Informatik – Forschung und Entwicklung manuscript No.
(will be inserted by the editor)

Datenstrukturen visualisieren und animieren mit DDD*

Andreas Zeller

Lehrstuhl für Softwaresysteme, Fakultät für Mathematik und Informatik, Universität Passau, Innstraße 33, 94032 Passau
e-mail: `zeller@acm.org`

Eingegangen am ⟨Datum⟩ / Angenommen am ⟨Datum⟩

Zusammenfassung Der graphische Debugger *DDD* ist mit mehr als 250.000 Anwendern ein weitverbreitetes Werkzeug zur Softwarevisualisierung. Neben der für Debugger üblichen Funktionalität ermöglicht DDD die *Visualisierung von Datenstrukturen* im laufenden Programm. DDD kann verzeigerte Strukturen (wie Listen oder Bäume) als Graphen darstellen, aber auch numerische Felder zwei- oder dreidimensional darstellen und im Programmlauf animieren. Dieser Beitrag beschreibt die technischen Grundlagen der Visualisierung, die verwendeten Plazierungsverfahren und die Animation von Algorithmen.

Schlüsselwörter Visuelles Debuggen – Visualisierung der Daten und Abläufe in Anwendungen – Visualisierung von Datenstrukturen – Algorithmenanimation

Abstract With more than 250.000 users, the graphical debugger DDD is a widespread software visualization tool. Besides the usual debugger features, DDD allows *visualizing data structures* from the running program. DDD can visualize pointer structures (such as lists or trees) as graphs, but also numerical arrays as two- or three-dimensional plots—and animate it all during the program run. In this paper, we describe the visualization techniques, the layout methods used as well as the animation of algorithms.

Key words visual debugging – visualizing data and processes in applications – visualizing data structures – algorithm animation

CR Subject Classification: **D.2.5** Testing and Debugging—*Debugging aids*, **D.2.6** Programming Environments—*Graphical environments*, **H.5.1** Multimedia Information Systems—*Animations*

* Dies ist eine erweiterte und überarbeitete Fassung des gleichnamigen Beitrags aus dem *GI-Workshop Softwarevisualisierung*, Schloß Dagstuhl, 11.–12. Mai 2000.

1 Einführung: Visuelles Debuggen

Trotz aller Fortschritte in den frühen Phasen der Softwareentwicklung gehört die *Fehlersuche* weiterhin zum täglichen Brot der Programmierer. *Interaktive Debugger* haben sich als Universal-Werkzeuge zur Fehlersuche etabliert. Mit einem Debugger kann man

- das Programm in einer definierten Umgebung *ausführen*,
- das Programm unter bestimmten Bedingungen *anhalten lassen*,
- den Zustand des angehaltenen Programms *untersuchen* und
- den Zustand des angehaltenen Programms *verändern.*

Wenn es um die Untersuchung des Programmzustands geht, ist der zentrale Aspekt die *Datenausgabe.* Traditionelle Debugger geben ihre Daten als *Text* aus – so etwa der GNU-Debugger GDB:

```
(gdb) display *tree
*tree = {value = 7, _name = 0x8049e88 "Ada",
  _left = 0x804d7d8, _right = 0x0,
  left_thread = false, right_thread = false,
  date = {day_of_week = Thu,
   day = 1, month = 1, year = 1970,
   _vptr. = 0x8049f78 ⟨Date virtual table⟩},
   static shared = 4711}
(gdb) _
```

Abb. 1 Textuelle Datenausgabe mit GDB

Auch wenn moderne Debugger die Gesamtausgabe nicht mehr als Fließtext gestalten, werden doch die Daten nach wie vor als *Text* ausgegeben. Für einzelne Daten ist diese textuelle Darstellung gewiß angemessen – wie wollte man eine Zeichenkette schon betrachten, wenn nicht als Text? Schwierig wird es aber, wenn es um *Beziehungen* zwischen Daten geht – Verweise, Indizes und dergleichen. Worauf etwa zeigt `tree→_left`? Natürlich kann der Anwender dies individuell abfragen. Wenn aber zwei Zeiger denselben Wert haben,

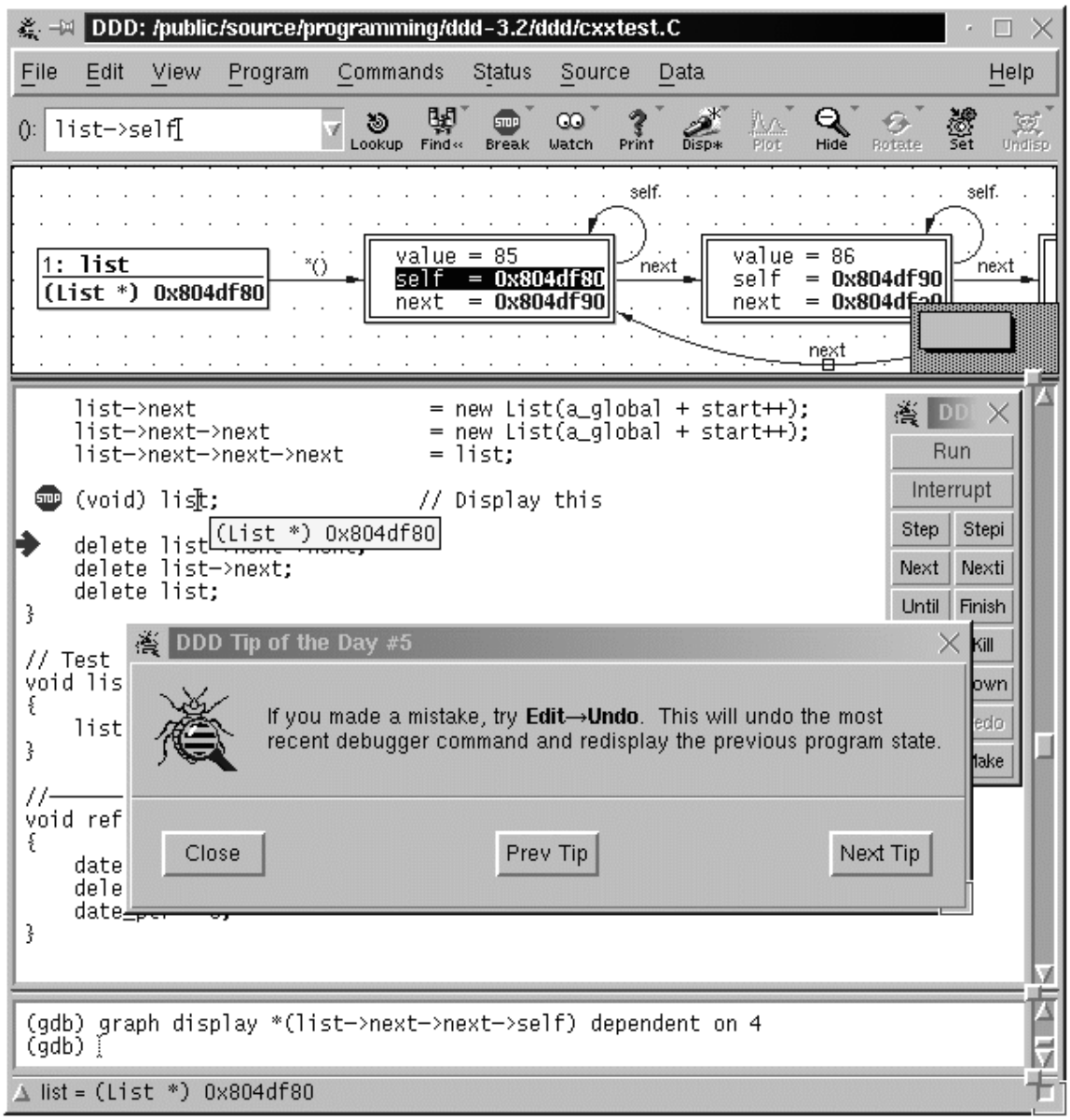


Abb. 2 Die DDD-Bedienoberfläche mit Programmdaten, Quelltext, einem Bedienungshinweis, und einer Konsole für zusätzliche Befehle.

kann der Anwender dies nur durch mühsames Vergleichen der Adressen bestimmen.

Getreu dem Motto, daß der Mensch bildliche Eindrücke besonders effizient verarbeiten kann und konkrete Objekte besser erfassen kann als abstrakte Beschreibungen, haben wir einen Debugger konstruiert, der über *eingebaute Datenvisualisierung* verfügt – den *Data Display Debugger* oder kurz DDD.

DDD ist technisch realisiert als *Front-End* zu herkömmlichen Kommandozeilen-Debuggern. DDD unterstützt derzeit GDB, DBX, LADEBUG, XDB, JDB, den Python-Debugger und den Perl-Debugger – und damit nahezu alle gängigen imperativen Programmiersprachen. Die Oberfläche (Abbildung 2) bietet allen gewohnten Komfort einschließlich einer detaillierten zustands- und kontext-sensitiven Hilfe. DDD ist als offizielle GNU-Software im Quelltext frei erhältlich und wird heute von über 250.000 Anwendern zur Programmentwicklung eingesetzt; die Liste der Anwender reicht von Adobe und Boeing bis zu Texas Instruments und Xerox.

Im folgenden zeigen wir, wie sich DDD als Universalwerkzeug zur dynamischen Softwarevisualisierung einsetzen läßt. Wir beschreiben zunächst die Visualisierung von Datenstrukturen aus dem laufenden Programm – als Knoten (Abschnitt 2), Bäume (Abschnitt 3) und schließlich Graphen (Abschnitte 4 und 5).

In den Abschnitten 6 und 7 zeigen wir, wie sich diese Darstellungen im Programmlauf animieren lassen und wie dies für das Programmverstehen genutzt werden kann. Abschnitt 8 diskutiert die Darstellung numerischer Werte; Wege zu weiteren Darstellungen werden in Abschnitt 9 skizziert. Verwandte Arbeiten sind in Abschnitt 10 behandelt; Abschnitt 11 schließt mit Zusammenfassung und Ausblick.

2 Visualisierung individueller Daten

Alle Daten, die DDD in seinem Datenfenster darstellt, erhält DDD vom darunterliegenden Kommandozeilen-Debugger, etwa GDB. GDB unterhält eine *Display-Liste* mit symbolischen Ausdrücken, die bei jedem Programmhalt ausgegeben werden. Das GDB-Kommando `display *tree` etwa fügt den Ausdruck `*tree` zur Display-Liste hinzu und läßt GDB bei jedem Halt den Wert von `*tree` ausgeben – etwa, wie in Abbildung 1 auf Seite 1 gezeigt. Diese Ausgabe wird von DDD verarbeitet und im Datenfenster dargestellt – jedoch in Form eines strukturierter graphischen *Displays* (Abbildung 3).

Wie erzeugt DDD ein solches Display? Grundidee ist, jedem individuellen Datum einen eigenen rechteckigen Bereich, eine sogenannte *Schachtel* (engl. *box)*, zuzuweisen. Eine Schachtel ist entweder

- *atomar* und enthält einen einzelnen Text oder ein graphisches Grundelement oder
- *zusammengesetzt* und umfaßt mehrere Schachteln horizontal oder vertikal.

In Abbildung 3 etwa kann man deutlich sehen, daß jedem Wert des Verbundes eine eigene Schachtel zugewiesen ist. Der Wert von `date` ist ebenfalls ein Verbund (und somit zusammengesetzt); die anderen Werte sind atomar.

Um das Display zu erzeugen, analysiert DDD den von GDB zurückgegebenen textuellen Wert und erzeugt daraus zunächst einen *abstrakten Syntaxbaum*. Dieser wird in einen *Term* umgewandelt, in dem jedem Knotentyp eine *Display-Funktion* zugewiesen wird. Jede Display-Funktion berechnet eine knotenspezifische Schachtel anhand ihrer Argumente. Der Term liefert so eine Schachtel, die den textuellen Wert graphisch darstellt.

Hier ein Beispiel. Der Text t mit dem Inhalt

$$\{\ \texttt{value = 7, \_name = 0x8049e88 "Ada",...}\}$$

etwa wird DDD-intern zum Term

$$\begin{aligned}&\mathit{struct\_value}\Big(\\&\quad \mathit{struct\_member}\big(\texttt{"value"}, \mathit{simple\_value}(\texttt{"7"})\big),\\&\quad \mathit{struct\_member}\Big(\texttt{"\_name"}, \mathit{sequence\_value}(\\&\qquad \mathit{pointer\_value}(\texttt{"0x8049e88"}),\\&\qquad \mathit{simple\_value}(\texttt{"\textbackslash"Ada\textbackslash""})\big)\Big)\Big)\end{aligned}$$

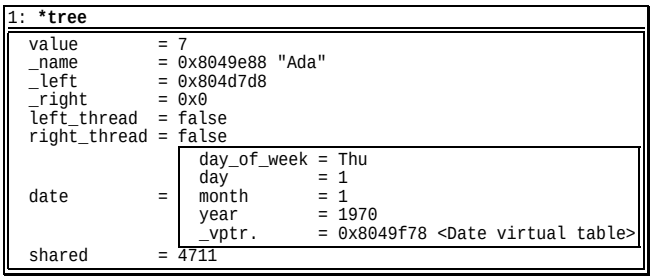


Abb. 3 Ein geschachteltes DDD-Display

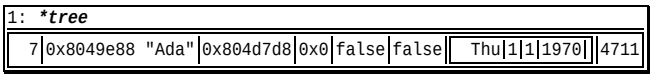


Abb. 4 Alternative Darstellung eines DDD-Displays. Durch Weglassen der Elementnamen wird Platz eingespart.

Um die Schachtel für t zu bestimmen, wertet DDD diesen Term aus, und zwar anhand der *Definitionen* der einzelnen Display-Funktionen. Diese sind in der DDD-eigenen Sprache VSL (*visual structure language*) angegeben. Die Display-Funktion *struct_member*, die ein Name-Wert-Paar eines Verbundelementes darstellt, ist in DDD wie folgt definiert:

$$\mathit{struct\_member}(\mathit{name}, \mathit{value}) = \mathit{name}\ \&\ \texttt{" = "}\ \&\ \mathit{value}$$

Hierbei ist & der VSL-Operator, der einzelne Schachteln horizontal zusammenfaßt; der Text `" = "` steht für eine Schachtel mit diesem Inhalt. Die Display-Funktion sorgt dafür, daß Verbund-Elemente als Name/Wert-Paare, getrennt durch ein Gleichheitszeichen dargestellt werden.[1] Die weiteren Display-Funktionen sind auf ähnliche Weise realisiert.

DDD enthält bereits eine vollständige VSL-Bibliothek mit allen nötigen Definitionen. Der Benutzer kann aber diese Vorgaben durch Eigenentwicklungen ersetzen. Um etwa die Ausgabe der Elementnamen zu unterdrücken (und somit Platz zu sparen), könnte der Benutzer *struct_member* umdefinieren zu

$$\mathit{struct\_member}(\mathit{name}, \mathit{value}) = \mathit{value}$$

Wird nun noch *struct_value* so umdefiniert, daß die Verbundelemente horizontal statt vertikal dargestellt werden, und werden außerdem alle Elemente mit Namen `_vptr.` unterdrückt, so ergibt sich die platzsparende Darstellung in Abbildung 4.

Der Benutzer kann so definierte Änderungen als sogenanntes *Thema* (*theme*) in einer Datei zusammenfassen. Diese Themen können zur Laufzeit eingeschaltet (= nachgeladen) und wieder abgeschaltet werden; es ist sogar möglich, sie selektiv nur auf bestimmte Ausdrucksmuster anzuwenden. Paßt ein darzustellender Ausdruck auf ein solches Muster, wird er im ausgewählten Thema dargestellt.

Das wichtigste dieser Themen ist das vollständige *Unterdrücken* eines Ausdrucks, etwa durch die Definition

$$\mathit{simple\_value}(\mathit{value}) = \texttt{""}$$

Wird dieses Thema etwa auf das Muster `*->date` angewandt, werden die Verbundelemente mit Namen `date` nicht mehr angezeigt. Andere Themen verkleinern oder vergrößern die Schriftart oder ändern die Positionierung von Displays. Auf diese Weise läßt sich die Darstellung an das jeweilige Problemfeld anpassen.

[1] Alle hier angegebenen Definitionen sind vereinfacht. Im Beispiel von *struct_member* sorgt die tatsächliche VSL-Realisierung etwa dafür, daß alle Gleichheitszeichen einer Verbund-Darstellung vertikal ausgerichtet werden.

3 Entfalten von Datenstrukturen

Die eigentliche Stärke von DDD liegt jedoch nicht im Visualisieren individueller Daten, sondern im Visualisieren ihrer *Beziehungen*. Benutzer können ein Display nicht nur durch Eingabe des Ausdrucks erzeugen, sondern auch, indem sie eine *Operation* auf ein bestehendes Display anwenden, um ein neues Display zu erzeugen.

Operationen werden als Ausdrücke der Programmiersprache angegeben, wobei die Zeichenkette `()` durch den Ausdruck des bestehenden Displays ersetzt wird. Die Operation `().next` etwa, wiederholt angewandt, erzeugt aus `a` das Display `a.next`, daraus `a.next.next` und so weiter. Wird nun ein neues Display d' aus einem bestehenden Display d erzeugt, so stellt DDD die Beziehung durch eine *gerichtete Kante* von d nach d' dar.

Die wichtigste Operation ist das *Dereferenzieren* eines Zeigers (in C und C++: `*()`, in Pascal und Modula: `()^`), ausgelöst durch einen Doppelklick auf den Zeigerwert. Nehmen wir an, der Ausdruck `*tree` würde wie in Abbildung 3 auf der vorherigen Seite in DDD dargestellt. Dann kann der Benutzer durch Doppelklicken auf Zeigerwerte wie `_left` die Zeiger dereferenzieren. Im neuen Display kann er wieder die Zeiger dereferenzieren und so die komplette Datenstruktur entfalten. Am Ende steht der Baum aus Abbildung 5.

Wichtig bei diesem *Entfalten* von Datenstrukturen ist, daß der Programmierer selbst entscheidet, welchen Querverweisen er folgen möchte. Mit dem interaktiv-inkrementellen Ansatz entfällt die Notwendigkeit, eine Datenstruktur am Stück zu erkennen und alle ihre Elemente zu positionieren. DDD kennt lediglich vier einfache Regeln, anhand deren neue Displays positioniert werden.

Ausgangslage ist, daß der Anwender ein neues Display d' aus einem bestehenden Display d erzeugt – etwa, indem er d dereferenziert und d' erhält. Dann lassen sich vier Fälle unterscheiden, dargestellt in Abbildung 6:

- Hat d (der „Vater“) bereits einen Nachfolger d_1, so wird d' neben d_1 plaziert. Genauer gesagt, wird d' plaziert
  - *unterhalb* von d_1, wenn d und d_1 *nebeneinander* stehen (= ihre X-Koordinaten unterscheiden sich stärker als ihre Y-Koordinaten)
  - *rechts* von d_1, wenn d und d_1 *übereinander* stehen (= ihre Y-Koordinaten unterscheiden sich stärker als ihre X-Koordinaten).

  Dies ermöglicht das „Auffächern“ mehrerer Nachfolger.
- Hat d mehrere Nachfolger $d_1, \ldots, d_n$, wird d' neben dem letzten Nachfolger d_n in Verlängerung des Abstandes zum vorletzten Nachfolger d_{n-1} plaziert. Dies führt dazu, daß alle Nachfolger aneinander ausgerichtet sind.
- Hat d keinen Nachfolger, aber einen Vorgänger d_0, wird d' in Verlängerung des Abstands zwischen d und d_0 plaziert. Hiermit werden lineare Listen ausgerichtet.
- Hat d weder Vorgänger noch Nachfolger, wird d' neben d plaziert. Die Vorgabe für diese Ausrichtung ist „rechts von d“; der Benutzer kann dies durch Rotieren des Graphen ändern.

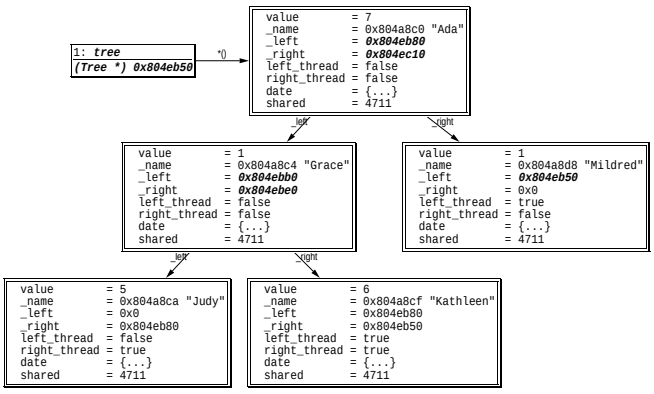


Abb. 5 Ein Baum in DDD. Durch fortgesetztes Dereferenzieren wird der komplette Baum entfaltet.

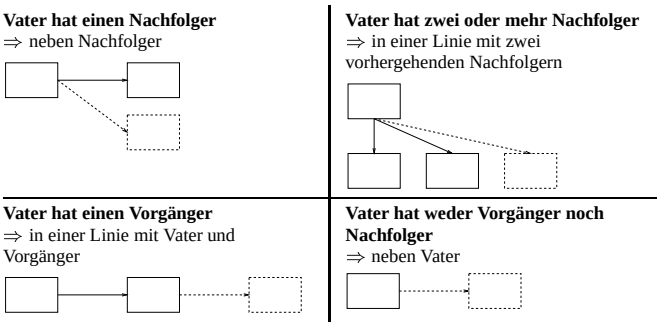


Abb. 6 Positionierungsregeln in DDD. Das neu zu positionierende Display ist gestrichelt.

Nach der ursprünglichen Vorgabe kann der Programmierer die Display-Position nach Belieben verändern, indem er die Displays wie Bildschirmfenster verschiebt; die Kanten folgen den Displays. Der Baum in Abbildung 5 ist mit einem solchen manuellen Eingriff entstanden: der linke Sohn wurde gemäß den Regeln zunächst rechts von der Wurzel positioniert. Nach dem Verschieben links unterhalb der Wurzel konnten die weiteren Knoten gemäß den Regeln automatisch positioniert werden.

DDD bietet auch die Möglichkeit, alle Displays des Graphen neu zu positionieren. Hierfür setzt DDD das Verfahren von Sugiyama und Misue [10] ein. Dabei geht die bisherige Ortsinformation verloren – jedes Display erhält einen neuen Platz. Daher muß sich der Anwender komplett neu orientieren, was ärgerlich ist. Unserer Erfahrung nach bevorzugen die Benutzer die manuell-inkrementelle Positionierung.

4 Visualisierung von Datenstrukturen

Das DDD-Anzeigeverfahren zieht beim Erzeugen eines neuen Displays d' eine Kante vom Ursprungs-Display d nach d'. Dies bedeutet aber, daß auf jedes neu erschaffene Datum maximal eine Kante zeigen kann – die Darstellung ist also auf Baumstrukturen beschränkt, wie etwa in Abbildung 7 auf der nächsten Seite.

Tatsächlich ist die Struktur in Abbildung 7 jedoch kein Baum. Wie der (textuelle) Vergleich der Zeigerwerte ergibt,

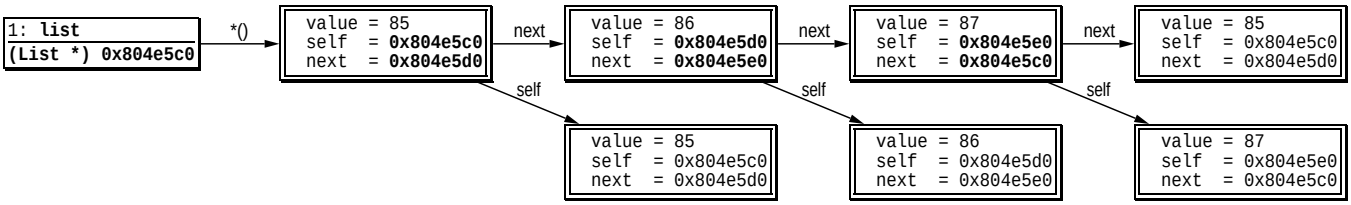


Abb. 7 Darstellung verketteter Listen ohne Aliaserkennung. `list→next→next→next` ist identisch mit `*list`.

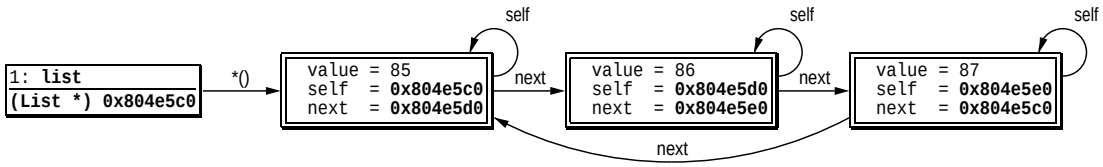


Abb. 8 Die Liste aus Abbildung 7 – mit Aliaserkennung. Die Aliaserkennung verdeutlicht den Zyklus.

gilt zum einen, daß die `self`-Zeiger jedes Elements auf das Element selbst zeigen (es gilt also $e\text{→}\texttt{self} = e$ für alle Elemente e); außerdem ist das letzte gezeigte Element der Liste, `list→next→next→next`, identisch mit dem ersten Element `*list`: Es handelt sich also um eine zyklische Liste. Solche Eigenschaften sind aus der Baum-Darstellung in Abbildung 7 nicht ersichtlich; auch kein anderer Debugger kann sie visualisieren.

DDD verfügt aber über ein besonderes Verfahren, diese Eigenschaften zu verdeutlichen – die sogenannte *Aliaserkennung*. Grundidee ist, die Displays, die an derselben Adresse im Speicher stehen (*Aliase*), zu *verschmelzen*, um so mehrere Verweise auf ein Datum zu ermöglichen. Ergebnis der Aliaserkennung ist eine *Graphdarstellung* mit beliebigen Querverweisen zwischen Displays – im Fall der Liste aus Abbildung 7 etwa die Darstellung in Abbildung 8. Das Bild zeigt, welche Zeiger auf welche Daten verweisen, ohne daß der Anwender erst deren Adressen textuell vergleichen müßte.

Formal funktioniert die Aliaserkennung wie folgt. Stellen zwei Displays d_1 und d_2 denselben Ausdruck dar, schreiben wir dies als $d_1 = d_2$. Haben zwei Displays d_1 und d_2 die gleiche Speicheradresse, schreiben wir dies als $d_1 \mathrel{\mathring{=}} d_2$. Gleiche Ausdrücke bedingen gleiche Speicheradressen ($d_1 = d_2 \implies d_1 \mathrel{\mathring{=}} d_2$); umgekehrt gilt dies jedoch nicht.

Ziel ist es nun, aus einer Menge D von Displays eine Untermenge $D' \subseteq D$ zu bestimmen, so daß in D' für jede Speicheradresse nur ein Display in D' enthalten ist. In D' müssen zwei Displays d_1 und d_2 mit unterschiedlichen Ausdrücken also unterschiedliche Speicheradressen haben ($\forall d_1, d_2 \in D' \cdot d_1 = d_2 \vee d_1 \mathrel{\mathring{\neq}} d_2$). DDD bestimmt D' durch einfaches Einteilen der Displays in Äquivalenzklassen gemäß $\mathring{=}$ und entfernt („unterdrückt“) alle Displays aus der Darstellung, die in D, jedoch nicht in D' sind.

Kanten, die zu unterdrückten Displays führten, zieht DDD nun zu den verbliebenen äquivalenten Displays, und zwar in Form eines Bogens, dessen Eintritts- und Austrittswinkel so gewählt wird, daß der Bogen einer direkten Verbindung möglichst nahe kommt, jedoch kein bestehendes Display schneidet. Dies sorgt für die Darstellung in den Abbildung 8.

Als weiteres Beispiel sei der Baum von Abbildung 5 auf der vorherigen Seite genannt, der tatsächlich ein *gefädelter Baum* ist. In Abbildung 9 sorgt die Aliaserkennung für die korrekte Darstellung der Fädelungen.

Ein Detail fehlt noch: Wenn DDD die Menge D' aus D bestimmt, wie wählt DDD im Fall $d \mathrel{\mathring{=}} d'$ das Display aus, das weiterhin angezeigt werden soll? Die Regel ist ganz einfach: das *ältere* der beiden (dasjenige, das *zuerst* angezeigt wurde) bleibt erhalten. Diese pragmatische Entscheidung sorgt nicht nur dafür, daß der *einfachere* Ausdruck erhalten bleibt (Benutzer „klicken“ sich gewöhnlich von *list* zu *list→next* zu *list→next→next* und so fort). Sie sorgt auch für Beständigkeit in der Darstellung: ist ein Display d bereits vorhanden, und soll ein neues Display $d' \mathrel{\mathring{=}} d$ gezeigt werden, wäre es unsinnig, d zu löschen und stattdessen d' darzustellen.

5 Weitere Darstellungsmöglichkeiten

Neben dem einfachen Dereferenzieren von Zeigern unterstützt DDD beliebige andere Verweis-Operationen. Erzeugt der Benutzer etwa einen Tabelleneintrag $t[i]$ aus einem Index i, so wird eine Kante von i zum neu erzeugen Display $t[i]$ gezogen. Die Operationen werden in einem Menü gespeichert und dienen so zur Erkundung und Darstellung beliebiger Verweise. Die Alias-Erkennung funktioniert weiterhin:

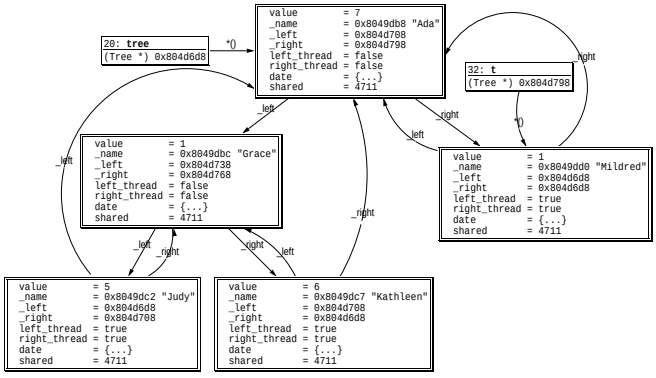


Abb. 9 Der Baum aus Abbildung 5 – mit Fädelung

Existiert bereits ein Alias $d \triangleq t[i]$, wird die Kante von i zu d gezogen; das neue Display $t[i]$ wird unterdrückt.

Das inkrementelle Entfalten bleibt jedoch mühsam, wenn große Strukturen Stück für Stück erkundet werden müssen. Für künftige DDD-Versionen ist deshalb ein *automatisches Entfalten* von Datenstrukturen geplant: Nach dem Positionieren der ersten zwei oder mehr Displays werden die Operationen solange auf die Nachfolger angewandt, bis ein Fixpunkt erreicht ist. Auf diese Weise lassen sich beliebige Datenstrukturen durch einfaches automatisches Wiederholen der Zugriffsmechanismen entfalten.

Die konkrete Darstellung der Daten und Beziehungen in DDD hat den Nachteil, daß wenig Raum für abstrakte (Text-) Daten bleibt. Neuere DDD-Versionen bieten deshalb die Möglichkeit, Displays in *Cluster* zusammenzufassen. In einem Cluster werden Variablen und deren Werte in Form einer Liste dargestellt – ähnlich wie die Verbundelemente in Abbildung 3. Werden *alle* Displays zu einem Cluster zusammengefaßt, erhält man eine platzsparende Textdarstellung wie auch in herkömmlichen Debuggern, verzichtet aber gleichzeitig auf die Visualisierung von Querbeziehungen.

6 Dynamische Visualisierung

Bei jedem Halt des Programms bringt DDD alle angezeigten Werte auf den neuesten Stand. Für die dargestellten Displays bedeutet dies:

- Hat sich ein Einzel-Datum geändert, wird dessen Schachtel neu berechnet; analog werden die Schachteln seiner Vaterknoten im abstrakten Syntaxbaum neu bestimmt. Veränderte Werte werden in der Darstellung hervorgehoben.
- Es ist möglich, daß der Wert eines Displays nicht mehr bestimmt werden kann, weil die Berechnung des Ausdrucks einen Fehler verursacht. (Dies ist etwa bei einem Display `*tree` der Fall, wenn `tree` auf NULL gesetzt wurde.) In diesem Fall wird das Display als *abgeschaltet* (*disabled*) gekennzeichnet.
- Nach dem Aktualisieren aller Displays wird die Alias-Erkennung erneut ausgeführt; neu unterdrückte Displays werden aus der Darstellung entfernt, andere können nun als nicht unterdrückte Displays wieder erscheinen.

Diese Eigenschaften sorgen dafür, daß die dargestellten Datenstrukturen auch nach größeren Änderungen stets auf dem neuesten Stand bleiben. Abbildung 10 etwa zeigt, wie ein Programm Elemente aus einer verketteten Liste löscht. Ein Haltepunkt stoppt das Programm nach jedem Löschen, und DDD stellt den neuen Wert der Liste dar. Zum Schluß bleibt nur der NULL-Zeiger `list` übrig, der nicht mehr dereferenziert werden kann.

7 Programme verstehen und präsentieren

Wie am Beispiel aus Abbildung 10 deutlich wird, läßt sich DDD nicht nur zur Fehlersuche einsetzen, sondern auch zum

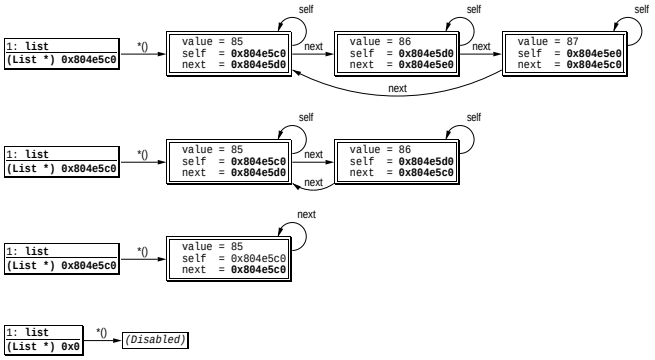


Abb. 10 Elemente aus einer verketteten Liste löschen. Bei jedem Programmhalt wird die Darstellung aktualisiert.

Präsentieren von Algorithmen und Datenstrukturen, etwa in der Lehre. Das beste Einsatzgebiet hat DDD jedoch im *Programmverstehen*, wo diese beiden Bereiche zusammenfallen: DDD ist prädestiniert für das interaktive Untersuchen von Programmen und Datenstrukturen durch den Lernenden.

Speziell für den Einsatz in der Lehre und der Präsentation bietet DDD einige Besonderheiten, die hier näher erläutert werden sollen. Zum einen ermöglicht es DDD, den kompletten DDD-Zustand als *Sitzung* (*session*) abzuspeichern und später wieder neu zu laden. Hierbei wird der komplette Speicherinhalt des untersuchten Programms als Speicherauszug (*core dump*) abgelegt, der dann beim Wiederherstellen *post mortem* untersucht werden kann. Auch alle Displays, Haltepunkte und DDD-Einstellungen werden gesichert. Damit ist es möglich, ausgewählte Programmzustände für spätere Präsentationen oder Untersuchungen zu konservieren.

Zum anderen kennt DDD die Möglichkeit, früher erreichte Programmzustände erneut darzustellen. Durch Klick auf *Undo* wird der zuvor von DDD beobachtete Programmzustand erneut gezeigt; analog zeigt *Redo* spätere Programmzustände an. DDD speichert genau die Aspekte, die DDD während des Laufs bekannt werden — etwa die per Display dargestellten Variablen, der Programmzähler und der aktuelle Funktionsstapel. Nach einem *Undo* können also auch nur diese Werte erneut abgefragt werden; um mehr zu erfahren, muß das Programm erneut gestartet werden.

Will man einen Algorithmus präsentieren, so kann man durch geschicktes Setzen von Haltepunkten *Schlüsselstellen* des Programmablaufs auswählen – etwa den Zustand nach dem Löschen eines Elements wie in Abbildung 10. Anschließend wird das Programm ausgeführt, wobei beim Programmhalt ausgewählte Variablen als *Display* dargestellt werden. Nach Programmende steuert man die Schlüsselstellen per *Undo*/*Redo* an, wobei die zuvor angezeigten Variablen wieder mit ihren früheren Werten erscheinen.

Mit Schlüsselstellen und *Undo*/*Redo* lassen sich passable Präsentationen gestalten. Leider wird die *Undo*/*Redo*-Historie beim Speichern einer Sitzung nicht mit abgelegt; Präsentationen, die auf *Undo*/*Redo* setzen, können also nicht konserviert werden. Innerhalb einer Präsentation ist es jedoch praktisch, durch Rückwärts- und Vorwärtsschreiten wiederholt den Effekt einer Anweisung zu demonstrieren.

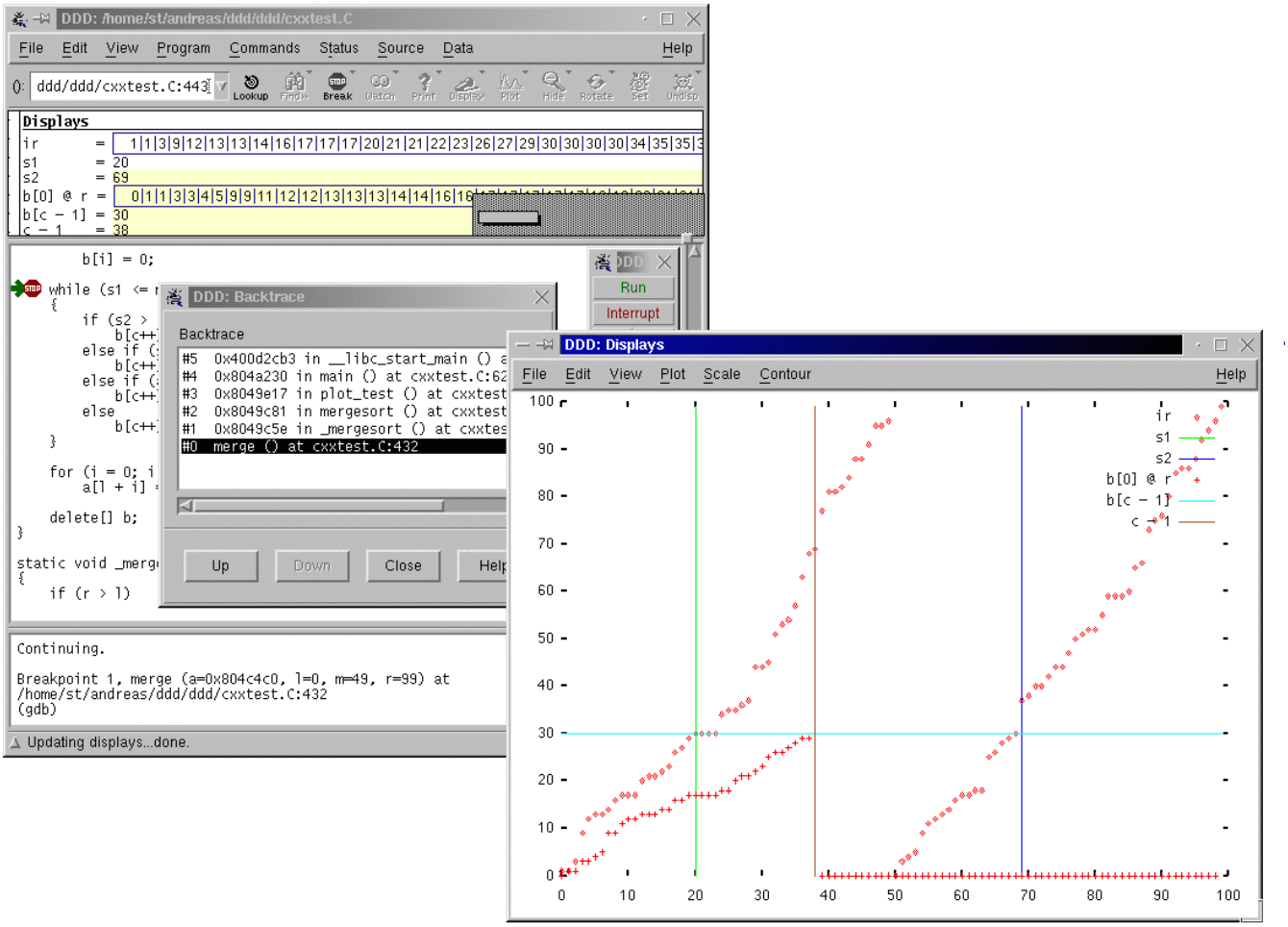


Abb. 11 Diagramme in DDD. Feldelemente erscheinen als Werte über dem Index, einfache Werte als Linien.

8 Visualisierung numerischer Werte

Neben der Darstellung von Datenstrukturen erlaubt es DDD, numerische Werte in Form von *Diagrammen* (engl. *plots*) darzustellen. Hierfür greift DDD auf die Dienste des externen Werkzeugs GNUPLOT zurück. Das Grundverfahren ist ganz einfach:

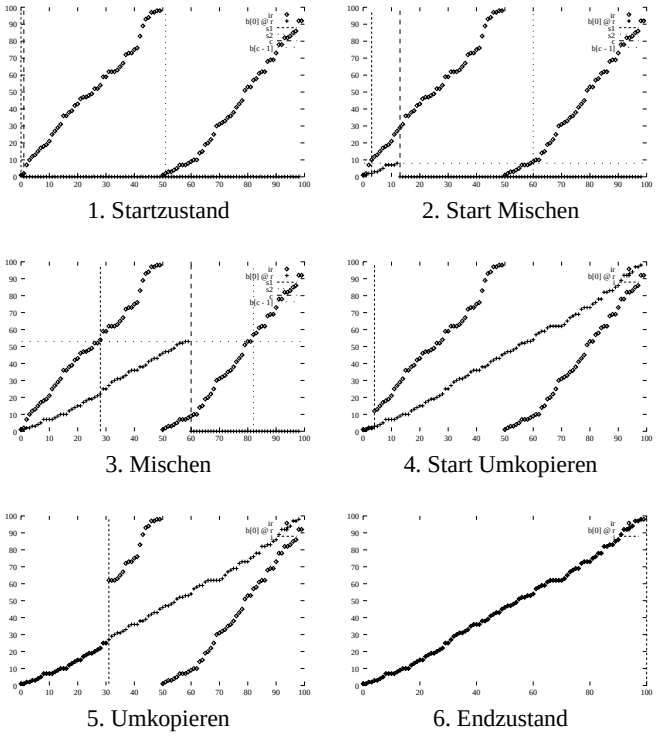
1. Der Benutzer wählt ein Datenfeld aus, das in Form eines Diagramms dargestellt werden soll.
2. DDD erfragt vom Kommandozeilen-Debugger die Werte und sendet sie an GNUPLOT.
3. DDD führt die von GNUPLOT generierten Zeichenanweisungen in einem eigenen Fenster aus.

DDD kann folgende Datentypen als Diagramme darstellen:

– *Eindimensionale Felder* werden zweidimensional über den Index gezeichnet.
– *Zweidimensionale Felder* werden dreidimensional über beide Indizes gezeichnet.
– *Einfache numerische Werte* werden als Linien gezeichnet. Wird der Wert zusammen mit einem Feld gezeichnet und liegt der Wert im Indexbereich des Feldes, so zeichnet DDD den Wert als vertikale x-Konstante, ansonsten als horizontale y-Konstante. Der Benutzer kann die Ausrichtung selbst ändern.
– *Zusammengesetzte Werte* (Cluster, Verbünde, usw.) werden in einem Diagramm zusammengefaßt. DDD berücksichtigt hierbei alle auftretenden numerischen Werte – Zeichenketten oder Zeiger werden übergangen.

Der letzte Punkt ist besonders wichtig, wenn man mehrere Werte zueinander in Beziehung setzen will. In Abbildung 11 etwa sehen wir einen Ausschnitt aus der *merge*-Funktion des *mergesort*-Algorithmus: zwei bereits sortierte Teilfolgen a_1 und a_2 werden in eine neue Teilfolge b sortiert zusammengemischt. Dies sind die dargestellten Werte:

– Das Feld *ir* faßt die Folgen a_1 und a_2 zusammen. a_1 umfaßt die Elemente $ir[0..49]$ und a_2 die Elemente $ir[50..99]$; für jeden gezeichneten Punkt (x, y) gilt $y = ir[x]$.
– In das Feld b wird hineingemischt. Die ersten 38 Elemente sind bereits definiert.
– Die für das Lesen aus a_1 und a_2 benutzten Indizes s_1 und s_2 sowie der für das Schreiben in b benutzte Index $c - 1$ sind als vertikale Linien dargestellt.
– $b[c-1]$, der maximale Wert in b, ist als horizontale Linie dargestellt. Der Schnittpunkt von $c-1$ und $b[c-1]$ ist der

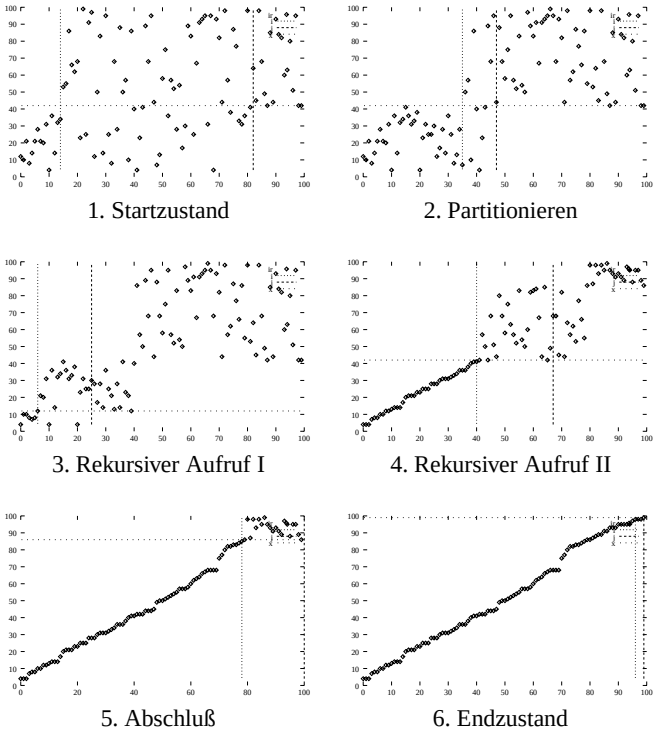


1. Startzustand 2. Start Mischen

3. Mischen 4. Start Umkopieren

5. Umkopieren 6. Endzustand

Abb. 12 Animation eines *mergesort*-Algorithmus in DDD. Zwei bereits sortierte Teilfolgen werden in ein Hilfsfeld zusammengemischt und in das Ursprungsfeld zurückkopiert. Die vertikalen Linien zeigen die Indizes der aktuellen Elemente.



1. Startzustand 2. Partitionieren

3. Rekursiver Aufruf I 4. Rekursiver Aufruf II

5. Abschluß 6. Endzustand

Abb. 13 Animation eines *quicksort*-Algorithmus in DDD. Alle Elemente, die kleiner sind als das Pivot-Element (horizontale Linie), gelangen in die linke Teilfolge, alle größeren Elemente in die rechte Teilfolge. Durch rekursiven Aufruf wird die Liste sortiert.

letzte in b geschriebene Wert; der Schnittpunkt von $c-1$ und s_1 bzw. s_2 kennzeichnet den letzten aus a_1 bzw. a_2 gelesenen Wert.

DDD stellt Menüs zur Verfügung, mit denen die wichtigsten Aspekte des Diagramms verändert werden können; auch Drucken und Exportieren von Diagrammen ist möglich.

Arbeitet ein Algorithmus auf numerischen Werten, empfiehlt es sich, seine Daten als Diagramm zu animieren. Abbildung 12 etwa zeigt weitere Ausschnitte aus dem Ablauf des *mergesort*-Algorithmus aus Abbildung 11. Deutlich ist zu erkennen, wie die Werte aus den beiden Hälften zusammengemischt und schließlich umkopiert werden.

Ein zweites Beispiel zeigt Abbildung 13: Hier animiert DDD einen *quicksort*-Algorithmus. Dargestellt ist der *Partitionierungsvorgang* – alle Werte, die kleiner sind als das Pivot-Element (horizontale Linie), werden im Feld nach links bewegt; alle Werte, die größer sind, werden nach rechts bewegt. Die vertikalen Linien wandern von außen nach innen und stellen den Fortschritt des Verfahrens dar; treffen sich die Linien, ist die Partitionierung auf dieser Ebene abgeschlossen. In den späteren Schritten sieht man, wie der rekursive Aufruf der Partitionierung das gesamte Feld sortiert.

Noch wirkungsvoller werden diese Darstellungen, wenn man sie *vollautomatisch* ablaufen läßt. Hierzu setzt man einen Haltepunkt, der mit einem *continue*-Kommando gekoppelt ist. Beim Erreichen des Haltepunkts bringt DDD die Darstellung auf den neuesten Stand und führt das *continue*-Kommando aus, wodurch der Programmlauf fortgesetzt wird. Mit *bedingten Haltepunkten* kann die Ausführung an ausgewählten Stellen unter definierten Bedingungen angehalten werden – etwa im erstmaligen rekursiven Aufruf oder bei Unterschreiten einer gewissen Schranke für den Index.

Als besondere Eigenschaft erlaubt es DDD schließlich, die *Historie* eines Wertes als Diagramm darzustellen – also die Kette der früheren Werte während der Programmausführung. Beim Anzeigen der Historie wird nun die Folge der Variablenwerte über die Anzahl der Programmhalte gezeichnet. Hiermit lassen sich ungewöhnliche Wertentwicklungen „auf einen Blick“ erkennen. Dies kann ebenfalls als Animation eingesetzt werden: mit jedem Halt verlängert sich das Diagramm um einen weiteren Wert.

9 Alternative Darstellungen

Natürlich bleibt DDD ein Debugger und somit ein Universalwerkzeug mit großem Freiheitsgrad – am besten geeignet für Situationen, in denen man sich nicht von vorneherein in den zu untersuchenden Aspekten beschränken möchte. Die Hauptschwäche von DDD tritt dann zutage, wenn man sich von den im Programm vorhandenen Datenstrukturen lösen will und auf einer höheren Abstraktionsebene arbeiten und visualisieren will.

Hier ein Beispiel: Ein Programm arbeitet auf einem Graphen. Der Graph wird intern als Adjazenz-Matrix dargestellt. DDD kann diese Adjazenz-Matrix tatsächlich visualisieren, aber eben nur als Tabelle oder Diagramm, keineswegs jedoch

als Graph aus dem Lehrbuch. Eine Menge, realisiert durch einen Bitvektor, läßt sich ebenfalls nicht als Menge darstellen, sondern eben nur als Folge von Nullen und Einsen.

Wir arbeiten derzeit daran, DDD so zu erweitern, daß neben den vordefinierten Darstellungen ebenfalls freidefinierbare Darstellungen möglich werden. Grundidee ist, aus der konkreten Realisierung (die dem Debugger zur Verfügung steht) die abstrakte Darstellung zu extrahieren. Hierzu müssen die Darstellungsfunktionen um Möglichkeiten erweitert werden, selbständig die darzustellenden Datenstrukturen zu durchwandern und die abstrakte Information herauszuziehen.

Unser aktueller DDD-Prototyp kennt zu diesem Zweck drei neue interne Funktionen, die zur Definition von Display-Funktionen benutzt werden können:

- *display*() liefert den vollständigen Namen des darzustellenden Ausdrucks.
- *eval*(e) erfragt den Wert des Ausdrucks e über GDB und liefert den Wert als Text zurück.
- *parse*(v) nimmt einen Wert v und wandelt ihn gemäß den aktuellen Regeln in eine strukturierte Schachtel.

Typischerweise werden diese drei Funktionen gemeinsam angewandt: der ursprüngliche darzustellende Ausdruck wird mit *display* bestimmt und (textuell) um geeignete Operationen erweitert. Der so entstehende neue Ausdruck e wird mit *eval*(e) ausgewertet; der Wert v wiederum wird mit *parse*(v) in eine Schachtel umgewandelt.

Hier ein konkretes Beispiel: Nehmen wir an, wir möchten eine Liste komplett in einem einzelnen Display darstellen – statt des Wertes des *next*-Zeigers möchten wir direkt den Nachfolger einsetzen. Hierzu ergänzen wir die Display-Funktion *struct_member* aus Abschnitt 2 um folgenden Spezialfall:

$$\textit{struct\_member}(\textit{name}, \textit{value}) = \begin{cases} s_1 & \text{falls } \textit{name} = \texttt{"next"} \\ s_2 & \text{sonst} \end{cases}$$

mit

$$s_1 = \textit{parse}\big(\textit{eval}(\texttt{"*("} \,\&\, \textit{display}() \,\&\, \texttt{")"})\big)$$
$$s_2 = \textit{name} \,\&\, \texttt{" = "} \,\&\, \textit{value}$$

Was geschieht hier? Die Alternative s_1 von *struct_member* wird nur angewandt, wenn das Element *next* heißt (also die dem Elementnamen zugewiesene Schachtel den Wert `"next"` hat); für alle anderen Elemente kommt die Standard-Definition s_2 zum Tragen. s_1 bestimmt (mit *display*()) den darzustellenden Ausdruck e, bildet den dereferenzierenden Ausdruck $*(e)$, wertet ihn aus (mit *eval*) und wandelt ihn in eine Schachtel um (mit *parse*).

Dies bewirkt, daß bei der Darstellung eines `next`-Zeigers nicht dessen Wert, sondern das vollständige dahinterstehende Element an dessen Stelle gesetzt wird – die komplette Liste erscheint zusammengefaßt in einem Display (Abbildung 14). Mit weiteren Darstellungsregeln könnten die Listenelemente nebeneinander angeordnet werden; auch eine

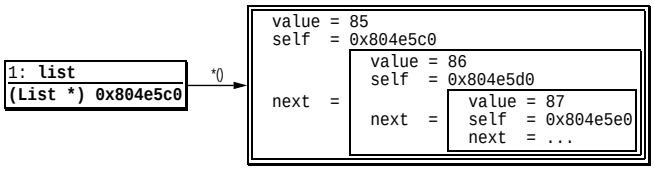


Abb. 14 Eine komplette Liste in einem DDD-Display. Durch selbstdefinierte Darstellungsregeln erscheinen die Elemente einer verketteten Liste verschachtelt.

graphische Darstellung von Listenwerten wäre denkbar. Die alternativen Darstellungen können als Themen (siehe Abschnitt 2) abgelegt werden; damit kann der Benutzer zwischen konkreter und abstrakter Darstellung wählen.

Der Haken an der Sache ist, daß für jede spezifische Anwendung eigene Darstellungsfunktionen programmiert werden müssen. Und diese sind trickreich. So scheitert die obige Definition von *struct_member* etwa an zyklischen Listen. In der konkreten Realisierung muß man sich die bereits dargestellten Elemente merken und stets prüfen, ob ein neues Element bereits dargestellt wurde. Für komplexere graphische Darstellungen wie etwa Graphen müssen zunächst geeignete Bibliotheken mit Hilfsfunktionen erstellt werden.

Letzten Endes sind auch die Display-Funktionen berechnungsuniversell. Die Frage ist jedoch, welche Abstraktionen nötig sind, um den Aufwand ihrer Erstellung zu minimieren – und unter welchen Umständen es sich lohnt, aus einer konkreten Realisierung die abstrakte Darstellung zu extrahieren.

10 Verwandte Arbeiten

Neben DDD gibt es eine Reihe von Projekten, die ebenfalls Programmuntersuchungen mit Visualisierung verbinden. Wir beginnen zunächst mit Werkzeugen, die primär der *Präsentation* dienen und für den Einsatz in der Lehre entwickelt wurden.

VCC [1] visualisiert Programme, indem es beim Übersetzen automatisch Aufrufe für eine Animations-Bibliothek in den Programmcode hinein übersetzt. Im Gegensatz zu DDD ist somit ein eigener Übersetzungsschritt erforderlich; auch ist VCC auf die Programmiersprache C beschränkt. VCC bietet Sichten auf den Code, die aktive Funktion, den Baum der Funktionsaufrufe sowie Datenstrukturen.

ZStep95 [7] ist eine Visualisierungsumgebung für Common Lisp. ZStep95 fügt wie VCC dem Programm spezielle Animations-Aufrufe hinzu und speichert wie DDD die Historie von Daten während eines Programmlaufs, die dann mittels eines „Videorekorders“ vor- und zurückgespult werden können. Im Gegensatz zu DDD (der nur dargestellte Daten speichert), merkt sich ZStep95 speicheraufwendig den kompletten Programmzustand.

Lens [9] ist wie DDD ein graphisches Debugger-Frontend, in diesem Fall für den DBX-Debugger. Lens umfaßt ein Entwicklungswerkzeug, in dem interaktiv Animationen

zum Code erstellt werden können. Im Gegensatz zu DDD, wo die Animation über wiederholte Darstellung der Daten während des Programmlaufs erfolgt, müssen in Lens spezielle *Animationsanweisungen* in das Programm eingefügt werden, die dann während des Ablaufs für die Animation sorgen. Damit ermöglicht Lens vielfältige Eingriffe in die Animation – etwa Farben, Strukturen, usw. – die bei DDD nur als Themen realisiert werden können.

In den folgenden Forschungsprototypen wurden herkömmliche Debugger gezielt um Visualisierungskomponenten erweitert.

DEET [6] ist ebenfalls eine graphische Erweiterung für Kommandozeilen-Debugger. In DEET liegt der Schwerpunkt auf *Einfachheit:* Für die graphische Datendarstellung werden externe Positionierer und Zeichenprogramme benutzt. Da DEET keine Aliases erkennt, ist die Darstellung auf Baumstrukturen beschränkt (die aber im Gegensatz zu DDD automatisch entfaltet werden); Displays können in DEET nicht interaktiv positioniert werden.

DUEL [4] ist eine spezielle Programmiersprache für Debugger, mit der Daten auf einer höheren Abstraktionsstufe analysiert werden können. So bietet DUEL Quantoren, Filter und Mengenoperationen. Die Erweiterung XDUEL visualisiert die aus DUEL gewonnenen Daten. Der Hauptvorteil von DUEL ist gleichzeitig sein Nachteil: Die speziellen DUEL-Operatoren und -Befehle lösen sich von der Programmiersprache und erschweren so Anfängern die Bedienung.

ISVL [3] ist eine Arbeitsumgebung zur verteilten kollaborativen Fehlersuche. Die Visualisierungskomponente visualisiert eine Ausführungshistorie (ähnlich ZStep95) auf einem Web-Server, über den eine Gruppe von Programmierern ihre Hypothesen über die Fehlerursache austauschen und diskutieren können. ISVL ist auf die Programmiersprache PROLOG beschränkt.

Wir schließen mit dem einzigen uns bekannten kommerziell erhältlichen Debugger mit integrierter Datenvisualisierung.[2]

PRISM [8] ist Bestandteil des *High Performance Computing*-Entwicklungspakets von Sun. Die Stärke des PRISM-Debuggers liegt in der Visualisierung *numerischer Werte*, wie sie insbesondere im wissenschaftlichen Rechnen auftreten. Neben gewöhnlichen Diagrammen bietet PRISM Oberflächenvisualisierungen, farblich codierte Wertebereiche, Vektordarstellungen – kurz, alles, was moderne wissenschaftliche Visualisierung zu bieten hat. Einzige Schwäche: Mehrere Datenquellen lassen sich nicht kombinieren. In der Visualisierung von *Datenstrukturen* kann der PRISM-Benutzer diese automatisch entfalten und auch den Maßstab frei wählen (Zoom). Die Darstellung ist wie in DEET auf Baumstrukturen beschränkt.

[2] Kundenspezifisch eingerichtete oder vorübersetzte DDD-Versionen sind ebenfalls kommerziell erhältlich.

11 Zusammenfassung und Ausblick

DDD ist ein *Universalwerkzeug* zur dynamischen *Softwarevisualisierung*. DDD kann verzeigerte Strukturen (wie Listen oder Bäume) als Graphen darstellen, aber auch numerische Felder in zwei- oder dreidimensionalen Diagrammen darstellen und im Programmlauf animieren. Herausragend in DDD sind der schnelle und flexible Zugang zu visualisierten Daten (man benötigt weder aufwendige Neuübersetzungen noch komplexe Befehlsfolgen noch langwierige Interaktionen) sowie die Darstellung von komplexen Datenstrukturen im *Lehrbuch-Stil*, einschließlich aller Querbeziehungen. Man kann DDD damit nicht nur für die klassische Fehlersuche, sondern auch für das Verstehen von Algorithmen und Datenstrukturen einsetzen.

Die Zukunft des visuellen Debuggens liegt in der Kunst, Datenaspekte so darzustellen, daß sie vom Menschen so einfach wie möglich erfaßt werden können. Hierzu bedarf es individuell anpaßbarer Abstraktions- und Darstellungstechniken. Was *numerische Werte* angeht, können wir viel von der wissenschaftlichen Visualisierung lernen. Der PRISM-Debugger zeigt mit seinen fortgeschrittenen Visualisierungstechniken, wohin die Reise gehen kann. Bei der Visualisierung klassischer *Datenstrukturen* hat DDD Pionierarbeit geleistet, was die *Beziehungen* zwischen Daten angeht. Aus Benutzersicht wünschenswert wären jedoch selbstdefinierte Darstellungen – insbesondere Darstellungen von Mengen, Graphen oder Abbildungen. Auch die automatische Positionierung in DDD könnte verbessert werden. Vielversprechend sind constraint-basierte Positionierungsverfahren, wie sie zur Animation von Algorithmen eingesetzt werden [2, 5].

Daß jegliche Visualisierungen letztendlich ihre Grenze in der kognitiven Aufnahmefähigkeit des Menschen finden, ist eine Binsenweisheit. Wer mit DDD oder anderen Werkzeugen große Datenstrukturen visualisiert und hofft, so Fehler zu finden, benutzt womöglich das falsche Werkzeug. Schließlich will man auch nicht die Inhalte einer Datenbank komplett visualisieren, sondern nur die Ergebnisse ausgeklügelter *Anfragen* – automatisierte Anfragen, wie sie etwa DUEL zur Verfügung stellt, die man aber auch als Zusicherungsfunktionen in einem Programm vorsehen kann.

Eine Darstellung im Lehrbuch-Stil ist gut geeignet für Lehrbücher – eben *Beispiele*, die eine bestimmte Größe nicht überschreiten. Und für das Verstehen eines Algorithmus sollten überschaubare Datenmengen ohnehin genügen. Ist das Problem erst einmal auf Beispielgröße reduziert, bringt Visualisierung großen Gewinn – im Programmverstehen wie in der Präsentation, mit DDD und anderen Werkzeugen.

Danksagung. Holger Cleve, Jens Krinke, Kerstin Reese und Torsten Robschink gaben wertvolle Kommentare zu diesem Beitrag. Dorothea Krabiell (geb. Lütkehaus) ist Co-Autorin der ersten DDD-Fassung.

DDD, Dokumentation und Online-Verweise auf verwandte Projekte finden Sie unter

`http://www.gnu.org/software/ddd/` .

Literatur

1. BAEZA-YATES, R., G. QUEZADA und G. VELMADRE: *Visual Debugging and Automatic Animation of C Programs*. In: EADES, P. und K. ZHANG (Hrsg.): *Software Visualisation*. World Scientific Press, 1996.
2. BROWN, M. H.: *Algorithm Animation*. ACM Distinguished Dissertation, MIT Press, Cambridge, MA, 1988.
3. DOMINGUE, J. und P. MULHOLLAND: *Fostering Debugging Communities on the Web*. Communications of the ACM, 40(4):65–71, Apr. 1997.
4. GOLAN, M. und D. R. HANSON: *DUEL—a very high-level debugging language*. In: *Proceedings of the Winter USENIX Technical Conference*, S. 107–117, San Diego, California, USA, Jan. 1993.
5. GRAF, W. H.: *LayLab: A constraint-based layout manager for multimedia presentations*. In: CATARCI, T., G. SALVENDY und S. LEVIALDI (Hrsg.): *Human-Computer Interaction: Software and Hardware Interfaces*, Bd. 36 d. Reihe *World Scientific Series in Computer Science*, S. 365–385. World Scientific Press, Singapur, 1992.
6. HANSON, D. R. und J. L. KORN: *A Simple and Extensible Graphical Debugger*. In: *Proceedings of the USENIX 1997 Annual Technical Conference*, Bd. 183-174, Anaheim, California, USA, Jan. 1997.
7. LIEBERMANN, H. und C. FRY: *ZStep95: A Reversible, Animated Source Code Stepper*. In: STASKO, J., J. DOMINGUE, B. PRICE und M. BROWN (Hrsg.): *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1997.
8. SUN MICROSYSTEMS INC.: *Prism 6.1 User's Guide*. Sun Microsystems Inc., Palo Alto, California, USA, 2000. Teil der *Sun HPC 3.1 AnswerBook Collection;* online verfügbar unter http://docs.sun.com/.
9. STASKO, J. T. und S. MUKHERJEA: *Toward Visual Debugging: Integrating Algorithm Animation Capabilities Within a Source-Level Debugger*. ACM Transactions on Computer-Human Interaction, 1(3):215–244, Sep. 1994.
10. SUGIYAMA, K. und K. MISUE: *Visualization of Structural Information: Automatic Drawing of Compound Digraphs*. IEEE Transactions on Systems, Man, and Cybernetics, SMC-21(4):876–892, Juli 1991.

Andreas Zeller ist wissenschaftlicher Assistent am Lehrstuhl für Softwaresysteme der Universität Passau. Dort erforscht er neue Synergien aus Programmanalyse, Software-Test, Konfigurationsmanagement und Software-Visualisierung. Sein Buch *Programmierwerkzeuge* erschien Mai 2000 im dpunkt-Verlag.