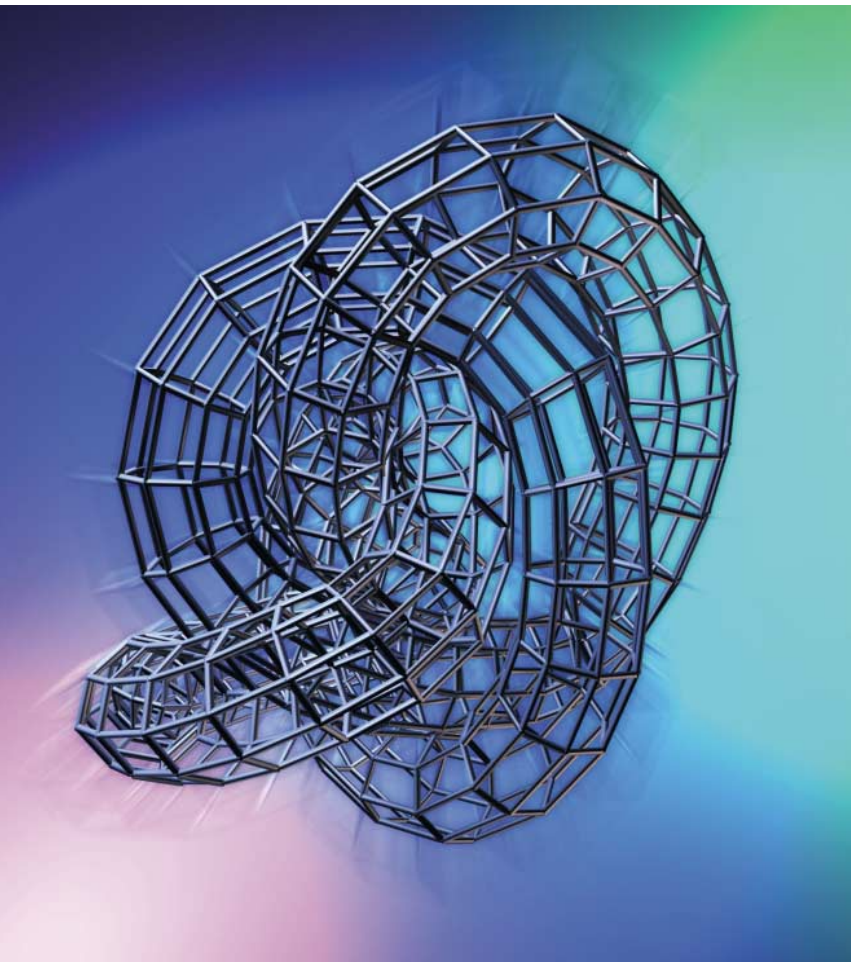


Graphen erstellen und manipulieren mit Graphviz

Knoten mit Kanten

Jürgen Schuck

Als Leonhard Euler 1736 das Königsberger Brückenproblem löste, konnte er noch nicht wissen, dass er damit die Graphentheorie aus der Taufe hob – und schon gar nicht, dass es ein paar Jahrhunderte später Werkzeuge geben würde, mit denen sich Graphen in null Komma nichts beschreiben und darstellen lassen. Dazu zählt unter anderm das frei verfügbare Graphviz.

Arbeitsabläufen und Klassenbibliotheken ist gemeinsam, dass sich ihre Organisation und Wirkungsweise grafisch darstellen lassen und zwar mit den gleichen Stilmitteln: Kästen und Kreise stellen wesentliche Elemente wie Arbeitsschritte und Klassen dar, Verbindungslinien zeigen den Ablauf der Arbeitsschritte und legen die Erbfolge der Klassen fest. Die Beispiele lassen sich erweitern um Aktivitätsdiagramme aus der Unified Modeling Language (UML), Beziehungen zwischen Instanzen ausführbarer Programme (Prozesse) und Abhängigkeiten der Quelltextmodule eines Softwarepakets. Was diese Beispiele außerdem gruppiert, ist die Rangfolge ihrer Elemente, der so genannten Nodes: Es gibt Abhängige, die ihrerseits weitere abhängige Nodes haben. Sie ordnen sich somit zu einer Rangfolge oder Hierarchie, die der Jargon als „directed graph“ bezeichnet. Natürlich gibt es auch das Gegenteil, den „undirected graph“, zur Darstellung von Entity-Relation-Diagrammen und Zustandsautomaten. Hierbei sind sämtliche Nodes gleichwertig, da es keine kausalen Zusammenhänge zwischen ihnen gibt.

Zu den effizientesten Zeichenwerkzeugen für Graphen gehören zweifellos das Whiteboard sowie Papier und Bleistift. Sie sind jedoch nicht überall verfügbar, außerdem lässt sich das Arbeitsergebnis nicht unmittelbar elektronisch verarbeiten. Diese Makel beseitigen Computerprogramme, die im Wesentlichen in zwei Kategorien fallen: Die Low-Level-Bibliotheken für Programmierer einerseits und Zeichenprogramme oder Editoren zur interaktiven Bedienung auf der anderen Seite (siehe Kasten „Weitere freie Tools …“).

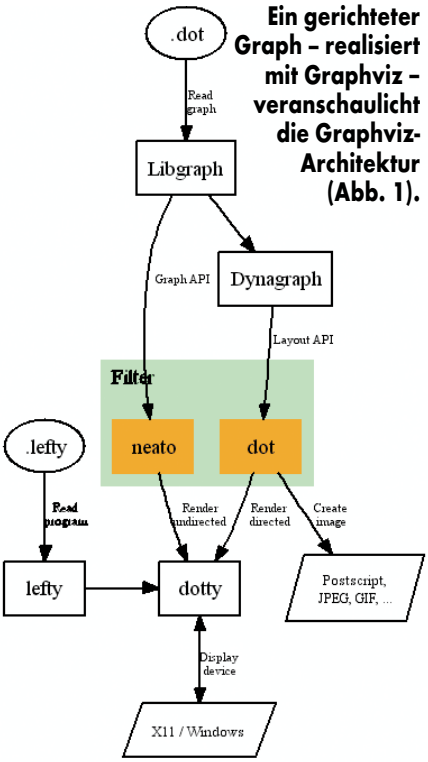
Filtern und dynamisch verändern

Zwischen beiden ist das Graph Visualization Toolkit (Graphviz) von AT&T angesiedelt, das Graphbeschreibungsdateien (ASCII) in Linux-Filtermanier interpretiert (*stdin*), berechnet und als Graphen in verschiedenen Formaten wie Postscript, JPG, PNG oder GIF ausgibt (*stdout*). Eine weitere interessante Eigenschaft des Tools ist die dynamische Veränderung von Graphen: Sie ermöglicht beispielsweise die automatische Aktualisierung der Darstellung einer Prozessgruppe, wenn sich Prozesse beenden oder neue hinzukommen – praktisch eine grafische Version des Prozessmonitors *top*.

Die Graphviz-Struktur (Abbildung 1) lässt die Gliederung in APIs, Filter-

iX-TRACT

- Graphviz bietet eine Filterschnittstelle zur skriptbasierten Erstellung von Graphen.
- Das Graphbeschreibungsformat ist einfach und leicht aus Programmen und Scripts zu erzeugen (ASCII).
- Ein frei programmierbarer Editor ermöglicht interaktive Darstellungen von Graphen. Die Funktionen für das Handling von Graphen sowie deren Layout stehen als C-API zur Verfügung.



Ein gerichteter Graph – realisiert mit Graphviz – veranschaulicht die Graphviz-Architektur (Abb. 1).

GraphViz-Bestandteile und ihr Zusammenwirken

werkzeuge und interaktive Programme erkennen. Die Bibliotheken *Libgraph* und *Dynagraph* (nicht zu verwechseln mit dem gleichnamigen 3D-Plotting-Werkzeug für X11) bilden die Programmierschnittstelle. *Libgraph* liest Graphbeschreibungen aus unterschiedlichen Quellen wie Dateien, Pipes oder der Standardeingabe und transformiert sie in eine interne Repräsentation. Die Bibliothek enthält Funktionen zum I/O, zum Einfügen und Löschen von Elementen der internen Graphabstrahierungen und für das Speichermanagement. Ein Callback-Konzept dient zur Registrierung von Funktionen, die *Libgraph* bei Veränderungen an den Graphen aufruft. Den Diagrammbeschreibungen liegt eine so genannte Graph Description Language (GDL) zu Grunde, die als *dot*-Datei gespeichert wird. Diese Formatbezeichnung leitet sich von dem gleichnamigen Programm ab, dem – wenn überhaupt – bekanntesten Werkzeug des Toolkits.

Dynagraph, das auf *Libgraph* aufsetzt, bietet eine ereignisgesteuerte Funktionsschnittstelle für dynamische Layout-Anpassungen bei Veränderungen an einem Graphen. Damit lassen sich Darstellungswerkzeuge entwickeln, die unmittelbar auf Änderungen reagieren, indem sie das Layout anpassen und die Darstellung aktualisieren. Eine Übersicht der APIs beider Bibliotheken findet sich zusammen mit einigen einführenden Sätzen auf AT&Ts Website [1]. Zum tieferen Verständnis muss der Programmierer jedoch die Quellen existierender Applikationen studieren, wozu sich die Filterwerkzeuge *dot* und *neato* gut eignen.

Von den beiden Filterwerkzeugen des Toolkits ist *dot* das Gebräuchlichere. Im Gegensatz zu *neato*, das für die Darstellung ungerichteter Graphen optimiert ist, erzeugt *dot* Abbildungen gerichteter Graphen. Beide Werkzeuge beherrschen aber auch den anderen Typ. *dot* und *neato* lesen eine Graphbeschreibung im *dot*-Format von der Standardeingabe, erzeugen ein Layout und schreiben es im gewünschten Format auf die Standardausgabe. Im Fall reiner Image-Formate wie JPEG und GIF bietet sich anstelle von *stdout* die Verwendung einer Datei an, was wie beim Ausgabeformat über eine Option auf der Kommandozeile geschieht.

Tools kommunizieren über Pipes

Schließlich gibt es mit *dotty* ein interaktives Werkzeug zur Darstellung und Manipulation von Graphen. Je nach Typ des Graphen verwendet *dotty* *dot* oder *neato* zur Layout-Erzeugung. Die Kommunikation erfolgt über Pipes. Als Grafikeditor dient *lefty*, ein Zeichenprogramm für technische Abbildungen, das Darstellungen und Editierfunktionen in einer Interpreter-Sprache ausdrückt, die ebenfalls *lefty* heißt.

Die Syntax des *dot*-Formats ist einfach: es gibt Graphen, die sich aus Knoten (Nodes) und deren Kanten (Edges) zusammensetzen. Der Ausdruck

```
digraph graphviz {
        Libgraph -> Dynagraph
}
```

definiert einen gerichteten Graphen namens *graphviz*, der aus den Nodes *Libgraph* und *Dynagraph* besteht, die ebenso bezeichnet und über eine *Edge* verbunden sind. Eine Erweiterung des Ausdrucks zu

```
digraph graphviz {
  ".dot" -> Libgraph [ label = "Read\ngraph" ]
  Libgraph -> Dynagraph
}
```

führt mit *.dot* einen weiteren Node ein, der mit *Libgraph* verbunden ist. Die Edge ist außerdem mit dem Attribut *label* versehen, dessen Wert sie in der Abbildung bezeichnet. Ohne Quotes ist

Listing 1

```
digraph graphviz {
      label = "Graphviz-Bestandteile und ihr Zusammenwirken"
      dot [ color = ".098 , 1 , .931" , style = filled ]
      neato [ color = ".098 , 1 , .931" , style = filled ]
      ".dot" -> Libgraph [ label = "Read\ngraph" ]
      Libgraph -> Dynagraph
      dot -> dotty [ label = "Render\ndirected" ]
      neato -> dotty [ label = "Render\nundirected" ]
}
```

Über ein Attribut lässt sich ein Graph benennen.

der Zeichenvorrat für Nodes auf A–Z, a–z, 0–9 und _ beschränkt. Attribute gibt es außer für Edges für Nodes und Graphen. Eine Liste findet sich ebenfalls bei AT&T.

Listing 1 zeigt neben dem durch das Attribut *label* bezeichneten Graph mit *dot* und *neato* zwei weitere, farbig ausgefüllte Nodes, wobei die Farbangabe als HSB-Wert (Hue, Saturation, Brightness) eine von drei Möglichkeiten darstellt, die *dotguide.pdf* [1] beschreibt.

Die reservierten Wörter *node* und *edge* legen die Werte von Attributen fest, falls dies nicht ausdrücklich mit der Definition von Nodes und Edges erfolgt. Das Graphattribut *rank* steuert die Anordnung von Nodes: Der Wert *same* ordnet alle folgenden Nodes auf gleicher Höhe an.

Listing 2 legt für sämtliche Nodes des Graphen *graphviz* eine rechteckige Form fest. Die geschweiften Klammern fassen *.dot* und *.lefty* zu einem anonymen Subgraphen zusammen, der die Wirkung von Attributen, wie hier *node* zur Festlegung einer ovalen Form, auf die enthaltenen Nodes beschränkt. Für die Beschriftungen an Edges ist außerdem eine Zeichengröße von 8 Punkten festgelegt. Schließlich erzwingt der Subgraph mit *rank* = *same* die Anordnung der Nodes *lefty* und *dotty* auf gleicher Höhe, während das Layout-Programm *dot* die Positionen der übrigen Nodes bestimmt. Das Layout berück-

Listing 2

```
digraph graphviz {
      label = "Graphviz-Bestandteile und ihr Zusammenwirken"
      { node [ shape = ellipse ] ; ".dot" ; ".lefty" }
       node [ shape = box ]
       edge [ fontsize = 8 ]
       { rank = same ; lefty ; dotty }
      dot [ color = ".098 , 1 , .931" , style = filled ]
      neato [ color = ".098 , 1 , .931" , style = filled ]
      ".dot" -> Libgraph [ label = "Read\ngraph" ]
      Libgraph -> Dynagraph
      dot -> dotty [ label = "Render\ndirected" ]
      neato -> dotty [ label = "Render\nundirected" ]
}
```

Geschweifte Klammern fassen mehrere Graphen zu einem Subgraphen zusammen.

Listing 3

```
digraph graphviz {
     label = "Graphviz-Bestandteile und ihr Zusammenwirken"
     compound = true
     { node [ shape = ellipse ] ; ".dot" ; ".lefty" }
      node [ shape = box ]
      edge [ fontsize = 8 ]
      { rank = same ; lefty ; dotty }
     subgraph cluster_filters { label = "Filter" ; labeljust = l ;
                    color = "palegreen" ; style = filled ; dot ; neato }
     dot [ color = ".098 , 1 , .931" , style = filled ]
     neato [ color = ".098 , 1 , .931" , style = filled ]
     ".dot" -> Libgraph [ label = "Read\ngraph" ]
     Libgraph -> Dynagraph
     Dynagraph -> dot [ label = "Layout API" , lhead = cluster_filters ]
     Libgraph -> neato [ label = "Graph API" , lhead = cluster_filters ]
     dot -> dotty [ label = "Render\ndirected" ]
     neato -> dotty [ label = "Render\nundirected" ]
}
```

Das Attribut *compound* legt fest, wo die Verbindungen zwischen externen und internen Nodes beginnen.

Listing 4

```
digraph graphviz {
     label = "Graphviz-Bestandteile und ihr Zusammenwirken"
     compound = true
     { node [ shape = ellipse ] ; ".dot" ; ".lefty" }
      node [ shape = box ]
      edge [ fontsize = 8 ]
      { rank = same ; lefty ; dotty }
     subgraph cluster_filters { label = "Filter" ; labeljust = l ;
                    color = "palegreen" ; style = filled ; dot ; neato }
     display [ shape = polygon , sides = 4 , skew = .3 , label = "X11 / Windows" ,
               fontsize = 10 ]
      image [ shape = polygon , sides = 4 , skew = .3 ,
               label = "Postscript,\nJPEG, GIF, ..." , fontsize = 10 ]
     dot [ color = ".098 , 1 , .931" , style = filled ]
     neato [ color = ".098 , 1 , .931" , style = filled ]
     ".dot" -> Libgraph [ label = "Read\ngraph" ]
     Libgraph -> Dynagraph
     lefty -> dotty
     ".lefty" -> lefty [ label = "Read\nprogram" ]
     Dynagraph -> dot [ label = "Layout API" , lhead = cluster_filters ]
     Libgraph -> neato [ label = "Graph API" , lhead = cluster_filters ]
     dot -> dotty [ label = "Render\ndirected" ]
     neato -> dotty [ label = "Render\nundirected" ]
     dot -> image [ label = "Create\nimage" , ltail = cluster_filters ]
     dotty -> display [ label = "Display\ndevice" , arrowtail = normal]
}
```

Diese Datei erzeugt den in Abbildung 1 zu sehenden Graphen.

sichtigt ausdrücklich bezeichnete Subgraphen, indem es die enthaltenen Nodes in einem eigenen Kasten anordnet. Dazu müssen die Namen dieser Subgraphen mit *cluster* beginnen. Syntaktisch unterscheiden sich Subgraphen nicht von Graphen. Spezielle Edge-Attribute legen fest, ob Verbindungen zwischen externen und internen Nodes bereits am Kasten eines Subgraphs enden oder beginnen. Zur Verwendung dieser Attribute – *lhead* und *ltail* – ist das Graphattribut *compound* mit dem Wert *true* notwendig.

Der Ausdruck zur Definition des Subgraphs *cluster_filters* in Listing 3 zeigt, wie das Attribut *labeljust* mit dem Wert *l* die Positionierung der Bezeichnung auf der linken Seite anstatt in der Mitte bewirkt. Außerdem ist mit *color = „palegreen“* eine weitere Möglichkeit zur Farbangabe gezeigt. Die Farbbezeichnungen von Graphviz sind im übrigen X11 entlehnt.

Eigene Visualisierungstools entwickeln

Soweit die wesentlichen Elemente von Graphviz. Wenige, in Listing 4 zu sehende Ergänzungen, die keine neuen Konzepte mehr einführen, machen aus dem Beispiel die in Abbildung 1 zu sehende Architekturbeschreibung von Graphviz.

Das Kommando *dot -Tgif -ograph-viz.gif graphviz.dot* erzeugt eine Abbildung im GIF-Format, wenn die Datei *graphviz.dot* die Beschreibung des Graphs als ASCII-Text enthält. Wer sich für die Unterschiede des *neato*-Layouts interessiert, ersetzt in *graph-viz.dot digraph* durch *graph* und sämtliche „->“ durch „–“ sowie in der Kommandozeile *dot* durch *neato*.

Da die einfache Syntax des *dot*-Formats leicht durch Scripts und Programme zu erzeugen ist, bietet sie sich zusammen mit *dot* als Grundlage zur Entwicklung automatisierter Visualierungswerkzeuge an. So entsteht aus wenigen Zeilen *awk* ein Tool zur Darstellung von Prozesshierarchien:

```
ps -elf | awk 'BEGIN {print "digraph ps {"}
          NR>1 {print $5 "->" $4}
END {print "}"}' | dot -Tgif -ops.gif
```

Aus diesen Zeilen resultiert eine Darstellung der Prozesse eines Linux-Systems, die zwar einfach ist, aber immerhin schon das Zusammenwirken der einzelnen Prozesse auf einen Blick erkennen lässt. Einige wenige Zeilen zusätzlichen Codes machen daraus eine Übersicht, die den Informationsgehalt der *ps*-Ausgabe auf einen Blick erkennen lassen (Listing 5).

Wer das Ganze etwas genauer haben möchte und über ausreichend großes Papier verfügt, dem sei ein Blick in die Manualseite des *lsof*-Kommandos empfohlen, das eine Liste der geöffneten Dateien ausgibt, inklusive der verschiedenen Attribute wie Typ und Modus (Schreiben/Lesen) sowie der Prozesszuordnung.

Was jetzt noch fehlt, sind Interaktionsmöglichkeiten zum Verschieben von Nodes oder zum Beenden von Prozessen sowie dynamische Anpassungen an Zustandsänderungen, die beispielsweise die Terminalzuordnung, die Wartekanäle oder Prozessbeendigungen betreffen. Hier ist das Einsatzgebiet von *dotty*, dem interaktiven und programmierbaren Grapheditor. Seine optische Ausstattung ist zwar nicht preisverdächtig – Athena Widgets lassen grüßen – was aber die freie Programmierbarkeit in der Interpretersprache *lefty* durchaus wettmacht. Eine

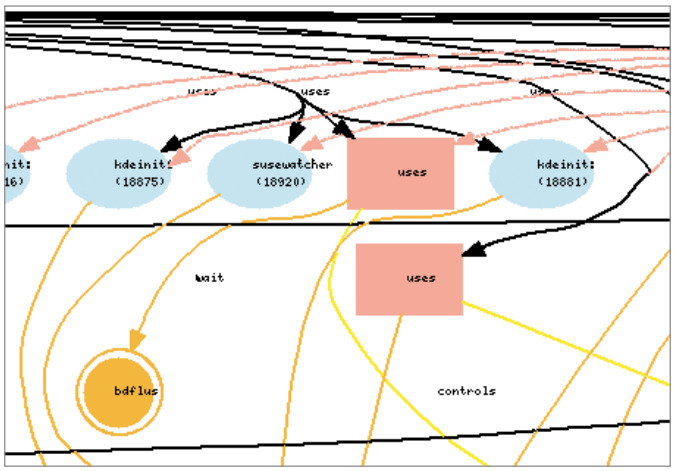
Installation

Die Installation von Graphviz kann auf auf zwei Arten erfolgen: aus einem vorkompilierten Binary oder durch Übersetzen der Sourcen auf dem Zielsystem. Binaries gibt es für praktisch alle populären Plattformen: Linux, FreeBSD, OpenBSD, MAC OS X, Sun, HP-UX, SGI sowie Windows und Cygwin. Sie liegen im Package-Format des jeweiligen Systems vor, was eine einfache Installation sicherstellt.

Bei der Übersetzung der aktuellen Quellen in der Version 1.14 unter Cygwin und Linux sind einige Dinge zu beachten. Unter Cygwin stellt das *configure*-Script die Erzeugung von DLLs ein, was (wie so häufig) nicht funktioniert. Also statische Bibliotheken binden und *configure* dazu mit der Option *–disable-shared* aufrufen:

```
tar zxf graphviz-1.14.tar.gz
cd graphviz-1.14
./configure –disable-shared
make
make install
```

Derartige Probleme kennt Linux nicht, das die *.so*-Dateien problemlos erzeugt. Wer nur mal hineinschnuppern und daher die Installation nach */usr/local* vermeiden möchte, muss nach *make* den Pfad entsprechend einstellen: *export PATH=`pwd`/dotneato:`pwd`/dotty:`wd`/lefty:$PATH*. Unter Linux muss die Shell-Variable *LD_LIBRARY_PATH* auf die Verzeichnisse mit den *.so*-Dateien zeigen, die man mit *find `pwd` -name '*.so*' -exec dirname {} \; | sort -u* erhält. Zum Ausprobieren findet man in den Verzeichnissen *./graphs/directed* und *./graphs/undirected* eine Reihe von Beispielen für *dot, neato* und *dotty*. Beispiele für *lefty* enthält das Verzeichnis *./lefty/examples*.



Listing 5

```
ps -elf |\
awk 'BEGIN{print "digraph ps {label = \"Visual ps(1) by dot\" ; \
  concentrate = true ; node [ fontsize = 8 ] ; edge [ fontsize = 8 ]"}
   NR>1 {if ($2~/S/) s="", color = lightblue, style = filled"
      if ($2~/R/) s="", color = palegreen, style = filled"
      if ($2~/Z/) s="", color = lightgrey, style = filled"
      printf("%d [ label = \"%s\\n(%d)\"%s ]\n", $4, $15, $4, s)
      printf("%s [ shape = parallelogram, color = salmon, style = filled ]\n", $3) ;
      if ($13!~/?/) {
           printf("\"%s\" [ shape = box, color = yellow, style = filled ]\n", $13)
           printf("\"%s\" -> %s [ label = controls, color = yellow ]\n", $13, $4)
           printf("%s -> \"%s\" [ label = uses, color = salmon ]\n", $3, $13)
      } else
           printf("%s -> %s [ label = uses, color = salmon ]\n", $3, $4)
      if ($11!~/[?~]/) {
           printf("%s [ shape = doublecircle, color = orange, style = filled ]\n", $11)
           printf("%s -> %s [ label = wait, color = orange ]\n", $4, $11)
      }
      print $5 "->" $4}
END{print "}"}' |\
dot -Tgif -ops.gif
```

Wenige Zeilen *awk* machen aus der Ausgabe des *ps*-Kommandos im Zusammenspiel mit Graphviz eine intuitive Grafik. Sie zeigt die Benutzer in Rot, Terminals in Gelb und die Wartekanäle in Orange. Die Prozessellipsen sind im schlafenden Zustand blau und bei laufenden Prozessen grün. Ein grauer Prozess ist ein Zombie (Abb. 2, Ausschnitt).

Vorstellung von *lefty* kann an dieser Stelle nicht erfolgen; mit der Sprachreferenz [1] ist aber eine rasche Einarbeitung möglich. Einmal dabei, sollte man sich gleich anhand von *dottyguide.pdf* [1] mit der Programmierung von *dotty* vertraut machen, womit man die Voraussetzungen zur Entwicklung eines visuellen *top* – wie eingangs erwähnt – geschaffen hat.

Die Interaktionsmöglichkeiten von *dotty* erschließt man sich am besten, indem man die Ausgabe des *awk*-Skripts anstelle von *dot* auf die Eingabe des Kommandos *dotty* - umlenkt. Das *dotty*-Fenster mit der Prozessdarstellung bietet unter der rechten Maustaste ein Kontextmenü an, mit dem sich Nodes und Edges beispielsweise löschen lassen. Mit der linken Taste kann man Nodes verschieben oder erzeugen; dazu die Taste mit dem Zeiger auf dem Hintergrund betätigen. Die mittlere Maustaste dient zum Verbinden von Nodes.

Für die angesprochene Dynamik muss man programmieren. Die Konzepte von *dotty* und der Sprach- und Funktionsumfang von *lefty* bieten kreativen Geistern weitreichende Möglichkeiten. So kann man zum Beispiel die Funktion des Kontextmenüs zum Löschen von Nodes um den Aufruf des *kill*-Kommandos erweitern oder die Prozessstruktur zyklisch lesen und Änderungen im Graphen aktualisieren.

Fazit

Graphviz enthält leicht erlernbare Werkzeuge zur Darstellung von Graphen, die von Low-Level-Bibliotheken, über Filter- und Skript-Tools bis zum interaktiven Grapheditor reichen. Insbesondere durch seine Filter- und Skripteigenschaften ist es ein wertvolles Werkzeug, um ansprechende Darstellungen aus textuellen Formaten technischer Relationen schnell zu erstellen. Als solches kann es zum Beispiel zur automatischen Generierung grafischer Netzwerk- und Serverpläne dienen, deren Aktualität bei manueller Erstellung oft nicht gegeben ist. (ka)

WEITERE FREIE TOOLS ZUR DARSTELLUNG VON GRAPHEN

Mascopt: LGPL; Java API zur Optimierung von Netzen und Graphen	www-sop.inria.fr/mascotte/mascopt/
Pajek: frei bei nicht kommerzieller Nutzung; Windows-GUI zur Analyse von Netzen	vlado.fmf.uni-lj.si/pub/networks/pajek/
GDToolkit: frei bei nicht kommerzieller Nutzung; C++-API und Kommandozeilen-Tools mit XML-ähnlicher Graphbeschreibungssprache	www.dia.uniroma3.it/~gdt/
Tulip: GNU GPL; X11-GUI zur Darstellung großer Graphen mittels OpenGL; Interaktionsmöglichkeiten, automatische Cluster-Bildung, einfache Graphbeschreibungssprache (ASCII)	www.tulip-software.org
P.I.G.A.L.E.: GNU GPL; C++-API und auf Trolltechs Qt basierender Graph-Editor für Windows und Linux; makrofähig	pigale.sourceforge.net
AGD: frei bei nicht kommerzieller Nutzung; C++-API für Windows und Unix; Algorithmen zur Darstellung von Graphen; keine Graphbeschreibungssprache	aragorn.ads.tuwien.ac.at/AGD/

INFOS IM WEB

Homepage des Graph Visualization Toolkits bei AT&T	www.research.att.com/sw/tools/graphviz/
Homepage des Graph-Drawing-Symposiums mit einer Liste einschlägiger Literatur	graphdrawing.org
Weitere Visualisierungsprojekte von AT&T	www.research.att.com/areas/visualization/

JÜRGEN SCHUCK

ist Mitarbeiter der Materna GmbH und als Projektleiter im Bereich IT-Service-Management tätig.

Literatur

[1] AT&T Labs – Research; www.research.att.com/sw/tools/graphviz/[GN99.pdf, dotguide.pdf, neatoguide.pdf, dottyguide.pdf, leftyguide.pdf, libguide.pdf] (Emden R. Gansner, Stephen C. North, Eleftherios Koutsofios)

[2] John Hamer; Visualising Java Data Structures as Graphs; University of Auckland, Department of Computer Science (2004); portal.acm.org/citation.cfm?id=979985