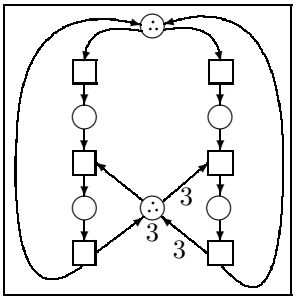
Entwicklung eines konzeptorientierten generischen Grafikeditors in COMMON LISP

Michael Wessel

2. Januar 1996

Studienarbeit
Betreuer:
Dr. V. Haarslev
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich „Kognitive Systeme“



Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung und Übersicht

1.1 Motivation und Inhalt dieser Studienarbeit

Diese Studienarbeit beschreibt die Konzeption und Entwicklung eines *wissensbasierten Grafikeditors* namens GENED. GENED ist ein Akronym für *generischer Editor*. Generisch bedeutet in diesem Fall die Verwendbarkeit des Grafikeditors für nicht feststehende Einsatzbereiche - so läßt sich GENED prinzipiell sowohl für die Erstellung von z.B. Petrinetzen als auch Schaltplänen nutzen.[1] Mit GENED können technische Grafiken bzw. Zeichnungen erstellt werden - im Gegensatz zu konventionellen Grafikeditoren unterstützt jedoch eine wissensbasierte Komponente den Benutzer beim Erstellen der Grafik dadurch, daß gewisse Konstellationen grafischer Elemente (wie z.B. ein Transistor in einem Schaltplan) vom System *erkannt* werden können.

Wesentlich zur Erreichung dieses Ziels ist die *Konzeptorientierung* und *Konfigurierbarkeit* des Systems. Ein *Wissensrepräsentationssystem* bietet hier die benötigten Repräsentationsformalismen und Dienste. Bezüglich der *Interaktionsformen* mit den dargestellten Grafikobjekten muß GENED als *objektorientiert* bezeichnet werden. GENED ist aber auch im softwaretechnischen Sinne objektorientiert entworfen. In einem Satz: GENED ist wissensbasiert, konzept- und objektorientiert.

In den folgenden Kapiteln werden u.a. auch Implementationsaspekte sowie die für ein Verständnis der Arbeit notwendigen Grundlagen betrachtet und benutzte Softwaresysteme diskutiert. Allerdings muß eine gewisse Vertrautheit mit der Sprache COMMON LISP und ihrer objektorientierten Erweiterung CLOS vorausgesetzt werden sowie allgemeine Erfahrung im Entwurf objektorientierter Software und KI-Programmierung (für eine Einführung s. [14] und [20, Kap. 1 - 3]).

Die Arbeit basiert auf einer Idee meines Betreuers, Herrn Dr. V. Haarslev, der bereits einen ähnlichen Grafikeditor auf dem Apple Macintosh implementiert hat (s. [1]). Ein verwandtes Programm wurde auch von Wang und Lee geschaffen (s. [2]) - dort wird jedoch im Gegensatz zur hier diskutierten wissensbasierten Annäherung ein Typ-theoretischer Ansatz verfolgt.

1.2 Allgemeines und Systemanforderungen

Mit Hilfe von GENED soll die Erstellung komplexer Grafiken *technischer Natur* ermöglicht werden. Im Gegensatz zu konventionellen Grafikeditoren zeichnet sich GENED dadurch aus, daß bestimmte grafische Konstellationen automatisch erkannt, also vom System *klassifiziert* werden können. Durch die Integration einer *Wissensbasis* läßt sich u.a. die Konfigurierbarkeit erreichen und der Klassifizierer realisieren. GENED läßt sich somit (ohne eine genauere Definition zu geben) als *wissensbasiertes System* betrachten. Wesentlich sind hier Methoden der *Wissensrepräsentation*.

GENED soll auch Anwendung bei der Erstellung von Programmen für vollständig *visuelle Programmiersprachen (VPs)* finden. Die Programme einer VP sind Grafiken, denen eine Semantik zugeschrieben wird. Vollständig in diesem Sinne bedeutet, daß alle Information bzgl. des momentanen Berechnungszustandes des Programmes in der Grafik selbst enthalten ist - ein solches Programm läßt sich in jedem Berechnungszustand von einem Rechner auf einen anderen übertragen und dort fortsetzen, da alle hierzu benötigten Zustandsinformationen im visuellen Programm selbst er-

[1] Dennoch ist und soll GENED kein elaboriertes CAD-Programm sein.

sichtlich sind. U.a. sind keine Informationen über Prozessorstack und -register o.ä. zu übertragen - das Progamm *ist* die Grafik.

Bei GENED handelt es sich um einen *objektorientierten* Grafikeditor, der keine der für *pixelorientierte* Grafikeditoren typischen Werkzeuge und Funktionen (wie versch. Füllmuster, Farbverläufe etc.) bietet. Letztendlich kann mit einem pixelorientierten Grafikeditor die Farbe eines jeden Pixels beliebig verändert werden - dies ist zudem die einzig mögliche Operation auf Pixeln. Im Gegensatz hierzu stehen bei einem objektorientierten Grafikeditor nicht manipulierbare Pixel oder Pixelgruppen im Vordergrund der Betrachtung, sondern die dargestellten *Objekte selbst.* Dementsprechend werden Interaktionsmöglichkeiten und Operationen angeboten, die der Natur oder Art dieser Objekte entsprechen. Die dargestellten Objekte werden als individuelle Entitäten betrachtet. Ein Pixel als eigenständige Entität bzw. Objekt zu betrachten ist in diesem Kontext nicht sinnvoll, da es im Gegensatz zu einem (komplexeren) Objekt keine Semantik trägt - die *Granularität* der Betrachtung ist zu fein; man kann auch von einer *subsymbolischen Ebene* sprechen.

Typischerweise werden von objektorientierten Grafikeditoren geometrische Objekte wie Kreis, Rechteck, Strecke usw. angeboten und als Ganzes manipuliert und betrachtet. Hier findet also eine Interaktion mit den Objekten selbst statt, wodurch eine konzeptionell höherstehende Betrachtungsebene gegeben ist, bzw. eine Abstraktion von Pixelgruppen erreicht wird. Allerdings wird hiermit ein Verlust an Allgemeinheit erkauft - nicht jedes Bild läßt sich durch Gruppierung geometrischer Objekte erstellen. Dafür gewinnt der Benutzer konzeptionell adäquatere, nämlich spezifischere Interaktionsformen.

Technische Grafiken und Programme visueller Programmiersprachen zeichnen sich durch klare Linienverläufe und einen begrenzten Satz an *grafischen Primitiven* (Kreis, Rechteck, Polygon etc.) aus, da die eindeutige und bequeme Interpretation durch den Betrachter oder die VP zu erreichen ist.[2] Zur Verdeutlichung denke man nur an Schalt- oder U-Bahnpläne. Somit liegt es nahe, einen objektorientierten Grafikeditor für die Erstellung und Manipulation derartiger Grafiken zu verwenden, da die grafischen Primitiven der technischen Zeichnung direkt mit den Objekten des Grafikeditors korrespondieren oder aus ihnen aggregiert werden können.

1.3 Benutzte Softwaresysteme im Kurzüberblick

GENED ist im ANSI-genormten LISP-Dialekt COMMON LISP geschrieben (s. [16]), wobei extensiv von den Möglichkeiten der (ebenfalls genormten) objektorientierten Erweiterung CLOS (COMMON LISP Object System) Gebrauch gemacht wurde (s. [15]). Für die Erstellung einer (für heutige Maßstäbe selbstverständlichen) direktmanipulativen Benutzeroberfläche wurde der COMMON LISP Interface Manager CLIM verwendet (s. [17]). Die Wissensbasis ist mit Hilfe einer Wissensrepräsentationssprache der KL-ONE-Familie namens CLASSIC (Classification of Individuals and Concepts) repräsentiert (s. [8], [9]). CLASSIC und CLIM werden in eigenen Kapiteln (soweit für das Verständnis dieser Arbeit notwendig) ausführlicher diskutiert.

[2] Eine Ausnahme bildet die VP „Pictorial Janus“ (PJ) - für ihre Programme kommt es nur auf die *Beziehungen* der Grafikelemente untereinander an.

1.4 Kapitelübersicht

Kapitel 2 macht den Leser mit der Idee und den wesentlichen Schlüsselkonzepten der Arbeit vertraut. Diverse notwendige Begriffe werden eingeführt und in späteren Kapiteln präzisiert.

Kapitel 3 beschreibt Ansätze und Formalismen zur Repräsentation und Modellierung von Beziehungen zwischen zweidimensionalen Grafikobjekten in der Zahlenebene $\mathbb{R}^2$ (s. auch [4, Kap. 1.4.3]). Ein eigener Vorschlag zur Realisierung und Berechnung derartiger Beziehungen wird dargestellt und formal spezifiziert.

Kapitel 4 beschreibt die Verwendung der Wissensrepräsentationssprache CLASSIC und gibt eine anwendungsspezifische Einführung anhand von Beispielen, wozu auch LISP-Code präsentiert wird, da dem Leser so eine konkretere Vorstellung der CLASSIC-Sprache vermittelt werden kann (u.a. der Syntax).

Kapitel 5 bringt eine kurze Vorstellung von CLIM, ebenfalls anhand von Beispielen. Dem Leser soll u.a. auch die Eignung der Sprache LISP für die Oberflächenprogrammierung verdeutlicht werden.

Kapitel 6, 7 und 8 beschreiben Entwurfsentscheidungen, Handhabung und Implementationsaspekte von GENED. Bedauerlicherweise können nur einige sehr zentrale Punkte ausführlicher dargestellt werden.

1.5 Danksagungen

In erster Linie danke ich Dr. Volker Haarslev für seine ständige Gesprächsbereitschaft und Unterstützung sowie unzählige Tips bei der Lösung diverser Pobleme und das Makro `defqualifiedsubrole`.

Ralf Möller gab mir einen CLIM-Crashkursus. Durch diverse Gespräche verdanke ich ihm ein teilweise stark revidiertes Bild der Sprache LISP. Auch er half bei diversen Implementierungs- und Verständnisproblemen, insbesondere bei der Partitionierung des Systems.

Dietrich Fahrenholz danke ich für die Vorführung seiner Diplomarbeit, ein Exemplar seiner Studienarbeit sowie viele Tips im Umgang mit dem UNIX-System.

Eberhard Mattes danke ich für sein hervorragendes emTEX, mit dem diese Arbeit gesetzt wurde, und Georg Horn für sein zwar etwas eigenwillig zu benutzendes, aber praktisches TEXcad-Programm. Daniel S. Baker danke ich für sein Photo Lab-Programm, welches das Dithering der Bilder vornahm. Skaliert wurden die Bilder vom Graphic Workshop.

Allen Mitarbeitern des Arbeitsbereiches „Kognitive Systeme“ danke ich für die Duldung meiner resourcenintensiven Prozesse auf den „Sparc 10“-Maschinen.

2 Die Idee eines konzeptorientierten generischen Grafikeditors

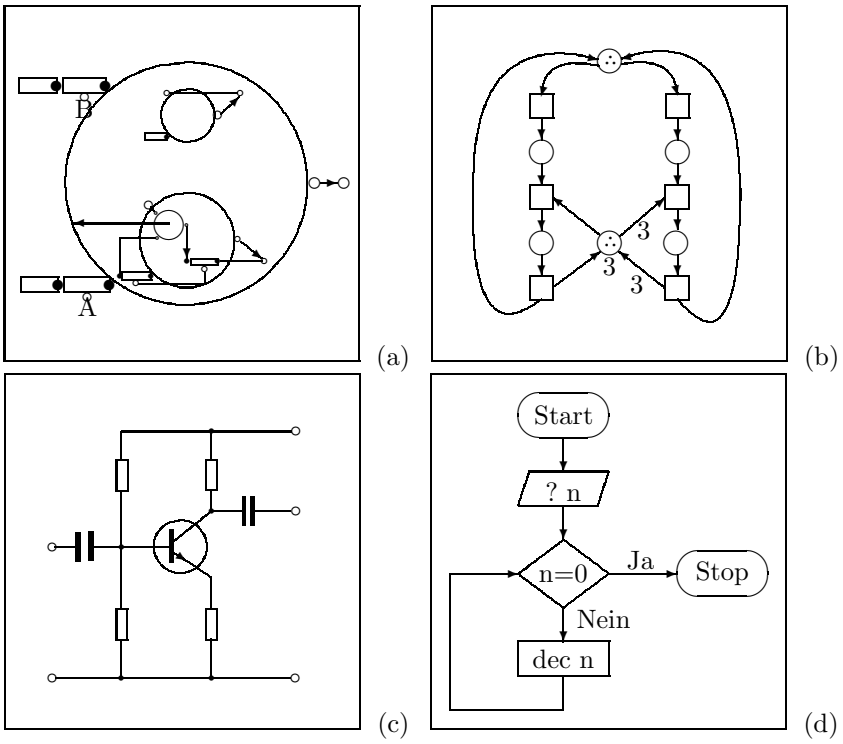


Abbildung 1: (a) PJ-Programm, (b) Petrinetz, (c) Schaltplan und (d) Flußdiagramm

2.1 Motivation

Dieses Kapitel stellt die Idee und Schlüsselkonzepte der Arbeit vor. Es werden bereits hier einige Begriffe zur Sprache gebracht, die in späteren Kapiteln präzisiert werden.

2.2 Visuelle Programmiersprachen, Netze und Piktogramme

Der generische Grafikeditor soll u.a. als konfigurierbares Werkzeug zur Erzeugung von Programmen für visuelle Programmiersprachen (VPs) dienen. Die Programme einer solchen Programmiersprache bestehen aus Grafiken, denen eine Bedeutung (Semantik) zugeschrieben wird. Wie auch bei der konventionellen (textuellen) Programmierung müssen Programme f. VPs einen bestimmten *Wohlgeformtheitsbegriff* erfüllen, was auch für Schaltpläne und Petrinetze gilt. Stets müssen gewisse Bedingungen erfüllt sein. Während bei textuellen Programmiersprachen durch die Entscheidung des Wortproblemes der zugrundeliegenden (meist eingeschränkt kontextfreien) Grammatik der Programmiersprache verifiziert werden kann, ob ein Programm syntaktisch korrekt ist oder nicht, müssen zur Entscheidung der Wohlgeformtheit visueller Progamme bzw. Grafiken andere Formalismen verwenden werden. Da in einem textverarbeitendem Compiler diese Frage in der Syntaxanalyse-

phase vom Parser entschieden wird, spricht man in Analogie hierzu bei visuellen Programmiersprachen auch vom „visual parsing“ (s. [1]).

Um einen Eindruck der Erscheinungsform von Programmen visueller Sprachen zu vermitteln, zeigt Abb. 1 ein „Pictorial Janus“-Programm (a) (APPEND, s. [3]), ein Petrinetz (b), einen Schaltplan (c) und ein Flußdiagramm (d). Auch wenn es sich ausschließlich bei dem APPEND-Programm um ein visuelles Programm handelt, so fallen doch einige Gemeinsamkeiten zwischen diesen Abbildungen ins Auge, die als universell und allgemeingültig für technische Grafiken postuliert werden:

- Grafische Elemente stehen mit anderen grafischen Elementen in bestimmten *Beziehungen* und geben der Grafik somit ihre Struktur. Welcher Art einige dieser Beziehungen sind und wie sie formalisiert werden können, wird später erörtert.
- Es scheint eine gewisse Anzahl *grafischer Primitive* zu existieren, aus denen die Grafik komponiert ist: Kreis, Strecke, Pfeil, Rechteck, Text, etc. Sie werden primitive Elemente/Objekte gennant. Bei textuellen Sprachen spielen die *Terminale* der zugrundeliegenden Grammatik diese Rolle.

2.3 Anforderungen an einen generischen Grafikeditor

Es stellt sich die Frage, wie ein Werkzeug zur Erstellung von visuellen Programmen, Petrinetzen oder Schaltplänen den Benutzer *unterstützen* sollte: Betrachtet man existierende Texteditoren in integrierten Systemen für z.B. die Pascal-Programmierung unter Windows, so fällt auf, daß Schlüsselwörter farblich hervorgehoben und die Programme automatisch korrekt eingerückt werden. Ob ein Progamm ein Wort der der Programmiersprache zugrundeliegenden Grammatik ist, kann nach einer syntaktischen Analyse entschieden werden. Im Falle eines Syntaxfehlers springt der Texteditor dann gleich an die fehlerhafte Stelle, woraufhin der Programmierer Korrekturen vornehmen kann. Eine vergleichbare Funktionalität würde sich der Benutzer auch von einem generischen Grafikeditor erhoffen - insbesondere sollten „Syntaxfehler“, also *fehlerhafte Konstellationen (Anordnungen) von Grafikelementen* erkannt und markiert werden. Automatische Layoutalgorithmen könnten dem Benutzer in Analogie zu einem einrückenden Texteditor Formatierungsarbeit abnehmen - doch ohne sehr spezielles und umfangreiches Domänenwissen bezüglich ästhetischer und funktioneller Aspekte solcher Grafiken können keine ansprechenden Formatierungen erzeugt werden. Eine alle Einsatzgebiete befriedigende, mit vertretbarem Aufwand implementierbare generische Lösung ist nicht in Sicht, weswegen dem Benutzer hier ein Layout von Hand zugemutet werden muß.

Ein System zur Erstellung von Grafiken wie die der Abb. 1 sollte die Möglichkeit bieten, mit Hilfe einiger Grundbausteine (grafischer Primitive) ein komplexes Bild zu erstellen. Damit das Werkzeug den Benutzer über fehlerhafte Konstellationen von Grafikelementen (bezogen auf die Einsatzdomäne) informieren kann, muß im System eine Komponente vorgesehen werden, die bestimmte *Konstellationen* grafischer Elementen erkennen kann. Diesen Dienst leistet ein *Klassifizierer*. Mit Unterstützung eines Klassifizierers kann u.a. eine *automatische Verifikation* gewisser Aspekte (z.B. der statischen Semantik) eines visuellen Programmes erreicht werden (s. [1]).

Die hier skizzierte Art der Unterstützung setzt ohne Frage die Existenz von geeignet repräsentiertem *Wissen* bezüglich des speziellen Einsatzgebietes im Grafikeditor voraus. Sicherlich benötigt ein Petrinetz-Grafikeditor anderes Wissen als ein „Pictorial Janus“-Grafikeditor.

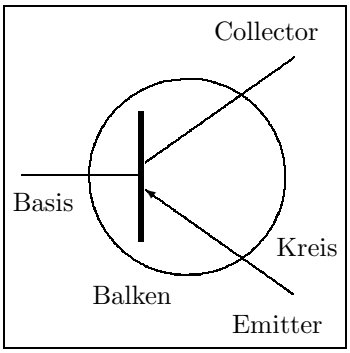


Abbildung 2: Transistorschaltzeichen

Als konkretes Beispiel zur Verdeutlichung soll ein Transistorschaltzeichen in einem Schaltplan dienen (Abb. 2)[3]: Eine derartige *Konstellation* sollte zum Transitorschaltzeichen klassifiziert werden können, wenn im System Wissen vorhanden ist, wie die visuelle Repräsentation eines Transistors in einem Schaltplan auszusehen hat. Ein solches Gebilde muß, damit es zum Transitorschaltzeichen klassifiziert wird, alle die für ein Transistorschaltzeichen *charakteristischen Eigenschaften* erfüllen. Dies wiederum setzt eine *formalisierte Beschreibungsform* für grafische Konstellationen voraus, so daß die Beziehungen und Eigenschaften der Elemente erkannt werden und als Anhaltspunkte für die Klassifizierung dienen können. Eine Schlüsselidee ist die Integration einer *Wissensbasis.* Sie enthält entsprechend kodiertes Wissen. Da der Editor *generisch* verwendbar sein soll, muß die Wissensbasis zudem austauschbar sein. Im folgenden wird das Wort Transistor im Sinne von Transistorschaltzeichen verwendet.

Wünschenswert ist auch eine Art „visueller Bibliothek“: Möchte man den Grafikeditor für die Erstellung von Schaltplänen verwenden, so wäre es äußerst lästig, jeden Transistor durch eine Folge von drei oder vier Handlungen erzeugen zu müssen. Außerdem würde jeder Transistor u.U. ein wenig verschieden aussehen, wodurch der Schaltplan ein unordentliches Aussehen erhielte. Eventuell hätte dies auch eine Beeinträchtigung des Klassifizierers zur Folge - je nachdem, wie groß die Abweichungen von Transistor zu Transistor sein dürften. Stattdessen möchte man die einmal erzeugte *Visualisierung des Konzeptes Transistor* in einer *Bibliothek* abspeichern und bei Bedarf beliebig oft mit einer einzigen Handlung erzeugen.

Es ist somit notwendig, bestimmte Grundelemente zu einem Ganzen zusammenzufassen, bzw. eine *Aggregation auf Objektebene* durchzuführen. Ein Transistor könnte als zusammengesetztes Objekte beschrieben werden, dessen Komponenten ein Kreis, drei Strecken und ein Pfeil sind, die wiederum in bestimmten Beziehungen zueinander stehen. Im weiteren Verlauf werden diese Objekte als aggregierte Objekte oder Kompositionsobjekte bezeichnet. Letztendlich *abstrahiert* der Benutzer von den einzelnen Komponenten: er sieht und benennt nur noch das gesamte Gebilde. So wird ein Gebilde wie ein Transistor-Schaltzeichen auch als Piktogramm bezeichnet - hierbei handelt es sich um ein *grafisches Symbol*, also ein Zeichen mit international festgelegter Semantik. Operationen wie z.B. das Verschieben eines Transistors auf der Arbeitsfläche des Grafikeditors werden ebenfalls durch Aggregation ermöglicht: statt jede Komponente einzeln zu verschieben wird das gesamte Aggregat neu pla-

[3]Tatsächlich sieht das Schaltzeichen etwas anders aus (bzgl. der Position der Pfeilspitze) - der Einfachheit halber wird jedoch diese Visualisierung diskutiert.

ziert. Das System stellt dem Benutzer somit eine seiner Sichtweise des Objektes entsprechende Interaktionsform bereit.

2.4 FOL als Beschreibungsform für grafische Konstellationen

FOL (First Order Logic, Prädikatenlogik 1. Stufe) kann verwendet werden, um die Beziehungen (Relationen) und Eigenschaften der Grafikelemente zu beschreiben. Umgangssprachlich läßt sich das Element Transistor so beschreiben (im Gegensatz zum oben gesagten wird ein Transistor hier nicht als aggregiertes Objekt definiert):

- Ein Transistor ist ein Kreis.
- Dieser Kreis enthält in seiner linken Hälfte eine senkrechte Strecke (Balken).
- Der Abstand der Strecke (Balken) zum Kreis von oben und unten ist gleich.
- In der Mitte der Strecke (Balken) berührt von der linken Seite eine waagerechte Strecke (Basis); sie schneidet den Kreis.
- Im Winkel von 50 Grad zur Strecke (Balken) berührt in der oberen Hälfte der Strecke (Balken) eine Strecke (Collector); sie schneidet den Kreis.
- Im Winkel von 130 Grad zur Strecke (Balken) berührt in der unteren Hälfte der Strecke (Balken) ein Pfeil (Emitter); er schneidet den Kreis, der Pfeilkopf ist innerhalb des Kreises.

Eine formale Definition erfolgt in FOL:

$$
\begin{aligned}
transistor(x) \quad \Leftarrow \quad & circle(x) \wedge \\
& (\exists_{=1}\, balken\, \exists_{=1}\, basis\, \exists_{=1}\, emitter\, \exists_{=1}\, collector\, : \\
& \quad line\_segment(balken) \wedge line\_segment(basis) \wedge arrow(emitter) \wedge \\
& \quad line\_segment(collector) \wedge \\
& \quad contains(x, balken) \wedge vertically(balken) \wedge \\
& \qquad (x(center\_position(balken)) < x(center\_position(x))) \wedge \\
& \qquad (y(center\_position(balken)) = y(center\_position(x))) \wedge \\
& \quad intersects(basis, x) \wedge touches(basis, balken) \\
& \qquad (angle(basis, balken) = 90) \wedge \\
& \qquad (x(center\_position(basis)) < x(center\_position(balken))) \wedge \\
& \quad intersects(collector, x) \wedge touches(collector, balken) \wedge \\
& \qquad (angle(balken, collector) = 50) \wedge \\
& \qquad (y(center\_position(balken)) > y(touch\_point(collector, balken))) \\
& \quad intersects(emitter, x) \wedge touches(emitter, balken) \wedge \\
& \qquad (angle(balken, emitter) = 130) \wedge \\
& \qquad (y(center\_position(balken)) < y(touch\_point(emitter, balken))) \wedge \\
& \qquad (contains(x, endpoint\_of(emitter))))
\end{aligned}
$$

Dabei ist *transistor* nun als unäres Prädikat definiert, wobei folgende Basisprädikate verwendet wurden:

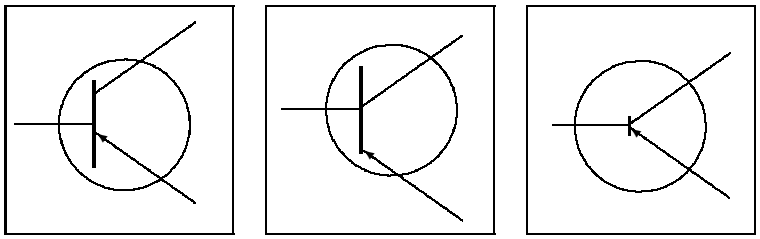


Abbildung 3: Ebenfalls Transistoren im Sinne der Definition

- $contains(A, B) \Leftrightarrow$ „Objekt A enthält Objekt B“
- $touches(A, B) \Leftrightarrow$ „Objekt A berührt Objekt B“
- $intersects(A, B) \Leftrightarrow$ „Objekt A schneidet Objekt B (überlappt oder kreuzt)“
- $vertically(A) \Leftrightarrow$ „Objekt A hat senkrechte Lage bzgl. des Bildschirmes“

Es wurden folgende Funktionen verwendet:

- $center\_position(A) = (x_c, y_c) \equiv$ „Der Mittelpunkt (x_c, y_c) des Objektes A“
- $x((x_p, y_p)) = x_p \equiv$ „Die X-Koordinate des Paares (x_p, y_p)“
- $y((x_p, y_p)) = y_p \equiv$ „Die Y-Koordinate des Paares (x_p, y_p)“
- $angle(A, B) \equiv$ „Der Winkel zwischen Objekt A und B (im Uhrzeigersinn), wenn beide eindimensional sind“
- $touch\_point(A, B) = (x_t, y_t) \equiv$ „Der Punkt, an dem sich A und B berühren (wenn eindeutig)“
- $endpoint\_of(A) = (x_e, y_e) \equiv$ „Der Endpunkt des Pfeiles A“

Ein x ist daher ein Transistor, wenn die genannten Bedingungen (Prämissen) dieses Prädikates erfüllt sind. Eventuell kann ein Transistor auch ganz anders aussehen - wird jedoch eine grafische Konstellation dieser Art in einem Schaltplan entdeckt, so handelt es sich sicher um einen Transistor, und $transistor(x)$ wird wahr. Der Umkehrschluß gilt jedoch nicht. Relationen wie *contains*, *touches*, *intersects* scheinen eine wichtige Rolle bei der Beschreibung zu spielen. Da sie paarweise Beziehungen zwischen Grafikelementen in der Zahlenebene $\mathbb{R}^2$ ausdrücken, werden sie *topologische Relationen* genannt. Sie werden in Kap. 3 ausführlicher diskutiert.

Anhand von Abb. 3 wird deutlich, daß die hier diskutierte *Definition* eines Transistors bereits gewisse *Abstraktionen* vornimmt: Für jedes dieser Gebilde wird das Prädikat *transistor* wahr, und somit handelt es sich ebenfalls um Transistoren. Zudem ist diese Definition bereits so kompliziert, daß ihre Übersetzung in Sprachen wie CLASSIC erhebliche Formulierungsschwierigkeiten bereiten würde. Gleichzeitig verdeutlicht sie aber auch die *Ausdruckskraft* von FOL.

Um in einem Schaltplan einen Transistor erkennen zu können, ist nun eine geeignete Repräsentation obiger FOL-Formel (oder einer ähnlichen) im System erforderlich. Aufgabe des Systems ist es dann, von den beobachteten *Prämissen* für ein Objekt x auf der Editorarbeitsfläche auf das Prädikat $transistor(x)$ zu schließen. Dies ist

im allgemeinen Aufgabe eines *prädikatenlogischen Theorembeweisers.* Letztendlich ist FOL jedoch viel zu mächtig und daher in ihrer Komplexität auf einem Rechner nicht beherrschbar. Das Problem, ob eine Aussage aus einer anderen logisch folgt, ist außerdem *unentscheidbar* (s. [4, S. 23]), wodurch die korrekte Klassifizierung eines Transistors gefährdet werden könnte. Eine geeignetere Repäsentation geschieht daher mit Hilfe einer Wissensrepräsentationssprache, in diesem Fall CLASSIC (s. [8]).

In der Terminologie der Wissensrepräsentation werden unäre Prädikate wie $transistor(x)$, die wie folgt definiert sind, bezeichnet als:

$$transistor(x) \Leftrightarrow .... \; Konzept$$
$$transistor(x) \Rightarrow .... \; primitives \; Konzept$$
$$transistor(x) \Leftarrow ....$$

> Die letztgenannte Form der Definition gibt es zumindest in CLASSIC nicht: Stattdessen müßte man mit $\Leftrightarrow$ definieren (allerdings gibt es *Regeln*).

Es handelt sich hierbei um sog. *Konzeptdefinitionen.* Sicherlich kann es nicht Aufgabe des Wissensrepräsentationssystems sein, anhand der grafischen Konstellation auf dem Editorschirm eine geeignete Beschreibung dieser für sich selbst zu erzeugen, auf der dann der Klassifizierer arbeiten kann - somit muß im Editor eine Art Geometriemodul vorgesehen werden, welches die benötigten topologischen Relationen errechnen und diese Informationen *zusichern* kann.

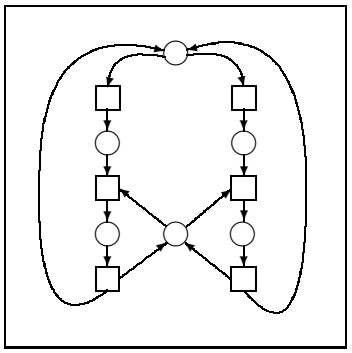
Eine Menge derartiger Konzeptdefinitionen macht nun im Sinne der Wissensrepräsentation eine *Wissensbasis* aus. Sie ermöglicht es, *Schlüsse (Inferenzen)* über die Anwendungsdomäne zu ziehen. Wie in nahezu allen Bereichen der Informatik hat auch hier die „richtige“ Wahl der Repräsentation einen entscheidenden Einfluß: Offensichtlich wird durch die hier diskutierte Konzeptdefinition ein Kreis zum Transistor klassifiziert, was eventuell nicht in jedem Fall adäquat ist, da die Objekte, mit denen der Kreis in Beziehung steht (drei Strecken und ein Pfeil) von der Klassifizierung nicht betroffen sind. Sie werden nicht zum Transistor klassifiziert - der Kreis ist somit das Zentrum der Betrachtung, sozusagen ein Stellvertreter für das gesamte Piktogramm. Eventuell sollen aber alle Komponenten zum Transistor oder Komponente-von-Transistor klassifiziert werden: Es handelt sich hierbei um ein typisches *Aggregationsproblem,* wie es in analoger Form auch in anderen Kontexten auftritt. Unterschiedliche Modellierungen führen hier zu unterschiedlichen Klassifizierungen. (Konzeptbezeichner werden ab nun in dieser Schriftart gesetzt.)

2.4.1 Beispiel: Petrinetze

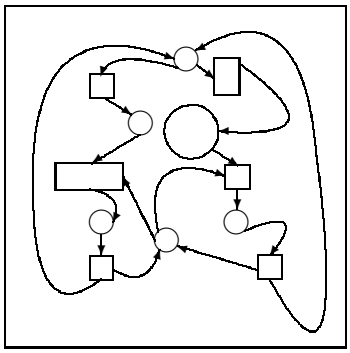
Als weiteres Beispiel, wie mit Hilfe von FOL grafische Konstellationen formal definiert und beschrieben werden können, sollen Petrinetze dienen. Sie sind wie folgt definiert (s. [10, S. 9]):

Definition 1 (Netz) *Ein Netz ist ein Tripel N = (S,T,F), bestehend aus einer Menge S von Stellen, man sagt auch Plätzen, einer dazu disjunkten nichtleeren Menge T von Transitionen und einer Flußrelation* $F \subset (S \times T) \cup (T \times S)$.

Transitionen sind somit stets über Netzkanten mit Stellen verbunden, und andersrum. Stellen werden durch Kreise visualisiert, Transitionen durch Rechtecke. Die Paare der Flußrelation sind durch Pfeile dargestellt, s. Abb. 1.



(a)



(b)

Abbildung 4: Ein oder zwei Petrinetze?

Wie könnte nun eine FOL-Wissensbasis aussehen, um Rechtecke zu Transitionen, Kreise zu Stellen, sowie Pfeile zu Netzkanten klassifizieren zu können? Es werden folgende Konzepte definiert:

$$
\begin{aligned}
stelle(x) \quad &\Leftrightarrow \quad circle(x) \wedge \\
&\quad (\neg\exists\, y : (intersects(x,y) \vee contained\_by(x,y) \vee contains(x,y))) \wedge \\
&\quad (\forall\, y : touches(x,y) \Rightarrow netzkante(y)) \wedge \\
&\quad (\forall\, y : linked\_over\_with(x,y) \Rightarrow transition(y))
\end{aligned}
$$

$$
\begin{aligned}
transition(x) \quad &\Leftrightarrow \quad rectangle(x) \wedge \\
&\quad (\neg\exists\, y : (intersects(x,y) \vee contained\_by(x,y) \vee contains(x,y))) \wedge \\
&\quad (\forall\, y : touches(x,y) \Rightarrow netzkante(y)) \wedge \\
&\quad (\forall\, y : linked\_over\_with(x,y) \Rightarrow stelle(y))
\end{aligned}
$$

$$
\begin{aligned}
netzkante(x) \quad &\Leftrightarrow \quad (arrow(x) \vee directed\_spline\_chain(x)) \wedge \\
&\quad (\forall\, y : touches(x,y) \Rightarrow (stelle(y) \vee transition(y)))
\end{aligned}
$$

Werden alle Komponenten eines Petrinetzes nun als Ganzes betrachtet, also als Petrinetz selbst, dann führt diese Aggregation zu folgender Definition: ein Petrinetz soll - als Ganzes betrachtet - ein zusammengesetztes Objekt sein, dessen Komponenten ausschließlich Transitionen, Stellen und Netzkanten sind. Mit Hilfe der *has_part*-Relation läßt sich diese Abstraktion so formulieren:

$$
\begin{aligned}
petrinetz(x) \quad &\Leftrightarrow \quad composite\_thing(x) \wedge \\
&\quad (\forall\, y : has\_part(x,y) \Rightarrow (stelle(y) \vee transition(y) \vee netzkante(y)))
\end{aligned}
$$

Mit Hilfe dieser Wissenbasis ist nun eine unendliche Menge von Netzen beschrieben - allerdings würde nicht jedes Modell bzw. Petrinetz im Sinne dieser Formelmenge von einem menschlichen Betrachter ebenfalls als Petrinetz klassifiziert werden (s.u.). Diese Beschreibung abstrahiert von Objekteigenschaften wie Position, Farbe

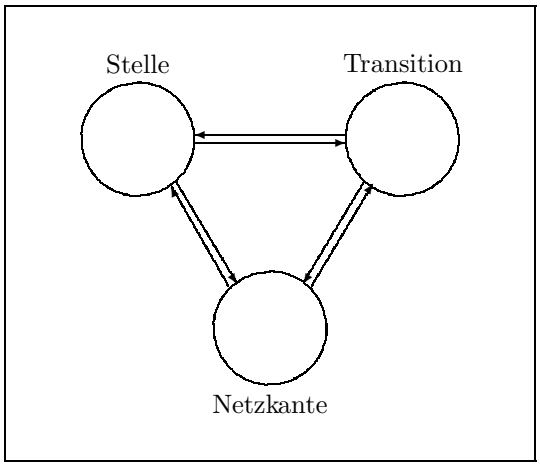


Abbildung 5: Zyklische Definitionen

und Abmessungen. Ein Kreis ist ein Kreis, unabhängig von seiner Lage und Größe. Dies sei anhand von Abb. 4 verdeutlicht: Wenn die Objekte in (a) und (b) paarweise gleiche Namen und „Typen“ haben, so werden die entsp. Prädikate für Abb. (a) genau dann wahr, wenn sie auch für Abb. (b) wahr werden. Sie bilden dann die gleiche Belegung bzw. Interpretation obiger FOL-Formeln. Ein menschlicher Betrachter würde das Gebilde von Abb. (b) in der Regel nicht als Petrinetz bezeichnen - da in den üblichen Lehrbüchern zum Thema Petrinetze (wie [11], [10]) die Petrinetzdarstellungen selbst jedoch nicht formal definiert bzw. „normiert“ werden[4], kann für die hier vorgestellte Wissensbasis weder Vollständigkeit noch Korrektheit bewiesen werden. Anschaulich ist diese Wissensbasis jedoch vollständig, da alles, was normalerweise als Petrinetzdarstellung bezeichnet wird, daß Prädikat *petrinetz* auch wahr macht. Als intuitiv korrekt kann man diese Wissensbasis wohl nicht bezeichnen, da ein menschlicher Betrachter die in Abb. 4(b) dargestellte Konstellation normalerweise nicht als Petrinetz klassifizieren würde. Dies ist jedoch lediglich eine Frage der (wenn auch nicht formal beschriebenen) Konvention - prinzipiell könnten Stellen auch als Rechtecke und Transitionen als Kreise visualisiert werden.

Zudem fällt auf, daß die Wissensbasis *rekursiv* definiert ist. Zyklische Definitionen lassen sich jedoch in einer Wissensrepräsentationssprache wie CLASSIC nicht formulieren. Eine „Terminationsbedingung“ obiger Definitionen ist nicht ersichtlich - sie lassen sich beliebig oft *expandieren,* also stets ineinander einsetzen, wodurch eine nichteindeutige unendliche Menge entsteht (Fixpunkte). Ein Abhängigkeitsgraph verdeutlicht diese Zyklen (Abb. 5). Eine in CLASSIC formulierbare Petrinetz-Wissensbasis wird in Kap. 4 vorgestellt.

[4]Schließlich stehen hier die Modelle bzw. mathematischen Strukturen selbst und nicht deren Visualisierungen im Vordergrund der Betrachtung.

3 Topologische Relationen

3.1 Motivation

Beim Definitionsversuch des Transistorpiktogrammes in Kap. 2 wurden Relationen wir $touches, intersects, contains$ etc. verwendet. Sie werden als topologische Relationen bezeichnet und drücken paarweise Beziehungen zwischen Grafikelementen in der Zahlenebene $\mathbb{R}^2$ aus (s. [4, Kap. 1.4.3]). Eine interessante Eigenschaft dieser Relationen ist Invarianz bezüglich affiner Transformationen (wie Skalierung, Translation und Rotation).

Topologische Relationen scheinen geeignet, gewisse Abstraktionen von quantitativen Eigenschaften wie Position und Größe der Grafikelemente vorzunehmen, um qualitative Aussagen in den Vordergrund der Betrachtung zu stellen. Bei VPs wie „Pictorial Janus“ scheint dies genau die benötigte Art der Abstraktion zu sein (s. [1]), da hier u.a. die Form der Grafikobjekte irrelevant ist (sie müssen lediglich konvex sein). So kann z.B. das APPEND-Programm (s. Abb. 1(a)) auch mit Rechtecken statt mit Kreisen visualisiert werden. Relevant sind hingegen die topologischen Relationen der Objekte untereinander.

Es wird nun zunächst die notwendige Theorie beleuchtet, um dann Implementationsaspekte zu betrachten. Schließlich muß eine Art Geometriemodul im Grafikeditor vorgesehen werden, welches die Relationen errechnen kann. Vorausgeschickt sei, daß die vom Geometriemodul von GENED berechneten Relationen in vielen Details stark von der im folgenden dargestellten Theorie abweichen. Es wurde versucht, mit angemessenem Aufwand eine generische Implementation für verschiedene Objektarten zu schaffen.

3.2 Punktemengen

Die Definition der topologischen Relationen geht auf eine Publikation von Egenhofer zurück (s. [5]). Danach wird jedes Grafikobjekt wie in der Topologie als *Punktemenge* im $\mathbb{R}^2$ aufgefasst, notiert mit λ. Grafikobjekte müssen zusammenhängend sein und dürfen keine Löcher aufweisen. Es bezeichnet dann

- $\partial\lambda$ den Rand, und
- $\lambda^0 = \lambda - \partial\lambda$ das Innere des Objektes.

Für $\partial\lambda$ gilt:

- ∂ Punkt : $\emptyset$.
- ∂ Linie : $\emptyset$ für zirkuläre Linien, sonst die Menge der beiden Endpunkte $\{P_{start}, P_{end}\}$.
- ∂ Fläche : Die zirkuläre Linie, die den Rand beschreibt, also die Menge aller Randpunkte der Fläche.

Egenhofer definierte die topologischen Relationen ursprünglich nur für Flächen; andere erweiterten die Arbeit dann um Relationen für Linien und Punkte.[5] Die Funktion $\partial\lambda$ wurde dementsprechend definiert. Topologische Relationen lassen sich nun

[5] Der Begriff „Linie“ wird hier für eine endliche oder zyklische mathematische Kurve benutzt - Linien in diesem Sinne sind weder Strecken noch Geraden, können aber wie Strecken dann Start- und Endpunkt haben.

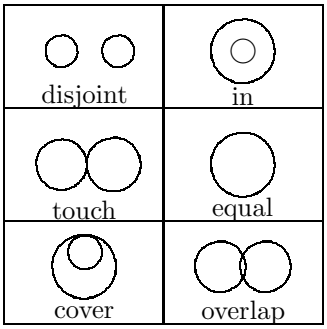


Abbildung 6: Topologische Relationen nach Egenhofer

nach einer bestimmten Anzahl von Fällen klassifizieren, wobei folgende Ausdrücke verwendet werden:

- $S_1 = \partial A_1 \cap \partial A_2$
- $S_2 = \partial A_1 \cap A_2^0$
- $S_3 = A_1^0 \cap \partial A_2$
- $S_4 = A_1^0 \cap A_2^0$

Da so ein Schnitt immer leer oder nichtleer sein kann, gibt es $2^4 = 16$ mögliche Kombinationen. Von diesen 16 bleiben jedoch nur 6 topologisch sinvolle bzw. mögliche Relationen:

$S_1 = \partial A_1 \cap \partial A_2$	$S_2 = \partial A_1 \cap A_2^0$	$S_3 = A_1^0 \cap \partial A_2$	$S_4 = A_1^0 \cap A_2^0$	Name
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	A_1 disjoint A_2
$\emptyset$	$\emptyset$	$\neg\emptyset$	$\neg\emptyset$	A_2 in A_1
$\emptyset$	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$	A_1 in A_2
$\neg\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	A_1 touch A_2
$\neg\emptyset$	$\emptyset$	$\emptyset$	$\neg\emptyset$	A_1 equal A_2
$\neg\emptyset$	$\emptyset$	$\neg\emptyset$	$\neg\emptyset$	A_1 cover A_2
$\neg\emptyset$	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$	A_2 cover A_1
$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$	A_1 overlap A_2

Die Tabelle ist also als eine genau-dann-wenn-Definition der Relationen zu verstehen. Abb. 6 veranschaulicht die 6 Fälle.

So bedeutet z.B. die Definition von $< A_1, cover, A_2 >$, daß sich die Ränder der Objekte A_1 und A_2 schneiden, der Schnitt des Randes des Objekte A_1 mit dem Inneren von Objekt A_2 leer ist, der Schnitt des Inneren des Objektes A_1 mit dem Rand des Objektes A_2 nicht leer ist, und daß der Schnitt des Innerern beider Objekte nicht leer ist.

Ein Nachteil dieser Methode ist jedoch, daß Fälle, die für den Benutzer verschieden aussehen, anhand dieser Klassifizierung nicht unterschieden werden: so wird z.B. die Dimension der Schnitte nicht berücksichtigt. Die Egenhofer-Methode unterscheidet somit nicht, ob z.B. ein Schnitt der Ränder zweier Rechtecke eindimensional oder

nulldimensional ist (in jedem Fall gilt $touches(A, B)$, doch im ersten Falle handelt es sich um eine Berührung zweier Kanten, im zweiten um eine Berührung zweier Eckpunkte). Clementini et al. (s. [6]) erweiterten die Egenhofer-Methode und nahmen zur Unterscheidung der Fälle noch die Dimension der Schnitte hinzu, wodurch sich dann insgesamt 52 Fälle unterscheiden lassen. Die Dimension der Schnitte wird dabei wie folgt definiert:

$$dim(S) = \begin{cases} - & \text{wenn } S = \emptyset \\ 0 & \text{wenn } S \text{ mindestens einen Punkt und keine Linien oder Flächen enthält} \\ 1 & \text{wenn } S \text{ mindestens eine Linie und keine Flächen enthält} \\ 2 & \text{wenn } S \text{ mindestens eine Fläche enthält} \end{cases}$$

Diese Methode wird „The Dimension Extended Method“ genannt. Sie hat jedoch den Nachteil, daß 52 unterschiedliche Fälle mit verschiedenen Bezeichnungen für den Benutzer schlicht zu viele sind - er kann sie sich nicht merken. Daher definierten Clementini et al. die sog. „Calculus Based Method“, die sich im Gegensatz zu ersterer dadurch auszeichnet, daß die Relationen *überladen* (also *generisch*) sind, da sie sich sowohl auf Punkte als auch Linien und Flächen anwenden lassen, ohne jeweils neue Relationen zu vergeben (wie *touch_point*, *touch_line*, ...). Die Autoren zeigten zudem, daß beide Methoden gleichmächtig sind, die Fälle sich wechselseitig ausschließen und alle möglichen Fälle (wie auch bei der „Dimension Extended Method“) berücksichtigt werden, was als *Vollständigkeit* bezeichnet werden kann. An dieser Stelle sei ihre (übersetzte) Definition zitiert:

Definition 2 *Die* **Berührt**-*Relation (touch) kann angewendet werden auf Situationen der Art Fläche/Fläche, Linie/Linie, Linie/Fläche, Punkt/Fläche, Punkt/Linie, aber nicht der Art Punkt/Punkt:*

$$< \lambda_1, touch, \lambda_2 > \Leftrightarrow (\lambda_1^0 \cap \lambda_2^0 = \emptyset) \wedge (\lambda_1 \cap \lambda_2 \neq \emptyset)$$

Definition 3 *Die* **Enthält**-*Relation (in) kann auf jede Situation angewendet werden:*

$$< \lambda_1, in, \lambda_2 > \Leftrightarrow (\lambda_1 \cap \lambda_2 = \lambda_1) \wedge (\lambda_1^0 \cap \lambda_2^0 \neq \emptyset)$$

Definition 4 *Die* **Kreuzt**-*Relation (cross) kann angewendet werden auf Linie/Linie und Linie/Fläche-Situationen:*

$$< \lambda_1, cross, \lambda_2 > \Leftrightarrow dim(\lambda_1^0 \cap \lambda_2^0) = (max(dim(\lambda_1^0), dim(\lambda_2^0)) - 1) \wedge$$
$$(\lambda_1 \cap \lambda_2 \neq \lambda_1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_2)$$

Definition 5 *Die* **Überlappt**-*Relation (overlap) kann angewendet werden auf Fläche/Fläche und Linie/Linie-Situationen:*

$$< \lambda_1, overlap, \lambda_2 > \Leftrightarrow (dim(\lambda_1^0) = dim(\lambda_2^0) = dim(\lambda_1^0 \cap \lambda_2^0)) \wedge$$
$$(\lambda_1 \cap \lambda_2 \neq \lambda_1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_2)$$

Definition 6 *Die* **Nichts-Gemeinsam**-*Relation (disjoint) kann auf jede Situation angewendet werden:*

$$< \lambda_1, disjoint, \lambda_2 > \Leftrightarrow \lambda_1 \cap \lambda_2 = \emptyset$$

Alle Relationen bis auf die **Enthält**-Relation sind

- symmetrisch (also $< \lambda_1, r, \lambda_2 > \Leftrightarrow < \lambda_2, r, \lambda_1 >$), die dafür als einzige
- transitiv ist (also $< \lambda_1, r, \lambda_2 > \wedge < \lambda_2, r, \lambda_3 > \Rightarrow < \lambda_1, r, \lambda_3 >$).

3.3 Implementationsbetrachtungen

Um topologischen Relationen nun für Konzeptdefinitionen wie in Kap. 2 benutzen zu können, müssen sie von einem Geometriemodul anhand einer geeigneten Repräsentation der grafischen Konstellation auf dem Editorschirm berechnet werden. Dabei wurde ein pragmatischer Weg eingeschlagen, da das Geometriemodul für die mathematischen Objekttypen Punkt, Strecke und Streckenpfad sowie beliebige Flächen funktionieren sollte und mit vertretbarem Aufwand implementiert werden mußte. Letztendlich läßt sich jedes Flächenobjekt (da es aus Pixeln zusammengesetzt auf dem Bildschirm erscheint und somit diskret ist) im Rechner als ein Polygon darstellen. Angestrebt wurde ein „What You See Is What You Get"(WYSIWYG): die vom Geometriemodul berechneten Relationen sollten möglichst mit den vom Benutzer auf dem Bildschirm beobachteten Relationen übereinstimmen. Einzig Textelemente werden nicht polygonisiert - stattdessen wird der Einfachheit halber das *kleinste umschließende Rechteck (Bounding Box)* genommen.

Aufgrund offensichtlicher Schwierigkeiten (überabzählbare Punktemengen) wurde nicht versucht, die topologischen Relationen direkt ihrer mathematischen Definition entsprechend nach Clementini oder Egenhofer zu implementieren. Daher decken sich die im Geometriemodul realisierten Relationen nicht mit den obigen Definitionen. Um den Benutzer jedoch nicht im Unklaren zu lassen, wird später eine genaue Spezifikation der Relationen in FOL gegeben.

Das Geometriemodul kennt grundsätzlich nur vier Objektarten:

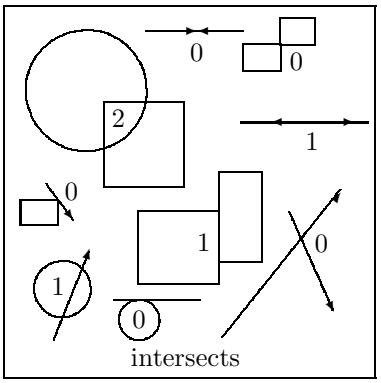
- **Punkte**, repräsentiert durch ein Gleitkommazahlen-Paar (Real).
- **Strecken**, repräsentiert durch Start- und Endpunkt.
- **Polygone**, repräsentiert durch ein Feld (Array) von Strecken (Polygonkanten), was als geschlossener Streckenpfad interpretiert wird.
- **Streckenpfade/ -ketten**, repräsentiert durch ein Feld von Strecken.

Dabei wurde das Geometriemodul objektorientiert in CLOS entworfen. Die Objektarten korrespondieren somit zu CLOS-Klassen - für jedes auf dem Editorschirm dargestellte Grafikobjekt muß eine für das Geometriemodul geeignete Repräsentation erzeugt werden, also eine Instanzen einer dieser CLOS-Klassen. Die Typen Polygon und Streckenpfad sind als eine einzige Klasse implementiert, wobei ein Flag `closed` entscheidet, ob das Feld als offen oder geschlossen interpretiert wird.

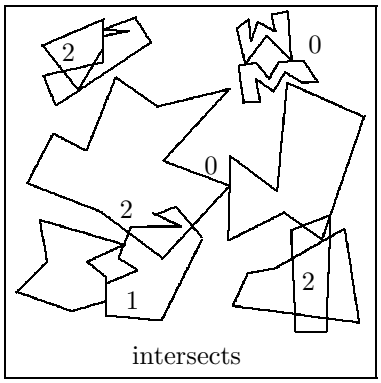
Das Modul implementiert folgende Relationen:

- *disjoint*
- *touches*
- *intersects*
- *contains* $\leftrightarrow$ *contained_by*
- *covers* $\leftrightarrow$ *covered_by*

Alle Relationen sind für alle Kombinationen der vier Objektarten definiert (total). Implementiert sind sie als *generische Funktionen* in CLOS. *intersects* bestimmt zudem die Dimension des Schnittes zweier Objekte. Sie bedarf einer Erläuterung:

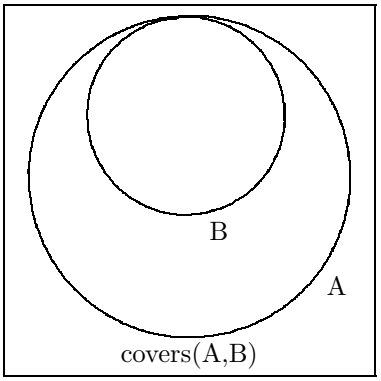


(a)

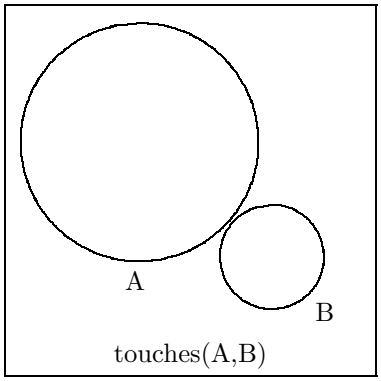


(b)

Abbildung 7: Schnitte und ihre Dimensionen



(a)



(b)

Abbildung 8: Die Überdeckt- und Berührt-Relation

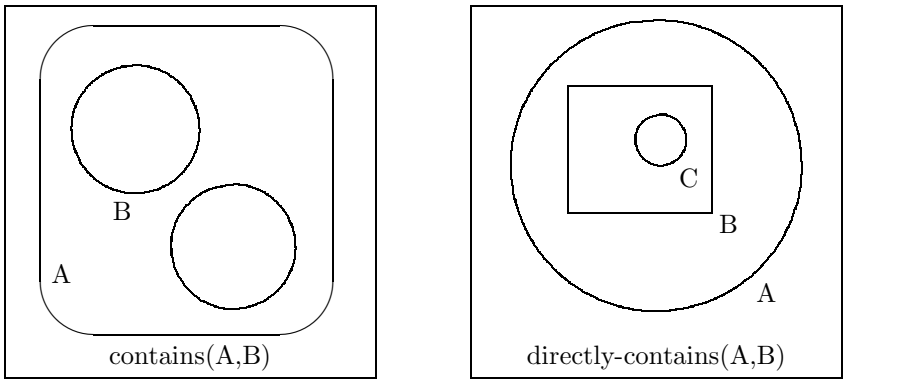


Abbildung 9: Die Enthält- und Enhält-Direkt-Relation

- Zwei **Punkte** können sich schneiden, wenn sie gleiche Koordinaten haben. Der Schnitt ist dann nulldimensional.
- Zwei **Strecken** oder **Streckenpfade** können sich schneiden, ihr Schnitt ist ein- oder nulldimensional.
- Zwei **Polygone** können sich schneiden, ihr Schnitt ist null-, ein- oder zweidimensional.

Abb. 7, 8 und 9 veranschaulichen die vom Geometriemodul berechneten Relationen. Insbesondere sind in Abb. 7 die Schnittdimensionen eingetragen.

3.3.1 Spezifikation der Relationen mit FOL

Objekte haben weder einen Rand noch ein Inneres (im Sinne von $\partial\lambda$ und λ^0), stattdessen werden sie wie oben diskutiert repräsentiert. Nun bezeichnet

- $\mathfrak{Edges}(X)$ die Menge aller **Strecken** (Kanten) des Objektes X, wobei X vom Typ **Polygon** oder **Streckenpfad** ist, und
- $Start(L)$ bzw. $End(L)$ den Start- bzw. Endpunkt der **Strecke** L.

Für die folgenden Formeln werden Typbezeichnungen der Objekte verwendet:

- P für **Punkte** (Points),
- L für **Strecken** (Line Segments),
- A für **Polygone** (Areas),
- S für **Streckenpfade** (Line Segment Chains),
- X für **Streckenpfade und Polygone** (als gemeinsamen Obertyp).

Der Ausdruck $type(A, B) = (X, Y)$ wird genau dann wahr, wenn A vom Typ X ist und B vom Typ Y ($X, Y \in \{P, L, A, S, X\}$).

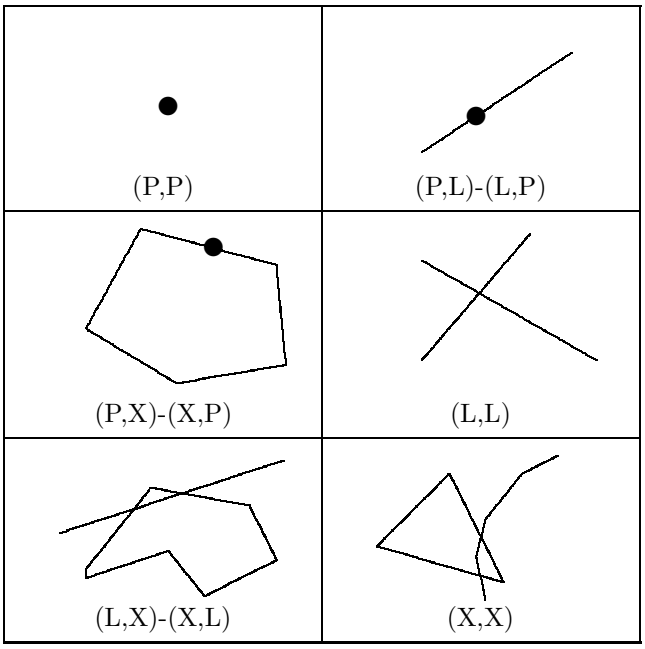


Abbildung 10: Die Fälle der *intersects*-Relation

Das rekursive Prädikat *intersects* entscheidet nun, ob zwei Objekte A, B sich schneiden. Wie auch die folgenden Prädikate verwendet es Basisprädikate (implementiert als Basisalgorithmen), die später erläutert werden. Alle diskutierten Prädikate sind total auf $\{P, L, A, S, X\}$. Daher wird *intersects* für solche Argumente stets terminieren, wenn es von oben nach unten ausgewertet wird:

$$
\begin{array}{lll}
intersects(A,B) & \Leftrightarrow & (type(A,B) = (P,P) \quad \wedge \quad (A.x = B.x) \wedge (A.y = B.y) \\
& \vee & (type(A,B) = (P,L) \quad \wedge \quad lies\_on(A,B)) \\
& \vee & (type(A,B) = (P,X) \quad \wedge \quad \exists\, V \in \mathfrak{Edges}(B) : intersects(A,V)) \\
& \vee & (type(A,B) = (L,L) \quad \wedge \quad intersects\_line\_seg(A,B)) \\
& \vee & (type(A,B) = (L,X) \quad \wedge \quad \exists\, V \in \mathfrak{Edges}(B) : intersects(A,V)) \\
& \vee & (type(A,B) = (X,X) \quad \wedge \quad \exists\, V \in \mathfrak{Edges}(A) : intersects(V,B)) \\
& \vee & intersects(B,A)
\end{array}
$$

Folgende Prädikate ergeben die Dimension des Schnittes:

$$
\begin{array}{lll}
intersects\_dim\_0(A,B) & \Leftrightarrow & intersects(A,B) \wedge dim(A,B) = 0 \\
intersects\_dim\_1(A,B) & \Leftrightarrow & intersects(A,B) \wedge dim(A,B) = 1 \\
intersects\_dim\_2(A,B) & \Leftrightarrow & intersects(A,B) \wedge dim(A,B) = 2
\end{array}
$$

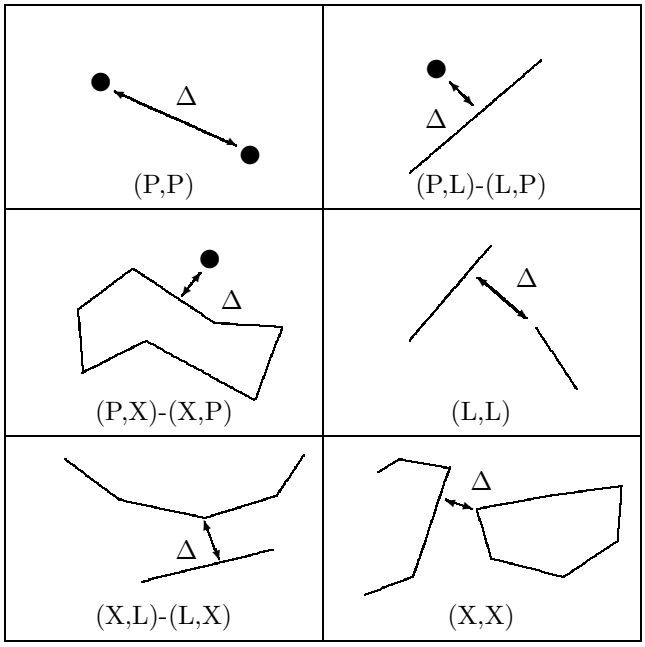


Abbildung 11: Die Fälle der Δ-Funktion

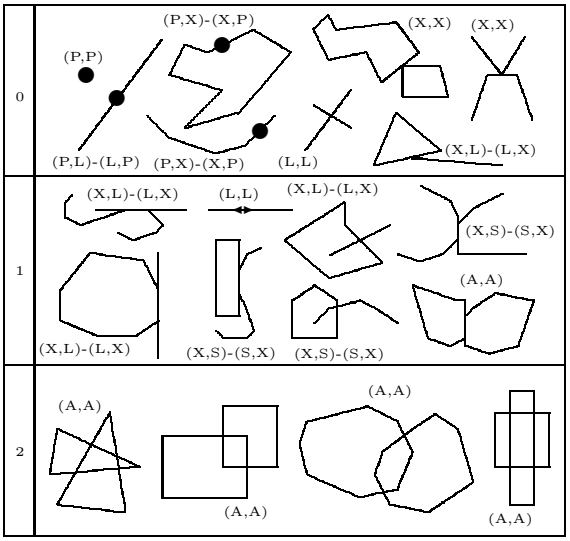


Abbildung 12: Verschanschaulichung der *dim*-Funktion

Dabei wird die Funktion *dim* verwendet, die wie folgt definiert ist (Visualisierung der einzelnen Fälle s. Abb. 12):

$$dim(A,B) = \begin{cases}
0 : & (type(A,B) = (P,P)) & & \\
\vee & (type(A,B) = (P,L)) & & \\
\vee & (type(A,B) = (L,P)) & & \\
\vee & (type(A,B) = (P,X)) & & \\
\vee & (type(A,B) = (X,P)) & & \\
\vee & (type(A,B) = (L,L)) & \wedge & \neg intersects_line_seg_dim_1(A,B)) \\
\vee & (type(A,B) = (L,X)) & \wedge & max(\forall\, V \in \mathfrak{Edges}(B) : dim(A,V)) = 0) \\
\vee & (type(A,B) = (X,L)) & \wedge & dim(B,A) = 0) \\
\vee & (type(A,B) = (X,X)) & \wedge & max(\forall\, V \in \mathfrak{Edges}(A) : dim(V,B)) = 0) \\
1 : & (type(A,B) = (L,L)) & \wedge & intersects_line_seg_dim_1(A,B)) \\
\vee & (type(A,B) = (L,X)) & \wedge & ((max(\forall\, V \in \mathfrak{Edges}(B) : dim(A,V)) = 1) \vee \\
 & & & \quad (type(B) = A \wedge one_part_inside(A,B)))) \\
\vee & (type(A,B) = (X,L)) & \wedge & dim(B,A) = 1) \\
\vee & (type(A,B) = (X,S)) & \wedge & \exists\, V \in \mathfrak{Edges}(B) : dim(V,A) = 1) \\
\vee & (type(A,B) = (S,X)) & \wedge & dim(B,A) = 1) \\
\vee & (type(A,B) = (A,A)) & \wedge & \forall\, V \in \mathfrak{Edges}(A) : \neg one_part_inside(V,B) \\
 & & \wedge & \exists\, V_1 \in \mathfrak{Edges}(A),\, \exists\, V_2 \in \mathfrak{Edges}(B) : dim(V_1,V_2) = 1) \\
2 : & (type(A,B) = (A,A)) & \wedge & \exists\, V \in \mathfrak{Edges}(A) : one_part_inside(V,B)
\end{cases}$$

Die generische Funktion Δ berechnet den kleinsten mathematischen Abstand zwischen zwei Objekten (s. Abb. 11). Ihre Definition ist:

$$\Delta(A,B) = \begin{cases}
d_{PP}(A,B) & : & type(A,B) = (P,P) \\
d_{PL}(A,B) & : & type(A,B) = (P,L) \\
\Delta(B,A) & : & type(A,B) = (L,P) \\
min(\forall\, V \in \mathfrak{Edges}(B) : \Delta(A,V)) & : & type(A,B) = (P,X) \\
\Delta(B,A) & : & type(A,B) = (X,P) \\
min(\Delta(Start(A),B), \Delta(End(A),B)) & : & type(A,B) = (L,L) \\
min(\forall\, V \in \mathfrak{Edges}(B) : \Delta(A,V)) & : & type(A,B) = (L,X) \\
\Delta(B,A) & : & type(A,B) = (X,L) \\
min(\forall\, V \in \mathfrak{Edges}(B) : \Delta(V,A)) & : & type(A,B) = (X,X)
\end{cases}$$

d_{PP} berechnet den euklidischen Abstand zweier Punkte (oder die Norm eines Vektors):

$$d_{PP}(P_1,P_2) = \sqrt{(P_1.x - P_2.x)^2 + (P_1.y - P_2.y)^2}$$

$d_{PL}(P,L)$ berechnet den minimalen Abstand eines Punktes P zur Strecke L. Dabei wird zuerst versucht, ein Lot von P auf L zu fällen. Gelingt dies (Strecken sind endlich!), so ist die Norm des Lotes der Wert der Funktion. Gelingt es nicht, das Lot zu fällen, so ist der Wert der Funktion das Minimum der euklidischen Abstände vom Punkt P zum Start- und Endpunkt der Strecke, also:

$$d_{PL}(P,L) = \begin{cases}
||\,\vec{lot}\,|| & : & 0 \le \lambda_p \le 1 \\
min(d_{PP}(P,Start(L)), d_{PP}(P,End(L))) & : & \text{sonst}
\end{cases}$$

$$\vec{a} = \vec{Start}(L) - \vec{End}(L)$$

$$\lambda_p = \frac{\vec{a}\,(\vec{P} - \vec{Start}(L))}{\vec{a}^{\,2}}$$

$$\vec{lot} = \vec{Start}(L) - \vec{P} + \lambda_p \vec{a}$$

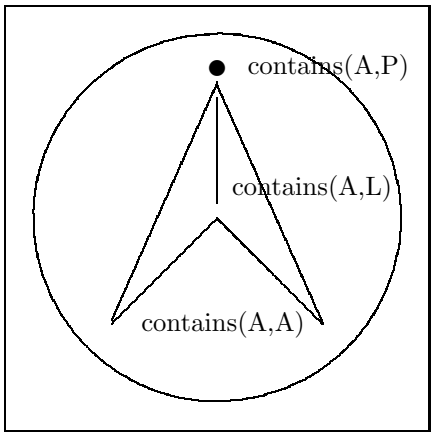


Abbildung 13: Die Fälle der *contains*-Relation

Die übrigen Relationen sind so definiert (s. Abb. 13):

$$
\begin{aligned}
contains(A,B) \quad \Leftrightarrow \quad & \neg intersects(A,B) \wedge \\
& (\quad (type(A,B) = (A,P) \quad \wedge \quad point\_inside\_polygon(B,A)) \\
& \quad \vee \quad (type(A,B) = (A,L) \quad \wedge \quad point\_inside\_polygon(Start(B),A) \\
& \qquad\qquad\qquad\qquad\quad \wedge \quad point\_inside\_polygon(End(B),A)) \\
& \quad \vee \quad (type(A,B) = (A,A) \quad \wedge \quad \forall\, V \in \mathfrak{Edges}(B) : contains(A,V)))
\end{aligned}
$$

$$
\begin{aligned}
touches(A,B) \quad \Leftrightarrow \quad & \neg intersects(A,B) \wedge \neg contains(A,B) \wedge \neg contains(B,A) \wedge \\
& \Delta(A,B) \leq \epsilon
\end{aligned}
$$

ϵ ist ein Begrenzungswert, der angibt, wie nahe (mathematisch) sich zwei Objekte kommen müssen, damit sie als berührend gelten.

$$
\begin{aligned}
covers(A,B) \quad \Leftrightarrow \quad & contains(A,B) \wedge \\
& \Delta(A,B) \leq \epsilon
\end{aligned}
$$

Die Relation *covers* ist also eine Teilmenge der Relation *contains*. Gilt keine der obigen Relationen, so gilt automatisch $disjoint(A,B)$. Auf die Kodierung einer $equal(A,B)$-Relation wurde verzichtet, da ihre Anwendung extrem beschränkt ist.

3.3.2 Basisalgorithmen

Zur Definition obiger Relationen wurden diverse Basisprädikate verwendet. Das Geometriemodul implementiert diese durch folgende Basisalgorithmen:

1. Der Algorithmus, der entscheidet, ob zwei Strecken sich schneiden (*intersects_line_seg*?).
2. Der Algorithmus, der entscheidet, ob ein Punkt auf einer Strecke liegt (*lies_on*?).
3. Der Algorithmus, der entscheidet, ob zwei Strecken einen eindimensionalen Schnitt bilden (*intersects_line_seg_dim_1*?).

4. Der Algorithmus, der entscheidet, ob ein Punkt in einem Polygon enthalten ist (*point_inside_polygon*?).

5. Der Algorithmus, der entscheidet, ob ein Teil einer Strecke (der durch rekursives Teilen der Strecke gewonnen wird) in einem Polygon liegt (*one_part_inside*?).

6. Die Abstandsfunktionen (d_{PP} und d_{PL}).

Eine Quelle für den **1. Algorithmus** ist z.B. [12, S. 402 - 404]: er bedient sich einer Funktion, die drei Punkte nimmt und anhand dieser feststellt, ob ein Weg vom ersten über den zweiten zum dritten Punkt mit dem Uhrzeigersinn oder gegen ihn beschritten wird. Diese Hilfsfunktion heißt *ccw*, sie wird nicht weiter erklärt. *ccw* ist dreiwertig:

$$ccw(p_0, p_1, p_2) = \begin{cases} -1 & \text{falls mit dem Uhrzeigersinn,} \\ & \text{oder falls } p_0 \text{ zwischen } p_2 \text{ und } p_1 \text{ liegt} \\ 0 & \text{falls } p_2 \text{ zwischen } p_0 \text{ und } p_1 \text{ liegt} \\ 1 & \text{falls gegen den Uhrzeigersinn,} \\ & \text{oder falls } p_1 \text{ zwischen } p_0 \text{ und } p_2 \text{ liegt} \end{cases}$$

Mit Hilfe von *ccw* wird nun das Basisprädikat *intersects_line_seg*? implementiert, Sedgewick schreibt:[6]

> „Falls die beiden Endpunkte einer Strecke sich auf verschiedenen Seiten der anderen Strecke befinden (versch. *ccw*-Werte haben), müssen sich die Strecken schneiden.“

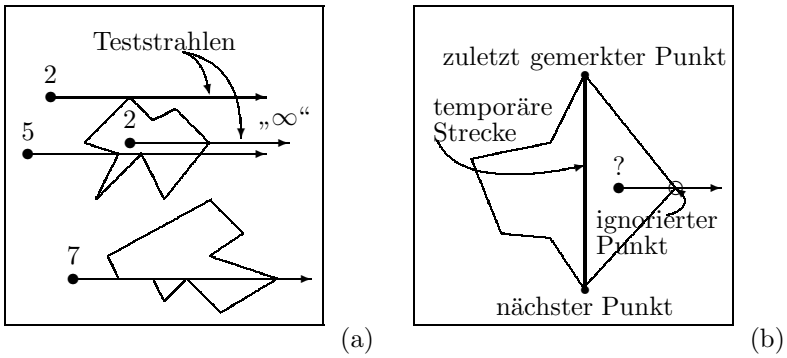


Abbildung 14: (a) Fehler durch Ecken, (b) Fehler in Sedgewicks Algorithmus

Der **2. Algorithmus** (*lies_on*?, liegt Punkt auf Strecke?) greift auf *ccw* zurück: entweder muß gelten

$$ccw(Start(Strecke), End(Strecke), Punkt) = 0$$
$$\text{oder}$$
$$ccw(End(Strecke), Start(Strecke), Punkt) = 0$$

[6] Tatsächlich ist die von mir verwendete Funktion ein ganzes Stück komplizierter, da die Funktion *dim* an dieser Stelle berechnet werden kann, was über Schlüsselwort-Parameter eingestellt wird.

Der **3. Algorithmus** (*intersects_line_seg_dim_1?*, eindimensionaler Schnitt zweier Strecken?) ist nicht trivial: erstens müssen sich die Strecken schneiden (Algorithmus 1), zweitens müssen sie parallel sein, und drittens müssen sie so zueinander liegen, daß die Schnittmenge mehr als nur einen Punkt enthält. Die Schnittmenge enthält nur einen Punkt, wenn sich z.B. ausschließlich die Endpunkte beider Strecken „schneiden“, also gleich sind. Parallelität ist somit nur notwendig, aber nicht hinreichend. Hier sind etliche Fälle zu berücksichtigen.

Für den **4. Algorithmus** (*point_inside_polygon?*, Punkt innerhalb Polygon?) wurde zunächst wieder das Buch von Sedgewick konsultiert (s. [12]). Sedgewicks Algorithmus stellte sich interessanterweise als falsch heraus, weswegen das Problem hier ausführlicher diskutiert wird. Ein Polygon wird - wie schon erwähnt - als Feld von Strecken repräsentiert. Prinzipiell wird nun von dem fraglichen Punkt aus ein „Teststrahl“ in irgendeine Richtung ausgesandt, und zwar so weit, daß der Endpunkt des Strahls sicher außerhalb des Polygones liegt (nahe „∞“). Für diesen Teststrahl wird nun für alle Polygonkanten (in der Reihenfolge des Poylgonstreckenpfades) entschieden, ob sie den Teststrahl schneiden (mit bereits erwähnter Funktion *intersects_line_seg?*). Die Anzahl der Schnitte wird gezählt - ist sie gerade, so ist der Punkt außerhalb des Polygones, ansonsten ist er innerhalb. Dabei muß nicht vorausgesetzt werden, daß das Polygon konvex ist.

Ein Problem ist nun jedoch, daß es passieren kann, daß der Teststrahl genau auf eine *Polygonecke* trifft: dann würden zwei Schnitte gezählt, da ein Eckpunkt zu genau zwei Polygonkanten gehört. Die Prozedur würde sagen, der Punkt sei innen - dies kann aber falsch sein (s. Abb. 14(a)). Natürlich hilft es auch nicht, diese Ecken als Eins zu zählen. Tatsächlich müssen die Eckpunkte, die auf dem Teststrahl liegen, gesondert behandelt werden. Der Sedgewick-Algorithmus geht so vor, daß stets der letzte nicht auf dem Teststrahl liegende Punkt memoriert wird. Alle nach ihm auf dem Teststrahl liegende Punkte werden ignoriert, der Zähler wird nicht verändert. Der nächste nicht mehr auf dem Teststrahl liegende Punkt jedoch wird dann zusammen mit dem zuletzt memorierten Punkt verwendet, um eine „temporäre Strecke“ zu bilden. Schneidet diese den Teststrahl, so wird der Zähler inkrementiert. Schließlich wird wieder anhand des Zählers entschieden. Aber auch diese Vorgehensweise ist nicht korrekt (siehe Abb. 14(b)): eventuell entstehen nämlich überhaupt keine Schnitte mit der temporären Strecke, obwohl der Punkt innerhalb ist. Der Zähler ist null, also gerade - die Funktion liefert den Wert für „außerhalb“. Korrekt wäre es, einen anderen Teststrahl in eine zufällige Richtung zu schicken, sobald eine Ecke angetroffen wird. Dies geschieht sooft, bis eine Entscheidung vorgenommen werden kann. Ein anderes korrektes Verfahren geht so vor, daß wie bei Sedgewicks Algorithmus der letzte und erste nicht mehr auf dem Teststrahl liegende Eckpunkt ermittelt wird, wenn eine Folge von Punkten zwischen diesen beiden auf dem Teststrahl liegt. Anhand der beiden ermittelten Eckpunkte wird nun entschieden, ob der Zähler inkrementiert wird oder nicht: liegen sie nämlich auf verschiedenen Seiten des Teststrahls (Abb. 15, So.fall a)), so wird der Zähler inkrementiert, ansonsten jedoch nicht (Abb. 15, So.fall b)). Dieses Verfahren wurde laut [13, S. 137 - 138] implementiert. Die „temporäre Strecke“ versucht gerade, diese Zählweise explizit zu machen. Man kann versucht sein, zu glauben, daß die temporäre Strecke genau dann einen Schnitt mit dem Teststrahl ergibt, wenn die diskutierten, nicht mehr auf dem Teststrahl liegenden Eckpunkte auf verschiedenen Seiten des Teststrahls liegen. Ein Schnitt mit der temporären Strecke impliziert natürlich, daß diese Punkte auf verschiedenen Seiten liegen, jedoch gilt die Umkehrung nicht. Genau hier liegt der Fehler.

Der **5. Algorithmus** (*one_part_inside?*, Teil einer Strecke im Polygon enthalten?) bedient sich des 4. Algorithmus (*point_inside_polygon?*): die fragliche Strecke wird

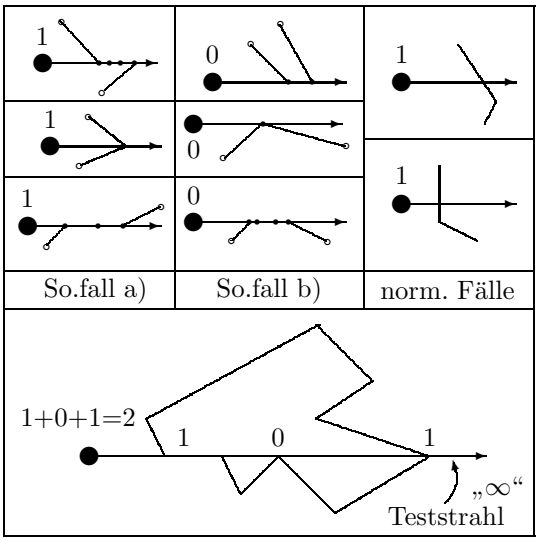


Abbildung 15: Korrekte Zählung der Ecken

dabei solange rekursiv zerteilt, bis die Länge der Stücke einen unteren Grenzwert erreicht hat oder aber sowohl Start- als auch Endpunkt des Streckenstückes im Polygon enthalten sind und die Teilstrecke das Polygon nicht schneidet.

3.3.3 Weitere nützliche Relationen

Nachdem nun vom Geometriemodul die Relationen zwischen den Grafikelementen einer grafischen Konstellation berechnet und in den Objekten gespeichert worden sind, lassen sich anhand der gewonnen Daten weitere, sekundäre Relationen erkennen. Dabei sind Grafikelemente CLOS-Instanzen, die entsp. Slots für diesen Zweck haben: so enthält z.B. der Slot `contained-by-objects` eines Objektes Referenzen auf alle Objekte, in denen es enthalten ist. Es stellt sich als günstig heraus, eine Relation wie *directly_contains*(A, B) einzuführen, da sie wahrscheinlich von vielen Anwendungen benötigt wird (z.B. PJ). Sie läßt sich nach Haarslev so definieren (s. Abb. 9(b)):

$$directly\_contains(A, B) \Leftrightarrow |contained\_by\_objects(B) - contained\_by\_objects(A)| = 1$$

Um einige weitere Relationen zu erkennen, muß also das Geometriemodul nicht unbedingt geändert werden. Stattdessen kann eine weitere Softwareschicht diesen Dienst leisten und anhand der berechneten primären Relationen sekundäre extrahieren.

Abweichend von der oben definierten Relation *covers* stellt es sich als günstig heraus, diese Relation so umzudefinieren, daß sie eine Teilmenge der Relation *directly_contains* wird. Dabei werden alle bisher schon (als nun fälschlich) erkannten *covers*-Elemente aus dem Slot `covers-objects` wieder ausgetragen, die nicht auch in der zuvor berechneten Relation *directly_contains* stehen. Abb. 16 veranschaulicht diese weitaus intuitivere Redefinition von (a) nach (b). Nun gilt also:

$$covers(x, y) \Rightarrow directly\_contains(x, y) \Rightarrow contains(x, y)$$

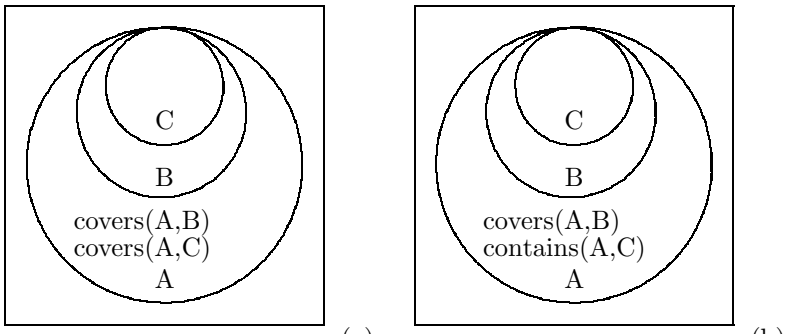


Abbildung 16: Redefinition der *covers*-Relation von (a) nach (b)

Für Gebilde wie Petrinetze ist es zudem notwendig, zu erkennen, ob bestimmte Objekte miteinander verbunden sind. Eine derartige Relation könnte *linked_over_with* genannt werden, wobei die Verbindung selbst gerichtet oder ungerichtet sein kann (z.B. durch einen Pfeil oder eine Strecke). Diese Relation ist aber im Geometriemodul nicht vorgesehen: sie läßt sich jedoch anhand der berechneten Relationen in Verbindung mit einem einfachen Suchalgorithmus erkennen.

Dafür ist es notwendig, zusätzlich die Relationen für die Start- und Endpunkte aller eindimensionalen Objekte zu berechnen. Ein einfacher Suchalgorithmus wird dann alle eindimensionalen Objekte L daraufhin überprüfen, ob es genau ein zweidimensionales Objekt A gibt, mit dem der Endpunkt von L in einer der Relationen *contains* (und somit auch *directly_contains* und *covers*, da diese Teilmengen von *contains* sind) oder *touches* steht, sowie ein zweidimensionales Objekt B entsprechend für den Startpunkt von L. Außerdem soll für das fragliche Objekt L gelten:

$$
\begin{array}{lllll}
\text{(Fall 1)} & intersects(A,L) & \wedge & intersects(B,L) & \vee \\
\text{(Fall 2)} & touches(A,L) & \wedge & touches(B,L) & \vee \\
\text{(Fall 3)} & touches(A,L) & \wedge & intersects(B,L) & \vee \\
\text{(Fall 3)} & intersects(A,L) & \wedge & touches(B,L) & \vee \\
\text{(Fall 4)} & touches(A,L) & \wedge & contains(B,L) & \vee \\
\text{(Fall 4)} & contains(A,L) & \wedge & touches(B,L) & \vee \\
\text{(Fall 5)} & contains(A,L) & \wedge & intersects(B,L) & \vee \\
\text{(Fall 5)} & intersects(A,L) & \wedge & contains(B,L) &
\end{array}
$$

In diesem Fall ist Objekt A mit Objekt B verbunden: es gilt $linked\_over\_with(A,B) \wedge linked\_over\_with(B,A)$. Handelt es sich bei Objekt L auch noch um ein gerichtetes Objekt, so läßt sich diese Aussage verfeinern bzw. verstärken: es gilt $start\_linked\_over\_with(A,B) \wedge end\_linked\_over\_with(B,A)$ oder andersrum, wobei es sich um Teilmengen der Relation *linked_over_with* handelt. Die Allgemeinheit dieses Ansatzes erlaubt es, sowohl Pfeile als auch Strecken als auch Splineketten usw. als Verbinder zu klassifizieren. Die genau Art der Vebindung kann auch der Wissensbasis zugänglich gemacht werden, welche dann (wenn Start- und Endpunkt auch als Objekte repräsentiert sind) weitere Klassifikationen vornehmen kann. Abb. 17 zeigt einige verbundene Objekte.

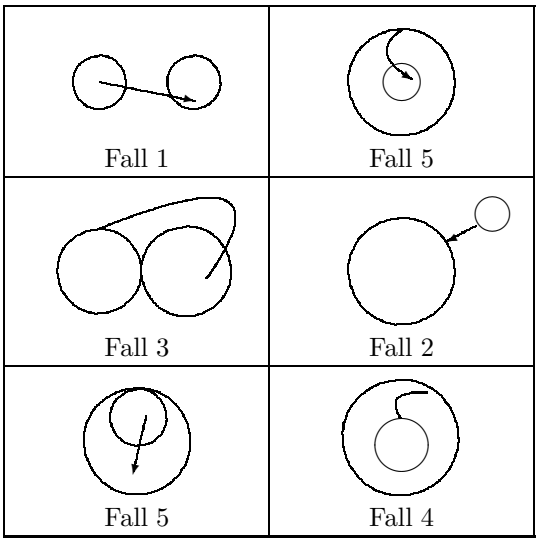


Abbildung 17: Verbundene Objekte

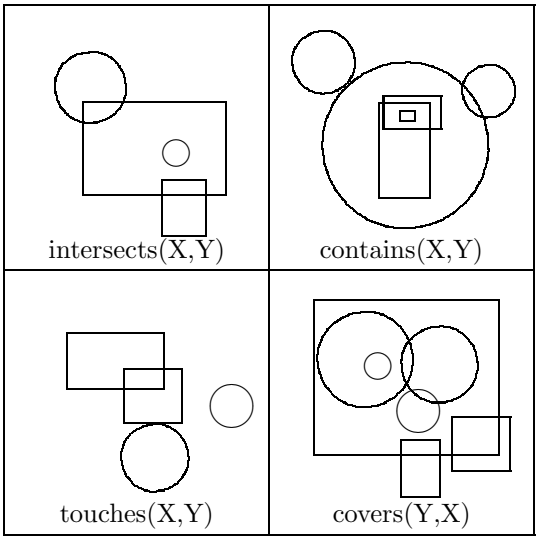


Abbildung 18: Relationen für Kompositionsobjekte

3.3.4 Topologische Relationen für Kompositionsobjekte

Alle bisher diskutierten Relationen sind ausschließlich für primitive Objekte gedacht: diese zeichnen sich dadurch aus, daß sie zusammenhängend sind und keine Löcher haben. GENED bietet jedoch die Möglichkeit, verschiedene Einzelobjekte zu einem Ganzen zusammenzufassen und als ein Kompositionsobjekt zu betrachten. Somit besteht die Notwendigkeit, topologische Relationen auch für diese Objekte zu berechnen. Dabei spielt die Funktion $has\_parts(object)$ eine wichtige Rolle: sie liefert die Menge der Komponentenobjekte, falls *object* ein zusammengesetztes Objekt ist, falls *object* jedoch primitiv ist, das *object* selbst.

Tatsächlich kann das Geometriemodul keine Relationen für Kompositionsobjekte berechnen: alle zusammengesetzten Objekte werden für das Geometriemodul in ihre Komponentenobjekte zerlegt. Für diese werden Relationen berechnet. Danach werden die Komponenten wieder zusammengefügt, und bestimmte Algorithmen berechnen dann (wie es z.B. auch f. *directly_contains* und *linked_over_with* der Fall ist) anhand der so gewonnen Beziehungungen zwischen den Komponenten die Relationen für die Kompositionsobjekte selbst. Am Ende erhält der Benutzer tatsächlich Relationen für aggregierte Objekte, obwohl diese nicht im Geometriemodul implementiert sind. Die skizzierten Suchalgorithmen sind wie auch die im letzten Unterkapitel diskutierten sekundären Relationen in einer das Geometriemodul benutzenden Softwareschicht implementiert. Abb. 18 veranschaulicht die Relationen für Kompositionsobjekte: alle Kreise sollen stets Komponenten von X sein, alle Rechtecke Komponenten von Y. Zusätzlich werden noch die *directly_contains* und *linked_over_with*-Relationen berechnet.

$$
\begin{aligned}
intersects(X,Y) \quad &\Leftrightarrow \quad \exists\, Part_X \in has\_parts(X), \\
&\qquad \exists\, Part_Y \in has\_parts(Y) : intersects(Part_X, Part_Y) \\
\\
contains(X,Y) \quad &\Leftrightarrow \quad \exists\, Part_X \in has\_parts(X), \\
&\qquad \forall\, Part_Y \in has\_parts(Y) : contains(Part_X, Part_Y) \\
\\
touches(X,Y) \quad &\Leftrightarrow \quad \neg intersects(X,Y) \wedge \neg contains(X,Y) \wedge \neg contains(Y,X) \wedge \\
&\qquad (\exists\, Part_X \in has\_parts(X), \\
&\qquad\ \exists\, Part_Y \in has\_parts(Y) : touches(Part_X, Part_Y)) \\
\\
covers(X,Y) \quad &\Leftrightarrow \quad \exists\, Part_X \in has\_parts(X) : \\
&\qquad (\forall\, Part_Y \in has\_parts(Y) : contains(Part_X, Part_Y) \wedge \\
&\qquad\ \exists\, Part_Y \in has\_parts(Y) : covers(Part_X, Part_Y))
\end{aligned}
$$

Folgende Prädikate ergeben die Dimension eines Schnittes - die Schnittdimension ist als Maximum der Dimensionen der Schnitte der Komponentenobjekte definiert:

$$
\begin{aligned}
intersects\_dim\_0(X,Y) \quad \Leftrightarrow \quad & intersects(A,B) \wedge \\
& (\forall\, Part_X \in has\_parts(X), \\
& \quad \forall\, Part_Y \in has\_parts(Y) : dim(Part_X, Part_Y) = 0)
\end{aligned}
$$

$$
\begin{aligned}
intersects\_dim\_1(X,Y) \quad \Leftrightarrow \quad & intersects(A,B) \wedge \\
& (\exists\, Part_X \in has\_parts(X), \\
& \quad \exists\, Part_Y \in has\_parts(Y) : dim(Part_X, Part_Y) = 1) \wedge \\
& (\forall\, Part_X \in has\_parts(X), \\
& \quad \forall\, Part_Y \in has\_parts(Y) : dim(Part_X, Part_Y) \leq 1)
\end{aligned}
$$

$$
\begin{aligned}
intersects\_dim\_2(X,Y) \quad \Leftrightarrow \quad & intersects(X,Y) \wedge \\
& (\exists\, Part_X \in has\_parts(X), \\
& \quad \exists\, Part_Y \in has\_parts(Y) : dim(Part_X, Part_Y) = 2)
\end{aligned}
$$

3.3.5 Skalierungs- und Δ-Probleme

Die Funktion Δ berechnet den kleinsten mathematischen Abstand zwischen zwei Objekten. Speziell für die Relationen *touches* und *covers* wird sie gebraucht. Für einen menschlichen Benutzer hängen diese Relationen u.a. auch von der Skalierung ab: ist die Zeichenfläche sehr verkleinert dargestellt (also $Global\_Scale\_Factor \ll 1$), so werden auch alle Abstände zwischen den Objekten entsprechend verkleinert dargestellt. Δ liefert unabhängig hiervon für 2 Objekte jedoch stets den gleichen Wert, nämlich den „mathematische“ Abstand. Kleine Positionsveränderungen der Objekte auf dem Schirm haben große Differenzen in den Koordinaten zufolge, so daß keine genauen Positionierung aufgrund der beschränkten Auflösung vorgenommen werden können. Der umgekehrte Fall tritt bei hoher Vergrößerung auf (also $Global\_Scale\_Factor \gg 1$): die Objekte können sehr genau positioniert werden, und kleine Veränderungen der Schirmposition haben kleine Auswirkungen in den Koordinaten. Dementsprechend kann ein Benutzer z.B. zwei Kreise in einer Skalierungsstufe als *touches*, in einer anderen aber als *disjoint* ansehen, da die Differenz zwischen den beiden Objekten auch skaliert und so mal größer, mal kleiner erscheint. Eine „gute“ *touches*-Relation sollte also das ϵ anhand der Bildschirmauflösung und des Skalierungsfaktor berechnen: geht $Global\_Scale\_Factor$ Richtung 0, sollte ϵ immer größer werden, geht der Faktor Richtung ∞, so sollte ϵ immer kleiner werden. Zudem spielt die relative Größe zweier Objekte für einen menschlichen Betrachter oftmals eine Rolle.

Ein anderes Problem ist, daß Textobjekte in der momentanen CLIM-Implementation nicht skaliert werden können: ihre Pixelausdehnung auf dem Schirm ist stets gleich, die der anderen Objekte schrumpft oder wächst jedoch entsprechend der Skalierung. Somit sind die vom Geometriemodul berechneten Relationen nicht invariant bezüglich affiner Transformationen. Sieht man die Invarianz als wesentlich an, so sind die hier implementierten Relationen keine topologischen Relationen - pragmatisch gesehen erüllen sie jedoch in diesem Rahmen ihren Zweck, wenn auch nicht in Übereinstimmung mit den mathematischen Definitionen. Die Diskretisierung erfordert also einige Kompromisse.

3.3.6 Der Entscheidungsbaum des Geometriemoduls

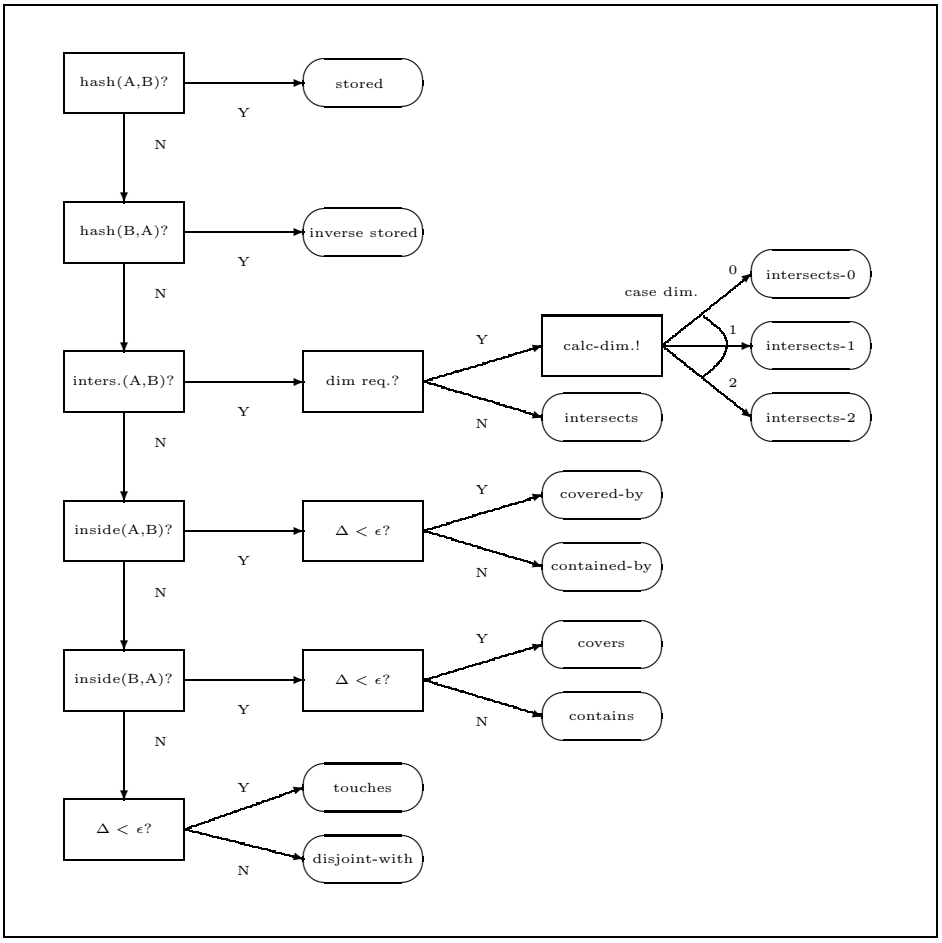


Abbildung 19: Entscheidungsbaum

Das Geometriemodul verwendet einen *Entscheidungsbaum,* der in Abbildung 19 dargestellt ist. Berechnende LISP-Funktionen erscheinen als Rechtecke, LISP-Prädikate enden mit „?“. In den Ovalen stehen jeweils die zurückgegebenen Relationssymbole, bis auf `stored`: der Wert dieser Variable ist ein aus einer Hashtabelle gewonnenes Relationssymbol. Bevor irgendwelche Berechnungen vorgenommen werden, wird zunächst stets diese Hashtabelle konsultiert und ein für die beiden Objekte eventuell bereits zuvor berechnetes Relationssymbol in `stored` gespeichert und dann zurückgegeben (evt. auch das inverse Relationssymbol `inverse stored`).

Offensichtlich ist die Vollständigkeit, da stets ein Symbol zurückgeliefert wird. Ebenso schließen sich die einzelnen Fälle gegenseitig aus. Die Funktion wird sukzessive mit allen Objektpaaren als Argumenten aufgerufen, wobei die Hashtabelle eventuell überflüssige Berechnung (aus $contains(a, b)$ folgt $contained\_by(b, a)$, s. `inverse stored`) abfängt. Zusätzlich läßt sich über einen Schlüsselwort-Parameter steuern (`dimension required?`), ob auch die Dimension der Schnitte berechnet werden soll. Diese Berechnung ist sehr teuer (rekursive Algorithmen), weswegen zunächst geprüft wird, ob die Objekte sich überhaupt schneiden. Im nächsten Kapitel wird die Verwendung der Relationen in einem größeren Kontext dargestellt.

4 Die Wissensrepräsentationssprache CLASSIC

4.1 Motivation

Der *Klassifizierer* des anvisierten Grafikeditors benötigt eine geeignete Repräsentation des Domänenwissens, damit gewisse Konstellationen grafischer Objekte erkannt werden können. Wie in Kapitel 2 diskutiert, könnte hierzu FOL in Verbindung mit spez. topologischen Relationen dienen, wären traktable und entscheidbare Algorithmen implementierbar (was nicht der Fall ist). Eine geeignetere Repräsentation läßt sich jedoch durch die Benutzung einer Wissensrepräsentationssprache wie CLASSIC (Classification of Individuals and Concepts) erreichen. Dieses Kapitel soll dem Leser - so weit wie erforderlich - ein Verständis für die sich aus dem wissensbasierten Ansatz ergebenden Möglichkeiten vermitteln. Letztendlich können jedoch nur die für diese Arbeit relevanten Dinge beleuchtet werden. Eine Einführung in CLASSIC findet der Leser in [8], Details in [9]. Eine allgemeine Einführung in die Wissensrepräsentation ist in [4, Kap. 1.1, Kap. 4.5] sowie Lehrbüchern der *Künstlichen Intelligenz (KI)* (wie z.B. [21], [19]) zu finden.

4.2 Allgemeines zur KL-ONE-Familie und CLASSIC

Durch die Benutzung einer Wissensrepräsentationssprache wie CLASSIC läßt sich eine *Taxonomie von Konzepten* aufbauen. Neue Konzepte bzw. Begriffe werden durch *Spezialisierungen* bereits bestehender Konzepte konstruiert, was teils durch Konjunktion, teils durch andere Einschränkungen geschieht. Diese Betrachtungsweise ist z.B. durch das *Entity-Relationship*-Modell oder die *Semantischen Netze* der KI bekannt. Die Semantischen Netze lieferten u.a. die Entwurfsperspektive für das KL-ONE-System (Knowledge Representation Language One), den Urvater der KL-ONE-Familie, zu der auch CLASSIC gehört (s. auch [7]). Konzepttaxonomien bilden den Kern der zur Definition von neuem Wissen bzw. neuen Begriffen erforderlichen *terminologischen Komponente* (T-Box) derartiger Systeme (s. [4, S. 331]).

Ein Standardbeispiel eines Semantischen Netzes zeigt Abb. 20 (aus [19, S. 460], nach P. Winston): es beschreibt einen Torbogen und die Beziehungen seiner Teilobjekte untereinander: dabei bezeichnet „A“ den Torbogen als Ganzes, der aus den drei Teilobjekten „B“, „C“ und „D“ besteht. Alle Objekte sind Instanzen des Konzeptes „Brick“. Objekt „B“ muß von „C“ und „D“ getragen werden. Zudem dürfen sich „C“ und „D“ nicht berühren und stehen in den Beziehungen „links-von“ und „rechts-von“ zueinander.

Die Semantik der Sprachausdrücke KL-ONE-ähnlicher Sprachen kann durch FOL formal spezifiziert werden - in diesem Zusammenhang spricht man auch von *Beschreibungslogiken (Description Logics)*. Um Eigenschaften wie Vollständigkeit und Traktabilität gewisser Dienste und Inferenzalgorithmen solcher Sprachen zu erreichen, muß oftmals die Ausdrucksfähigkeit (im Vergleich zu FOL) stark beschnitten werden. Auch CLASSIC wurde ursprünglich mit dem Ziel entworfen, möglichst große Ausdrucksfähigkeit bei vollständigen und traktablen Inferenzalgorithmen zu erreichen (s. [9, S. 1]). Zudem muß das Problem der *Nichtmonotonie*, welches sich mit konventioneller (monotoner) FOL nicht beschreiben läßt, angemessen behandelt werden.

Ähnlich einer relationalen Datenbank werden nun sogenannte *Individuen*, also Domänenobjekte bzw. Konzeptinstanzen erzeugt, die dann mit Hilfe von *Rollen (Roles)* bzw. Relationen zueinander in Beziehung gesetzt werden können. Was ein

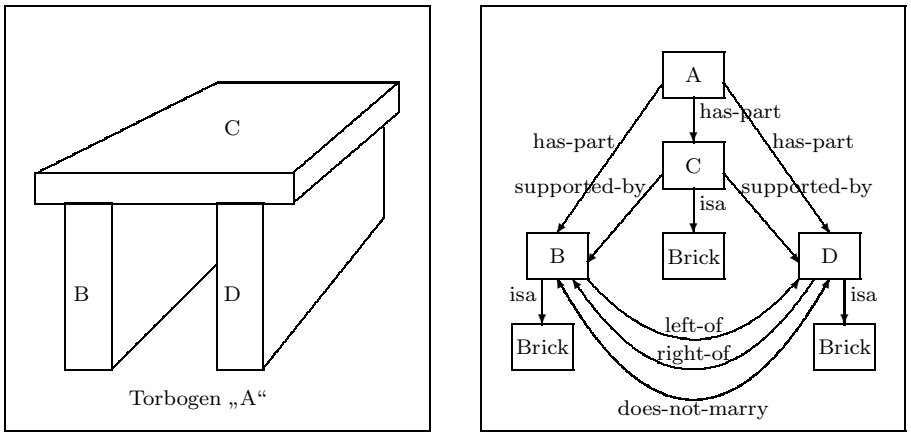


Abbildung 20: Torbogen und Semantisches Netz

derartiges Wissensrepräsentationssystem gegenüber einer relationalen Datenbank auszeichnet, ist die Fähigkeit, *selbständig die Position eines neuen Konzeptes oder Individuums in der Taxonomie von Konzepten berechnen zu können.* Tatsächlich ist der angestrebte *Klassifizierer* nun also ein Basisdienst von CLASSIC, wodurch in CLASSIC repräsentierte grafische Konstellationen erkannt werden können. Dies wird möglich aufgrund der logischen Äquivalenz der Semantik der Sprachausdrücke zu eingeschränkter FOL. Die *Konsistenz* der Wissensbasis wird automatisch sichergestellt bzw. erzwungen - der Benutzer darf und kann keine beliebigen Veränderungen der Wissensbasis vornehmen (Problem der Nichtmonotonie, Revisionsproblem). Zudem bietet CLASSIC *Regeln (Rules)*, womit sich u.a. regelbasierte Expertensysteme implementieren lassen.

4.3 Konzepte und Individuen

Konzepte sind Mengen, die durch logische Formeln beschrieben bzw. definiert werden, genauer: *unäre Prädikate.* Ist also ein Element in der durch die Konzeptdefinition beschriebenen Menge enthalten, so gilt das entsprechende Prädikat, andernfalls nicht.

So läßt sich das Konzept Transition wie in Kap. 2 folgendermaßen beschreiben:

$$\begin{aligned} transition(x) \quad \Leftrightarrow \quad & rechteck(x) \wedge \\ & (\neg\exists\, y : (intersects(x,y) \vee contained\_by(x,y) \vee contains(x,y))) \wedge \\ & (\forall\, y : touches(x,y) \Rightarrow netzkante(y)) \wedge \\ & (\forall\, y : linked\_over\_with(x,y) \Rightarrow stelle(y)) \end{aligned}$$

Damit ist begrifflich, also *konzeptionell*, die Menge aller Transitionen (aus einer bestimmen Modellierungsperspektive) beschrieben. Es gilt:

$$x \in \mathfrak{Transition} \Leftrightarrow transition(x)$$

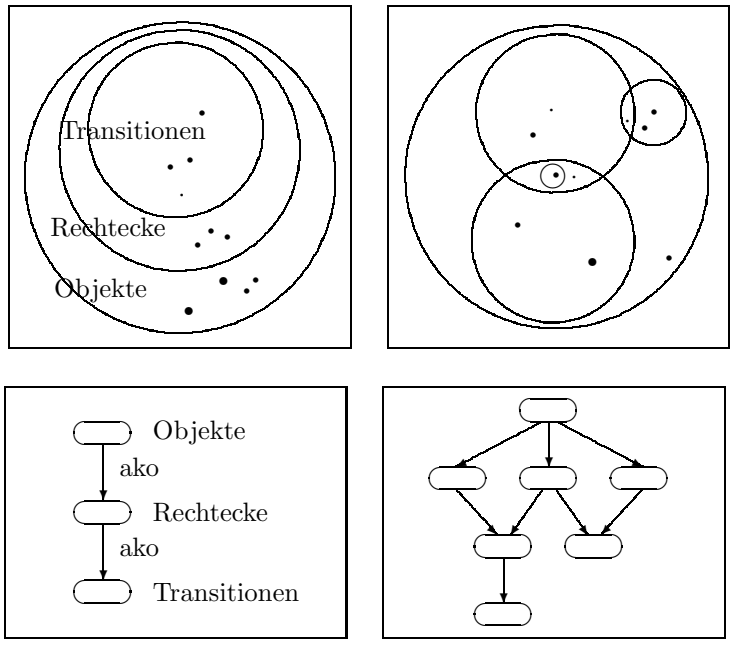


Abbildung 21: Mengeninklusionen und DAGs

Die hier vorgestellte Formel beschreibt eine Teilmenge einer anderen Menge, nämlich der Menge der Rechtecke, denn es ist für eine Transition unter anderem auch notwendig und hinreichend, Rechteck zu sein. Hier liegt bereits ein Beispiel einer sehr einfachen *Konzepthierarchie* vor: jede Transition ist auch Rechteck, die entsprechende Beziehung heißt *ako* von *A Kind Of*. Eine Transition ist also *A Kind Of* Rechteck. Grafisch läßt sich diese *Mengeninklusion* wie in Abb. 21 veranschaulichen.

Hier gilt also:

$$\text{Transitionen} \subset \text{Rechtecke} \subset \text{Objekte}$$

Kompliziertere Wissensbasen erfordern meist eine Konzepttaxonomie, also einen gerichteten, nichtzyklischen Graphen (DAG).

Durch die *Mengensemantik* ergeben sich gewisse Eigenschaften, u.a. die Vererbung von definierenden Eigenschaften: eine Transition hat - unter anderem - auch alle Eigenschaften eines Rechteckes. Die konzeptdefinierenden Formeln sind durch eine logisches Konjunktion verknüpft. Die entsprechende Mengenoperation ist die Schnittbildung. In Systemen wie CLASSIC können Schnittmengen beliebig vieler Obermengen gebildet werden - Voraussetzung ist jedoch, daß dabei nicht die leere Menge entsteht. Diese Menge wird in der Wissensrepräsentation *inkonsistent* genannt. Ein Konzept oder Individuum wird immer dann inkonsistent, wenn die Beschreibungen, aus denen es aggregiert wird, sich logisch widersprechen, also unerfüllbar sind, wie in dem Beispiel

$$odd\_and\_even\_number(x) \Leftrightarrow even\_number(x) \wedge odd\_number(x)$$

Es ist offensichtlich, daß es solche x nicht geben kann und die dadurch beschriebene Menge leer ist, denn ihre Beschreibungen sind *kontradiktorisch*.[7] Systeme wie CLASSIC entdecken und melden die Inkonsistenz.

[7] Tatsächlich ließen sich schon die Konzepte even_number und odd_number in CLASSIC gar nicht ohne „Tricks“ formulieren.

Wie in Abb. 21 ersichtlich, kann ein Element in mehreren Konzeptmengen enthalten sein. Ein solches Element erfüllt dann mehrere Konzeptdefinitionen. Ein Individuum `Transition-123` ist konzeptionell sowohl eine Transition, ein Rechteck, als auch ein Objekt. Das Element `Transition-123` erfüllt für alle diese Konzepte jeweils die definierenden Eigenschaften.

Ein Konzept kann Vorgänger- und Nachfolgerkonzepte haben: so ist das Konzept Transition ein Nachfolgerkonzept und das Konzept Objekt ein Vorgängerkonzept des Konzeptes Rechteck. Die direkten Vorgängerkonzepte (Ancestors Concepts) nennt man Elternkonzepte (Parent Concepts), die direkten Nachfolgerkonzepte (Descendant Concepts) hingegen Kindkonzepte (Child Concepts). Je tiefer ein Konzept im DAG steht, desto *spezieller* ist es; je weiter oben, desto *genereller.*

Elemente einer nichtleeren Konzeptmenge werden in CLASSIC Individuen genannt. Die entsprechende Relation heißt *isa* ($\in$). Für ein Element `Individuum-123` der Menge $\mathfrak{Transition}$ läßt sich also schreiben: *isa(Individuum-123,Transition)*, was $Individuum\_123 \in \mathfrak{Transition}$ bzw. $transition(Individuum\_123)$ bedeutet. CLASSIC zeichnet sich dadurch aus, daß die Sprachelemente zur Bildung von Ausdrücken (Beschreibungen) sowohl für Individuen als auch für Konzepte die selben sind. Diese Eigenschaft macht die Sprache übersichtlich und leichter erlernbar.

Wie bereits erwähnt, können Systeme wie CLASSIC nun durch Ausnutzung der Äquivalenz zu eingeschränkter FOL die Position eines neuen Konzeptes oder Individuums im Konzept-DAG selbständig berechnen. Dieser Prozeß wird *Klassifikation* genannt: jedes Konzept bzw. Individuum wird aufgrund seiner logischen Eigenschaften im DAG eingeordnet. Dabei spielt der *Subsumptionsalgorithmus* eine entscheidende Rolle: es handelt sich hierbei um die Lösung des *Entscheidungsproblemes,* ob ein Konzept A genereller (oder zumindestens äquivalent) als ein Konzept B ist (s. [4, S. 30]), ob also gilt: $\mathfrak{A} \subseteq \mathfrak{B}$. Tatsächlich wird der ganze DAG mit Hilfe des Subsumptionsalgorithmus' aufgebaut. In vielen Sprachen der KL-ONE-Familie ist Subsumption entweder nichttraktabel oder gar unentscheidbar, wodurch die korrekte Klassifikation nicht in allen Fällen gewährleistet ist.

Wenn also ein neues Konzept oder Individuum beschrieben wird (wodurch es bestimmte logische Eigenschaften hat), so berechnet CLASSIC anhand der Beschreibung automatisch alle subsumierenden (bereits definierten) Konzepte für dieses Objekt. Daraus ergibt sich dann die Klassifikation desselben - da die Konzepttaxonomie statisch ist, sind die speziellsten gefundenen subsumierenden Konzepte des Objektes dessen Eltern-, alle anderen subsumierenden jedoch Vorgängerkonzepte.

Doch wie sieht nun die CLASSIC-Syntax, also Individuen- und Konzeptbeschreibungssprache aus? Obige FOL-Formel (Def. Transition) läßt sich in CLASSIC wie folgt übersetzen:

```
(cl-define-concept 'transition
                   '(and rechteck
                         (at-most 0 intersects)
                         (at-most 0 contained-by)
                         (at-most 0 contains)
                         (all touches netzkante)
                         (all linked-over-with stelle)))
```

Wenn auch die ursprüngliche Formel wiederzuerkennen ist, so wird doch deutlich, daß hier bereits die Beschränkungen der Sprache in Kauf genommen werden müssen: der negierte Existenzquantor wird hier durch `(at-most 0 ...)` formuliert. Außerdem gibt es keine logisches Disjunktion: `(at-most 0 (or intersects`

`contained-by...))` ist kein Wort der CLASSIC-Grammatik. Stattdessen wurde aus

$$(\neg\exists\, y : (intersects(x, y) \vee contained\_by(x, y) \vee contains(x, y)))$$

die Formel

$$(\forall\, y : (\neg intersects(x, y) \wedge \neg contained\_by(x, y) \wedge \neg contains(x, y)))$$

Diese Umformung ist sogar äquivalent (im Sinne der Logik), jedoch läßt sich für die meisten komplizierteren Formeln keine bedeutungserhaltende CLASSIC-Beschreibung angeben.

Offensichtlich handelt es sich bei *intersects* oder auch *contained_by* um Relationen: gilt also $intersects(A, B)$, so ist in der Mengensemantik das Paar (A, B) Element der Menge Intersects. Relationen werden in CLASSIC als *Rollen (Roles)* bezeichnet; die Individuen, die durch die Relation in Beziehung gesetzt werden, als *Füller (Filler)* der Rolle. Füller können bezüglich Art und Anzahl restringiert werden. Ist ein Element x Instanz des Konzeptes Transition, so bedeutet ein Ausdruck wie `(all touches netzkante)`, daß alle Individuen y, mit denen x in der Relation *touches* steht, Instanzen des Konzeptes Netzkante sein müssen.

Die spezielle Syntax (Listenstruktur, `'` und `@`) der CLASSIC-Ausdrücke ergibt sich aus der Einbettung in das Wirtssystem COMMON LISP. `@name` bezeichnet im folgenden stets das Objekt mit Namen `name` *selbst.* Rollen und Konzepte müssen bereits definiert sein, bevor sie verwendet werden können - zyklische Definitionen sind nicht formulierbar.

4.3.1 Konzepte und Primitive Konzepte

Bis jetzt wurden nur die *Konzepte (Concepts)* erwähnt. Im logischen Sinne liegt bei einer Konzeptdefinition eine *genau-dann-wenn-Beziehung,* $\Leftrightarrow$ vor: wenn ein Individuum beschrieben wird, wird es anhand seiner logischen Eigenschaften klassifiziert. Dabei wird also anhand der Beschreibung (Prämisse) auf das Konzept (die Konklusion) geschlossen. Wird hingegen ein Individuum mit der Aussage erzeugt, daß es Instanz bestimmter Konzepte sei, so gelten für dieses Individuum auch alle anderen, nicht erwähnten logischen Einschränkungen, die natürlich für alle Elemente dieser Menge gelten. Derartige Aussagen werden *Zusicherungen (Assertions)* genannt. Die diskutierte Art der Konzeptdefinition ist somit *hinreichend* als auch *notwendig.*

Als Beispiel dienen wieder Transitionen. Mit

```
(cl-create-ind <name> <classic-description>)
```

wird in CLASSIC ein Individuum `<name>` mit der Beschreibung `<classic--description>` (ein Wort der CLASSIC-Grammatik) erzeugt. `<classic--description>` kann auch ein einfaches Konzept sein:

```
(cl-create-ind 'transition-123 'transition)
```

$$\Rightarrow$$

$$\begin{aligned}
&rechteck(x) \wedge \\
&(\neg\exists\, y : (intersects(x, y) \vee contained_by(x, y) \vee contains(x, y))) \wedge \\
&(\forall\, y : touches(x, y) \Rightarrow netzkante(y)) \wedge \\
&(\forall\, y : linked_over_with(x, y) \Rightarrow stelle(y))
\end{aligned}$$

All diese Eigenschaften gelten von nun an für das Individuum, insbesondere auch die über Rechteck und Objekt vererbten.

Wird hingegen ein Individuum mit `<classic-description>` erzeugt, wird das Individuum automatisch klassifiziert:

```
(cl-create-ind 'individuum-123
               '(and rechteck
                     (at-most 0 contains)
                     (at-most 0 intersects)
                     (at-most 0 contained-by)
                     (all touches netzkante)
                     (all linked-over-with stelle)))
```

$$\Rightarrow$$

$$transition(individuum\_123) \equiv individuum\_123 \in \mathfrak{Transition}$$

Der *Subsumptionstest* - der mit allen bereits definierten Konzepten (der Form $\Leftrightarrow$) durchgeführt wird - ergibt, daß das `Individuum-123` u.a. auch Instanz des Konzeptes Transition ist.

Ein Beispiel für ein inkonsistentes Individuum ist

```
(cl-create-ind 'inconsistent-123
               '(and transition
                     (at-least 1 intersects)
                     (all touches netzkante)
                     (all linked-over-with stelle)))
```

denn die über das Konzept Transition vererbte logische Eigenschaft bzw. Einschränkung

`(at-most 0 intersects)`
widerspricht sich direkt mit dem angegebenen
`(at-least 1 intersects)`

In Wissensrepräsentationssprachen der KL-ONE-Familie gibt es nun noch eine weitere Konzeptart: die *Primitiven Konzepte (Primitive Concepts)*. Dabei handelt es sich um Definitionen der Art

$$\text{Konzeptname} \Rightarrow \text{Konzeptdefinition}$$

während es sich bei den *Konzepten* um Definitionen der Art

$$\text{Konzeptname} \Leftrightarrow \text{Konzeptdefinition}$$

handelt.

Bei einem *primitiven Konzept* wird das System also niemals von den Eigenschaften des Individuums auf das Konzept selbst schließen: stattdessen muß vom Benutzer zugesichert werden, daß jenes Individuum Instanz dieses Konzeptes ist. Solche Definitionen sind also notwendig, aber nicht hinreichend. Man findet die primitiven Konzepte in der Konzepttaxonomie weit oben; sie sind meist die Wurzeln des DAG.

Für ein System wie einen generischen Grafikeditor ist es ohne Methoden der Bildverarbeitung nicht möglich, z.B. einen Kreis zu klassifizieren. Ein Kreis kann auf CLASSIC-Ebene nicht mit angemessenem Aufwand konzeptionell definiert werden - somit wird das Konzept Kreis ein primitives Konzept sein müssen. Es wird also vom Grafikeditor aus zugesichert, daß ein Objekt Instanz des Konzeptes Kreis ist. Primitive Konzepte werden in CLASSIC so definiert:

```
(cl-define-primitive-concept 'kreis 'objekt)
```

Ein anderes Beispiel ist das Konzept Person: eine Person hat so viele Eigenschaften, daß es nicht möglich ist, eine vollständige Definition anhand von notwendigen und hinreichenden Bedingungen zu geben. Somit kann ein KL-ONE-System niemals von selbst eine Objekt anhand seiner Beschreibung als Person klassifizieren.

Eine weitere Art der primitiven Konzepte sind die *unvereinbar primitiven Konzepte (Disjoint Primitive Concepts):* bei ihnen handelt es sich ebenfalls um primitive Konzepte, jedoch ist bekannt, daß die Schnittmenge zwischen einigen (einer Gruppe) leer ist. Die Konzepte Kreis und Rechteck sollten sicherlich unvereinbar sein, da kein Kreis gleichzeitig ein Rechteck sein kann, und umgekehrt. Bei allen anderen Konzepten wird diese Annahme hingegen nicht gemacht (es sei denn, ihre Beschreibungen sind logisch kontradiktorisch). Ein Versuch, ein Individuum mit

```
(cl-create-ind 'kreis-rechteck
               '(and kreis rechteck))
```

zu erzeugen, für also zu einem inkonsistenten Individuum, wenn Kreis und Rechteck wie folgt definiert worden sind:

```
(cl-define-disjoint-primitive-concept 'kreis 'objekt 'gruppe)
(cl-define-disjoint-primitive-concept 'rechteck 'objekt 'gruppe)
```

Das Symbol gruppe nimmt dabei die erwähnte Gruppierung vor, so daß CLASSIC erkennen kann, welche Konzepte nun unvereinbar sein sollen.

4.3.2 Beispiel: Basiskonzepttaxonomie von GENED

In Abb. 22 ist ein Teil der Basiskonzepttaxonomie von GENED dargestellt. Alle Konzepte sind primitiv und teilweise unvereinbar: Classic-Thing ist ein eingebautes Wurzelkonzept, wovon alle Domänenkonzepte abzuleiten sind (das andere eingebaute Wurzelkonzept heißt Host-Thing, womit die Objekte der Wirtssprache COMMON LISP gemeint sind). Einige Konzepte dienen nur der Strukturierung der Taxonomie; von ihnen werden niemals Individuen erzeugt, so z.B. Gened-Thing, 1d, 2d. Die Konzepte, die mit g- beginnen, werden hingegen direkt von GENED für die Individuen zugesichert, denn für jedes Grafikobjekt auf dem Editorschirm muß ein korrespondierendes CLASSIC-Individuum erzeugt werden. Die dargestellte Konzepttaxonomie muß stets als Basis erhalten bleiben - alle GENED-Wissensbasen müssen auf ihren Definitionen aufbauen, was später anhand einer Petrinetz-Wissensbasis verdeutlicht wird.

Zur Konzeptdefinition bietet CLASSIC noch diverse andere Sprachkonstrukte an. Sie sollen jedoch nicht weiter diskutiert werden. Zudem ist es möglich, sogenannte Testfunktionen in die Konzeptdefinition mit einzubringen: ein Individuum ist nur dann Instanz dieses Konzeptes, wenn auch die Testfunktion für das Individuum wahr zurückgibt. Doch dazu später mehr.

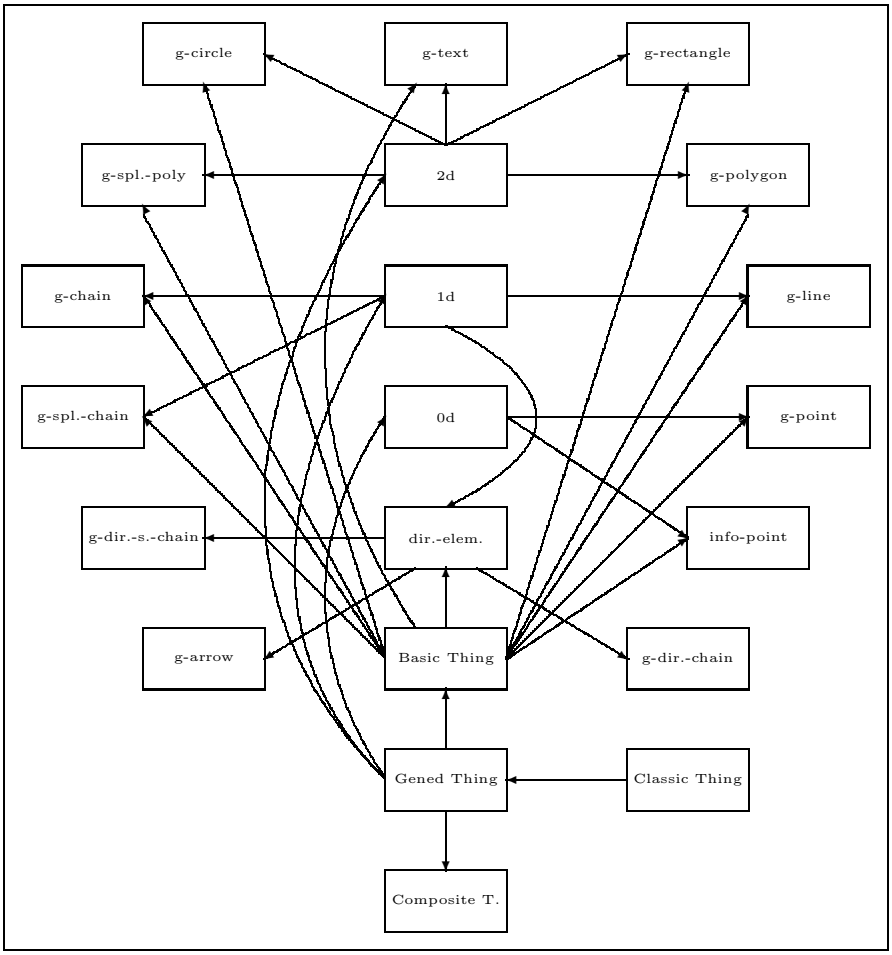


Abbildung 22: Die GENED-Konzeptaxonomie

4.4 Rollen und Relationen

Bei Rollen handelt es sich um Relationen, und Relationen sind binäre Prädikate, also Mengen von Paaren. Also gilt

$$intersects(A, B) \Leftrightarrow (A, B) \in \mathfrak{Intersects}$$

Da die Relation *intersects* symmetrisch ist, gilt

$$\begin{gathered} intersects(A, B) \Leftrightarrow intersects(B, A) \\ \equiv \\ (A, B) \in \mathfrak{Intersects} \Leftrightarrow (B, A) \in \mathfrak{Intersects} \end{gathered}$$

Man sagt, daß eine Relation rel_1 die *Inverse* einer Relation rel_2 ist, wenn gilt:

$$\begin{gathered} rel_1(A, B) \Leftrightarrow rel_2(B, A) \\ \equiv \\ (A, B) \in \mathfrak{Rel}_1 \Leftrightarrow (B, A) \in \mathfrak{Rel}_2 \end{gathered}$$

Die Relation *intersects* ist also, da sie symmetrisch ist, zu sich selbst invers. In CLASSIC würde man definieren:

```
(cl-define-primitive-role 'intersects
   :parent 'spatial-relation
   :inverse 'intersects
   :inverse-parent 'spatial-relation)
```

In diesem Beispiel ist die Rolle `intersects` Teil einer *Rollenhierarchie.* Da Relationen Mengen von Paaren sind, kann man somit ebenfalls von *spezialisierteren* oder *generelleren Rollen* entsprechend der Teilmengeneigenschaft (wie bei den Konzepten) sprechen: obige CLASSIC-Definition bedeutet, daß die Relation *intersects* eine Teilmenge der Relation *spatial_relation* ist (`:parent 'spatial-relation`), daß also gilt: $\mathfrak{Intersects} \subseteq \mathfrak{Spatial\_Relation}$. Oberrelationen müssen zuvor definiert worden sein. Im Gegensatz zu Konzepttaxonomien bilden Rollenhierarchien jedoch keine DAGs, sondern Bäume, da eine Rolle stets nur eine Oberrolle haben kann. Wie auch bei den Konzepten wird nicht angenommen, daß die Unterrelationen disjunkt sind. Die Relation *intersects* ist zu sich selbst invers - bei Relationen wie *contains* hat die Inverse jedoch den Namen *contained_by*.

CLASSIC-Ausdrücke der Art `(at-least <n> <role-name>)` und `(at-most <n> <role-name>)` werden als *Kardinalitätsbeschränkungen (Number Restrictions)* der Rolle `<role-name>` bezeichnet. Bei Ausdrücken wie `(all <role-name> <concept-name>)` ist hingegen von *Wertebeschränkungen (Value Restriction)* der Rolle `<role-name>` die Rede.

Allgemein bedeutet nun für Individuum x

`(at-most <n> <role>)`
$$\Leftrightarrow$$
$$|\{y : role(x, y)\}| \leq n$$

`(at-least <n> <role>)`
$$\Leftrightarrow$$
$$|\{y : role(x, y)\}| \geq n$$

und

`(all <role> <concept>)`
$$\Leftrightarrow$$
$$(\forall y \in \{y : role(x, y)\} : concept(y))$$

Die Ausdrücke sind hier also als Prädikate bezüglich x (objektzentriert) definiert. Wenn im folgenden von Rollenfüllern der Rolle `<role>` die Rede ist, so sind bezüglich des Objektes x stets alle Individuen y der Menge $\{y : (x, y) \in \mathfrak{Role}$ bzw. $role(x, y)\}$ gemeint. Letztendlich kommt die Bezeichnungsweise Füller von den „Slot And Filler Structures“ oder „Frame Systems“, wo Relationen objektzentriert in „Slots“ der einzelnen Objekte (z.B. x) verwaltet werden.

Wiederum ergibt sich für Rollen einer Rollenhierarchie durch die Teilmengeneigenschaft die Vererbung von Beschränkungen: Gilt für eine Oberrolle `<role>` bezüglich des Individuum x eine Kardinalitätsbeschränkung von z.B. `(at-most <n> <role>)`, so ist klar, daß diese Beschränkung bezüglich x auch für alle Unterrollen `<role-i>` gelten muß, denn aus

$$|\{y : role(x,y)\}| \leq n \wedge \forall i : \mathfrak{Role}_i \subseteq \mathfrak{Role}$$

$$\text{folgt wegen } \forall i : |\mathfrak{Role}_i| \leq |\mathfrak{Role}|$$

$$\forall i : |\{y : role_i(x,y)\}| \leq n$$

Der letzte Ausdruck bedeutet aber für x gerade `(at-most <n> <role-i>)` für alle Unterrollen. Wenn für eine Unterrolle `<role-i>` von `<role>` hingegen `(at-least <n> <role-i>)` gilt, so muß dies natürlich auch für alle Oberrollen gelten. Während `at-most`-Beschränkungen den Rollenbaum hinunterpropagieren (bzgl. eines Individuums x), so propagieren `at-least`-Beschränkungen von den Blättern zur Wurzel hinauf. Werden nun zwei Individuen A und B durch eine Rolle in Beziehung gesetzt, so stehen beide Objekte auch über alle Oberrollen dieser Rolle zueinander in Beziehung, was ebenfalls durch die Teilmengeneigenschaft verständlich wird. Alle Rollen sind primitiv (wie im Sinne der primitiven Konzepte), so daß CLASSIC niemals Relationen klassifiziert. Stattdessen muß der Benutzer die Subsumptionsbeziehung explizit machen (durch `:parent`).

Einen spezialisierten Rollentyp bilden die *Attribute*: hierbei handelt es sich um Relationen, deren Kardinalität auf eins beschränkt ist. Attribute können nicht Teil einer Rollenhierarchie sein, denn eine Teilmengenspezialisierung macht keinen Sinn, da hierbei die leere Menge entstünde. Dafür ist es aber möglich, mit ihnen „Mengengleichheit“ auszudrücken, die für allgemeine Relationen in CLASSIC nicht formulierbar ist (Operator `same-as`).

Ein Beispiel für ein Attribut ist die Rolle

```
(cl-define-primitive-role 'x-position :attribute t)
```

Es dient dazu, im Individuum die X-Bildschirmposition des korrespondierenden Grafikobjektes zu speichern. So wird es möglich, CLASSIC prozedural die Entfernung zweier Objekte berechnen zu lassen (was das LISP-System erledigt).

4.4.1 Beispiel: Basisrollenhierarchie von GENED

Abb. 23 zeigt den größten Teil der Basisrollenhierarchie von GENED. Die wirklichen Namen konnten nur angedeutet werden. Attribute sind als Ovale visualisiert, alle anderen Rollen als Rechtecke. Ob zwei Rollen invers zueinander sind, ist im Baum nicht ersichtlich:

Zueinander inverse Rollen		
contains-objects	↔	contained-in-objects
directly-contains-objects	↔	directly-contained-by-object
covers-objects	↔	covered-by-object
start-linked-over-with	↔	end-linked-over-with
linker-objects	↔	points-related-with
start-linker-objects	↔	startpoint-related-with
end-linker-objects	↔	endpoint-related-with
has-parts	↔	part-of
has-points	↔	point-part-of
has-startpoint	↔	startpoint-part-of-directed-element
has-endpoint	↔	endpoint-part-of-directed-element

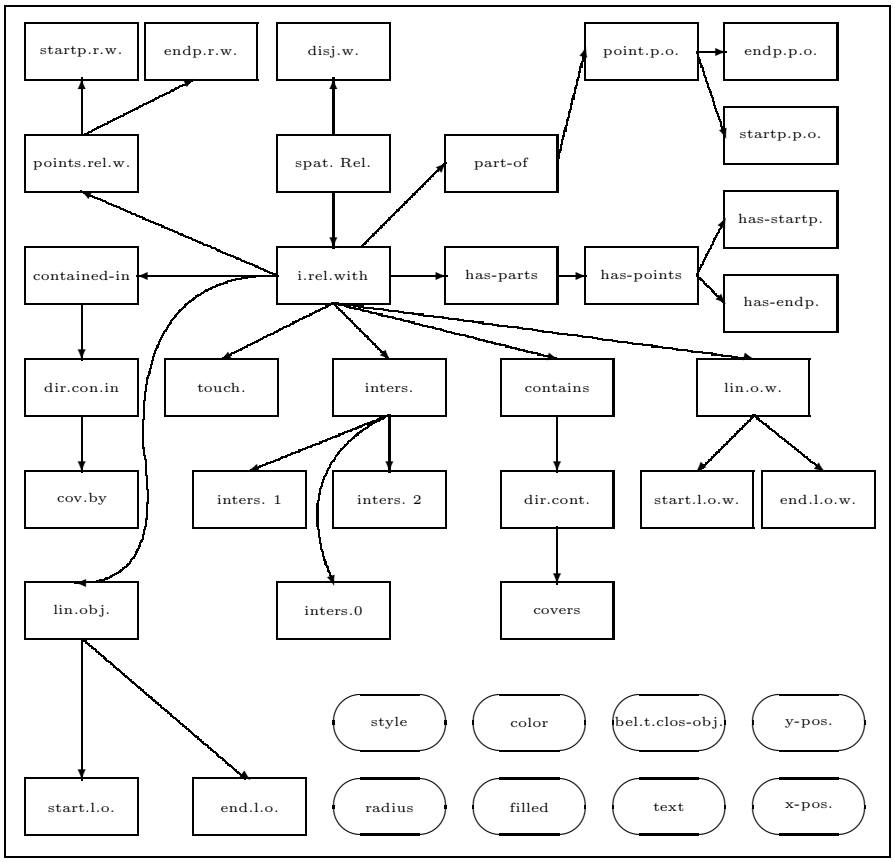


Abbildung 23: Die GENED-Rollenhierarchie

Die meisten Rollen entsprechen den in Kap. 3 diskutierten topologischen Relationen (teilw. sind die Namen leicht verschieden). Zusätzlich sind Relationen wünschenswert, die es erlauben, Aussagen über mögliche Verbindungen zwischen Objekten zu machen. Gilt $linker\_objects(A, B)$, so ist B ein Verbindungselement, daß Objekt A mit einem anderen Objekt verbindet. Für gerichtete Verbindungselemente (Instanzen des Konzeptes Directed Element) wie Pfeile o.ä. kann zusätzlich noch die Richtung der Verbindung relevant sein: gilt z.B. $start\_linker\_objects(A, B)$, so ist B ein Verbindungselement, daß an Objekt A beginnt. Da die Start- und Endpunkte eines solchen Verbindungsobjektes als eigene Objekte und auch CLASSIC-Individuen repräsentiert werden, kann für diese Punkte über die Relationen *start/endpoint_related_with* ermittelt werden, für welche Individuen ihr Verbindungselement die Verbindung herstellen. Ist das Verbindungselement ungerichtet (Strecke o.ä.), so können für diese Punkte nur Aussagen der Art *points_related_with* gemacht werden. Zudem weiß ein Verbindungsobjekt, welche Start- und Endpunkte zu ihm gehören (*has_start/endpoint*). Ist das Verbindungsobjekt jedoch ungerichtet, so gilt nur die Relation *has_points* (Oberrelation von *has_startpoint* und *has_endpoint*). Die Endpunkte können ihrerseits über die Inversen (*point_part_of* bzw. *startpoint_part_of_directed_element*, *endpoint_part_of_directed_element*) die korrespondierenden Verbindungsobjekte ermitteln, zu denen sie gehören.

4.5 Zustandsveränderungen und Nichtmonotonie

CLASSIC erlaubt es, Aussagen mit unvollständiger Information zu machen. So läßt sich z.B. ein Individuum erzeugen mit der Beschreibung

```
(cl-create-ind 'test '(and kreis (at-least 2 touches-objects)))
```

Diese Individuum berührt also zwei andere Individuen - welche dies sind, ist jedoch zur Zeit nicht bekannt. Die Information ist *unvollständig*. CLASSIC macht also nicht die aus Sprachen wie PROLOG bekannte *Closed World Assumption (CWA)*, welche besagt, daß alle nicht bekannte Information falsch ist. Stattdessen wird hier zugesichert, daß es diese Information gibt, was als *Metainformation* gedeutet werden kann: im Gegensatz zur direkten Kenntniss der Information ist dies eine Information über Information.

Eine Rolle kann sich in einem von zwei Zuständen befinden: „offen“ oder „geschlossen“. In diesem Beispiel ist die Rolle `touches-objects` offen - denn es steht noch Information aus. Erst wenn die Information vollständig ist, führt CLASSIC die meisten Inferenzen durch. Eine korrekte Klassifikation eines Individuums erfolgt erst, wenn alle Rolle geschlossen sind (das gilt natürlich nicht für Konzepte, welche ja statische Gebilde sind und somit keiner Veränderung unterworfen sind). Ein Beispiel verdeutlicht das Problem der *Nichtmonotonie* von Zustandsveränderungen der Wissensbasis (s. auch [4, Kap. 1.2], [18, Kap. 3.3.1], [21, S. 321]):

```
(cl-define-concept 'transition
                   '(and rechteck
                         ...
                         (all linked-over-with stelle)))

(cl-create-ind 'transition?
               '(and rechteck
                     (fills linked-over-with stelle1 stelle2 stelle3)))
```

Zu diesem Zeitpunkt subsumiert das Konzept Transition das Individuum `transition?`, wenn die Individuen `stelle1-3` tatsächlich Instanzen des Konzeptes Stelle sind (dies sei angenommen). Würde CLASSIC nun sofort folgern, daß `transition?` Instanz des Konzeptes Transition ist, so müsste diese Klassifizierung nach folgender Zusicherung zurückgenommen werden:

```
(cl-ind-add @transition? '(fills linked-over-with rechteck-123))
```

Ein Rechteck ist sicherlich keine Stelle, und somit wäre `transition?` keine Transition - zusätzliche Information hätte also einen bereits durchgeführten Schluß (nämlich die Klassifizierung) ungültig gemacht. Dies ist das Problem der *Nichtmonotonie*.

In CLASSIC wird das Individuum `transition?` im Gegensatz hierzu jedoch erst dann vollständig klassifiziert, wenn die Rolle `linked-over-with` geschlossen wird, also die Information gegeben wird, daß alle Information für diese Rolle vollständig ist. Erst dann kann eine Allquantifizierung oder `at-most`-Beschränkung geprüft werden - derartige Inferenzen werden daher von CLASSIC *verzögert* ausgeführt. Solange eine Rolle offen ist, können ihr beliebig Füller hinzugefügt und entfernt werden (wenn keine Inkonsistenz mit zuvor gemachten Zusicherungen entsteht). Bei einer geschlossenen Rolle ist eben dies nicht möglich: der Benutzer erhält eine Fehlermeldung, wenn er einer geschlossenen Rolle einen Füller hinzuzufügen oder zu entfernen versucht. Letztendlich bedeutet ein Schließen der Rolle `<role>` nur eine Konjunktion des bislang für das Individuum aggregierten CLASSIC-Ausdrucks

mit einer zusätzlichen `(at-most <n> <role>)`-Beschränkung, wenn daß Individuum zu diesem Zeitpunkt mit `<n>` Rollenfüllern bzgl. der Rolle `<role>` in Beziehung steht. Wenn jedoch von CLASSIC selbst eine `at-most`-Beschränkung einer Rolle für das Individuum abgeleitet werden kann, so braucht der Benutzer diese Rolle nicht manuell zu schließen.

So führt folgende Sequenz zu einem Widerspruch:

```
(cl-create-ind 'test '(and kreis (at-least 2 touches-objects)))

(cl-ind-close-role @test @touches-objects)
```

Weil keine Füller für die Rolle `touches-objects` eingetragen wurden, gibt es hier die Kontradiktion

```
(at-least 2 touches-objects)
          ⇕
(at-most 0 touches-objects)
```

CLASSIC unterscheidet zwischen „Told Information“ und „Derived Information“: so wurde die Aussage `(at-least 2 touches-objects)` nicht von CLASSIC abgeleitet, sondern vom Benutzer zugesichert - solche Information wird „Told Information“ genannt, die entsprechende Operation des Zusicherns oft als „Tell“-Operation bezeichnet. Derartige Informationen haben den Status einer eventuell zu revidierenden *Annahme*. Dies gilt auch für die Aussage `(at-most 0 touches-objects)`, welche durch das Schließen der Rolle für das Individuum erzeugt wurde. CLASSIC wird somit das Schließen dieser Rolle nicht zulassen - die Aussage `(at-most 0 touches-objects)` wird vom System zurückgezogen, die Rolle bleibt offen, und der Benutzer erhält eine Fehlermeldung aufgrund der (temporär) entstandenen Kontradiktion. Der Zustand des Individuums hat sich somit nicht verändert. Alternativ hierzu könnte vom Benutzer die Einschränkung `(at-least 2 touches-objects)` zurückgenommen werden, woraufhin CLASSIC ein Schließen der Rolle akzeptieren würde.

„Told Informations“ dienen als Antezedenzen der CLASSIC-Inferenzen, welche als Konsequenzen aus diesen „Derived Information“ ableiten. CLASSIC erlaubt nur die Zurücknahme von „Told Information“ - von CLASSIC selbst inferierte Information kann vom Benutzer nicht zurückgenommen werden, da andernfalls Inkonsistenzen in der Wissensbasis entstünden. Wird hingegen eine „Told Information“ zurückgenommen, so werden auch alle auf ihr basierenden Konsequenzen automatisch von CLASSIC zurückgezogen, da sonst die Wissensbasis ebenfalls wieder inkonsistent würde. Aufgabe eines *Begründungsverwaltungssystems (Truth Maintenance System, TMS)* ist es u.a., auf effiziente Weise die zurückzunehmenden Folgerungen zu identifizieren.

Der generelle Weg, einem existierenden Individuum Information zuzusichern, besteht im Aufruf der Funktion

```
(cl-ind-add <individual> <classic-description>)
```

womit also einem Individuum `<individual>` durch die `<classic-description>` (also ein Wort der CLASSIC-Grammatik) beschriebene Information zugesichert wird („Tell“). Somit ließe sich in Bezug auf obiges Beispiel formulieren:

```
(cl-ind-add @test '(fills touches-objects test-object-2))
```

Nun handelt es sich bei der Information, daß das Objekt `test` das Objekt `test-object-2` berührt, um „Told Information“. Diese läßt sich (wenn die Rolle `touches-objects` noch offen ist) zurücknehmen mit:

```
(cl-ind-remove @test '(fills touches-objects test-object-2))
```

Wäre die Rolle bereits geschlossen, so ließe sie sich mit

```
(cl-ind-unclose-role <individual> <role>)
```

wieder öffnen. Geschlossen wird sie entweder von CLASSIC oder manuell per

```
(cl-ind-close-role <individual> <role>)
```

Wird z.B. ein Individuum mit der Zusicherung erzeugt, daß es zwei andere Individuen berührt wie in

```
(cl-create-ind 'test2 '(and kreis (at-most 2 touches-objects)))
```

und werden Rollenfüller eingetragen mit

```
(cl-ind-add @test2 '(fills touches-objects stelle1 stelle2))
```

so kann CLASSIC ableiten, daß die Rolle `touches-objects` keine weiteren Füller mehr haben kann und somit geschlossen ist. Das Individuum kann dann z.B. als Instanz des Konzeptes Berührt-Ausschließlich-Stellen klassifiziert werden.

4.6 Beschränkungen der CLASSIC-Sprache

Diverse Beschränkungen sind bereits zur Sprache gekommen: u.a. gibt es in CLASSIC keine logische Disjunktion.[8] Ein Beispiel ist die FOL-Definition des Konzeptes Netzkante:

$$\begin{aligned} netzkante(x) \quad \Leftrightarrow \quad & (arrow(x) \vee directed\_spline\_chain(x)) \wedge \\ & (\forall\, y : touches(x, y) \Rightarrow (stelle(y) \vee transition(y))) \end{aligned}$$

Dennoch kann man sich mit einem Trick behelfen: man bildet ein Konzept Stelle-oder-Transition und macht es zum gemeinsamen Elternkonzept von Stelle und Transition. Wenn nun noch angenommen wird, daß Stelle und Transition das gemeinsame Elternkonzept vollständig partitionieren, so kann es als Vereinigung beider Konzepte betrachtet werden. Dies kann von CLASSIC nur dann sichergestellt werden, wenn u.a. alle Unterkonzepte (von denen die Vereinigung gebildet werden soll) als `disjoint-primitive` einer Gruppe definiert worden sind und das Elternkonzept ebenfalls primitiv ist. Auch kann die vollständige Partitionierung des gemeinsamen Elternkonzeptes durch diese Konzepte nur dann funktionieren, wenn niemals Individuen des Elternkonzeptes direkt, sondern stets nur Instanzen der Kindkonzepte erzeugt werden. Zudem muß die Taxonomie entsprechend „künstlich“ konstruiert

[8] Wahrscheinlich, weil Disjunktionen in der Regel Bäume aufspannen und somit exponentielle Algorithmen verursachen.

werden. Wäre das als Vereinigung dienende Elternkonzept jedoch nicht primitiv, so könnte ein Individuum wiederum Instanz dieses Konzeptes durch Klassifikation werden, ohne auch Instanz eines der Kindkonzepte zu werden: somit ließe sich das Elternkonzept nicht als Vereinigung der Kindkonzepte nutzen, sondern nur als Obermenge der Vereinigungsmenge. Letztendlich ist die entsp. Anwendung für die adäquate und korrekte Verwendung einer solch aufgesetzten Semantik verantwortlich. Für nicht-primitive Konzepte ist hingegen eine Konstruktion denkbar, die, sobald ein Individuum zur Instanz einer der Konzepte klassifiziert wird, von denen die Vereinigung zu bilden ist, dem Individuum das die Vereinigung repräsentierende „Oder"-Konzept per „Tell" zusichert. Diese Konstruktion läßt sich durch eine *Regel* realisieren. Alle diskutierten Vorschläge sind höchst unvollständig und müssen daher mit großer Vorsicht angewandt werden.

Auch die Benutzung von Rollen ist stark eingeschränkt: so lassen sich zwar „Number" und „Value Restrictions" zur Einschränkung benutzen, fast unverzichtbar sind aber auch Aussagen der Art

```
(exactly 1 intersects-objects kapazitaets-label)
```

womit also Kardinalität und Konzept der Füller eingeschränkt werden, was sich in CLASSIC ebenfalls nicht formulieren läßt. Hier ist nicht gemeint, daß es nur einen Füller der Rolle `intersects-objects` geben darf und dieser Instanz des Konzeptes Kapazitäts-Label sein muß, sondern daß es unter allen Füllern der Rolle `intersects-objects` (über deren Anzahl nichts gesagt wird) genau einen geben muß, der Instanz des Konzeptes Kapazitäts-Label ist. Es handelt sich also nicht um eine Definition der Art

```
(and (all intersects-objects kapazitaets-label)
     (exactly 1 intersects-objects))
```

Stattdessen ist bzgl. des Individuums x folgendes gemeint:

`(at-most <n> <role> <concept>)`
$$\Leftrightarrow$$
$$|\{y : role(x,y) \wedge concept(y)\}| \leq n$$

`(at-least <n> <role> <concept>)`
$$\Leftrightarrow$$
$$|\{y : role(x,y) \wedge concept(y)\}| \geq n$$

und als bequeme Abkürzung

`(exactly <n> <role> <concept>)`
$$\Leftrightarrow$$
$$|\{y : role(x,y) \wedge concept(y)\}| = n$$

Solche Rollen werden von Wissensrepräsentationssprachen wie LOOM angeboten und dort *qualifizierte Unterrollen* genannt. So wird von Systemen wie LOOM auch die Subsumptionsbeziehung zwischen Relationen berechnet - derartige Rollen sind also nicht mehr primitiv (im Sinne der primitiven CLASSIC-Rollen), sondern können ähnlich wie Konzepte vom System klassifiziert werden.

Wie dargestellt wünscht sich aber auch der CLASSIC-Benutzer diese Ausdrucksmöglichkeiten - sofern er auf Subsumption und Vollständigkeit verzichtet,

kann mit Hilfe eines prozeduralen Vorgehens eine zumindest für die Konzeptdefinition verwendbare Hilfskonstruktion geschaffen werden. Ihre Implementation wird weiter unten erläutert - letztendlich handelt es sich jedoch um eine außerhalb der Semantik der CLASSIC-Sprache liegende, höchst unvollständige Konstruktion. Diese „Erweiterung“ wird im folgenden als *pseudo-qualifizierte Unterrolle* bezeichnet.

4.7 Wissensbasen

Das CLASSIC-Handbuch definiert eine Wissensbasis als eine Sammlung von Konzepten, Individuen, Rollen, Regeln und Testfunktionen.

4.7.1 Eine Wissenbasis für S/T-Netze

Als komplexeres Beispiel soll nun eine Wissensbasis für S/T-Netze angegeben werden, wofür ihre Definition laut [11, S. 71] zitiert sei:

Definition 7 (S/T-Netz) *Ein 6-Tupel $N = (S, T, F, K, W, M)$ heißt Stellen/Transitions-Netz (S/T-Netz), falls gilt:*

1. *(S, T, F) ist ein Netz aus Stellen S und Transitionen T.*
2. *$K :\rightarrow \mathbb{N} \cup \{\omega\}$ erklärt eine (möglicherweise unbeschränkte) Kapazität für jede Stelle.*
3. *$W : F \rightarrow \mathbb{N} - \{0\}$ bestimmt zu jedem Pfeil des Netzes ein Gewicht.*
4. *$M :\rightarrow \mathbb{N} \cup \{\omega\}$ ist eine Anfangsmarkierung, die die Kapazitäten respektiert, d.h., für jede Stelle $s \in S$ gilt : $M(s) \leq K(s)$.*

Für S/T-Netze sind die Begriffe *Markierung* und *Aktivierung* wichtig:

Definition 8 (Markierung, aktive Transition) *Sei N ein S/T-Netz.*

1. *Eine Abbildung $M : S_N \rightarrow \mathbb{N} \cup \{\omega\}$ heißt Markierung von N, falls für jede Stelle $s \in S_N$ gilt: $M(s) \leq K_N(s)$. Sei M eine Markierung von N.*
2. *Eine Transition $t \in T_N$ heißt M-aktiviert, falls gilt: $\forall\, s \in {\bullet t} : M(s) \geq W_N(s,t)$ und $\forall\, s \in t{\bullet} : M(s) \leq K_N(s) - W_N(t,s)$.*

Dabei bezeichnet

$$\bullet t \equiv \{s : (s,t) \in F\}$$

und

$$t\bullet \equiv \{s : (t,s) \in F\}$$

Um einen Eindruck der üblichen Visualisierungen der S/T-Netze zu vermitteln, zeigt Abb. 24 zwei bekannte Netz aus [10, S. 39, S. 197]. Dabei sind die kleinen schwarzen Kreise Marken (ihre Verteilung auf die Stellen bildet die Markierung M), große Kreise sind Stellen, und Rechtecke Transitionen. Die Pfeile sind Netzkanten. Die Zahl- bzw. Textelemente (Kapazitäts-Label) schneiden mit ihren kleinsten umschließenden

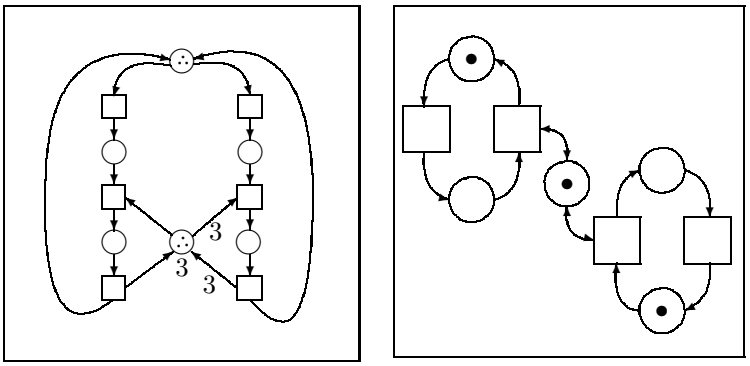


Abbildung 24: Leser-Schreiber- und Erzeuger-Verbraucher-Problem

Rechtecken jeweils eine Netzkante oder eine Stelle und geben so die Funktionen W und K an. Netzkanten ohne Kapazitäts-Label haben die Kapazität 1, Stellen ohne Kapazitäts-Label hingegen die Kapazität $\omega = \infty$. Transitionen haben keine Kapazität.

Eine Wissensbasis für S/T-Netze wird auf Konzepten für allgemeine Petrinetze aufbauen. In Kap. 2 wurden schon einige hierfür benötigte Konzepte definiert, jedoch zyklisch. Im folgenden wird teilweise das Makro `define-concept` statt `cl-define-concept` benutzt, daß einige Syntax-Erweiterung zu bieten hat, die man KRSS-Syntax nennt: u.a. `(some <role>)` ≡ `(at-least 1 <role>)`, `(none <role>)` ≡ `(at-most 0 <role>)`, `(exactly <n> <role>)` ≡ `(and (at-least <n> <role>) (at-most <n> <role>))`. Da es sich um ein Makro handelt, kann auch das Quotieren („‘“) konstanter Ausdrücke entfallen. Einige Sprachoperatoren sind jedoch nur über `cl-define-concept` nutzbar (wenn z.B. Testfunktionen in die Konzepdefinitionen eingebracht werden müssen). Zunächst werden nun Konzepte für Stelle, Transition und Netzkante definiert. Instanzen von Konzepten, die mit einem `?` enden, haben einen unsicheren Status: so ist für Instanzen von Stelle? noch nicht sicher, ob sie tatsächlich Stellen sind. Diese Hilfskonzepte dienen dazu, Zyklen zu vermeiden und die Taxonomie zu strukturieren.

```
(define-concept stelle-oder-transition?
   (and basic-thing
        (none contained-in-objects)
        (some touches-objects)
        (all touches-objects-directed-element tt-verbinder)))

(define-concept stelle?
   (and kreis stelle-oder-transition?))

(define-concept transition?
   (and rechteck stelle-oder-transition?
        (none contains-objects)
        (none intersects-objects)
        (at-least 2 linked-over-with)
        (at-least 1 start-linker-objects)
        (at-least 1 end-linker-objects)))

(define-concept stelle
   (and stelle?
        (all linked-over-with transition?)))
```

```
(define-concept transition
   (and transition?
        (all linked-over-with stelle?)))

(define-concept netzkante
   (and tt-verbinder
        (all touches-objects stelle-oder-transition?)))

(define-concept simple-stelle
   (and stelle
        (none contains-objects)
        (none intersects-objects)))
```

Bei der Rolle `touches-objects-directed-element` handelt es sich um eine Spezialisierung der Rolle `touches-objects`: In diese Rolle werden nur Instanzen des Konzeptes Directed-Element eingetragen. Da es sich jedoch um eine anwendungsspezifische Rolle handelt, wird sie von GENED nicht wie die anderen Rollen korrekt zugesichert (die ja allgemein, also generisch sind). CLASSIC müsste also selbst eine Partitionierung der Oberrelation *touches_objects* bezüglich der Unterrelationen vornehmen - da aber alle Rollen primitiv sind und somit keine Subsumptionsbeziehungen berechnet werden, kann CLASSIC eben dies nicht tun. Stattdessen werden die Füller prozedural berechnet und von einer sog. *Füllerregel* in die ihrem Konzept entsprechende Unterrolle eingetragen - konzeptionell würde für die Definition dieses Konzeptes jedoch eine *qualifizierte Unterrolle* benötigt werden. Die Implementation der hier verwendeten *pseudo-qualifizierten Unterrollen* wird später erläutert.

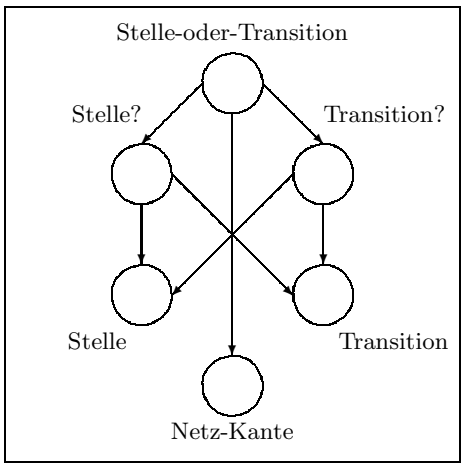


Abbildung 25: Abhängigkeitsgraph

Ein TT-Verbinder ist ein Directed-Element (z.B. ein Pfeil o.ä.), daß die Verbindung zwischen zwei Objekten dadurch herstellt, daß es beide berührt („Touch/Touch“). Außerdem werden in die Rolle `start-linker-objects` bzw. `end-linker-objects` alle Verbinder (Linker) eingetragen, die an dem entsp. Objekte starten bzw. enden. Den Abhängigkeitsgraphen dieser Definitionen zeigt Abb. 25. Für alle eindimensionalen Objekte werden zusätzlich deren Start- und Endpunkte als Individuen repräsentiert. Auch werden alle topologischen Relationen für diese berechnet; sie sind Instanzen des Konzeptes Info-Point. Das Konzept TT-Verbinder ließe sich daher so definieren:

```
(define-concept verbinder-punkt
  (and info-point
       (exactly 1 point-part-of)))

(define-concept t-verbinder-punkt
  (and verbinder-punkt
       (exactly 1 touches-objects)))

(define-concept verbinder
  (and directed-element
       (exactly 2 points-related-with)))

(define-concept tt-verbinder
  (and verbinder
       (all has-points t-verbinder-punkt)))
```

Der Leser mag sich fragen, warum bei der Definition von Stelle nicht wie bei Transition die Restriktion `(and (none intersects-objects) (none contains-objects))` erscheint. Stattdessen erscheint sie in der Definition des Konzeptes Simple-Stelle. Der Grund dafür ist, daß später Spezialisierungen von Stelle für Stellen mit Marken und Stellen mit Kapazität formuliert werden. Für diese Konzepte gilt obige Restriktionen nämlich nicht, und per Vererbung würde ein Widerspruch entstehen. Fraglich ist auch die Definition des Konzeptes Stelle-oder-Transition?: warum wurde anstelle der pseudo-qualifizierten Unterrolle nicht die Restriktion `(and (all touches-objects directed-element) (some touches-objects))` verwendet? Wie erwähnt, werden alle Start- und Endpunkte eindimensionaler Objekte ebenfalls als Individuen repräsentiert, und daher befinden sich diese Individuen ebenfalls in der Rolle `touches-objects`. Eine Folge ist, daß eine Instanz auch dann z.B. zur Stelle klassifiziert wird, wenn Instanzen andere Konzepte das Individuum berühren. Stattdessen müsste eine Restriktion der Art `(all touches-objects (or info-point directed-element))` Teil der Definition sein, was in CLASSIC nicht formulierbar ist.

Wiederum kann ein Kompositionsobjekt gebildet und als Petrinetz klassifiziert werden:

```
(define-concept petrinetz
   (and composite-thing
        (at-least 5 has-parts)
        (some has-parts-stelle)
        (all has-parts-rechteck transition)
        (all has-parts-directed-element netzkante)))

(define-concept simples-petrinetz
   (and petrinetz
        (all has-parts-kreis simple-stelle)))
```

Die Restriktion `(all has-parts-kreis stelle)` ist nicht Teil der Definition von Petrinetz, da eine Spezialisierung hiervon, nämlich S/T-Petrinetz (s.u.) auch Marken als Teilkomponenten enthält. Die Definition von Petrinetz bleibt unbefriedigend, da keine Disjunktion formuliert werden kann. Alle `has-parts-...`-Rollen sind ebenfalls pseudo-qualifizierte Unterrollen. Andere interessante Konzepte sind:

```
(define-concept start-stelle
   (and stelle
        (none start-linked-over-with)))
```

```
(define-concept end-stelle
   (and stelle
        (none end-linked-over-with)))

(define-concept normale-stelle
   (and stelle
        (some start-linked-over-with)
        (some end-linked-over-with)))
```

4.7.2 Weitere Konzepte

Im linken Petrinetz von Abb. 24 sind Textelemente ersichtlich, die eine natürliche Zahl als Text haben und deren kleinste umschließenden Rechtecke jeweils Stellen oder Netzkanten schneiden.[9]. Ein solches Textelement soll vom Konzept Kapazitäts-Label sein. Dafür ist es nötig, zu erkennen, ob ein Text-String (der von GENED als Attribut `text-string` für das Individuum zugesichert wird) eine Zahl ist. Dies liegt außerhalb des in CLASSIC durch normale Konzeptdefinitionen Formulierbaren. Es wird also ein *prozedurales Testprädikat* (das hier den Namen `string-is-integer-p` hat) benötigt. Prinzipiell könnte GENED für Zahl-Textelemente die entsp. Konzepte zusichern, jedoch ist dieser Ansatz für einen Grafikeditor mit nicht festgelegtem Einsatzgebiet nicht allgemein genug. Letztendlich könnte eine Anwendung verlangen, daß ein Textelement genau drei Ypsilons enthält. Damit solch ein Konzept für das Textelement selbst zugesichert werden könnte, müsste GENED mit einer entsprechenden Erkennungsroutine versehen, also umprogrammiert werden. Daher wird das Problem in die Konzepdefinitionen der Wissensbasis verlagert, die ja ohnehin austauschbar ist.

Dem prozeduralen Testprädikat `string-is-integer-p` wird also der String bzw. das Textelement (ein „Host Individual“, also Objekt der Sprache LISP) übergeben, und dann versucht, den Text in eine Zahl zu konvertieren und `integerp` darauf anzuwenden. Dies leistet die Konstruktion `(test-h string-is-integer-p)` („h“ für „Host Individual“). Testfunktionen sind eine sehr mächtige Erweiterung, sie liegen jedoch außerhalb der Semantik von CLASSIC.

Eine Testfunktion ist eine *Black Box:* die Subsumptionsfrage für allgemeine LISP-Testfunktionen ist unentscheidbar. Ebenso lassen sich Testfunktionen für CLASSIC-Individuen schreiben, wozu der Termbildungsoperator `test-c` benutzt wird.

```
(defun string-is-integer-p (string)
  (integerp (read-from-string string)))

(cl-define-concept 'kapazitaets-label
   '(and text
         (at-least 1 intersects-objects)
         (all text-string
              (and (test-h string-is-integer-p)))))
```

Nun lassen sich hierauf aufbauend weitere Konzepte für Stellen mit Marken, Stellen mit Kapazität etc. definieren. Für das Verständnis der folgenden Konzepte sind einige Erläuterungen notwendig: `filled-bool` ist ein Attribut, das von GENED aus zugesichert wird und angibt, ob das korrespondierende Grafikobjekt mit Farbe gefüllt ist oder nicht (`t` bzw. `nil`). `radius-real` ist ein Attribut und gibt den Radius eines Kreis-Individuums an. CLASSIC bietet `min-` und `max`-Operatoren (also

[9] Alle topologischen Relationen für Textelemente sind anhand der „Bounding Box“ definiert, siehe Kap. 3.

Intervalle für Zahlindividuen), für die auch Subsumption berechnet wird. Number ist ein eingebautes Konzept und bezeichnet den gleichnamigen COMMON LISP-Typ. Der `fills`-Operator drückt aus, daß ein bestimmtes Individuum in einer Rolle enthalten sein muß. Dieses Individuum muß jedoch schon zur Zeit der Konzeptdefinition existieren, wodurch die Anwendung diese Operators stark beschränkt ist. Eine l-aktive-Transition ist eine Transition t, für die gilt:

$$\forall\, s \in \bullet t : M(s) \geq W_N(s,t)$$

Genau dies wird von der Testfunktion `genug-input-marken?` geprüft.

Ein r-aktive-Transition ist eine Transition t, für die gilt:

$$\forall\, s \in t\bullet : M(s) \leq K_N(s) - W_N(t,s)$$

Dies wird von der Testfunktion `output-moeglich?` geprüft. Wenn eine Transition Instanz beider Konzepte ist, ist sie laut Def. 8 *aktiviert* und kann *schalten.*

Testfunktionen sind dreiwertig: sie liefern jeweils `t`, `nil` oder `?`. Dabei bedeutet `t`, daß das Individuum konsistent mit der Testfunktion ist, `nil`, daß es inkonsistent ist, und `?`, daß es momentan konsistent ist, aber inkonsistent werden könnte, wenn Information hinzugefügt wird. Der Wert einer Testfunktion darf, solange noch Information hinzugefügt wird, niemals von `t` nach `nil` gehen, sondern stets nur von `nil` oder `?` nach `t` (Monotonie). In dieser Wissensbasis müssen daher alle betroffenen Rollen geschlossen sein, bevor die Testfunktion zur Anwendung kommt. Dies wird ebenfalls prozedural geprüft, und zwar mit der Testfunktion `cl-test-closed-roles`. Die Testfunktionen sind recht kompliziert, weswegen sie an dieser Stelle nicht erscheinen.

```
(define-concept stelle-mit-kapazitaet
   (and stelle
        (exactly 1 intersects-objects)
        (all intersects-objects kapazitaets-label)))


(cl-define-concept 'marke
   '(and kreis
         (all contained-in-objects stelle)
         (fills filled-bool t)
         (all radius-real
              (and number
                   (min 2.0)
                   (max 10.0)))))


(define-concept stelle-mit-marken
   (and stelle
        (some contains-objects)
        (all contains-objects marke)))


(define-concept netzkante-mit-kapazitaet
   (and netzkante
        (exactly 3 intersects-objects) ; Start- und Endpunkt und Label
        (exactly 1 intersects-objects-kapazitaets-label)))
```

```
(cl-define-concept 'l-aktive-transition
   '(and transition
         (all start-linked-over-with stelle-mit-marken)
         (all end-linker-objects netzkante)
         (test-c cl-test-closed-roles?
                 (end-linker-objects start-linked-over-with))
         (test-c genug-input-marken?)))

(cl-define-concept 'r-aktive-transition
   '(and transition
         (all end-linked-over-with stelle)
         (all start-linker-objects netzkante)
         (test-c cl-test-closed-roles?
                 (start-linker-objects end-linked-over-with))
         (test-c output-moeglich?)))

(define-concept aktive-transition
   (and l-aktive-transition
        r-aktive-transition))
```

Eine besondere Teilmenge der S/T-Netzen bilden die B/E-Netze (Bedingungs-Ereignis-Netze) - bei ihnen handelt es sich um spezielle S/T-Netze, für die gilt (s. [10, S. 38]):

$$\forall\, s \in S : K(s) = 1 \wedge \forall\, (x, y) \in F : W(x, y) = 1$$

Alle Stellen und Netzkanten eines B/E-Netzes haben also eine maximale Kapazität von 1.

```
(define-concept s/t-petrinetz
   (and petrinetz
        (some has-parts-stelle-mit-marken)))

(define-concept b/e-petrinetz
   (and s/t-petrinetz
        (all has-parts-stelle b/e-stelle)
        (all has-parts-directed-element b/e-netzkante)))
```

Ein Petrinetz ist somit ein S/T-Petrinetz, sobald es nur eine Stelle als Komponente hat, die Marken enthält. Die Definitionen von B/E-Stelle und B/E-Netzkante sollen hier nicht weiter diskutiert werden. Eine Konflikt-Stelle ist eine Stelle eines S/T-Netzes, von der mind. 2 Netzkanten ausgehen. Eine Marke, die solch eine Stelle verlassen will, muß sich also „entscheiden“, welchen Weg sie wählt.

```
(define-concept konflikt-stelle
   (and stelle
        (all part-of s/t-petrinetz)
        (at-least 2 end-linked-over-with)))
```

Es sollte deutlich geworden sein, daß die hier diskutierten Konzeptdefinitionen viele Kompromisse aufgrund eingeschränkter Ausdrucksfähigkeit erfordern.

4.8 Das Makro `defqualifiedsubrole`

Die Nützlichkeit der qualifizierten Unterrollen wurde bereits hinlänglich diskutiert. Es stellt sich nun die Frage, wie die in der Petrinetzwissensbasis verwendete Hilfs-

konstruktion der pseudo-qualifizierten Unterrolle implementiert werden könnte. Für Konzeptdefinitionen ist es ausreichend, wenn das System eine Konstruktion anbietet, die automatisch Rollenfüller in einer bestimmten Oberrolle betrachtet und diese Füller anhand ihrer momentanen Konzepte in dementsprechende Unterrollen einträgt. Die Konstruktion stellt somit sicher, daß in jeder dieser pseudo-qualifizierten Unterrollen stets nur die Füller der Oberrolle stehen, die die entsprechende Konzepteinschränkung für die Unterrolle erfüllen.

Als konkretes Beispiel soll nun eine pseudo-qualifizierte Unterrolle der Rolle `intersects-objects` diskutiert werden, deren Füller ausschließlich Instanzen des Konzeptes Kapazitäts-Label sind. Diese Rolle heißt dann `intersects-objects--kapazitaets-label`; ihre Füller müssen anhand der Rolle `intersects-objects` prozedural bestimmt und eingetragen werden, denn alle potentiellen Füller sind hier zu finden.

Ein Gedankenbeispiel verdeutlicht eine mögliche Vorgehensweise eines Algorithmus':

Gilt für eine Instanz `label1` des Konzeptes `g-text`, daß sie die beiden Individuen `x1` und `x2` schneidet (es gilt somit für `x1` und `x2` jeweils `(fills intersects-objects label1)`), und wird die Instanz `label1` nun von CLASSIC zur Instanz des Konzeptes Kapazitäts-Label klassifiziert, so muß in beiden Individuen `x1` und `x2` diese Instanz in die Rolle `intersects-objects-kapazitaets-label` eingetragen werden. Dies könnte erreicht werden durch eine *Regel*, die auf Instanzen des Konzeptes Kapazitäts-Label feuert - diese Regel wird also aktiviert, sobald `label1` von CLASSIC zum Kapazitäts-Label klassifiziert wird. Sie ruft dann eine mit ihr assoziierte Funktion mit dem entsprechenden Individuum (hier also `label1`) als Argument auf. Diese Funktion extrahiert dann alle Füller der Oberrolle `intersects-objects` des Individuums `label1` (in der ja `x1` und `x2` stehen) und trägt diese dann in die *Inverse* der Rolle `intersects-objects-kapazitäts-label` des Individuums `label1` ein. Diese Inverse hat z.B. den Namen `intersects-objects-kapazitaets-label-inverse`, und daher erscheint in der Rolle `intersects-objects-kapazitäts-label` der Individuen `x1` und `x2` korrekterweise (durch Propagierung) jeweils das Individuum `label1` - die Konstruktion hat also ihren Zweck erfüllt, da für eine Rolle mit inverser Rolle immer beide Individuen aktualisiert werden, wenn ein Individuum zum anderen in Beziehung gesetzt wird.

In CLASSIC wird solch eine Regel als *Füllerregel (Filler Rule)* bezeichnet, denn sie berechnet Rollenfüller (in diesem Fall für die Rolle `intersects-objects--kapazitaets-label-invers`) für ein Individuum (hier `label1`). Dabei muß eine solche Regel stets - in welchem Zustand das Individuum, auf dem sie feuert, sich auch befindet - die selben Füller erzeugen. Das ist in dem oben skizzierten Algorithmus der Fall, denn die Regel feuert erst, wenn die Oberrolle (hier `intersects-objects`) geschlossen ist, womit keine weiteren Füller verfügbar werden können.

Das Aufsetzen der Füllerregel und die Definition der Unterrolle sowie ihrer notwendigen Inversen geschieht durch das Makro `defqualifiedsubrole`. Seine Syntax ist:

```
(defqualifiedsubrole <role> <concept-qualification>)
```

Die hier diskutierte pseudo-qualifizierte Unterrolle kann mit seiner Hilfe wie folgt definiert werden:

```
(defqualifiedsubrole intersects-objects kapazitaets-label)
```

Die Namen lassen sich jedoch auch über Schlüsselwort-Parameter spezifizieren und somit adäquater vergeben.

5 CLIM - COMMON LISP Interface Manager

5.1 Motivation

Für ein System wie einen Grafikeditor ist es unerläßlich, dem Benutzer eine grafische Oberfläche zu bieten. Diese sollte möglichst *direktmanipulativ* sein: der Benutzer soll das Gefühl haben, er könne die Objekte auf dem Schirm anfassen und - wie in der realen Welt - beliebig manipulieren, ohne sich Gedanken über die Auswahl und Planung hierzu nötiger Menüpunkte bzw. Sequenzen zu machen. Die Benutzung der Maus spielt dabei eine zentrale Rolle. Das hier diskutierte CLIM-System unterstützt benutzerfreundliche Interaktionsformen durch diverse Konzepte, die im weiteren Verlauf dargestellt werden. Zudem soll dem Leser die Eignung der Sprache LISP für die Oberflächenprogrammierung verdeutlicht werden.

5.2 Was ist CLIM?

Die GENED-Benutzeroberfläche wurde mit CLIM erstellt. Bei CLIM handelt es sich um ein standardisiertes, objektorientiertes Oberflächengestaltungssystem. CLIM schirmt den Entwickler durch Errichtung diverser sehr hoher Abstraktionsbarrieren nahezu vollständig vom darunterliegenden Wirtsfenstersystem ab: der Preis derartig hoher und mächtiger Abstraktionen ist allerdings große Ineffizienz. Die Anzahl und Qualität der von CLIM angebotenen Basisdienste ist überwältigend, und die *Orthogonalität* LISP-typisch sehr hoch. Beim Programmieren wird man also kaum mit zeitaufwendigen Details belästigt und kann sich so auf das Wesentliche konzentrieren. Oberflächen hoher Qualität und Ästhetik lassen sich (nach einer gewissen Einarbeitungszeit) in kürzester Zeit erstellen.

CLIM erlaubt aber auch, auf diverse eingebaute und teure Dienste zu verzichten und bietet zudem direkten Zugriff auf das Wirtsfenstersystem: so erschien es bei der Programmierung von GENED sinnvoll, die Behandlung einiger Mausereignisse selbst zu übernehmen, was mit einer Umgebung namens `tracking-pointer` erreicht wurde.

Dieses Kapitel stellt nur einige der wichtigsten CLIM-Schlüsselkonzepte anhand von Beispielen vor. Das gesamte System ist wiederum viel zu komplex, um auch nur annähernd in dieser Arbeit dargestellt zu werden. Einige Details in der Benutzung von CLIM werden in Kap. 6 vorgestellt. Dort finden sich auch Bilder von GENED, womit die Erscheinungsform von CLIM-Applikationen verdeutlicht wird.

5.3 Portabilität durch Abstraktion

CLIM schirmt den Entwickler durch die Errichtung sehr hoher Abstraktionsbarrieren fast vollständig vom Wirtsfenstersystem ab. Der Entwickler programmiert seine Oberfläche mit Hilfe der angebotenen hochsprachlichen Abstraktionen. Das darunterliegende Fenstersystem (z.B. Windows oder X/Motif) ist jedoch letztendlich für die Darstellung der GUI-Elemente (Graphical User Interface) verantwortlich: alle Abstraktionen müssen auf diese Ebene heruntergebrochen werden, was vom sog. *Rahmenverwalter (Frame Manager)* vorgenommen wird. Dabei können verschiedene Fenstersysteme Ziel dieser Abbildung sein, womit ein immer wichtiger werdendes Ziel erreicht wird: Portabilität. Der gleiche Ansatz wird auch bei Systemen wie StarView von StarDivision oder SmallTalk von ParcPlace-Systems verfolgt.

Mit heutigen Fenstersystemen scheint diese Abbildung mit gerade noch angemessenem Aufwand implementierbar - so wird es jedoch mit fortschreitender Entwicklung der Fenstersysteme schwieriger werden, deren Funktionalität und auch Bedienkonzept angemessen auszunutzen und über diverse Abstraktionsebenen zu retten. Schließlich muß eine Art gemeinsamer Kern aller Oberflächen angeboten werden, womit kein Raum für Spezialisierung einzelner Systeme bleibt. Ohne eine Angleichung bzw. Standardisierung der einzelnen Fenstersysteme untereinander wird ein zukünftiges CLIM also entweder stark unter den Fähigkeiten der Wirtsfenstersysteme bleiben oder aber zu völlig untypischen Bedienkonzepten führen, wie es teilweise schon jetzt der Fall ist (aufgrund der Symbolics LISP Machine-Herkunft). Dies könnte die Akzeptanz und Verbreitung deutlich mindern.

5.4 Würdigung einiger CLIM-Dienste

CLIM bietet selbstverständlich alle möglichen Grafikprimitive, wie Kreise, Ellipsen, Polygone, Strecken, Pfeile und sogar Bezier-Ketten. Flächige Objekte können mit beliebigen Füllmustern gefüllt und Pinsel selbst definiert werden. Komfortabel sind auch CLIM-Makros, die dynamische Zeichenumgebungen etablieren, so daß alle in diesem Kontext ausgeführten Zeichenoperationen automatisch transformiert werden (Rotationen, Skalierungen etc.). Die Transformationsmatrizen können dabei vom Entwickler definiert und miteinander komponiert werden. Eine andere Umgebung ermöglicht es, alle Grafikausgaben nach PostScript zu konvertieren und in eine Datei oder auf den Drucker umzulenken - so erhält man einen Ausdruck hoher Qualität, ohne auch nur zu wissen, was PostScript eigentlich ist (im Anhang finden sich klassifizierte Petrinetze, die auf diese Art ausgedruckt wurden). Diverse nützliche Standarddialoge sind schon vorgesehen: so z.B. ein komfortabler Dateiauswähler. Er läßt sich mit einem einzigen Funktionsaufruf öffnen (s. Abb. 28). Es lassen sich hierarchische Menüs erzeugen, deren Einträge beliebige Grafiken sein können. Grafiktext, verschiedene Schriftarten und Grafikbitfelder (Bitmaps) sind auch einsetzbar; CLIM kann Graphen und Tabellen formatieren. Leider macht das Handbuch (s. [17]) einige Versprechungen, die von der derzeitigen CLIM-Implementation (2.0) nicht erfüllt werden. So schreibt das Handbuch:

> „A region is an object that denotes a set of points in the plane. Regions include their boundaries, that is, they are closed. Regions have infinite resolution.“

Tatsächlich existieren Region-Unterklassen wie Polygon, Circle etc., aber die im Handbuch versprochenen generischen Prädikate wie `region-contains--position-p, region-contains-region-p` etc. sind nur für Rechtecke definiert, so daß sie leider nicht zur Implementation des Geometriemoduls verwendet werden konnten.

5.5 Die Anwendungsschleife

Eine Benutzer muß mit einer Anwendung in Interaktion treten - hierzu wird von der Anwendung prinzipiell eine Schleife der in Abb. 26 dargestellten Art zyklisch durchlaufen. Die Anwendung muß auf momentan zu berücksichtigende Kommandos des Benutzers reagieren (READ bzw. ACCEPT), diese entsprechend ausführen (EVAL) und schließlich den neuen internen Zustand der Applikation reflektieren (PRINT bzw. PRESENT), also den Schirm entsprechend aktualisieren, da es anderseits zu Inkonsistenzen zwischen anwendungsinterner Objektrepräsentation und

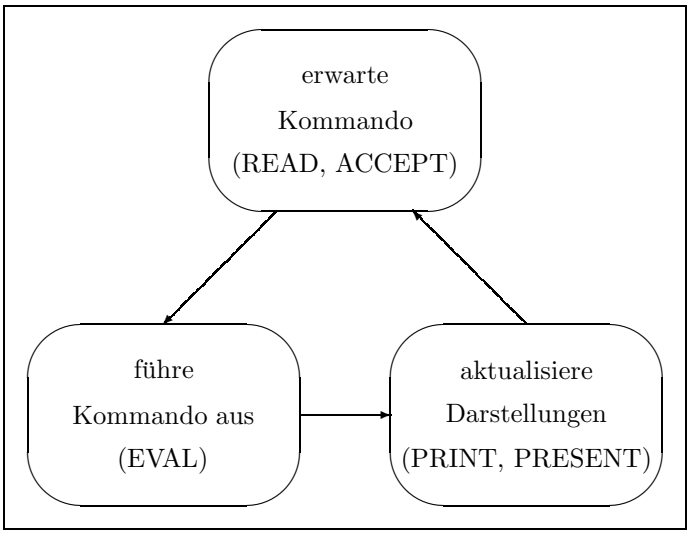


Abbildung 26: Die Anwendungsschleife

den dargestellten Visualisierungen dieser Objekte kommen könnte. Welche Kommandos momentan von der Anwendung verarbeitet bzw. vom Benutzer eingegeben werden können, bestimmt der *Eingabekontext* - wenn eine Anwendung z.B. die Betätigung eines virtuellen OK-Schalters per Maus verlangt, wird sie solange auf keine anderen Kommandos reagieren, bis dies geschehen ist. Diese Interaktionsform wird als *modal* bezeichnet. Zudem muß ein Mechnismus vorgesehen werden, der es erlaubt, nur die Visualisierungen der Objekte neu darzustellen, deren interner Zustand sich auch wirklich verändert hat - alles andere wäre ineffizient. Ein Vergleich des hier gesagten mit einem LISP-Interpreter zeigt, daß sich eine derartige Anwendung ebenfalls als *Interpreter* betrachten läßt: die Anwendung interpretiert nämlich Benutzerkommandos, ebenso wie eine LISP-Zeile (Listener) LISP-Ausdrücke erwartet. Dies entspricht auch der CLIM-Sichtweise: der Programmierer spezifiziert einen Interpreter, der von CLIM dann verwaltet und ausgeführt wird. Tatsächlich ist dieser Interpreter in großen Teilen schon vorhanden: der CLIM-Entwickler paßt diese Maschinerie seinen Bedürfnissen an, wozu er Konstrukte benutzt, die als *deklarativ* einzustufen sind.

5.6 Anwendungsrahmen

Die Basis einer jeden CLIM-Anwendung ist ein *Anwendungsrahmen (Application Frame)*: mit seiner Definition erhält die Anwendung ihre Fenster (die wiederum Subfenster haben können), Dialogelemente, Menüs, usw. Der Rahmen regelt zudem die *Ereignisbehandlung (Event Handling)*, so daß der Programmierer sich hierum nicht zu kümmern braucht.

Der *Rahmenverwalter (Frame Manager) konkretisiert* die abstrakten Anforderungen des Progammieres, indem er entsprechende GUI-Elemente des Wirtsfenstersystems erzeugt und verwaltet. Dieser Abbildungsprozeß (die Konkretisierung der Anforderungen) kann in vielfältiger Weise vom Programmierer beeinflußt werden. Bei der Definition eines Anwendungsrahmens lassen sich viele Optionen über Schlüsselwortparameter angeben, so z.B.

- ob die Fenster Schiebeleisten (Scroll Bars) haben,
- welche Platzaufteilung bezüglich der Subfenster vorgenommen werden soll,
- ob eine Menüleiste eigebaut werden soll,
- Farbe und Abmessungen, etc.

Diverse voreingestellte Standardargumente (Defaults) und Methoden machen es zudem meist nicht nötig, viele dieser Optionen zu spezifizieren.

Anwendungsrahmen sind Klassen im Sinne von CLOS, wodurch sich die gesamte CLOS-Maschinerie mit ihnen nutzen läßt. So erklärt sich auch die hohe Orthogonalität und Durchdachtheit von CLIM - vom Programmierer wird jedoch eine gewisse Vertrautheit im Umgang mit CLOS erwartet. Wie in SmallTalk heißen die Fenster in CLIM-Terminologie *Panes*; diese sind ebenfalls Instanzen von Klassen, wovon es diverse vordefinierte gibt:

- Anwendungsfenster (Application Pane),
- Kommandozeile (Interactor Pane),
- Werteakzeptierungsfenster (Accept Values Pane),
- Mausdokumentationszeile (Pointer Documentation Pane),
- Titelzeile (Title Pane),
- Kommandomenüfenster (Command Menu Pane) und
- Menüzeile (Menu Bar Pane).

Die *Kommandozeile (Command Listener)* ist einer *LISP-Zeile (LISP Listener)* ähnlich: sie liest textuelle Benutzerkommandos und veranlaßt ihre Ausführung. Die *Mausdokumentationszeile (Pointer Documentation Pane)* ist eine automatisch verwaltete Statuszeile, die den Benutzer z.B. über ausführbare Kommandos oder Maustastenbelegungen informiert. Beide haben ihre Wurzeln in der LISP Machine. Mit dem *Werteakzeptierungsfenster (Accept Values Pane)* lassen sich beliebige Dialogelemente zu einem Benutzerdialog kombinieren.

Da es nicht sinnvoll ist, diverse CLIM-Optionen aufzuzählen, sei hier ein Beispiel eines Anwendungsrahmens gegeben (leider muß LISP-Vetrautheit vorausgesetzt werden):

```
(define-application-frame browser
    ()
  ((described-concept :initform nil
                      :accessor browser-described-concept)
   (current-concepts :initform nil :initarg :roots
                     :accessor browser-current-concepts)
   (show-super-concepts-p :initform nil
                     :accessor browser-show-super-concepts-p))

  (:command-definer t)
  (:command-table (browser))
  (:panes

   (concept-hierarchy-display-pane
    :application
    :incremental-redisplay t
```

```
   :display-function 'draw-concept-hierarchy
   :label "Concept Hierarchy"
   :scroll-bars ':both
   :end-of-page-action ':allow
   :end-of-line-action ':allow)

  (concept-info-pane
   :application
   :incremental-redisplay t
   #+allegro :excl-recording-p #+allegro t
   :display-function 'draw-concept-description
   :label "Concept Info"
   :text-style *command-listener-text-style*
   :scroll-bars ':both
   :end-of-page-action ':allow
   :end-of-line-action ':allow)

 (options-pane
   :accept-values
   :display-function
   '(accept-values-pane-displayer
      :displayer ,#'(lambda (frame stream)
                      (accept-options frame stream))))

 (listener-pane
   :interactor
   :label nil
   :text-style *command-listener-text-style*
   :scroll-bars ':both
   :min-height '(4 :line)
   :max-height '(4 :line)
   :height '(4 :line))

  (pointer-documentation-pane
   (make-clim-stream-pane
    :type 'pointer-documentation-pane
    :foreground +white+
    :background +black+
    :text-style
      (make-text-style
        :sans-serif :bold :small)
    :scroll-bars nil
    :min-height '(1 :line)
    :max-height '(1 :line)
    :height '(1 :line))))
 (:layouts
  (default
      (vertically ()
(horizontally ()
  (1/4 options-pane)
    (2/4 concept-hierarchy-display-pane)
  (1/4 concept-info-pane))
listener-pane
pointer-documentation-pane))))
```

Hier wird also eine Anwendungs- bzw. Rahmenklasse `browser` definiert, deren Liste von Oberklassen leer ist - tatsächlich ist implizit die Klasse `standard-application-frame` Standardoberklasse (Default Superclass). Es folgen die Anwendungs-CLOS-Slots, `described-concept`, etc. Sie werden wie bei `defclass` definiert. Da `:command-definer t` angegeben ist, wird auch ein Makro mit Namen `define-browser-command` für die Definition von Kommandos erzeugt. Kommandos werden später erläutert - es handelt sich bei ihnen um den grundsätzlichen Mechanismus, um in Interaktion mit der Anwendung zu treten. Die *Kommandotabelle (Command Table)* enthält nun diese Kommandos: ihr Name ist ebenfalls `browser`. Unter `:panes` sind die einzelnen Subfenster der Anwendung de-

finiert: zwei Anwendungsfenster (mit Namen `concept-hierarchy-display-pane`, `concept-info-pane`), eine Kommandozeile und eine Mausdokumentationszeile. Die Darstellungsfunktion (`display-function`) eines Anwendungsfenster ist für die Darstellung des Fensterinhalts verantwortlich. Sicherlich wäre es ineffizient, bei jeder kleinen Veränderung den gesamten Fensterinhalt neu zu zeichnen: stattdessen schaltet `:incremental-redisplay t` einen Mechanismus ein, der nur das neu zeichnet, was sich wirklich verändert hat. Hierzu wird von CLIM für jedes darzustellende Grafikelement ein bestimmter Wert, der von einer vom Programmierer definierten Funktion berechnet wird, memoriert und mit dem Grafikelement assoziiert. Hat sich dieser mit dem darzustellenden Grafikelement assoziierte Wert seit dem letzten Aufruf der `display-function` geändert, so wird das Objekt neu gezeichnet, ansonsten jedoch nicht. Unter `:layouts` ist nun die geometrische Anordnung der Panes angegeben, wobei die Brüche relative Platzbedürfnisse bezeichnen.

Das Werteakzeptierungsfenster stellt einen Benutzerdialog bereit: in der Darstellungsfunktion `accept-options`, die vom Programmierer zu definieren ist, werden entsp. Dialogelemente wie *An/Aus-Schalter (Toggle Buttons)* und *Listenfelder (List Boxes)* erzeugt, wozu bestimmte Umgebungen zu benutzen sind. Die GUI-Elemente werden dann automatisch verwaltet: Eingabewerte werden validiert und können in der Funktion `accept-options` an Symbole gebunden werden.

Die anderen Attribute sprechen weitgehend für sich selbst und werden hier nicht weiter erläutert. Es soll nur ein Eindruck vermittelt werden.

5.7 Darstellungstypen und Ausgabespeicherung

Der Benutzer einer grafischen Oberfläche muß mit den auf dem Schirm dargestellten Grafikobjekten in Interaktion treten, um sie zu manipulieren. Eine typische Aufgabe eines Oberflächenprogrammierers besteht z.B. darin, Routinen zu schreiben, die erkennen, ob der Benutzer ein Grafikobjekt mit der Maus selektiert. Der Programmierer muß dafür sorgen, daß die Objekte auf dem Schirm mit den internen Repräsentationen der Anwendungsobjekte korrespondieren, sie eventuell optisch hervorheben, wenn die Maus auf sie zeigt, etc.

Bestimmte Operationen (sie heißen in CLIM *Kommandos (Commands)*) sind zudem nur auf bestimmten Objekten möglich. Skalierungen und Rotationen machen diesen Verwaltungsaufwand für den Programmierer schnell zur ermüdenden Last - doch CLIM nimmt sie ihm vollständig ab. Der Schlüssel zur Erreichung dieser Funktionalität ist die Verknüpfung der Visualisierung eines Objektes mit einem Typ und dem Objekt selbst. In CLIM wird dieser Typ als *Darstellungstyp (Presentation Type)* bezeichnet, während die grafische Darstellung, also die Visualisierung bzw. *Präsentation (Presentation)* in einem *Ausgabeband (Presentation Record)* gespeichert wird. Hierbei handelt es sich um eine baumähnliche Datenstruktur, die die Zeichenoperationen speichert, die für das Zustandekommen der Präsentation ausgeführt worden sind. Letztendlich speichert sie eine Sequenz von Zeichenoperationen (daher die Übersetzung „Band“, in Analogie zu einem Magnetband, daß Aufnehmen und Wiedergeben kann).

CLIM kann somit u.a. selbständig das Grafikobjekt neuzeichnen: dieses ist erforderlich, wenn es z.B. von einem anderen Fenster verdeckt und dann wieder sichtbar wurde, oder aber eine Verschiebung des Fensterinhalts per Schiebebalken stattfand (Scrolling). In Windows-Terminologie nennt man diese Ausgabebänder *Metafiles*. Ausgabebänder benötigen viel weniger Speicher als Grafikbitfelder, und sind dennoch (da als Baum gespeichert) effizient.

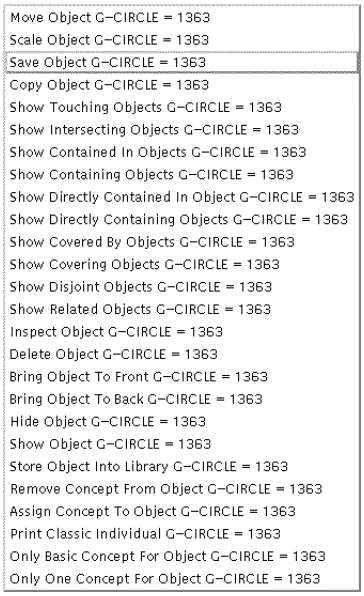


Abbildung 27: Kontextmenü

Die Zeichenfläche eines Fensters ist also meist kein passives Bitfeld, sondern eine Datenstruktur, die aus sogenannten *Präsentationen (Presentations)* besteht. Jede Präsentation besteht aus drei Dingen:

1. einer *Referenz* auf das Anwendungsobjekt,
2. dem *Darstellungstyp* und
3. dem *Ausgabeband*.

Zeigt der Benutzer nun mit der Maus auf eine Präsentation, so kann CLIM das Objekt visuell hervorheben (da das Ausgabeband gespeichert wurde). Zudem kennt CLIM den Darstellungstyp des Grafikobjektes, womit von CLIM festgestellt werden kann, welche der vorhandenen Kommandos anwendbar sind - denn die Argumente der Kommandos werden über Darstellungstypen spezifiziert, ähnlich wie Argumente in CLOS-Methoden durch Klassennamen spezifiziert werden. Bei Druck auf die rechte Maustaste erscheint nun ein Menü, welches dem Benutzer alle anwendbaren Kommandos für dieses Objekt zur Auswahl anbietet (s. Abb. 27). In der Terminologie des Betriebssystemes OS/2 wird dieses Menü *Kontextmenü* genannt. Kommandos, die nun auf diesem Objekt ausgeführt werden, bekommen außerdem automatisch über die gespeicherte Referenz das Anwendungsobjekt als Argument, womit desses Zustand sofort geändert werden kann. Wenn Kommandos auf ein Objekt angewendet werden können, wird seine Präsentation optisch hervorgehoben - es ist *sensitiv*. Sind keine Kommandos anwendbar, ist es nicht sensitiv.

Jede CLOS-Klasse ist als Darstellungstyp verwendbar, womit sich wiederum Vererbung usw. nutzen läßt. Darstellungstypen sind sogar mächtiger als CLOS-Klassen, da sie sich zusätzlich parametrieren lassen. Der Typ `(integer 0 10)` beschreibt die Menge der Ganzzahlen von 0 bis 10: eine solche Zahl könnte somit mit der Maus selektiert werden. Die *Metadarstellungstypen* `or` und `and` ermöglichen Kommandos, z.B. Argumente vom Typ `(or integer string)` zu übernehmen. Wieder gibt es

diverse vordefinierte Darstellungstypen, u.a. für Dateipfade, Sequenzen, Teilmengen und LISP-Ausdrücke (S-Expressions).

Der Einsatz von *Sichten (Views)* erlaubt es, Präsentationen eines gleichen Darstellungstyps unterschiedlich darzustellen, so z.B. textuell oder grafisch. Die Methode, die für die Erzeugung der Visualisierung eines Objektes entsprechenden Typs verantwortlich ist, heißt `present` (s. Abb. 26).

Ebenso wie *Ausgaben (Output)* sind in CLIM auch *Eingaben (Input)* dynamisch typisiert - wenn eine Applikation die Eingabe einer Hausnummer erwartet (z.B. „5“), so kann der Benutzer mit der Maus auf ein vorher präsentiertes Objekt klicken - vorausgesetzt, der Typ der so selektierten Visualisierung eines Objektes ist auch vom Typ des momentan vorliegenden *Eingabekontextes (Input Context)*, in diesem Fall also Hausnummer. Der Eingabekontext bestimmt also, welche Art von Objekten gerade von der Applikation als Eingabe verlangt wird. Wurde die „5“ jedoch als Darstellungstyp `integer` präsentiert, so ist das Objekt „5“ nicht sensitiv, da der Eingabekontext keine `integers`, sondern `hausnummern` verlangt, obwohl die Visualisierung von Objekten dieser Typen sogar gleich sein können.

Hier wird ersichtlich, welchen Komfort ein derartig objektorientierter Ansatz ermöglicht. Ein Eingabekontext kann u.a. mit `accept` aufgesetzt werden - so kann man schreiben: `(accept 'hausnummer)`.

Doch wie kommt die Hausnummer nun auf den Schirm? In CLIM wird mit einem Konstrukt der Form

```
(with-output-as-presentation (stream object type) &body body)
```

eine Präsentation des Objektes `object` vom Darstellungstyp `type` auf dem Ausgabestrom bzw. Fenster `stream` erreicht. Im Körper `body` dieses Makros stehen dann Zeichenroutinen, welche die visuelle Repräsentation, die als *Ausgabeband* gespeichert wird, erzeugen. Für Hausnummern z.B.

```
(with-output-as-presentation (stream number-object 'hausnummer)
  (write-to-string number))
```

Dabei wird hier durch `(write-to-string number)` eine textuelle Präsentation vom Typ `hausnummer` erzeugt. Denkbar wäre natürlich auch eine ansprechendere, grafische Präsentation (z.B. ein kleines Hauspiktogramm m. eingezeichneter Nummer). Normalerweise spezialisiert der Programmierer die Methode `present`, um eine angemessene Präsentation für Objekte seines selbstdefinierten Darstellungstyps zu erzeugen.

An dieser Stelle seien einige weitere Beispiele für Darstellungstypen gegeben:

```
(declass student () ; jede CLOS-Klasse kann als Darstellungstyp dienen
  ((name :reader student-name :initarg :name)
   (courses :accessor student-courses :initform nil)))

(define-presentation-type puzzle-cell () ; fuer ein 8-Puzzle
  :inherit-from '(integer 1 15))

(define-presentation-type line-style-type () ; fuer einen Grafikeditor
  :inherit-from '((completion (:solid :dashed))
                  :name-key identity
                  :printer present-line-style
                  :highlighter hightlight-line-style))
```

Interessant ist der letzte Darstellungstyp: er erbt vom Darstellungstyp `completion` und wird noch mit diversen Optionen verfeinert. Ein genau Erklärung würde zu weit führen.

5.8 Kommandos und Gesten: CLIM-Interaktionsformen

Ein typisches Kommando eines Grafikeditors wäre z.B. das Verschieben eines Grafikobjektes: der Benutzer führt die Maus über das Grafikobjekt (es ist sensitiv und wird optisch hervorgehoben), selektiert es durch Maustastenbetätigung, verschiebt Maus und somit auch Objekt und klickt erneut mit der Maus. Nun hat das Grafikobjekt seine neue Position. In CLIM-Terminologie werden derartige *Handlungssequenzen Gesten (Gestures)* genannt. Prinzipiell gibt es *Mausgesten (Pointer Gestures)* und *Tastaturgesten (Keyboard Gestures)*. Das gleiche Ergebnis obiger Handlungssequenz könnte der Benutzer auch auf andere Art erzielen: so z.B., indem er in eine Kommandozeile (Interactor Pane) ein Kommando der Art „move-object circle-123“ tippte. Alsdann könnte er mit der Maus die neue Position per Klick bestimmen, oder aber Koordinaten textuell eingeben.

Es kann also auf verschiedene Interaktionsarten das gleiche Ergebnis erzielt werden: es ist möglich, von der genauen Interaktionsart oder Handlungssequenz zu abstrahieren. Genau diese Abstraktion wird in CLIM mit den *Kommandos (Commands)* in Verbindung mit *Darstellungsübersetzern (Presentation Translators)* unterstützt. CLIM bietet generell vier verschiedene Interaktionsarten:

1. Mausinteraktion über Menüs,
2. Mausinteraktion über Darstellungsübersetzer und Gesten,
3. Tastaturinteraktion über die Kommandozeile und
4. Tastaturinteraktion über *Einzeltastenbetätigungen (Keyboard Accelerators, Bindkeys)*.

Ein Darstellungsübersetzer ist ein Abbildungsmechanismus, der Präsentationen eines Ursprungsdarstellungstyps in Präsentations eines Zieldarstellungstyps „übersetzt“ bzw. abbildet. Er tut dies, wenn ein Eingabekontext vom Darstellungstyp Zieltyp besteht, die Präsentation des fraglichen Objektes vom Darstellungstyp Ursprungstyp ist, und der Benutzer mit einer entsp. Interaktionsform (einer Geste) die fragliche Präsentation selektiert. Wenn z.B. ein CAD-Programm zum Schaltkreisentwurf einen Widerstandswert (in Form eines `real`) verlangt, so kann der Benutzer auf einen Widerstand klicken. Dessen Widerstandswert wird dann genommen. Dieser Übersetzer nimmt also eine Abbildung bzw. Übersetzung vom Darstellungstyp `resistor` auf den Darstellungstyp `real` vor. Der Übersetzer kann so definiert werden:

```
(define-presentation-translator resistor-resistance
  (resistor real ECAD-command-table ; Ursprungstyp, Zieltyp
    :documentation "Resistance of this resitor"
    :gesture :select)
  (object) ; wird an das Widerstandsobjekt selbst gebunden
  (resistor-resistance object)) ; ein REAL wird zurueckgegeben
```

Der Übersetzer wird per `:select`-Geste (linker Mausknopf) auf die Präsentation eines Widerstandes angewendet. Er gibt dann einen REAL-Wert als Objekt vom

Darstellungstyp Zieltyp zurück. Dabei ist dieser Wert ein Slot des Widerstandsobjektes, der mit `(resitor-resistance object)` extrahiert wird. Wieder ist zwischen dem Objekt selbst und seiner Präsentation zu unterscheiden: der Benutzer selektiert die Präsentation, der Darstellungsübersetzer bekommt aber das Anwendungsobjekt selbst (an `object` gebunden) als Argument.

Kommandos sind *Objekte 1. Klasse (First Class Objects)* und auch Darstellungstypen. Jedes Kommando repräsentiert eine einzelne Benutzerinteraktion. Kommandos werden in sog. Kommandotabellen eingetragen und gespeichert, welche ebenfalls CLOS-Instanzen sind. Auch Kommandos können somit Zieldarstellungstyp eines Übersetzers sein oder per Maus angeklickt werden. Soll z.B. eine Datei, dessen Pfadname gegeben ist, gelöscht werden, so kann man einen Übersetzer wie folgt definieren:

```
(define-presentation-to-command-translator delete-file
  (pathname com-delete-file my-command-table
    :tester ((object) (not (file-deleted-p object)))
    :documentation "Delete this file"
    :gesture :delete)
  (object) ; wird an das Anwendungs-Objekt gebunden
  (list object)) ; Argumentliste f. das Kommando com-delete-file
```

Dies ist nur eine Kurzform für

```
(define-presentation-translator delete-file
  (pathname cmmand my-command-table
    :test ((object) (not (file-deleted-p object)))
    :documentation "Delete this file"
    :gesture :delete)
  (object) ; wird an das Anwendungs-Objekt gebunden
  (com-delete-file object))
```

`com-delete-file` ist der Name eines Kommandos, welches ein Argument `object` vom Darstellungstyp `pathname` erwartet. Der Übersetzer ist jedoch nur anwendbar, wenn die Datei, deren Pfadname an `object` gebunden wird, nicht schon gelöscht ist (`:tester`). Das Kommando selbst kann wie folgt definiert werden:

```
(define-command (com-delete-file :name "Delete File" :keystroke (:d)
                                 :menu "Delete File By Menu"
                                 :command-table my-command-table)
  ((object 'pathname)) ;  hat den Darstellungstyp Pathname
  (delete-file object))
```

Dieses Kommando befindet sich in der Kommandotabelle `my-command-table` (`:command-table`) und verlangt ein Objekt, dessen Präsentation den Darstellungstyp `pathname` hat. Dieses Kommando hat folgende Eigenschaften:

1. Es läßt sich mit einem Tastendruck „d“ aktivieren (`:keystroke`),
2. erscheint auch als Menüeintrag Delete File By Menu in der Menüleiste (`:menu`),
3. und läßt sich von der Kommandozeile textuell einlesen, wozu der Benutzer „Delete File“ tippen muß (`:name`).

Für die letztgenannte Interaktionsform ist es zudem notwendig, daß CLIM ein solches Objekt *parsen*, also anhand einer textuellen Präsentation das Anwendungsobjekt eindeutig identifizieren kann. Hierfür müssen eventuell sogenannte `accept`-Methoden geschrieben werden, welche den Eingabekontext aufsetzen und das Parsing durchführen.

Oftmals kann man sich das Schreiben eines Darstellungsübersetzers sparen, da bei der Definition eines Kommandos mit angegeben werden kann, mit welcher Geste ein Kommandoargument gewonnen werden kann, wie in dem Beispiel

```
(define-command (com-move-object :command-table gened-table)
    ((object 'thing :gesture :move))
  (move-object object))
```

Hierbeit soll also das Argument `object` vom Darstellungstyp `thing` sein und per `:move`-Geste über die Maus bereitgestellt werden. Der Darstellungsübersetzer, der von Objekten des Darstellungstyp `thing` auf Objekte vom Darstellungstyp `command` abbildet, indem er das Kommando `com-move-object` zurückgibt, wird implizit erzeugt.

Sobald nun ein solcher Darstellungsübersetzer auf ein Objekt anwendbar ist, ist es sensitiv: seine Präsentation wird optisch hervorgehoben, wozu das gespeicherte Ausgabeband benutzt wird. Somit bekommt der Benutzer eine direkte Rückmeldung, welche zudem noch sehr ansprechend aussieht.

5.9 Benutzerdialoge, Sichten und Menüs

Die einfachste Form eines Benutzerdialoges besteht in einem Aufruf der Funktion `notify-user` - sie versieht den Benutzer mit einer Warnung, Fehlermeldung, Ja/Nein-Frage o.ä. Ein komfortabler Dateiauswähler wird ebenfalls mit einem einzigen Funktionsaufruf von `select-file` geöffnet (s. Abb. 28). Als Beispiel dient eine den Dateiauswähler verwendende Funktion:

```
(defun file-selector (title directory)
  (with-application-frame (gened-frame)
    (let ((file (select-file gened-frame
                  :title title
                  :directory directory)))
      (if (and file (not (string= "" file)))
          (if (char= (elt file
                          (1- (length file)))
                     #\/)
              (progn
                (notify-user gened-frame
                  "You must select a file, not a directory!"
                  :style :error)
                nil)
              file)
          (progn
            (notify-user gened-frame
               (format nil "No file selected!")
               :style :error)
            nil)))))
```

Komplexere, nichtvordefinierte Dialogelemente (Aggregate aus virtuellen *Druckschaltern, Schiebereglern und Radioschaltern* etc.) lassen sich mit Hilfe der Umgebung `accepting-values` erzeugen, welche in einem Werteakzeptierungsfenster

(Accepting Values Pane) zu benutzen ist. Hervorzuheben ist hierbei die Möglichkeit, die angesprochenen Formatierungsmöglichkeiten (für Tabellen, Reihen, Spalten) von CLIM für die Ausrichtung der Dialogelemente untereinander zu nutzen.

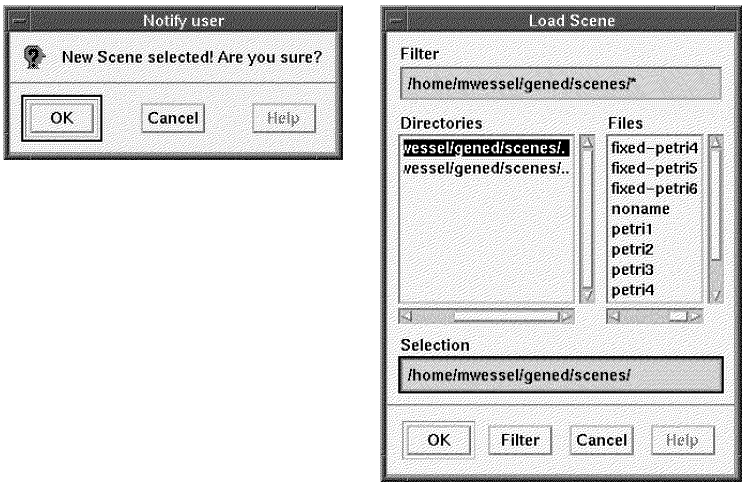


Abbildung 28: Standarddialoge

Jeder der schon mal mit einem der typischen *Interface Builder* für z.B. die Programmierung unter Windows gearbeitet hat, weiß wie mühsam es ist, entsprechende Dialogelemente manuell anzuordnen.[10] Zudem berechnet CLIM Formatierungen auch selbständig und stellt viele Standardeinstellungen bereit. Hier ein Beispiel für einen Dialog, der in GENED verwendet wird:

```
(defun accept-it (stream type default prompt query-id
                  &key (view 'gadget-dialog-view))
  (let (object ptype changed)
    (formatting-cell (stream :align-x :center
                             :align-y :center)
      (multiple-value-setq (object ptype changed)
        (accept type
          :stream stream :default default
          :query-identifier query-id
          :prompt prompt :view view)))
    (values object changed)))

(defmethod accept-options ((frame gened) stream)
  (with-slots (default-line-style default-arrow-head) frame
    (formatting-table (stream)
      (formatting-row (stream)

        (multiple-value-bind (line-style changed1)
            (accept-it stream 'line-style-type default-line-style
              "Style" 'style :orientation orientation)
          (setf default-line-style line-style))

        (multiple-value-bind (boolean changed2)
            (accept-it stream 'boolean default-arrow-head
              "Head " 'head :orientation orientation)
          (setf default-arrow-head boolean))))))
```

[10] Allerdings muß erwähnt werden, daß einige der Formatierungsfunktionen noch nicht ausreichend implementiert sind - so war die Erstellung des GENED-Layouts nur mit Tricks möglich.

Die Methode `accept-options` wird nun als Darstellungsfunktion (`:display--function`) eines Werteakzeptierungsfensters angegeben, genauso wie bei dem oben diskutierten Anwendungsrahmen `browser`. Die einzelnen Dialogelemente werden durch `accept` erzeugt.

Darstellungstypen lassen sich im durch `accept` erzeugten Eingabekontext sogenannten *Sichten (Views)* zuordnen. Die Sicht auf einen Darstellungstyp bestimmt dann das konkrete GUI-Element, das benutzt wird, um innerhalb dieser Umgebung das Objekt des entsprechenden Darstellungstyps einzulesen. So kann also der Abbildungsprozeß des Rahmenverwalters (Frame Managers) beeinflußt werden. Für den vordefinierten Darstellungstyp `boolean` ist das Standard-GUI-Element ein einfacher *An/Aus-Schalter (Toggle Button).*

Für einen entwicklerdefinierten Darstellungstyp wie `line-style-type` ergeben sich mögliche Sichten meist durch Vererbung - weiter oben wurde dieser Typ als Subtyp von `completion` abgeleitet, der eine endliche Menge von sich gegenseitig ausschließenden Möglichkeiten bezeichnet. Eine mögliche Sicht für `completion` ist z.B. `+radio-box-view+`: per Radioschalter kann ein Element ausgewählt werden. Eine andere mögliche Sicht für diesen Darstellungstyp ist `+list-pane-view+`: hier wird stattdessen ein Listenfeld (List Box) erzeugt.

Für `integer` könnte ein Schieberegler (Slider) als Sicht angegeben werden, mit dem sich die Zahl einstellen läßt (`+slider-view+`), oder aber ein `+text-field-view+`, um die Zahl textuell in ein Editorfeld einzutippen.

Sichten sind wiederum Klassen, nämlich Subklassen von `gadget`. Die Sicht `+text-field-view+` erzeugt Instanzen der Klasse `text-field`, ebenso wie `+list-pane-view+` Instanzen der Klasse `list-pane` erzeugt. Die oben verwendete Sicht `'gadget-dialog-view` wird jeweils auf die *Standardsicht (Default View)* des in `accept` verwendeten Darstellungstyps abgebildet. Die entsprechenden Dialogelemente lassen sich dynamisch erzeugen und sind auch *Panes,* wie in den folgenden Beispielen:

```
(with-radio-box ()
  (make-pane 'toggle-button :label "Mono")
  (radio-box-current-selection
    (make-pane 'toggle-button :label "Stereo"))
  (make-pane 'toggle-button :label "Quadrophonic"))

(make-pane 'list-pane
  :value '("Lisp" "C++")
  :mode :nonexclusive
  :value-changed-callback 'list-pane-changed-callback
       ; wird bei Veraenderung aufgerufen
  :items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada"))
```

Nun noch ein paar Worte zu den Menüs: Die Menüleiste einer Anwendung kann bevölkert werden, indem bei den einzelnen Kommandos das Schlüsselwort `:menu-item t` angegeben wird, bzw. eine Zeichenkette anstatt `t` als Name für den Menüeintrag.

Dynamische Menüs außerhalb der Menüleiste lassen sich u.a. mit der Funktion `menu-choose` erzeugen. Auch ihr kann man wieder etliche Optionen übergeben. Ein einfaches Beispiel ist

```
(menu-choose '("One" "Two" "Three"))
```

Hier wird ein einfaches Menü dynamisch erzeugt, und der Benutzer kann dann eines der drei Elemente auswählen. Die entsprechende Zeichenkette wird dann zurückgegeben. Ein komplexeres Beispiel zeigt, daß auch beliebige Grafiken als Menüeinträge verwendet werden können:

```
(menu-choose '(circle square triangle)
  :printer #'(lambda (item stream)
               (case item
                 (circle (draw-circle* stream 0 0 100))
                 (square (draw-polygon* stream
                           '(-8 -8 -8 8 8 8 8 -8)))
                 (triangle (draw-polygon* stream
                             '(10 8 0 -10 -10 8))))))
```

Zurückgegeben wird in diesem Fall eines der Symbole `circle, square` oder `triangle`. Zusätzlich lassen sich Abbildungsfunktionen spezifizieren, die Abbildungen von Menüeinträgen auf Rückgabewerte, Namen o.ä. vornehmen. Hierarchische Menüs lassen sich erzeugen, indem man als Menüeinträge Listen angibt, die hinter dem Schlüsselwort `:items` geschachtelt sind:

```
(menu-choose
   '(("123" :items (("1" :items ("one" "eins"))
                    ("2" :items ("two" "zwei"))
                    ("3" :items ("three" "drei"))))))
```

Selektiert der Benutzer „123“, so erscheint ein Submenü mit den drei Einträgen „1“, „2“ und „3“. Wählt der Benutzer einen der drei Einträge, so erscheint ein weiteres Menü mit jeweils zwei Einträgen, daß dem Benutzer in Abhängigkeit der zuvor gewählten Zahl das deutsche und englische Wort für diese Zahl zu Auswahl anbietet. Diese Zeichenkette wird dann zurückgegeben. Weiter kann auf CLIM leider nicht eingegangen werden. Den interessierten Leser verweise ich auf [17].

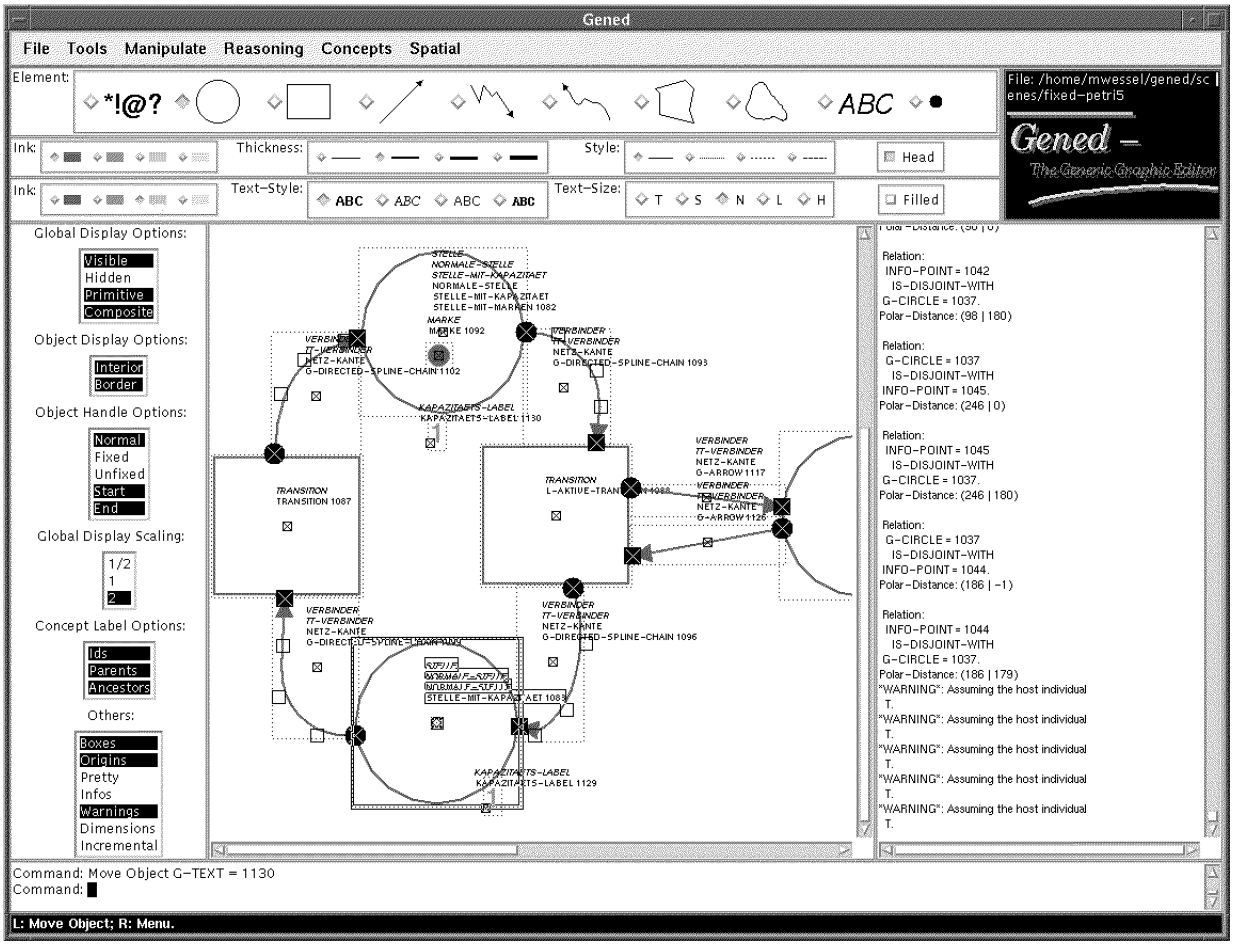


Abbildung 29: Die GENED-Oberfläche

6 GENED - Versuch eines konzeptorientierten generischen Grafikeditors

6.1 Motivation

Dieses Kapitel dient der Vorstellung von GENED, dem konzeptorientierten generischen Grafikeditor. Diskutiert werden Bedienkonzept, Interaktionsformen und der GENED-Entwurf im Groben - ausführlicher zu Implementationsaspekten informiert jedoch Kap. 8. Ein Benutzungsbeispiel ist in Kap. 7 dargestellt - einige Menüpunkte müssen dennoch hier in ihrer Funktionalität dargestellt werden, da sonst Struktur und Handhabung des Systems nicht deutlich werden. Desweiteren wird Vertrautheit in der Bedienung *grafischer Benutzeroberflächen* sowie der Handhabung der *Maus* vorausgesetzt.

6.2 Vorstellung der Oberfläche

Abb. 29 zeigt die GENED-Oberfläche. Zu erkennen sind diverse Dialogelemente, sowie drei vertikale große Fenster (v. li. n. r.):

Das Optionenfenster bietet einige Schaltergruppen zur teilweise exklusiven, teilweise nichtexklusiven Einstellung diverser Parameter.

Global Display Options bestimmt, welche Objekte dargestellt werden - durch eine Auswahl Visible Primitive Composite werden alle sichtbaren primitiven und aggregierten Objekte dargestellt, während Hidden Composite nur die unsichtbaren aggregierten Objekte anzeigt. Dabei werden diese natürlich sichtbar - Visible und Hidden sind nur Bezeichnungen für zwei verschiedene *Sichtbarkeitswelten*, die der Benutzer wahlweise ein- bzw. ausblenden kann.

Object Display Options bestimmt, ob Rand und Inneres der Objekte oder nur eines dargestellt wird. Das Innere eines Objektes kann nur dargestellt werden, wenn zum Zeitpunkt der Erzeugung der Schalter Filled an gewesen ist.

Object Handle Options bestimmt, welche Arten von *Object Handles* angezeigt werden sollen. Handles werden im folgenden als *Punktmanipulatoren* bezeichnet, da mit ihrer Hilfe einzelne signifikante Punkte eines Objekte gezielt verändert werden können. Auf die verschiedenen *Arten* von Punktmanipulatoren wird noch eingegangen.

Global Display Scaling erlaubt es, das gesamte Entwurfsfenster zu skalieren.

Concept Label Options bestimmt, welche Arten von Konzeptbezeichnern (die von CLASSIC für die Grafikobjekte berechnet werden) im Entwurfsfenster erscheinen sollen. Vorgängerkonzepte (Ancestor Concepts) erscheinen im Gegensatz zu den Elternkonzepten (Parent Concepts) in kursiver Schrift.

Others faßt diverse andere Optionen zusammen: so lassen sich mit Boxes die begrenzenden Rechtecke (Bounding Boxes) der Objekte an- und ausschalten, und mit Origins die lokalen Koordinatenursprünge. Warnings bezieht sich auf CLASSIC-Warnungen. Mit Dimensions wird dem Geometriemodul mitgeteilt, ob es auch die Dimension der vorhandenen Schnitte berechnen soll, und Incremental schaltet den *inkrementellen Modus* ein oder aus.

In CLIM-Terminologie handelt es sich bei diesem Fenster um eine Instanz der Klasse „Accepting Values Pane“ (s. Kap. 5).

Das Entwurfsfenster zeigt momentan einen Ausschnitt eines Petrinetzes. Es sind diverse Grafikelemente zu erkennen: die Objekte selbst, ihre begrenzenden Rechtecke, der Ursprung ihrer lokalen Koordinationsysteme (meist die Mitte des Objektes, dargestellt durch kleines Quadrat mit Kreuz), diverse Konzeptbeschriftungen sowie kleine Quadrate und Kreise (teilw. mit Kreuz), welche als Punktmanipulatoren dienen. Alle Objekte sind sensitiv: in Abb. 29 zeigt der Benutzer gerade mit der Maus auf die unterste Netzstelle, die daher optisch hervorgehoben erscheint. Das Fenster ist eine Instanz der CLIM-Klasse „Application Pane“.

Das Infofenster zeigt gerade einige berechnete topologische Relationen an. Ob in ihm Informationen und Hilfestellungen für den Benutzer z.B. bei der Erzeugung von Objekten gegeben werden, hängt vom Schalter Infos im Optionenfenster ab. CLASSIC-Fehler erscheinen stets hier, Warnungen jedoch in Abhängigkeit von der Stellung des Schalters Warnings. Wieder handelt es sich um eine „Application Pane“.

Unterhalb der drei Fenster ist die Kommandozeile (Interactor Pane) zu erkennen - sie fordert zur textuellen Eingabe eines Kommandos auf. Es folgt die Dokumentationszeile (Pointer Documentation Pane), die u.a. über Maustastenbelegungen informiert.

Oberhalb der drei Fenster finden sich wieder diverse Schalter, welche teilweise mit Text, teilweise mit Grafikobjekten bezeichnet sind. Aus der Gruppe [Element] wählt der Benutzer die im folgenden zu benutzende Objektart - so befindet sich GENED in Abb. 29 gerade im Modus für die Kreiserzeugung. Hinter dem opaken Ikon [*!@?] verbirgt sich ein beliebiges Grafikobjekt der GENED-Bibliothek. [ABC] bezeichnet Textelemente. Die Schaltergruppen der beiden Zeilen unterhalb von [Element] bedeuten folgendes:

[Ink Thickness Style Head] bezieht sich auf den Rand ([Border]) der noch zu erzeugenden Grafikobjekte, wobei sich mit [Head] festlegen läßt, ob die angebotenen eindimensionalen Objekte (also Strecke, Kette und Spline-Kette) einen Pfeilkopf haben sollen oder nicht. Eindimensionale Objekte mit Pfeilkopf werden als gerichtet bezeichnet. Es erschien nicht sinnvoll, ihnen jeweils einen eigenen Schalter in der Gruppe [Element] zu geben.

[Ink Text-Style Text-Size Filled] bezieht sich auf das Innere ([Interior]) der noch zu erzeugenden zweidimensionalen Grafikobjekte sowie Textobjekte. [Ink] bestimmt die Farbe des Inneren ([Interior]) (wenn der Schalter [Filled] gedrückt ist), während [Text-Style] und [Text-Size] nur Wirkung haben, wenn Elemente der Art [ABC] (Text) erzeugt werden.

6.3 Objektarten in GENED

Jede grafische Konstellation muß mit einer bestimmten Anzahl grafischer Primitive aufgebaut werden. Grundsätzlich gibt es in GENED zwei Objektarten: primitive und aggregierte Objekte.

6.3.1 Primitive Objekte

Alle möglichen Primitive sind in der [Element]-Gruppe aufgeführt, v. li. n. r.: Kreis, Rechteck, Pfeil oder Strecke, gerichtete oder ungerichtete Kette, gerichtete oder ungerichtete Spline-Kette, Polygon, Spline-Polygon, Text und Punkt. Punkte werden als kleine gefüllte Kreise visualisiert, vom Geometriemodul jedoch punktförmig behandelt. Da Spline-Kurven nicht von CLIM berechnet werden, mußte ein eigenes Modul geschrieben werden.

Ein *Spline-Polygon* ist eine geschlossene Spline-Kurve (s. Abb. 30): die Übergänge zeichnen sich durch Stetigkeit aus, so daß der Eindruck eines amorphen, organischen Objektes entsteht. Um diesen Übergang zu erreichen, werden die ersten drei Punkte einfach wieder ans Ende der Spline-Liste angehängt.

6.3.2 Aggregierte Objekte

Ein Beispiel für ein Kompositionsobjekt ist z.B. das Schaltzeichen eines Transistors: verschiedene Primitive bilden das *Piktogramm,* der Betrachter abstrahiert von der Identität der Komponentenobjekte und gibt dem Ganzen einen Namen, nämlich

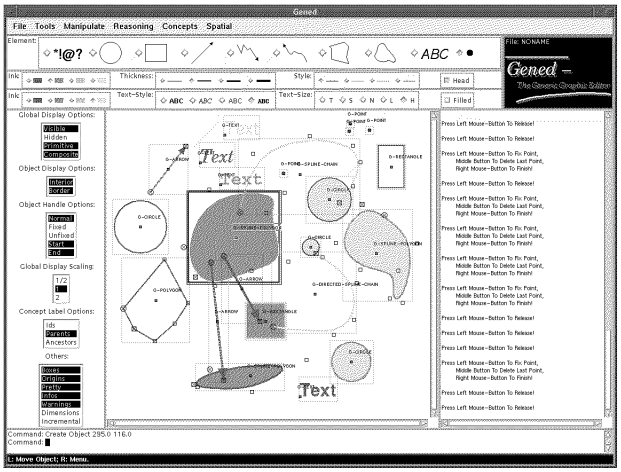


Abbildung 30: Primitive Objekte in GENED

Transistor bzw. Transistorschaltzeichen. Dies läßt sich als Konzeptbildung auffassen. Der Editor sollte eine derartige Betrachtungsweise des Benutzers also berücksichtigen (s. Kap. 2).

6.3.3 Punktmanipulatoren

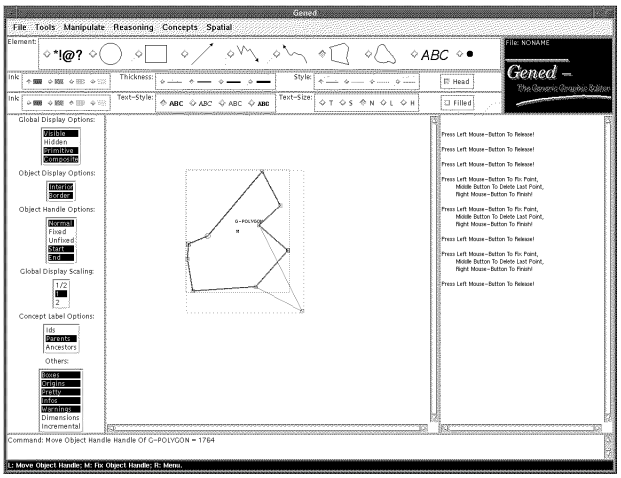


Abbildung 31: Punktmanipulatoren

Punktmanipulatoren sind keine selbständigen Objekte - sie gehören immer zu einem primitiven Objekt, nämlich zu Pfeil oder Strecke, gerichteter oder ungerichteter Kette, Polygon oder Spline-Polygon. Mit ihrer Hilfe können signifikante Punkte dieser Objekte direktmanipulativ verändert werden: der Benutzer klickt einen Manipulator mit der Maus an und bewegt die Maus, wodurch die Position dieses signifikanten Punktes mit der Maus verschoben wird. Das Objekt selbst wird automatisch angepaßt und normalisiert - so muß z.B. der Objektursprung neu berechnet werden, wenn er per Definition stets in der Mitte liegen soll. Abb. 31 verdeutlicht diese Interaktionsform.

Bei Polygonen und Ketten wird validiert, daß sie sich nicht selbst überschneiden. Bei der Bewegung der Manipulatoren bekommt der Benutzer wie auch bei den

Erzeugungsroutinen eine sofortige Rückmeldung - schon während der Veränderung kann er erkennen, wie das Objekt einmal aussehen wird („What You See Is What You Get“ oder kürzer: WYSIWG). Eine Ausnahme bilden die Spline-Objekte, da es zu aufwendig wäre, bei jeder Positionsveränderung der Maus die Splines neu zu berechnen. Stattdessen werden nur die Stützstellen der Spline-Kurven während der Manipulation neu gezeichnet und miteinander verbunden (wie bei einfachen Ketten oder Polygonen). Erst wenn der Benutzer den Manipulator wieder frei gibt wird das Spline-Objekt vollständig neu berechnet und dargestellt.

Es gibt verschiedene Arten von Punktmanipulatoren: Start- und Endpunktmanipulatoren für gerichtete eindimensionale Objekte, alle andere Manipulatoren werden hingegen als „normal“ bezeichnet. Manipulatoren lassen sich zudem an anderen Objekten *befestigten.* Wird solch ein Objekt verschoben, so werden alle an ihm befestigten Manipulatoren ebenfalls mitverschoben, was dazu führt, daß die Objekte, zu denen die Manipulatoren gehören, ebenfalls manipuliert werden: einige ihrer signifikanten Punkte bekommen neue Positionen, nämlich die, die den befestigten Manipulatoren entsprechen.

Die Schalter der Object Handle Options-Gruppe bestimmen, welche Art von Punktmanipulatoren im Entwurfsfenster dargestellt werden: eine Auswahl Start Fixed zeigt alle befestigten Startpunktmanipulatoren. Punktmanipulatoren können auch an Kompositionsobjekten befestigt werden, wo sie die gleiche Wirkung haben.

Visualisiert werden sie folgendermaßen:

- ein Startpunktmanipulator (Start Handle) als kleiner Kreis mit Kreuz,
- ein normaler Manipulator (Handle) als kleines Quadrat,
- ein Endpunktmanipulator (End Handle) als kleines Quadrat mit Kreuz, und
- befestigte Manipulatoren (Fixed Handles) werden invers zu den unbefestigten Manipulatoren (Unfixed Handles) dargestellt (s. Abb. 29 - alle dargestellten Manipulatoren sind befestigt).

6.3.4 Objekterzeugung und Bibliotheksbenutzung

Für unterschiedliche Objektarten sind verschiedene Interaktionsformen zur Erzeugung notwendig.

Alle Grafikobjekte werden mit Hilfe der Maus erzeugt - hierbei soll der Benutzer schon während der Erzeugung sehen, wie das fertige Objekt einmal aussehen wird (WYSIWG). Unterschiedliche Objektarten verlangen jedoch verschiedene Erzeugungssequenzen - für einen Kreis ist eine andere Interaktionsform vorzusehen und zu implementieren als für ein Polygon. Für Textobjekte muß hingegen eine Interaktion realisiert werden, die es ermöglicht, einen Text vom Benutzer über einen Dialog einzulesen. Diese Handlungssequenz hat sicherlich nichts mit der für eine Kreiserzeugung benötigten Routine zu tun. Abb. 32 verdeutlicht die unterschiedlichen Interaktionsformen.

Unterschiedliche Objekte erforden also vollkommen verschiedene Erzeugungs- und Interaktionsarten. Bei einem generischen, universell verwendbaren Grafikeditor stellt sich die Frage, wie dieses Problem angesichts nicht feststehender Einsatzdomänen gelöst werden kann. Schließlich ist die Anzahl und Art der vom Benutzer geforderten Grafikobjekte nicht vorhersehbar. In Kap. 2 wurde jedoch schon

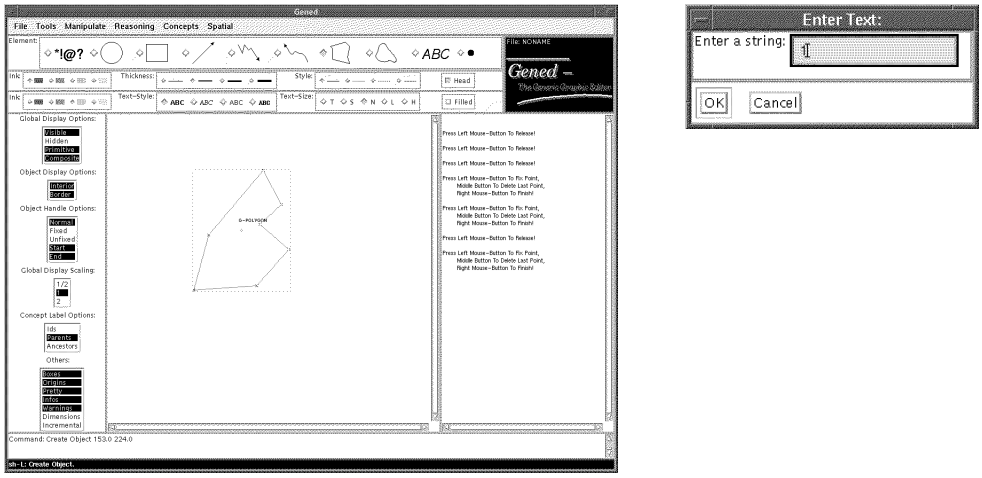


Abbildung 32: Erzeugungssequenzen

erwähnt, daß viele Grafikobjekte aus Primitiven aufgebaut werden können: der Benutzer kann also den Weg gehen, ein Objekt aus verschiedenen Primitiven zu aggregieren und dann dieses Objekt in einer Bibliothek abzuspeichern. GENED bietet dem Benutzer die wohl am häufigsten benötigten Primitive als Grundbausteine. Ist das aggregierte Objekt in der Bibliothek gespeichert, läßt es sich beliebig oft mit einer Geste instantiieren.

So kann z.B. ein Transitor als aggregiertes Objekt in die Bibliothek eingetragen werden, ohne daß im Editor eine spezielle Routine zur Erzeugung von Transistoren existieren muß. Diese wäre sicherlich auch von den nötigen Handlungssequenzen sehr kompliziert. Durch die Benutzung der Bibliothek braucht der Editor somit nicht umprogrammiert werden. Er kann auf diese Art ebenso Petrinetzstellen mit Marken und Kapazitätsbeschriftung handhabbar machen.

Sobald Transitoren oder Stellen mit Marken in der Bibliothek gespeichert sind, sind sie fest: sie lassen sich nicht weiter parametrieren bzw. variieren, wie es z.B. bei Kreisen mit dem Radius möglich ist - schließlich existiert ja eine speziell für diesen Zweck implementierte Kreiserzeugungsroutine. Von Bibliotheksobjekten werden hingegen immer nur exakte Kopien erzeugt. Jedoch lassen sich verschiedene Transistoren bzw. Visualisierungen des Konzeptes Transistor in der Bibliothek abspeichern - der Benutzer wählt dann die Instanz, die ihm am geeignetsten scheint. Somit ist eine endliche Anzahl von Variationen auch für benutzerdefinierte Objektarten möglich.

Abb. 33 zeigt den Benutzer beim Erzeugen eines Kompositionsobjekte, wozu er die Komponentenobjekte mit einem mausgesteuerten Rechteck umschließt. Alle Objekte, deren Mittelpunkt innerhalb diese Rechteckes ist, werden zu Teilobjekten und verlieren ihre Identität. Sie sind nun nicht mehr maussensitiv. Wird das Kompositionsobjekt verschoben, so sind alle Teilobjekte betroffen.

6.3.5 Repräsentation von Grafikobjekten

Grafikobjekte sind Instanzen von CLOS-Klassen (s. Kap. 8). Für CLASSIC muß zudem eine interne Repräsentation dieser Grafikobjekte erzeugt werden; das gleiche gilt für das Geometriemodul. Für jedes Grafikobjekt muß ein *korrespondierendes CLASSIC-Individuum* sowie eine Instanz einer der im Geometriemodul definierten CLOS-Klassen erzeugt werden, damit topologische Relationen berechnet werden

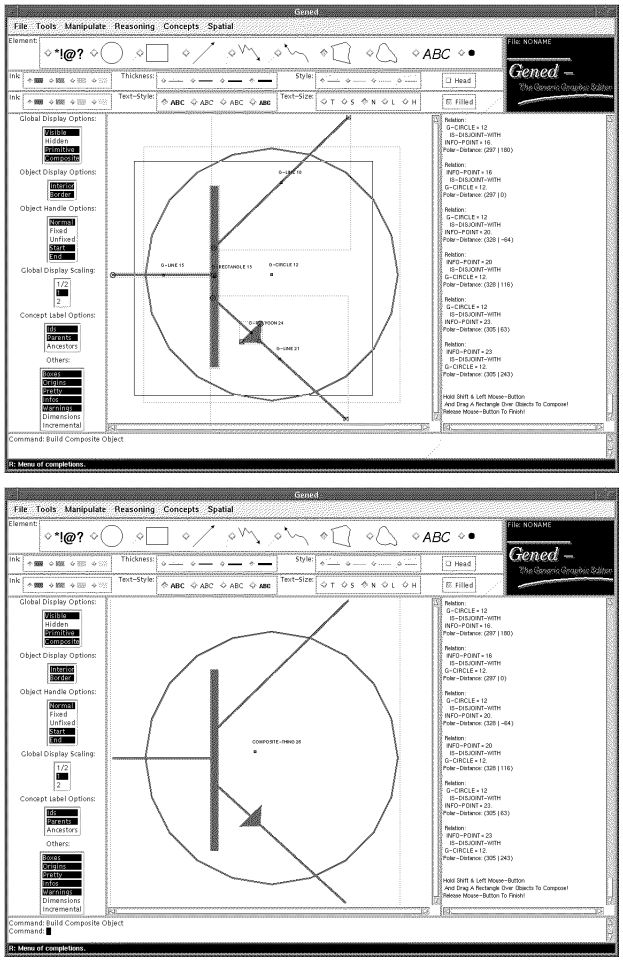


Abbildung 33: Bildung eines Kompositionsobjektes

können.

6.4 Interaktion mit Grafikobjekten

Ohne Möglichkeiten der *Manipulation* bereits existierender Grafikobjekte ist ein Grafikeditor natürlich nutzlos - schließlich kann es keinem Benutzer zugemutet werden, schon mit der Erzeugung die endgültige Position, Größe, relative Lage usw. zu allen anderen Objekten vorherzusehen. Die von GENED angebotenen *Interaktionsmöglichkeiten* werden deshalb kurz erläutert.

Eine fundamentale Operation ist z.B. das Verschieben eines Objektes - die Handlungssequenz ist offensichtlich:

1. Bestimme ein sensitives Objekt per Mausklick.
2. Das Objekt ist selektiert und folgt den Bewegungen der Maus.
3. Ein weiterer Klick befreit das Objekt - es hat nun seine neue Position.

Ähnliche Sequenzen erwartet der Benutzer auch für die im folgenden diskutierten Operationen, wie z.B. das Rotieren oder Skalieren eines Objektes.

6.4.1 Operationen auf Objekten

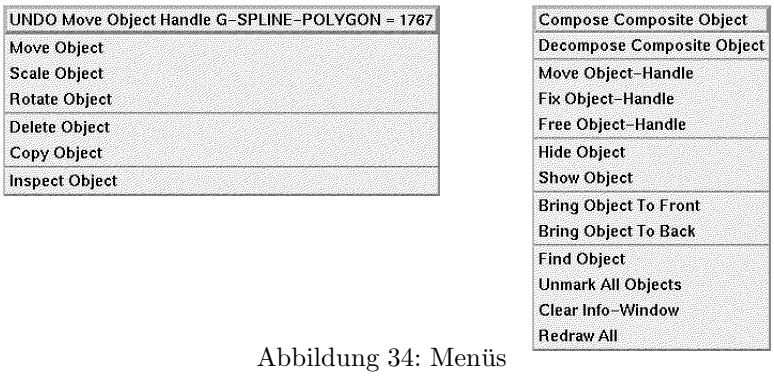


Abbildung 34: Menüs

Die wichtigsten Operationen auf Grafikobjekten werden in GENED mit einer sog. *Geste* initiiert: hierbei ist vom Benutzer die Maus über ein sensitives Objekt zu führen und eine Maustaste zu drücken. Teilweise muß er zusätzlich eine der Tasten Shift, Meta oder Control betätigen.

Dem objektorientierten Ansatz folgend, sind Grafikobjekte nun auch Objekte im Sinne der objektorientierten Programmierung (OOP). Objekte verstehen also Methoden, bzw. Operationen können auf sie wirken. Diese Operationen lassen sich somit nach Objekten sortieren (s. „Abstrakter Datentyp“).

Alle Objekte lassen sich also

- erzeugen,
- löschen,
- kopieren,
- verschieben,
- verstecken,
- sichtbar machen,
- inspizieren mit dem Inspektor,
- in den Vordergrund holen,
- in den Hintergrund tun,
- aggregieren,
- als Befestigungsobjekt für Manipulatoren benutzen,
- in die Bibliothek abspeichern und
- als Datei abspeichern.

Auf das Verstecken und Widersichtbarmachen wurde bereits eingegangen: hierbei handelt es sich um zwei verschiedene Sichtbarkeitswelten für Objekte. Während der Konstruktion eines Petrinetzes lassen sich so bestimmte Teile des Netzes ausblenden, um das Augenmerk auf die verbleibenden Netzbestandteile zu richten.

Da zweidimensionale Grafikobjekte gefüllt erscheinen können und somit eventuell andere verdecken, ist es oftmals notwendig, gerade die verdeckten Objekte zu manipulieren. Es wird also eine Operation benötigt, die ein verdeckendes Objekt hinter alle andere Objekte (also in den Hintergrund) stellt, und eine hierzu inverse Operation, die ein gerade verdecktes Objekt vor alle anderen Objekte (in den Vordergrund) befördert. Diese Operationen sind von Fenstersystemen hinlänglich bekannt.

Kompositionsobjekte lassen sich zusätzlich noch

- zerlegen (deaggregieren).

Es handelt sich um die inverse Operation zur Aggregation: die Aggregation wird aufgehoben, und die ehem. Teilobjekte werden wieder zu Individuen.

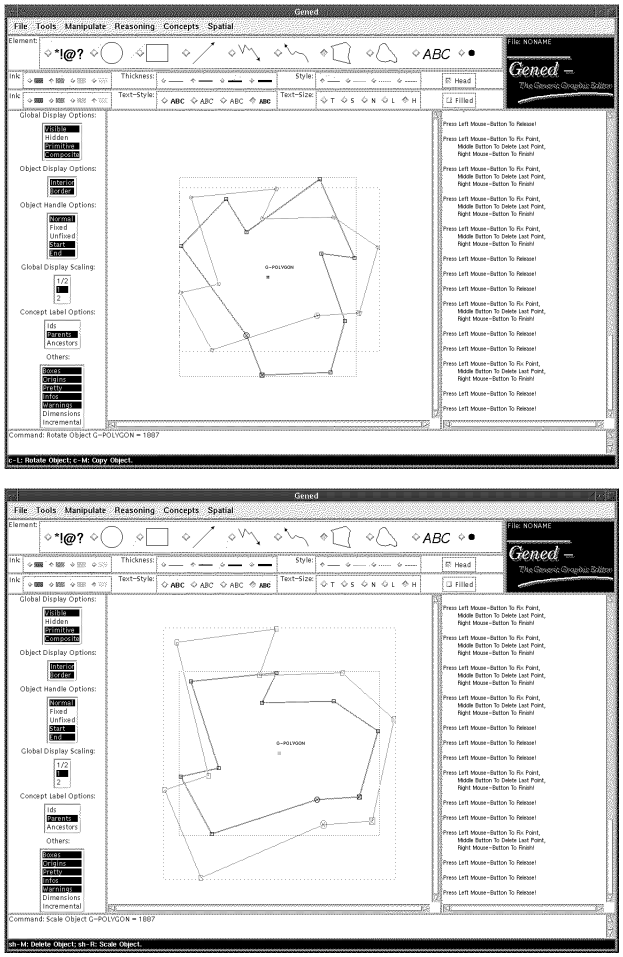


Abbildung 35: Rotationen und Skalierungen

Die Objekte $\{Rechteck, Kreis, Polygon, Spline\_Polygon\}$ lassen sich auch

- skalieren (bzgl. ihres lokalen Koordinatenursprungs, s. Abb. 35).

Ein Problem ensteht aufgrund des Schalters Global Display Scaling - evt. müssen nämlich *alle* Grafikobjekte entsprechend skaliert werden, auch Textelemente. Dies ist in der momentanen CLIM-Implementation leider nicht möglich, und folglich sind

die für die Grafikobjekte berechneten topologischen Relationen von der Stellung dieses Schalter abhängig. Dessen muß sich der Benutzer bewußt sein.

Alle Objekte außer $\{Kompositionsobjekt, Text, Punkt, Kreis\}$ lassen sich zusätzlich

- rotieren (bzgl. ihres lokalen Koordinatenursprungs, s. Abb. 35).

Punktmanipulatoren lassen sich an anderen Objekten

- befestigen, wodurch sie zu *befestigten Punktmanipulatoren (Fixed Object Handles)* werden,
- und wieder befreien, wodurch sie wieder zu *unbefestigten Punktmanipulatoren (Unfixed Object Handles)* werden.

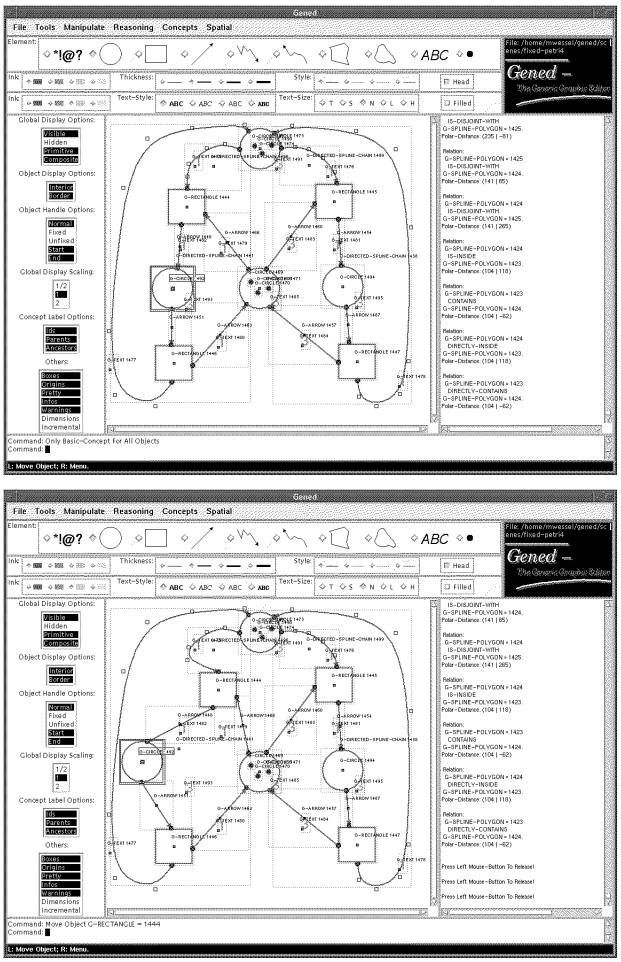


Abbildung 36: Vor und nach dem Verschieben zweier Objekte

Dieser Mechanismus ist z.B. für die Petrinetzkonstruktion nützlich: soll eine einzelne Stelle verschoben werden, so müssen auch die mit ihr in Beziehung stehenden Netzkanten und Marken mitverschoben werden. Das Mitverschieben der Marken kann durch Bildung eines Kompositionsobjektes erreicht werden. Die Netzkanten gehören jedoch nicht zur Stelle - sie sollten Individuen bleiben. Stattdessen können

nun die Netzkanten-Endpunktmanipulatoren (Start/End Handles) an dieser Stelle befestigt werden: bei jeder Positionsveränderung der Stelle werden die befestigten Punktmanipulatoren mitverschoben und die Objekte, zu denen sie gehören, neu ausgerichtet (normalisiert). Abb. 36 verdeutlicht diesen Vorgang.

6.4.2 Der Inspektor

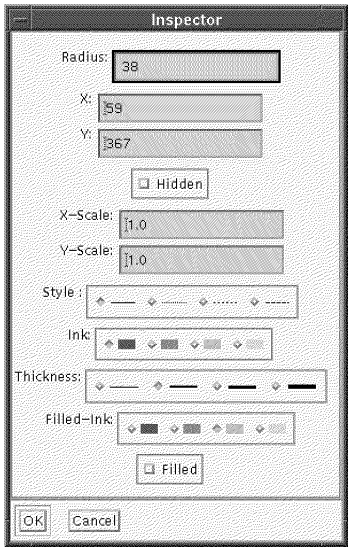


Abbildung 37: Inspektor

Oftmals bietet sich auch eine Interaktion über einen *Dialog* mit dem Benutzer an, um einige Objektattribute zu ändern: bei einem Textobjekt läßt sich der Text selbst am einfachsten durch einen Dialog mit dem Benutzer ändern, indem er zur Eingabe eines neuen Textes über die Tastatur aufgefordert wird. Für diese Art der Interaktion ist also ein *Inspektor* vorgesehen: mit ihm lassen sich einzelne Objektattribute einsehen und verändern. Abb. 37 zeigt einen typischen Inspektordialog. Koordinaten, die sehr genau sein müssen, lassen sich am besten textuell bestimmen.

Wiederum erleichtert der objektorientierte Ansatz die Implementierung: da Grafikobjekte CLOS-Instanzen sind, weiß jedes Objekt, welche seiner Slots in vernünftiger Form von außen veränderbar sind. Genau diese bietet der Inspektordialog dann zur Veränderung an. Der LISP-Code des Dialoges wird dabei dynamisch anhand des Typs des zu inspizierenden Objektes erzeugt bzw. aggregiert und dann ausgeführt. Bei einem Polygon ist es sicher nicht angebracht, die Liste der einzelnen Punkte dem Benutzer zur Inspektion anzubieten. Schließlich könnte er einzelne Punkte bequemer durch die Punktmanipulatoren verändern. Attribute wie Linienstil und -breite sowie Farbe lassen sich jedoch bequem vom Benutzer über den Inspektor verändern.

6.4.3 Interaktion mit Kompositionsobjekten

Die Interaktionsmöglichkeiten mit aggregierten Objekten sind gering: wie oben schon aufgezählt, lassen sie sich verschieben, abspeichern, in die Bibliothek eintragen, weiter aggregieren (Kompositionsobjekte können wiederum Teil anderer Kompositionsobjekte sein, es wird ein Baum gebildet), und schließlich wieder dekomponieren (und zwar schrittweise: die Baumwurzel wird entfernt und die dabei entstehenden Teibäume werden wieder als Individuen verfügbar).

Ursprünglich waren auch noch Rotation und Skalierung um den gemeinsamen Ursprung vorgesehen - dies bleibt eine zukünftige Erweiterungsmöglichkeit.

6.4.4 UNDO-Mechanismus

Ein *UNDO-Mechanismus* bietet die Möglichkeit, jeweils die letzten „n“ vom Benutzer ausgeführten Operationen zurückzunehmen. Solch ein Mechanismus ist für den Benutzer von großem Wert und steigert die Benutzungsfreundlichkeit sehr, da er es erlaubt, Operationen spielerisch und versuchsweise anzuwenden, ohne irreversible Zustandsveränderungen oder gar den Verlust von Objekten befürchten zu müssen. Dabei zeigen die meisten Programme in ihrem UNDO-Menüpunkt an, welche Operation als nächstes zurückgenommen werden kann. So auch GENED: wurde z.B. das Objekte `circle-123` bewegt, so steht im entsp. Menüpunkt UNDO Move Circle-123. Bei GENED sind grundsätzlich alle objektzustandsverändernden Operation zurücknehmbar (bis zu einer maximalen, jedoch frei einstellbaren Tiefe).

Der UNDO-Mechanismus ist mit Hilfe einer allgemeinen Kopierfunktion realisiert, die eine tiefe Kopie eines Objektes erzeugt: bevor eine zustandsverändernde oder gar objektlöschende Operation vorgenommen wird, wird das betroffene Objekt kopiert und als oberstes Element eines Stapelspeichers abgelegt.

6.5 GENED-Bibliothek

Die Nützlichkeit einer Bibliothek wurde bereits hinreichend diskutiert. Die GENED-Bibliothek arbeitet *konzeptorientiert:* sie wird vom `knowledge`-Package, das die Wissensbasis enthält, durch eine Konstante namens `+library-concepts+` über die sinnvoll in einer Bibliothek zu speichernden Konzeptvisualisierungen informiert. Die Konstante `+known-concepts+` enthält alle GENED bekannten Konzeptnamen. Hinter einem Bibliothekskonzept können sich verschiedene Visualisierungen, also unterschiedliche CLOS-Grafikobjekte verbergen, womit die oben angesprochene Variierung erreicht wird: es können verschiedene Visualisierungen des Konzeptes Transistor gespeichert werden. Abb. 38 zeigt das Bibliotheksmenü Select Concept Visualization.

6.5.1 Interaktion mit der GENED-Bibliothek

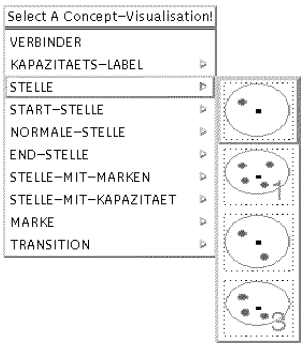


Abbildung 38: Bibliothek

Zunächst einmal müssen überhaupt erst irgendwelche Objekte in der Bibliothek eingetragen sein. Alsdann kann der Benutzer unter dem Menüpunkt

Select Concept Visualization eine ihm angemessen erscheinende Instanz bzw. Visualisierung eines Bibliothekskonzeptes auswählen (s. Abb. 38). Befindet sich GENED im *!@?-Modus, so wird das nächste vom Benutzer erzeugte Grafikobjekt eine Kopie der aus der Bibliothek gewählten Visualisierung sein. Wie bereits erwähnt sind diese Visualisierungen nur memorierte CLOS-Instanzen - die Objekte bzw. Kopien haben jedoch entsprechende Zustandsinformationen, Instanzen welcher Konzepte sie laut Bibliothek sein sollen. Generell hat jedes CLOS-Grafikobjekt eine Liste von momentanen Eltern- und Vorgängerkonzepten - dies gilt auch für Bibliotheks-Visualisierungen. Anhand dieser Listen werden korrespondierenden CLASSIC-Individuen erzeugt (was teils sofort, teils verzögert geschieht, s. inkrementeller Modus), welche dann per Zusicherung die korrekten Konzepte haben (s. auch Kap. 8).

Die gesamte Bibliothek läßt sich löschen, als Datei abspeichern und wieder laden. Einzelne Konzeptvisualisierungen können entfernt und hinzugefügt werden: Einträge in die Bibliothek werden gemacht, indem der Benutzer den Menüpunkt Store Object Into Library auswählt und das entsprechende Grafikobjekt selektiert. Alsdann wird er aufgefordert, aus der Liste der Elternkonzepte des Grafikobjektes das Konzept zu wählen, unter welchem es als Konzeptvisualisierung in der Bibliothek erscheinen soll. Dies kann beliebig oft mit allen Elternkonzepten des Objektes geschehen. Dabei sei darauf hingewiesen, daß Objekten auch von Hand Elternkonzepte zugegewiesen werden können, ohne einen Klassifikationsprozeß durch CLASSIC.

6.6 CLASSIC-Anbindung

Da die *Konfigurierbarkeit* und somit mögl. Unterstützung für die Erstellung von Programmen für VPs usw. bei GENED durch die Verwendung einer austauschbaren Wissensbasis gegeben ist, wird an dieser Stelle beschrieben, wie CLASSIC und GENED zusammenarbeiten, um die gewünschte Funktionalität (s. Kap. 2) zu erreichen.

Zwischen CLASSIC und GENED ist ein *beidseitiger Informationsfluß* notwendig: so will der Ersteller eines Petrinetzes wissen, ob seine Grafikobjekte korrekt klassifiziert worden sind (also Kreise zu Stellen, Rechtecke zu Transitionen, etc.). GENED zeigt ihm auf Wunsch die Eltern- und/oder Vorgängerkonzepte (über die Schaltergruppe Concept Label Options einstellbar) der korrespondierenden CLASSIC-Individuen an. Diese Information kann von CLASSIC mit entsp. Funktionsaufrufen erfragt werden. In Abb. 29 sind die Grafikobjekte mit entsprechenden Konzeptbezeichnern beschriftet. Die Vorgängerkonzepte erscheinen kursiv. Die Klassfikation der Grafikobjekte durch CLASSIC (Klassifizierungsphase) wird durch den Menüpunkt Classic, classify! angestoßen. Anderseits muß alle Information über die vom Benutzer erstellte grafische Konstellation CLASSIC übermittelt werden.

Zusätzlich sind teilweise Konsistenzprüfungen notwendig: GENED läßt es zu, einem Objekt versuchsweise jedes beliebige Konzept zuzuweisen. Dabei kann es natürlich zu Inkonsistenzen auf CLASSIC-Ebene kommen, woraufhin GENED diese Konzeptkonjunktion nicht zulassen sollte. So sind z.B. (s. Kap. 4) die Konzepte `g-rectangle` und `g-circle` als sich wechselseitig ausschließend primitiv definiert - der Benutzer sollte also eine Fehlermeldung erhalten, wenn er versuchte, einem Kreis (der schon mit der Erzeugung das Elternkonzept `g-circle` hat) zusätzlich das Konzept `g-rectangle` zuzusichern. Tatsächlich erhält der Benutzer diese Meldung sofort nur im inkrementellen Modus; ansonsten wird sie bis zur Klassifizierungsphase verzögert.

6.6.1 Der inkrementelle Modus

Befindet sich GENED im inkrementellen Modus, so zieht jede Zustandsänderung eines CLOS-Objektes sofort auch eine entsp. Änderung des korrespondierenden CLASSIC-Individuums nach sich. Normalerweise befindet sich GENED jedoch im nichtinkrementellen, *normalen Modus:*

In diesem werden alle CLASSIC-Individuen genau einmal verwendet, und es muß somit keine Information jemals zurückgenommen oder revidiert werden. Somit sind auch keine nichtmonotonen Änderungen der Wissensbasis erforderlich. Wählt der Benutzer den Menüpunkt Classic, classify!, so wird für jedes CLOS-Grafikobjekt ein korrespondierendes CLASSIC-Individuum erzeugt, und alle Rollen werden der grafischen Konstellation entsprechend gefüllt, indem die vom Geometriemodul berechneten topologischen Relationen den CLASSIC-Individuen zugesichert werden, etc. Wird dieser Menüpunkt das nächste Mal gewählt, so werden *neue* CLASSIC-Individuen erzeugt, womit alle seitdem erfolgten Veränderungen der grafischen Konstellation keine Relevanz für die „alten“ CLASSIC-Individuen mehr haben. Sie werden (da nun keine Referenzen mehr existieren) nach einiger Zeit automatisch vom „Garbage Collector“ des LISP-Systems vernichtet.

Im *inkrementellen Modus* ist eben dies nicht der Fall: ändert sich die Position eines Grafikobjektes (die CLOS-Slots `x-position`, `y-position`), so werden auch die entsp. Attribute der CLASSIC-Individuen sofort verändert. Es werden *keine* neuen CLASSIC-Individuen erzeugt, so daß die Veränderung der grafischen Konstellation in den CLASSIC-Individuen reflektiert werden muß - Rollen und Attribute müssen aktualisiert werden, da es sonst zu Inkonsistenzen kommen würde. Alle topologischen Relationen, in denen das Objekt stand, werden neu berechnet. Berührte z.B. Objekt A ein Objekt B und wurde Objekt A so verschoben, daß sich A und B nun schneiden, so muß entsprechend Information subtrahiert und addiert werden: $\ominus touches(A, B) \wedge \oplus intersects(A, B)$. Der Füller A der Rolle `touches` des Objektes B muß also entfernt und dafür in die Rolle `intersects` eingetragen werden.

Für den inkrementellen Modus müssen somit Funktionen vorhanden sein, die alle Differenzen zwischen CLOS-Objekten bzw. beliebigen Konstellationen von CLOS-Objekten erkennen und protokollieren können. Diesen Dienst leistet das `Delta`-Modul: Es memoriert von Aufruf zu Aufruf alle für die CLASSIC-Rollen und Attribute relevanten momentanen CLOS-Slots, wie z.B. `x-position`, `y-position` sowie alle Slots, in denen die topologischen Relationen objektzentriert vermerkt werden. Beim nächsten Aufruf vergleicht es die momentanen Slot-Füller der CLOS-Objekte mit den memorierten Slot-Füllern und protokolliert entsprechend alle Abweichungen, also ob Füller hinzugekommen oder entfernt wurden ($\oplus$, $\ominus$). Die `Delta`-Funktion wird nach jeder zustandsverändernden Operation aufgerufen: auch hier erleichtert die objektorientierte Vorgehensweise wieder die Programmierung, da alle Objekte wissen, welche ihrer Slots von der `Delta`-Funktion zu memorieren sind. Wiederum sind die Funktionen so programmiert, daß GENED konfiguriert werden kann, indem Slot-Namen in Listen geschrieben werden, so daß keine weiteren Programmänderungen nötig sind. Die `Delta`-Funktion muß CLASSIC-Individuen auch löschen und erzeugen können - sie kontrolliert und steuert also den inkrementellen Informationsfluß zwischen CLASSIC und GENED im inkrementellen Modus.

Da das Problem der Nichtmonotonie solcher Zustandsveränderungen der Wissensbasis in CLASSIC durch den Begriff der geschlossenen bzw. offenen Rolle umgangen wird, müssen die betroffenen Rollen offen sein. In angemessener Zeit funktioniert der inkrementelle Modus nur, wenn alle Rollen aller Individuen bereits offen sind und nicht erst noch geöffnet werden müssen. Das Schließen und Öffnen der Rol-

len sind die zeitaufwendigsten Operationen in CLASSIC, da beim Schließen fast alle Inferenzen durchzuführen sind und diese beim Öffnen wieder zurückgenommen werden, wobei sehr komplizierte Abhängigkeiten bestehen. Zudem läßt sich nicht jede Information zurücknehmen (abgesehen davon, daß die Rollen offen sein müssen): von CLASSIC selbst inferierte Information (Derived Information) kann nicht zurückgenommen werden, da es sonst zu Inkonsistenzen in der Wissensbasis kommen würde (s. Kap. 4). Stattdessen muß die Antezedenz dieser Inferenz entfernt werden (Told Information), womit CLASSIC automatisch die Konsequenz zurückzieht. Dies funktioniert insbesondere auch für Regeln, die aufgrund des Makros `defqualifiedsubrole` in GENED massiv verwendet werden.

6.6.2 Interaktion mit CLASSIC

Grundsätzlich kann ein Individuum auf zwei verschiedene Arten Instanz eines Konzeptes werden:

1. Das Konzept subsumiert das Individuum, was von CLASSIC gefolgert wurde, oder
2. Der Benutzer sichert das entspr. Konzept für das korrespondierende Grafikobjekt manuell von Hand zu, was CLASSIC als Zusicherung (Tell) kommuniziert wird.

Beide Formen werden von GENED unterstützt, wobei bei letzterer eine Konsistenzprüfung vorgenommen wird. Dabei wird eine Inkonsistenz im inkrementellen Modus sofort entdeckt und gemeldet, ansonsten jedoch verzögert (in der Klassifizierungsphase).

Die Interaktion sieht so aus, daß der Benutzer den Menüpunkt Assign Concept To Object wählt und ein Grafikobjekt mit der Maus selektiert, woraufhin ein Menü erscheint, welches ihn auffordert, ein weiteres Elternkonzept für dieses Objekt aus der Liste `+known-concepts+` zu bestimmen.

Das gewählte Konzept wird somit in die Liste der Elternkonzepte des Objektes aufgenommen und im inkrementellen Modus CLASSIC sofort mitgeteilt. Tatsächlich werden im inkrementellen Modus die Slots `parent-concepts` und `ancestor-concepts` so verwaltet, daß das neue Konzept versuchsweise zu `parent-concepts` hinzugefügt wird, CLASSIC diese Information zugesichert wird, und dann die Eltern- und Vorgängerkonzepte von CLASSIC mit den Funktionen `get-parent-concepts, get-ancestor-concepts` erneut erfragt werden. Diese Information wird an die Grafikobjekte geschrieben. Inkonsistenzen werden von CLASSIC direkt im Infofenster von GENED dargestellt.

Zudem gibt es Menüpunkte, um ein Konzept aus der Liste der Elternkonzepte zu entfernen, einem Objekt ausschließlich ein Elternkonzept zu geben, und um einem Objekt ausschließlich das initiale Elternkonzept zu geben `(type-of object)`.

Der Benutzer sollte eine Möglichkeit haben, das CLASSIC-Individuum selbst zu inspizieren: eine Mausgeste führt die Funktion `print-classic-individual` auf dem korrespondierenden CLASSIC-Individuum aus. Da die Endpunkte von eindimensionalen Objekten ebenfalls als CLASSIC-Individuen existieren (um Verbinder klassifizieren zu können), kann diese Mausgeste auch auf Start- bzw. Endpunktmanipulatoren ausgeführt werden.

Weiterhin kann der GENED-Benutzer alle Rollen aller Individuen schließen und auch wieder öffnen, wovor er gewarnt sei, da diese Operationen sehr zeitaufwendig sind.

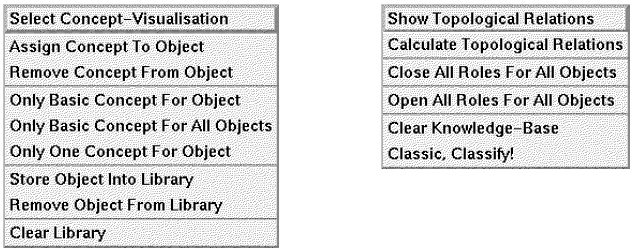


Abbildung 39: Konzeptunterstützung

Die eigentliche Klassifizierung erfolgt dann über den Menüpunkt Classic, classify!. Im inkrementellen Modus ist alle Information bereits vorhanden, so daß nur noch die Rollen der CLASSIC-Individuen geschlossen werden müssen, wodurch die meisten Inferenzen durchgeführt werden. Im normalen Modus werden hingegen erst die topologischen Relationen berechnet, dann die Individuen erzeugt, deren Rollen gefüllt, und schließlich alle Rollen geschlossen. Nach Beendigung der Klassifizierungsphase werden die Eltern- und Vorgängerkonzepte aller Objekte erfragt und Grafikobjekte eventuell adäquat (in Abhängigkeit von Concept Label Options) beschriftet, wodurch der Benutzer sehen kann, ob alles korrekt klassifiziert wurde. Angezeigt werden jedoch nur die Konzepte, die in der Liste `+known-concepts+` stehen (s. auch Abb. 39).

6.7 Anbindung des Geometriemoduls

Das Geometriemodul ist gemäß der formalen Spezifikationen in Kap. 3 implementiert. Dabei berechnet das Geometriemodul also Basisbeziehungen, worauf aufbauend dann andere Algorithmen weitere Relationen extrahieren können (z.B. *directly_contains*). Diese Algorithmen sind nicht im Geometriemodul, sondern im `Spatial`-Modul von GENED selbst beheimatet.

Für jedes CLOS-Grafikobjekt muß eine für das Geometriemodul geeignete Repräsentation erzeugt werden: diese müssen Instanzen der im Geometriemodul für diesen Zweck definierten Klassen sein. So müssen z.B. alle Kreise als Instanzen der Klasse `geom-polygon` für das Geometriemodul polygonisiert werden.

GENED ruft nun sukzessive für alle Objektpaare (A, B) die Funktion `relate-object-to-object(A,B)` (s. Kap 3) auf, welche ein Symbol zurückgibt, das die soeben berechnete Relation bezeichnet. Entsprechend wird diese Information in den CLOS-Slots der Objekte memoriert.

An dieser Stelle muß folgende Bemerkung gemacht werden: Objekte können sich gegenseitig verdecken - das Geometriemodul von GENED behandelt jedoch alle Objekte so, als ob sie transparent wären. Dies muß der Benutzer beachten, wodurch auch eine Einschränkung des zuvor gestellten WYSIWYG-Zieles gegeben ist. Ebenso kann die Liniendicke oder Linienart einen Einfluß auf die Interpretation eines menschlichen Betrachters haben - das Geometriemodul ignoriert diese Attribute jedoch, sowie auch die Farbe eines Objektes an dieser Stelle keine Rolle spielt.

6.7.1 Topologische Anfragen

Da die bereits berechneten topologischen Relationen objektzentriert in den Objekten gespeichert wurden (so enthält z.B. der Slot `intersects-objects` eines CLOS-Objektes Referenzen auf alle Objekte, die es schneidet), ist es nun möglich, topologische Anfragen an bestimmte Grafikobjekte zu stellen: der GENED-Benutzer wählt z.B. den Menüpunkt Show Intersecting Objects, selektiert dann das gefragte Grafikobjekt, und alle dieses Objekt schneidenden Objekte werden markiert. Implementiert sind Anfragen für alle primären Basisrelationen sowie einige sekundäre Relationen (wie *directly_inside*, *linked_over_with*, *start_linked_over_with* etc.). Abb. 40 zeigt die möglichen Anfragen, Abb. 41 hingegen das Ergebnis einer topologischen Anfrage der Art Show Containing Objects an eine Petrinetzstelle. Die Marken sind mit einem dicken Rechteck markiert.

6.7.2 Topologische Relationen für Kompositionsobjekte

Es wurde ja bereits in Kap. 3 erwähnt, daß das Geometriemodul nur Relationen für einfache Objekte berechnet. Alle aggregierten Objekte müssen in ihre Teilkomponenten zerlegt werden, und für diese werden dann die Relationen berechnet. Nach der Berechnung werden die Teilobjekte wieder zusammengefügt, und bestimmte Algorithmen bestimmen dann anhand der berechneten Relationen zwischen den Komponenten die Relationen für die Kompositionsobjekte.

Da Komponentenobjekte Teil einer „Black Box“ sind (von der immerhin noch die Komponenten bekannt sind), werden alle topologischen Relationen, die Komponenten involvieren, gelöscht. Allerdings existieren die Komponenten auf CLASSIC-Ebene, da Aussagen über die *has_parts*-Relation wünschenswert sind (s. Kap. 4: dort ist ein Petrinetz als Kompositionsobjekt definiert, dessen Teilkomponenten alle Netzelemente sind).

Es wäre wünschenswert, für aggregierte Objekte auch topologische Relationen zwischen den Komponenten *untereinander* zu berechnen, sie also wie andere Objekte auch zu betrachten - aufgrund der Baumstruktur der Kompositionsobjekte wird der Rechenaufwand jedoch sofort exponentiell, denn Kompositionsobjekte können selbst wieder andere Kompositionsobjekte enthalten. Alle möglichen Baumebenen (wobei Blätter primitive Objekte sind und Knoten Kompositionsobjekte) müssen zu allen möglichen Baumebenen in Relation gesetzt werden, um dies zu leisten. Die Idee wurde daher nicht weiter verfolgt. Möglich wäre, eine derartige Berechnung bis zu einer bestimmten Tiefe oder aber für ausgezeichnete Komponenten durchzuführen, die sozusagen als „Ports“ für die Außenwelt dienen. Momentan können in GENED also keine Konzeptdefinitionen verwendet werden, die Aussagen über Kompositionsobjekte und die Beziehungen ihrer Komponentenobjekte zueinander benötigen.

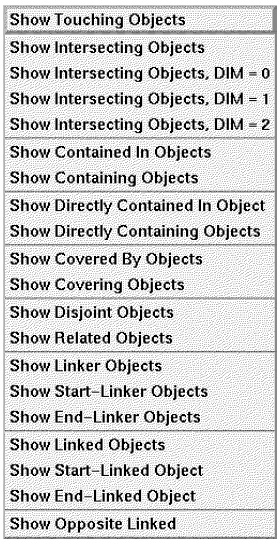


Abbildung 40: Mögliche topologische Anfragen

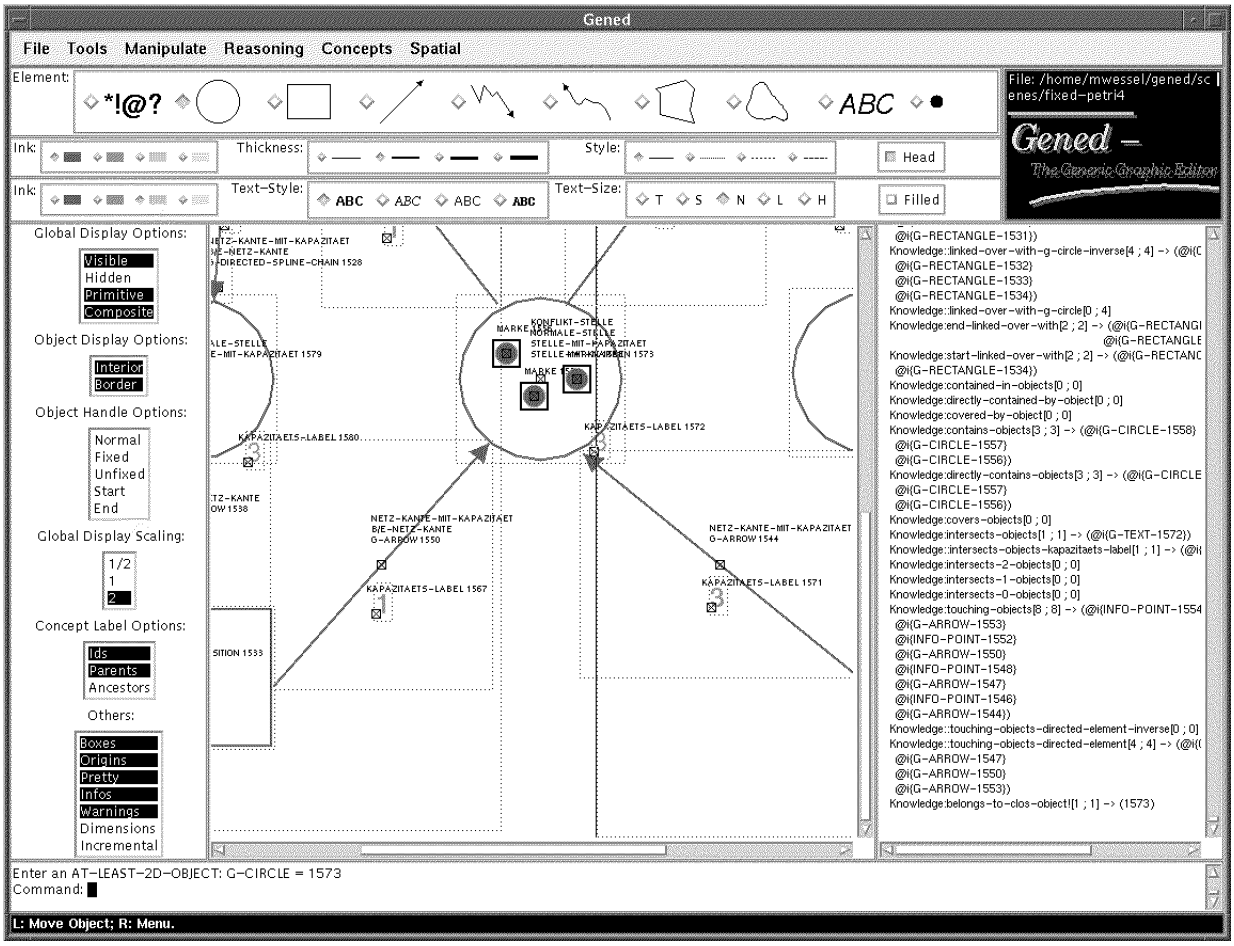


Abbildung 41: Ergebnis einer topologischen Anfrage - Marken sind hervorgehoben

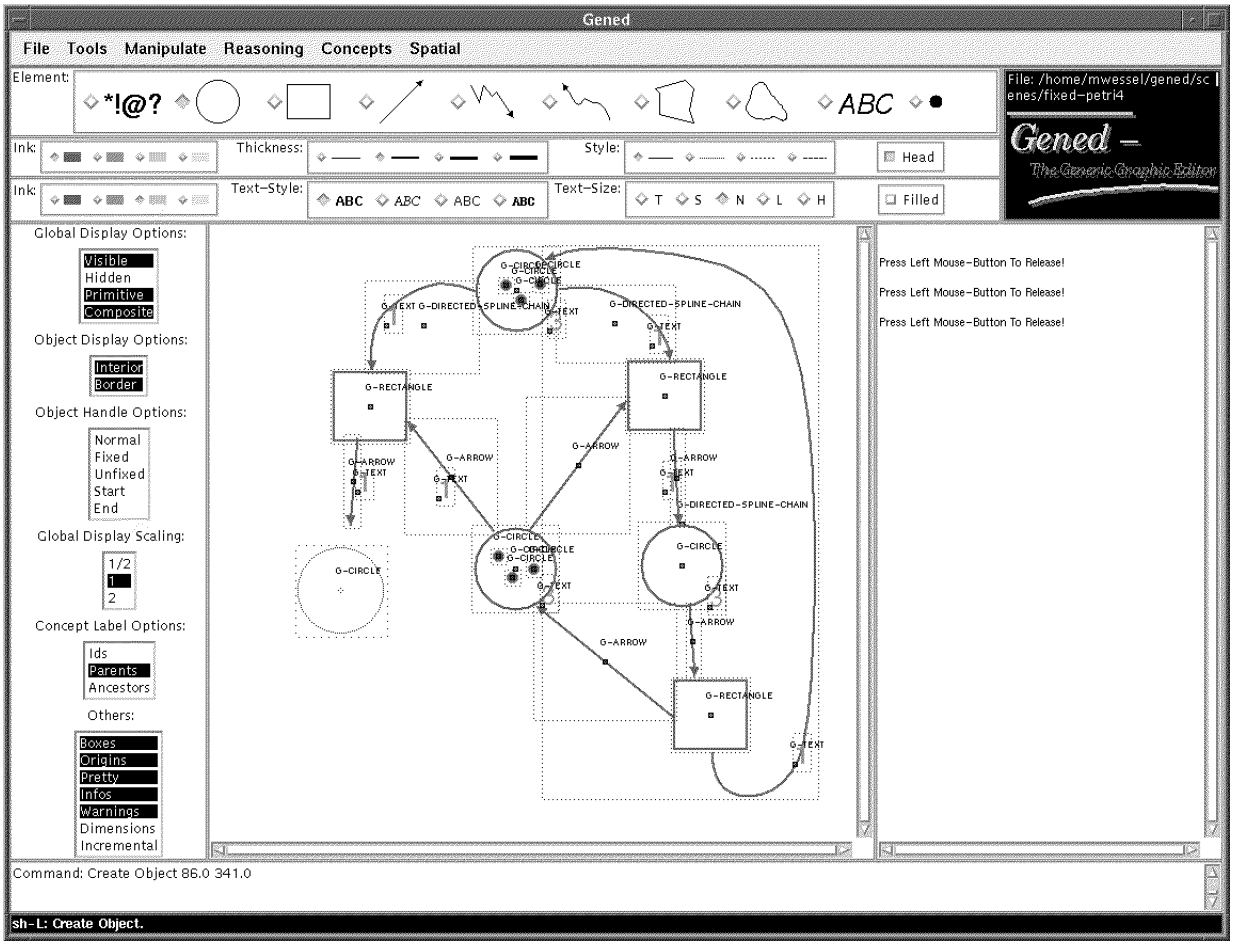


Abbildung 42: Erstellung einer Grafik

7 Ein Benutzungsbeispiel: Petrinetze

7.1 Motivation

Für die erfolgreiche GENED-Benutzung sind einige Hinweise notwendig, weswegen hier ein eigenes Kapitel zu diesem Thema erscheint.

7.2 Erstellung einer Wissensbasis

Für die Erstellung einer anwendungsspezifischen Wissensbasis muß der Benutzer zuerst CLASSIC lernen, wobei ihm vielleicht Kap. 4 hilft. Die Anpassung der für die Kommunikation des `Knowledge`-Moduls mit GENED nötigen Konstanten ist in Kap. 8 beschrieben.

7.3 Erstellung einer Grafik

Der Benutzer erstellt mit Hilfe der angebotenen Grafikprimitive eine grafische Konstellation. Abb. 42 zeigt den Benutzer beim Erzeugen eines Kreises als weitere Stelle

für ein schon erkennbares Petrinetz. Befindet sich GENED im inkrementellen Modus, so werden alle Zustandsänderungen durch Handlungen des Benutzer sofort an CLASSIC kommuniziert, was teilweise etwas Zeit benötigt, bis der Benutzer die nächste Handlung durchführen kann. Außerdem werden die topologischen Relationen - so weit wie nötig - stetig neu berechnet. Ist der Infos-Schalter an, so wird der Informationsfluß zwischen CLASSIC und GENED im Infofenster angezeigt, was auch zeitaufwendig ist, da teilweise sehr viel Information kommuniziert werden muß. Eventuell sollte der Benutzer den inkrementellen Modus ausschalten, oder zumindest den Schalter Infos.

7.4 Klassifizierung der Objekte

Ist die Grafik erstellt, so ist es sinnvoll, zunächst einige topologische Anfragen zu stellen, um zu validieren, daß sich die Grafikobjekte auch tatsächlich in den vom Benutzer intendierten Relationen befinden. Ansonsten kann es passieren, daß nach Meinung des Benutzers einige Grafikobjekte anscheinend nicht korrekt klassifiziert werden - eine Beziehung zwischen A und B, die der Benutzer für eine $touches(A, B)$-Relation hält, kann in Wirklichkeit eine $intersects(A, B)$-Beziehung sein. Wenn A und B sich nur ein oder zwei Pixeln überschneiden, so ist die Entdeckung dieser Tatsache für den menschlichen Benutzer per Auge recht schwierig. Durch entsprechende topologische Anfragen kann er jedoch überprüfen, ob die berechneten Relationen mit seinen Vorstellungen übereinstimmen. Im normalen Modus müssen jedoch zuvor die topologischen Relationen über den Menüpunkt Calculate Topological Relations berechnet worden sein. Dies ist auch nach jeder Zustandsänderung irgendwelcher Objekte nötig, da gespeicherte Relationen eventuell nicht mehr gültig sind (da veraltet). Der Menüpunkt Show Topological Relations hat die gleiche Wirkung, gibt aber eine u.U. sehr lange Liste der topologischen Relationen im Infofenster aus (sie wächst quadratisch mit der Anzahl der Grafikobjekte), weswegen dieser Menüpunkt mit großer Vorsicht zu gebrauchen ist, da GENED sonst evtl. erst wieder nach Minuten benutzbar wird. Im inkrementellen Modus ist es hingegen nicht notwendig, einen dieser Menüpunkte anzuwählen, da die Relationen stetig neu berechnet werden. Anfragen können somit zu jedem Zeitpunkt gestellt werden.

Nachdem der Benutzer also mit den Relationen zufrieden ist, wählt er (in welchem Modus GENED sich auch befindet) den Menüpunkt Classic, Classify!. Eine Klassifikation des hier dargestellten Petrinetzes (64 Objekte) benötigt ca. 6 bis 7 Minuten auf einem „Sparc 10“-Rechner. Auf einer „Sparc Classic“ wird das Programm gänzlich unbenutzbar. Das korrekt klassifizierte Netz wird in Abb. 43 gezeigt. Nach einer Klassifikation sollte der Benutzer auf jeden Fall den inkrementellen Modus ausschalten, da sonst z.B. eine Skalierung des Entwurfsfensters oder eine einfache Verschiebung eines einzelnen Objektes eine sehr lange Wartezeit verursacht, denn alle Rollen sind nach der Klassifikation geschlossen. Eine Skalierung oder sonstige Zustandsveränderung irgendeines Objektes im inkrementellen Modus würde viel Information verändern - Rollen müssten wieder geöffnet und CLASSIC-Individuen entsprechend aktualisiert werden. Selbst ein einfaches Verschieben bedeutet dann eine Wartezeit von mehreren Minuten. Im normalen Modus tritt dieses Problem nicht auf.

7.5 Inspektion der Individuen und Ergebnisse

Wurde der inkrementelle Modus also ausgeschaltet, so kann der Benutzer das Entwurfsfenster skalieren und so die Konzeptbeschriftungen an den Grafikobjekten ge-

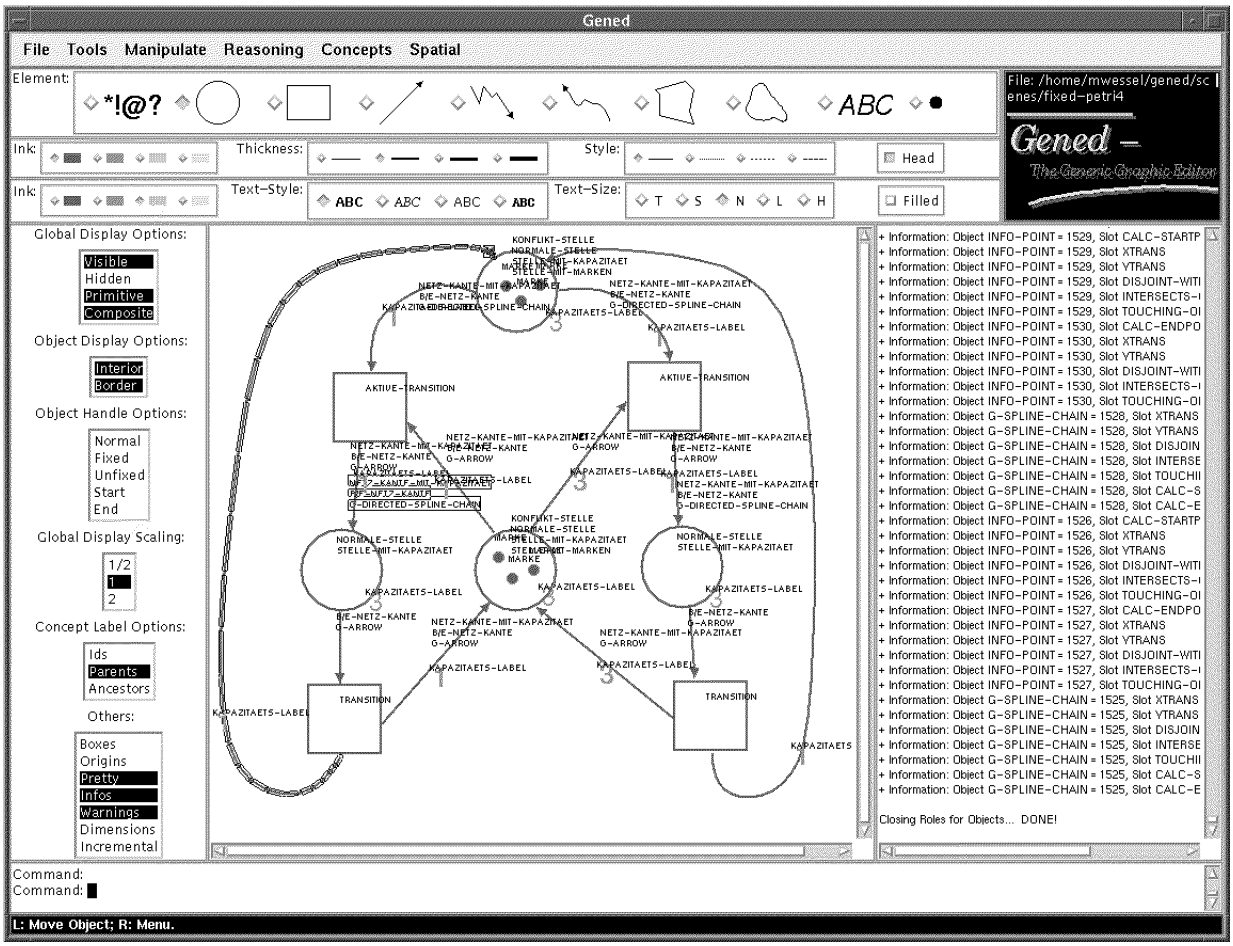


Abbildung 43: Das korrekt klassifizierte Netz

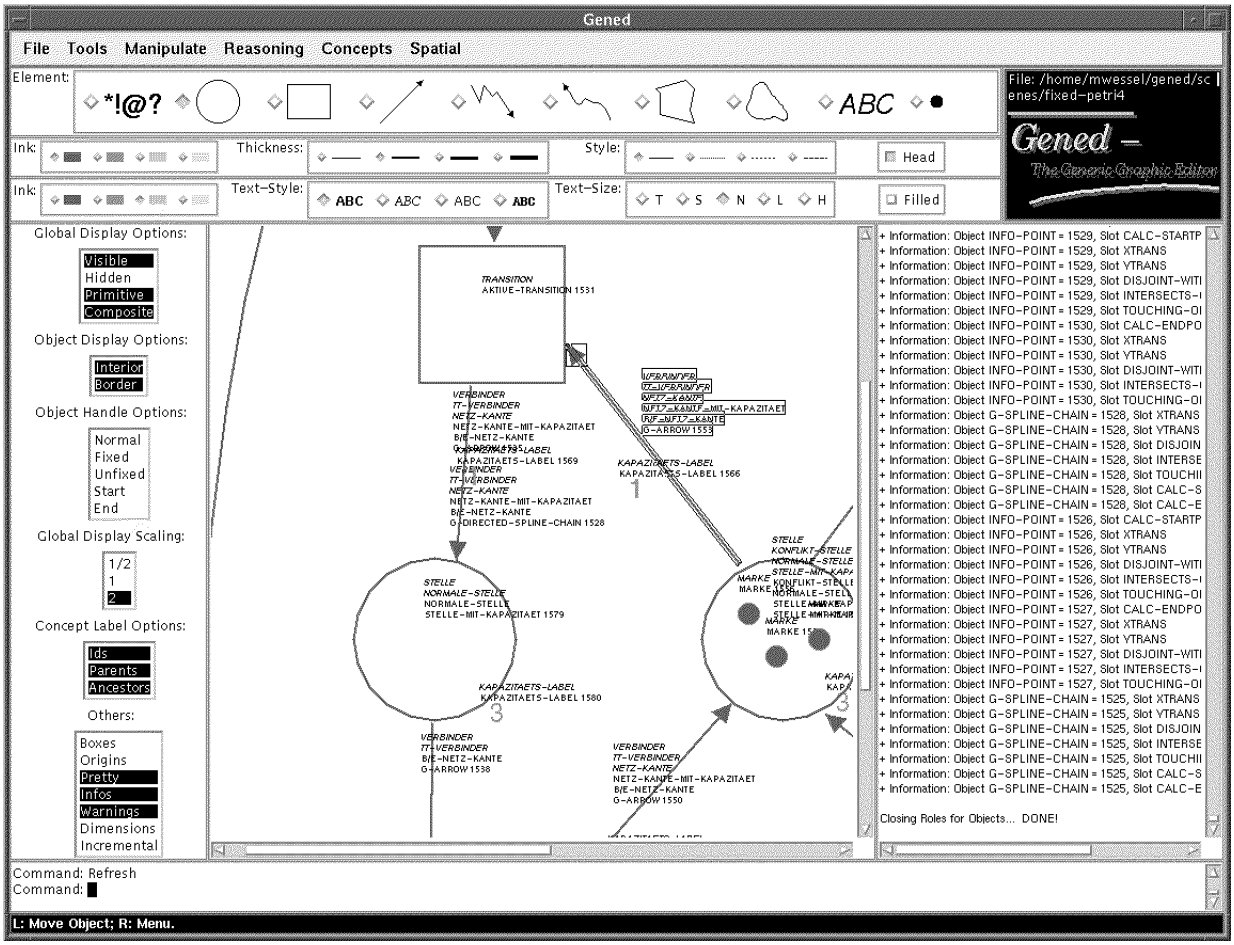


Abbildung 44: Genaue Inspektion durch Skalierung

nauer inspizieren, s. Abb. 44.

Sinnvoll ist nun eine Inspektion einzelner CLASSIC-Individuen, was per Mausgeste durchgeführt werden kann: der Benutzer erhält eine CLASSIC-Beschreibung des Individuums im Infofenster. Sollte etwas nicht entsprechend seinen Vorstellungen klassifiziert worden sein, so läßt sich in Verbindung mit den topologischen Anfragen und der Inspektion der CLASSIC-Individuen der Fehler meist recht schnell finden. Eventuell führt die so gewonnene Einsicht zur Redefinition von Konzepten o.ä.

7.6 Erstellung einer Bibliothek für spätere Verwendung

Die Bildung einer Bibliothek setzt in der Regel erfolgreich klassifizierte Objekte voraus - nur so kann der Benutzer sicher sein, daß die Objekte auch den Definitionen der Konzepte genügen, als deren Visualisierung er sie in die Bibliothek abspeichert. Besteht kein Zweifel, daß ein Grafikobjekt Instanz eines bestimmten Konzeptes ist, so kann der Benutzer dem Grafikobjekt dieses Konzept manuell zuweisen, so daß keine Klassifikation durch CLASSIC notwendig ist. Ansonsten entstünden jedoch Inkonsistenzen.

Wenn der Benutzer eine Visualisierungen eines Konzeptes aus der Bibliothek in-

stantiiert (wenn er z.B. einen Transistor benutzen will), so hat diese Kopie genau die Eltern- und Vorgängerkonzepte, die das ursprüngliche Grafikobjekt zu dem Zeitpunkt hatte, als es in die Bibliothekt übernommen wurde - es ist also schon „klassifiziert“. Hier tritt folgendes Problem auf: so wurde z.B. das Konzept Stelle-mit-Marken-und-Kapazität als Spezialisierung des Konzeptes Stelle-mit-Marken, dieses wiederum als Spezialisierung von Stelle und dieses als Spezialisierung von Kreis definiert. Im wesentlichen wird also ein Kreis zur Stelle-mit-Marken-und-Kapazität klassifiziert. Wird jedoch der zur Stelle-mit-Marken-und-Kapazität klassifizierte Kreis in die Bibliothek eingetragen ohne die dazugehörigen Marken und das Kapazitäts-Label, so erfüllt diese Visualisierung, wenn sie wieder aus der Bibliothek kopiert wird, sicherlich nicht die Definition einer Stelle-mit-Marken-und-Kapazität, da die anderen Objekte, mit denen es per Definition in Relation stehen muß, fehlen. Das Objekt wird also bei einer Klassifikation inkonsistent. Unkritisch sind sicherlich Grafikobjekte, die Instanzen primitiver Konzepte sind - die Visualisierungen nichtprimitiver Konzepte sind jedoch kritisch.

Eine Abhilfe des Problems schafft folgendes Vorgehen:

> nachdem ein Kreis von CLASSIC korrekt zur Stelle-mit-Marken-und-Kapazität klassifiziert wurde, bildet der Benutzer ein Kompositionsobjekt und weist diesem von Hand per [Assign Concept To Object] das Konzept Stelle-mit-Marken-und-Kapazität zu. Alsdann trägt er das Kompositionsobjekt unter diesem Konzept in die Bibliothek ein. Der Benutzer muß nun beim Instantiieren diese Objektes aus der Bibliothek nur noch daran denken, das Objekt wieder zu zerlegen, denn schließlich ist das Konzept Stelle-mit-Marken-und-Kapazität nicht als Spezialisierung des Konzeptes Kompositionsobjekt definiert: das aggregierte Objekt verschwindet also, und die Komponenten - u.a. auch der oben diskutierte Kreis - haben das korrekte Konzept. Die Komponenten stehen außerdem untereinander in den notwendigen Relationen, so daß der große Kreis sicherlich die Konzeptdefinition von Stelle-mit-Marken-und-Kapazität erfüllt. Das Kompositionsobjekt bildet also nur einen Transportbehälter bzw. eine Art „Verpackung“, um die einzelnen notwendigen Teilkomponenten unter einer adäquaten Bezeichnung und zusammengehörig in der Bibliothek zu verwahren. Die Operation [Assign Concept To Object] hat nur den Zweck, eben diese Bezeichnung zu erzeugen. Die im Kompositionsobjekt verwahrten Komponenten selbst haben jedoch - durch vorherige Klassifikation durch CLASSIC - die korrekten Bezeichnungen bzw. Elternkonzepte.

Hier stellt sich die Frage, ob die Wissensbasis diese Konzepte nicht hätte anderes definieren sollen, nämlich das Konzept Stelle-mit-Marken-und-Kapazität als Spezialisierung eines Kompositionsobjektes und nicht als Spezialisierung eines Kreises. Da GENED jedoch keine topologischen Relationen für und zwischen Komponenten von aggregierten Objekten berechnet, könnten keine so komplizierte Konzeptdefinitionen wie

> „alle Komponenten, die Marken sind, sind in der größten Kreis-Komponente enthalten, und es gibt genau eine Kapazitäts-Label-Komponente, die die größte Kreis-Komponente schneidet, und ...“

gebildet werden. Selbst *wenn* die für solche Definitionen benötigten topologischen Relationen zwischen Teilkomponenten von GENED berechnet würden, ließe sich diese Definition nur mit hohem Aufwand (wenn überhaupt) in CLASSIC formulieren.

8 Struktur und Implementation von GENED

„ (The LISP-ability) ... to delay decisions - or more accuralely, to make temporary, nonbinding decisions - is usally a good thing, because it means that irrelevant details can be ignored “
P. Norvig (s. [20, S. 25])

8.1 Hauptkomponenten und Informationsfluß

GENED besteht in grober Näherung aus drei Subsystemen, die miteinander kooperieren und Information austauschen, wozu entsprechende Schnittstellen vorgesehen sind. Diese Hauptkomponenten sind

- der eigentliche Grafikeditor (Interaktion m. dem Benutzer, grafische Oberfläche etc.),
- das Geometriemodul sowie
- das CLASSIC-Wissenrepräsentationssystem mit domänenspezifischer Wissensbasis.

Jede dieser Komponenten benötigt zur Wahrnehmung ihrer Aufgaben eine adäquate interne Repräsentation der vom Benutzer auf dem Editorschirm erzeugten grafischen Konstellation. Während für die Grafikeditorkomponente die grafischen Eigenschaften und Attribute der Objekte bezgl. der Interaktion mit dem Benutzer im Vordergrund stehen und entsprechend verwaltet werden müssen, so benötigt das Geometriemodul eine Repräsentation der in Kap. 3 diskutierten Art, um die topologischen Relationen berechnen zu können (polygonisierte Objekte etc.). CLASSIC hingegen benötigt Konzept- und Rollendefinitionen. Dann werden zu den Grafikobjekten der grafischen Konstellation korrespondierende Konzeptinstanzen (CLASSIC-Individuen) erzeugt, wodurch eine für CLASSIC geeignete Repräsentation der Konstellation gegeben ist. CLASSIC kann somit Inferenzen durchführen, deren Konsequenzen dem Benutzer wiederum im GENED-Entwurfsfenster angezeigt werden. Wesentlich ist, daß alle drei Subsysteme eine einheitliche, konsistente Sicht auf die momentane grafische Konstellation haben. Speziell im *inkrementellen Modus* (s. Kap. 6) müssen Änderungen des Benutzers sofort an alle Hauptkomponenten kommuniziert werden, die dementsprechend ihre internen Repräsentationen aktualisieren.

8.2 Klassen

Die *Klassenstruktur* von GENED ist im wesentlichen eine *Mixin-Struktur*, da so am wenigsten *Klassenabhängigkeiten* entstehen und die *Kombinierbarkeit* hoch ist. Eine Neustrukturierung von Mixin-Klassen führt nicht zum Einsturz einer komplizierten Klassenpyramide, da die einzelnen Klassen aus vielen Einzelbausteinen komponiert sind und sich somit besser warten und anpassen lassen. So werden die einzelnen Klassen für Grafikobjekte von GENED durch eine Folge von *Mixins* definiert, wobei jedes Mixin bestimmte Eigenschaften und Funktionalität für Instanzen mit sich bringt. Abb. 45 zeigt in noch halbwegs übersichtlicher Weise den GENED-Klassengraphen. Klassen, von denen direkte Instanzen erzeugt werden, sind als Kreise visualisiert; die Mixin-Klassen durch Rechtecke. Die Namen sind dabei verkürzt dargestellt.

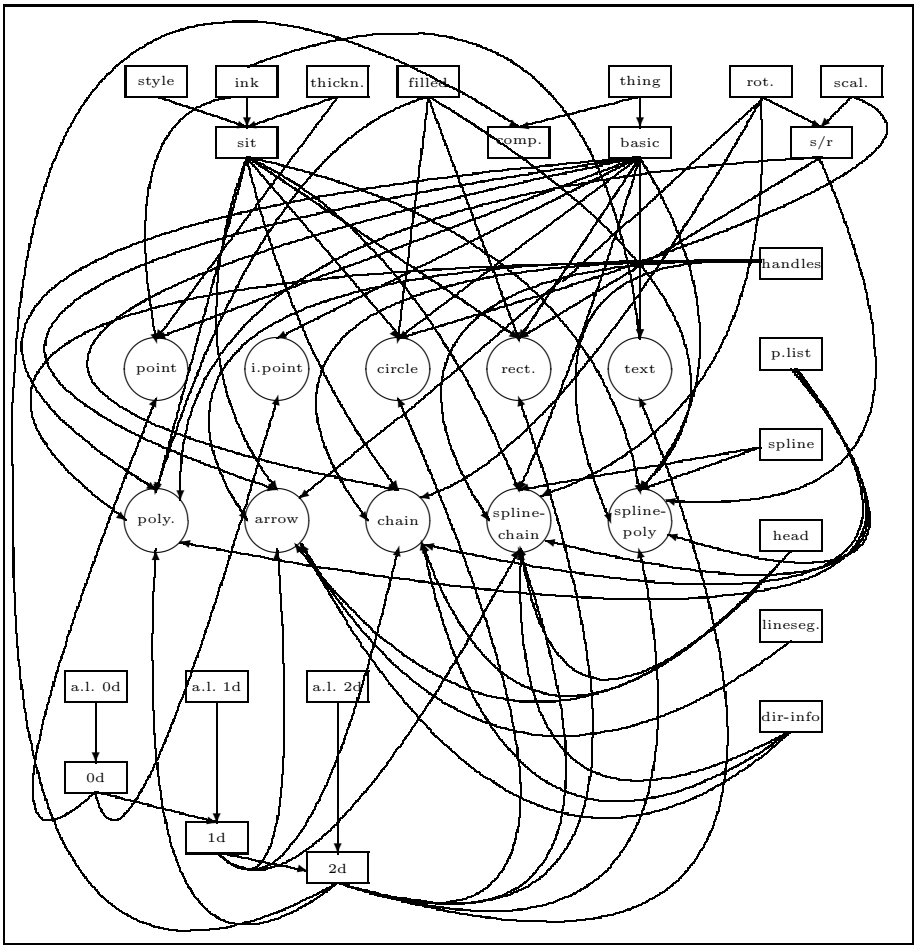


Abbildung 45: GENED-Klassen für Objekte

Die vielen Mixins machen es möglich, z.B. die auf den Objekten ausführbaren Kommandos bequem zu spezifizieren: so läßt sich ein Objekt nur dann rotieren, wenn es auch Instanz der Klasse `rotateable-thing` ist. Einen Kreis zu rotieren macht keinen Sinn, wohl aber Rechtecke.

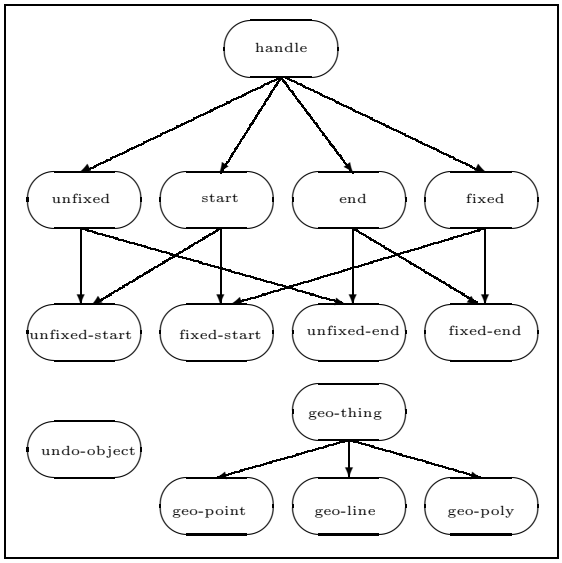


Abbildung 46: Weitere Klassen

In GENED gibt es noch diverse andere Klassen: die Klasse `undo-object` sowie die Klassen des Geometriemoduls `geom-thing, geom-point, geom-line, geom-polygon`, wobei Instanzen von `geom-polygon` ein Flag `closed` besitzen: ist es `t`, so handelt es sich um ein geschlossenes Polygon, ansonsten um einen Streckenpfad (Line Segment Chain). Normalerweise würde man an dieser Stelle zwei Klassen anbringen (`geom-polygon, geom-chain`). Aus anderen softwaretechnischen Gründen wurde jedoch erstere Lösung gewählt (s. auch Kap. 3).

Auch die GENED-Klassen für Grafikobjekte sind so entworfen, daß sie teilweise verschiedene CLASSIC-Konzepte (die sogar unvereinbar sind) abdecken: so z.B. die Klasse `g-arrow`. Instanzen dieser Klasses besitzen ein Flag `head` (durch `head`-Mixin): es bestimmt, ob die Instanz entweder als Pfeil (Arrow) oder Strecke (Line Segment) erscheint. In CLASSIC gibt es für beides Konzepte, und sie schließen sich gegenseitig aus (kein Pfeil kann Strecke sein, und andersrum) - die GENED-Klassenstruktur legt jedoch beides zusammen, ohne eine eigene Klasse für `line-segment` einzuführen. Abb. 46 zeigt weitere GENED-Klassen, u.a. auch die Klassen für die in Kap. 6 diskutierten Punktmanipulatoren.

8.3 Die Basisrollenhierarchie von GENED

Eine grafische Darstellung der Basisrollenhierarchie findet sich in Kap. 4. Der Benutzer kann die Wissensbasis um weitere Konzept- und Rollendefinitionen ergänzen. Insbesondere können mit Hilfe des bereits diskutierten Makros `defqualifiedsubrole` pseudo-qualifizierte Unterrollen aufgesetzt werden, so z.B. auf die Rolle *linked_over_with*:

```
(defqualifiedsubrole linked-over-with g-circle)
```

Wie in Kap. 4 erläutert, werden in diese Rolle nur Füller eingetragen, die Instanzen des Konzeptes g-circle sind.

8.4 Die Basiskonzepttaxonomie von GENED

Die *Basiskonzepttaxonomie* von GENED wurde schon in Kap. 4 diskutiert, wo sich ebenfalls eine grafische Darstellung findet.

8.5 Module

GENED besteht aus 28 LISP-Dateien, die ca. 300 kB Sourcecode ausmachen:

classes.lisp enthält den größten Teil der Klassendefinitionen, insbesondere alle Grafikobjekte- und Punktmanipulatorklassen.

cluster.lisp ist für die Implementierung der Kompositionsobjekte verantwortlich, insbesondere für die interaktive Erzeugung.

comtable.lisp enthält die CLIM-Kommandotabellen.

concepts.lisp ist für die Bibliothek sowie Konzeptinteraktion mit den Grafikobjekten und Bibliothek verantwortlich (z.B. Assign Concept To Object).

copy.lisp implementiert die tiefen Kopierfunktionen.

creator.lisp implementiert die interaktiven Erzeugungsroutinen für primitive Objekte.

delete.lisp regelt die korrekte Löschung von Objekten (so müssen z.B. alle an einem Objekt befestigten Punktmanipulatoren befreit werden, wenn dieses Objekt gelöscht wird - den Rest regelt natürlich der „Garbage Collector“ des LISP-Systems).

delta.lisp berechnet die Differenzen zwischen Szenen für den inkrementellen Modus und steuert den Informationsfluß zwischen GENED und CLASSIC in diesem Fall.

draw.lisp enthält alle Zeichenmethoden für die Grafikobjekte.

frame.lisp ist der CLIM-Anwendungsrahmen für GENED .

gened-packages.lisp definiert die nötigen Packages (Namensräume).

gened-sysdcl.lisp ist die Systemdefinition (welche Dateien daß GENED-System ausmachen, in welcher Reihenfolge sie kompiliert werden, etc.).

handles.lisp implementiert Interaktion und Operationen auf Punktmanipulatoren.

helpaux.lisp enthält diverse Hilfsfunktionen.

init.lisp enthält Initialisierungsroutinen.

inout.lisp ermöglicht alle Speicher- und Ladeoperationen (Objekte, Szenen, Bibliotheken) sowie PostScript-Ausdrucke.

inspect.lisp implementiert den Inspektor.

interface-to-classic.lisp enthält die Funktionen zur Kommunikation mit CLASSIC.

knowledge.lisp enthält die eigentliche Wissensbasis (eigenes Package: `knowledge`).

main.lisp läßt sich schlecht in Worte fassen, hält aber irgendwie das Ganze zusammen.

newgeo.lisp implementiert das Geometriemodul (eigenes Package: `geometry`).

polyrep.lisp erzeugt anhand der CLOS-Objekte eine für das Geometriemodul verwendbare Repräsentation.

rel-copy.lisp memoriert für das Deltamodul alle momentanen topologischen Relationen (und evtl. andere Slots, konfigurierbar).

spatial.lisp errechnet alle nicht vom Geometriemodul berechneten Relationen (z.B. *directly_contains*), u.a. auch Relationen für Kompositionsobjekte (sekundäre Relationen). Außerdem sind die Kommandos für topologischen Anfragen hier implementiert.

spline.lisp berechnet Spline-Ketten und -Polygone (eigenes Package: `splines`).

transfor.lisp implementiert Translations-, Rotations- und Skalierungsinteraktion für Grafikobjekte.

undo.lisp stellt die UNDO-Funktionalität bereit.

values.lisp definiert wichtige Konstanten, z.B. Verzeichnisse, Farben, gewisse Abmessungen etc. Hier kann viel vom Benutzer eingestellt werden (z.B. die Verzeichnisse, wo GENED Szenen, Objekte, Ausdrucke etc. per Default ablegt).

Ausführlicher können die Module nicht diskutiert werden; die Dateien mit den höchsten Versionsnummern sind jeweils aktuell.

8.6 CLASSIC-Anbindung

8.6.1 CLOS-Klassen, Konzepte und Schnittstelle

Alle Grafikobjekte sind Instanzen von CLOS-Klassen. Diese Klassen sind von ihrer Struktur und Funktionalität so entworfen, daß sie softwaretechnische Notwendigkeiten erfüllen. Wenn es auch logisch bzw. konzeptionell einleuchtend ist, daß die Klasse `Rechteck` eine Unterklasse der Klasse `Polygon` sein sollte, so ist dies im GENED-Klassengraph nicht der Fall, da andere Lösungen softwaretechnisch günstiger erschienen.

Schließlich kann die CLASSIC-Wissensrepräsentationssprache nichts mit CLOS-Objekten anfangen - jedem Grafikobjekt muß ein korrespondierendes CLASSIC-Individuum zugeordnet werden. Hier wird im wesentlichen eine 1-zu-1-Abbildung vorgenommen, mit der Ausnahme, daß für alle eindimensionalen Elemente zusätzlich für die beiden Endpunkte CLASSIC-Individuen erzeugt werden. Zudem werden alle topologischen Relationen für diese berechnet, wodurch es möglich wird, verschiedene Arten von Verbindern zu klassifizieren (s. Kap. 3).

Folgende Funktionen sind für die Erzeugung korrespondierender CLASSIC-Individuen verantwortlich:

```
(defun create-classic-ind-for (object)
  (unless (associated-classic-ind object)
    (when *info*
      (format t "~%~%Creating Classic Individual for Object ~A." object))
    (let* ((symbol (intern (name-for object)))
           (classic-ind
             (cl-create-ind symbol
              `(and
                ,@(parent-concepts object)))))
      (unless (eq classic-ind 'classic-error)
        (cl-ind-add (cl-named-ind symbol)
          `(fills belongs-to-clos-object
             ,(id-number object)))
        (setf (associated-classic-ind object) symbol)))))

(defmethod create-classic-ind ((object thing))
  (create-classic-ind-for object))

(defmethod create-classic-ind ((object composite-thing))
  (dolist (part (has-parts object))
    (create-classic-ind part))
  (create-classic-ind-for object))
```

Jedes CLOS-Objekt hat u.a. einen Slot `parent-concepts`, wobei es sich um eine Liste von Symbolen momentaner Elternkonzepte handelt. In ihr steht nach

der Erzeugung nur der Typ des Objektes, für den es per Definition auch ein CLASSIC-Konzept gibt. Per `(setf (parent-concepts object) (list (type-of object)))` wird so bei der Erzeugung das initiale Elternkonzept bestimmt. Die primitiven GENED-Konzepte `g-rectangle, g-circle` etc. dürfen nicht verändert oder gelöscht werden, da sonst keine korrespondierenden CLASSIC-Individuen für die Grafikobjekte erzeugt werden können.

Eventuell stellt sich nun das Objekt im Laufe der Klassifikation durch CLASSIC als Instanz weiterer Konzepte heraus, da es von ihnen subsumiert wird - in diesem Fall wird `parent-concepts` entsprechend aktualisiert. Die Slots `ancestor-concepts` und `parent-concepts` ermöglichen zudem die adäquate Konzeptbeschriftung der Grafikobjekte. Im CLOS-Objekt wird der Name des korrespondierenden CLASSIC-Individuums im Slot `associated-classic-ind` gespeichert, so daß das CLASSIC-Individuum per Mausgeste inspiziert werden kann. Hierzu dient die CLASSIC-Funktion `(cl-print-ind <classic-ind>)`. Komplementär dazu wird jedem CLASSIC-Individuum die Identifikationsnummer seines korrespondierenden CLOS-Objektes mitgegeben. Dabei ist `(cl-ind-add <individual> <classic-description>)` der generelle Weg, einem CLASSIC-Individuum `<individual>` Information zuzusicheren, die über `<classic-description>` (also ein Wort der CLASSIC-Grammatik) beschrieben ist: `‘(fills belongs-to-clos-object ,(id-number object))` fügt also den Füller `(id-number object)` (also eine Zahl, ein „Host Individual“ aus CLASSIC-Sicht) der Attributrolle `belongs-to-clos-object` des erzeugten CLASSIC-Individuums hinzu.

Um GENED über alle möglichen Konzepte und auch Bibliothekskonzepte zu informieren, sind zwei Konstanten im `Knowledge`-Package names `+known-concepts+` und `+library-concepts+` (eine Teilmenge von ersterer) vorgesehen. Diese werden von GENED importiert und gelesen. Der Ersteller einer Wissensbasis muß seine konzeptbezeichnenden Symbole in diese Listen eintragen.

8.6.2 CLOS-Slots, Rollen und Schnittstelle

Die CLASSIC-Individuen müssen nun noch zueinander in Beziehung gesetzt werden, und zwar gemäß den zuvor berechneten und in den CLOS-Objekten gespeicherten topologischen Relationen zwischen ihnen. GENED muß also einige Rollen der Wissensbasis kennen, um diese Information zusichern zu können. Diese Kenntnis wird durch Lesen der im `Knowledge`-Package definierten Konstante `+known-roles+` erhalten.

Dabei sind nicht alle von GENED zu füllenden Rollen topologische Relationen: es erscheint sinnvoll, möglichst viel Information zuzusichern. So macht ein Attribut `color` (also eine Rolle mit höchstens einem Füller) es auf CLASSIC-Ebene möglich, z.B. das Konzept Schwarzes-Rechteck zu definieren. Im Falle flächiger Objekte scheint auch ein Attribut `filled` sinnvoll, womit die Information, ob ein Objekt im Grafikeditor gefüllt ist oder nicht, auf dieser Ebene verfügbar wird. Eine Petrinetzmarke muß z.B. ein „kleiner“ ($2 \leq$ Radius ≤ 10), ausgefüllter Kreis sein (s. Kap. 4). Entsprechende Attributrollen ermöglichen die Klassifizierung. Bei Textobjekten muß zudem der Text selbst in CLASSIC repräsentiert sein, um eventuell Schlüsse über die Art des Textes zu ermöglichen (z.B.: enthält der Text mindestens drei Ypsilons?).

Es werden also Informationen aus Slots der CLOS-Objekte als Füller für bestimmte CLASSIC-Rollen genommen. Dabei korrespondierende Rollen mit CLOS-Slots:

```
(defun insert-fillers-for-slotname (object slot-name fillers-to-add)
  (let ((object-classic-ind (get-classic-ind object)))
    (when object-classic-ind
      (let* ((role-symbol (get-role-from-accessor slot-name))
             (present-fillers (cl-fillers object-classic-ind
                 (cl-named-role role-symbol))))

        (let ((filler-names
                (mapcar #'(lambda (object)
                            (cl-name
                               (get-classic-ind object)))
                  fillers-to-add))
              (present-filler-names
                (mapcar #'cl-ind-name present-fillers)))

          (unclose-all-roles-for-object-i.n. object)

          (dolist (filler-name filler-names)
            (let ((filler-ind (cl-named-ind filler-name)))
              (unclose-all-roles-for-classic-ind-i.n. filler-ind)
              (unless (member filler-name present-filler-names)
                (cl-ind-add object-classic-ind
                   `(fills ,role-symbol
                           ,filler-name)))))))))
```

Diese Funktion nimmt ein CLOS-Objekt `object`, einen `slot-name`, dessen korrespondierende CLASSIC-Rolle sie mit der Funktion `get-role-from-accessor` ermittelt, und eine Liste von CLOS-Objekten, zu welchen das `object` selbst in Beziehung gesetzt werden soll. Nun ermittelt die Funktion jeweils die korrespondierenden CLASSIC-Individuen der CLOS-Objekte mit der Funktion `get-classic-ind` und die Namen der bereits in dieser Rolle vorhandenen CLASSIC-Individuen. Zusätzliche Füller lassen sich nur in diese Rolle eintragen, wenn sie offen ist (s. Kap. 4). Eventuell muß sie erst noch geöffnet werden (`unclose-all-roles-for-classic-ind-if-necessary`). Außerdem wird achtgegeben, daß eine Information nur einmal addiert wird, was dann mit `cl-ind-add` geschieht.

Zudem muß es ähnliche Funktionen für die Entfernung von Rollenfüllern geben: der inkrementelle Modus macht dies nötig.

8.6.3 Inkrementeller Modus und Deltamodul

Wie in Kap. 6 diskutiert, protokolliert im inkrementellen Modus eine Funktion des Deltamoduls alle Zustandsveränderungen der CLOS-Objekte, indem sie eine Liste all der Slots eines CLOS-Objektes zurückgibt, die sich seit dem letzten Aufruf der Funktion verändert haben. Alle Slots eines Objektes, die in dieser Liste stehen, werden nun gezielt untersucht. Dazu werden die in dieser Liste stehenden Slots als Mengen/Listenvariblen betrachtet, und es können Mengendifferenzen gebildet werden: alle Füller, die in der Differenzmenge $\Delta\ominus = vorher(slot) - nachher(slot)$ stehen, werden aus der enstp. Rolle des korrespondierenden CLASSIC-Individuums entfernt. Analog dazu werden alle Füller, die in der Differenzmenge $\Delta\oplus = nachher(slot) - vorher(slot)$ stehen, der korrespondierenden Rolle hinzugefügt. Gilt für CLOS-Objekt x bzgl. des Slots `intersects-objects` z.B. $\Delta\ominus = \{a,b,c\} - \{a,c,d\} = \{b\}$ und somit auch $\Delta\oplus = \{a,c,d\} - \{a,b,c\} = \{d\}$ (m.a.W., Objekt x wurde so manipuliert, daß es nun nicht mehr Objekt b schneidet, dafür aber Objekt d), so wird im zu x korrespondierenden CLASSIC-Individuum der Füller b der zum CLOS-Slot `intersects-objects` korrespondierenden Rolle

entfernt, dafür jedoch d hinzugefügt.[11]

Das Verfahren funktioniert insbesondere auch für Attribute. So enthält der CLOS-Slot `x-pos` eines CLOS-Objektes keine Liste, sondern eine Zahl: ändert sich `x-pos` nun von vorher `10` auf hinterher `20`, so gilt (wenn der Slot als Menge betrachtet wird) $\Delta\ominus = \{10\} - \{20\} = \{10\}$ und $\Delta\oplus = \{20\} - \{10\} = \{20\}$. Also wird beim korrespondierenden CLASSIC-Individuum der Füller `10` der zum CLOS-Slot `x-pos` korrespondierenden Attributrolle entfernt, und dann `20` hinzugefügt. Die recht komplexen LISP-Funktionen/Methoden können hier nicht diskutiert werden.

8.6.4 Das Makro `defqualifiedsubrole`

Hier ist der Code des bereits in Kap. 4 ausführlich diskutierten Makros:

```
(defmacro defqualifiedsubrole (role
                               qualification
                               &key
                               (name
                                (intern
                                 (concatenate 'string
                                              (string role)
                                              "-"
                                              (string qualification))))
                               (inv-name
                                (intern
                                 (concatenate 'string
                                              (string name) "-INVERSE")))

                               (parent role)
                               (inverse-parent (or (cl-inv-role-name parent)
                                                   role)))
  `(progn
     (cl-define-primitive-role ',name :inverse ',inv-name
                               :parent ',parent
                               :inverse-parent ',inverse-parent)
     (cl-add-filler-rule ',(intern (concatenate 'string (string name)
                                                "-INV-FILLER-RULE"))
                         (cl-named-concept ',qualification)
                         (cl-named-role ',inv-name)
                         #'(lambda (ind role)
                             (declare (ignore role))
                             (or
                              (cl-fillers ind (cl-named-role ',inverse-parent))
                              (break "NIL as filler computed for ~A"
                                     ',inv-name)))
                         :filter
                         '(and (at-least 1 ,inverse-parent)
                               (test-c cl-test-closed-roles? (,role))))))
```

Die vom Makro aufgesetzte Füllerregel hat den Namen `...-INV-FILLER-RULE`, feuert auf Instanzen des in `qualification` angegebenen Konzeptes und berechnet Füller für die Rolle, deren Name Wert von `inv-name` ist. Eingetragen als Füller dieser Rolle werden alle Füller der Rolle, deren Name Wert von `inverse-parent` ist. Der `:filter` stellt sicher, daß die Regel nur dann feuert, wenn überhaupt solche einzutragenden Füller existieren und zusätzlich die entsp. Oberrolle des Individuums, auf dem die Regel feuert, geschlossen ist - denn eine Regel muß stets die selben Füller erzeugen. Sie kann dies also erst tun, wenn die Information bzgl. dieser Rolle

[11] In den Mengendifferenzen stehen zwar CLOS-Objekte, bzgl. der Rollen der CLASSIC-Individuen müssen natürlich die korrespondierenden CLASSIC-Individuen als Füller entfernt bzw. hinzugefügt werden.

komplett ist: genau dies stellt `(test-c cl-test-closed-roles? (,role))` bzgl. der Rolle `role` sicher.

8.7 Konfigurierung von GENED

Der Benutzer geht nun so vor, daß er obigen Teil der *Wissensbasis* um seine eigenen, *domänenspezifischen Konzeptdefinitionen und pseudo-qualifizierten Unterrollen erweitert.* Zusätzliche *primitive Rollen* lassen sich nicht so einfach hinzufügen, da es in den CLOS-Objekten korrespondierende Slots oder andere Funktionen geben muß, die diese Füller für das fragliche Objekt berechnen (hier wird eine generische Funktion aufgerufen, die anhand des Rollennamens erhalten wird, um den Slot-Inhalt des CLOS-Objektes zu extrahieren - sie ist meist der Slot-Accessor, kann aber natürlich auch eine andere generische Funktion sein, die anhand anderer Slots bzw. des Objektzustandes erst Füller berechnet, die aber gar nicht so im Objekt in einem Slot gespeichert sind. In diesem Fall muß die Funktion erst geschrieben werden). Eine Anpassung führt hier also über mehrere Schritte und eventuelle Programmänderungen von GENED, ist aber prinizipiell möglich, wenn auch meist nicht nötig, da nahezu alle relevante Information aus den CLOS-Slots ohnenhin schon zugesichert wird.

8.7.1 Anpassung der Konstanten

Jedes Konzept muß in der konstanten Liste `+known-concepts+` erscheinen.

Soll das Konzept jedoch auch als *Bibliothekskonzept* benutzbar sein, so muß das entsprechende Symbol auch noch in der Liste `+library-concepts+` erscheinen.

Alle Rollennamen (Symbole) müssen in der Konstanten `+known-roles+` erscheinen. Bei dieser Liste handelt es sich um einen *Baum der Tiefe zwei:*

```
(defconstant +known-roles+
  '((belongs-to-clos-object

     spatial-relation

     in-relation-with-objects
     disjoint-with

     touching-objects

     intersects-objects
     intersects-0-objects
     intersects-1-objects
     intersects-2-objects

     contains-objects
     contained-in-objects

     ..... )

    (has-parts-g-rectangle-rectangle
     has-parts-g-rectangle-inverse

     has-parts-directed-element
     has-parts-directed-element-inverse

     touching-objects-directed-element
     touching-objects-directed-element-inverse
```

```
intersects-objects-kapazitaets-label
intersects-objects-kapazitaets-label-inverse

has-parts-stelle
has-parts-stelle-inverse

..... )))
```

Der Benutzer muß den Namen (das Symbol) der neuen Rolle in eine der Unterlisten eintragen; dabei ist die Unterlistengliederung für die richtige Reihenfolge des Schließens und Öffnens der Rollen notwendig. Hierbei kann es zu Inkonsistenzen kommen, wenn die Reihenfolge, in der die Rollen geschlossen werden, nicht die Subsumptionsrelation der Rollen respektiert - eine pseudo-qualifizierte Unterrolle darf nicht vor ihrer Oberrolle geschlossen werden, da anderseits die durch `defqualifiedsubrole` erzeugte Füllerregel zu einem Zeitpunkt feuern würde, zu dem die Unterrolle schon geschlossen wäre - somit könnten keine Eintragungen in die Unterrolle gemacht werden. Zudem muß bemerkt werden, daß `defqualifiedsubrole` immer eine Inverse der zu definierenden pseudo-qualifizierten Unterrolle benötigt und bei Nichtexistenz auch anlegt (s. Kap 4). Diese hat den von `defqualifiedsubrole` generierten Namen `...-inverse` - der Name kann jedoch auch über den Schlüsselwort-Parameter `:inv-name` spezifiziert werden. Die hier diskutierten Inversen müssen ebenfalls in der Liste `+known-roles+` erscheinen.

Die Rollen werden also so geschlossen, daß erst alle Rollen der ersten Unterliste für alle Objekte geschlossen werden, dann alle Rollen der zweiten Unterliste für alle Objekte, etc. Beim Öffnen wird genau umgekehrt verfahren. In der Petrinetzdomäne reichen zwei Unterlisten aus. In einer Wissensbasis, die eine pseudo-qualifizierte Unterrolle benutzt, deren qualifizierendes Konzept in seiner Definition selbst wieder eine pseudo-qualifizierte Unterrolle benutzt, muß `+known-roles+` eine weitere Unterliste haben, etc. Bei Attributen oder nichthierarchischen Rollen ist die Listenposition hingegen irrelevant.

Alle Symbole, die nun neu im `Knowledge`-Package (also in der Wissensbasis) erscheinen, müssen in der *Package-Definition* (Datei `gened-packages.lisp`) als `Export`-Symbole des `Knowledge`-Package aufgeführt werden.

Mehr ist zur Konfigurierung von GENED nicht zu tun. Die eigentliche Arbeit liegt in der Erstellung einer domänenspezifischen Wissensbasis.

8.8 Erweiterungsmöglichkeiten

GENED ist noch längst nicht ausgereift und erst recht nicht fehlerfrei - schließlich ist das Programm erst 5 Monate alt. Ich würde es höchstens als Prototypen bezeichnen. In erster Linie sollten also weitere Fehler erkannt und beseitigt werden.

Eine sinnvolle Erweiterung wären Manipulatoren, mit deren Hilfe sich der *Mittelpunkt* eines Grafikobjektes an einem anderen *befestigen* läßt. Der gleiche Effekt (nämlich die Mitverschiebung eines Grafikobjektes bei Positionsveränderung eines anderen) kann auch durch Bildung eines (temporären) Kompositionsobjektes erreicht werden. Dieses Vorgehen kann jedoch nicht als benutzerfreundlich bezeichnet werden.

Nützlich wären auch weitere Grafikprimitive: *Halbkreise* (oder sogar bel. *Kreissegmente*), *Dreiecke* und *Rauten* sowie *Rechtecke mit abgerundeten Ecken*. Bei einigen dieser neuen Primitiven würde die Erzeugung der Objektrepräsentation für das Geometriemodul recht aufwendig werden - schließlich müssen flächige Objekt poly-

gonisiert werden.

Natürlich lassen sich *Rauten* und *Dreiecke* auch mit den jetzigen Primitiven erzeugen - jedoch existieren weder spezialisierte Erzeugungsroutinen (der Benutzer müsste ein *Polygon* für *Dreiecke* erzeugen, und vielleicht ein *Rechteck* und dieses um 45 Grad rotieren für eine *Raute*), noch korrespondierende Basiskonzepte in der Wissensbasis. Der Benutzer könnte zwar das Konzept Dreieck hinzuschreiben, GENED würde jedoch nicht von selbst das Basiskonzept Dreieck für ein Polygon mit drei Ecken zusichern - diese Objekt hätte das initiale Elternkonzept Polygon. Schließlich sollten die primitiven Grafikobjekte zu primitiven Konzepten korrespondieren.

Anderseit könnte eine Raute von CLASSIC erkannt werden mit folgender nichtprimitiver Konzeptdefinition:

```
(cl-define-concept 'diamond
                   '(and g-rectangle
                         (all total-rotate-angle
                              (one-of 45 135 225 315))))
```

oder ein Dreieck mit

```
(cl-define-concept 'triangle
                   '(and g-polygon
                         (fills number-of-edges 3)))
```

Die benutzten Attributrollen `total-rotate-angle, number-of-edges` existieren momentan noch nicht. Ihre Schaffung und Zusicherung von GENED aus wäre jedoch recht einfach.

Kreissegmente könnte der Benutzer aus *Spline-Kurve* und zwei *Strecken* anfertigen und daraus ein Kompositionsobjekt machen. CLASSIC kann aber unmöglich anhand der Eigenschaften der Komponenten auf ein *Kreissegment* schließen, weswegen in solchen Fällen neue Erzeugungsroutinen und primitive Basiskonzepte programmiert werden müssen. GENED muß ein solches Objekt also kennen und anbieten, um es zusichern zu können, da CLASSIC hier keine Klassifizierung vornehmen kann.

Eine immer bestehende Möglichkeit ist, ein primitives Basiskonzept zu schreiben und dem Benutzer die Zusicherung des initialen Elternkonzeptes von Hand aufzuerlegen (mit dem Menüpunkt Assign Concept To Object). Vorher müßte er aber alle anderen Konzepte des Objektes evtl. entfernt haben (mit Remove All Concepts From Object). So würde er ein *Dreieck* mit der *Polygonerzeugungsroutine* erzeugen, daß Konzept g-polygon vom Objekt entfernen und manuell g-triangle zusichern - sicherlich ein sehr unbequemes und fehlerträchtiges Vorgehen. Das Konzept g-triangle könnte in diesem Fall jedoch als primitiv definiert werden.

Ein anderes Problem besteht in den bereits angesprochenen topologischen Relationen für Kompositionsobjekte und deren Komponenten - die Problematik wurde in Kap. 7 diskutiert und soll hier nicht weiter vertieft werden. Weiterhin wird bisher nicht die Dimension des Schnittes von Kompositionsobjekten berechnet, was aber einfach zu implementieren ist: hier ist daß Maximum aller Schnitte aller Teilkomponenten zu nehmen (s. Kap. 3).

Literatur

[1] V. Haarslev: „Formal Semantics of Visual Languages Using Spatial Reasoning“, in V. Haarslev (Ed.): „Proceedings of the 11th IEEE Symposium on Visual Languages, 1995, Sept. 5-9, Darmstadt, Germany, IEEE Press 1995“, S. 156 - 163

[2] D. Wang, J.R. Lee: „Pictorial Concepts and a Concept-Supporting Graphical System“, in „Journal of Visual Languages and Computing“, Vol. 4, No. 2, 1993, S. 177 - 199

[3] K.M. Kahn: Technical Report SSl-90-38 [P90-00099], Xerox Palo Alto Research Center, 1990

[4] G. Görz (Hrsg.): „Einführung in die künstliche Intelligenz“, Addison-Wesley, Bonn, 1993

[5] M.J. Egenhofer: „Reasoning about Binary Topological Relations“, in O.Günther, H.-J. Schek (Eds.): „Advances in Spatial Databases, Second Symposium, SSD ’91, Zürich, Aug. 28-30“, in „Lecture Notes in Computer Sciene“, Vol. 525, Springer-Verlag, Berlin, 8/1991, S. 143 - 160.

[6] E. Clementini, P.D. Felice, P. v. Oosterom: „A Small Set of Formal Topological Relationships Suitable for End-User Interaction“, in D. Abel, B.C. Ooi (Eds.): „Advances in Spatial Databases, Third International Symposium, SSD ’93, Singapore, June 23-25, 1993“, in „Lecture Notes in Computer Science“, Vol. 692, Springer-Verlag, Berlin, 6/1993, S. 277 - 295

[7] W.A. Woods, J.G. Schmolze: „The KL-ONE Family“, in F. Lehmann (Ed.): „Semantic Networks in Artificial Intelligence“, Pergamon Press, 1992, S. 133 - 177

[8] R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, L.A. Reswnick, and A.Borgida: „Living with CLASSIC - When and How to Use a KL-ONE-like Language“, in J.F. Sowa: „Principles of Semantic Networks: Explorations in the Representation of Knowledge“, Morgan Kaufmann Publishers, San Mateo, California, 1991, S. 401 - 456

[9] L.A. Resnick, A. Borgida, R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, K.C. Zalondek: „CLASSIC Description and Reference Manual For the COMMON LISP Implementation Version 2.2“, 1993

[10] E. Jessen, R. Valk: „Rechensysteme - Grundlagen der Modellbildung“, Springer-Verlag, Berlin, 1987

[11] W. Reisig: „Petrinetze - Eine Einführung“, 2. Auflage, Springer-Verlag, Berlin, 1991

[12] R. Sedgewick: „Algorithmen in C“, Addison-Wesley, Bonn, 1993

[13] W.D. Fellner: „Computergrafik“, 2. Auflage, BI-Wissenschaftsverlag Mannheim, 1992

[14] P.H. Winston, B.K.P. Horn: „LISP“, 3. Ed., Addison-Wesley, Reading, Massachusetts, 1989

[15] S.E. Keene: „Object-Oriented Programming in COMMON LISP: A Programmer’s Guide to CLOS“, Addison-Wesley, Reading, Massachusetts, 1989

[16] G.L. Steele: „COMMON LISP - The Language“, 2. Ed., Digital Press, 1990

[17] Franz Inc.: „CLIM-Manual“, Rev. 3

[18] P. Schefe: „Künstliche Intelligenz - Überblick und Grundlagen“, 2. Auflage, BI-Wissenschaftsverlag, Mannheim, 1991

[19] E. Rich, K. Knight: „Artificial Intelligence“, 2. Ed., McGraw-Hill, NewYork, 1991

[20] P. Norvig: „Paridigms of Artificial Intelligence Programming - Case Studies in COMMON LISP“, Morgan Kaufmann Publishers, San Mateo, California, 1992

[21] S. Russell, P. Norvig: „Artificial Intelligence - A Modern Approach“, Prentice Hall, Englewood Cliffs, New Jersey, 1995

A Gesten für die GENED-Benutzung

GENED-Gesten		
Kontextmenü für Objekt	-	Rechter Mausknopf
Verschiebe Objekt	-	Linker Mausknopf
Erzeuge Objekt	Shift	Linker Mausknopf
Lösche Objekt	Shift	Mittlerer Mausknopf
Befestige Punktmanipulator	-	Mittlerer Mausknopf
Befreie Punktmanipulator	-	Mittlerer Mausknopf
Kopiere Objekt	Control	Mittlerer Mausknopf
Skalierere Objekt	Shift	Rechter Mausknopf
Rotiere Objekt	Control	Linker Mausknopf
Dekomponiere Kompositionsobjekt	Control	Linker Mausknopf
Inspiziere Objekt	Meta	Linker Mausknopf
Inspiziere CLASSIC-Individuum	Meta	Rechter Mausknopf

B Einige erfolgreich klassifizierte Petrinetze