DFKI

Document

D-93-22

Implementierung graphischer Benutzungsoberflächen mit Tcl/Tk und Common Lisp

Andreas Abecker

Dezember 1993

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Deutsches Forschungszentrum
für
Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application ar ea. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

Implementierung graphischer Benutzungsoberflächen mit Tcl/Tk und Common Lisp

Andreas Abecker

DFKI-D-93-22

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITWM-413 5839 ITW 9304/3).

Implementierung graphischer Benutzungsoberflächen mit Tcl/Tk und Common Lisp

Andreas Abecker
DFKI, Postfach 2080
W-67608 Kaiserslautern, Germany
aabecker@dfki.uni-kl.de

Zusammenfassung

Es wird das Programmiersystem Tcl/Tk als sinnvolles Hilfsmittel für die Implementierung graphischer Benutzeroberflächen für LISP-Anwendungen vorgestellt. Dazu wird auf die Kopplung zwischen LISP und Tcl/Tk eingegangen und einiges an nützlichen LISP-Funktionen für die Benutzung dieser Kopplung zur Verfügung gestellt. Als Anwendungsbeispiel wird ein Taxonomiebrowser für die im terminologischen Wissensrepräsentationssystem TAXON erstellten Begriffshierarchien implementiert.

Be simple. Be elegant.

(Entwurfsprinzipien für die Gestaltung graphischer Benutzungsoberflächen)

Inhaltsverzeichnis

1 Vorgeschichte und Aufgabenstellung

Graphisch orientierte Benutzungsoberflächen gewinnen zunehmend an Bedeutung, auch und gerade in den Bereichen Künstliche Intelligenz und Expertensysteme. Eine bedienerfreundliche und attraktive Benutzerschnittstelle steht heutzutage nicht nur fast automatisch im Pflichtenheft jedes Softwareproduktes, das Praxisrelevanz erlangen will – sie besitzt auch eigene softwareergonomische Qualität über den reinen „Hochglanzeffekt“ hinaus. Sinnvolle Visualisierung und Benutzerführung kann Inhalte besser veranschaulichen, Abläufe transparenter gestalten und die Benutzerinteraktion übersichtlicher und effektiver machen. Sie steigert dadurch Akzeptanz und Effizienz der Anwendung.

Deshalb besteht auch im Projekt VEGA [4] Interesse daran, den in Common Lisp [20] entwickelten (bzw. noch zu entwickelnden) Tools zur Validierung und Exploration von Wissensbasen ein ansprechendes optisches Äußeres zu geben. Gerade bei der gegebenen (Fern-)Zielsetzung – große, eventuell hybride Wissensbasen *interaktiv* mit dem Benutzer zu analysieren, vervollständigen und debuggen – gibt es besonderen Bedarf an sinnvoller Gestaltung der Arbeitsvorgänge bei dieser Benutzerinteraktion.

Aus diesem Kontext heraus ergibt sich folgendes Anforderungsprofil an ein Werkzeug für die Gestaltung graphischer Oberflächen:

1. Typische Lisp-Programmierer haben normalerweise wenig Erfahrung in der Graphikprogrammierung. Deshalb ist ein möglichst aus Lisp heraus zu benutzendes, einfach bedienbares System gesucht, das eine gewisse notwendige Standardfunktionalität auf hohem Abstraktionsniveau bietet[1] – z.B. Erstellung von Pulldown-Menüs, ohne diese selber aus Graphikprimitiven zusammenstellen zu müssen. Dabei muß das System aber auch hinreichend primitive Graphikoperationen sowie Kontrollstrukturen zur Verfügung stellen, um bei Bedarf flexibel neue Funktionalität „anbauen“ zu können – z.B. Darstellung von Graphen mit an den Knoten angehefteten auswählbaren Kommandos.
2. Da komplexe KI-Anwendungen oft in hohem Maße Rechnerressourcen verbrauchen, sollte der Lisp-Prozeß möglichst wenig durch die Graphikkomponente belastet werden.
3. Um einerseits die freie Verteilung von Forschungsprototypen zu ermöglichen, andererseits aber auch Kooperation mit Industriepartnern unter „Real-World“-Restriktionen zu gestatten, sollte die Lösung unter beliebigen Common Lisp Implementierungen lauffähig sein. Sie sollte als Public Domain Software frei verfügbar und hinsichtlich der verwendeten Hardware- und Betriebssystemvarianten hochgradig portabel sein.

Mehrere Anfragen in für diese Problematik relevanten internationalen Newsgruppen ergaben folgendes Bild:

- Es existieren einige sehr hochentwickelte Toolboxen für Graphikprogrammierung unter Lisp, wie z.B. CLIM (Common Lisp Interface Manager) [19], Garnet [10] oder GINA. Diese Systeme erfüllen die unter Punkt 1 genannten Forderungen an die Funktionalität fast optimal, werden aber durch die folgenden Eigenschaften, die auf diese Systeme großenteils zutreffen, ausgeschlossen:
  - Sie belegen beträchtliche Teile des Lisp-Arbeitsspeichers, weil sie als sehr große Lisp-Module implementiert sind, häufig sogar noch unter Rückgriff auf ihrerseits sehr aufwendige objektorientierte Lisp-Aufsätze (z.B. CLOS – Common Lisp Object System).

[1] Eine ähnliche Argumentation findet sich auch in [6].

In diesem Fall (betrachte z.B. CLIM) ist auch die Einarbeitung nicht ganz einfach, weil man sich auch CLOS aneignen muß.

- Sie werden teilweise kommerziell vertrieben.
- Sie laufen nur mit wenigen Lisp-Implementierungen zusammen.

- Ferner gibt es Lisp-Aufsätze wie CLX (Common Lisp X Interface [7]) oder CLUE, die vom Abstraktionsniveau her wesentlich niedriger angesiedelt sind (z.T. auch den obengenannten Systemen zugrundeliegen), d.h. sich nah an der X11-Programmierung bewegen. Dies widerspricht unserem obengenannten Kriterium 1, weil ein erheblicher Einarbeitungs- und Programmieraufwand erforderlich ist.

Als Ausweg bot sich das Programmiersystem Tcl/Tk [18; 22] von John Ousterhout (University of California, Berkeley) an.

Tcl (Tool Command Language) [16; 13; 11; 15] ist eine einfache, interpretierbare Kommandosprache (mit voller Turing-Mächtigkeit) mit der Möglichkeit, Scripts zu erstellen. Sie ist als Bibliothek von C-Prozeduren implementiert und daher gut einbettbar, erweiterbar und mit C-Prozeduren zu koppeln. Sie soll dazu dienen, Teile von Anwendungsprogrammen oder ganze Anwendungen zusammenzusetzen und ihre Kommunikation zu steuern, ferner die Implementierung der Schnittstellen zum Benutzer und zwischen den Systemteilen vereinfachen.

Tk (X11 Toolkit based on Tcl) [13; 17; 12; 14] ist ein auf Tcl aufsetzendes X11 Toolkit. Es stellt Kommandos zum Erzeugen, Anordnen und für die Kommunikation mit graphischen Objekten zur Verfügung. Die Verwendung solcher Tk-Befehle in Tcl-Scripts zusammen mit den Tcl Daten- und Kontrollstrukturen gestattet die einfache und dennoch mächtige Programmierung graphischer Benutzungsoberflächen. Die Verwendung von Tcl-Scripts als Kommandos erlaubt das Einziehen von Abstraktionsebenen und fördert damit sowohl die Oberflächenprogrammierung auf hohem Abstraktionsniveau wie auch die Wiederverwendung von Code.

Hinsichtlich der eingangs aufgeführten Anforderungen bietet Tcl/Tk als Programmiersystem für graphische Oberflächen eine Reihe von Vorteilen:

1. Die Grundlagen der Sprache sind einfach zu erlernen. Dennoch ist die Programmierung hinreichend mächtig.
2. Der Tcl-Interpreter läuft unabhängig vom Lisp-Prozeß, so daß die Graphikkomponente von Lisp aus angesprochen werden kann, ohne aber dort Speicherplatz zu beanspruchen. Die Implementierung auf C-Basis gewährleistet eine hohe Geschwindigkeit der Graphikkomponente.
3. Tcl/Tk ist via ftp als Public Domain Software verfügbar.

Diese Vorteile legten eine Verwendung von Tcl/Tk sehr nahe, verursachten allerdings auch eine Verschiebung der ursprünglichen Aufgabenstellung, da nun nicht mehr ein Lisp-Aufsatz zur Diskussion stand, sondern eine externe Graphikkomponente an das Lisp-System zu koppeln war.

Ein erster Schwerpunkt der Arbeit wird also die Frage sein, ob und wie sich Tcl/Tk an (möglichst beliebige Implementierungen von) Common Lisp anbinden läßt, wie die Kommunikation zu gestalten ist und was Lisp-seitig an Graphik-Befehlen zur Verfügung gestellt werden sollte. Mit diesen Fragen werden wir uns in Kapitel 2 beschäftigen.

Danach ist die Nützlichkeit dieser Koppelung anhand eines nichttrivialen Beispiels zu zeigen. Dazu werde ich in Kapitel 3 die Implementierung eines Browsers für mit TAXON [3] erstellte Begriffshierarchien beschreiben.

Die Anhänge erleichtern hoffentlich dem interessierten Leser die Einarbeitung in die Implementierung.

2 Die Kopplung von Lisp und Tcl/Tk

2.1 Zur Kommunikation zwischen Lisp und Tcl/Tk

Grundsätzlich stellt sich die Frage, wie die beiden Interpreterschleifen für Lisp (die Lisp-Toplevelschleife *read-eval-print* auf dem Lisp-Standard-I/O) und Tcl/Tk (der Wish-Interpreter) am sinnvollsten zu verbinden sind, so daß Lisp und Wish gleichzeitig laufen und sich gegenseitig Kommandos und/oder deren Ergebnisse zuschicken können:

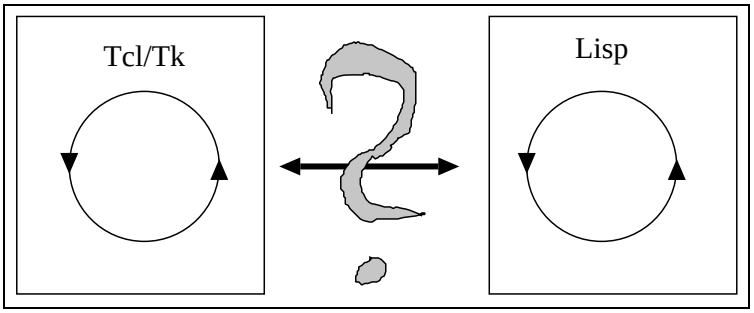

Abbildung 1: Die Kopplungsproblematik

Es gibt verschiedene Mechanismen, die möglicherweise zu einer Lösung dieser Frage beitragen können, z.B. :

1. Sowohl Lisp als auch Tcl/Tk können von/auf Files lesen/schreiben.
2. Auf Betriebssystemebene ließen sich die Standard-I/Os der beiden Prozesse koppeln.
3. Die meisten Lisp-Implementationen erlauben das Starten externer Prozesse und die Kommunikation mit diesen. Einen ähnlichen Mechanismus gibt es auch in Tcl.
4. Das Tcl-Kommando `send` erlaubt es, an den Wish-Interpreter Kommandos zur Ausführung zu schicken.

Die einfachste Lösung 1, d.h. die Kommunikation über ein File, ist nicht sehr elegant, und belastet den Speicher mit einem ständig wachsenden Kommunikationsfile. Sie ist dafür sehr einfach zu implementieren und vollkommen portabel für verschiedene Lisp-Implementationen, `Wish`-Versionen und Betriebssysteme.

Die Kopplung der Prozesse auf Betriebssystemebene 2 erfordert etwas Programmieraufwand, sollte aber die Vorteile von Lösung 1 erhalten, ohne deren Nachteile zu besitzen.

Variante 3 läßt sich z.B. in Lucid Common Lisp sehr einfach implementieren, benutzt dazu aber ein nicht im Kern von Common Lisp enthaltenes Kommando.

Man kann sich also durchaus mehrere verschiedene Ansätze zur Realisierung der Kopplung vorstellen. Dabei sollte als zusätzliches Qualitätskriterium im Hinblick auf die eventuelle spätere Verwendung in einer praktischen Anwendung mit sehr begrenzten Rechnerressourcen der sparsame Umgang mit Speicherplatz und Prozessen im Auge behalten werden. Außerdem müssen verschiedene Kopplungsmechanismen nicht unbedingt immer zum exakt gleichen Ergebnisverhalten führen.

Dabei läßt sich die Aufgabe zuerst einmal dadurch vereinfachen, daß für unsere intendierte Verwendung eine vollkommen synchrone Kommunikation von Lisp und Wish genügen sollte. Diese Voraussetzung ist nicht selbstverständlich; ihre Umsetzung und Umsetzbarkeit bestimmt wesentlich die Kopplungsstrategie (und umgekehrt). In Abschnitt 2.1.2 werde ich noch ein Kommunikationskonzept ansprechen, das eine asynchrone Kopplung implementiert.

Als Begründung für das Ausreichen einer synchronen Kopplung betrachten wir zuerst einmal das intendierte Endprodukt unserer Mühen:

Der Benutzer sieht eine graphische Oberfläche, Lisp ist währenddessen passiv. Aufgrund von Aktionen des Benutzers auf der Oberfläche (z.B. Drücken eines Buttons) kann eine Berechnung in Lisp notwendig werden. Tcl/Tk stößt diese an und wartet seinerseits passiv auf das Ergebnis. Dieses bewirkt evt. eine Veränderung in der Oberflächendarstellung, und der Zyklus beginnt von neuem.

Es ist also zu jedem Zeitpunkt nur ein Prozeß aktiv, der andere wartet auf das Ergebnis.

Wir benötigen daher prinzipiell in jedem System eine endlose Interpreterschleife, deren Ausgabe mit der Eingabe der jeweils anderen kommunizieren kann (s. Abb. 2).

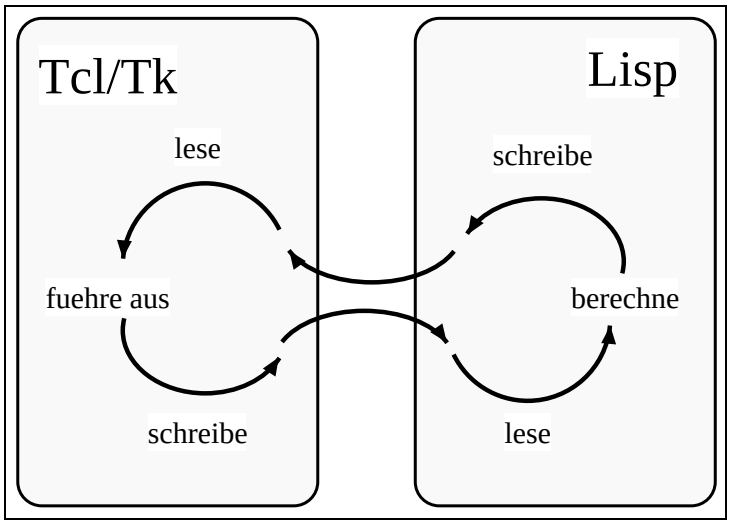

Abbildung 2: Einfachste synchrone Kommunikation

Dabei hat man nach außen hin im Grunde zwei gleichwertig zusammenarbeitende Systeme, wenn auch eventuell auf der Prozeßebene kopplungsabhängig ein Interpreter den anderen starten muß und damit im engeren Sinne die Kontrolle besitzt. Für den Benutzer ist das nicht relevant, ihm bleibt unbenommen, in welchem System er die wesentliche Programmierarbeit leistet.

In unserem Kontext ist klar, daß dies im Lisp-System geschieht, in dem die bekannten Sprach- und Debuggingmöglichkeiten genutzt, auf vorhandene Datenstrukturen zugegriffen und vorhandene Software-Module wiederverwendet werden können (z.B. DAG-Layout-Algorithmen, vgl. Kap. 3).[2]

Es ist also in Lisp ein Graphik-Paket zu entwerfen, das eine sinnvolle Einbindung der Tcl/Tk-Möglichkeiten in Lisp-Kontrollstrukturen erlaubt. Das Paket sollte generisch sein in dem Sinne,

[2]Es bleibt weiterhin die Möglichkeit, Lisp als Prototyping-Sprache zu nutzen und im weiteren Verlauf des Software-Lebenszyklus Teile der Anwendung zu isolieren und aus Effizienzgründen nach Tcl auszulagern.

als es nur einen kleinen, leicht austauschbaren, kopplungsspezifischen Kern enthält und im wesentlichen von der Art der konkreten Anbindung abstrahiert.

Abb. 3 zeigt diesen Realisierungsansatz: eine Tcl-Prozedur `tellisp` und eine Lisp-Funktion `tellwish` senden dem jeweils anderen Prozeß Nachrichten. Falls diese nicht direkt in den Evaluationszyklus hineinkommen, sondern nur auf einen Kommunikationspuffer geschrieben werden, benötigen die beiden Prozesse zusätzliche Mechanismen `readlisp, readwish` zum Lesen dieses Puffers. Der weitere systeminterne Überbau bleibt von der konkreten Implementierung der Kopplung unabhängig. Allenfalls die Art der Interpreterschleifen kann noch zum kopplungsspezifischen Teil gerechnet werden; beispielsweise muß bei bestimmten Kopplungsarten explizit „von Hand“ eine Read-Eval-Print-Schleife simuliert werden, die man geschenkt bekommt, wenn man Standard I/O der beiden Prozesse auf Betriebssystemebene verkoppelt und sich damit direkt in die Toplevel-Interpreterschleifen einhängt.

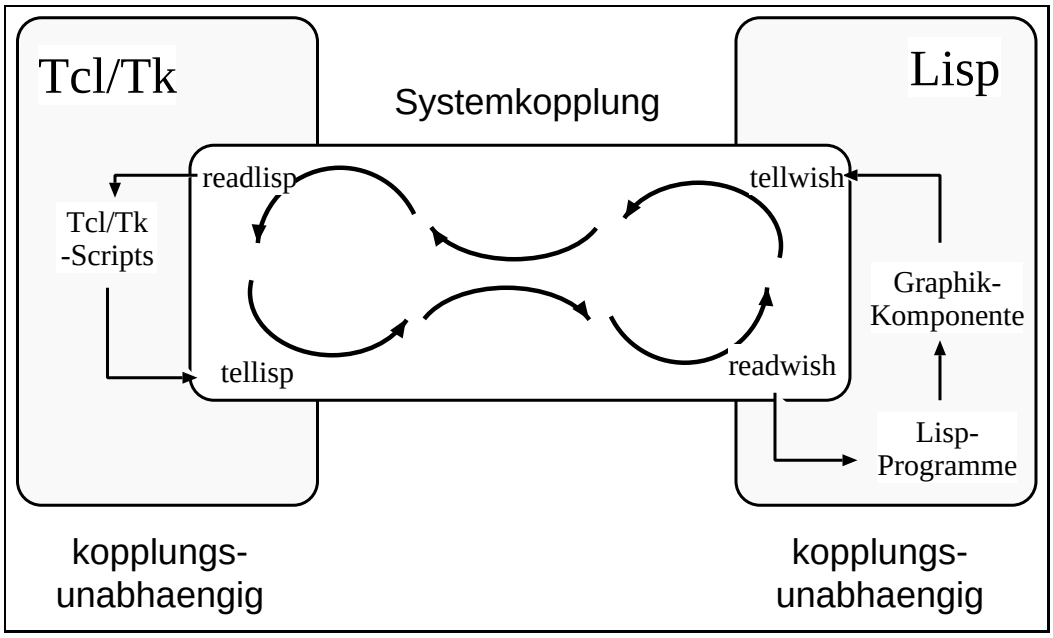

Abbildung 3: Kopplungsspezifische und -unabhängige Teile

Ich werden also versuchen, im folgenden

- einige grundlegende Ideen zu einer Kopplung vorzustellen, und
- einen Großteil des Systems so kopplungsunabhängig zu gestalten, daß je nach Lisp, Betriebssystem- oder persönlichen Vorlieben der eigentliche Kommunikationsmechanismus ausgetauscht werden kann, ohne den Überbau anzutasten.

2.1.1 Die Kopplung von Tcl/Tk und Lucid

Die einfachste Kopplung ist unter Lucid Common Lisp möglich, leider nur mit Hilfe der nicht im verbindlichen Kern von Common Lisp enthaltenen Funktion `run-program`. Diese gestattet aus Lisp heraus das Starten eines externen Prozesses, für dessen Standard-I/O Lisp-Streams als Schnittstellen angegeben werden können. Die Funktion

```
(defun init-interface ()
  (setq *lisp-to-wish-stream*
        (run-program "wish"
                     :input :stream
                     :output :stream
                     :wait nil)
  )
)
```

startet also einen Wish-Interpreter, dessen Ein-/Ausgabe mit dem bidirektionalen Stream `*lisp-to-wish-stream*` verbunden ist. Da dieser direkt den Standard-Output von Wish liest, ist die Prozedur `tellisp` trivial:

```
proc tellisp {lisp_callback} {
        puts stdout $lisp_callback
        flush stdout
}
```

Dabei ist das „Fluten“ mit `flush` notwendig, um evt. unerwünschte Zwischenpufferung auszuschalten. Ähnlich einfach ist hier der umgekehrte Weg:

```
(defun tellwish (tk_commandline)
        (format *lisp-to-wish-stream* tk_commandline)
        (terpri *lisp-to-wish-stream*)
        (force-output *lisp-to-wish-stream*)
)
```

Die Anweisung `(force-output *lisp-to-wish-stream*)` spielt hier dieselbe Rolle, wie oben `flush`.

Was ich im allgemeinen Modell mit `readlisp` bezeichne, ist hier unnötig, weil Lisp über den Standard-Input direkt in die Interpreterschleife gelangt. Auf der Lisp-Seite ist dagegen eine Funktion notwendig:

```
(defun readwish ()
       (read-line *lisp-to-wish-stream*)
)
```

Mit ihr können wir nach der Initialisierung mit `init-interface` und dem Aufbau der Tk-Benutzungsoberfläche eine Interpretationsschleife von Hand basteln:

```
(defun start-loop ()
       (loop (print (eval (readwish))))
)
```

Die Funktion `start-loop` kann tatsächlich noch etwas komplizierter sein, als hier dargestellt. Einerseits werden (wie in 2.2 erläutert) keine direkt ausführbaren Lisp-Kommandos übermittelt, sondern Abkürzungen, denen noch ein entsprechendes Kommando assoziiert werden muß. Andererseits kann es bei Kopplungen, die nur einen einzigen Kanal sowohl für von Tcl/Tk kommende Kommandos als auch für Ergebnisse von Lisp-Anfragen an Tcl/Tk (vgl. 2.2.5) benutzen, notwendig sein, zuerst einmal festzustellen, ob gerade der erste oder der zweite Fall vorliegt.

2.1.2 Ein anderer Ansatz: Tcl/Tk und C-Lisp

Nach dieser Lucid-spezifischen synchronen Kopplung möchte ich noch einige andere Gedanken zur Kopplungsproblematik anbringen, so daß der Leser, der für sein eigenes Lisp eine Kopplung durchführen muß, eine Vorstellung von der Variationsbreite bekommt.

Im Rahmen der KONUS-Systementwicklung [8] hat Thomas Malik eine teilweise asynchrone, ereignisorientierte Kopplung auf der Basis der Tcl-Erweiterung *add-inputs* und der sog. *Named Pipes* (nicht im Standard-UNIX-Umfang enthalten) implementiert. Diese wurde für C-Lisp und Kyoto Common Lisp (KCL) getestet, sollte jedoch auf der Lisp-Seite vollkommen portabel sein.

Grundidee: Tcl startet Lisp als Child-Prozeß und bekommt File-Deskriptoren für direkte Leitungen zum Standard-Input und vom Standard-Output von Lisp zurück. Tcl kann über diese Leitungen Anweisungen an Lisp schicken und Ergebnisse vom Standard-Output lesen. Das ist insofern dual zur Lucid-Kopplung, als dort von Lisp aus der `Wish`-Interpreter gestartet wird, an dessen Standard-I/O dann Streams gebunden sind.

Malik liest nun aber den Lisp-Output nur zu informatorischen Zwecken. Die eigentliche Nachrichtenübermittlung zu Tcl/Tk muß auch in Lisp explizit durch Schreiben auf eine Named Pipe erfolgen. Diese wurde von Lisp und Tcl aus mit demselben Namen eröffnet und ist in Lisp an einen Stream gebunden. Diese Named Pipe muß Tcl nicht ständig daraufhin abfragen, ob eine Nachricht geschickt wurde (so wie das in der Lucid-Kopplung Lisp mit dem bidirektionalen Stream tut), sondern mit Hilfe der *add-inputs*-Erweiterung kann die Named Pipe einen laufenden `Wish`-Interpreter unterbrechen und Befehle direkt in den Interpretations-Toplevel einspeisen. Wir haben hier also zwei interessante Gedanken:

- die Trennung von Standard-Output (z.B. zum Debugging) und expliziter Nachrichtenübermittlung (als Seiteneffekt) und
- die zumindest einseitig asynchrone Nachrichtenübermittlung.

Vorteil: Bei potentiell langwierigen Lisp-Berechnungen ist die Benutzeroberfläche nicht blockiert, sondern die Berechnung wird angestoßen, der Benutzer kann weiterarbeiten, und irgendwann meldet sich Lisp wieder mit einer Aktion auf der Oberfläche.

Problem: Es wäre auch vorstellbar, daß eine lange Berechnung in Lisp ein Ergebnis zu einem Zeitpunkt liefert, zu dem der Benutzer durch seine Aktionen schon einen neuen Systemzustand herbeigeführt hat, mit dem das „alte“ Ergebnis nicht mehr konsistent ist.

Sofern man aufgrund der zur Verfügung stehenden Kopplungshilfsmitel überhaupt eine Auswahl hat, ob man eine voll synchrone, teilweise asynchrone oder sogar vollkommen asynchrone Kopplung realisiert, sollte man sich also der Probleme beider Konzepte (Blockierung durch anderen Systemteil einerseits und nicht mehr aktuelle Antworten andererseits) bewußt sein und sie bei der Entwicklung von Anwendungen berücksichtigen.[3]

Kompatibilität: Die Umstellung einer mit der Lucid-Kopplung geschriebenen Anwendung auf die eben beschriebene und umgekehrt sollte keine Probleme bereiten. Beispiel für die erstere Umstellung: Lisp-seitig werden die *read-eval-print*-Loop und `init-interface`[4] hinfällig, `tellwish` ist auf das Schreiben auf die *Named Pipe* zu ändern. Tcl-seitig wird der Aufbau von Malik genommen und beim Starten von Lisp die Tcl/Tk-Oberfläche durch Laden eines Lisp-Files mit

[3] Die Problematik synchroner oder asynchroner Kommunikation ist ja auch auf dem Gebiet der Betriebssysteme wohlbekannt.

[4] Stattdessen wird von Tcl aus nach dem Starten von Lisp dieses veranlaßt, die Kommunikations-Pipe zu öffnen.

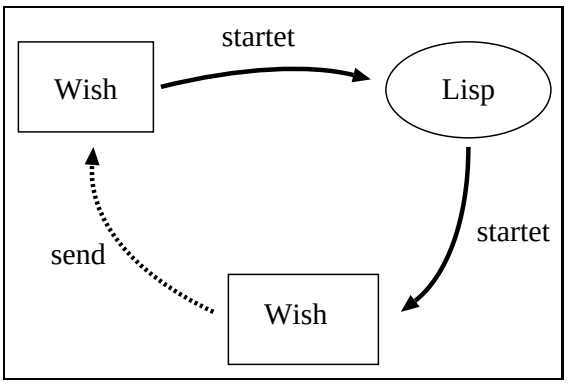

Abbildung 4: Kopplung mit zwei Interpretern

den notwendigen Wish-Kommandos aufgebaut.

Es sei dem Leser überlassen, sich zu überlegen, welche der in Abb. 3 benannten Funktionen `readwish, readlisp, tellwish, tellisp` in diesem Falle unnötig sind, bzw. wie das in einer vollkommen asynchronen und einer vollkommen synchronen Kopplung aussieht.

Noch kurz zwei weitere Kopplungsideen:

- Kommunikation über ein File. Tcl liest explizit dieses File. Da eine Endlosschleife im Wish-Interpreter die Graphik-Oberfläche blockieren würde, wird nur von Zeit zu Zeit, an einen Trigger gebunden, z.B. Mausbewegungen, das File abgefragt, ob neue Informationen eingegangen sind. Statt eines Files können auch *Named Pipes* verwendet werden.
- Simulation der *add-input*-Erweiterung durch einen zweiten Wish-Interpreter (s. Abb. 4): Tcl startet Lisp als Child-Prozeß und hat Zugriff auf die Eingabeleitung. Lisp startet einen zweiten Tcl-Interpreter und hat Zugriff auf dessen Eingabeleitung (diese Variante wurde konzipiert, um das Fehlen bidirektionaler Streams auf Pipes in C-Lisp zu umschiffen). Wenn Lisp dem ersten Interpreter antworten will, befiehlt es dem zweiten, eine Nachricht mit dem Tcl-Kommando `send` an den ersten Interpreter zu schicken. Dort erzeugt dieses `send` eine Unterbrechung und sorgt für die Ausführung des Befehls.

2.2 Der kopplungsunabhängige Überbau in Lisp

2.2.1 Vorüberlegungen

Wer Tcl/Tk kennt und die Verarbeitung von Strings in Lisp beherrscht, könnte nun mit Hilfe der Durchreichfunktion `tellwish` von Lisp aus die volle Funktionalität von Tcl/Tk nutzen. Es stellt sich die Frage, ob dennoch weitere Lisp-Funktionen zur Verfügung gestellt werden sollten, um die Graphikprogrammierung zu vereinfachen bzw. sich hinsichtlich der Bedienung vollständig vom unterliegenden Tcl/Tk zu lösen und ein eigenes darauf aufbauendes Lisp-Graphik-Paket zu erstellen. Solche Funktionen könnten z.B. dazu dienen, das umständliche String-Handling zu ersetzen, die Debugging-Möglichkeiten von Lisp zu nutzen oder objektorientierte Sichten auf die Graphik zu verwirklichen.

Ich habe damit begonnen, Teile dieser Ideen zu implementieren, bin aber dann zu der Überzeugung gelangt, daß es bei den gegebenen Design-Zielen sinnvoller ist, im Lisp nur die mini-

male Funktionalität zur Verfügung zu stellen, die notwendig ist, um einigermaßen bequem die Graphik-Operationen von Tk anzusprechen. Gründe für diese Entscheidung, möglichst nah an Tcl/Tk zu bleiben und einen „minimalen“ Lisp-Überbau zu verwenden, sind:

- Ein Nachbau von Teilen der Tcl/Tk-Welt kann auf den ersten Blick durchaus sinnvoll erscheinen. So könnte man z.B. in Lisp die Daten sammlen, die notwendig sind, um zu testen, ob die `Wish`-Aufrufe korrekt sind (Z.B.: Existieren die benutzten Objekte überhaupt? Sind Koordinatenangaben korrekt?). Damit wäre ein Debugging der Graphik-Programmierung vollständig im Lisp möglich, ohne sich auf die Tcl/Tk-Ebene begeben zu müssen. Eine konsequente Durchführung würde jedoch eine vollständige Nachmodellierung aller Tk-Konzepte erfordern, d.h. beträchtliche Zeit- und Speichererfordernisse mit sich bringen, ohne eine zusätzliche Funktionalität zu bringen. Damit hätte man dann ein Programm im Stile der in Kap. 1 abgelehnten Systeme gebaut, mit all ihren Nachteilen. Hier erscheint es angebrachter, eine effektive Entwicklungsumgebung für Tcl/Tk+Lisp zu entwerfen, die bei Bedarf die Fehlermeldungen von Tk zu benutzen erlaubt.
- Ähnlich wäre der Aufbau einer eigenen Tk-fremden Systemphilosophie auf höherem Abstraktionsniveau, z.B. die Zugrundelegung objektorientierter Gedanken für das Graphik-Paket. Man hätte dieselben Probleme wie oben, verlöre zusätzlich die einfache Bedienbarkeit von Tcl/Tk und hätte schließlich eine beträchtliche Dokumentationsarbeit für den Systembenutzer zu leisten. Bleibt man sehr nah an Tcl/Tk, läßt sich die dort vorhandene Dokumentation (Vorträge, Manual Pages etc.) nutzen.
- Erste Implementierungsversuche zeigen die Kommunikation als Flaschenhals für die Effizienz der Kopplung. Daraus ergibt sich als Ziel eine geschickte Balance von Berechnungen und Daten zwischen Lisp und Tcl/Tk, die möglichst wenig Austausch zwischen den beiden Teilen erfordert. So können z.B. mehrere Tk-Befehle auf den gleichen Daten zu einer Prozedur zusammengefaßt werden, statt jeden Befehl einzeln aus Lisp zu starten. Auch sollte Kommunikation zur Laufzeit möglichst Tk-intern gehalten werden, wenn der Umweg über Lisp nicht unbedingt erforderlich ist (z.B. weil Daten- oder Kontrollstrukturen von Lisp benötigt werden). Daraus ergibt sich, daß der Programmierer auf jeden Fall die Grundlagen der Tcl/Tk-Programmierung kennen muß. Ihm jetzt zusätzlich eine neue, andere Lisp-Graphik-Schnittstelle aufzubürden, würde die Systembenutzung unnötig verkomplizieren.

Aus diesen Gründen erscheint es angebracht, daß die Graphik-Programmierung aus Lisp möglichst Tk-ähnlich ist. Zumutbar erscheint mir dabei die Verwendung einiger weniger Grundregeln zur Überführung von Tk-Konzepten in analoge Lisp-Konstruktionen. Es bleibt also die Frage, wie ein „minimaler“ Überbau in Lisp aussieht, der bei aller Tk-Nähe ein bequemes, immer noch „Lisp-typisches“ Arbeiten ermöglicht.

Einige Anforderungen für die Lisp-Konstrukte drängen sich auf:

1. Die Benutzung sollte dem Programmierer wenig Quotierungen und String-Verarbeitung abverlangen.
2. In Graphik-Befehlen muß die Benutzung von Variablen und evaluierbaren Ausdrücken zugelassen sein.
3. Aus Effizienzgründen sollte die Kommunikation mit Tcl/Tk gering gehalten werden. Ferner können keine als Strings nicht sinnvoll darstellbaren Strukturen über den Kommunikationsweg geschickt werden (z.B. Hash-Tabellen, zyklische Strukturen, Lambda-Closures oder ähnliches).

4. Graphikbezogene Funktionsaufrufe sollten leserlich bleiben.
5. Sie sollten sich ferner nah an der Tcl/Tk-Syntax bewegen.
6. Die von Tcl/Tk lieferbaren Informationen müssen in Lisp erreichbar und verarbeitbar sein.

2.2.2 Die Architektur des Lisp-Überbaues

Zum Erfüllen dieser Anforderungen wurde der Lisp-Überbau für Tcl/Tk folgendermaßen strukturiert:

- Der Benutzer arbeitet nicht direkt mit Strings und `tellwish`, sondern mit dem Lisp-Makro `wish`. Dieses nimmt die an Tcl/Tk zu übermittelnden Kommandozeilen als Folgen von Ausdrücken, evaluiert diese, wenn nötig, überführt die Ergebnisse und konkateniert diese. Damit erfüllt es Kriterium 2 und 1. Die Erfordernisse von Kriterium 3 ergeben sich im Zuge der Evaluation, wenn bei der Auswertung von Ausdrücken Lisp-Funktionsaufrufe als Argumente von Wish-Befehlen (insbesondere bei den Auswahlen von Menüs) nicht direkt an Tcl/Tk geschickt werden, sondern in Form symbolischer Namen. Dies kann durch die Verwendung von Sonderfunktionen erreicht werden, die ich im nächsten Punkt schildere.

  Die Wish-Aufrufe erfolgen natürlich normalerweise ausschließlich wegen ihres Seiteneffekts, d.h. hier zur Änderung der durch Tcl/Tk aufgebauten Oberfläche. Ich habe nur noch eine Klasse von Makros oberhalb der `Wish`-Ebene vorgegeben, nämlich Makros zur Erzeugung graphischer Objekte, z.B. `make-button, make-message, ...` zur Erzeugung von Buttons, Messages usw. Diese Makros habe ich deshalb zur Verfügung gestellt, weil es häufig der Fall sein wird, daß man ein graphisches Objekt erzeugt und seinen Namen später in weiteren Befehlen für seine Plazierung, Konfiguration usw. benötigt. Deshalb wurden diese Erzeugermakros eingeführt, die nicht nur einen Seiteneffekt besitzen, sondern den Namen des erzeugten Objekts noch als Wert zurückliefern.

  Zur Vereinfachung der Schreibweise kann der Benutzer natürlich noch eigene Makros über `wish` bauen. Ich habe aber davon abgesehen, weil das wieder die Minimalitätsanforderung für die Schnittstelle verletzen würde.

- Für die Kriterien 4 und 1 gibt es einige Sonderfunktionen, die eine abkürzende, Lisptypischere Schreibweise Tcl/Tk-typischer Syntax ermöglichen:
  - das Lisp-Makro {} überführt Argumente *arg_1 ... arg_n*
    in einen String, bestehend aus der Zeichenfolge: {*arg_1 ... arg_n* }
  - das Lisp-Makro *[]* überführt Argumente *arg_1 ... arg_n*
    in einen String, bestehend aus der Zeichenfolge: *[arg_1 ... arg_n]*
  - das Lisp-Makro *dq*[5] überführt Argumente *arg_1 ... arg_n*
    in einen String, bestehend aus der Zeichenfolge: *"arg_1 ... arg_n"*
  - das Lisp-Makro `lispcommand` gewährleistet Bedingung 3. Da es nicht wünschenswert erscheint, Lisp-Funktionsaufrufe inklusive Bindungsumgebung über die Kommunikationspipeline zu schicken, wird einem dem Makro `lispcommand` als Argument mitgegebenen Funktionsaufruf ein einfacher String zugeordnet, der als Callback dann an Wish übermittelt werden kann. Bei einem Aufruf von Wish an Lisp wird dann wieder in Lisp der entsprechende Befehl assoziiert.

[5] *doublequote*

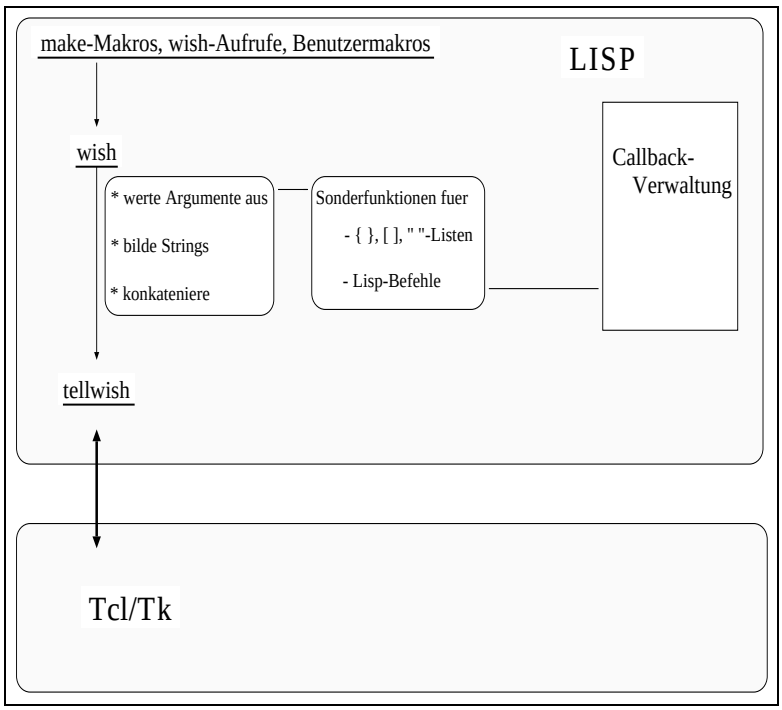

Abbildung 5: Die Lisp-seitige Systemstruktur

- Für Ziel 6 werden einige Funktionen zum Lesen von Tk-Information nach Lisp angeboten. Zur Minimierung von Kommunikation (3) werden Tk-Ergebnisse nicht ständig gelesen, sondern sind bei Bedarf explizit abzufragen.

Insgesamt ergibt sich die in Abb. 5 dargestellte Struktur des Lisp-Überbaues (ohne die Abfrage-Befehle aus 2.2.5):

Einige Teile dieser Struktur werden im folgenden noch etwas näher erläutert. Darüberhinaus sei auf das Beispiel in Kapitel 2.3 verwiesen. Zur weiteren Vereinfachung sei auch noch einmal erwähnt, daß dann, wenn keine Lisp-Aufrufe als Kommandos übermittelt werden müssen, die gesamte Wish-Anweisung natürlich auch direkt als String an das Makro `wish` übergeben werden kann.

2.2.3 Die Lisp-Callbacks

Die Datenstruktur `*list-of-callbacks*` implementiert eine eineindeutige Abbildung von mit `lispcommand` verarbeiteten Lisp-Ausdrücken und als Tcl/Tk-Callbacks verwendbaren Strings. Sie gestattet folgende Operationen:

- `insert-lispcommand-in-loc!` erzeugt für seinen Eingabe-Ausdruck einen neuen Callback-String, speichert die bijektive Zuordnung von Callback und Ausdruck und liefert den neuen Callback als Wert zurück.
- `lispcommand-in-loc?` nimmt einen Lisp-Ausdruck und überprüft, ob für ihn schon Callback existiert.

- `get-lispcommand-from-loc` findet zu einem Callback den assoziierten Lisp-Ausdruck.
- `get-callback-from-loc` findet zu einem Lisp-Ausdruck den assoziierten Callback.
- `print-loc` gibt den aktuellen Zustand der Callback-Liste aus.

Damit läßt sich die Funktion `lispcommand` folgendermaßen implementieren:

```
(defun lispcommand (p)
   (let ((cmd (if (lispcommand-in-loc? p)
                  (get-callback-from-loc p)
                  (insert-lispcommand-in-loc! p)
              )
         )
        )
        ({} tellisp p)
   )
)
```

2.2.4 Die Graphik-Befehle in Lisp

Die Tk-Kommandos lassen sich in vier Gruppen einteilen. Die genaue Syntax ist der Tcl/Tk-Dokumentation zu entnehmen, die grundlegenden Merkmale lassen sich aber schon aus einigen Beispielen erkennen:

1. **Kommandos zur Erzeugung graphischer Objekte:**
   `message .d.top -width 3i -bd 2 -relief raised -text ``Mate-Tee ist gesund!''`
   erzeugt ein Objekt vom Typ `message` mit Namen `.d.top`, einer Breite von 3 Zoll, einer Rahmenbreite von 2 Pixels, das erhaben dargestellt wird und den Text `Mate-Tee ist gesund!` anzeigt.
2. **Kommandos für das Geometrie-Management: der Packer und der Placer (Plazierung graphischer Objekte):**
   `pack append .b .b.header {top fillx} .b.hscroll {bottom fillx}`
   fügt die beiden Objekte `.b.header` und `.b.hscroll` nach ihrem Parentwindow `.b` in die Packerliste ein und benutzt dafür die Packer-Optionslisten {`top fillx`}, {`bottom fillx`}.
3. **Widget-Kommandos (Veränderung graphischer Objekte):**
   `.d.top configure -bd 3 -relief sunken`
   verändert in der Konfiguration des Objekts `.d.top` die Attribute `borderwidth` und `relief` auf die Werte `3` und `sunken`.
4. **Verbindungskommandos, z.B. für keystroke bindings, input focus, window management:**
   `bind .d.top <1> {puts stdout ``Hello World''}`
   bindet innerhalb des Objektes `.d.top` an das Ereignis „linke Maustaste gedrückt“ den Befehl, den String `Hello World` auf die Standardausgabe zu schreiben.

Wenn auch Tcl-intern ausschließlich Strings verarbeitet werden, so lassen sich doch nach außen hin und für die Verwendung im Lisp mehrere Arten auftretender Argumente identifizieren:

- Objektnamen (wie z.B. `.`, `.top`, `.top.hscroll`, `.top.vscroll`). Sie müssen Pfadnamen sein, d.h. durch Erweiterung eines bereits vorhandenen Objektnamens um einen Punkt `.` und Text (ohne Leerzeichen) entstanden sein. Sie dürfen nur aus Kleinbuchstaben bestehen!

- Schlüsselworte und Kommandonamen (wie z.B.`message`, `-width`, `-borderwidth`, `pack`, `append`, `configure`, `bind`). Auch sie bestehen nur aus Kleinbuchstaben.

- Zahlen oder andere für Tcl im Kontext sinnvoll zu interpretierende Strings (wie z.B. `2` oder `<1>`).

- Durch Tcl-Sprachkonstruktionen entstandene komplexere Strukturen (wie z.B. Listen {`puts stdout ``Hello World''`} oder längere Strings ``` ``Trinkt gesunden Mate-Tee!'' ```). Hier ist von Lisp aus zu beachten, daß z.B. in Texten die Großschreibung erhalten bleibt und daß Anführungsstriche um Strings als solche ankommen und nicht beim Konkatenieren im `Wish`-Makro unter den Tisch fallen.

Aufruf von Tk-Kommandos aus Lisp:

Kommandonamen zur Widget-Erzeugung werden durch das Präfix `make-` ergänzt, d.h. aus `message` wird `make-message`. Die `make-`Befehle liefern als Wert den Pfadnamen des erzeugten Widgets zurück.

Diese Kommandos setzen auf dem Makro `wish` auf, das beliebig viele Argumente nimmt, diese in einen Tcl/Tk-Kommandostring überführt und mit `tellwish` an Tcl/Tk weitergibt. `Wish` wird nur wegen des Seiteneffekts aufgerufen. Auch Tk-Befehle zur Abfrage von Werten können nicht zum automatischen Importieren nach Lisp verwendet werden. Stattdessen gibt es die explizite Werteabfrage mit den in Unterkap. 2.2.5 aufgeführten Funktionen.

Konventionen zur Umwandlung von Lisp-Argumenten in Tcl-Strings:

Alle Zahlen und Symbole werden in Strings aus Kleinbuchstaben umgewandelt, auswertbare Ausdrücke zuvor evaluiert. Strings werden unverändert übernommen, so daß Dinge, die mit Großbuchstaben in Tcl/Tk ankommen sollen, direkt als Strings eingegeben werden müssen.

Für die Manipulation von Widget-Namen gibt es die Funktion `expand-pathname`, die 2 Argumente bekommt und den in Kleinbuchstaben umgewandelten String aus der Konkatenation beider Argumentwerte nach Einfügung eines „.“ als Wert liefert.

Bei Verwendung des Root-Windows „.“ von `Wish` als Pfadname ist zu beachten, daß Lisp-Reader den Punkt als Symbol nicht akzeptieren, da er eine festgelegte Bedeutung in Dotted Pairs hat. Er muß also immer als String mit Anführungszeichen angegeben werden.

2.2.5 Abfrage-Befehle

Wie oben schon erwähnt, müssen alle benötigten Tcl/Tk-Informationen aus Lisp explizit abgefragt werden.

Einerseits kann dies (bei geeignetem Kommunikationskonzept) unnötiges Lesen eines Kommunikationskanals ersparen. Andererseits würde eine in sich schlüssige Lösung, die automatisch Daten nach Lisp importiert, wieder einen recht umfangreichen Nachbau von Teilen der Tcl/Tk-Welt bedingen (wenn z.B. bei der Anlage eines Checkbuttons auch automatisch eine Lisp-Variable erzeugt würde, die immer den aktuellen Zustande des Buttons hält). Deshalb wurde auch hier eine

„Minimallösung“ gewählt, die bisweilen etwas umständlicher ist und auch vom Programmierer erwartet, daß er sich jederzeit darüber im klaren ist, was gerade in Tcl/Tk passiert.

In Tcl/Tk kann man auf verschiedene Weise vom Interpreter Informationen bekommen (die normalerweise auf Standard-Output landen):

1. Durch Abfrage von Variablenwerten (z.B. die einem Checkbutton zugeordnete Zustandsvariable, die einem Menubutton zugeordnete Textvariable oder benutzerdefinierte Hilfsvariablen). Um den Wert zu lesen, kann man ihn z.B. auf Standard-Output schreiben lassen: `puts stdout $var`
2. Durch Verwendung von Tk-Kommandos in Form unvollständiger Spezifikationen von Konfigurationsoptionen, die eine Beschreibung des aktuellen Wertes für die betrachtete Option zurückliefern (z.B. könnte die Anweisung `.mb configure -relief` für einen Menubutton `.mb` die Beschreibung `-relief relief Relief flat raised` liefern). Von dieser Beschreibung interessiert aber in der Regel nur der letzte Wert `raised`.
3. Durch Verwendung von Kommandos, die eine Ausgabe auf Standard-Output zurückliefern, der vollständig von Interesse ist (nicht nur der letzte Teil): z.B. könnte `wm geometry .` den Wert `200x200+1277+37` liefern.
4. Durch Auswertung von Tcl-Ausdrücken. Beispiel: `puts stdout [expr 17 + 4]`.

Wir stellen zum Abdecken dieser vier Fälle vier Typen von Abfragemakros zur Verfügung:

1. Makros mit Namenspräfix `wvv-` (wie *wish-variable-value*) veranlassen das Schreiben des Wertes einer Tcl/Tk-Variablen auf die Kommunikationsleitung zu Lisp.
2. Makros mit Präfix `wcl-` (wie *wish-command-last*) schneiden das letzte Wort der Antwortbeschreibung für eine Konfigurationsanfrage ab.
3. Makros mit Präfix `wcc-` (wie *wish-command-complete*) geben die vollständige Antwort auf ein Tk-Kommando zurück. Dieses Tk-Kommando wird genau analog zur Eingabe an das Makro `wish` übergeben.
4. Makros mit Präfix `wec-` (wie *wish-eval-complete*) geben den in Wish auf Standard-Output gelegten Wert eines Tcl-Ausdrucks zurück. Auch hier wird die Eingabe wie an das Makro `wish` durchgeführt. Allerdings wird das umschließende “[expr .] “ automatisch ergänzt.

Der Tcl-Interpreter kennt nur Strings als Daten. Inhaltlich entsprechen die auftretenden Daten aber den Typen:

1. String
2. Zahlen
3. Boolean
4. Listen

Damit die Tcl-Daten in Lisp ihrem Inhalt entsprechend verwendet werden können (z.B. Boolesche Werte, in Tcl als Strings “0“ und “1“ repräsentiert, zur Steuerung bedingter Anweisungen), gibt der zweite Teil des Makronamens an, in welchen Lisp-Datentyp das Ergebnis der Tcl/Tk-Anfrage konvertiert werden soll:

- In einen String: `string`
- In eine Zahl: `number`
- In einen Booleschen Wert (T,NIL): `bool`
- In eine Liste von Strings: `stringlist`
- In eine Liste von Symbolen: `symbollist`
- In eine Liste von Zahlen: `numberlist`

Wir erhalten also eine Reihe von Makros aus Präfix zur Bestimmung des Anfragetyps und Bezeichner zur Charakterisierung der Antwortkonversion, die eine Tcl/Tk-Variable oder -Anweisung als Argument nehmen. Beispiele:

`(wvv-bool cb1)`: falls `cb1` die einem Checkbutton zugeordnete Zustandsvariable ist, kommt als Wert T oder NIL zurück.

`(wcl-string .mb configure -relief)`: könnte z.B. den String "`raised`" liefern.

`(wcc-numberlist wm maxsize)`: könnte z.B. die Liste `(200 300)` zurückgeben.

Die Verwendung dieser Anfragemöglichkeiten wird im nächsten Abschnitt gezeigt.

2.3 Ein kleines Beispiel

Als kleines Beispiel, insbesondere zur Darstellung der möglichen Verwendung der Abfrage-Makros, betrachten wir die in Abb. 6 definierte Lisp-Funktion.

Der Aufruf der Funktion baut die in Abb. 8 gezeigte kleine Graphik-Oberfläche auf.

Die intendierte Bedeutung der Elemente ist die folgende: Die weiter unten noch anzugebende Funktion `bewege-button` soll dann, wenn der Checkbutton `Sperren` nicht aktiviert ist, auf Drücken des Buttons `Tu es` den im unteren Frame angezeigten Button mit den ersten zehn Zeichen des in das Text-Widget geschriebenen Textes füllen und den Button dann in horizontaler und vertikaler Position um jeweils 40 Pixel bewegen. Ob diese Pixelanzahl von den aktuellen Koordinaten abgezogen oder ihnen hinzuaddiert wird, hängt vom Wert der Variablen `cv` ab, der mittels der beiden Radiobuttons selektiert wird. Dieser Effekt wird durch die Funktion `bewege-button` (s. Abb. 7) implementiert.

Wenn wir nun `(start-loop)` aufrufen, die Sperre nicht betätigen, den „+"-Knopf drücken, den Text „Trink Mate" eingeben und dann die Ausführung mit dem `Tu es`-Button veranlassen, ergibt sich das Bild in Abb. 9.

Schon dieses kleine Beispiel zeigt das Zusammenspiel von Oberflächenaktionen und Lisp-Berechnungen sowie den Informationsaustausch zwischen beiden Systemen.

Bemerkung: Analog zum hier vorgestellten Aufbau können auch bei einer Kontrolle von Tcl aus einfach Funktionen geschrieben werden, die Lisp nach Werten fragen. Konsequent fortgedacht ergibt sich ein verallgemeinertes System, das beide Komponenten unabhängig laufen läßt, asynchrone Methoden zur Einspeisung von Kommandos in den Interpreterzyklus erlaubt und in beide Richtungen Informationen abfragen kann. Sofern die verwendete Kopplung asynchrone Kommunikation ermöglicht, stellt die Implementierung dieses Konzeptes keine grundsätzliche Schwierigkeit dar. Ich verzichte dennoch darauf, weil es üblicherweise ausreicht, eine steuernde

```
(defun baue-auf ()
   (init-interface)
   (setf *x* 70 *y* 50)
   (make-frame  .oben  -width 400  -height 10)
   (make-frame  .unten -width 400  -height 390)
   (let ((mein-button (make-button .unten.b1 )))
      (wish place  .unten.b1 -in .unten -x *x* -y *y*)
      (wish pack append "." .oben  ({} left top)
                            .unten ({} left top expand))
      (make-radiobutton .oben.r1 -text     (dq "+")
                                 -variable  rv)
      (make-radiobutton .oben.r2 -text     (dq "-")
                                 -variable  rv)
      (wish set rv r1)
      (make-checkbutton .oben.c  -text     (dq "Sperren")
                                 -variable  cv)
      (make-button .oben.b       -text      (dq "Tu es")
                                 -command (lispcommand (bewege-button mein-button)))
      (make-text .oben.t -width 20 -height 1 -relief raised -borderwidth 2)
      (wish pack append .oben .oben.r1 ({} top left)
                              .oben.r2 ({} top left)
                              .oben.c  ({} top left)
                              .oben.b  ({} top left)
                              .oben.t  ({} top left expand) )
    )  ;; of let
)  ;; of defun
```

Abbildung 6: Die Funktion `baue-auf`

```
(defun bewege-button (b)
   (if (not (wvv-bool cv))
       (let ((functor (if (equal (wv-string rv)) "r1" #'+ #'-))
             (neuer-text (we-string .oben.t get "1.0 1.10")))
          (wish b configure -text (dq neuer-text))
          (wish place configure b  -x (setf *x* (apply functor (list *x* 40)))
                                   -y (setf *y* (apply functor (list *y* 40))))
   )  ;; of if
)  ;; of defun
```

Abbildung 7: Die Funktion `bewege-button`

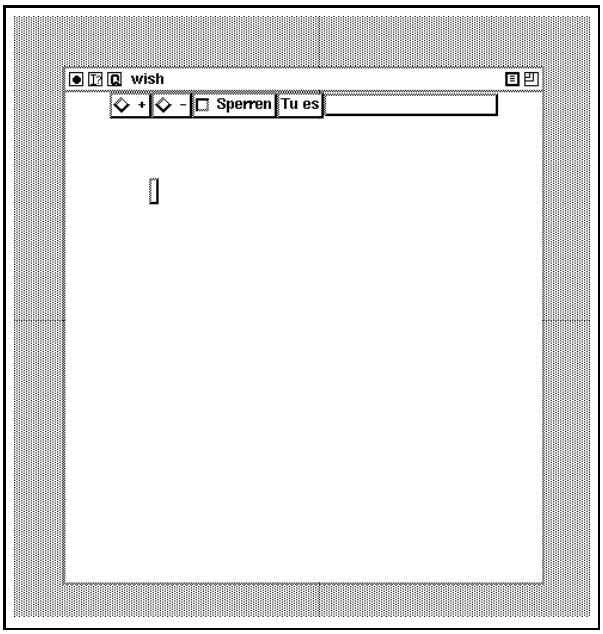

Abbildung 8: Unsere kleine Beispiel-Oberfläche

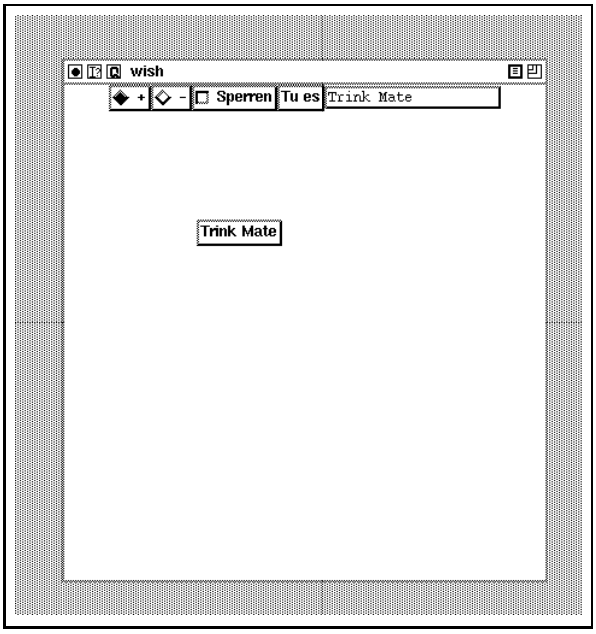

Abbildung 9: Die Auswirkungen der Benutzerinteraktion

Komponente zu besitzen, in der man den Kontrollfluß primär ansiedelt und eine untergeordnete, die auf Anweisung reagiert.

3 Ein Browser für Begriffshierarchien in TAXON

3.1 Problemstellung

Das terminologische System TAXON [3] berechnet in seiner TBox aus einer gegebenen Menge deklarativer Beschreibungen von Begrifflichkeiten (wir nennen sie in diesem Kontext *Konzepte* und ihre Beschreibungen *Konzeptdefinitionen*) die Menge aller Inklusionsbeziehungen (hier: *Subsumptionen*) zwischen solchen in ihren Definitionen deskriptiv gegebenen Mengen.

Diese Subsumptionsbeziehungen bilden eine Halbordnung, die man recht natürlich und übersichtlich als gerichteten azyklischen Graph (DAG) darstellen kann. Da eine solche Darstellung schon einige der gängigen Anfragemöglichkeiten eines terminologischen Systems unmittelbar beantwortet (z.B. Finden der direkten Ober- und Unterkonzepte) und andererseits eine textuelle Darstellung – wie in Lisp üblich – sehr aufwendig und unübersichtlich wird, drängt sich hier eine graphische Darstellung dieses DAG's auf.

Ein solcher *Taxonomiebrowser* kann zu rein informatorischen Zwecken dienen, um das erfaßte Wissen effektiv und bequem abzufragen, kann aber auch die Basis für spätere Erweiterungen zu Zwecken der Wissenseditierung oder -validierung sein.

Beispiel: Wir betrachten ein Expertensystem, das den Designprozeß eines Maschinenbauingenieurs unterstützt. Eine hierarchisch organisierte Werkstoffwissensbasis ist in TAXON modelliert. Während eines Entwurfs möchte der Ingenieur einen geeigneten Werkstoff auswählen. Er weiß zwar, daß für die gegebene Aufgabe eine bestimmte Kunststoffamilie geeignet ist, aber nicht, welches spezielle Produkt innerhalb dieser Familie am sinnvollsten ist. Mit Hilfe des Taxonomiebrowsers könnte er sich zu dem Konzept begeben, das diese Familie repräsentiert und zuerst einmal alle Unterfamilien betrachten. Aufgrund der Unterscheidungsmerkmale in den Konzeptdefinitionen könnte er sich für eine Unterfamilie entscheiden und innerhalb dieser fortfahren, bis er an einen Blattknoten des Begriffsbaumes angelangt ist. Dieser repräsentiert dann einen konkreten Werkstoff.

3.2 Funktionalität

Der Browser soll zum bequemen Navigieren zwecks Information Retrieval in einer großen statischen Begriffshierarchie dienen. Da außerdem der Aufbau der graphischen Darstellung einer solchen großen Hierarchie recht aufwendig sein kann, sollte der Taxonomie-Browser beim Hochfahren des umfassenden Systems (man denke sich z.B. einen Design-Assistenten im Sinne des oben ausgeführten Beispiels, wo das taxonomische Wissen nur einen kleinen Teilaspekt darstellt) einmalig initialisiert werden. Danach werden die erzeugten Graphiken iconifiziert und nur bei Bedarf aufgepopt und wieder geschlossen.

Zur Einbindung in das umfassende System bekommt die Funktion zum Aufbau des Browsers als Argument ein Parent Window (das als Wurzel für alle erzeugten Fenster benutzt wird) und liefert als Ergebnis einen Menubutton mit zwei Auswahloptionen zum Deiconifizieren der beiden eigentlichen Browser-Fenster.

Das eine Fenster zeigt eine „normalgroße“ Darstellung der Taxonomie, d.h.: jedes Konzept wird als Menubutton gezeigt, dessen Größe von der Länge des Konzeptnamens abhängt. Pfeile zwischen Konzepten stellen Subsumptionsbeziehungen dar. Jeder Menubutton erlaubt die Auswahl aus einer Reihe von Informationsdiensten zum betreffenden Konzept: Begriffsdefinition, direkte Unter- und Oberbegriffe, alle Unter- und Oberbegriffe, synonyme Begriffe (d.h. semantisch

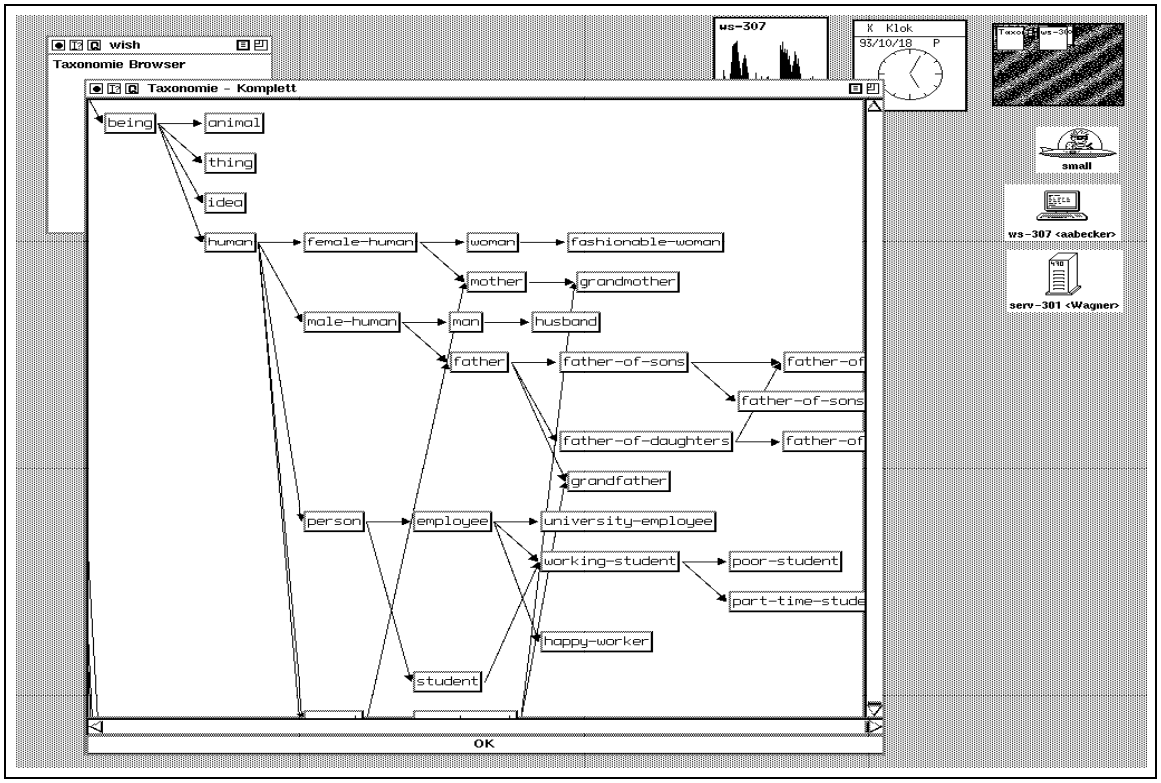

Abbildung 10: Ein Teil der Taxonomie in voller Größe

gleichwertige Konzepte), Ausschnittdarstellung.

Die von TAXON erfragten Antworten auf diese Fragen werden in frei positionierbaren Toplevel-Fenstern (Info-Boxen) angezeigt. Dabei zeigt die *Ausschnittdarstellung* einen Teilbaum des Begriffsbaumes mit dem aktuell betrachteten Knoten als Wurzel und einer Tiefe von maximal drei Schichten subsumierter Konzepte.

Abb. 10 läßt schon erkennen, daß diese Darstellung der Taxonomie in voller Größe bei größeren Hierarchien schon wieder viel zu unhandlich wird. Deshalb wird ein zweites Fenster geöffnet, in dem eine schematische Kleindarstellung der Taxonomie ausgegeben wird. Hier ist jedes Konzept durch ein kleines maussensitives Rechteck repräsentiert. Erst nach Aufforderung mit *Mouse-Left* wird der zugehörige Menubutton aufgebaut, der bei *Mouse-Right* wieder verschwindet. Abb. 11 zeigt die Schemadarstellung mit einer Info-Box für die Definition des betrachteten Konzepts.

Um den Einstieg in diese Schemadarstellung zu erleichtern, gibt es links eine Listbox mit einer alphabetischen Auflistung aller Konzepte der Taxonomie. Nach Selektion eines Konzeptnamens mit *Mouse-Left* kann der betreffende Knoten mit *Mouse-Left* markiert bzw. mit *Mouse-Right* wieder demarkiert werden. Diese Einstiegshilfe in die Übersichtsdarstellung zusammen mit der Ausschnittsdarstellung (die in diesem Fall eine echte Ausschnitts*vergrößerung* ist) und der Möglichkeit, durch Ausschnittauswahl innerhalb eines Ausschnitts (kaskadierendes Öffnen) sich immer weiter entlang eines interessanten Pfades zu hangeln, dürfte die vielversprechendste Art sein, im Sinne des Eingangsbeispiels den Browser im eigentlichen Wortsinne zu verwenden. Abb. 12 zeigt eine solche Suche in der Taxonomie durch sukzessives Öffnen von Ausschnittvergrößerungen.

Ich werde im folgenden die Struktur der Implementierung des Browsers vorstellen. Dabei gehe ich

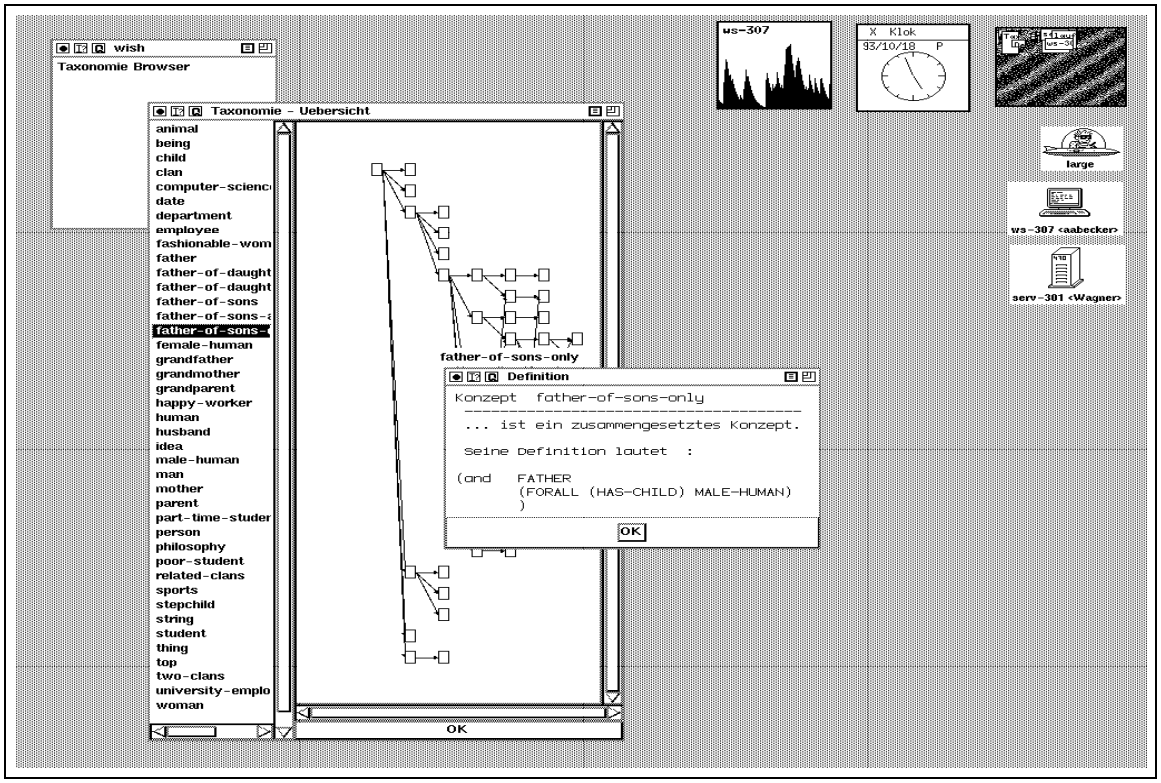

Abbildung 11: Die Übersichtsdarstellung der Taxonomie

detailliert genug auf die Schnittstellen zum DAG-Layout einerseits und zu TAXON andererseits ein, um den Leser in die Lage zu versetzen, selbständig den Browser zu modifizieren oder ähnliche Anwendungen zu erstellen. Die Implementierung des Browsers selbst wird eher konzeptionell beschrieben, für Details sei auf die Implementierungsfiles verwiesen.

3.3 Implementierung

3.3.1 Das DAG-Layout-Paket von Jürgen Wagner

Zum DAG-Layout wurde ein Lisp-Programmpaket von Jürgen Wagner [6] benutzt, dessen Struktur übersichtsweise in Abb. 13 dargestellt ist.

Die Funktion dag erhält folgende Argumente:

- Eine Liste `nodes` von Knoten der Quell-Datenstruktur (QDS, diejenige Struktur, die ich graphisch darstellen möchte), die als Einstiegsknoten für das Layout dienen sollen.
- Einen Schlüsselwortparameter `format`, der angibt, von welchem ausgabespezifischen Postprocessor die Layout-Daten verarbeitet werden sollen. Momentan gibt es hierfür nur die Eingabemöglichkeit `:tk` für eine Visualisierung in Tcl/Tk. Dies ist auch der Default-Wert.

[6] Jürgen Wagner, früher Projektleiter beim Institut für Arbeitswissenschaft und Organisation der Fraunhofer-Gesellschaft, inzwischen erreichbar an der Stanford University unter der E-Mail-Adresse `gandalf@csli.stanford.edu`

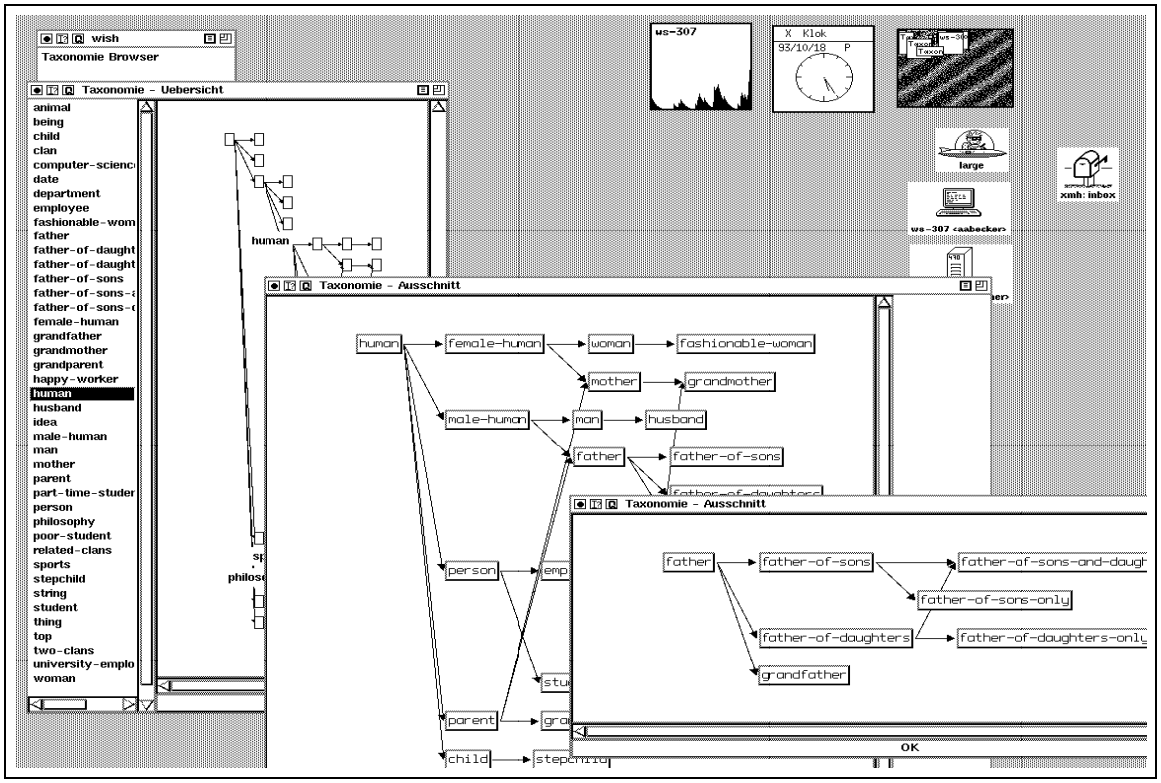

Abbildung 12: Kaskadierende Ausschnittvergrößerungen

- Einen Restlistenparameter für sonstige Argumente, die der Dag-Handler benötigt. Diese werden als Schlüsselwortparameter angegeben. Auf die möglichen Eingaben gehe ich weiter unten noch ein.

Die Funktion holt in Abhängigkeit vom gewünschten Output-Postprocessing (denkbar wären außer der Tcl/Tk-Verarbeitung z.B. auch Postprocessoren für verschiedene Graphik- oder Zeichenprogrammformate oder für Postscript) aus einer Handlerliste den zur Eingabe `format` passenden DAG-Handler und wendet ihn auf die gegebenen Parameter an.

Die Funktion `dag` ist also das einzige, was zum Aufruf der DAG-Darstellung bekannt sein muß. Sie überführt die QDS direkt in das Ergebnis des gewünschten Postprocessings.

Für die Installation eines Tcl/Tk-Handlers müssen wir also einen solchen definieren und in die Handler-Liste eintragen. Dazu gibt es:

Das Makro defdag erhält folgende Argumente:

- Obligatorisch den Formatnamen.
- Eine Liste `additional-args` mit Argumentnamen, die dem neuen DAG-Handler zusätzlich zu den Standardoptionen noch als Schlüsselwortparameter übergeben werden können sollen.
- Als Restlistenparameter den Body des neuen DAG-Handlers.

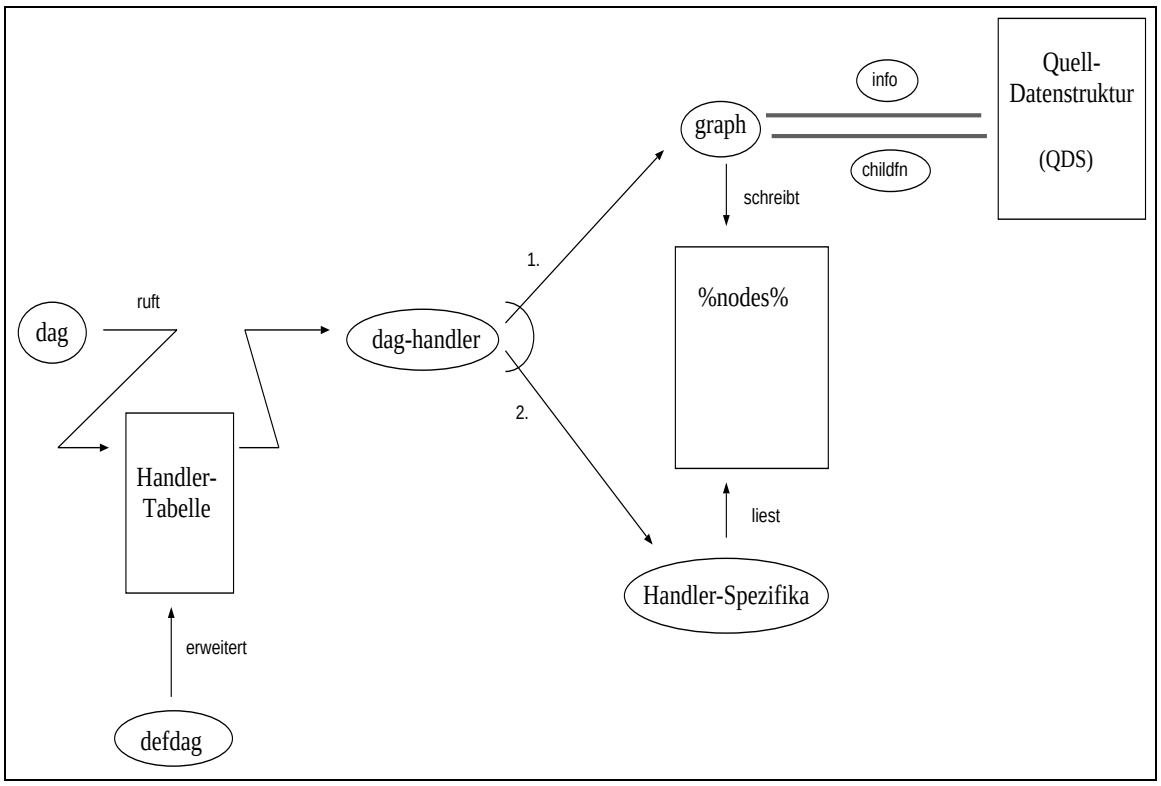

Abbildung 13: Prozeduren und Datenstrukturen beim DAG-Layout

Die Funktion definiert eine neue Handler-Funktion mit dem angegebenen Body und trägt sie unter dem Formatnamen in die Handler-Liste ein. Der Body hat Zugriff auf folgende Variablen, die der oben beschriebenen Funktion `dag` als Schlüsselwortparameter mitgegeben werden können: `width, height, link, sep, margin, depth, imargin, minchars, children, info, format, arg`; außerdem natürlich auf die Variable `nodes` und ggf. in der Liste `additional-args` angegebene spezifische Schlüsselwortargumente für diesen Handler. Bedeutung und Default-Werte der Variablen sind weiter unten erklärt.
Typischerweise hat ein DAG-Handler folgende Struktur: zuerst wird die Funktion `graph` gerufen, die das Layout des betrachteten DAGs berechnet. Sie liefert als Wert die Größe der benötigten Box zurück und als Seiteneffekt das Layout des Graphen in der Datenstruktur `%nodes%`. Die Daten in `%nodes%` werden dann in einem Format-spezifischen Teil weiterverarbeitet. Im Falle der Tcl/Tk-Anbindung wird z.B. (unter anderem) für jeden in `%nodes%` repräsentierten Knoten ein Befehl zum Zeichnen einer Box mit den gegebenen Koordinaten von Lisp nach Tcl/Tk geschickt.

Die Funktion graph erhält folgende Argumente:

- Obligatorisch:
  - **roots** – die Wurzelknoten des Graphen.
  - **childfn** – eine einstellige Funktion, die für einen Knoten der Quell-Datenstruktur die Liste seiner Nachfolgerknoten berechnet.
- Schlüsselwortparameter:

- **info** – einstellige Funktion, die auf einen Knoten der QDS angewandt, dazu eine Liste von Strings liefert, die in der entsprechenden Box des zu zeichnenden DAGs ausgegeben werden.
- **depth** – Wert ist NIL oder eine Zahl; gibt die maximale Tiefe der zu betrachtenden Knoten der QDS an – ausgehend von den Einstiegsknoten.[7]
- **minchars** – minimale Anzahl von Zeichen in einem Box-Text (default: 1).
- **imargin** – feste Zahl von Pixeln, die als Freiraum in jeder Box den Text umgeben sollen (default: 1).
- **margin** – extra Rand zwischen Graph und umgebender Box (default: 0).
- **width** – durchschnittliche Breite eines Zeichens im Label-Text (default: 1).
- **link** – minimale horizontale Distanz zwischen zwei durch eine Kante verbundenen Boxen (default: 1).
- **height** – Höhe einer Textzeile (default: 1).
- **sep** –minimale vertikale Distanz zwischen zwei adjazenten Boxen (default: 1).

Die Funktion führt eine einfache Layout-Berechnung für einen DAG zur gegebenen Quell-Datenstruktur durch, die dann später für Handler-spezifisches Postprocessing genutzt werden kann. Die globale Variable `*dag-max-nodes*` gibt die maximale Anzahl verarbeitbarer Knoten an. Das Layout ist abgelegt in der global erreichbaren Datenstruktur %nodes%.

Die Datenstruktur %nodes% repräsentiert die Knoten des berechneten DAGs. Diese sind mit der Zugriffsfunktion `svref` über einen Index erreichbar. Die Anzahl der gespeicherten Knoten (also der maximale vergebene Index in der **%nodes%**-Struktur) ist der Wert der globalen Variablen **%node-index%**. Die einzelnen Knoten sind mit Hilfe des Makros `defvector` definierte Strukturen mit folgenden Slots:

- **pred** – Liste mit Indizes der Vorgängerknoten (Indizes in der **%nodes%**-Struktur)
- **succ** – Liste mit Indizes der Nachfolgerknoten
- **name** – Name des Knotens (normalerweise ein Zeiger auf den betrachteten Knoten der QDS)
- **index** – Index des aktuellen Knotens (in der **%nodes%**-Struktur)
- **strings** – Liste von Strings, die in der Box angezeigt werden sollen (Ergebnis der Anwendung der `info`-Funktion auf den aktuellen Knoten der QDS)
- **x** – x-Koordinate der linken oberen Ecke der Box
- **y** – y-Koordinate der linken oberen Ecke der Box
- **xe** – x-Koordinate der rechten unteren Ecke der Box
- **ye** – y-Koordinate der rechten unteren Ecke der Box

[7] Vorsicht: nach meinen Erfahrungen funktioniert die Tiefenbeschränkung nicht ohne Fehler. Als Abhilfe kann man versuchen, eine Tiefenbeschränkung in die `child`-Funktion auf der Quell-Datenstruktur einzubauen.

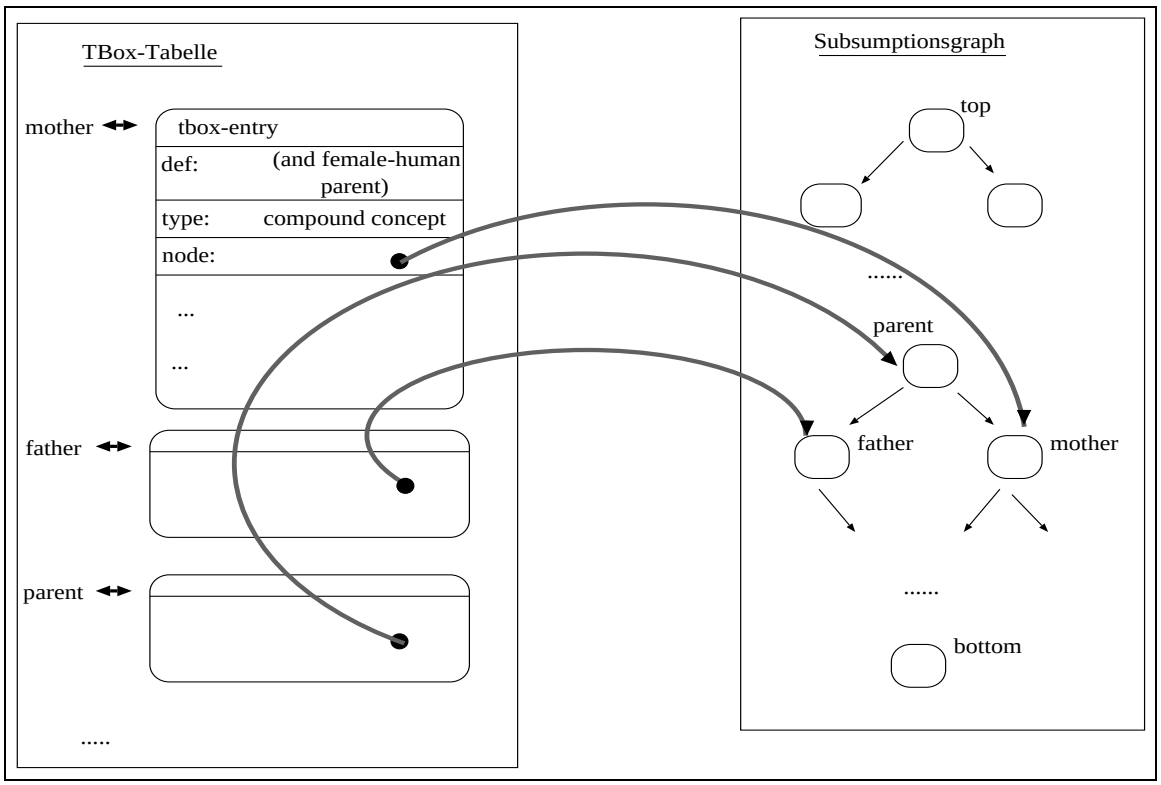

Abbildung 14: Die relevanten Datenstrukturen im Front-End von TAXON

Hat man einen Knoten der `%nodes%`-Struktur, so kann man auf die einzelnen Komponenten mit Funktionen `%node-pred, %node-succ, %node-name,` ... zugreifen.

Das vorgestellte Programm kann also zu gegebener Quell-Datenstruktur das Layout berechnen. Als nächstes betrachten wir TAXON, um zu sehen, wie wir überhaupt zu einer Quell-Datenstruktur kommen (exakter zu Funktionen `childfn, info`, von allem anderen können wir abstrahieren). Ferner ist zu klären, wie die an den Knoten angehefteten zusätzlichen Anfragemöglichkeiten realisierbar sind.

3.3.2 Die Schnittstelle zu TAXON

Das TAXON-System besteht aus mehreren Packages. Alle für unseren Browser benötigten Anfragefunktionen befinden sich im Package `front-end`, das die Funktionen für Benutzerein-/ausgabe und die Datenstrukturen für Kommunikation von und nach außen enthält.

Wir betrachten zwei Datenstrukturen: die **TBox**, die das Wissen über Konzeptdefinitionen enthält, und die Subsumptionshierarchie, die die aus den Konzeptdefinitionen berechneten Inklusionsbeziehungen repräsentiert (s. Abb. 14).[8]

Die TBox-Tabelle assoziiert einem Konzeptnamen einen TBox-Eintrag mit mehreren Slots, wie z.B.:

[8] Der Vollständigkeit halber sei noch die Tabelle `*separate-classes*` erwähnt, die die Klassen disjunkter primitiver Konzepte enthält. Sie wird jedoch für das weitere Verständnis nicht benötigt.

- **type** – gibt an, ob ein primitives, ein zusammengesetztes Konzept, eine offene Familie, ... vorliegt.
- **hierarchy-node** – ein Pointer auf den entsprechenden Knoten im Subsumptionsgraphen.
- **definition** – die vom Benutzer eingegebene Konzeptdefinition.

Den einem Konzept zugeordneten Eintrag erhält man mit der Funktion `get-tbox-entry`, die jeweiligen Slots über Strukt-Selekoren `tbox-entry-type, tbox-entry-hierarchy-node, ...`. Für unseren Browser sind solche Durchgriffe allerdings gar nicht notwendig, weil die benötigten Informationen zu gegebenem Konzeptnamen auch direkt abgefragt werden können. Funktionen dafür werde ich weiter unten vorstellen, vorher noch einiges zum Zusammenhang zwischen TBox-Tabelle und Subsumptionshierarchie:

Jeder TBox-Eintrag enthält auch einen Verweis auf den diesem Konzept zugeordneten Knoten der Subsumptionshierarchie. Dieser Knoten besitzt einen Slot `value` mit einer Liste der an dieser Stelle in der Hierarchie angesiedelten Konzepte; eine Liste deshalb, weil ja verschieden definierte Konzepte trotzdem als semantisch äquivalent klassifiziert werden können. Mit diesen Konzeptnamen lassen sich natürlich wieder über die TBox-Tabelle die entsprechenden TBox-Einträge assoziieren. Grundsätzlich könnten wir jetzt von der Implementierung des Subsumptionsgraphen als Quell-Datenstruktur abstrahieren und nur noch Konzeptnamen betrachten, für die die weiter unten beschriebenen Funktionen ja automatisch bei Bedarf auf die Hierarchie zugreifen.

Die Funktion `info` für das DAG-Layout wäre dann einfach der Ausdruck `#'(lambda (x) x)` und die Funktion `childfn` der Ausdruck `#'(lambda (x) (filo x))`.[9]

Nun enthält die Subsumptionshierarchie aber u.U. Konzepte, die für die Ausgabe gar nicht relevant sind. Das kann das `BOTTOM`-Konzept sein, das die Ausgabe unübersichtlich macht, ohne einen zusätzlichen Informationsgewinn zu bringen, das können vom System selbst aus technischen bzw. Effizienzgründen erzeugte Konzeptnamen sein; das können auch Konzepte sein, die für den sauberen strukturierten Aufbau der Wissensbasis notwendig waren, aber zur Laufzeit des Systems nicht von Interesse sind. Das TAXON-System sieht nun einige Möglichkeiten vor, solche Konzepte bei der Ausgabe auszublenden. Diese werden hauptsächlich über das `switch`-Kommando gesteuert (siehe dazu [1], für eine beispielhafte Verwendung dieser Möglichkeit auch [2]).

Solche Ausgabefilter für Konzepte sollten auch für den Taxonomie-Browser zur Verfügung stehen. Sie sind aber in der momentanen Taxon-Implementierung nur durch Funktionen realisiert, die direkt auf den Knoten des Subsumptionsgraphen arbeiten. Daher wurde bei der Implementierung des Browsers an einigen Stellen ein etwas umständliches Hin- und Herspringen zwischen Konzeptnamen und entsprechenden Hierarchieknoten notwendig.

So arbeiten die Funktionen zum Traversieren des Graphen und zum Anzeigen von Information mit Hierarchieknoten als Eingabe. Deshalb wird als Wurzelknoten zum Einstieg in die QDS der Wert des Funktionsaufrufes `(graph-top *SUBSUMPTION-GRAPH*)` benötigt, der die Wurzel des Subsumptionsgraphen liefert.

Die notwendigen Zugriffsfunktionen auf die QDS sehen dann folgendermaßen aus: `childfn` ist `#'(lambda (x) (im-lowers-passing-draw-test x))` und `info` ist `#'(lambda (x) (car (draw-info x)))`. Dabei findet die Funktion `im-lowers-passing-draw-test` alle direkten Nachfolgerknoten eines Hierarchieknotens unter Beachtung evt. gesetzter Filteroptionen, während

[9] `filo` bestimmt zu einem Konzept die Menge der direkten Unterkonzepte. Die Funktion wird weiter unten noch einmal erwähnt.

`draw-info` ein Selektor für die Liste aller dem Knoten assoziierten Konzeptnamen ist. Die Funktion `im-lowers-passing-draw-test` wird gesteuert von einer im TAXON-Front-End globalen Variablen `*draw-test*`, deren Wert eine einstellige Funktion ist, die zu einem Konzeptnamen entscheidet, ob das Konzept in der Hierarchiedarstellung auftauchen soll. Dieser Wert wird im Front-End-File `draw-hierarchy.lisp` gesetzt und hängt insbesondere von den mit dem `switch`-Kommando gesetzten Filteroptionen ab.

Mit Hilfe des Selektors `node-depth` für die Tiefe eines Knotens in der Subsumptionshierarchie, läßt sich eine beschränkte Nachfolgerfunktion[10] für einen Wurzelknoten `hierarchynode` z.B. folgendermaßen realisieren:

```
#'(lambda (x) (remove-if-not #'(lambda (x) (>= 3
                                               (- (node-depth x)
                                                  (node-depth hierarchynode)
                                               )
                                            )
                                  )
                                (im-lowers-passing-draw-test x)
                )
)
```

Wir beschreiben jetzt die oben schon angesprochenen Funktionen zur Informationsgewinnung für einzelne Konzepte. Bis auf die erste Funktion, die für das Wechselspiel von Konzeptname und Hierarchieknoten benötigt wird, dienen alle anderen der Bereitstellung von Informationen, die zu einem Konzept über das assoziierte Menü abgefragt werden kann. [11]

- **get-node-of-item** – bestimmt den Hierarchieknoten zu einem Konzept.
- **get-type-of-item** – bestimmt den Typ des Konzepts.
- **get-definition** – gibt die Definition eines Konzeptes zurück.
- **get-family** – gibt zu einem Mitglied einer Familie disjunkter primitiver Konzepte die Familie an.
- **get-family-type-of-item** – bestimmt zu einem Mitglied einer Familie disjunkter primitiver Konzepte den Typ der Familie (offen, abgeschlossen oder Subfamilie).
- **get-disjoint-concepts** – bestimmt die Menge aller anderen disjunkten Primitiven zu einem Mitglied einer Familie disjunkter primitiver Konzepte.
- **get-separation-class** – gibt zu einem Bezeichner für eine Familie disjunkter primitiver Konzepte alle Mitglieder der Familie an.

Mit Hilfe dieser Zugriffsfunktionen wurden folgende Funktionen zur Informationsdarstellung im TAXON-Browser implementiert:[12]

[10] Wie wir sie benötigen, um bei der Ausschnittvergrößerung nur einen begrenzten Teil der Taxonomie darszustellen, ohne auf die weiter oben angesprochene fehlerhafte Tiefenbeschränkung der Layout-Routinen angewiesen zu sein.

[11] Einige dieser Funktionen wurden gegenüber der bisherigen TAXON-Implementierung leicht abgeändert.

[12] Es fehlt noch die Realisierung der Auschnittdarstellung. Diese wird in Abschnitt 3.3.4 angesprochen.

- **show-equis-of-conc** – für die Menge aller äquivalenten Konzepte
- **show-imlo-of-conc** – für die Menge der direkt subsumierten Konzepte
- **show-iup-of-conc** – für die Menge der direkt subsumierenden Konzepte
- **show-lo-of-conc** – für die Menge aller subsumierten Konzepte
- **show-up-of-conc** – für die Menge aller subsumierenden Konzepte
- **show-def-of-conc** – für die Anzeige der Definition eines Konzepts

Alle Funktionen erhalten als Eingabe einen Window-Path und einen Hierarchieknoten. Sie fragen zuerst die relevante Information mit Hilfe der oben beschriebenen TAXON-Funktionen ab: `fequi, filo, fiup, flo, fup, get-definition`. All diese Funktionen erwarten einen Konzeptnamen als Eingabe. Diesen kann man über den `value`-Slot des betreffenden Knotens erhalten.

Komplikationen treten bei der Anzeige von Definitionen zusammengesetzter Konzepte auf, die als Lisp-Listen vorliegen, welche ohne Pretty-Printing recht unleserlich sind. Deshalb wurde die Funktion `ppconceptterm` implementiert, die zu einer Konzeptdefinition einen Tcl-String mit entsprechenden Steuerzeichen für eine übersichtlichere Ausgabe erzeugt (siehe dazu auch Abb. 11).

Die Funktionen rufen alle die Tcl/Tk-Prozedur `mkinfobox` zur Erzeugung eines Toplevel-Fensters für die Anzeige der entsprechenden Information. Zur Minimierung der Kommunikation werden nur die für den aktuellen Aufruf spezifischen Daten (z.B. Definition eines Konzeptes, Liste der Subsumierenden) als Argumente übergeben, der Typ der Informationsdarstellung (Anzeige einer Definition oder einer Äquivalenzklasse, Definition eines primitiven oder eines zusammengesetzten Konzeptes usw.) wird über einen Nummerncode mitgeteilt. Die genaue Art der Darstellung, insbesondere die für einen bestimmten Darstellungstyp invarianten Textteile, sind in der Implementierung der Tcl/Tk-Prozedur festgelegt.

3.3.3 Der Aufbau der Benutzungsoberfläche

Abbildung 15 zeigt die wesentlichen Schritte zum Aufbau der Benutzungsoberfläche. Ich werde die beteiligten Funktionen kurz umreißen.

Die Funktion make-taxonomy-browser ist die Schnittstelle der Browser-Implementation nach außen; sie baut alle in Unterkapitel 3.2 dargestellten Strukturen auf.

Dazu ruft sie zuerst eine Tcl/Tk-Prozedur `make_taxonomy_browser_toplevel` zum Aufbau der benötigten Fenster. Dann ruft sie zweimal die Funktion `dag` zum Zeichnen der Taxonomiebäume (normalgroß und schematisch) in den entsprechenden Fenstern auf. Als Argumente erhält `dag` die oben angegebenen `children-` und `info-`Funktionen, den Top-Knoten der Subsumptionshierarchie, die Namen der Fenster für die jeweilige Darstellung und die Angabe, ob die Taxonomie groß oder schematisch dargestellt werden soll.

Die Funktion dag gibt ihre Argumente pflichtgemäß an den Tk-Handler (den Default-Fall) weiter.

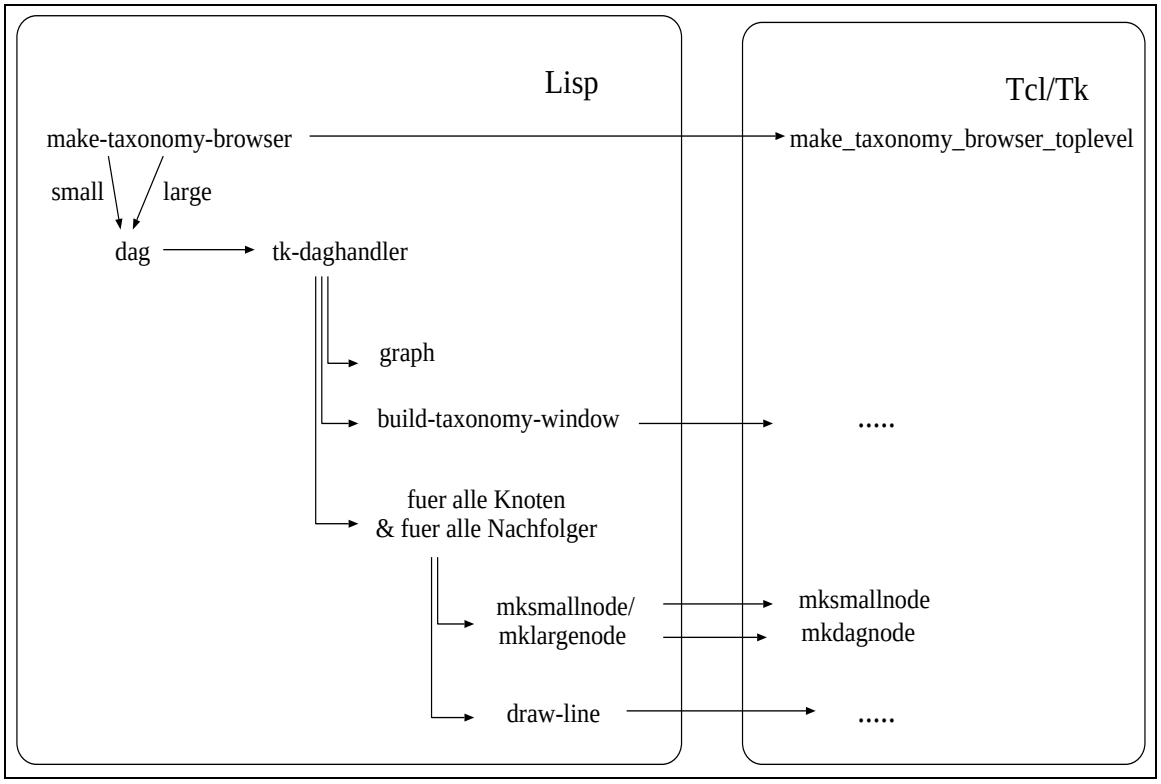

Abbildung 15: Aufrufmuster beim Aufbau der Benutzungsoberfläche

Der Tk-Handler wurde mit `defdag` erzeugt. Beim Aufruf von `defdag` hat das Argument `additional-args` den Wert `(parentwindow small-or-large-graph)`, weil diese beiden Parameter beim Tk-Handler zusätzlich zu den vorgegebenen Parametern zur Verfügung stehen sollen (siehe oben: die entsprechenden Werte werden von `make-taxonomy-browser` als Schlüsselwortargumente an `dag` übergeben).

Der Handler ruft zuerst die `graph`-Routine (mit für Groß- und Schemadarstellung unterschiedlichen Werten für die verschiedenen Größenparameter). Die Funktion `build_taxonomy_window` erzeugt in den vorher von `make_taxonomy_browser_toplevel` generierten Fenstern jeweils ein `Canvas`-Widget mit Scrollbars für die Anzeige der Taxonomie in der sich aus den Layout-Daten ergebenden Größe. Dann wird das in `%nodes%` abgelegte Layout folgendermaßen verarbeitet:

Für alle Knoten in `%nodes%`: lasse einen Knoten in der Taxonomiedarstellung anlegen. Dazu werden Tcl/Tk-Prozeduren `mksmallnode` und `mkdagnode` benutzt, die die jeweiligen Koordinaten und Label-Texte als Argument erhalten, sowie die Funktionsaufrufe, die an die Knoten als Menüpunkte angeheftet werden, um konzeptbezogene Informationen anzuzeigen. Für alle Nachfolgerknoten: zeichne einen Pfeil zu seiner Position. Dafür werden die Koordinaten des Nachfolgers abgefragt, daraus Pfeilkoordinaten bestimmt und mit diesen der entsprechende Tk-Befehl zum Zeichnen eines Pfeils aufgerufen.

Die Umsetzung in Tcl/Tk geschieht einerseits direkt durch Graphik-Befehle über das `Wish`-Kommando in Lisp, andererseits durch Aufruf von Tcl/Tk-Prozeduren, die jeweils mehrere Schritte zusammenfassen. Sie sind häufig gleichbenannten Lisp-Funktionen zugeordnet.

In Tcl/Tk entsteht folgender Aufbau: an einem Menubutton werden als Optionen die Befehle

angehängt, die Toplevel-Windows für Groß- und Schemadarstellung deiconifizieren (das macht die Prozedur `make_taxonomy_browser_toplevel`). In diesen Windows gibt es als zentrale Struktur jeweils ein Canvas, auf dem die Taxonomie angezeigt wird. Die Knoten sind bei der Schemadarstellung einfache `Rectangular`-Items, bei der Großdarstellung `Window`-Items, in denen sich Menubuttons befinden. Sie werden durch die Prozeduren `mksmallnode` bzw. `mkdagnode` erzeugt. Die Verwendung solcher Tcl/Tk-Prozeduren, die jeweils 10-20 einzelne Graphik-Befehle durchführen, hatte enorme Auswirkungen auf die Performanz des Gesamtsystems, weil sowohl die Anzahl der zu übertragenden Befehlszeilen als auch das zu übertragende Gesamtdatenvolumen drastisch reduziert wird (viele Befehle werden ja mit weitgehend gleichen Argumenten aufgerufen, so daß eine einmalige Übertragung ausreicht).

Die Prozedur `mkdagnode` plaziert als Knoten für ein Konzept direkt einen Menubutton, `mksmallnode` dagegen nur maussensitive Rechtecke, bei denen auf Wunsch ein Menü erzeugt und aufgeblendet werden kann.

Als technisches Detail bei `mkdagnode` ist noch zu erwähnen, daß die Layout-Berechnung bei der Bestimmung der Box-Breite (in Großdarstellung) nur einen Mittelwert für die Buchstabenbreite der verwendeten Schrift verwendet. Daher können suboptimale Plazierungen zustandekommen, wenn Konzeptnamen ungewöhnlich schmal oder breit werden. Als Abhilfe wurde hier eine nichtproportionale Schrift verwendet, bei der alle Buchstaben die gleiche Breite einnehmen. Steht keine solche Schrift zur Verfügung, kann man z.B. eine maximale Breitenangabe machen und den Text dann im (evt. zu großen) Label zentriert ausgeben.

Nach Aufbau der graphischen Oberfläche wartet das System auf eine Benutzeraktion, die dann weitere Reaktionen hervorruft. Diese werden im folgenden skizziert.

3.3.4 Die Kommunikation zur Laufzeit

Abbildung 16 zeigt die Aufrufstrukturen zur Beantwortung von Benutzerinteraktionen.

Für die Gestaltung der Vorgänge beim Aufbau und zur Laufzeit der graphischen Benutzungsoberfläche wurden zwei Prinzipien beherzigt:

1. Es sollte möglichst viel Kommunikation in Tcl/Tk intern bleiben, um die Verbindung zu Lisp als potentiellen Flaschenhals möglichst wenig zu belasten.

2. Der Aufbau der Oberfläche sollte nur die unbedingt notwendigen Teile umfassen. Insbesondere sollten erst auf Aufforderung des Benutzers sichtbare Teile möglichst auch erst aufgrund dieser Aufforderung erzeugt werden. Andernfalls kann der Aufbau des Taxonomie-Browsers (unnötig) aufwendig werden und das Laufzeitsystem sehr groß.

Als Beispiel für die Verfolgung beider Prinzipien sei die Behandlung von Konzept-Boxen in der Schemadarstellung genannt. Hier wird für ein Konzept ein ganz einfaches `Rectangular`-Item verwendet. An die Ereignisse *Mouse betritt Item* bzw. *Mouse verläßt Item* werden Änderungen der Füllungsfarbe zum Anzeigen des aktuellen Konzepts gebunden. An das Ereignis *Mouse-Click über Rechteck* wird eine Befehlsfolge gebunden, um: (i) einen Menubutton für das betreffende Konzept zu erzeugen (ii) die für ein Konzept erlaubten Befehle in das assoziierte Menü einzutragen (iii) ein `Window`-Item im Canvas zu erzeugen, in dem der Menubutton gezeichnet wird.

Dieses Vorgehen benötigt für die beschriebenen Aktionen keinen Zugriff auf Lisp, weil ja keine zusätzlichen Informationen notwendig sind, die nicht bei der Anlage des Knotens schon da

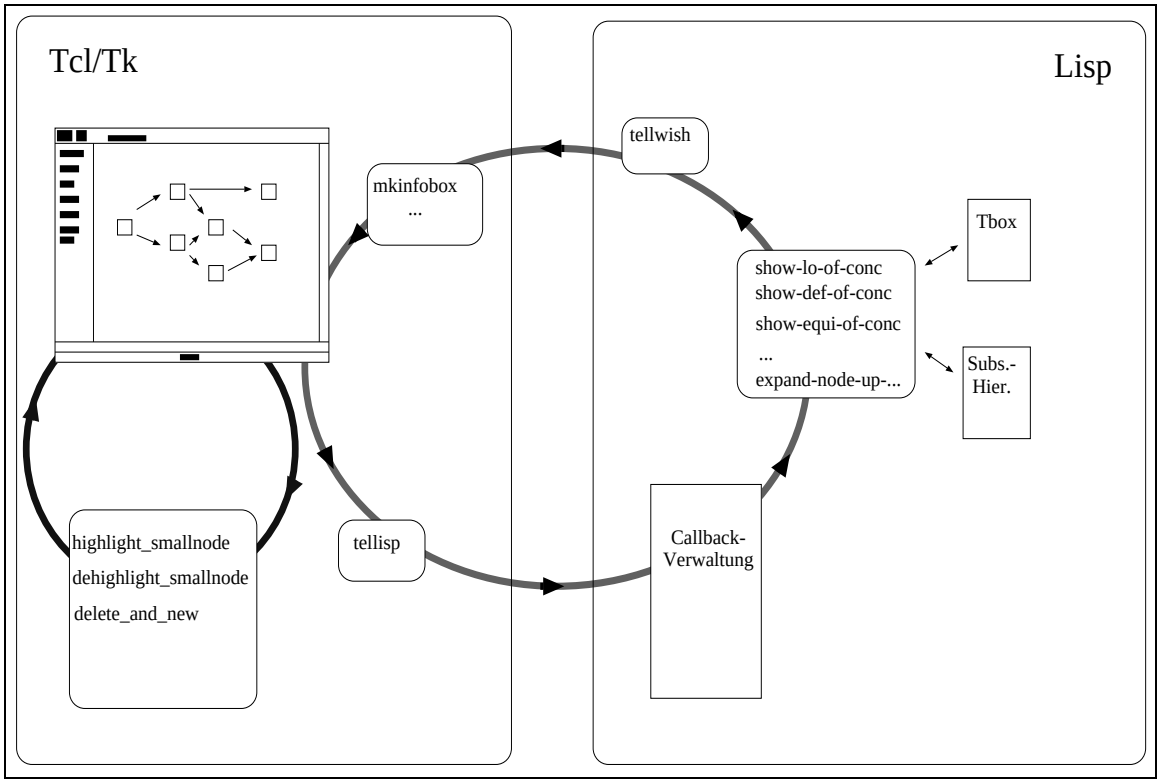

Abbildung 16: Kommunikation zur Laufzeit des Browsers

wären. Dadurch werden auch nicht Unmengen von Menubuttons und `Window`-Items erzeugt, die bei Bedarf genauso gut und schnell kurzfristig aufgebaut werden können.

Es besteht allerdings auch ein gewisser Zielkonflikt zwischen den beiden Anforderungen. Für sehr große Taxonomien wäre es z.B. vorstellbar, daß das Anhängen komplexer Befehlsfolgen an Items das Tcl/Tk-Laufzeitsystem sehr belastet (bzw. auch schon beim Aufbau das Herüberschieben der Menübefehle für die einzelnen Konzepte sehr aufwendig ist). Dann könnte es sinnvoll sein, an den Knoten nur den Konzeptnamen als Information zu halten und alle weiteren Befehle bei Bedarf aus Lisp abzurufen. Dann wären ein schnellerer Aufbau und ein „leichteres“ Laufzeitsystem mit mehr Kommunikation zwischen den Teilen zur Laufzeit bezahlt.

Jedenfalls sollten diese beiden Dimensionen der Optimierung klar sein. Die Entscheidung, wo welche Information abgelegt ist und wann man sie wie abruft, kann die Performanz beträchtlich beeinflussen.

Wir haben somit zur Laufzeit zwei Informationsflüsse:

Eine **Tcl/Tk-interne Schleife**. Diese verarbeitet ohne Rückgriff auf das Lisp-System das Markieren der aktuellen Knoten in der Schemataxonomie, das Aufbauen der Menubuttons zu Schemaknoten und die Interaktion zwischen Listbox und Schemaknoten (Aufzeigen des aktuell selektierten Konzepts in der alphabetischen Auflistung).

Einen **Kommunikationsfluß zwischen Tcl/Tk und Lisp**. Dieser betrifft das Aufzeigen von Informationsboxen zu Konzepten und die Ausschnittvergrößerungen.

Als Befehl an Knoten oder Ereignissen wird hier jeweils die Anweisung gegeben, über `tellisp`

eine Botschaft an Lisp zu übermitteln. Lisp wartet (zumindest in der in Unterkapitel 2.1.1 beschriebenen Kopplung auf der Basis von Lucid Lisp) in einer Endlosschleife auf Anweisungen von Tcl/Tk und assoziiert über die Callback-Tabelle die entsprechenden Lisp-Ausdrücke.

Diese rufen Informationsfunktionen wie z.B. `show-def-of-conc` (siehe 3.3.2) oder die Funktion `expand-node-up-to-depth-3` zum Anzeigen einer Ausschnittvergrößerung.

Funktionen wie `show-def-of-conc` fragen die benötigten Informationen von der TBox bzw. dem Subsumptionsgraphen ab und rufen dann die Tcl/Tk-Prozedur `mkinfobox`, die die Information in der gewünschten Art und Weise in einem Toplevel-Window darstellt.

`expand-node-up-to-depth-3` erzeugt ein neues Toplevel-Window und ruft die Funktion `dag` zum Zeichnen eines Graphen in diesem Fenster, mit dem dem aktuellen Konzept zugeordneten Hierarchieknoten als QDS-Einstieg und der in 3.3.2 beschriebenen tiefenbeschränkten `children`-Funktion.

Durch den Aufruf der jeweiligen Tcl/Tk-Prozeduren geht die Kontrolle (in der genannten Kopplung) wieder an Tcl/Tk über und Lisp wartet auf die nächste Anweisung auf dem Kommunikationskanal.

4 Zusammenfassung, Bewertung, Ausblick

Ausgehend von der Zielsetzung, eine sinnvolle Programmierumgebung für graphische Oberflächen im Rahmen des VEGA-Projektes zu finden bzw. erstellen, habe ich begründet, warum Tcl/Tk ein geeignetes Werkzeug für die gestellten Aufgaben darstellt.

Ich habe die Kopplungsproblematik zwischen Lisp und Tcl/Tk eingehend erläutert und eine minimale Schnittstelle vorgestellt. Mit Hilfe dieser Schnittstelle wurde ein Browser für Begriffshierarchien in TAXON erstellt, der inzwischen auch in ein Expertensystem zur Konservierung des Wissens zum Design von Kurbelwellen integriert wurde [8].

Nach meinem persönlichen Eindruck bei der Implementierung des Browsers, der auch von anderen Entwicklern bestätigt wurde, die sowohl Tcl/Tk als auch andere (teils kommerzielle) GUI-Tools für Lisp kennen, zeichnet sich Tcl/Tk mindestens durch die folgenden Punkte aus:

- Leichte Bedienbarkeit.
- Einfache Erlernbarkeit aufgrund sehr guter Dokumentation.
- Gute Performanz.
- Robuste, stabile Implementierung.
- Public-Domain Software.

Gerade die beiden ersten Punkte, für jeden Einsteiger essentiell, sind bemerkenswert. Zur Einarbeitung stehen zur Verfügung: die Manual-Pages, über anonymes ftp erhältliche Referenzen, die ebenfalls durch ftp erhältliche *Widget-Tour* von Andrew Payne[13], und anderes. Für eine detaillierte Auflistung siehe Anhang B. Ferner existiert ein lebhafter Austausch auf kurzlebigeren elektronischen Medien, insbesondere der Newsgroup `comp.lang.tcl`.

Die Entscheidung für einen minimalen Überbau in Lisp hat sich ebenfalls nicht gerächt. Der einfache Kopplungsmechanismus hat zum einen die schnelle Implementierung des Browsers als Test für die Brauchbarkeit von Tcl/Tk ermöglicht und gewährleistet zum anderen Portabilität, leichte Änderbarkeit und einfache Bedienung. Ein wesentlich umfangreicherer Lisp-Überbau erscheint mir nicht sinnvoll. Hier hätte allerdings die Erfahrung mit größeren, komplexeren graphischen Programmieraufgaben zu zeigen, ob nicht ausgefeiltere Konzepte, wie z.B. ein objektorientierter Überbau von Vorteil wären. Dabei stellt sich jedoch auch die Frage, ob solche komplexen Graphik-Anwendungen überhaupt als „typische“ Aufgabenstellungen im betrachteten Umfeld vorkommen, und ob nicht einfache Benutzungsoberflächen mit etwas Interaktion der Standardfall sind.

Die Frage nach einem „optimalen“ Kommunikationskonzept erscheint mir nicht abschließend geklärt. Die in 2.1.2 angesprochene Lösung wirkt allerdings schon recht brauchbar. Möglicherweise ist hier auch keine allgemeingültige Antwort zu geben, weil sie sowohl von Systemgegebenheiten wie auch vom intendierten Zielsystem abhängt. Auch hierzu sollten die oben angesprochenen elektronischen Medien beachtet werden. Beispielsweise haben sich zu Beginn unserer Beschäftigung mit dem Thema keine Lösungen für die Kopplungsproblematik gefunden, während inzwischen schon sehr ausgefeilte Kommunikationsprotokolle angeboten werden. Dabei stellt sich jedoch immer die Frage der möglichst großen Portabilität, weil komplexe Lösungen häufig spezielle nichtstandardisierte Implementierungsdetails ausnutzen.

[13] DEC Cambridge Research Lab, `payne@crl.dec.com`

Eine sehr interessante Lösung wird in [6] beschrieben. Dort ersetzen die Autoren Tcl als Kommandosprache für das Graphik-Toolkit durch Scheme [21]. Dieser Ansatz, nicht zwei isolierte Systeme kommunizieren zu lassen, sondern eine tiefe Integration durchzuführen, hat Vorteile z.B. bei der Performanz. Allerdings ist so die gewünschte Portabilität und Unabhängigkeit vom Lisp-Dialekt mit Sicherheit nicht zu erreichen.

Weitere Arbeiten sehe ich im wesentlichen in zwei Bereichen:

Ausbau einer Entwicklungsumgebung. Es bietet sich an, unter Ausnutzung der Möglichkeiten von Tcl/Tk und Lisp und eines geeigneten Kommunikationskonzepts eine graphisch orientierte Programmierumgebung für Benutzungsoberflächen zu gestalten.

Als erste Schritte zählen dazu ein Editor, die Möglichkeit, die Fehlermeldungen und Debugger beider Systeme zu lesen und nutzen, und die Möglichkeit, wirklich interpretativ im Rapid-Prototyping-Stil zu arbeiten, d.h. auch an der Implementierung arbeiten zu können, ohne daß der Test das Editieren blockiert.

In der momentanen Lucid-Kopplung wird z.B. durch das Laufen der Tk-Oberfläche Lisp blockiert, weil die Read-Eval-Print-Loop für Botschaften aus Tcl/Tk eine Endlosschleife auf Lisp-Toplevel darstellt, deren Unterbrechung (z.B. um einen Programmfehler zu beheben) auch den Wish-Interpreter beendet. Hier ist z.B. die Kopplungsstrategie aus 2.1.2 schon intelligenter. Allerdings kann bei komplexeren Kommunikationsmustern dann leicht wieder das Problem mangelnder Portabilität auftreten.

In einer ersten Ausbaustufe könnten auch Hilfestellungen zur Syntax und Semantik integriert sein, wie die leichte Navigation im Online-Manual, Boxen zur Anzeige möglicher Optionen für einen gegebenen Befehl (oder auch graphisch interaktive Auswahl: Wahl der Rahmen oder Schriftspezifikation durch Anklicken eines Beispielobjekts) oder Übersichten aktuell im System angelegter Objekte oder Tags usw.

Eine spätere Ausbaustufe könnte das interaktive, graphisch orientierte Anlegen von Benutzungsoberflächen unterstützen, wie es mit dem Interface Builder von InterViews [9] möglich ist. Inzwischen gibt es auch schon einen interaktiven Interface Builder XF [5] für Tcl/Tk, wobei allerdings noch nicht geklärt ist, was die Verwendung eines integrierten Systems wie des vorgestellten für Auswirkungen auf das Arbeiten mit einem solchen Tool hätte.

Aufbau einer kompletten Benutzungsoberfläche für TAXON. Der vorgestellte Browser stellt ja nur eine Visualisierung bereits in der TBox fertig vorhandener Information dar. Wesentlich interessanter ist es, den gesamten interaktiven Aufbau einer Wissensbasis graphisch zu unterstützen.

Dazu wäre ein Editor für Konzeptdefinitionen zu schreiben, der in die TAXON-Wissensbasis einträgt. Anfragemöglichkeiten oder Auswahlmöglichkeiten (z.B. Filteroptionen) wären über Menüauswahlen zu steuern, Hilfestellungen könnten zur Verfügung gestellt werden (wie oben: Online-Manual u.ä.). Die graphische Darstellung zur Taxonomie könnte sowohl zur ständigen Plausibilitätskontrolle des Editierens (Wird ein Konzept an einer für sinnvoll erachteten Stelle in die Taxonomie gefügt?) als auch als Teilfunktion des Editierens dienen (Z.B.: Definition einer Konzeptkonjunktion nicht nur rein textuell möglich, sondern auch durch Anklicken der Konzepte, deren Konjunktion gebildet werden soll.).

Die Browserfunktionalität könnte erweitert werden, z.B. könnte zusätzlich zur textuellen Darstellung von subsumierenden oder subsumierten Konzepten auch ein farbliches Absetzen der

anzuzeigenden Konzepte in der Taxonomie erfolgen. Auch die Anzeige aller disjunkten Konzepte könnte ein nützlicher Dienst sein. Ferner wäre die Verwendung weiterer, eventuell anwendungsabhängiger Filtermöglichkeiten zu überlegen (Z.B.: Zeige alle Konzepte an, die irgendeine anwendungsspezifische Eigenschaft erfüllen.). Schließlich müßte die Performanz des Browsers anhand eines großen Beispiels getestet werden.

Eine interessante Frage, die bei einem interaktiven Editieren der visualisierten Taxonomie auftaucht, ist die nach einem geschickten Verarbeiten der dabei ständig lokal sich ändernden Darstellung. Ein häufiges lokales inkrementelles *Redrawing* der Taxonomie ist mit der hier vorgestellten Browser-Implementierung natürlich nicht vernünftig zu realisieren, weil für jede Änderung der DAG komplett neu durchgerechnet und gezeichnet werden müßte.

Ebenfalls offen bleibt die Frage nach einer Visualisierung der TAXON-ABox. Ein Ansatz dazu wurde unter CLIM schon für die Ergebnisdarstellung des deklarativen Arbeitsplanungsssystems DECPLAN im ARC-TEC Projekt implementiert. Die Darstellung der ABox kann gerade für die Veranschaulichung der dynamischen Abläufe bei Inferenzprozessen oder für die Anzeige komplexer in der ABox gespeicherter Antworten auf Inferenzprozesse nützlich sein. Hier ist auch zu klären, wie weit man terminologisches und assertionales Wissen in einer einzigen Darstellung sinnvoll und übersichtlich integrieren kann (und will).

Literatur

[1] A. Abecker and Ph. Hanschke. TAXON: The Terminological Subsystem of COLAB. System Description and User Manual, March 1993.

[2] H. Boley, Ph. Hanschke, M. Harm, K. Hinkelmann, Th. Labisch, M. Meyer, J. Müller, Th. Oltzen, M. Sintek, W. Stein, and F. Steinle. *μCAD2NC: A Declarative Lathe-Workplanning Model Transforming CAD-like Geometries into Abstract NC Programs.* DFKI Document D-91-15, November 1991.

[3] H. Boley, Ph. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: a hybrid compilation laboratory. In *3rd International Workshop on Data, Expert Knowledge and Decisions*, September 1991. To appear in a special issue of 'Annals of Operations Research'. Also as DFKI RR-93-03.

[4] H. Boley, Ph. Hanschke, K. Hinkelmann, M. Meyer, and M.M. Richter. VEGA – Knowledge Validation and Exploration by Global Analysis. Project Proposal, DFKI Kaiserslautern, October 1992.

[5] S. Delmas. XF – Design and Implementation of a Programming Environment for Interactive Construction of Graphical User Interfaces. Master's thesis, Technische Universität Berlin, März 1993.

[6] E. Gallesio. Embedding a Scheme Interpreter in the Tk Toolkit. In *Tcl 93 Workshop Proceedings*, 1993.

[7] Texas Instruments Inc. *CLX (Common Lisp X Interface) Programmer's Reference.* 1989.

[8] O. Kühn and B. Höfling. Conserving corporate knowledge for crankshaft design. Submitted for Publication, 1993.

[9] M.A. Linton, P.R. Calder, J.A. Interrante, S. Tang, and J.M. Vlissides. InterViews Reference Manual Version 3.0.1, 1991.

[10] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Van Der Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. GARNET – Comprehensive Support for graphical, highly-interactive User Interfaces. *IEEE Computer*, 11(23):71–85, November 1990.

[11] John K. Ousterhout. An Introduction To Writing Tcl Scripts. Vortrag, Folienkopien via FTP erhältlich.

[12] John K. Ousterhout. Building User Interfaces With Tcl and Tk. Vortrag, Folienkopien via FTP erhältlich.

[13] John K. Ousterhout. Tcl and Tk: A Programming System for X11 User Interfaces. Vortrag, Folienkopien via FTP erhältlich.

[14] John K. Ousterhout. Writing A New Widget Class Using C and Tk. Vortrag, Folienkopien via FTP erhältlich.

[15] John K. Ousterhout. Writing Tcl-Based Applications In C. Vortrag, Folienkopien via FTP erhältlich.

[16] John K. Ousterhout. Tcl: An Embeddable Command Language. In *Proceedings 1990 Winter USENIX Conference*, 1990.

[17] John K. Ousterhout. An X11 Toolkit based on the Tcl Language. In *Proceedings 1991 Winter USENIX Conference*, 1991.

[18] John K. Ousterhout. *An Introduction To Tcl and Tk.* Addison-Wesley, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1994. In Vorbereitung, Draft via FTP erhältlich, voraussichtliche ISBN 0-201-63337-X, voraussichtlicher Erscheinungstitel *Tcl and the Tk Toolkit* .

[19] R. Rao, W.M. York, and D. Doughty. A Guided Tour of the Common Lisp Interface Manager. *Lisp Pointers, ACM*, IV(1), January 1990.

[20] G. L. Steele. *Common Lisp: The Language. 2nd Edition.* Digital Press (Bedford MA), 1990.

[21] G.J. Sussman, A. Abelson, and J. Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill (New York), 1985.

[22] D. Theobald. Tcl/Tk in a Nutshell. Technical Report FZI Report 17/93, Forschungszentrum Informatik, Karlsruhe, Juli 1993.

A Verfügbarer Befehlsvorrat

Hier noch einmal eine kurze Übersicht der in der Kopplung verfügbaren Befehle. Diese werden vom Graphik-Package *Wish* exportiert.

Kommunikation mit Tcl/TK

wish, Makro, beliebig viele Argumente, übergibt diese als Kommandozeile an Tcl/Tk. Kopplungsunabhängig. Benutzt **tellwish**. Wertet Ausdrücke aus, überführt sie in Strings, führt Kleinschreibung ein (Ausnahme: als Strings gegebene Argumente).

Makros **{}, [], dq**, beliebig viele Argumente, werten Ausdrücke aus wie **wish**, erzeugen dann String mit durch Kommandonamen bezeichneter Klammerung.

expand-pathname, Funktion, nimmt zwei Strings oder string-wertige Variablen und erzeugt die Konkatenation mit eingefügtem Punkt.

init-interface, Funktion, 0 Argumente, baut Kommunikation auf. Implementierung kopplungsabhängig.

lispcommand, Makro, 1 Argument, bindet Lisp-Kommando an Tcl-Callback, erzeugt Tcl/Tk-Befehl zum Aktivieren des Callbacks. Kopplungsunabhängig. Benutzt **tellisp**.

start-loop, Funktion, 0 Argumente, startet Read-Eval-Print-Loop für Lesen und Ausführen der ankommenden Tcl/Tk-Callbacks. Kopplungsabhängig.

Abfragefunktionen {**wvv**|**wcc**|**wcl**|**wec**}-{**string**|**number**|**bool**|**stringlist**|**numberlist**|**symbollist**}, beliebig viele Argumente für {wcc|wcl|wec}-Aufrufe. Übergabe als Tcl/Tk-Kommando, Lesen des Kommunikationskanals von Tcl/Tk, Umsetzung in bezeichneten Lisp-Datentyp, im wcl-Falle vorher Abschneiden des letzten Tokens. Für wvv-Fall 1 Argument, Variablenbezeichner, veranlaßt Schreiben des Variablenwertes auf `tellisp`-Kanal, liest Wert, wandelt um in bezeichnetes Format. Kopplungsabhängig.

Definitionsmakros für Widgets **make-{button|canvas|checkbutton| entry|frame|label| listbox|menu|menubutton|message|radiobutton|scrollbar|text|toplevel}** nehmen einen Pfadnamen und evt. Konfigurationsoptionen für die Widget-Erzeugung als Argumente, liefern Widgetnamen als String zurück, erzeugen Widget in Tcl/Tk. Kopplungsunabhängig.

B Informationsquellen zu Tcl/Tk

Mir sind folgende ftp-Server für Material mit Tcl/Tk-Bezug bekannt:

sprite.berkeley.edu (Internet 128.32.150.27)
Verzeichnis: `/tcl/`.

harbor.ecn.purdue.edu
Verzeichnis: `/pub/tcl/`.

syd.dit.csiro.au
Verzeichnisse: `/pub/tk/contrib/` und `/pub/tk/sprite/`.

ftp.ibp.fr (Internet 132.227.60.2)
Verzeichnisse: `/pub/tcl/distrib/` und `/pub/tcl/contrib/` und `/pub/tcl/expect/`

Es folgt eine Auflistung der hier am Institut zur Zeit verfügbaren Informationsquellen zu Tcl/Tk, jeweils mit Titel, Beschreibung, Bezugsquelle und aktuellem Ablageort am DFKI Kaiserslautern.

Die FAQ-Liste der Tcl-Newsgroup

Beschreibung: FAQ-Liste (*frequently asked questions*) der Newsgroup `comp.lang.tcl`. Wichtigste Einstiegsreferenz, enthält – ständig aktualisiert – Einstiegsfragen, aktuelle Entwicklungen und Bezugsquellen.
Bezugsquelle: Aktuelle Version jeweils in der Newsgroup zu finden, letzte Fassung in den Tcl/Tk-Archiven, z.B. bei `harbor.ecn.purdue.edu` in `/pub/tcl/docs/tcl-faq.p0[1-5]`.
Ablage: Der erste von fünf Teilen der Fassung vom 19. Oktober 1993 liegt in `/home/taxon/TK-INFOS/FAQ/tcl-faq.part01`.

Kurzbeschreibung: Tcl/Tk in a Nutshell

Beschreibung: Stichwortartige Einführung [22] in Grundlagen von Tcl, Tk und XF.
Bezugsquelle: Über anonymes ftp vom Rechner `ftp.fzi.de`, dort im File `/pub/OBST/OBST3-3/psfiles/TclTk_notes.ps.Z`.
Ablage: In komprimierter Form unter `/home/taxon/TK-INFOS/TclTk_notes.ps.Z`.

Buch zu Tcl/Tk

Beschreibung: Detailliertes und gut verständliches Buch [18] zur Beschreibung des gesamten Sprachumfanges von Tcl/Tk von John Ousterhout. Liegt am DFKI mehrfach als Postscript vor. Umfaßt mit allen Kapiteln inzwischen circa 400 Seiten. Postscript-Version ist syntaktisch nicht mehr ganz auf der Höhe der aktuellen Tcl/Tk-Distribution, kann aber die meisten Konzepte gut veranschaulichen. Wer nur Tk-Oberflächen programmieren will, benötigt nur kleine Teile des gesamten Buches zum Einstieg in die Grundkonzeption. Danach ist ein Arbeiten anhand von Beispielen und mit Unterstützung der Manual Pages m.E. ausreichend. Das Buch soll im Frühjahr 1994 bei Addison-Wesley erscheinen.
Bezugsquelle: Über anonymes ftp von `sprite.berkeley.edu` in den Files `/tcl/book.p*.ps.Z`.

Ablage: In den Files `/home/taxon/TK-INFOS/TCL-Book/book.p*.ps`.

Die Widget-Tour

Beschreibung: Ein interaktiver tutorieller Tcl/Tk-Beispieldialog. Es werden die verschiedenen Befehlselemente von Tk eingeführt, indem zu einem Beispielfile das graphische Resultat angezeigt wird und bei interaktiver Veränderung des Files durch den Lernenden die resultierende Veränderung am graphischen Objekt durchgeführt wird. Sicherlich sehr empfehlenswert zur spielerischen Einarbeitung.
Aufruf: `/home/taxon/TK-INFOS/WTour/wtour`. Im Verzeichnis `/home/taxon/TK-INFOS/WTour/` liegen noch weitere Informationen zur Widget-Tour. Wer mit Tvtwm arbeitet, sollte das Programm nur starten, wenn er sich im linken oberen Teil des virtuellen Bildschirms befindet, weil nicht immer die korrekte Zusammenarbeit von Tcl/Tk mit Tvtwm gewährleistet ist.

Die Tk-Demos

Beschreibung: Über ein Menü abrufbare Demo-Programme für die verschiedenen Gestaltungselemente in Tk. Zeigen teilweise auch nichttriviale Anwendungen (z.B. scrollable canvas oder floorplan). Zum Lernen aus Beispielen können die Implementierungsfiles herangezogen werden. Auch hier Vorsicht: nicht verwirren lassen, viele Demo-Fenster erscheinen bei Benutzung des Tvtwm im Root-Bildschirm und nicht im aktuellen Teil.
Aufruf: `wish -f /home/taxon/TK-INFOS/Demos/widget`.
Im Verzeichnis `/home/taxon/TK-INFOS/` stehen auch die Implementierungsfiles.

Tcl-Referenzkarte

Beschreibung: Kurzreferenzkarte zu Tcl.
Bezugsquelle: Über anonymes ftp von `harbor.ecn.purdue.edu` im File `/pub/tcl/docs/QuickRef.tar.Z`.
Ablage: Im File `/home/taxon/TK-INFOS/tcl-ref.ps`.

Tk-Referenzkarte

Beschreibung: Eine 4-seitige Referenzkarte mit den Tk-Befehlen und ihren jeweiligen Optionen.
Bezugsquelle: Über anonymes ftp von `bohr.physics.upenn.edu` im File `/pub/tk/tkrefcard.tar.Z` oder `harbor.ecn.purdue.edu` in `/incoming/tkrefcard.tar.Z`.
Ablage: Im File `/home/taxon/TK-INFOS/TK-Refcard/tk.ps`.

Manual Pages

Beschreibung: Postscript-Versionen der Online-Dokumentation zu Tcl/Tk. Vollständige Beschreibungen von Syntax und Semantik der Tcl/Tk-Befehle. Für das Arbeiten mit dem Toolkit sehr nützlich.

Bezugsquelle: Über anonymes ftp von `harbor.ecn.purdue.edu` in `/pub/tcl/docs/`.
Ablage: Der Tk betreffende Teil befindet sich im Verzeichnis `/home/taxon/TK-INFOS/ManPages/`.

Tcl/Tk-Vorträge

Beschreibung: Die Folienkopien zu den Vorträgen [13; 11; 15; 17; 12; 14] von John Ousterhout. Die Beispielsyntax dürfte nicht mehr in allen Punkten auf der Höhe der aktuellen Tcl/Tk-Distribution sein. Gibt allerdings einen guten Überblick über die Grundkonzepte. Außerdem gibt es einführende Papiere [16; 17] über Tcl bzw. Tk von den USENIX Konferenzen 1990 und 1991.
Ablage: Im Verzeichnis `/home/taxon/TK-INFOS/TCL-Book/`.

Vertiefende Aufsätze zu Tcl/Tk

Es gibt weitere Konferenzbeiträge und Proceedings der Tcl-Konferenz 1993 – wie z.B. [6] – bei `harbor.ecn.purdue.edu`. Detaillierte Beschreibungen sind in der oben angegebenen FAQ zu finden. Teilweise sind diese Beiträge in `/home/taxon/TK-INFOS/TCL-Book/tcl93-proceedings/` zu finden.

Abschließend ist zu sagen, daß die Tcl/Tk-Szene sehr aktiv ist, so daß eine Betrachtung der jeweils aktuellen FAQ immer von Nutzen sein kann. Außerdem existiert inzwischen ein interaktiver Interface-Builder für Tk [5], den man sich ebenfalls einmal anschauen sollte. Er wird mit dem Kommando `xf` aufgerufen. Informationen über neu am DFKI Kaiserslautern installierte Tcl/Tk-Versionen oder -Erweiterungen werden in der Newsgroup `dfki.isg` verbreitet. Die in der vorliegenden Arbeit beschriebene Software ist im Verzeichnis `/home/taxon/ABI-TK/` abgelegt.