angesprochene Beschreibung des Verhaltens von Objekten ist das Verfahren, das der Propagation von Berechnungen in der Oberfläche dient.

Der Benutzer erwartet, daß alle Berechnungen, die aufgrund einer seiner Aktionen ausgeführt werden können, automatisch durchgeführt werden, ohne daß dabei unnötige Schritte wiederholt werden. Die möglichen Aktionen, die Auswirkungen auf den Zustand des Programmnetzes haben können sind im folgenden aufgeführt. Verschiebungen von Operatoren betreffen lediglich die Darstellung und sind deshalb ausgelassen.

1. das Verändern eines Steuerparameters (z.B. mittels eines Schiebereglers)
2. das Knüpfen einer Verbindung
3. das Lösen einer Verbindung

Aufgrund der fehlenden Statik der Netztopologie, die unmittelbar aus Punkten 2 und 3 folgt, sind Ansätze aus der Theorie der Petrinetze nicht oder nur bedingt anwendbar [Bau90, Rei90], weil dort lediglich topologisch statische Netze betrachtet werden.

Die angestrebte Vorgehensweise entspricht der des UNIX-Make, welches beim Management großer Software-Projekte dazu verwendet wird, die Abhängigkeiten zwischen Quell- und Objektdateien derart auszunutzen, daß jeweils nur diejenige (minimale) Teilmenge von Schritten ausgeführt wird, die ein konsistentes Ergebnis zur

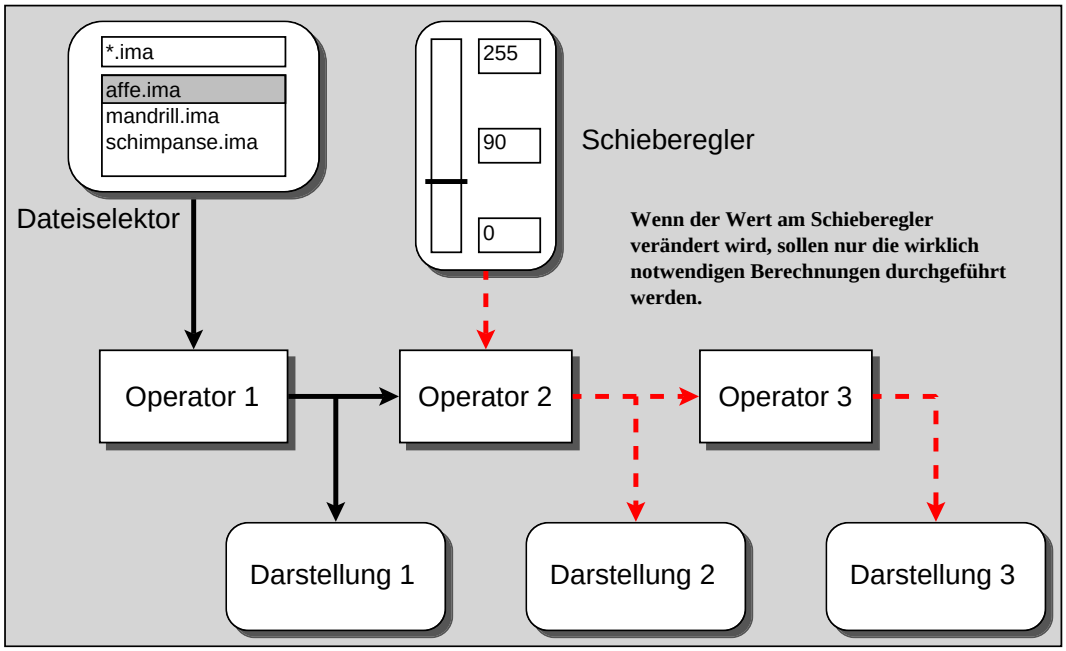


Abbildung 7.1: Reaktion auf Veränderung von Parametern

Folge hat. Im wesentlichen läuft der hierbei verwendete Algorithmus auf die Bildung einer transitiven Hülle hinaus.

Die Informationen über die Abhängigkeiten, die beim UNIX-Make im sogenannten Makefile abgelegt sind, liegen in der Oberfläche in Form von Verbindungen vor, so daß im Prinzip der selbe Algorithmus verwendet werden könnte.

Als Beispiel soll das in Abbildung 7.1 dargestellte Programm verwendet werden und zwar unter der Annahme, daß alle Werte anfangs schon berechnet waren, d.h. daß die Kanten belegt sind, und damit die Darstellungen gültig. Wenn der Benutzer jetzt z.B. den am Schieberegler eingestellten Wert verändert, sollte das Propagationsverfahren die Eigenschaft haben, daß nur die Operatoren 2 und 3 berechnet werden, nicht aber Operator 1.

Da bei dem verwendeten Ansatz die Objekte auf der Leinwand eigene „Intelligenz“ haben, liegt es nahe, auch dieses Verfahren in Form von Verhalten zu beschreiben:

1. Ein Operator kann genau dann zur Anwendung kommen, wenn alle Eingabeparameter eine gültige Belegung aufweisen.

2. Zur Anwendung von Regel 1 muß der Operator benachrichtigt werden.

3. Nach der (fehlerfreien) Anwendung des Operators auf seine Eingabeparameter werden alle Nachfolger vom Vorhandensein der berechneten Ergebnisse benachrichtigt.

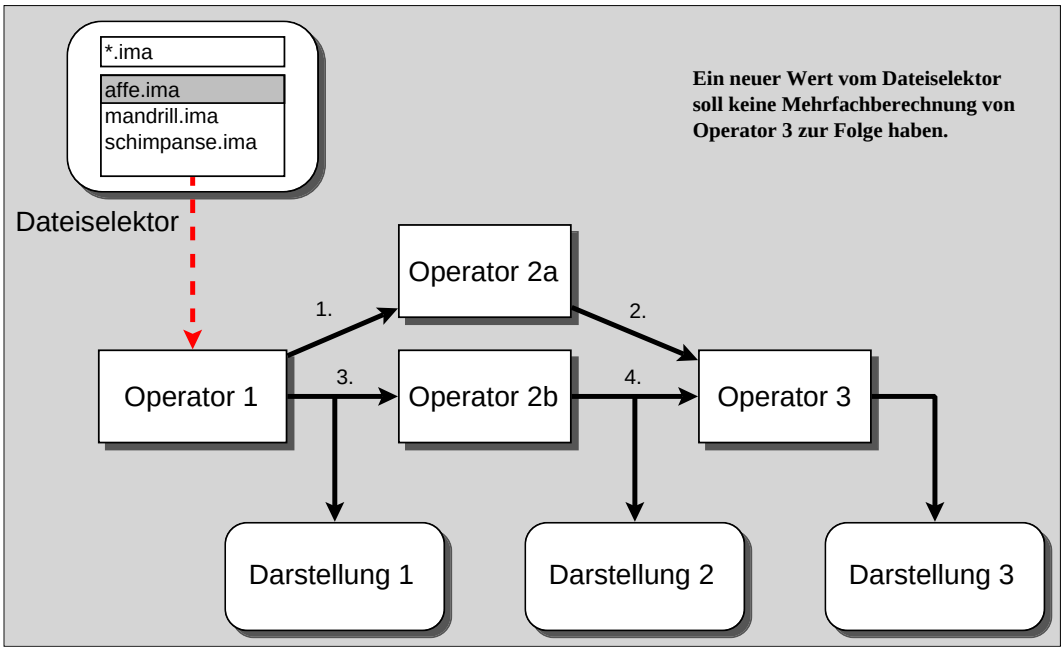


Abbildung 7.2: Minimal notwendige Berechnungen

4. Falls nicht alle Eingabeparameter belegt sind, wird der Operator nicht angewendet.

Tatsächlich könnte es bei vorhandenen *Schlingen*[1] zu Endlosschleifen in der Ausführung kommen, es sei denn, daß bei der Verknüpfung dafür gesorgt wird, daß die Werte zunächst immer invalidiert werden. Dann sorgt die Nichtausführbarkeit des Nachfolgeoperators automatisch dafür, daß die Werte innerhalb der Schlinge ungültig bleiben.

Da bei einem ungültigen Ergebnis die Berechnung im nachfolgenden Zweig nicht weiter propagiert wird, kann es dazu kommen, daß „alte“ Ergebnisse stehenbleiben. Diese beiden Möglichkeiten für ein auftretendes Fehlverhalten sind Auswirkungen der Tatsache, daß i.a. die Anwendbarkeit einer Operation lokal nicht entscheidbar ist.

Als Beispiel hierfür können in Abbildung 7.2 zwei Verhaltensweisen beim Selektieren eines neuen Dateinamens auftreten, die beide unerwünscht sind:

1. Wenn die Daten zuerst von Operator 1 an Operator 2a weitergeleitet werden und vorher Operator 2b gültige Ergebnisse hatte, wird Operator 3 zuerst mit veralteten Daten angewendet, die zu einer veralteten Visualisierung bei Darstellung 3 führen. Wenn dann neue Daten über Operator 2b an Operator 3 an-

[1] Zyklen im Graph, durch die keine gültige Ordnung mehr bestimmt werden kann.

gelegt werden, wird dieser ein zweites Mal berechnet, um dann erst korrekte Ergebnisse zu liefern.

2. Falls das obige Problem etwa durch eine Vorabinvalidierung beseitigt werden soll, führt dies zwar dazu, daß keine Mehrfachberechnung von Operator 3 mehr stattfindet, weil die Daten von Operator 2b noch ungültig sind, wenn die Daten von Operator 2a angelegt werden. Aber das Verschicken der Invalidierung führt dann dazu, daß Darstellung 2 und 3 kurz leer sind, bis durch Verfolgen des anderen Berechnungszweigs wieder alle Ergebnisse gültig werden.

Deshalb wurde die folgende Erweiterung des beschriebenen Verhaltens vorgenommen, das bei jeder der in Frage kommenden Aktionen auf der Leinwand die Konsistenz erhält:

1. Schritt: Verschicken einer *Invalidierung* unter gleichzeitiger Schlingenvermeidung, allerdings *ohne* die Darstellung zu beeinflussen. Dieser Schritt ist notwendig, um einerseits zu gewährleisten, daß alle abhängigen Operationen garantiert beeinflußt werden, andererseits sollen gültige Ergebnisse, die möglicherweise vorher vorhanden waren und im 2. Schritt mit neuen Werten berechnet werden, nicht kurzzeitig durch ungültige Ergebnisse ersetzt und dargestellt werden.

2. Schritt: Ausgehend nur von den im ersten Schritt invalidierten Datensenken werden die Ergebnisse der Vorgänger angefordert, und zwar „rückwärts“. Dies geschieht wiederum unter Detektion und Vermeidung von Schlingen.

Mit dem beschriebenen Verfahren werden nur diejenigen Berechnungen durchgeführt, deren Ergebnisse tatsächlich benötigt werden. Wenn keine Visualisierer oder andere Operatoren, die Ergebnisse benötigen, im Graphen enthalten sind, werden die Berechnungen im zweiten Schritt überhaupt nicht erst durchgeführt.

Gleiches gilt für Alternativen, weil nach Berechnung der angelegten Bedingung nur derjenige Eingangswert „geholt“ werden muß, der jeweils benötigt wird.

7.2 Datenquellen und -senken

In Abschnitt 3.9 wurde bereits darauf hingewiesen, daß es Klassen von Operatoren gibt, bei denen besondere Maßnahmen getroffen werden müssen, damit die Propagation funktioniert. Beispielsweise hätte eine Datensenke, wie eine Operation `write_image` vermutlich nur zwei Eingabeparameter, nämlich das zu schreibende Objekt und den Dateinamen. Da kein Ausgabeparameter existiert, kann das Ergebnis dieser Operation niemals abgerufen werden. Bei dem im letzten Absatz beschriebenen Propagationsverfahren würde diese Operation also niemals ausgeführt.

Abbildung 7.3: Normaler Operator, Datenquelle und -senke

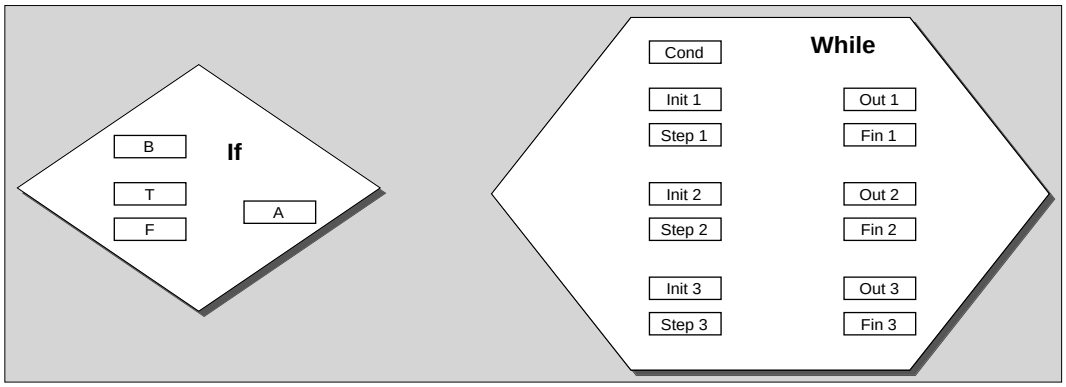


Abbildung 7.4: Konstrukte für Fallunterscheidungen und Schleifen

Auf der anderen Seite könnte es Operationen geben, wie das Holen eines Bildes von einem Framegrabber (`grab_image`), die keine Eingabeparameter haben und die nur dann, wenn Daten angefordert werden, ausgeführt werden, also Datenquellen.

Um auch mit solchen Operationen arbeiten zu können, müssen diese in der Operator-Datenbasis als Datenquelle bzw. -senke vermerkt sein. Bei der Darstellung dem Benutzer gegenüber muß eine Aktivierungsmöglichkeit, z.B. in Form eines *Knopfs*[2] vorgesehen werden (siehe Abbildung 7.3), mit der die Ausführung veranlaßt werden kann. Die Wirkung ist im Fall einer Datensenke die selbe, als wäre ein Ausgabeparameter vorhanden, an dem Daten abgeholt werden sollen. Im Fall einer Datenquelle wirkt die Aktivierung so, als ob der Benutzer einen neuen Wert an einem (nicht vorhandenen) Eingabeparameter einstellt.

7.3 Konstrukte zur Ablaufsteuerung

7.3.1 Fallunterscheidungen

Ein Konstrukt wie eine Fallunterscheidung läßt sich scheinbar leicht in das grafische Konzept integrieren. Sie besteht aus drei Eingabeparametern, nämlich einer Bedin-

[2] Aktiver Bereich, der auf Mausklicks reagiert.

gung *B* und zwei Alternativen *T* und *F*, sowie einem Ausgabewert *A*:

$$\text{if}(B,T,F,A)$$

und ist semantisch eher das Äquivalent zu einer C-Alternative:

$$A = B\ ?\ T : F$$

Bezüglich der Propagation können und sollten jedoch einige Verbesserungen gegenüber dem Standardverfahren angebracht werden:

1. Es müssen nicht wie bei normalen Operatoren alle Eingabeparameter mit gültigen Werten belegt sein, um die Anwendbarkeit zu gewährleisten. Die gültige Belegung von *B* plus der gültigen Belegung derjenigen Alternative *T oder F*, die durch den booleschen Wert von *B* selektiert wird, ist hinreichend, da die Werte des passiven Zweiges nicht in weitere Berechnungen einfließen und daher ohne Belang sind.

2. Auch muß bei Invalidierung oder Änderung der Werte von *T* bzw. *F* nur dann weiter propagiert werden, falls die jeweilige Alternative durch den Wert von *B* selektiert ist. Dies wirkt sich aus, als ob die Verbindung zum passiven Eingabewert nicht vorhanden wäre, so als ob zwei alternative Netztopologien existieren.

Aus der Einführung des Fallunterscheidungskonstruktes ergibt sich das direkte Erfordernis für *typlose* Kanten, denn verbindet man *T* oder *F* mit einem Eingabeprimitiv, während *A* noch unverbunden ist, kann nicht bestimmt werden, welcher Typ Werte für *T* und *F* gefordert sind.

Selbstverständlich ist das Konzept der Fallunterscheidung anstatt mit booleschen Werten völlig analog auch auf eine Selektion, d.h. Auswahl aus zwei oder mehr Werten, beispielsweise mit ganzen Zahlen, zu erweitern.

Real ergeben sich zwei Probleme bei dieser Art Fallunterscheidung. Zum einen zeigt Abbildung 7.5, daß eine Datensenke, wie hier der mit einer gestrichelten Linie angeschlossene Visualisierer, die unangenehme Eigenschaft hat, die Berechnung des eigentlich nicht benötigten Teils (für den Fall, daß Operator 2 *true* liefert) erzwingt. Dies folgt direkt aus dem verwendeten Propagationsverfahren. Wenn eine Änderung des Wertes von Operator 1 erfolgt und die Invalidierung Operator 5 erreicht hat, dann wird durch Rückwärtspropagation in der Fallunterscheidung zwar der inaktive Zweig nicht evaluiert, aber der Visualisierer wurde ebenfalls invalidiert und fordert neue Werte von Operator 4 an.

Wenn in einem solchen Konstrukt der Operator 4 eine Rekursion darstellt, d.h. einen Aufruf desjenigen Operators, in dem dies Konstrukt enthalten ist, dann könnte eine Endlosrekursion die Folge sein.

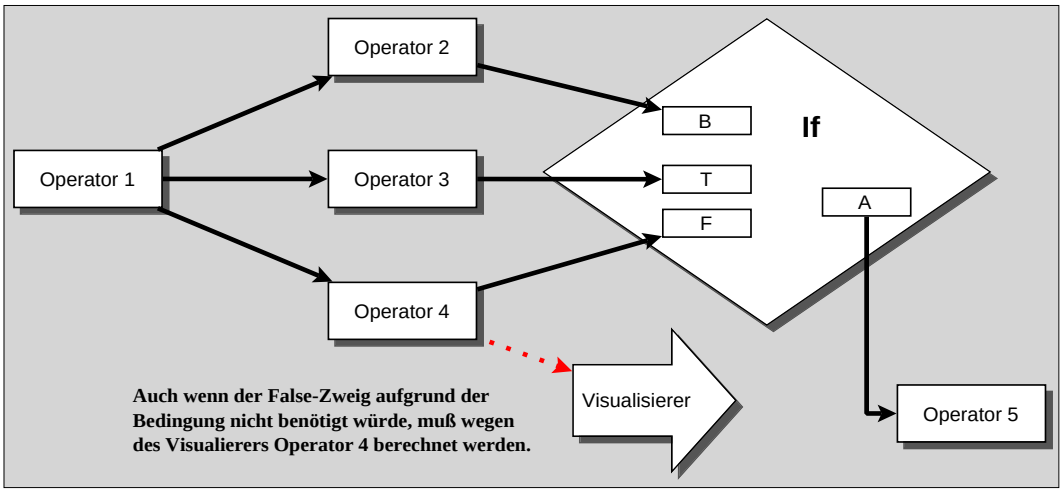


Abbildung 7.5: Problem durch erzwungene Berechnung des nicht aktiven Zweigs

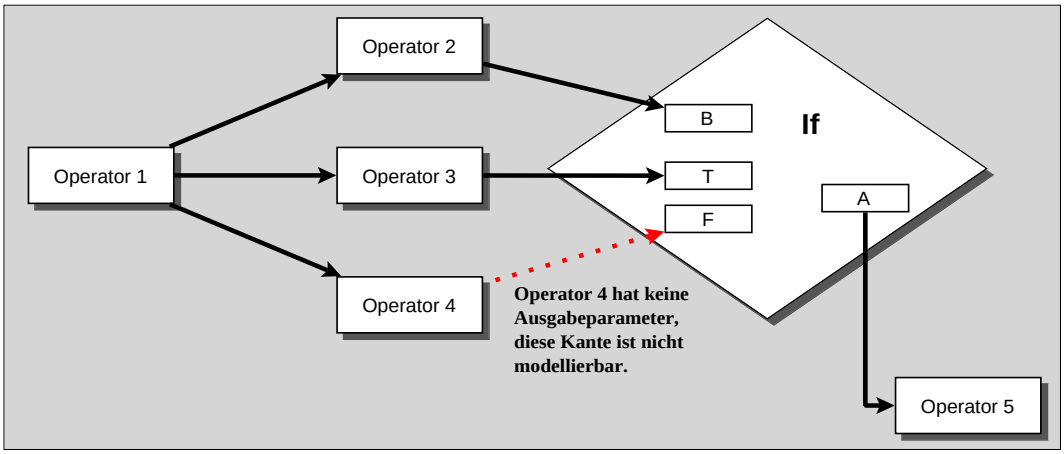


Abbildung 7.6: Fehlende Modellierbarkeit der Reihenfolge bei Auswahl

In Abbildung 7.6 ist ein weiteres Problem dargestellt, nämlich der Operator 4, der nur einen Eingabeparameter hat, aber keinen Ausgabeparameter (die gestrichelte Linie soll nur die Abhängigkeit von der Fallunterscheidung zeigen). Bei der zunächst gewählten Art der Fallunterscheidung ist diese Beziehung nicht modellierbar, weil keine Datenabhängigkeit zwischen dem „If“ und dem Operator 4 besteht.

Hier zeigt sich, daß eigentlich zwei Arten von Fallunterscheidungen existieren:

1. Eine, die eine Auswahl aus zwei Möglichkeiten, Daten zu beschaffen, darstellt (wie die zuerst gewählte).

2. Eine, die zwei Alternativen für die weitere Ausführung bietet.

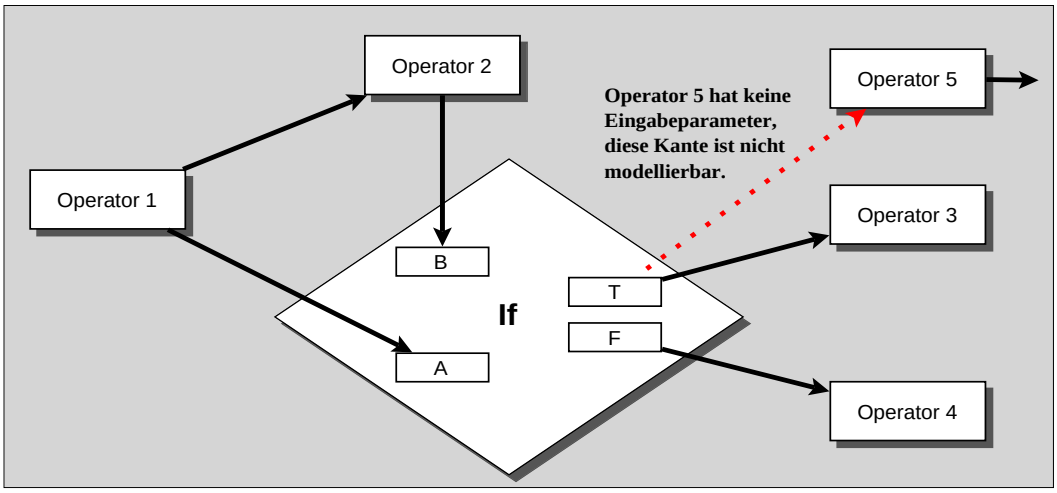


Abbildung 7.7: Fehlende Modellierbarkeit der Reihenfolge bei Alternative

Dargestellt wird die zweite Möglichkeit in Abbildung 7.7. Aber auch diese Art „If“ wirft Probleme auf. Wenn Operator 5 in der Abbildung nur eine Datenquelle ist, ist die durch Strichelung angedeutete Abhängigkeit hier nicht modellierbar.

Die korrekte Darstellung der Fallunterscheidung ist in Abbildung 7.8 dargestellt. Es findet eine komplette Kapselung statt, d.h. das Editieren dieses Konstrukts geschieht in einer separaten Unterleinwand. Hier sind beide Möglichkeiten des „If“ in einem Konzept zusammengefaßt, durch die Kapselung muß allerdings jeder verwendete Wert in Form eines Eingabeparameters ins Innere des Konstrukts transportiert werden. Die Anzahl der Ein- und Ausgabeparameter muß in der externen Repräsentation, d.h. der Außenansicht spezifiziert werden. Diese Außenansicht fungiert ansonsten wie ein normales Atom, lediglich die internen Propagationsregeln sind leicht abgewandelt, um unnötige Berechnungen der inaktiven Zweige zu vermeiden.

7.3.2 Schleifen

Ein typisches Konzept im Von-Neumannschen Rechnermodell ist die Schleife, also ein Konstrukt, das erlaubt, eine Folge von Anweisungen so oft auszuführen, bis eine bestimmte Bedingung erfüllt ist. Leider ergeben sich bei der üblichen Anwendung zwei Probleme, dazu hier ein Beispiel in C, bei dem die Werte von zehn Feldelementen aufsummiert werden sollen:

```
A = 0;
I = 1;
while (I < 10) {
    A = A + Y[I];
    I = I + 1;
}
```

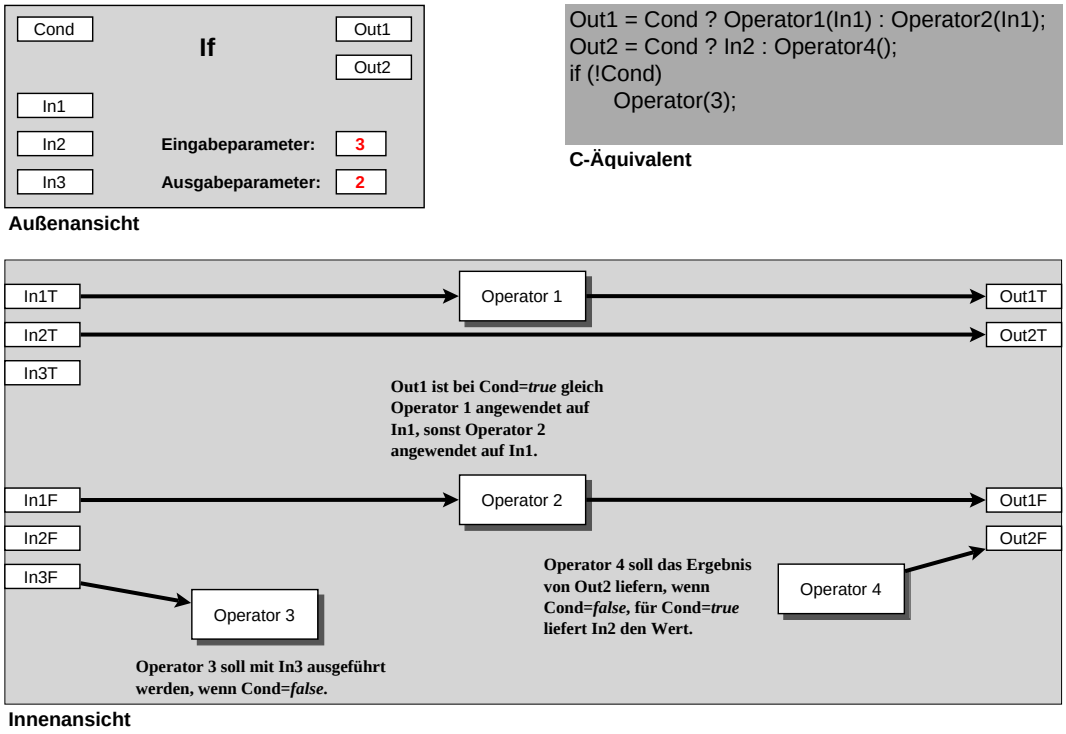


Abbildung 7.8: Richtige Darstellung der Fallunterscheidung

Zu beachten sind in diesem Fall die unter Berücksichtigung der Darstellung als Datenflußgraph folgenden Probleme:

1. Mindestens die Variable, die im Ablauf die Terminationsbedingung steuert, muß zwei Quellen haben:
   (a) Den initialen Wert (im Beispiel also 1)
   (b) Den durch den Iterationsschritt berechneten Wert (I + 1).
2. Normalerweise ist es explizites Ziel, durch Seiteneffekte der Schleifenausführung Einfluß auf die nachfolgenden Berechnungen zu nehmen (im Beispiel die Aufsummierung der Werte Y_I).

Wir führen ein Konstrukt ein, das wie folgt definiert ist:

- Es existiert genau ein Eingabeparameter, der eine Bedingung darstellt, genannt *Cond*.
- Für jeden innerhalb der Schleife verwendeten Wert *x* existieren zwei Ein- und zwei Ausgabeparameter, nämlich:

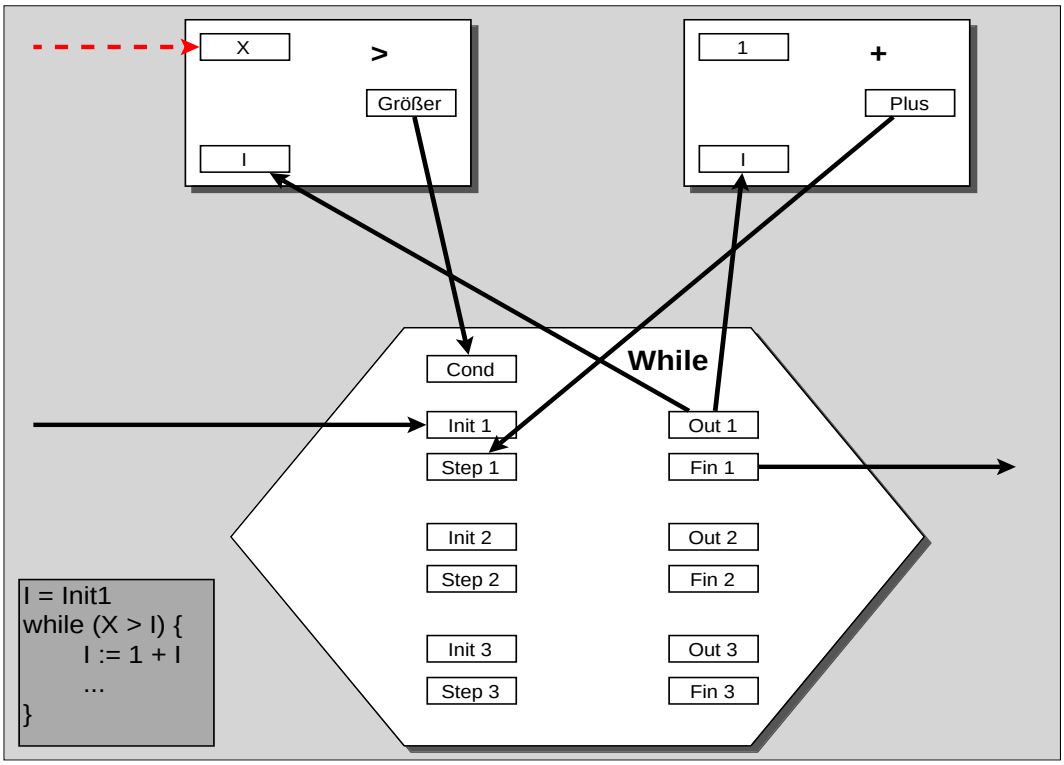


Abbildung 7.9: Einwirkung auf die Berechnung von außerhalb der Schleife

$Init_x$: Von außen vorgegebener initialer Wert für Variable x.

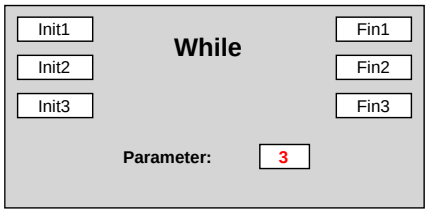
$Step_x$: Wert, der nach jedem Schritt verwendet werden soll (stammt normalerweise aus einer Berechnung innerhalb der Schleife).

Out_x: Ausgabewert, der bei *Cond == true* weitergereicht wird, normalerweise also innerhalb der Schleife verwendet werden kann.

Fin_x: Finaler Ausgabewert, der bei *Cond == false* nach außen weitergereicht werden soll, im Listing also der Wert von *A* nach der letzten Iteration.

Bei genauerer Betrachtung des beschriebenen Konstrukts ergeben sich folgende Fragestellungen:

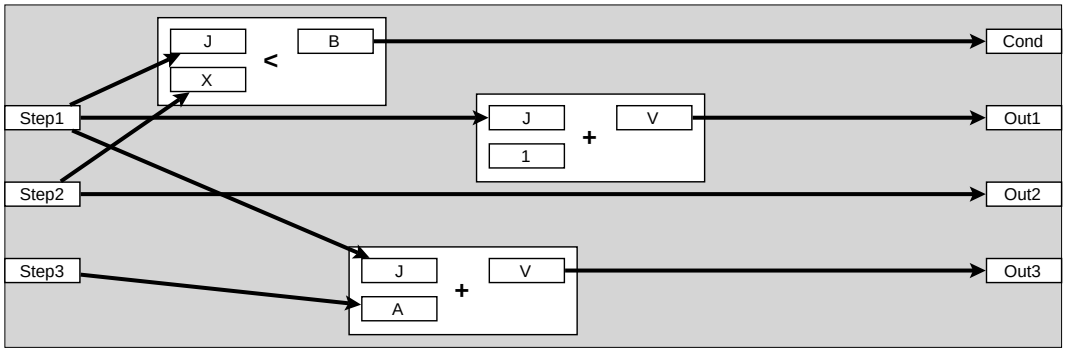
1. Was geschieht mit Ausgaben, „aus dem Schleifenrumpf“?
   Dies entspricht in Abbildung 7.9 dem Fall, daß eine Verbindung von Out_1 zu nachfolgenden Operatoren besteht. Diese sind als solche nichtexistent, sie entsprechen in `C` etwa Anweisungen, die *im Rumpf* einer Schleife stehen, bilden also das Äquivalent einer falsch gesetzten schließenden Klammer. Hier werden Berechnungen ausgeführt, die innerhalb der Schleife liegen und außerhalb nicht weiter verwendet werden.



Außenansicht

```
J = Init1;
A = Init3;
while (J < Init2) {
    A = A + J;
    J = J + 1;
}
Fin1 = J;
Fin2 = Init3;
Fin3 = A;
```

C-Äquivalent



Innenansicht

Abbildung 7.10: Gekapselte Darstellung der Schleife

2. Was geschieht bei Veränderungen „von außen“?
   In Abbildung 7.9 ist dieses Problem illustriert, es wird oben links ein Wert in die (eigentlich innerhalb der Schleife befindliche) Bedingung eingestellt. Die möglichen Behandlungsweisen werfen dann aber neue Probleme auf, denn wenn der „von innen“ kommende Wert für *I* noch undefiniert ist, dann führt das neue Anlegen eines Werts für *X* nicht zur Neuberechnung der Schleife.

Wenn also Schleifen in dieser Art verwendet werden, ist es sowohl vom Verständnis des Benutzers als auch von der formalen Beschreibung her problematisch, daß nicht vollständig klar ist, welche Operatoren innerhalb und welche außerhalb der Schleife liegen.

Um diesen Problemen abzuhelfen, könnte man, ähnlich wie schon bei den Fallunterscheidungen, eine Sicht „von innen“ und eine „von außen“ unterscheiden, so daß eine klare Abgrenzung gegeben wäre (siehe Abbildung 7.10). Auch hier ergibt sich damit wiederum die Notwendigkeit, die Erzeugung von Schleifen gekapselt zu behandeln, also in einer separaten Leinwand. Genau wie bei Fallunterscheidungen müssen dann alle innerhalb des Konstrukts verwendeten Werte mit Knöpfen nach außen geführt werden.

Leider ist trotz der exakten Beschreibungsmöglichkeit der Semantik und der Möglichkeit, mit diesem Konstrukt Schleifen auszudrücken, eben diese Art der Beschreibung recht unelegant, weil sehr viele Elemente verbunden werden müssen, um selbst

einfachste Schleifen darzustellen.

Selbstverständlich wäre das Hinzufügen von Fallunterscheidungen allein bereits hinreichend, um überhaupt Turing-Vollständigkeit zu erhalten, denn Rekursion mit Abbruchbedingungen (also Fallunterscheidungen) liefert schon die Möglichkeit der mehrfachen Ausführung. Rekursion ohne Abbruchbedingung aber ist, wie bereits erwähnt, schon inhärent im Molekülkonzept enthalten.

7.4 Konstante und Variablen

Ein großes Problem beim Design des Editors war die Behandlung von Konstanten und Variablen.

Solange nämlich lediglich innerhalb des Editors getestet wird, können alle Parameter in Form von Schiebereglern o.ä. spezifiziert werden. Will man dann das entstandene Modul als Molekül ablegen, muß es möglich sein, anzugeben, welche Daten als neue Eingabeparameter von außen angelegt werden sollen, bzw. welche Daten zu Ausgabeparametern werden. Um das zu tun, müßte der Benutzer die während des Testens verwendeten Konstanten-Elemente durch Variablen-Elemente ersetzen, was bei späteren Modifikationen des Moleküls wieder rückgängig gemacht werden müßte, denn wie sollten dedizierte Variablen-Elemente beim Test mit Werten belegt werden?

Es wäre nun möglich, einzeln stehende Variablen-Knöpfe für diesen Zweck zu schaffen, aber neue Konzepte kosten nicht nur für den Entwickler neuen Aufwand bei der Implementierung[3], sondern auch für den späteren Benutzer bei der Einarbeitung. Das hier verwendete Prinzip ist es, die Anzahl der Konzepte möglichst klein zu halten, und dafür diese entsprechend mächtig auszugestalten (siehe hierzu auch Abschnitt 3.1).

Die Idee zur Lösung des angesprochenen Problems liegt recht nahe: Bei jedem Element ist per Umschaltknopf wählbar, ob es „später“ ein externer Ein- oder Ausgabeparameter werden soll. Beim Abspeichern als Molekül wird das Element dann entweder zu einer Konstanten im Programmablauf oder zu einer Variablen, je nach Zustand des Umschaltknopfs. In Abbildung 7.11 sind Ein- und Ausgabewerkzeuge mit verschiedenen Zuständen des Knopfs mit dem Namen „Variable“ zu sehen. Die dunkleren Parameterfelder zeigen an, daß diese Parameter beim Abspeichern als Molekül zu Ein- bzw. Ausgabeparametern werden.

[3] Es existieren bis hier zwar schon Knöpfe als Objekte, aber diese sind nicht ohne ihren Operator in der Lage, auf Verschiebungen zu reagieren

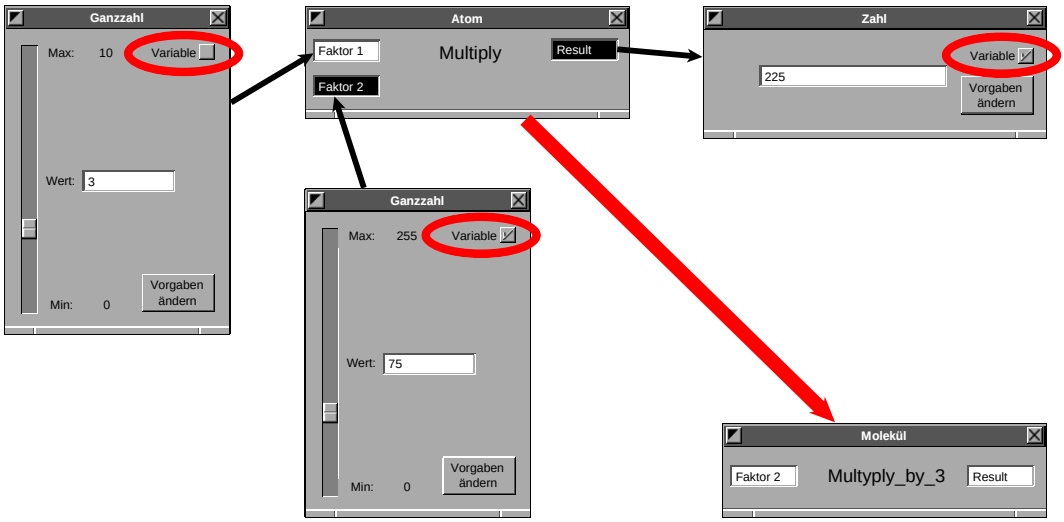


Abbildung 7.11: Konstante und Variablen, resultierendes Molekül

7.5 Parameterbelegung

Wenn in einer textuellen Programmiersprache ein numerischer Parameter für eine Prozedur angegeben werden soll, ist es zumeist dem Benutzer überlassen, welchen Wertebereich er verwenden kann. Ihm ist normalerweise klar, daß nur Werte zwischen 0 und 255 angegeben werden können, wenn der Schwellwert für eine Segmentierung auf ein Byte-Bild spezifiziert werden muß. Schon bei Bildern in Gleitpunktdarstellung kann der Benutzer (und erst recht ein Bildanalysesystem) normalerweise nicht wissen, welcher Wert für eine Schwelle sinnvoll ist.

Streng genommen müßten manche Restriktionen also sogar dynamisch (abhängig von Eingabetypen oder sogar von Parameterbelegungen) bestimmt werden. Ein Beispiel ist der Schwellwertoperator, der abhängig vom Datentyp der Eingabewerte Werte von 0 bis 255 bei Byte-Bildern, aber auch negative und sehr viel größere Werte bei Float-Bildern verarbeiten kann. Ein anderes Beispiel sind Datenabhängigkeiten, wie beim Operator `edges`, dessen Parameter *Filter* eine Zeichenkette ist, die die Art des verwendeten Kantenfilters bestimmt. Abhängig von diesem Parameter ist der Wirkungszusammenhang völlig unterschiedlich. Eigentlich handelt es sich bei jedem durch den *Filter*-Parameter ausgewählten Filter um eine eigene Operation. Solche und ähnliche (siehe Absatz 3.9) Entwurfsprobleme, die oft aus Zeiten stammen, in denen objektorientierte Ansätze nicht berücksichtigt wurden, stellen große Hindernisse für eine vernünftige Modellierbarkeit von Systemen wie dem angestrebten dar. Im implementierten System wurde zugunsten einer weitestgehenden Auflösung der angesprochenen Probleme auf der Bibliotheksseite (wie in diesem Fall durch Auftrennung in mehrere einzelne Operatoren) auf eine unelegante Modellierung verzichtet.

Abbildung 7.12: Eingabeprimitiv für ganze Zahlen und Restriktionsfenster

Bei der Umsetzung in einer grafischen Programmiersprache ist es aber schon zur Darstellung des Eingabeprimitivs notwendig, bestimmte Daten abfragen zu können. Um für numerische Parameter einen Schieberegler zu visualisieren, werden mindestens die Minimal- bzw. Maximalwerte und möglichst auch der Wirkungszusammenhang[4] benötigt.

Erst entsprechend dieser Informationen läßt sich der Schieberegler dem Benutzer dann präsentieren (siehe Abbildung 7.12, links). Gleichzeitig wird dieser darüber informiert, welchen Restriktionen der betreffende Parameter unterliegt. Für den Benutzer ist solch ein Systemverhalten fast selbstverständlich, aber um diesen Grad der Unterstützung anbieten zu können, ist auf verschiedenen Ebenen des Bildanalysesystems zum Teil *erheblicher* Aufwand nötig: Für jeden Parameter eines jeden Operators müssen diese Informationen in einer Operator-Datenbasis vorhanden sein.

Das gewählte Beispiel eines numerischen Parameters steht dabei nur exemplarisch da. Auch bei scheinbar simplen Datentypen wie Zeichenketten ergeben sich Folgerungen. Natürlich können die Daten vom Benutzer immer mit einem simplen Textfeld angefordert werden, aber es ist in jedem Fall besser, wenn die Datentypen semantisch tiefer aufgegliedert sind.

In HORUS gibt es nicht nur Zeichenketten (Abbildung 7.13), die per Textfeld ange-

[4]Hiermit ist die Information gemeint, ob der Parameter eher linear, logarithmisch oder exponentiell auf die Wirkungsweise des Operators Einfluß nimmt.

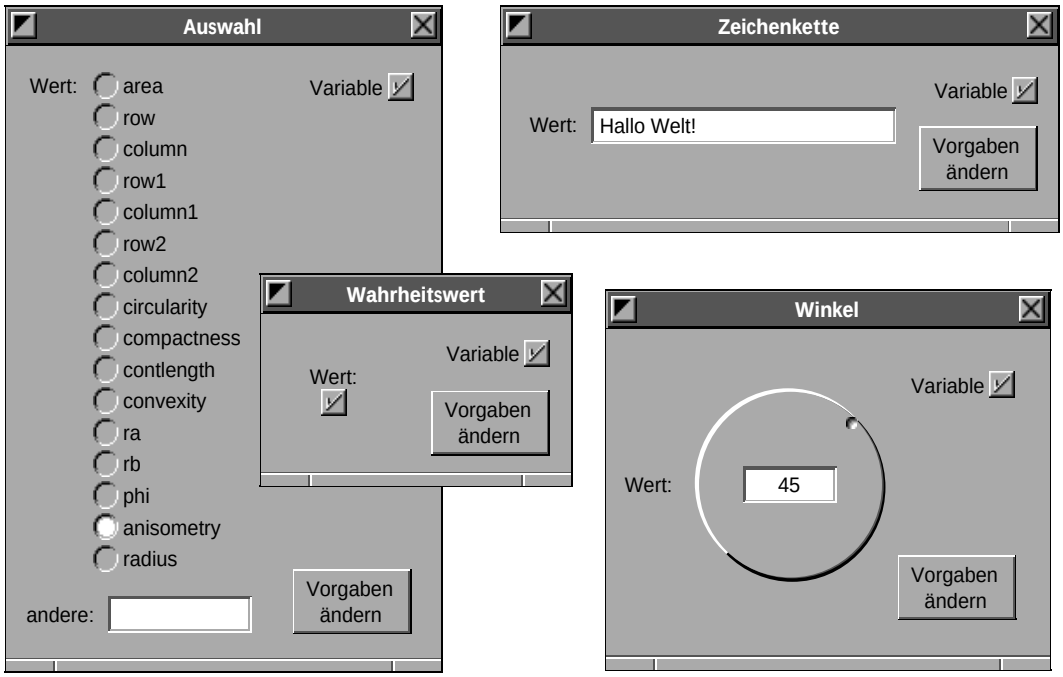


Abbildung 7.13: Auswahllisten, Wahrheitswerte, Zeichenketten und Winkel

fordert werden, sondern auch Auswahllisten[5], die per Menü abgefragt werden, sowie Dateinamen (diese werden in einem Dateiselektor angefragt, siehe Abbildung 7.14). Normalerweise sind folgende Arten von Parametern relevant:

Numerischer Wert: mit den Untertypen

Feste Liste: eine Aufzählung von möglichen Werten

Ganze Zahl: mit Minimum, Maximum und minimaler Schrittweite sowie Wirkungszusammenhang (linear, logarithmisch, quadratisch)

Gleitpunktzahl: mit Minimum, Maximum und sinnvoller Schrittweite sowie Wirkungszusammenhang (linear, logarithmisch, quadratisch)

Zeichenkette: mit den Untertypen

Dateiname: mit Angabe der möglichen Dateiarten (z.B. `.tiff, .fil, .reg, .ps`)

Feste Liste: eine Aufzählung von möglichen Werten

Beliebige Zeichenkette: allgemeine Form

[5]Also fest begrenzte Auswahlmöglichkeiten, z.B. Merkmale für `select_shape`: ”area”, ”row”, ”column”…

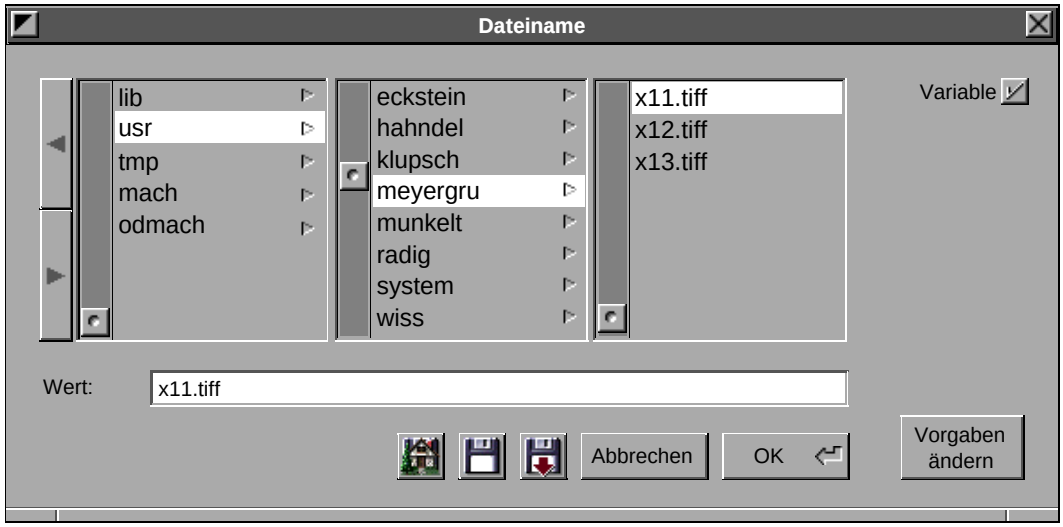


Abbildung 7.14: Eingabeprimitiv für Dateinamen

Wahrheitswerte: *true* oder *false*

Bilder: ikonische Daten, z.B. Grauwert- oder Farbbilder

Regionen: Bilddaten, die nur noch Ausdehnungen besitzen, ohne Intensitätsinformation

Extended Line Descriptions: linienhafte Datenstrukturen

XY-Koordinaten: mit Angabe des Wertebereichs

Ausdehnungen: mit Angabe der Breite und Höhe

Auch für Koordinaten und Ausdehnungen bietet es sich an, spezielle Eingabewerkzeuge zu definieren, um dem Benutzer die Semantik näherzubringen (siehe Abbildung 7.15). Soweit möglich, wurden die Parametertypen in den HORUS-Sprachschnittstellen als semantische Datentypen abgebildet. So hat in der `Smalltalk`-Schnittstelle von HORUS der Operator `mean` nur zwei Eingabeparameter, nämlich das Eingabebild und die Ausdehnung des anzuwendenden Mittelwertfilters, während in der `C`-Version drei Parameter nötig sind: Eingabebild, Breite und Höhe. Selbst wenn in `C` Breite und Höhe zu einer Struktur zusammengefaßt würden, wäre deren Spezifikation im `C`-Programm direkt innerhalb des Aufrufs nicht möglich, in `Smalltalk` dagegen existiert für Ausdehnungen eine eigene Notation.

Bei den höher strukturierten Datentypen, beispielsweise bei Bildern und Regionen wurde zunächst keine Eingabemöglichkeit vorgesehen, weil diese in der Bildanalyse normalerweise nicht als Steuer- sondern als Objektparameter verwendet werden.

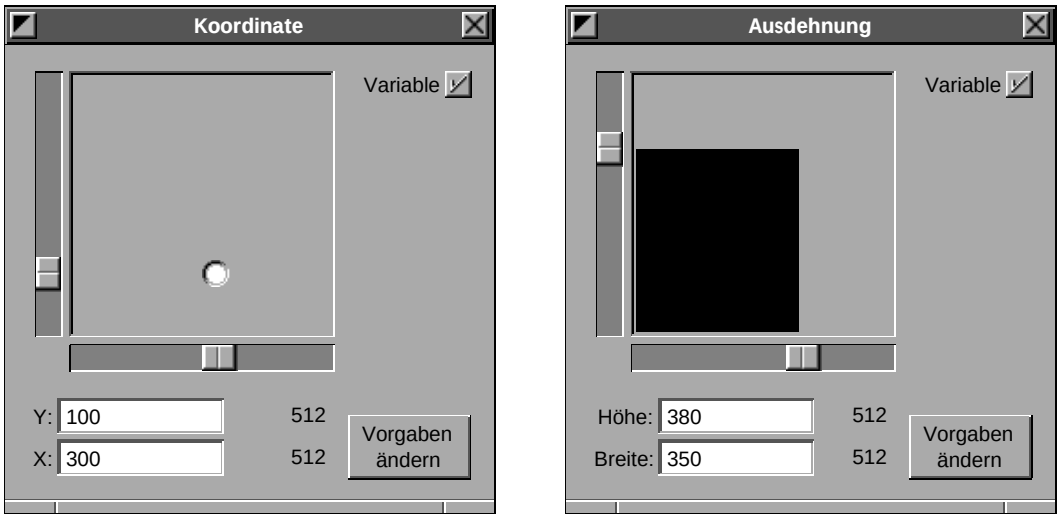


Abbildung 7.15: Eingabeprimitive für Koordinaten und Ausdehnungen

Bei der Generierung des Eingabewerkzeugs werden entsprechend des Parametertyps die verschiedenen Angaben aus der Operator-Datenbasis geholt und in Instanzvariablen abgelegt. Der Benutzer kann aber jederzeit dieser Werte nachträglich seinen Bedürfnissen anpassen. In speziellen Anwendungen kann es sinnvoll sein, Wertebereiche einzuschränken oder selbst den voreingestellten Datentyp eines Parameters zu verändern (wenn beispielsweise der voreingestellte Datentyp eine Ganzzahl ist, aber auch Gleitpunktzahlen verwendet werden dürfen.

Bei fehlenden Angaben (entweder in der Datenbasis nicht enthalten oder nicht erzeugbar) werden Standardvorgaben verwendet, denn es gibt bei der Anfrage für Parameterwerte zur Darstellung eines Wertes zwei Hauptprobleme:

1. Der Parameter indirekt innerhalb eines Ausdrucks verwendet werden. Das kann bei Skalierungen geschehen, z.B. im Ausdruck: $3X + 6$. Wenn bekannt ist, daß der resultierende Parameter im Bereich von 0 bis 255 liegen kann, muß der Schieberegler entsprechend durch Rückwärts-Intervallrechnung den Bereich von -2 bis 83 abdecken. Sofort klar ist in diesem Fall, daß das Intervall auch durchaus nicht in jedem Fall angegeben kann, z.B. für den Ausdruck $0X$. Ferner kann das Quellintervall auch zerfallen, z.B. für $(x-5)^2$. Ist die Operation hinreichend komplex, kann ebenfalls kein sinnvolles Intervall angegeben werden. Es ist also ohnehin nicht sinnvoll, durch HORUS-Operationen selbst rückwärts zu propagieren, wenn beispielsweise die Zahl der resultierenden Regionen eines Teilkomplexes wiederum als Parameter für einen weiteren Schritt des Verfahrens dienen soll.

2. Falls zwischen Ein- und Ausgabeparametern nicht streng unterschieden wird, kann es vorkommen, daß ein Eingabeparameter für mehr als eine Operation

Verwendung findet. Dann muß festgelegt werden, wie verfahren werden soll. Eine Möglichkeit besteht darin, den Durchschnitt der Intervalle zu verwenden.

3. Beim Konzept von benannten, während des gesamten Prozesses gültigen Variablen kann der Fall auftreten, daß ein Wert sowohl Ausgabeparameter einer Operation als auch durch einen Schieberegler spezifizierter Eingabeparameter einer nachfolgenden Operation ist[6]. In diesem Fall ist zunächst gar nicht klar, welcher Wert für die nachfolgende Operation verwendet wird. In einem dynamischen System wird sich zwangsläufig durch die Abarbeitung eine möglicherweise (bedingt durch asynchrone Benutzerinteraktion) zufällige Ablaufreihenfolge ergeben, aber ein *vorhersehbares* Verhalten ist selbstverständlich wichtig und muß daher gewährleistet werden.

Da es, wie dargestellt, vorkommen kann, daß ein sinnvolles Intervall nicht angegeben werden kann, wird für den Benutzer die Möglichkeit eröffnet, selbst die für die zur Visualisierung nötigen Parameter zu spezifizieren, obwohl wenn möglich, die Vorgaben durch das System in der beschriebenen Art und Weise errechnet werden.

7.6 Ablegen von Molekülen

Um eine durch alle Sprachschnittstellen gleichartige Behandlung von Operatoren zu gewährleisten, müßte eine Sprache für die Darstellung von Molekülen definiert werden, um den HORUS-Kern in die Lage zu versetzen, unabhängig von der Wirtssprache die Moleküle ausführen zu können. Hierzu müssen entsprechende Erweiterungen an der Operator-Datenbasis sowie im Kern vorgenommen werden. Bestandteile dieser Beschreibung umfassen:

1. wie bei Atomen die Informationen über Art, Anzahl und Eigenschaften von Parametern und

2. anders als bei Atomen nicht die C-Prozedur, die in der Bibliothek real hinter der gewünschten Funktion steht, sondern eine textuelle Aufschreibung der auszuführenden Funktionen, die im einfachsten Fall vom HORUS-Kern direkt ausgeführt werden kann. Dadurch sind diese Funktionen vom Benutzer praktisch nicht von Atomen zu unterscheiden.

Sinnvoll ist dies sicherlich nur dann, wenn alle Werkzeuge, die zum Erstellen von Programmen verwendet werden (z.B. HDEVELOP) dieselbe Spezifikationssprache verwenden. In einem System, das von Haus aus Persistenz bietet, können die Moleküle dagegen direkt im Image abgelegt werden.

[6] Ebenso können dann zwei Ausgabekanten von HORUS-Operationen zusammenfließen

7.7 Fokussierung

Um dem Benutzer ein einfaches Umgehen mit den auf der Leinwand dargestellten Objekten zu ermöglichen, ist ein sparsamer Umgang mit den Ressourcen notwendig. Speziell trifft dies auf den zur Verfügung stehenden Platz auf dem Bildschirm zu, denn die resultierenden Programme werden schon bei mittlerer Komplexität sehr ausladend.

Um die Deutsch-Grenze von ca. fünfzig gleichzeitig auf dem Schirm sichtbaren Objekten nicht zu überschreiten, wurden Maßnahmen getroffen, um auch bei größeren Applikationen den Überblick zu wahren. Die Technik, die bei derartigen Problemen meist Anwendung findet, ist die *Fokussierung* [Wir94]. Bestimmte Teile des Ablaufplanes werden dabei entfernt (= *implodiert*) und durch einen Platzhalter ersetzt.

Durch das in dieser Arbeit verfolgte Konzept von Atomen und Molekülen ist es sehr einfach, diese Technik zu integrieren, indem es um eine temporäre Variante erweitert wird. Es gibt jeweils nicht nur das eine bearbeitete Molekül, sondern neben dessen atomaren Bestandteilen können beliebig geschachtelte Sub-Moleküle vorhanden sein. Diese haben immer einen von zwei Zuständen: *implodiert*, d.h. alle beteiligten Operatoren werden durch einen einzigen Vertreter dargestellt, so als ob es sich dabei um einen Operator handelt, der die entsprechende Funktionalität aufweist, oder *explodiert*, d.h. vollständig in Einzelschritten dargestellt.

7.8 Topologisches Sortieren

Das Molekül, das zur Laufzeit des Editors verwendet wird, darf, wie beschrieben, Schlingen enthalten. Bei bestimmten Operationen ist es jedoch notwendig, eine bestimmte logische Abfolge zu erzwingen, beispielsweise, wenn eine automatische Anordnung auf der Leinwand veranlaßt werden soll, z.B. um ein nach längerem Arbeiten verworrenen Graphen entflechten zu lassen. Die normale westliche Lesart erfordert, die Objekte von links nach rechts etwa in der Ablaufreihenfolge zu ordnen. Das selbe Problem stellt sich, wenn ein Molekül als ablauffähiges Programm in einer imperativen Programmiersprache wie `C` abgespeichert werden soll. Dort lassen sich Schlingen nicht notieren, weil das Laufzeitsystem fehlt.

Für solche Zwecke ist es nötig, die Knoten topologisch zu sortieren, was natürlich nur möglich ist, wenn keine Schlingen vorliegen.

Der entwickelte Algorithmus realisiert gleichzeitig beide Ziele, nämlich das Finden von Schlingen und die topologische Sortierung. Besonders interessant ist in diesem Zusammenhang, daß beide in der Literatur gefundenen Algorithmen ungeeignet waren, weil sie entweder zu speziell ([Knu73a, Knu81, Knu73b]) oder zu allgemein waren ([AHU85]).

Um das selbe Verfahren beispielsweise auch beim automatischen Layout von Operatoren (siehe Abschnitt 8.8) anwenden zu können, muß es *stabil* gemacht werden. Mit diesem Begriff werden Sortierverfahren bezeichnet, die die Eigenschaft haben, Elemente mit gleichen Schlüsseln in ihrer ursprünglichen Reihenfolge zu belassen [Wir83]. Diese Eigenschaft ist notwendig, wenn z.B. durch mehrfache Sortierung mit unterschiedlichen Schlüsseln sortiert werden soll, z.B. eine Adressenliste zuerst nach Vornamen und danach mit Nachnamen.

Bei topologischen Sortierverfahren ist es besonders schwierig, sie stabil zu machen, denn normalerweise sind topologische Sortierungen nicht eindeutig, dies entspricht dem Fall von gleichen Schlüsseln bei normalen Sortierungen. Gerade beim automatischen Layout würde sich das Fehlen der Stabilität stark bemerkbar machen, denn mehrfaches Sortieren hätte jedesmal eine völlig andere Anordnung auf der Leinwand zur Folge. Das von Ullman in [AHU85] vorgeschlagene Verfahren hatte unmodifiziert genau dieses Verhalten.

7.9 Probleme bei Interaktion

Wie sich herausstellte, war VisualWorks nicht ganz so perfekt bei der Behandlung von Interaktion, wie zuerst angenommen. Dies hing damit zusammen, daß bei der Entwicklung der Bibliotheken lediglich der statische Aspekt eine Rolle gespielt hatte. Augenscheinlich wurde niemals damit gerechnet, daß sich zwei Sichten überlappen könnten (was bei statischer Anordnung ja auch praktisch nicht vorkommt). Die Art und Weise, wie die zu den Sichten gehörenden Kontrollinstanzen agierten, hatte zur Folge, daß es zu Fehlern kam. Das Auffinden und die Korrektur der relevanten Methoden war entsprechend kompliziert und zeitaufwendig.

7.10 Detektion von Mausklicks

Selbst im Zeitalter von grafischen Oberflächen scheinbar so selbstverständliche Dinge wie das Detektieren eines Mausklicks können recht kompliziert werden, wenn gewohnte Pfade verlassen werden. In „sogenannten“ grafischen Oberflächen handelt es sich bei den zu selektierenden Objekten meist um sehr simple Primitive, wie rechteckige (und orthogonal ausgerichtete) Knöpfe [VM94].

Schwieriger ist die Sachlage, wenn es sich wie im Fall von HCANVAS um eine Verbindung zwischen zwei Operatoren handelt, die aus Geradenstücken beliebiger Ausrichtung besteht. Zum einen muß ein „Fangbereich“ definiert werden (in Abbildung 7.16 der Bereich zwischen A, B, C und D), der den subjektiven Erwartungen des Benutzers bezüglich Toleranz genügt, zum anderen müssen geeignete Verfahren implementiert werden, um die Frage zu klären, ob ein Punkt innerhalb dieses

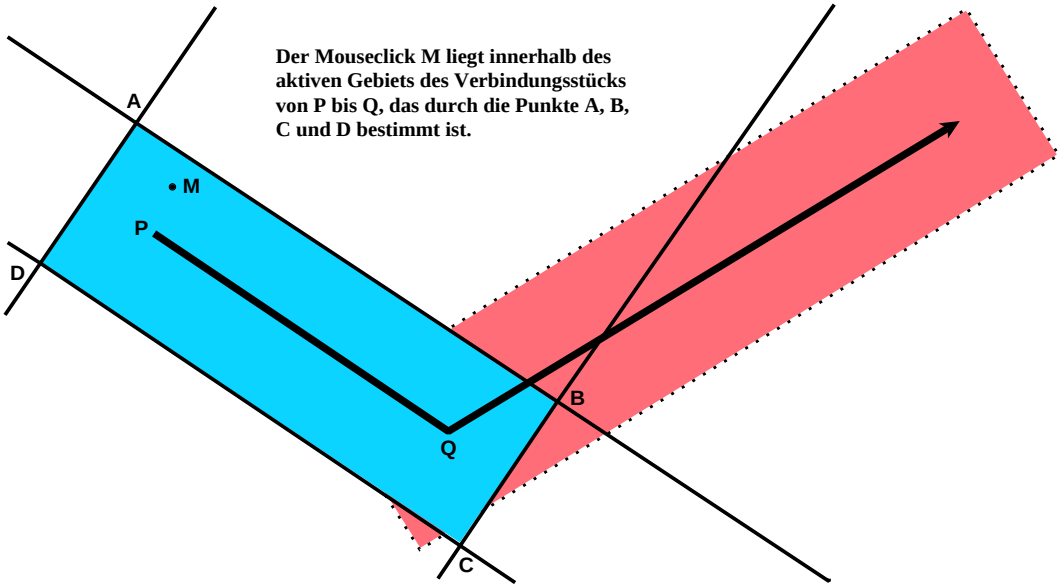


Abbildung 7.16: Aktives Gebiet eines Verbindungsstücks

Fangbereichs liegt.

Dabei ist die Ermittlung der Punkte A, B, C und D aus P und Q natürlich trivial, die zunächst naheliegende Methode, nämlich die vier Eckpunkte A, B, C und D durch Geraden zu verbinden und durch Fallunterscheidungen festzustellen, ob der zu testende Punkt M innerhalb liegt, stellt sich bei näherer Betrachtung jedoch als recht kompliziert heraus, weil zu den vier relevanten Geraden noch zusätzliche Fallunterscheidungen für exakt vertikale (oder horizontale) Linien entstehen.

Der hier verwendete Ansatz nimmt ein bekanntes Verfahren zur Hilfe, das feststellt, ob drei Punkte X, Y und Z einen Linksumlauf bilden, d.h. wenn eine Gerade durch X und Y gezogen wird, ob Z links von dieser liegt, und zwar von X aus gesehen. Bilden dann jeweils (A, M, B), (B, M, C), (C, M, D) und (D, M, A) einen Linksumlauf, befindet sich M innerhalb des von A, B, C und D gebildeten (beliebig orientierten) Rechtecks.

Kapitel 8

Implementierung

8.1 Entwicklungsgeschichte

Um die in den vorangegangenen Kapiteln dargestellten Konzepte zu validieren, wurde zunächst ein Prototyp der Oberfläche in `Smalltalk` erstellt. Es war absehbar, daß der Umfang der Lösung für die gestellte Aufgabe so groß sein würde, daß es nicht praktikabel gewesen wäre, ohne diesen Schritt auszukommen.

Mit dem erstellten, ersten Prototyp konnten bereits Programme erzeugt werden, allerdings waren diese innerhalb der Oberfläche noch nicht ablauffähig, da Möglichkeiten zur Parameterbelegung fehlten. Der Prototyp beherrschte bereits das Abspeichern von erzeugten Molekülen in einer für `Smalltalk` weiterverwendbaren Form, allerdings ohne Informationen, wie sie in der Operator-Datenbasis vorliegen, abgesehen von Parameternamen (z.B. „Dateiname“) und -typen (z.B. `HInString`), wie in Abbildung 8.1 zu sehen. Auch eine Auswahl von Operatoren nach Kapiteln sowie nach passenden Parametertypen („Typvorschlag“) war schon vorhanden (in der Abbildung 8.1 die *Sichten*-Spalten). Die gewünschte Auswahl in der Spalte *Operatoren* konnte durch Selektieren mehrerer Gruppen in einer der beiden Spalten und Bildung des Schnitts mit den selektierten Gruppen der zweiten Spalte gebildet werden, beispielsweise im Bild die Operatoren, die von den Parametertypen her geeignet sind (im Bild: *Sichten 1: Typvorschlag*), davon aber ausschließlich diejenigen, die HORUS-Atome (*Sichten 2: HAtom*) sind und keine etwa passenden Moleküle.

Einige Konzepte sind in ähnlicher Form auch in nicht vollständig graphischen Repräsentationen anwendbar, z.B. die grafische Parameterversorgung von ansonsten textuell dargestellten Konstrukten in HDEVELOP. Diese Integration wurde im Rahmen einer Diplomarbeit [Sch95b] vorgenommen.

Anschließend wurde eine Diplomarbeit [Was95] vergeben, die eine möglichst vollständige Implementierung, ebenfalls in `Smalltalk` und auf Basis des ersten Pro-

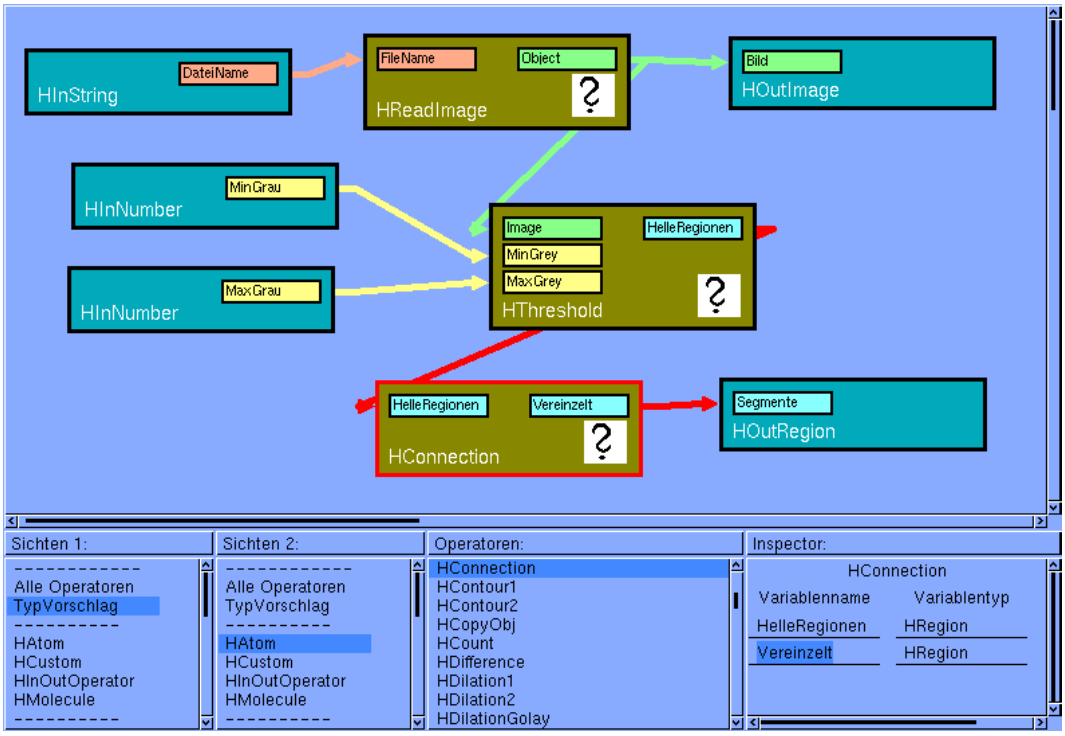


Abbildung 8.1: Erster `Smalltalk`-Prototyp der Oberfläche

totyps, zum Ziel hatte. Die Resultate nachfolgend zusammengefaßt.

Eine Art *Hypermanual*[1] war im Vorgängersystem, HDEVELOP, durch die Integration von Vorgänger- und Nachfolgeroperatoren in die HORUS-Operator-Datenbasis schon seit einiger Zeit vorhanden, die wissensbasierte Unterstützung (siehe Kapitel 6) bei der Auswahl von geeigneten Operatoren fehlte jedoch. Hierzu wurde neben der Erweiterung der `Smalltalk`-Oberfläche um diese Funktionalität auch eine Implementierung im Rahmen einer weiteren Diplomarbeit [Gün96] auf einer anderen objektorientierten Plattform, nämlich NEXTSTEP[2], begonnen. Hier sind andere grafische Paradigmen als in `Smalltalk` gegeben, die andere Interaktionsmöglichkeiten, beispielsweise Paletten nahelegen [LMNR90].

8.2 Grafischer Editor HCANVAS

Die aktuelle Implementierung in `Smalltalk` umfaßt ca. 80 Klassen und 1000 Methoden, insgesamt etwa 13000 Zeilen [Was95]. Einige Ausschnitte der Klassenhier-

[1] Elektronisches Handbuch, bei dem über Verweise zu anderen Themen verzweigt werden kann. Hierdurch wird die normalerweise vorhandene, eindimensionale Struktur von Texten erweitert.

[2] Objektorientiertes, grafisches UNIX-Betriebssystem von NeXT Computer, Inc., das auf Display Postscript aufbaut

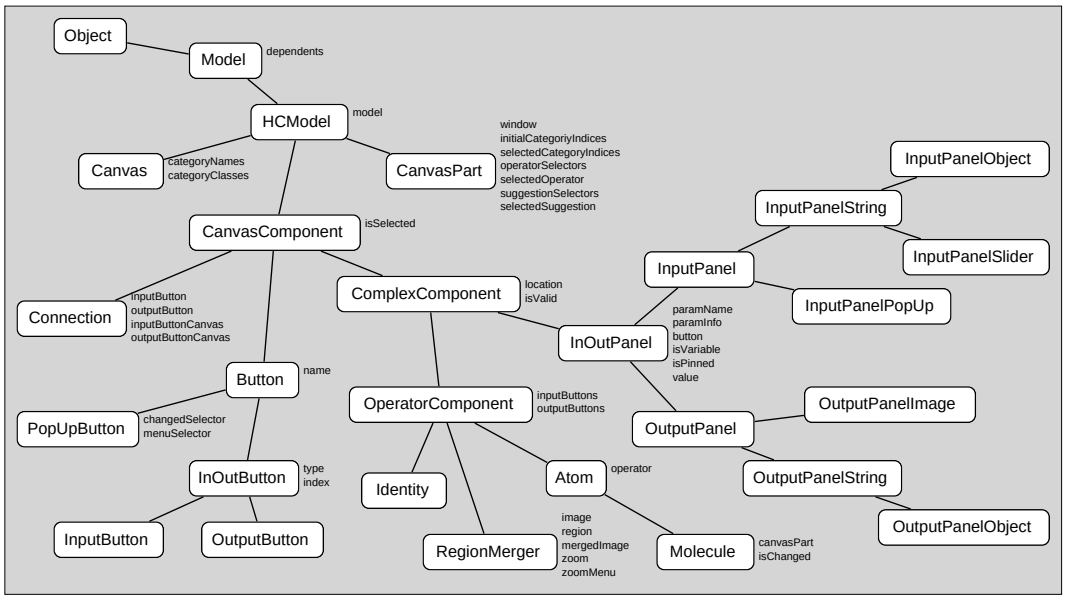


Abbildung 8.2: Statische Klassenhierarchie der Modelle (aus: [Was95])

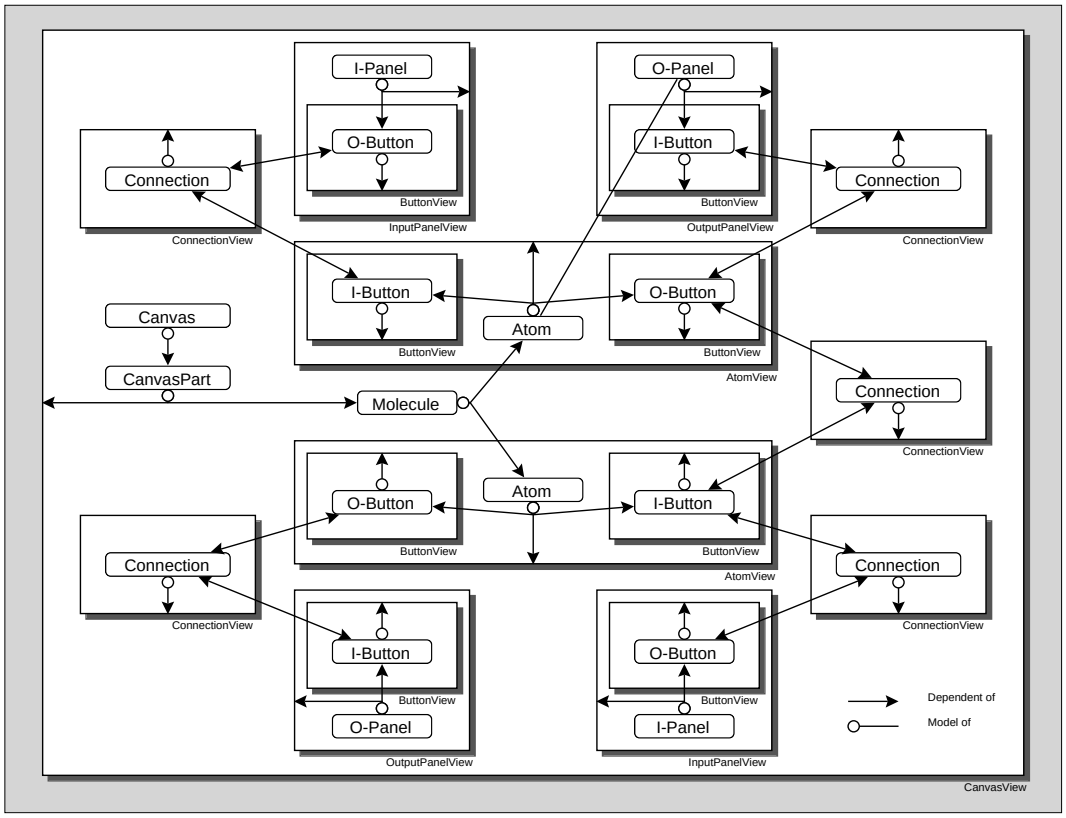


Abbildung 8.3: Dynamische Hierarchie von Molekülen (aus: [Was95])

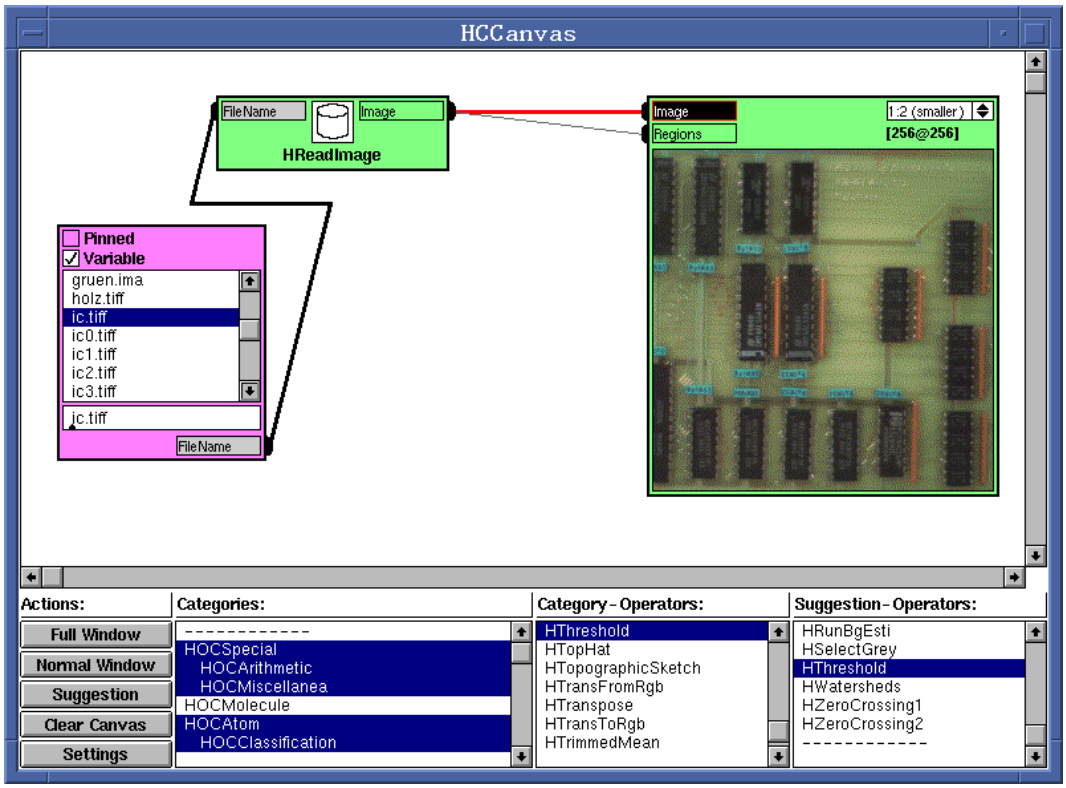


Abbildung 8.4: Leinwanddarstellung von HCANVAS

archien sind in Abbildung 8.2 zu sehen, wobei in `Smalltalk` die statischen Klassenhierarchien weniger komplex als die dynamischen Hierarchien beispielsweise der Sichten sind (siehe Abbildung 8.3).

8.3 Interaktionsbereich von HCANVAS

Das Interaktionsfenster von HCANVAS ist in fünf Bereiche gegliedert. In Abbildung 8.4 ist im oberen Bereich die eigentliche Arbeitsfläche, die Leinwand zu sehen, die innerhalb einer Rollsicht liegt. Darunter befinden sich vier Spalten:

Aktionsknöpfe: Mit diesen können oft benötigte Aktionen ausgelöst werden:

***Full Window*:** Vergrößern des Fensters auf volle Bildschirmgröße

***Normal Window*:** Normalgröße des Fensters

***Suggestion*:** Starten der Vorschlagskomponente

***Clear Canvas*:** Löschen der gesamten Leinwand

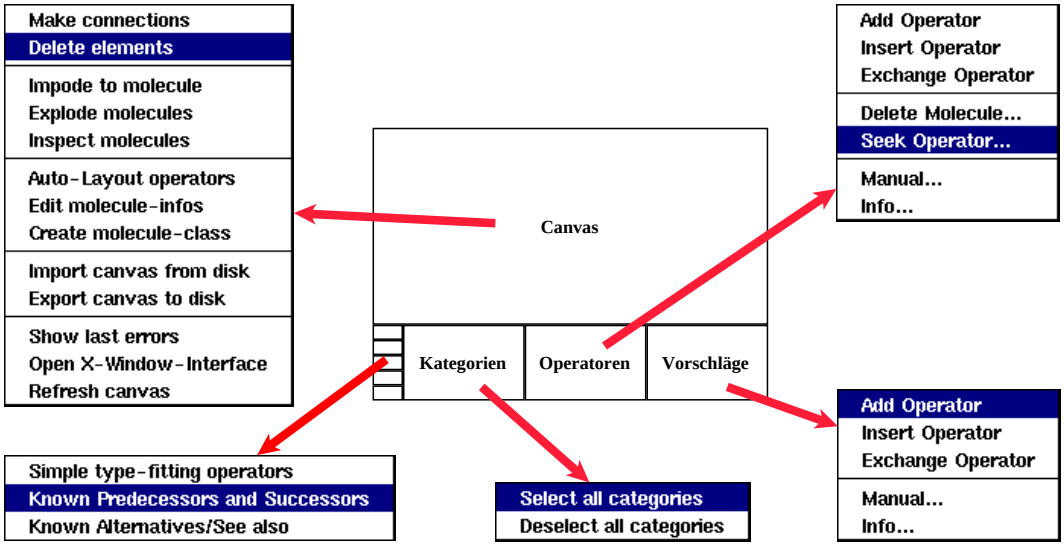


Abbildung 8.5: Kontextmenüs für verschiedene Bereiche von HCANVAS

***Settings*:** Setzen von Präferenzen

Kategorienliste (*Categories*): Hier finden sich die HORUS-Kapitel, durch Einrückung hierarchisch gegliedert. Der Benutzer kann beliebig viele daraus auswählen, wobei die jeweiligen Unterkapitel jeweils mit selektiert werden. Um also alle Atome zu selektieren, muß nur einmal die Klasse *HAtom* selektiert werden, nicht jede Unterklasse extra.

Operatorliste (*Category-Operators*): Liste mit allen Operatoren aus Gruppen, die in der Kategorienliste selektiert sind.

Vorschlagsliste (*Suggestion-Operators*): Liste mit den Operatoren, die durch den letzten Aufruf der Vorschlagskomponente erzeugt wurde.

8.4 Kontextabhängige Menüs

Jeder der drei Listen (Kategorien, Operatoren und Vorschläge), sowie die Leinwand selbst verfügt darüber hinaus über ein eigenes Kontextmenü. Beim Start der Vorschlagskomponente kann ebenfalls noch näher spezifiziert werden, nach welchen Kriterien Operatoren gesucht werden sollen. Diese Kontextmenüs sind in Abbildung 8.5 dargestellt. Innerhalb der Listen mit Operatornamen (Operatoren und Vorschläge) kann jeweils der selektierte Namen benutzt werden, um einen Operator dieses Namens der Leinwand hinzuzufügen (*Add Operator*), in eine bestehende Verbindung einzuketten (*Insert Operator*), oder einen selektierten Operator zu ersetzen (*Exchange Operator*). Ferner kann die Kurzbeschreibung des Operators (*Info…*)

und die Hilfskomponente (Abbildung 6.1) mit der kompletten Erklärung abgerufen werden (*Manual…*). Wenn in der Hilfskomponente dann über Querverweise o.ä. zu anderen Operatoren verzweigt wird, wirkt diese Selektion auf HCANVAS zurück und umgekehrt. Auf die Informationen aus der Operator-Datenbasis kann in diesem Hypermanual bequem zugegriffen werden. In der Operatorliste ist es darüber hinaus möglich, mit Mustern nach Operatoren zu suchen (*Seek Operator…*) und erzeugte Moleküle zu löschen (*Delete Molecule…*).

Das Leinwand-Kontextmenü erlaubt folgende Aktionen:

***Make connections*:** Verbinden von selektierten Knöpfen mit sogenannten *Verbindungen*[3], die Alternative hierzu ist das in Abschnitt 8.6.2 beschriebene Verfahren

***Delete elements*:** Löschen von selektierten Operatoren oder Verbindungen

***Implode to molecule*:** Implodieren der selektierten Operatoren zu Molekülen (dies ist auch mehrstufig möglich)

***Explode molecules*:** Explodieren der selektierten Moleküle (Atome bleiben unverändert)

***Inspect molecules*:** Startet eine Unterleinwand mit dem selektierten Molekül im explodierten Zustand, auf der ursprünglichen Leinwand bleibt das Molekül implodiert (siehe Abschnitt 8.10.3)

***Auto-layout operators*:** Einfache Layoutfunktion, die Operatoren in eine topologische Sortierung bringt und neu auf der Leinwand anordnet (siehe Abschnitt 8.8).

***Edit molecule-infos*:** Hiermit lassen sich die Informationen zum Molekül editieren, die bei Atomen in der Operator-Datenbasis abgelegt sind. Dazu gehören die Kurz- und Langbeschreibung, Querverweise, Vorgänger und Nachfolgerfunktionen und so weiter (siehe Abschnitt 8.10.1)

***Create molecule-class*:** Erzeugen eines HORUS-Operators aus einem Molekül (siehe Abschnitt 8.11).

***Import canvas from disk*:** Laden eines Moleküls von Datei

***Export canvas to disk*:** Speichern eines Moleküls als Datei

[3] Kanten zwischen Ein- und Ausgabeparametern (Knöpfen) von Operatoren.

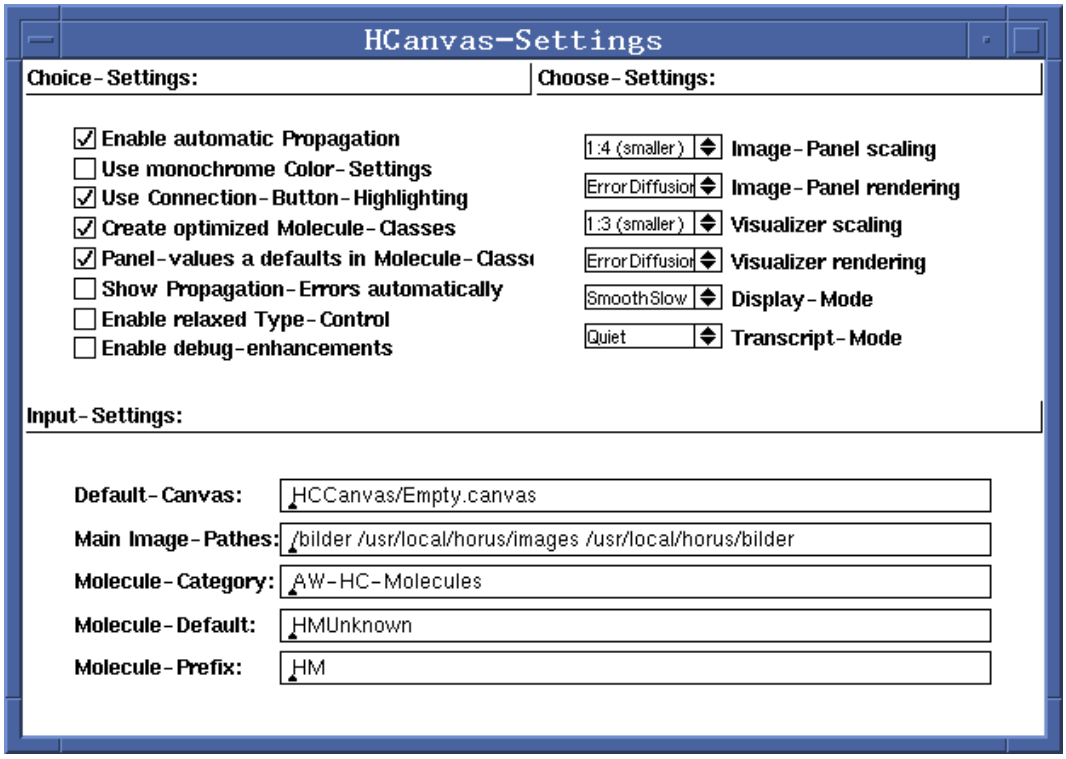


Abbildung 8.6: Präferenzeinstellungen für HCANVAS

***Show last errors*:** Wenn die Präferenz für sofortige Fehleranzeige aus ist, kann bei Auftreten eines Fehlers der letzte in HORUS aufgetretene Fehler abgerufen werden. Daß überhaupt ein Fehler aufgetreten ist, läßt sich an der roten Schrift im fehlerhaften Operator erkennen (siehe Abbildung 8.12)

***Open X-Window-Interface*:** Starten des Fensters zum Einstellen der Darstellungsparameter für die direkte HORUS-Visualisierung (siehe Abbildung 8.15 und Abschnitt 8.7)

***Refresh canvas*:** Neuzeichnen der Leinwand

Viele der oft benötigten Aktionen sind auch per Tastendruck auslösbar, beispielsweise das Löschen von Elementen mit der Löschtaste.

8.5 Präferenzeinstellungen

Jedes hinreichend komplexe System wird von verschiedenen Benutzern in unterschiedlicher Weise verwendet. Deshalb ist es nötig, Präferenzen zur Verfügung zu

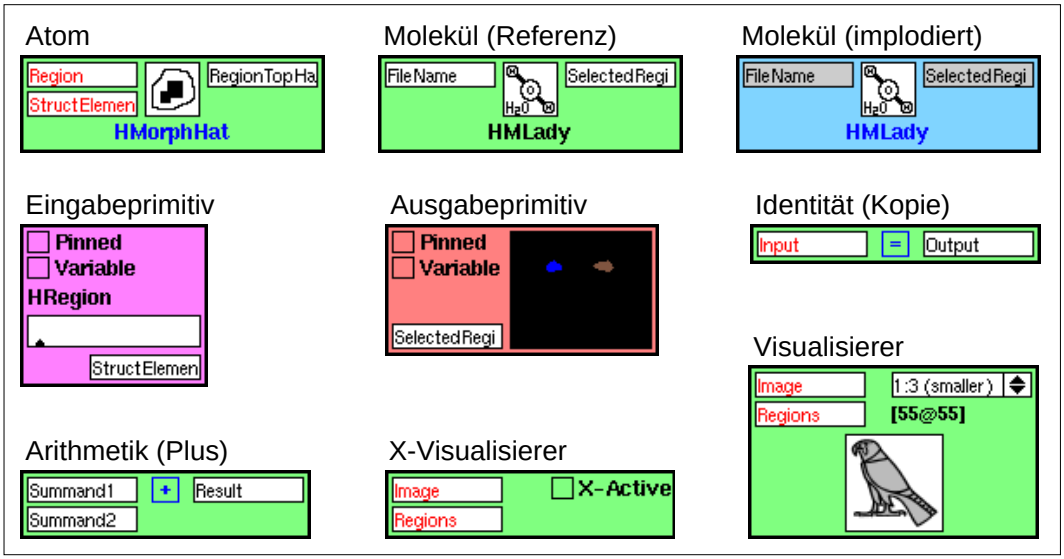


Abbildung 8.7: Arten von grafischen Elementen

stellen. Das Voreinstellungsfenster für HCANVAS ist in Abbildung 8.6 dargestellt. Hier lassen sich neben Einstellungen wie:

- Farb- oder Monochromdarstellung
- verwendete Algorithmen für die Visualisierung
- automatische Propagation an/aus
- automatische Fehleranzeige an/aus
- Standardgröße für Visualisierungsfenster

auch Zugriffspfade für benötigte Dateien einstellen. Auch Möglichkeiten für das Debugging werden angeboten, z.B. läßt sich das normale Leinwand-Kontextmenü um einige Punkte erweitern und das Protokollieren wichtiger Ereignisse im `Smalltalk`-Transcript kann veranlaßt werden.

8.6 Die Arbeitsfläche: Die Leinwand

8.6.1 Arten von grafischen Elementen

In Abbildung 8.7 sind die verschiedenen Arten von grafischen Primitiven dargestellt, die auf der Leinwand residieren können. Schon durch die unterschiedliche Farbgebung sollen bestimmte Klassen von Elementen für den Benutzer leicht zu unterscheiden sein. Grün werden alle Elemente dargestellt, die Operatorfunktionen ausüben

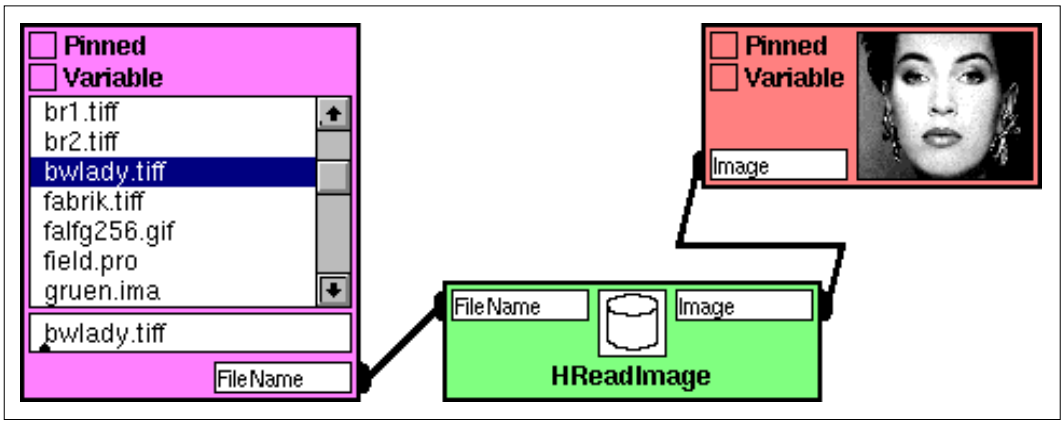


Abbildung 8.8: Operator mit seinen Ein- und Ausgabeprimitiven

und nicht weiter teilbar sind. Hierzu gehören neben Atomen auch Spezialoperatoren, die nicht in HORUS enthalten sind, wie Arithmetikoperatoren oder Visualisierer. Auch Moleküle sind eventuell grün dargestellt, nämlich genau dann, wenn sie *per Referenz* verwendet werden, also falls ein entsprechender HORUS-Operator in `Smalltalk` vorhanden ist und benutzt wird, wie jedes normale HORUS-Atom[4]. Wird das Molekül dagegen vollständig verwendet und ist nur wegen der Implosion als ein einzelnes Element auf der Leinwand sichtbar, dann wird es blau dargestellt.

Neben diesen beiden Grundtypen von Elementen existieren noch Ein- und Ausgabeprimitive, die lila bzw. rosa dargestellt werden. Die einzelnen Primitive besitzen Knöpfe, die miteinander mit Verbindungen verbunden werden können, wie in Abbildung 8.8 gezeigt. Bei Ein- und Ausgabeprimitiven ist die Besonderheit, daß sie immer fest an ihrem jeweiligen Knopf angehängt sind, sie können beispielsweise nicht durch Löschen der Verbindung von ihrem Operator „abgehängt“ werden. Aus diesem Grund können auch die Verbindungen zu Ein- bzw. Ausgabeprimitiven nicht selektiert werden und es können auch keine Vorschläge zu solchen Kanten erzeugt werden.

Die Operatoren der verschiedenen Funktionsgruppen könnten natürlich durch weitere Feinunterteilung weiter farblich unterschieden werden. Das Ergebnis eines solchen Vorgehens wäre aber mit Sicherheit ein sogenannter ”angry fruit salad”, Zitat aus [TNHD93]:

> **:angry fruit salad:** *n.* A bad visual-interface design that uses too many colors. This derives, of course, from the bizarre day-glo colors found in canned fruit salad. Too often one sees similar effects from interface designers using color window systems such as X; there is a tendency to create displays that are flashy and attention-getting but uncomfortable for long-term use.

[4]Hiermit ist auch die Möglichkeit für Rekursionen gegeben, allerdings ohne Abbruchbedingung, für die Implikationen siehe Abschnitt 8.11.1.

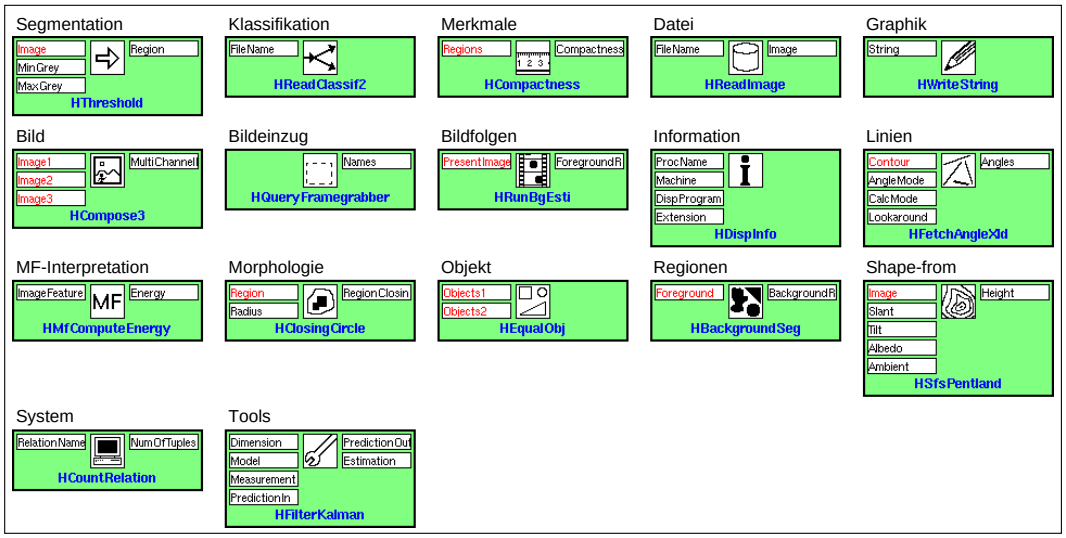


Abbildung 8.9: Operatoren aus verschiedenen HORUS-Kapiteln

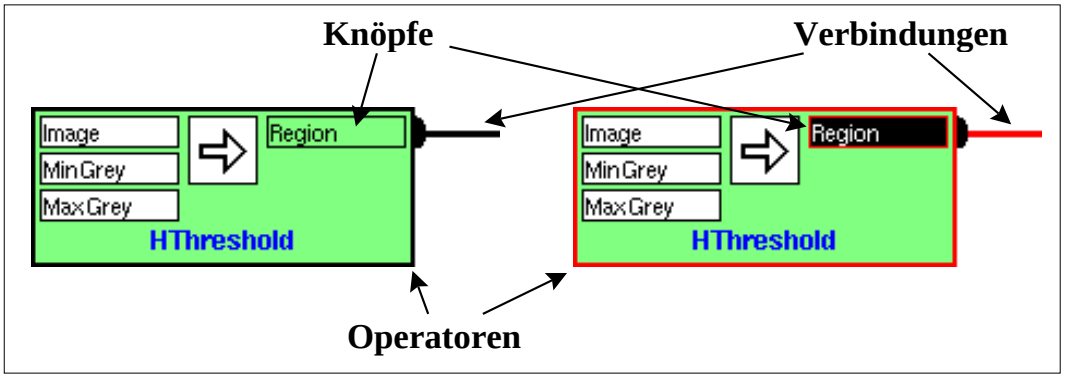


Abbildung 8.10: Hervorhebung der selektierten Elemente

Aus diesem Grund wurden stattdessen die unterschiedlichen Gruppen von Operationen durch Symbole unterschieden, wie in Abbildung 8.9 illustriert (vergleiche auch Abbildung 5.3).

8.6.2 Animation und visuelles Feedback

Es wurde versucht, dem Benutzer so viel visuelles Feedback über laufende Aktionen und Zustände zu geben wie möglich.

Eine wesentliche Grundlage für die verschiedensten Interaktionen ist die Möglichkeit, Elemente zu selektieren. Dies wird beispielsweise benötigt, wenn Vorschläge gemacht, Verbindungen geknüpft, oder mehrere Elemente gleichzeitig miteinander

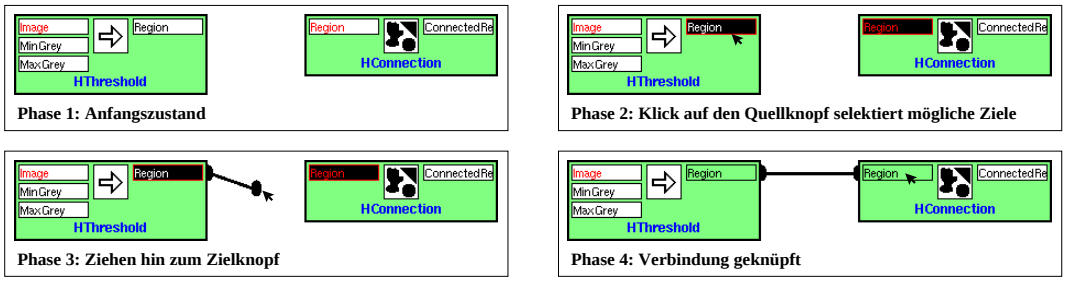


Abbildung 8.11: Phasen während des Knüpfens einer Verbindung

bewegt werden[5], kurz, wenn festgelegt werden soll, auf welche Elemente sich die nächste Aktion beziehen soll. Selektion in HCANVAS geschieht durch einfaches Anklicken mit der Maus, jeweils im Bereich, der durch die Sicht des Elements abgedeckt wird. Dies sind meist rechteckige Bereiche, die aber nicht notwenigerweise orthogonal ausgerichtet sein müssen (siehe Abschnitt 7.10). Nochmaliges Klicken deselektiert Elemente, Klicken auf die Leinwand deselektiert alles, Ziehen eines Rechtecks mit gedrückter Umschalttastes selektiert alle Operatoren und Verbindungen innerhalb des Rechtecks. In HCANVAS können Knöpfe, Operatoren und Verbindungen selektiert werden und werden in diesem Zustand anders dargestellt (siehe Abbildung 8.10).

In Abbildung 8.11 ist der zur Auswahl im Leinwand-Kontextmenü alternative Vorgang zum Knüpfen einer Verbindung zwischen zwei Knöpfen dargestellt. Im Ablauf wird zunächst ein Knopf mit niedergedrückter Umschalttaste angeklickt, hierdurch werden alle möglichen Zielknöpfe selektiert (also solche mit kompatiblem Datentyp, die noch nicht anderweitig verbunden sind).

Dann wird die Verbindung wie ein Gummiband zum gewünschten Zielknopf gezogen. Wenn dann die Maustaste auf dem Knopf losgelassen wird, wird die Verbindung geknüpft, was in der Abbildung am Farbwechsel der an der Verbindung beteiligten Knöpfe sichtbar ist.

Ein weiterer Bereich, in dem visuelles Feedback hilfreich ist, ist die Darstellung von Berechnungszuständen von Operatoren. In Abbildung 8.12 sind alle möglichen Zustände dargestellt. Der Name des Operators (`HReadImage`) ist normalerweise schwarz, wenn der Operator korrekt berechnet wurde und ein Ergebnis vorliegt. Er kann blau sein, wenn der Wert kurzzeitig während einer Propagation invalidiert wird (siehe Abschnitt 7.1), weil das Ergebnis nicht gebraucht wird oder weil notwendige Eingabeparameter fehlen, für die es in der Operator-Datenbasis für diesen Operator auch keine Vorgabewerte gibt (z.B. das Eingabebild für einen Mittelwertfilter, eine Vorgabe für die Größe dagegen existiert). Im letzteren Fall wird zusätzlich der Name des betroffenen Eingabeparameters rot dargestellt. Die roten Eingabeknöpfe sind

[5] Selbstverständlich läßt sich ein einzelnes Element einfach durch Anfassen mit der Maus an eine beliebige Position bringen.

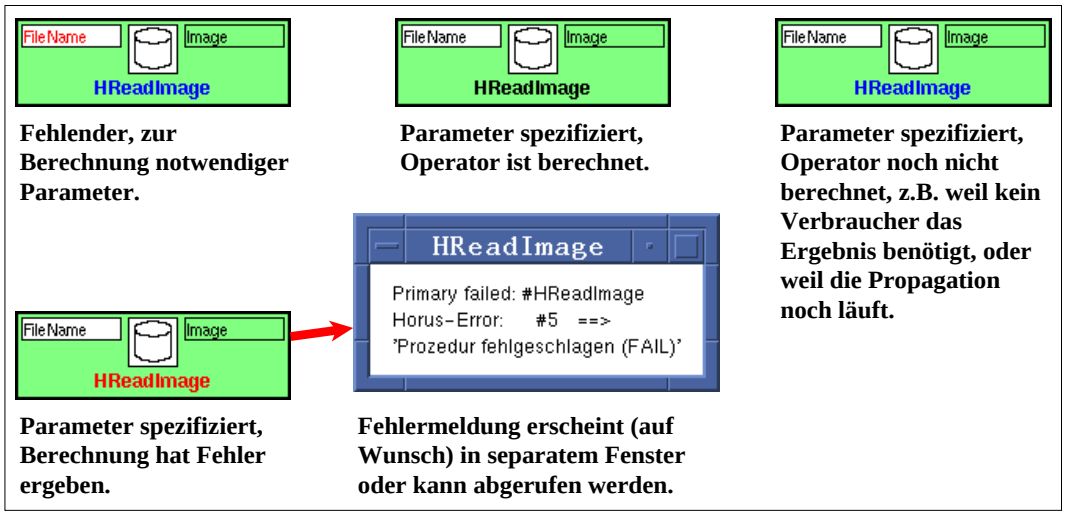


Abbildung 8.12: Darstellung verschiedener Zustände

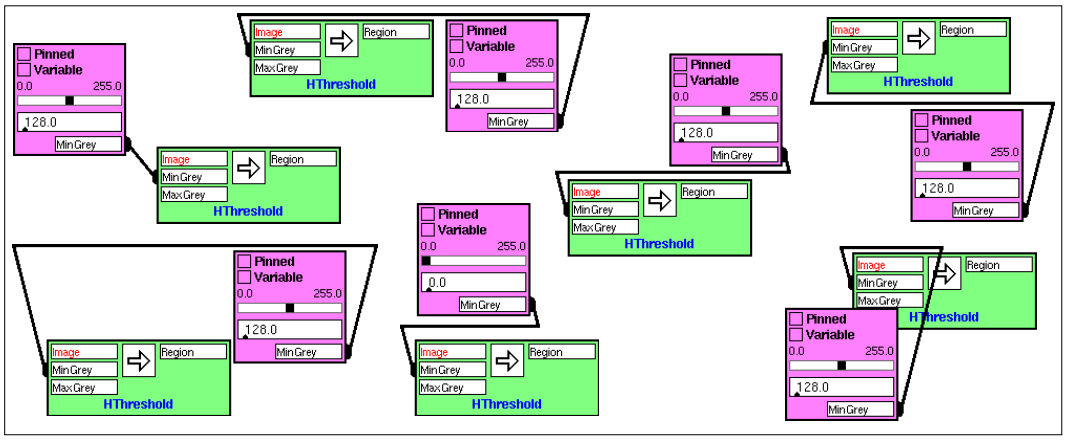


Abbildung 8.13: Automatisches, positionsabhängiges Führen der Verbindungen

immer diejenigen, die eine Parametrierung durch den Benutzer benötigen. Der Operatorname ist dagegen rot, wenn ein Fehler bei der Berechnung aufgetreten ist. In diesem Fall kommt je nach Präferenzeinstellung automatisch oder auf Anforderung eine Fehlermeldung auf den Bildschirm, wie in der Abbildung dargestellt.

8.6.3 Automatisches Führen von Verbindungen

Für die Führung der Verbindungen selbst sind zwei konträre Ansätze denkbar: entweder vollautomatisch in Art gängiger Werkzeuge im Bereich Platinenlayout, so daß möglichst Kreuzungen und Verdeckungen durch Operatoren vermieden werden, oder die Möglichkeit, Stützpunkte in eine normalerweise auf kürzestem Weg geführ-

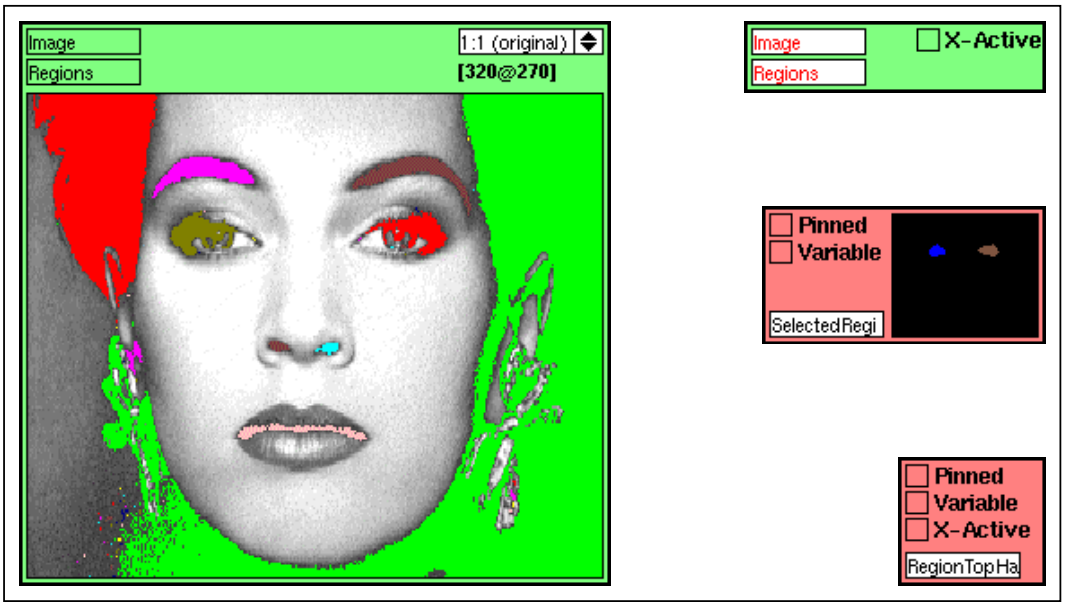


Abbildung 8.14: Visualisierungsoperatoren für Bilder und Regionen

te Verbindungslinie einzufügen. Die erste Möglichkeit führt zu nichtpolynomiellem Rechenaufwand, die zweite ist für den Benutzer unbefriedigend.

Hier wurde deshalb ein Kompromiß gewählt, nämlich eine gute Heuristik. Vollständig vermieden werden kann eine Kreuzung oder Überdeckung dadurch nicht, aber der Benutzer muß sich keine Gedanken über das Führen der Verbindungen machen. In Abbildung 8.13 ist das Führen für eine einzelne Verbindung zwischen einem Operator und einem Eingabeprimitiv für verschiedene Lagen zueinander gezeigt.

8.7 Spezielle Operatoren

In Abbildung 8.14 sind vier Formen von Visualisierungswerkzeugen zu sehen, in rosa Ausgabeprimitive, in grün Spezialoperatoren. Im Fall von Ausgabeprimitiven kann jeweils lediglich derjenige Wert dargestellt werden, der aktuell am Ausgabeknopf des Primitivs anliegt, bei bildhaften Daten also nur entweder Bilder *oder* Regionen.

Eine häufig gewünschte Funktionalität ist jedoch, das Ursprungsbild mit dem überlagerten Segmentierungsergebnis *zusammen* darzustellen. Dies ist beim Ausgabeprimitiv nicht möglich, weil dessen Verbindung fixiert ist.

Für diesen Spezialfall wurde ein Operator geschaffen (`HCVisualizer`), der zwei Eingabewerte, nämlich ein Bild und eine Region besitzt und beide überlagert darstellt.

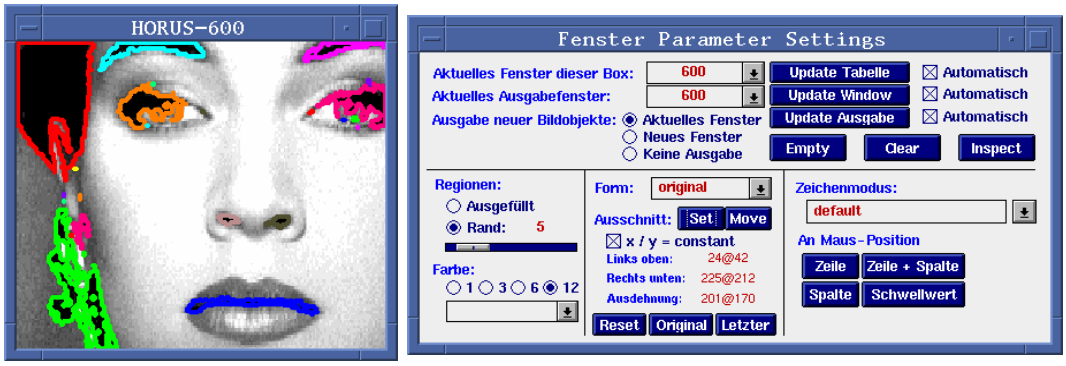


Abbildung 8.15: X-Visualisierung mit Darstellungsparametern

Neben den jeweiligen Versionen, die eine Visualisierung in `Smalltalk` direkt auf der Leinwand vornehmen, existieren sowohl von der Primitiv- als auch von der Operatorvariante jeweils noch eine Version, die direkt die HORUS-eigene X-Windows-Visualisierung benutzt und auf der Leinwand selbst keine Darstellung erzeugt. Vorteile ergeben sich hier bei der Geschwindigkeit und den gebotenen Möglichkeiten bei der Visualisierung. Das Konzept funktioniert ähnlich wie die Kopplung zwischen HDEVELOP und HINSPECTOR (siehe Abschnitt 4.1.3). Für die direkte HORUS-Visualisierung existiert in der `Smalltalk`-Schnittstelle zwar kein so ausgefeiltes Werkzeug wie HINSPECTOR, aber immerhin werden erweiterte Möglichkeiten zur Einstellung von Darstellungsparametern angeboten, wie in Abbildung 8.15 zu sehen ist.

8.8 Gruppierung

In Abbildung 8.16 ist das in Abschnitt 2.1 vorgestellte Anwendungsbeispiel dargestellt, und zwar mit fast allen Primitiven im geöffneten Zustand. Wie leicht erkennbar, ist diese Darstellung sehr unübersichtlich, selbst wenn nur die relevanten Operatoren und nicht ihre Parameterprimitive zu sehen sind, wie in Abbildung 8.17. Der Platz zwischen den Operatoren, der vorher für die Darstellung der Primitive gebraucht wurde, ist jetzt leer, aber die Ausdehnung des Graphen hat sich nicht wesentlich verringert.

Außer der Möglichkeit, Teile des Graphen zu Molekülen zu implodieren, gibt es noch eine weitere Möglichkeit, die Übersicht zu verbessern, nämlich, eine automatisches Layout vorzunehmen. Hierbei werden die auf der Leinwand befindlichen Operatoren zuerst topologisch sortiert (siehe Abschnitt 7.8) und dann entsprechend der aktuell sichtbaren Leinwandgröße angeordnet, mit ausreichend Platz für Parameterprimitive dazwischen. In Abbildung 8.18 ist das Ergebnis des automatischen Layouts für den in Abbildungen 8.16 und 8.17 gezeigten Graphen zu sehen.

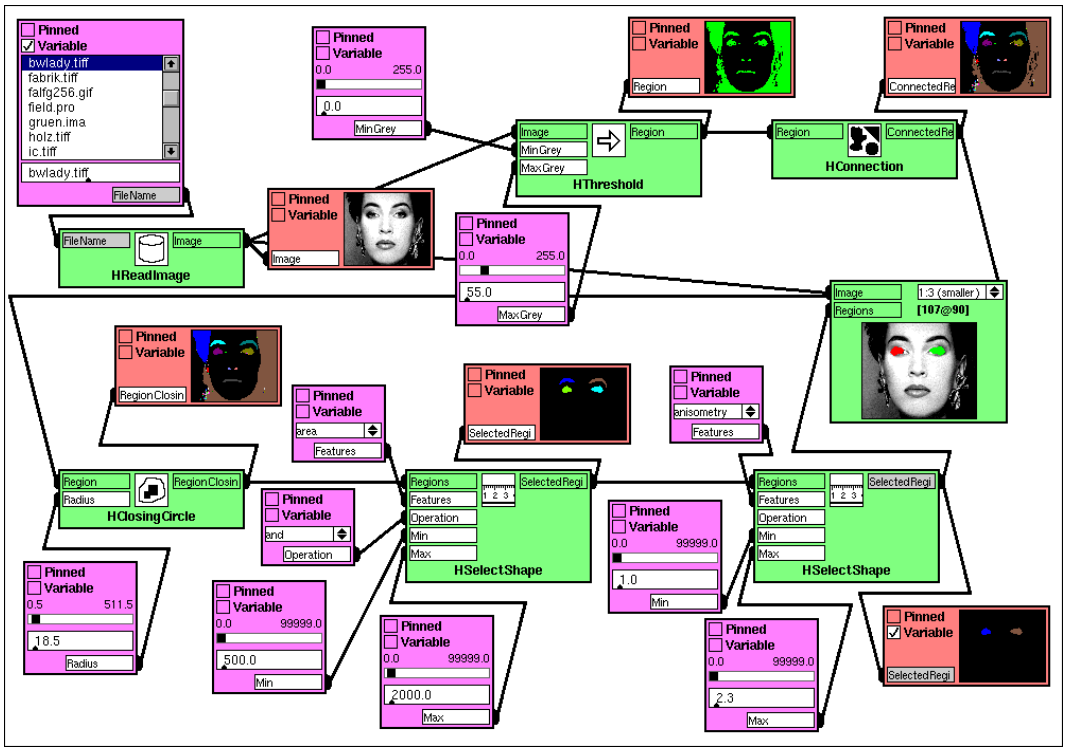


Abbildung 8.16: Beispielprogramm, fast alle Primitive geöffnet

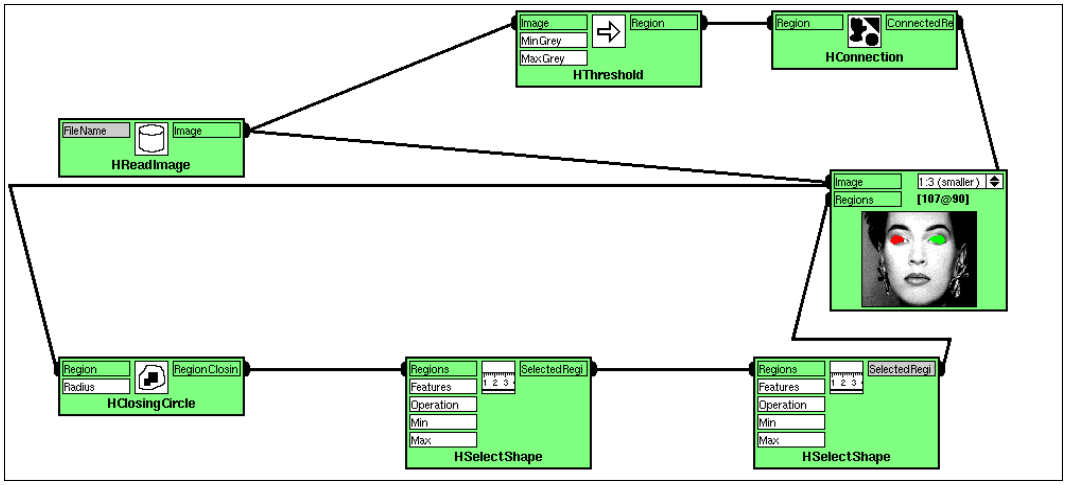


Abbildung 8.17: Beispielprogramm, alle Primitive geschlossen

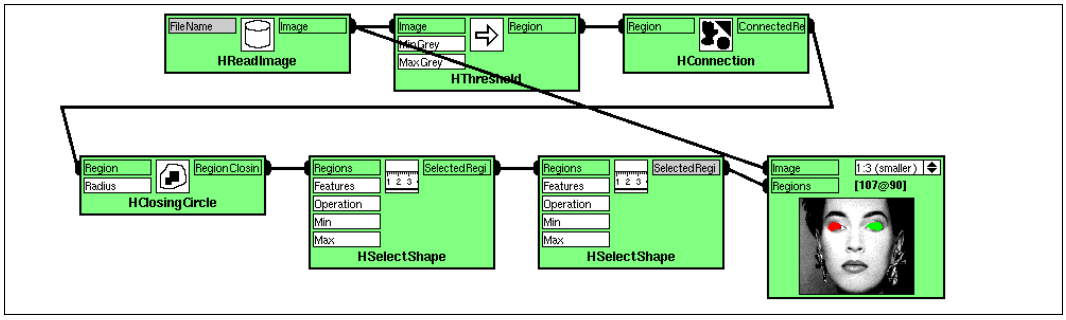


Abbildung 8.18: Beispielprogramm, Operatoren nach automatischem Layout

8.9 Vorschlagskomponente

Mittels der in den vorangehenden Abschnitten vorgestellten Konzepte lassen sich im Sinne der in Abschnitt 2.5 angesprochenen *schrittweisen Verfeinerung* Bildanalyseprogramme aufbauen. In Abbildung 8.19 ist oben zunächst der anfängliche Inhalt einer leeren Leinwand dargestellt. Unter leerer Leinwand ist dabei eine Startkonfiguration zu verstehen, die zwar mit den Präferenzeinstellungen (Abschnitt 8.5) verändert werden kann, aber normalerweise aus einer Bildquelle (`HReadImage`) und einem Visualisierer (`HCVisualizer`) besteht, der in Abschnitt 8.7 beschrieben wurde.

Durch Selektieren der Verbindung zwischen dem eingelesenen Bild und der Regionenkomponente des Visualisierers wird die Stelle markiert, an der die Verfeinerung ansetzen soll. Das Anklicken des *Suggestion*-Knopfs ruft verschiedene Vorschlagsarten ab, hier werden zunächst die bekannten Vorgänger- und Nachfolgeroperatoren (*Known Predecessors and Successors*) verwendet. Diese werden in der Spalte *Suggestion Operators* angezeigt. Durch Auswahl eines Namens in dieser Liste und Anklicken von *Insert Operator* im Kontextmenü wird der Operator `HThreshold` dann in die selektierte Verbindung eingefügt.

Wenn der Anwender danach die Operation durch eine andere ersetzen möchte, weil die Ergebnisse nicht befriedigend sind, kann er jederzeit den Operator selektieren und durch einen anderen ersetzen. In der Abbildung wird die Vorschlagskomponente dazu nochmals angestoßen, diesmal mit Alternativen (*Known Alternatives/See also*). Unter den hier abgerufenen Alternativen wird der Name `HRegionGrowing` gewählt und im Kontextmenü dann mit *Exchange Operator* der selektierte Operator ersetzt.

Bei allen Aktionen des Benutzers werden die Ergebnisse hierbei sofort sichtbar, so daß ein Experimentieren mit verschiedenen Operatoren und Parametrierungen ermöglicht wird.

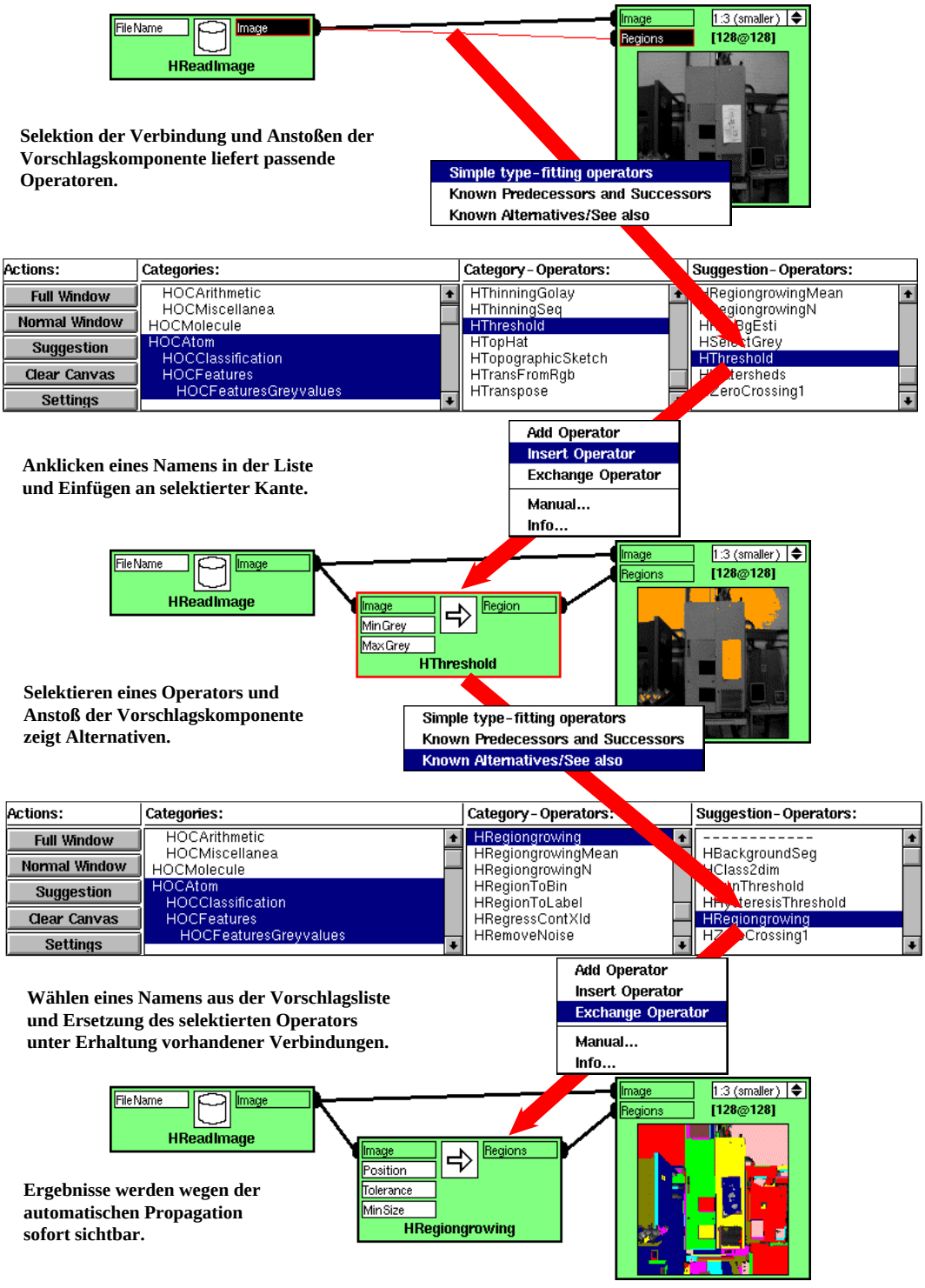


Abbildung 8.19: Schrittweise Verfeinerung

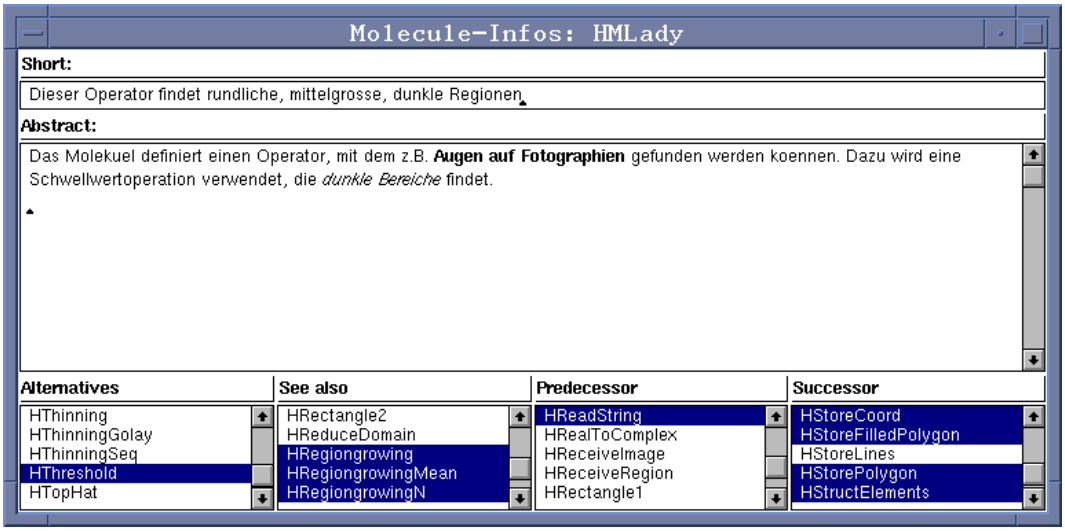


Abbildung 8.20: Semantische Informationen über das Molekül

8.10 Erzeugung von Molekülen

8.10.1 Ändern von semantischen Informationen

Wenn das Konzept des Bildens von Molekülen nicht nur auf den Zweck der Fokussierung (siehe Abschnitt 7.7) beschränkt bleiben soll, sondern auch, um neue, komplexere Operatoren auf Basis von bestehenden zu erzeugen, ist es notwendig, die entsprechenden Informationen für die Operator-Datenbasis verändern zu können. Zum einen muß zumindest eine Kurzbeschreibung angelegt werden, ohne die die Funktionsweise des erzeugten Moleküls nur durch Analyse herauszubekommen wäre oder anhand der Namensgebung erraten werden könnte. Das Dialogfenster zum Ändern dieser Informationen ist in Abbildung 8.20 dargestellt.

Zum anderen ist es sinnvoll, den Ein- und Ausgabeparametern des Moleküls neue Namen, entsprechend ihrer neuen Bedeutung zu geben. Auch alle anderen Informationen können natürlich geändert werden. Beispielsweise wäre es möglich, den Operator `HReadImage` zu einem Molekül zu machen, dessen Eingabeparameter nicht mehr vom Datentyp *Dateiname* sein soll, sondern vom Typ *Zeichenkette* oder dessen Vorgabewert für den Dateinamen nicht mehr `'fabrik'`, sondern `'hammer'` ist.

Die Fenster zum Ändern der Parameter für Ein- und Ausgabeprimitive sind in Abbildungen 8.21 und 8.22, zusammen mit den jeweiligen Primitivdarstellungen für die möglichen Datentypen zu sehen, siehe auch Abschnitt 7.5.

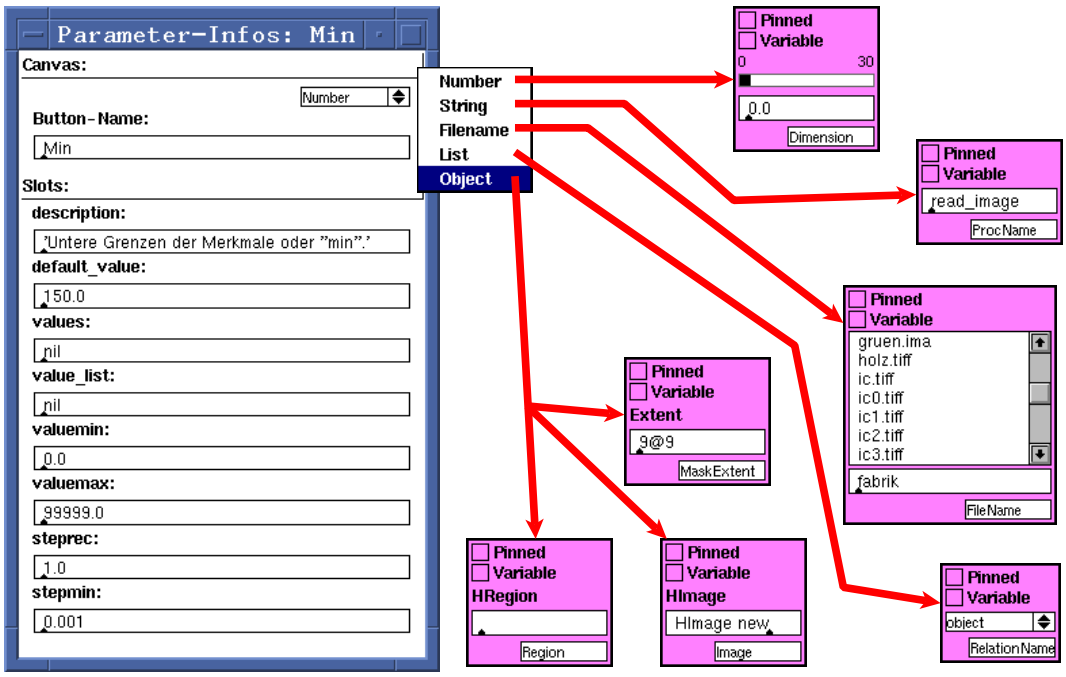


Abbildung 8.21: Eingabeprimitive für verschiedene Datentypen

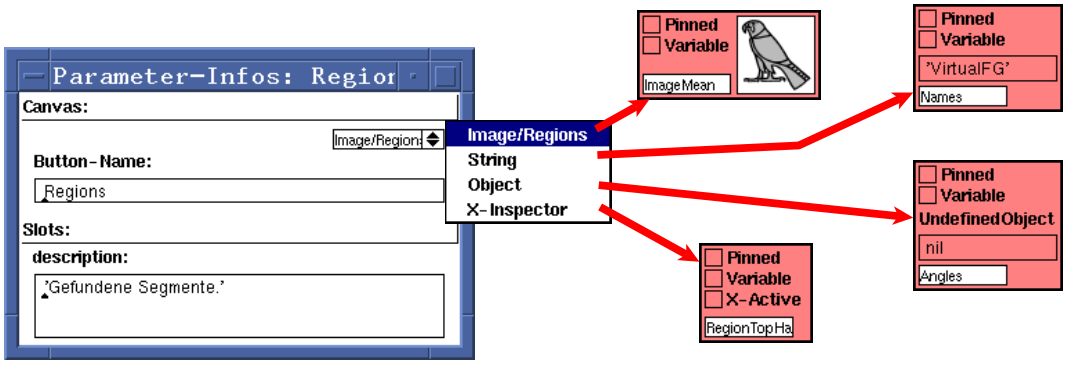


Abbildung 8.22: Ausgabeprimitive für verschiedene Datentypen

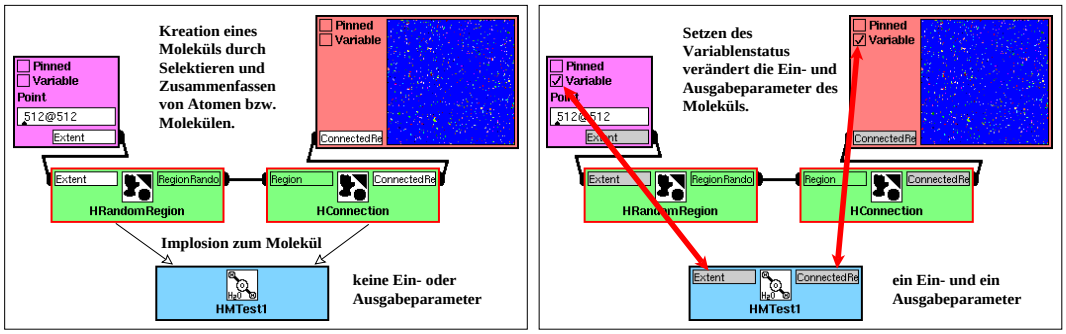


Abbildung 8.23: Konstante und Variable

8.10.2 Konstante und Variable

Das bereits in Abschnitt 7.4 vorgestellte Konzept zur Spezifikation, welche Werte im späteren Molekül zu Parametern werden und welche Konstante bleiben sollen, ist praktisch exakt umgesetzt worden. Die Hintergrundfarbe von Parametern wird beim Setzen des Variablenstatus leicht verändert, um sofort erkennen zu können, bei welchen es sich um Variablen handelt.

In Abbildung 8.23 sind links zwei Operatoren zu sehen, die zu einem Molekül (`HMTest1`) implodiert werden, das weder Ein- noch Ausgabeparameter hat, weil keine Knöpfe Variablenstatus besitzen und auch keine Verbindungen das Molekül verlassen. Durch Setzen des Variablenstatus bei einem Ein- und einem Ausgabeknopf bekommt das implodierte Molekül einen Ein- und einen Ausgabeparameter (im Bild rechts).

8.10.3 Fokussierung mit Unterleinwänden

Um implodierte Moleküle in ihrem Zustand zu belassen und dennoch inspizieren zu können (siehe Abschnitt 7.7), wurde die Möglichkeit geschaffen, separate Inspektorfenster aufzumachen, sogenannte Unterleinwände. Dies Konzept ist in Abbildung 8.24 dargestellt. Jede Änderung in der Unterleinwand wird sofort in der Hauptleinwand sichtbar und umgekehrt. Hierzu wurde eine entsprechende dynamische Verwaltung in `Smalltalk` realisiert, die die implodierten Moleküle entsprechend darstellt (siehe Abbildung 8.25).

8.11 Erzeugung von Code

Als letzter Schritt bei der Erzeugung eines Moleküls wird dieses, wie ein HORUS-Operator, als `Smalltalk`-Klasse erzeugt (siehe Abschnitt 5.4). Wenn kein Molekül auf der Leinwand selektiert ist, wird die gesamte Leinwand als Molekül abgelegt.

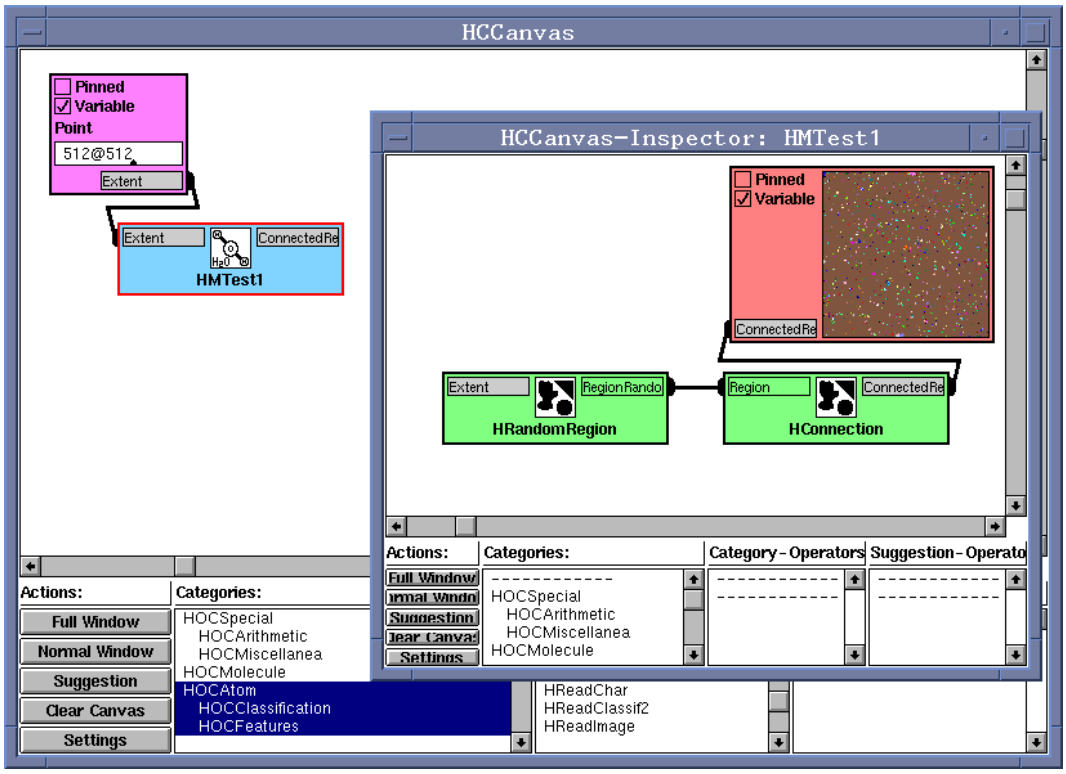


Abbildung 8.24: Unterleinwand zum Inspizieren eines implodierten Moleküls

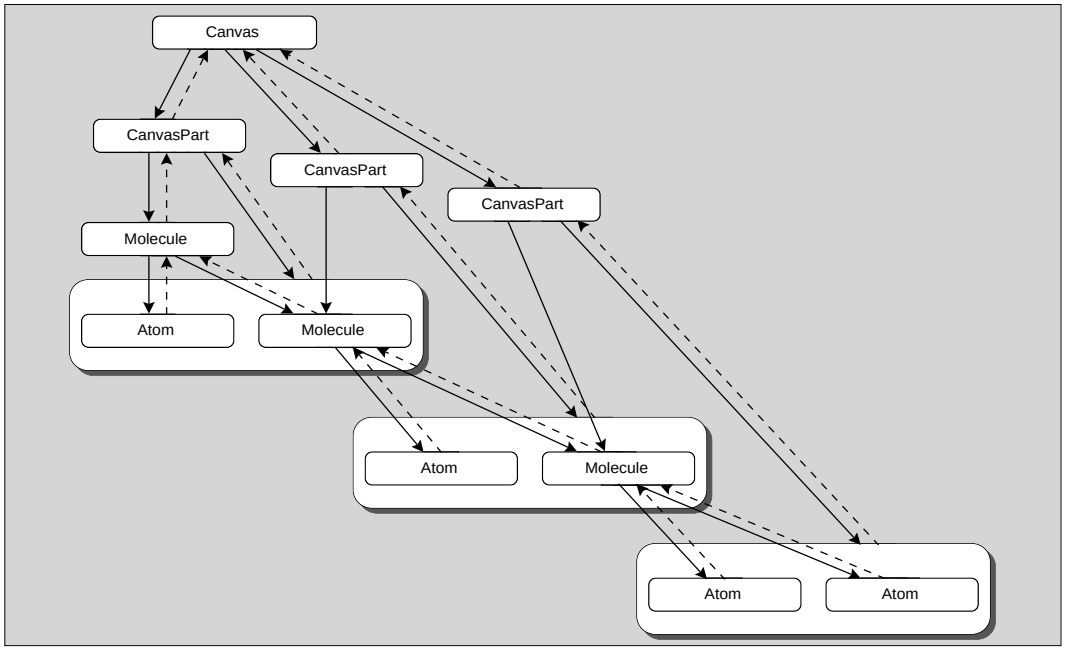


Abbildung 8.25: `Smalltalk`-Realisierung von Unterleinwänden (aus: [Was95])

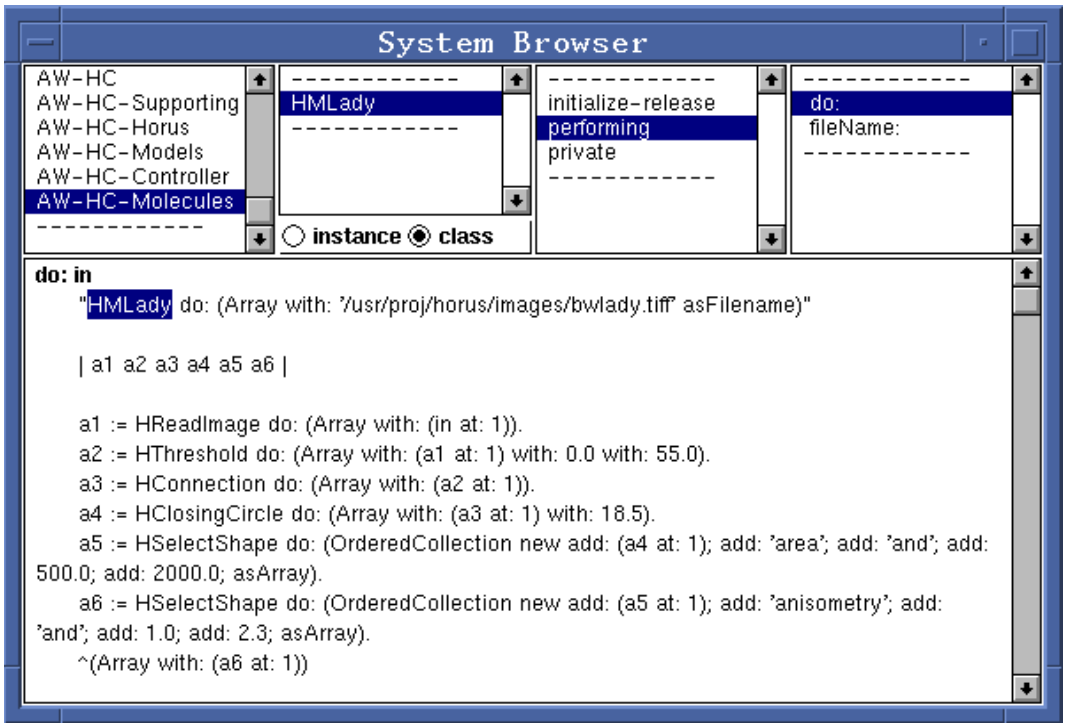


Abbildung 8.26: Generierter `Smalltalk`-Code für ein Molekül

Der so erzeugte Operator läßt sich danach auch direkt in `Smalltalk` verwenden und ist praktisch ununterscheidbar von einem HORUS-Atom. In Abbildung 8.26 ist der generierte `Smalltalk`-Code für das in Abschnitt 2.1 vorgestellte Beispiel gelistet. Für die Ausführung dieses Codes wird keine Propagationskomponente mehr benötigt, es handelt sich einfach um die Linearisierung der auszuführenden Operationen mit allen zuletzt eingestellten Parametrierungen. Neben dem Programmcode werden natürlich sämtliche anderen Informationen abgelegt, z.B. die Positionen der das Molekül bildenden Operatoren auf der Leinwand und die semantischen Informationen der Operator-Datenbasis.

Bei der sequentiellen Aufschreibung der beteiligten Operatoren werden durch das in Abschnitt 7.8 beschriebene Verfahren möglicherweise im Molekül vorhandene Schlingen entdeckt und der Benutzer entsprechend gewarnt. Entsprechende Sicherheitsabfragen vor Überschreiben schon existenter Klassen sind ebenfalls vorhanden.

8.11.1 Nicht implementierte Konzepte

Die folgenden in Kapiteln 5 bis 7 vorgestellten Konzepte wurden bislang noch nicht in der Implementierung berücksichtigt:

Aktivierung von Datenquellen und -senken: Aufgrund der Tatsache, daß Datenquellen in HORUS sehr selten vorkommen, wurde ihre korrekte Behandlung noch nicht implementiert. Bei Datensenken wurde eine einfache Heuristik implementiert, nämlich, daß sich Operatoren, die keine Ausgabeparameter besitzen, immer so verhalten, als wäre ein Ausgabeprimitiv angehängt. Hierdurch wird ihre Berechnung nach einer Invalidierung immer erzwungen.

Verbesserte Visualisierungskomponente: Neben der Visualisierung in Ausgabeprimitiven und der Möglichkeit, die HORUS-`Smalltalk`-Schnittstelle zur Darstellung zu verwenden, wäre es denkbar, entweder das Visualisierungswerkzeug HINSPECTOR anzubinden oder in `Smalltalk` selbst ein ähnlich mächtiges Werkzeug anzubieten.

Abspeichern in anderen Zielsprachen: Diese Funktionalität ist sehr einfach für gängige Programmiersprachen zu realisieren, denn schon beim Abspeichern als Molekül in `Smalltalk` wird eine Linearisierung, d.h. topologische Sortierung vorgenommen. In anderen Programmiersprachen ist der prinzipielle Mechanismus der selbe, anders ist lediglich der „syntaktische Zucker“.

Schleifen, Fallunterscheidungen: Wie schon in den Abschnitten 3.4 und 7.3 dargelegt wurde, ist die Umsetzung von Schleifen und Fallunterscheidungen mit einigen Problemen verbunden, ohne jedoch für die beabsichtigte Domäne wesentliche Vorteile zu bringen. Es wurde deshalb darauf verzichtet, die in Abschnitt 7.3 theoretisch entwickelten Konzepte zu implementieren. Falls Turing-Vollständigkeit ein absolutes Muß darstellte, würde es im Prinzip reichen, *nur* noch Fallunterscheidungen zu realisieren, denn aufgrund des Molekülkonzepts ist Rekursion bereits möglich, indem ein Molekül per Referenz verwendet wird. Lediglich die Abbruchbedingung müßte hinzugefügt werden.

Funktionsgruppen: Die in Kapitel 6 beschriebenen Funktionsgruppen und Eigenschaften von Operatoren ließen sich noch wesentlich erweitern. Die tatsächliche Unterteilung der Operator-Datenbasis ist zur Zeit lediglich nach HORUS-Kapiteln vorgenommen. Hieraus resultiert auch das bisherige Fehlen von Meta-Molekülen (siehe Abschnitt 9.2.1), die ohne entsprechende Beschreibungen ihrer Funktionalität nicht sinnvoll wären.

Molekül-Datenbank für Arbeitsgruppen: Die in Abschnitt 1.2 angesprochene Unterteilung der Operator-Datenbasis für Moleküle auf der Ebene von Arbeitsgruppen wurde noch nicht in Angriff genommen. Diese Funktionalität stellt sprachübergreifend ein gewisses Problem dar, wenn die Funktionalität des Wiedereinstellens von Informationen in die Operator-Datenbasis nicht im HORUS-Kern zur Verfügung gestellt wird. In der derzeitigen Implementierung werden Spezialoperatoren wie Visualisierer (`HCVisualizer`) oder

arithmetische Operationen (`HCMultiply`, `HCAdd` u.ä.), sowie Moleküle direkt in das `Smalltalk`-Image übernommen.

Kapitel 9

Zusammenfassung und Ausblick

9.1 Zusammenfassung

Die vorliegende Arbeit hatte zum Ziel, den Gedanken des Computer Aided Vision Engineering (CAVE) im Bereich von unterstützenden Werkzeugen voranzubringen. Es wurden Konzepte aufgezeigt, wie dieses Ziel erreicht werden kann, indem Vereinfachungen für Benutzer mit unterschiedlichem Vorwissen auf dem Gebiet der Bildanalyse erzielt werden.

Diese Vereinfachungen bestehen einerseits in der wesentlichen Verkürzung des normalen Erstellungsprozesses von Bildanalyselösungen, indem eine interaktive Auswahl und Parametrierung von Bildanalyseoperationen unter sofortiger Darstellung der Ergebnisse ermöglicht wird. Bildanalyseprogramme können praktisch ad hoc generiert werden, ohne daß syntaktische Regularien eingehalten werden müssen, die in der Vergangenheit immer wieder Probleme gemacht haben.

Die Notwendigkeit, Variablen zu deklarieren oder die richtigen Include-Dateien zu verwenden, entfallen durch interaktive Oberflächen wie HDEVELOP bereits weitgehend.

Wenn darüber hinaus, wie im erstellten System HCANVAS, eine echte grafische Programmierumgebung zur Verfügung steht, die auf Veränderungen mit konsistenter, aber minimaler Neuberechnung der benötigten Ergebnisse reagiert, dann kann ein echtes, spielerisches Erschließen des komplexen Gebiets „Bildanalyse“ erfolgen. Diese Minimalisierung erfüllt dabei gleichzeitig zwei Zwecke: Ressourcenschonung bei der Rechenzeit und Verringerung der Reaktionszeit des Systems für den Benutzer auf die Zeit zur Berechnung der notwendigen Operatoren plus einem Verwaltungsanteil; diese zusammen genommen liegen auf gängigen Workstations meist im Bereich von Sekundenbruchteilen.

Andererseits wird speziell für den weniger erfahrenen Benutzer die Auswahl und Parametrierung von Operatoren mit wissensbasierten Mitteln unterstützt. Diese Unter-

stützung bietet nicht nur die Möglichkeit, schneller die richtigen Operatoren auszuwählen, sondern führt gegenüber der klassischen Methode zu einer drastischen Verringerung der Fehlerquote, weil die meisten Syntax- und Semantikfehler (falscher Datentyp, Wertebereichsüberschreitung) von vornherein ausgeschlossen werden.

Die Möglichkeit zur Erstellung von Molekülen bietet die Grundlage, losgelöst von Implementierungsaufwand auf der untersten Ebene, Bildanalyseoperatoren zu erzeugen, die so mächtig sind, daß Bildanalysesysteme in Zukunft Operationen anbieten könnten, die in der Begriffswelt des Benutzers eine konkrete Bedeutung haben.

In der Arbeit wurde somit eine Vielzahl von Konzepten teils neu erarbeitet, teils in neuartiger Weise angewendet, so daß durch ihr Zusammenwirken eine neue Qualität bei der Lösung von Aufgabenstellungen entsteht, die in dieser Art bei ähnlichen Vorläufern wie Khoros, Explorer, KBVision oder AVS nicht vorhanden war.

Ein implizites Ziel der Arbeit war die Entwicklung möglichst allgemeingültiger Konzepte, losgelöst von der konkreten Implementierung in `Smalltalk`, die als ein gelungenes Beispiel für die Realisierbarkeit angesehen werden kann. Die hier erzielten Ergebnisse fließen direkt auch in andere Werkzeuge für Bildanalyse mit ein, z.B. wird die interaktive, nicht-textuelle Parametrierung zur Zeit in HDEVELOP integriert. Die meisten Konzepte sind außerdem auch auf andere Basissysteme als HORUS übertragbar. Voraussetzung dafür ist, daß eine geeignete Wissensbasis schon auf der Ebene der Operatoren zur Verfügung steht.

Durch diese Wissensbasis wird eine interaktive Parametrierung und die Konstruktion von Molekülen erst ermöglicht, indem Informationen wie Vorgänger- oder Nachfolgeroperatoren, Querverweise u.ä. dazu verwendet werden, dem Benutzer die Frage: „Und was kann ich jetzt tun?“ zu beantworten.

Die Übertragbarkeit der erarbeiteten Konzepte auf ähnlich strukturierte Anwendungsgebiete wie etwa Prozeßvisualisierung, Fertigungssteuerung oder Mathematik ist ebenfalls gegeben.

9.2 Ausblick

Nichts ist perfekt — obwohl die bis hier erzielten Ergebnisse bereits entscheidende Bereiche von CAVE abdecken, sind trotzdem Erweiterungen und Verbesserungen möglich, bei denen die bis jetzt implementierten Teile eine sehr gute Grundlage bilden.

9.2.1 Automatische Planung

Der Bereich „wissensbasierte Unterstützung“ ist ein Thema für sich, weshalb ihm in Kapitel 6 auch so viel Raum gewidmet wurde. Eine noch bessere Benutzerunterstützung wäre denkbar, indem eine Spezifikationskomponente für Zielvorgaben

hinzugefügt würde, mit der z.B. die ungefähren Regionen, die gefunden werden sollen, vom Benutzer vorgegeben werden. Das System versucht dann, den Weg von den Ausgangsdaten zur Zielspezifikation selbst zu finden. Dies wäre eine Art „Spezifikation durch Beispiel“, gepaart mit einer automatischen Planung, um das gewünschte Ziel zu erreichen.

Wie in Abschnitt 2.3 dargelegt wurde, bietet der hier vorgestellte Ansatz die Möglichkeit, sich die besten Fähigkeiten sowohl des Computers (Geschwindigkeit, Wissensbasis), als auch des menschlichen Benutzers (Vorstellungsvermögen, Deduktionsfähigkeit) zunutze zu machen.

Eine mögliche Erweiterung liegt darin, anstelle eines Brute-Force-Ansatzes (der im Bereich Bildanalyse kaum erfolgversprechend erscheint), das in der HORUS-Operator-Datenbasis vorliegende operatorspezifische Wissen um entsprechendes applikationsspezifisches Domänenwissen zu erweitern, z.B. in der Art von „Meta-Molekülen“. Diese könnten mit dem grafischen Editor erstellt werden und eine Art „Ideenbasis“ darstellen, in die schon einmal gewählte Ansätze eingestellt werden können, z.B. einen typischen Teilablauf zum Auffinden von Kanten (siehe Abbildung 9.1). Hierbei sollen die unterlegten Felder die Meta-Atome symbolisieren, die nur Äquivalenzklassen von Operatoren darstellen und hier noch nicht instantiiert sind. Jede funktionale Gruppe von Operatoren würde dann, solange sie nicht instantiiert ist, durch einen ausgezeichneten Repräsentanten dieser Gruppe vertreten, der die minimal notwendigen Parameter aufweist, bei einem Glättungsfilter beispielsweise würde lediglich ein Ein- und ein Ausgabebild verlangt werden. Bei einem Austausch dieses Repräsentanten durch eine „echte“ Instanz sind dann eventuell mehr Parameter zu belegen, was aber in HORUS durch entsprechende Vorgabewerte kein Problem darstellt. Möglicherweise vorhandene, nicht belegte Ausgabeparameter werden hierbei einfach nicht verwendet, sind also auch unproblematisch.

Eine saubere Trennung in drei Wissensbasen ist hierbei vonnöten:

1. **Operatorspezifisches Wissen**, z.B. Klassifizierungen bezüglich der Wirkungsweise, wie Kantenfilter, Glättungsfilter.
   Dieses Wissen wird in der Operator-Datenbasis abgelegt.

2. **Vorgehensspezifisches Wissen**, also domänenspezifisch „typische“ Herangehensweisen. Dieses wird in Form von „Meta-Molekülen“ abgelegt.

3. **Applikationsspezifisches Wissen**, das jeweils für ein spezifisches Anwendungsgebiet (Röntgenbildanalyse, Satellitenbildauswertung, Robotik usw.) gilt. Dieses wird ebenfalls in Form von Meta-Molekülen abgelegt.

Dabei könnte das applikationsspezifische Wissen jeweils vom Benutzer ausgewählt werden, abhängig von seiner Anwendung.

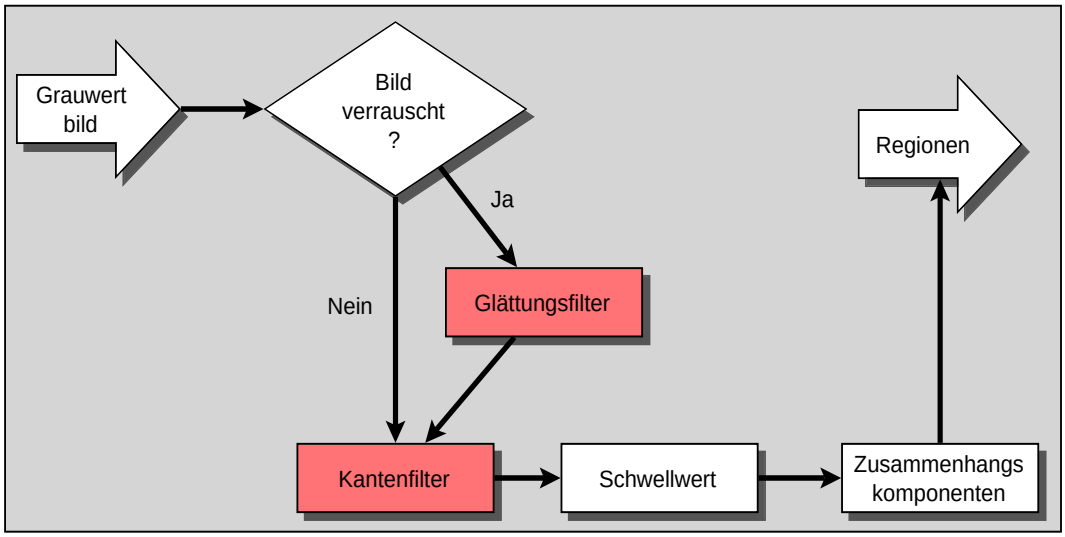


Abbildung 9.1: Typischer Ansatz zum Finden von Kanten im Bild

Bei den Eigenschaften von Operatoren wäre es auch denkbar, z.B. die Rechenzeit einzubeziehen, so daß Restriktionen vorgegeben werden können, beispielsweise, daß eine Berechnungsfolge in einem Teilgraph eine bestimmte maximale Zeitspanne nicht überschreiten darf.

9.2.2 Parallele Ausführung

Es ist denkbar, daß die Ausführung der Operatoren auf verschiedenen Prozessoren parallelisiert durchgeführt wird. Hierzu müßten Konzepte zur Lastverteilung integriert werden. Prinzipiell ist die Möglichkeit zur Ausführung von HORUS-Aufrufen über ein Netzwerk bereits vorhanden [LE90], aber die Entscheidung, welche Teilpfade parallel ausgeführt werden können, erfordert tiefgreifende Modifikationen des bis jetzt verwendeten Propagationsverfahrens.

In [LP95] wird allerdings zum Thema der datenflußbasierten Netzwerke festgestellt, daß diese bei Parallelisierung auch theoretisch recht große Herausforderungen darstellen, so daß eine stärker formal geprägte Vorgehensweise nötig ist.

Eine Erweiterung durch parallele Ausführung von Operatoren ist wünschenswert, wenn ein System wie HCANVAS für Echtzeitanwendungen oder in der industriellen Anwendung im Dauerbetrieb am Fließband eingesetzt werden soll. Das in Abschnitt 7.2 dargestellte Konzept zur benutzerseitigen Aktivierung von Berechnungen läßt sich sehr einfach für beliebige Ereignisse verallgemeinern. Hierunter fallen beispielsweise Signale wie „Neues Werkstück ist sichtbar“, „Neues Bild liegt vor“ oder auch durch den Prozeß selbst erzeugte Ereignisse, wie Timerevents oder der sofortige Start einer neuen Berechnung nach Abschluß der vorangehenden.

Literaturverzeichnis

[AHU85] A. V. Aho, J. E. Hopcroft und J. D. Ullman, *Data structures and algorithms*. Addison-Wesley, 1985. ISBN 0-201-00023-7.

[Ame93] Amerinex Artificial Intelligence, Inc., 409 Main Street, Amherst, MA 01002. *Knowledge-Based Vision Systems: Visual Programming Environment*, 1993.

[Aue92] U. Auer. *Integration von OSF/Motif in das Bildverarbeitungssystem HORUS*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1992.

[AVS92a] Advanced Visual Systems, Inc., 300 Fifth Ave., Waltham, MA 02154. *AVS Application Guide*, Mai 1992.

[AVS92b] Advanced Visual Systems, Inc., 300 Fifth Ave., Waltham, MA 02154. *AVS Technical Overview*, Oktober 1992.

[AVS92c] Advanced Visual Systems, Inc., 300 Fifth Ave., Waltham, MA 02154. *AVS User's Guide*, Mai 1992.

[AVS93] Advanced Visual Systems, Inc., 300 Fifth Ave., Waltham, MA 02154. *AVS Module Reference*, Februar 1993.

[AZJA94] M. S. Atkins, T. Zuk, B. Johnston und T. Arden. Role of Visual Languages in Developing Image Analysis Algorithms. In: *Proceedings IEEE Symposium on Visual Languages*, Seiten 262–269. IEEE Computer Society Press, 1994.

[Bau90] B. Baumgarten, *Petri-Netze: Grundlagen und Anwendungen*. BI-Wissenschafts-Verlag, 1990. ISBN 3-411-14291-X.

[BB82] D. H. Ballard und C. M. Brown, *Computer Vision*. Prentice-Hall, 1982.

[BD80] J. B. Brooke und K. D. Duncan. Experimental studies of flowchart use at different stages of program debugging. *Ergonomics*, 23:1057–1091, 1980.

[Ber91] M. Berger. Application Visualization System (AVS). In: *2. Workshop Sichtsysteme: Visualisierung in der Simulationstechnik*, Nummer 294 in Informatik Fachberichte, Seiten 110–118, 1991.

[Ber92] M. Berger. Ergebnisvisualisierung und Simulation mit AVS. *Bild & Ton*, 45(12):343–346, 1992.

[BG90] A. Benveniste und P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. In: *IEEE Transactions Autom. Contr.*, Band 35, Seiten 525–546, Mai 1990.

[Cha87] S.-K. Chang. Visual Languages: A Tutorial and Survey. *IEEE Software*, Seiten 29–39, Januar 1987.

[Cit93] W. Citrin. Requirements for Graphical Front Ends for Visual Languages. In: *Proceedings of the IEEE 1993 Symposium on Visual Languages*, August 1993. Bergen, Norway, URL: ftp://ftp.cs.colorado.edu/pub/techreports/citrin/VL93.ps.Z.

[CSK+89] B. Curtis, S. Sheppard, E. Kruesi-Bailey, J. Bailey und D. Boehm-Davis. Experimental evaluation of software documentation formats. *Journal of Systems and Software*, 9(2):167–207, 1989.

[Eck88] W. Eckstein. Das ganzheitliche Bildverarbeitungssystem HORUS. In: H. Bunke, O. Kübler und P. Stucki, Herausgeber, *Proc. 10. DAGM Symposium*, Band 180 von *Informatik-Fachberichte*, Seiten 53–59. Springer-Verlag, 1988. Zürich.

[Eck93] W. Eckstein. *HORUS Referenzmanual, Version 3.14*. Technische Universität München, Forschungs- und Lehreinheit Informatik IX, Forschungsgruppe Bildverstehen, 1993.

[ELM+93] W. Eckstein, G. Lohmann, U. Meyer-Gruhl, R. Riemer, L.A. Robles und J. Wunderwald. Benutzerfreundliche Bildanalyse mit HORUS: Architektur und Konzepte. In: S. J. Pöppl, Herausgeber, *Mustererkennung*, Informatik aktuell. Proc. DAGM Symposium, Springer-Verlag, 1993.

[Eur94] G. Euringer. *Entwicklung eines objektorientierten Bildinspektors unter HORUS*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1994.

[Exp92a] Silicon Graphics, Inc., Mountain View, California, USA. *IRIS Explorer 2.0 Module Writer's Guide*, 1992.

[Exp92b] Silicon Graphics, Inc., Mountain View, California, USA. *IRIS Explorer 2.0 User's Guide*, 1992.

[FLD90] M. Flickner, M. Lavin und S. Das. An Object-oriented Language for Image and Vision Execution (OLIVE). In: *International Conference on Pattern Recognition*, Seiten 561–571. IEEE Computer Society Press, 1990. Atlantic City, New Jersey.

[GGBM91] P. Le Guernic, T. Gauthier, M. Le Borgne und C. Le Maire. Programming real-time applications with SIGNAL. In: *Proceedings of the IEEE*, Band 79, September 1991.

[GHK95] H. Gall, M. Hauswirth und R. Klösch. Objektorientierte Konzepte in Smalltalk, C++, Eiffel und Modula-3. *Informatik-Spektrum*, 18(4):195–202, August 1995.

[Gli90] E. P. Glinert. Nontextual programming environments. In: S.-K. Chang, Herausgeber, *Principles of Visual Programming Systems*. Prentice-Hall, 1990.

[Gol84] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984. ISBN 0-201-11372-4.

[Gol85] A. Goldberg und D. Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1985. ISBN 0-201-11371-6.

[GP92] T. R. G. Green und M. Petre. When Visual Programs are Harder to Read than Textual Programs. In: G. C. van der Meer, M. J. Tauber, S. Bagnarola und M. Antavolits, Herausgeber, *Human-Computer Interaction: Tasks and Organisation, Proc. 6th European Conference on Cognitive Ergonomics*, 1992. CUD: Rome, URL: ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/tg/VisAndText.ps.

[GP95] T. R. G. Green und M. Petre. Usability Analysis of Visual Programming Environments. URL: ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/tg/VisProgEnvs.ps, 1995.

[GPB91] T. R. G. Green, M. Petre und R. K. E. Bellamy. Comprehensibility of visual and textual programs: a test of 'Superlativism' against the 'match-mismatch' conjecture. In: J. Koenemann-Belliveau, T. Moher und S. Robertson, Herausgeber, *Empirical Studies of Programmers: Fourth Workshop*, Seiten 121–146. Ablex, 1991.

[GQ94] M. Gorlick und A. Quilici. Visual Programming-in-the-Large versus Visual Programming-in-the-Small. In: *Proceedings IEEE Symposium on Visual Languages*, Seiten 137–144. IEEE Computer Society Press, 1994.

[Gre77] T. R. G. Green. Conditional Program Structures and Their Comprehensibility to Professional Programmers. *Journal for Occupational Psychology*, 50:93–109, 1977.

[Gre89] T. R. G. Green. Cognitive Dimensions of Notations. In: A. Sutchiff und L. Macaulay, Herausgeber, *People and Computers V*, Seiten 443–460. Cambridge University Press, 1989.

[Gre95] T. R. G. Green. Noddy's Guide to Visual Programming. In: *Human-Computer Interaction Group, „Interfaces“*, 1995. URL: ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/tg/NoddyOnVPLs.ps.

[Gün96] M. Günther. *Eine objektorientierte, grafische Oberfläche für das Bildanalysesystem HORUS unter NEXTSTEP*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1996. Noch nicht beendet.

[Hab89] P. Haberäcker, *Digitale Bildverarbeitung: Grundlagen und Anwendungen*. Hanser-Verlag, dritte Ausgabe, 1989. ISBN 3-446-15644-5.

[HBM+94] W. J. Hansen, B. Bell, G. A. McKaskle, G. Smedley und J. Poswig. The 1994 Visual Languages Comparison. In: *Proceedings IEEE Symposium on Visual Languages*, Seiten 90–97. IEEE Computer Society Press, 1994.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond und D. Pilaud. The synchronous data flow processing language LUSTRE. In: *Proceedings of the IEEE*, Band 79, Seiten 1305–1319, September 1991.

[Hil92] D. D. Hils. Visual Languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, Seiten 69–101, März 1992.

[IDL90] Research Systems, Inc., 777 29th Street, Suite 302, Boulder, CO 80303, USA. *Introduction to IDL*, März 1990.

[IDL93] Research Systems, Inc., 2995 Wilderness Place, Boulder, CO 80301, USA. *Interactive Data Language*, 1993.

[IUE95] J. Mundy, C. Kohl, T. Binford, D. Lawton, T. Boult, T. O'Donnell, M. C. Chiang, S. Fenster, D. Morgan, A. Hanson, R. Beveridge, K. Price, R. Haralick, V. Ramesh und T. Strat. *IUE Overview Document*. Image Understanding Environment Program, Juni 1995. URL: ftp://ftp.aai.com/.

[Jäh91] B. Jähne, *Digitale Bildverarbeitung*. Springer-Verlag, zweite Ausgabe, 1991.

[Jai88] R. Jain. Perception engineering. *Machine Vision and Application*, 1:73–74, 1988.

[Jak95] K. Jakisch. *Ein Vergleich von Oberflächen zur Bildanalyse*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1995.

[Klo93] K. Klotz, *Eine mehrschichtige Architektur zur Fehlerdiagnose und Fehlerbehebung bei der Entwicklung von logischen Programmen*. Infix-Verlag, 1993. Dissertation, Technische Universität München, Lehrstuhl Informatik IX.

[KM94] C. Kohl und J. Mundy. The Development of the Image Understanding Environment. In: *Conference for Vision and Pattern Recognition*, 1994. URL: ftp://ftp.aai.com/.

[Kne91] D. Knestel. *Der TRIAS-Graphen Editor*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1991.

[Knu73a] D. E. Knuth, *The art of computer programming*, Band 1: Fundamental algorithms. Addison-Wesley, 1973. ISBN 0-201-03809-9.

[Knu73b] D. E. Knuth, *The art of computer programming*, Band 3: Sorting and searching. Addison-Wesley, 1973. ISBN 0-201-03803-X.

[Knu81] D. E. Knuth, *The art of computer programming*, Band 2: Seminumerical algorithms. Addison-Wesley, 1981. ISBN 0-201-03822-6.

[KR90] B. W. Kernighan und D. M. Ritchie, *Programmieren in C*. Hanser-Verlag und Prentice-Hall, Zweite Ausgabe, 1990. ISBN 3-446-15497-3.

[LB95] C.-E. Liedtke und A. Blömer. Wissensrepräsentation im Konfigurationssystem für Bildanalyseprozesse CONNY. In: G. Sagerer, S. Posch und F. Kummert, Herausgeber, *Mustererkennung*, Informatik aktuell. Proc. DAGM Symposium, Springer-Verlag, 1995.

[LE90] W. Langer und W. Eckstein. Konzeption und Realisierung des netzwerkfähigen Bildverarbeitungssystems HORUS. In: *Proc. DAGM-Symposium*, Seiten 444–450. Springer-Verlag, 1990. Oberkochen-Aalen.

[LEM+93] P. Levi, W. Eckstein, U. Meyer-Gruhl, J. Pauli und K. Klotz, B. Radig, Herausgeber. *Verarbeiten und Verstehen von Bildern*. Handbuch der Informatik. Oldenbourg-Verlag, 1993. ISBN 3-486-20787-3.

[LM90] D. T. Lawton und D. M. Mead. A Modular Object Oriented Image Understanding Environment. In: *10th International Conference on Pattern Recognition*, Seiten 611–616. IEEE Computer Society Press, 1990.

[LMNR90] D. Larkin, M. Morse, J. Neider und C. Rose. *NeXTstep Concepts*. NeXT Computer, Inc., Redwood City, California, USA, 1990.

[Lob91] P. Lobner. *Aufbau eines Programms zur Erstellung und Ausgabe von Bitmap-Image-Demonstrationen unter Berücksichtigung der Anforderungen einer benutzerfreundlichen interaktiven grafischen Eingabe unter PHIGS/PHIGS+*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1991.

[LP95] E. A. Lee und T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–798, Mai 1995.

[M+93] J. Mundy et. al. The Image Understanding Environment: Overview. In: *Proceedings DARPA Image Understanding Workshop*, Seiten 283–288, April 1993. Washington, D.C.

[Man93] O. Mansfeld. *Entwicklung und Implementierung einer interaktiven Programmierumgebung für das Bildanalysesystem HORUS*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1993.

[Mes92] T. Messer, *Wissensbasierte Synthese von Bildanalyseprogrammen*. Infix-Verlag, 1992. Dissertation, Technische Universität München, Lehrstuhl Informatik IX.

[MR93] A. Morse und G. Reynolds. Overcoming current growth limits in UI development. *Communications of the ACM*, 36:73–81, April 1993.

[Mül92] R. Müller. *Entwurf und Implementierung einer interaktiven, graphischen Benutzeroberfläche für das Bildverarbeitungssystem HORUS*

unter C und Motif. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1992.

[Mye90] B. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[Nic94] J. V. Nickerson. Visual programming: Limits of graphic representation. In: *Proceedings IEEE Symposium on Visual Languages*, Seiten 178–179. IEEE Computer Society Press, 1994.

[NO92] A. Nye und T. O'Reilly, *X Toolkit Intrinsics Programming Manual*. O'Reilly & Associates, Inc., 1992.

[OKHC92] T. J. Olson, N. G. Klop, M. R. Hyett und S. M. Carnell. MAVIS: A visual environment for active computer vision. In: *Proc. IEEE Workshop on Visual Languages*, Seiten 170–176. IEEE Computer Society Press, September 1992. Seattle, WA.

[OSF90] Open Software Foundation. *OSF/Motif Programmer's Guide*, 1990.

[Pau92] D. W. R. Paulus, *Objektorientierte und wissensbasierte Bildverarbeitung*. Vieweg-Verlag, 1992. Promotionsschrift, Friedrich-Alexander-Universität Erlangen-Nürnberg.

[Pau93] J. Pauli, *Erklärungsbasiertes Computer-Sehen von Bildfolgen*. Infix-Verlag, 1993. Dissertation, Technische Universität München, Lehrstuhl Informatik IX.

[PBLR95] J. Pauli, A. Blömer, C.-E. Liedtke und B. Radig. Zielorientierte Integration und Adaption von Bildanalyseprozessen. *Künstliche Intelligenz*, 3:30–34, 1995.

[PL94] A. R. Pope und D. G. Lowe. Vista: A Software Environment for Computer Vision Research. In: *Conference for Vision and Pattern Recognition*, 1994.

[PVM93] J. Poswig, G. Vrankar und C. Moraga. Interactive Animation of Visual Program Execution. In: *Proceedings IEEE Symposium on Visual Languages*, Seiten 180–187. IEEE Computer Society Press, 1993.

[RA91] J. Rasure und D. Argiro. *Khoros User's Manual*. University of New Mexico, 1991. Albuquerque, NM 87131.

[Rei90] W. Reisig, *Petri-Netze: Eine Einführung*. Springer-Verlag, 1990.

[REK+92] B. Radig, W. Eckstein, K. Klotz, T. Messer und J. Pauli. Automatization in the design of image understanding systems. In: F. Belli und F. J. Radermacher, Herausgeber, *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Proc. 5th Int. Conference, IAE/AIE-92*, Seiten 35–45. Springer-Verlag, 1992. LNAI 604.

[RW91] J. Rasure und C. S. Williams. An integrated visual language and software development environment. *Journal of Visual Languages and Computing*, 2:217–246, 1991.

[RY92] J. Rasure und M. Young. Data flow visual languages. *IEEE Potentials*, 11(2):30–33, 1992.

[SA90] M. H. Sunwoo und J. K. Aggarwal. VisTA For A General Purpose Computer Vision System. In: *10th International Conference on Pattern Recognition*, Seiten 635–641. IEEE Computer Society Press, 1990.

[Sch95a] H. A. Schmid. Entwurf eines objektorientierten Baukastens zur Steuerung von Fertigungsanlagen. *Informatik-Spektrum*, 18(4):211–221, August 1995.

[Sch95b] W. Schneider. *Interaktive Eingabe von Prozedur-Parametern und Echtzeitvisualisierung im Bereich der Bildanalyse*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1995.

[SPI83] SPIDER Working Group. *SPIDER User's Manual*. Joint System Development Corp., Dezember 1983.

[Sud93] B. Sudar. *Entwurf einer visuellen Sprache für ein Bildverarbeitungssystem*. Diplomarbeit, Technische Universität München, Institut für Informatik, 1993.

[TNHD93] *The New Hacker's Dictionary*, August 1993. ISBN 0-262-68079-3, URL: ftp://ftp.leo.org/pub/comp/doc/misc/jargon/.

[UFK+89] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz und A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics & Applications*, 9(4):30–42, 1989.

[VM94] G. Viehstaedt und M. Minas. Interaction in Really Graphical User Interfaces. In: *Proceedings IEEE Symposium on Visual Languages*, Seiten 270–277. IEEE Computer Society Press, 1994.

[VS91] K. Voss und H. Süße, *Praktische Bildverarbeitung*. Hanser-Verlag, 1991.

[VW86] M. Vose und G. Williams. LabView: Laboratory Virtual Instrument Engineering Workbench. *Byte*, 11(9):84–92, September 1986.

[VWCB94] ParcPlace Systems, Inc. *VisualWorks Cookbook*, 1994.

[VWOR94] ParcPlace Systems, Inc. *VisualWorks Object Reference*, 1994.

[VWUG94] ParcPlace Systems, Inc. *VisualWorks User's Guide*, 1994.

[Wäh91] F. Wähner. *Entwurf und Implementierung einer interaktiven Bildverarbeitungsumgebung*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1991.

[Was95] A. Wastl. *Implementierung einer grafischen Oberfläche für das Bildanalysesystem HORUS unter Smalltalk*. Diplomarbeit, Technische Universität München, Lehrstuhl Informatik IX, 1995.

[Wil90] T. D. Williams. Image understanding tools. In: *International Conference on Pattern Recognition*, Seiten 606–610. IEEE Computer Society Press, 1990. Atlantic City, New Jersey.

[Wir83] N. Wirth, *Algorithmen und Datenstrukturen*. Teubner, dritte Ausgabe, 1983.

[Wir94] G. Wirtz. Modularization and Process Replication in a Visual Parallel Programming Language. In: *Proceedings IEEE Symposium on Visual Languages*, Seiten 72–79. IEEE Computer Society Press, 1994.

[Wol91] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, zweite Ausgabe, 1991. ISBN 0-201-51502-4.

[ZM93] S. Zehetbauer und U. Meyer-Gruhl. Segmentierung und Analyse drei- und vierdimensionaler Ultraschalldatensätze. In: S. J. Pöppl, Herausgeber, *Mustererkennung*, Informatik aktuell. Proc. DAGM Symposium, Springer-Verlag, 1993.

Index

P

R

S

T

U

V

W

X

Z